Pratik Spring Core

Özcan Acar

PRATİK SPRING CORE

Pratik Spring Core

Yazılımcılar için Spring çatısını kullanma ve yazılım geliştirme rehberi

Özcan Acar

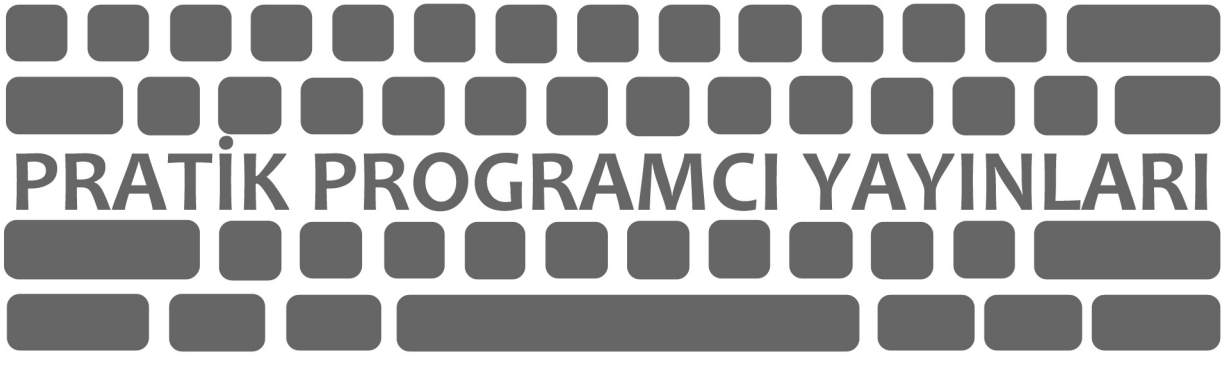
PRATİK PROGRAMCI YAYINLARI

Pratik Programcı 1

PRATIK SPRING CORE. Copyright © 2013 Pratik Programcı Yayınları.

Tüm telif ve yayın hakları Pratik Programcı Yayınları'na aittir. Telif hakkı sahibinin yazılı izni olmadan kısmen ya da tamemen alınyı yapılamaz,kopya edilemez, çoğaltılamaz, dağıtılamaz ve yayınlanamaz.

Yazar: Özcan Acar
Yayınevi: Pratik Programcı Yayınları
İlk sürüm: Ocak 2014
Kapak tasarımı: Ahmet Yıldırım
Düzelti: Nergis Şahin
Satış: http://www.pratikprogramci.com



pratikprogramci.com

Asya'daki (kızım) Vatan'ım (eşim), Türkiye'm için ...

Bilgi Paylaşım İle Çoğalır

Lütfen bu kitabın ve ihtiva ettiği bilginin yaygınlaşması için kitabı tanıdıklarınız için paylaşınız.

Pratik Programcı Yayınları
http://www.pratikprogramci.com

Bölüm Başlıkları

Kitap 18 bölümden oluşmaktadır. Ana bölüm başlıkları şunlardır:

- 1.Bölüm - Spring Nedir?
- 2.Bölüm - Spring İle Tanışalım
- 3.Bölüm - Spring İle Nesne Yaşam Döngüsü Yönetimi
- 4.Bölüm - Konfigürasyon Yönetimi
- 5.Bölüm - Spring İle Veri Tabanı İşlemleri
- 6.Bölüm - Spring İle Transaksiyon Yönetimi
- 7.Bölüm - Spring İle Hibernate Kullanımı
- 8.Bölüm - Spring İle JPA Kullanımı
- 9.Bölüm - Spring İle Aspect Oriented Programming
- 10.Bölüm - Spring MVC
- 11.Bölüm - Spring Security
- 12.Bölüm - Spring REST
- 13.Bölüm - Spring Remoting
- 14.Bölüm - Spring Web Service
- 15.Bölüm - Spring Testing
- 16.Bölüm - Spring JMX
- 17.Bölüm - Spring Task ve Scheduling
- 18.Bölüm - Spring E-Mail

İçindekiler

Bilgi Paylaşım İle Çoğalır 12
Bölüm Başlıkları 16
Önsöz 26
Yazar Hakkında 26
İlk E-Kitap 27
Neden Pratik Spring? 27
Spring Sürümü 28
Kitabın İçeriği Nedir? 28
Kitabın İçeriği Ne Değildir? 32
Kitap Kim İçin Yazıldı? 32
Kitap Nasıl Okunmalı? 32
Yazar İle İletişim 32
PratikProgramci.com 33
Kitapta Yer Alan Kod Örnekleri 33
1. Bölüm 35
Spring'e Giriş 35
Spring Filozofisi 37
- Dependency Injection 38
- Hollywood Prensibi 39

Spring Modülleri 40
Spring Modülleri İle Neler Yapabiliriz? 41
- Çekirdek Sunucu (Core Container) Modülü 42
- Spring AOP Modülü 42
- Veri Erişimi Modülü 42
- Spring MVC Modülü 43
- Spring Remoting Modülü 43
- Spring Test Modülü 44

Spring Uygulama Portföyü 44
Spring 3 İle Gelen Yenilikler 45
- Spring 3.0 45
- Spring 3.1 46
- Spring 3.2 47

Spring'in Uygulama Geliştirmedeki Rolü 47
Spring Yazılım Geliştirme Ortamı 48
Spring Jar Dosyalarını Nasıl Edinebilirim? 49
Spring Hello World 50
1. Bölüm Soruları 53
2. Bölüm 55
Spring İle Tanışalım 55
Bir Program Nasıl Oluşur? 56
Araç Kiralama Servisi 56
Program Alan Modeli (Domain Model) 57
Dizge Diyagramı (Sequence Diagram) 59
İlk Çözüm 60
Bağımlılıkların Enjekte Edilmesi (Dependency Injection) 62
Spring İle Bağımlılıkların Enjekte Edilmesi 65
Idref Kullanımı 71
Spring Sunucusu ve BeanFactory 72
Spring Nesne İsimlendirme 75
Bağımlılıkları Enjekte Etme Türleri 76

Listelerin Enjekte Edilmesi 80
Basit Değerlerin Enjekte Edilmesi 83
Null ya da Boş String Değerinin Enjekte Edilmesi 83
Nesne Oluşturma Sırasının Tanımlanması 84
Otomatik Veri Tipi Dönüşümü 85
Tekil Nesneler ve Bean Scope 85
Scope Bazlı Nesnelerin Enjeksiyonu 86
Tanımlanabilir (Custom) Bean Scope 88
Metot Enjeksiyonu 90
Metot Değiştirme Yöntemi 92
Fabrika Metodu 94
Fabrika Sınıfları 95
FactoryBean Interface Sınıfı 96
Sirküler Bağımlılıklar 98
Bağımlılıkların Enjekte Edilmesi Yönteminin Avantajları 98
Bağımlılıkları Enjekte Ederken Hangi Yöntem Kullanılmalı? 99
Spring'in Motoru Nasıl Çalıştırılır? 99
XML Konfigürasyon Dosyasının Yüklenmesi 100
Çoklu XML Konfigürasyonu 102
2. Bölüm Soruları 107
3. Bölüm 109
Spring İle Nesne Yaşam Döngüsü Yönetimi 109
Spring XML İsim Alanları (XML Namespaces) 110
Bir Spring Uygulamasının Yaşam Döngüsü 113
Kurulum Fazı 114
Nesnelerin Oluşturulması 117
BeanPostProcessor Kullanımı 119
@PostConstruct Anotasyonu 121
Yaşam Döngüsü Metotlarının Kombinasyonu 122
Kullanım Fazı 123
Vekil Nesne Oluşturma 124
İmha Fazı 129
3. Bölüm Soruları 132
4. Bölüm 134
Konfigürasyon Yönetimi 134
Bean Tanımlama ve Kalıtım 135
Dahili Bean Tanımlamaları (Inner Beans) 140
Konfigürasyon Dosyalarının İmport Edilmesi 141
P İsim Alanı 144
C İsim Alanı 145
Util İsim Alanı 145
<util:constant/> 145
<util:property-path/> 146
<util:properties/> 147
<util:list/> 148
<util:map/> 148
<util:set/> 149
Spring Expression Language (SpEL) 150
Anotasyon Bazlı Konfigürasyon 155
Standart Java Anotasyonları 164
Diğer Spring Anotasyonları 166
Java Bazlı Konfigürasyon 167

@Import Anotasyonu Kullanımı 171
@PropertySource Anotasyonu Kullanımı 171
Hangi Konfigürasyon Yöntemi Kullanılmalıdır? 172
4. Bölüm Soruları 174
5. Bölüm 176
Spring İle Veri Tabanı İşlemleri 176
JDBC İle Veri Tabanı İşlemleri 177
Spring JdbcTemplate Kullanımı 182
Callback Yöntemleri 186
Hangi Callback Yöntemi Kullanılmalı? 189
NamedParameterJdbcTemplate Kullanımı 190
SimpleJdbcInsert Kullanımı 192
JDBC Batch İşlemleri 194
Hata Yönetimi 195
JDBC ve Spring DAO Support 198
DataSource Konfigürasyonu 200
JDBC Sürücüsü İle DataSource Tanımlaması 201
JNDI DataSource Kullanımı 202
Bağlantı Havuzlu DataSource Tanımlaması 203
5. Bölüm Soruları 205
6. Bölüm 207
Spring İle Transaksiyon Yönetimi 207
Transaksiyon Nedir? 208
ACID Özelliği 209
Atomik İşlem 209
Lokal Transaksiyon Yönetimi 210
Spring İle Transaksiyon Yönetimi 214
Deklaratif Transaksiyon Yönetimi 215
Sınıf Bazında Transaksiyon Yönetimi 221
Transaksiyon Özellikleri 223
İzolasyon Seviyeleri (Isolation Levels) 223
Transaksiyon Yayılması (Transaction Propagation) 225
Zaman Aşımı (Timeout) 226
Read-Only Transaksiyonlar 226
Değişiklikleri Geri Alma Kuralları (Rollback-Rules) 227
XML İle Deklaratif Transaksiyon Yönetimi 228
Spring İle Programsal Transaksiyon Yönetimi 232
Dağıtık (Distributed) Transaksiyonlar 238
6. Bölüm Soruları 245
7. Bölüm 247
Spring İle Hibernate Kullanımı 247
Boyut Farkı 248
Nesne/Relasyonel Eşleme 248
Hibernate İle Eşleme (Mapping) 249
Anotasyon Yardımı İle Eşleme (Annotation Mapping) 249
SessionFactory Konfigürasyonu 251
Transaksiyon Yönetimi 256
Hata Yönetimi 259
XML Yardımı İle Eşleme (XML Mapping) 262
XML Eşleme İçin SessionFactory Konfigürasyonu 264
HibernateTemplate Kullanımı 265
7. Bölüm Soruları 270
8. Bölüm 272

Spring İle JPA Kullanımı 272
JPA Nedir? 273
EntityManagerFactory ve EntityManager 273
Uygulama Tarafından Yönetilen EntityManager Konfigürasyonu 275
Uygulama Sunucusu Tarafından Yönetilen EntityManager Konfigürasyonu 277
JPA Anotasyonları 278
EntityManager API 278
JNDI Üzerinden EntityManagerFactory Edinme 279
JPA İle DAO Kullanımı 279
JPA İle Transaksiyon Yönetimi 281
8. Bölüm Soruları 283
9. Bölüm 285
Spring İle Aspect Oriented Programming 285
AOP Konseptleri 290
AOP Harmanlama (Weaving) Türleri 291
AOP Türleri 291
Spring AOP 292
Advice Türleri 293
Around Advice 293
Pointcut Tanımlamaları 296
Around Advice İçin XML Konfigürasyonu 302
Before Advice 304
After Returning Advice 305
After Throwing Advice 307
After Advice 309
XML İle Named Pointcut tanımlaması 310
Aspekt Parametreleri 311
Advice Sırası 313
Sınıfların Dinamik Olarak Genişletilmesi 315
Spring AOP'nin Sınırları 320
9. Bölüm Soruları 321
10. Bölüm 323
Spring MVC 323
Spring MVC ile Kullanıcı İsteğinin İşlenişi 325
Spring MVC Kurulumu 327
Spring MVC ve Uygulama Mimarisi 331
Controller Tanımlaması 333
Model Taşıyıcı ModelMap 337
View Resolver Tanımlaması 337
View Resolver Türleri 339
Araç Kiralama Formu 340
Controller Sınıfları ve Bağımlılıkların Enjekte Edilmesi 352
Spring MVC ile Çoklu Konfigürasyon Kullanımı 355
@RequestParam Anotasyonu Kullanımı 357
@PathVariable Anotasyonu Kullanımı 357
Spring MVC Tarafından Tüketilebilecek Veri Türleri 359
Spring MVC Tarafından Oluşturulabilecek Veri Türleri 361
İç ve Dış Yönlendirme 361
Hata Yönetimi 363
Genel Hata Sayfası Konfigürasyonu 367
10. Bölüm Soruları 369
11. Bölüm 371

Spring Security 371
Güvenlik Terminolojisi 372
Bir Uygulamanın Güvenlik İhtiyaçları 373
Güvenlik Bir Aspekttir 374
Neden Spring Security? 375
Spring Security Modülleri 375
Core - spring-security-core.jar 376
Remoting - spring-security-remoting.jar 376
Web - spring-security-web.jar 376
Config - spring-security-config.jar 376
LDAP - spring-security-ldap.jar 376
ACL - spring-security-acl.jar 376
CAS - spring-security-cas.jar 376
OpenID - spring-security-openid.jar 376
Spring Security XML İsim Alanı 377
Spring Security İle Web Sayfası Güvenliği 378
Spring Security Uyumlu Login Sayfası 383
Basic Authentication 386
Logout İşlemi 387
Veri Tabanı Bazlı Authentication Provider Kullanımı 387
Beni Hatırla (Remember-Me Authentication) 390
HTTPS Kullanımı 391
SpEL İle Güvenlik Konfigürasyonu 392
Spring Security JSP Tag Kullanımı 393
Metot Bazında Güvenlik 395
@Pre ve @Post Anotasyonları 399
Pointcut Örneği 401
Alan Nesnesi Bazında Güvenlik 402
SecurityContext ve SecurityContextHolder 412
11. Bölüm Soruları 413
12. Bölüm 415
Spring REST 415
Kaynakların REST ile Adreslenmesi 417
HTTP Metotları 418
GET 418
POST 419
PUT 420
DELETE 420
Spring MVC ile REST Uygulaması 421
Kaynak Edinme (GET) 422
Kaynak Oluşturma (POST) 425
Kaynak Silme (DELETE) 427
Kaynak Güncelleme (PUT) 427
Statü Kodları 428
RestTemplate Kullanımı 428
Kaynak Formatını Belirleme Stratejisi 432
REST Uygulamalarında Hata Yönetimi 435
@ExceptionHandler İle Controller Bazlı Hata Yönetimi 435
HandlerExceptionResolver Bazlı Hata Yönetimi 436
@ControllerAdvice Bazlı Hata Yönetimi 437
12. Bölüm Soruları 439
13. Bölüm 441
Spring Remoting 441
RMI Service Exporter Kullanımı 443
RMI Client Kullanımı 447

HttpInvoker Kullanımı 450
HttpInvoker Client Kullanımı 452
Hessian ve Burlap Kullanımı 454
SimpleJaxWsServiceExporter İle Web Servis Kullanımı 456
JAX-WS Client Kullanımı 458
Hangi Çözümü Kullanmalıyım? 460
13. Bölüm Soruları 461
14. Bölüm 463
Spring Web Service 463
Web Servis Uygulama Mimarisi 464
Ortak Dilin Tanımlanması 465
Veri Tipi Sözleşmesi - Message Contract 467
JAXB ile XML/Java Dönüşümü 469
Spring WS Konfigürasyonu 471
Endpoint Tanımlaması 473
WSDL ve Contract First 475
Web Servis Kullanımı 478
SOAP Mesaj Yapısı 481
TCP Monitor Kullanımı 482
POX - Plain Old XML 484
İnterseptör Kullanımı 484
Hata Yönetimi 486
Web Servis Katmanının Test Edilmesi 488
MockServiceClient 488
MockWebServiceServer 490
14. Bölüm Soruları 496
15. Bölüm 498
Spring Testing 498
TDD ile Gösterim Katmanı 502
TDD ile Servis Katmanı 513
Veri Katmanı İçin Entegrasyon Testleri 516
EmbeddedDatabaseBuilder Kullanımı 518
Testlerde Kullanılabilecek Anotasyonlar 519
15. Bölüm Soruları 523
16. Bölüm 525
Spring JMX 525
MBean Sunucusu 530
MBean Kayıt Denetimi 532
Anotasyon Bazlı Spring JMX 533
16. Bölüm Soruları 536
17. Bölüm 538
Spring Task ve Scheduling 538
Spring TaskExecutor 540
SimpleAsyncTaskExecutor 541
SyncTaskExecutor 541
ConcurrentTaskExecutor 541
SimpleThreadPoolTaskExecutor 541
ThreadPoolTaskExecutor 541
TaskExecutor Kullanımı 542
TaskScheduler Kullanımı 544
Trigger Kullanımı 545
Task İsim Alanı 546
task:executor 546

task:scheduler 546
task:scheduled-tasks 547
Anotasyon Bazlı Task Konfigürasyonu 547
@Scheduled 548
@Async 549
Uygulama Sunucu Entegrasyonu 550
17. Bölüm Soruları 551
18. Bölüm 553
Spring E-Mail 553
MIME Tipi Mesaj Oluşturma 557
Dosya Gönderme 558
İçerik Formatlama 558
Şablon Kullanımı 559
18. Bölüm Soruları 562
Cevaplar 564
1. Bölüm Cevapları 566
2. Bölüm Cevapları 567
3. Bölüm Cevapları 569
4. Bölüm Cevapları 570
5. Bölüm Cevapları 572
6. Bölüm Cevapları 572
7. Bölüm Cevapları 574
8. Bölüm Cevapları 574
9. Bölüm Cevapları 575
10. Bölüm Cevapları 576
11. Bölüm Cevapları 577
12. Bölüm Cevapları 578
13. Bölüm Cevapları 578
14. Bölüm Cevapları 579
15. Bölüm Cevapları 580
16. Bölüm Cevapları 580
17. Bölüm Cevapları 581
18. Bölüm Cevapları 582
BTSoru.com 585
KurumsalJava.com 586
EOF (End Of Fun) 587

Önsöz

Merhaba, ismim Özcan Acar. Bu satırları yazarken takvimim 2013 senesinin son günlerini gösteriyor. Kitabın on sekiz bölümünü tamamladıktan sonra bu giriş sayfasını yazıyorum. Bir sene önce şu anda okudunuz Pratik Spring Core isimli kitabı yazmaya karar verdiğimde, kitabı bitirmemin bu kadar zaman alacağını düşünmemiştim. Daha önce kaleme aldığım Extreme Programing ve Tasarım Şablonları isimli kitaplarımı üç-altı aylık zaman dilimlerinde tamamlamıştım. Bu kitabı tamamlamam bir seneyi buldu, çünkü işlenmesi gereken çok konu vardı. Bu kitabı çok büyük zevk alarak yazdım, çünkü yazarken ben de çok şey öğrendim. Umarım kitap beklentilerinizi tatmin eder.

Yazar Hakkında

1974 İzmir doğumluyum. İlk ve orta öğrenimimi İzmir'de tamamladıktan sonra Almanya'da bulunan ailemin yanına gittim. Doksanlı yılların sonunda Almanya'nın Darmstadt şehrinde bulunan FH Darmstadt üniversiteden bilgisayar mühendisi olarak mezun oldum. 2001 senesinde ilk kitabım Perl CGI, 2008 senesinde Java Tasarım Şablonları ve Yazlım Mimarileri isimli ikinci kitabım, 2009 yılında Extreme Programming isimli üçüncü kitabım Pusula tarafından yayımlanmıştır.

KurumsalJava.com ve Mikrodevre.com adresleri altında blog yazıyorum. Kurduğum BTSoru.com'da bana yazılımla ilgili sorularınızı yöneltebilirsiniz.

14.05.2013 tarihinde SpringSource Certified Integration Specialist sertifikasını aldım.

18.12.2013 tarihinde SpringSource Certified Spring Professional sertifikasını aldım.

İlk E-Kitap

Pratik Spring benim bildiğim kadarıyla baskı yapılmadan sadece e-kitap (ebook) formatında satılan ilk Türkçe yazılım kitabı olma özelliğini taşıyor. Günümüzde bilgiye daha hızlı ulaşabilmek için e-kitap çok güzel bir seçenek. Birçok okur sahip oldukları e-kitap okurlarda (ebook reader) okumak istedikleri kitapları dijital formatta yanlarında taşıyabiliyor ve istedikleri yerde e-kitapları okuyabiliyorlar. Trende, gemide, otobüste yolculuk ederken bu şekilde kitap okuyabilmek gerçekten büyük bir lüks ve eğlence. Bunun yanı sıra e-kitap formatı okurlar için ne kadar cazip olsa da, yine de baskıdan taze çıkmış bir kitabı bir bardak çayı yudumlarken okumanın keyfi de bir başka güzel :)

Kitabımın e-kitap formatında yayımlanmasına karar vermeden önce maruz kalabileceğim bir sıkıntı hakkında çok düşündüm. Malumunuz dijital bir ürünü kopyalamak çok kolay. Bu kitabı herhangi bir arkadaşınıza e-posta iletisi olarak göndermeniz yeterli. Lakin lütfen bunu yapmayın! Bu kitabı satın aldığınızda, kitabı okumak için gerekli tüm dijital hakları edindiniz. Kitabı özel kullanımınız için istediğiniz kadar çoğaltabilir ve istediğiniz dijital aygıtta okuyabilirsiniz. Lakin edindiğiniz dijital haklar kitabı başka bir şahsa verme ya da satmayı kapsamıyor. Lütfen bunun bir suç teşkil ettiğini unutmayın. Lütfen bu kitabın dijital bir kopyasının karaborsaya düşmesine engel olun. Eğer size emanet ettiğim bu kopyayı kendiniz yazmış gibi korursanız, kitap hiçbir zaman karaborsaya düşmeyecek ve sizde emin ellerde olacaktır.

Kitabın kara borsaya düşmesi, benim verdiğim tüm emeklerin karşılıksız kalması anlamına gelmektedir. Ben serbest (freelance) çalışan bir programcıyım. Bu kitabı yazabilmek için belli bir süre çalışmadım ve hayatımı sürdürmek için kazanç sağlayamadım. Birikimlerimi kullanmak zorunda kaldım. Bundan sonraki yazılımcı hayatımda da her zaman çalışabilir durumda olamayabilirim. Bu kitabın satışından elde edeceğim gelir, serbest programcı olarak karşılaştığım maddi sıkıntıları kompanse etmemi sağlayabilir. Bunun gerçekleşmesi için size emanet ettiğim bu kitaba sahip çıkmanız gerekiyor. Bunu yapacağınıza canı gönülden eminim. İlginiz, anlayışınız ve bu konudaki katkılarınız için teşekkür ederim.

Neden Pratik Spring?

Ben Spring'i ilk günlerinden beri kullanan bir yazılımcıyım. Spring çatısını

kullanmadığım bir proje hemen, hemen hatırlamam. Geçen sene sonlarında **Core Spring** ve **Enterprise Integration with Spring** kurslarına katıldım. Amacım yukarıda bahsettiğim sertifikaları almaktı.

Spring sertifika sınavlarına hazırlanma aşamasında aklıma bu kitabı yazmak geldi. Pratik Spring bu konuda Türkçe yazılan ilk kaynak olma özelliğini taşıyor. Amacım pratik bir tarzda Spring konusunu okuyucuyla buluşturmaktı. Kitabın başından, sonuna kadar tüm örnekleri tek bir uygulamayı geliştirecek şekilde yapılandırdım. Bu uygulama ile kitabın ilk bölümlerinde tanışacaksınız. İlerleyen her bölümde bu uygulamanın bir parçasını bölüm konusuna uygun olarak geliştireceğiz.

Spring Sürümü

Bu kitap Spring 3.2.5 sürümünü baz almaktadır. Bu satırları kaleme aldığım sıralarda Spring ekibi 4.0.0-SNAPSHOT sürümü üzerinde çalışmalarını sürdürüyordu. Daha önce de bahsettiğim gibi bu kitabı bir sene önce yazmaya başladım. O tarihlerde güncel olan Spring sürümü 3.2 idi. 2013 senesinin ağustos ayında 3.2.4 kullanıma sunuldu. 2013 senesi sonunda da Spring 4.0 sürümünün kullanıma sunulması bekleniyor.

Kitabın İçeriği Nedir?

Kitap Spring çatısını ve kullanılış tarzını tanıtmaktadır. Kitap 18 bölümden oluşmaktadır. Bu bölümlerin içerikleri şöyle özetleyebiliriz:

Bölüm 1

Bu bölümde Spring'in varoluş nedenini ve çalışma filozofisi incelenmektedir. Spring çatışından bahsedildiğinde en çok kullanılan kelimeler bağımlılıkların enjekte edilmesi (dependency injection), kontrolün tersine çevrilmesi (inversion of control; IoC) ve Hollywood prensibidir. Birinci bölüm bu konulara ışık tutarken, Spring çatışını oluşturan modülleri ve bu modüller kullanılarak oluşturulan Spring uygulamalarını tanıtmaktadır. Son olarak Spring 3.x sürümü ile gelen yeniliklere göz atılmaktadır

Bölüm 2

Bu bölümde Spring çatısının bağımlılıkları nasıl yönettiğini ve enjekte ettiğini örnekler üzerinde incelenmektedir. Bu bölümde yer alan bilgiler daha sonraki

bölümlerde incelenecek diğer Spring konularının temelini oluşturacak.

Bölüm 3

Bu bölüm Spring ile geliştirilen uygulamalarda yaşam döngüsünün nasıl işlediğine ışık tutmaktadır. Spring kurulum, kullanım ve imha fazla örnekler üzerinde incelenmektedir.

Bölüm 4

Spring ilk etapta bir uygulamayı oluşturan modüller arasındaki mevcut bağımlılıkları yönetmek için kullanılan bir çatıdır. Spring uygulamaları XML konfigürasyon dosyaları ya da Java konfigürasyon sınıfları aracılığı ile konfigüre edilir. Bu bölüm her iki yöntemi de tanıtmaktadır.

Bölüm 5

JDBC (Java Database Connectivity) Java dünyasında veri tabanı işlemlerini yapmak için kullanılan temel teknolojidir. Spring JdbcTemplate ve Spring DAO (Data Access Object) desteği ile uygulamalarda JDBC kullanımını daha sade bir yazılım modeliyle desteklemektedir. Bu bölümde yer alan örnekler bu yazılım modelinin kullanılış tarzına ışık tutmaktadır.

Bölüm 6

Veri tabanı üzerinde yapılan işlemler verilerin yapısal bütünlüğünü bozma tehlikesine sahiptir. Verilerin deformasyona uğramasını önlemek için veri tabanı sistemlerinin sunduğu transaksiyon mekanizmaları kullanılır. Spring bünyesinde transaksiyonları yönetmek için birçok TransactionManager implementasyonu bulunmaktadır. Bunlar JDBC, Hibernate, JPA ya da JTA gibi teknolojilerin sundukları transaksiyon yönetim mekanizmalarını yazılımcının kullanabileceği tek bir yazılım modeli altında toplamakta ve uygulamaların sadece konfigürasyon değişikliğiyle değişik transaksiyon mekanizmalarını kullanabilir olmalarını mümkün hale getirmektedir. Altıncı bölümde Spring ile hem programsal hem de dekleratif transaksiyon yönetimi ve konfigürasyonu incelenmektedir.

Bölüm 7

Yedinci bölüm Spring uygulamalarında Hibernate, Hibernate ile transaksiyon yönetimi ve HiberneteTamplete kullanımını tanıtmaktadır.

Bölüm 8

Hibernate gibi bir ORM (Object Relational Mapping) aracının kullanımı kodu bu çatıya bağımlı kılabilir. Bir Java standartı olan JPA (Java Persistence API) bu tür bağımlılıkları yok etmeyi amaçlamaktadır. Bu bölüm JPA ve kullanımını tanıtmaktadır.

Bölüm 9

Spring çatısının temelini oluşturan teknolojilerin başında AOP (Aspect Oriented Programming) gelmektedir. AOP loglama, transksiyon yönetimi, güvenlik gibi uygulama genelinde geçerli olan fonksiyonları genelleyerek, işletme mantığından ayrıştırılmalarını mümkün kılan bir teknolojidir. Bu bölüm AOP ve kullanım tarzına ışık tutmaktadır.

Bölüm 10

Spring genelde mevcut yazılım API'lerini (Application Programming Interface) entegre ederek, programcalara daha kolay bir yazılım modeli sunma eğiliminde olan bir yazılım çatısıdır. Web uygulamaları geliştirmek için tasarlanan Spring MVC bu konuda bir istisna olma özelliğine sahiptir. Wicket ya da Struts gibi web çatıları ile Spring'i kullanmak mümkün iken, Spring MVC ile tam teşekküllü bir web çatısına sahiptir. Bu bölümde Spring MVC kullanımı incelenmektedir.

Bölüm 11

Spring bazlı uygulamaların güvenliğini sağlamak için Spring Security çatısı kullanılır. Web, metot ve alan nesnesi bazlı güvenliğin nasıl uygulandığı bu bölümde incelenmektedir.

Bölüm 12

Bu bölümün konusu REST (Representational State Transfer) mimarisidir. REST bir API ya da çatı (framework) değildir. Sadece bir yazılım mimari tarzıdır. REST HTTP protokolüne bağımlı değildir. Başka protokoller kullanarak REST uygulamaları geliştirmek mümkündür. Spring ile REST uygulamaları geliştirmek için kullanılan çatı Spring MVC'dir. Bu bölüm Spring ile REST mimarisine ışık tutmaktadır.

Bölüm 13

Java'da sunucu (server) uygulamalar geliştirmek için RMI, Web Service ya da

TCP/IP Socket teknolojileri kullanılır. Spring'in ihtiva ettiği Remoting modülü ile sıradan Java sınıflarını sunucu uygulama haline dönüştürmek mümkündür. Bu bölümde Spring Remoting modülü ve sunucu oluşturma işlevi örneklerle tanıtılmaktadır.

Bölüm 14

Entegrasyon teknolojisi denildiğinde ilk akla gelen Web Service teknolojisidir. Java dünyasında SOAP ve WSDL (Web Service Description Language) kullanılarak web servis uygulamaları geliştirilir. WSDL kullanıcı ile sunucu arasındaki veri alışverişini tanımlayan meta bilgileri ihtiva eden bir yapıdır. Spring Contract First, yani WSDL dosyasını oluşturarak yola çıkma filozofisini destekleyen web servis geliştirme modelini desteklemektiktir. Bu bölümde Spring ile web servis uygulamalarının nasıl geliştirildiği ve test edildiği incelenmektedir.

Bölüm 15

Spring yazılımcıyı test konusunda da yalnız bırakmamaktadır. Bu bölümde Spring ile birim ve entegrasyon testlerinin nasıl oluşturulduğu konusuna ışık tutulmaktadır.

Bölüm 16

Bu bölüm JMX (Java Management Extentions) konusunu incelemektedir. JMX çalışır durumda olan uygulamaların konfigürasyon yönetimi ve izlenimi (monitoring) için kullanılan bir Java API'sidir.

Bölüm 17

Spring uygulamalarında paralel görev (task) koşturma işlemi için TaskExecutor kullanılır. Bunun yanı sıra TaskScheduler belli zaman birimlerinde görev koşturmak için oluşturulmuş, Spring 3.0 ile kullanıma sunulan bir interface sınıftır. Bu bölümde Spring Task ve Scheduling modülünün parçası olan bu sınıflar ve kullanımları incelenmektedir.

Bölüm 18

Kitabın son bölümünde e-posta gönderme işlemlerinin Spring çatısı ile nasıl yapabileceği tanıtılmaktadır. Spring bünyesinde e-posta gönderme işlemleri için kullanılan merkezi sınıf org.springframework.mail.MailSender sınıfıdır. Bunun yanı sıra org.springframework.mail.javamail.JavaMailSenderImpl ile JavaMail

API'yi kullanan e-posta gönderme uygulamaları geliştirilebilir. Bu bölümde ayrıca Velocity çatısı ile e-posta iletilerinin şablonlar kullanılarak nasıl yapılandırılabileceği tanıtılmaktadır.

Kitabın İçeriği Ne Değildir?

Bu kitabın amacı Java dilinde nasıl program yazıldığını öğretmek değildir! Java dilinde kendisini geliştirmek isteyen okuyuculara diğer Java kaynakları tavsiye edilmektedir. Kitapta yer alan örnekler Java dilinde hazırlanmıştır. Bu sebeple okuyucunun Java dilini biliyor olmasında fayda vardır. Kitapta yer alan Java örnekleri, anlatımı kolaylaştırmak için basit tutulmuştur.

Kitapta anlatımı kolaylaştırmak için UML diyagramları kullanılmıştır. Okuyucunun temel UML bilgisine sahip olması, verilen örneklerin anlaşımını kolaylaştıracaktır. Kitabın amacı UML'i tanıtmak ya da öğretmek değildir. UML'i öğrenmek için diğer kaynaklar tavsiye edilmektedir.

Kitap Kim İçin Yazıldı?

Bu kitap Spring çatısını yakından tanımak ve kullanmak isteyen yazılımcılar için yazılmıştır. Verilen kod örnekleri Java dilinde hazırlanmıştır. Kitap tamamen programcılara hitap edebilmek amacıyla pratik uygulamalı tarzda şekillendirilmiştir.

Kitap Nasıl Okunmalı?

Kitabın ilk dört bölümü Spring'in temellerini göstermektedir. Bu sebepten dolayı bu ilk dört bölümün mutlaka tayin edilen sıraya göre okunmasında fayda vardır. Diğer bölümler okuyucunun isteği doğrultusunda okunabilir.

Her bölümde kod parçaları göreceksiniz. Bu kod parçaları bölüm ismini taşıyan, hazırladığım Spring projesinden alıntıdır. Bu Spring projelerini Eclipse altında kullanabilirsiniz. Kodların tümünü bu kitabı satın aldığınız adresten zip dosyası olarak edinebilirsiniz.

Yazar İle İletişim

Kitap ile ilgili sorularınızı acar@agilementor e-posta adresime gönderebilirsiniz.

Benimle iletişim kurmadan önce lütfen BTSoru.com adresinde sorunuz hakkında araştırma yapınız. Bilgi paylaşımını geniş çaplı tutmak için okurlarımın sorularına BTSoru.com'da cevap vermeye çalışıyorum. BTSoru.com'da araştırma yaparken ya da soru sorarken soruların bu kitaba ait olduğunu görebilmek için lütfen pratik-spring etiketini kullanın.

PratikProgramci.com

PratikProgramci.com kaleme aldığım ve bundan sonra almayı planladığım kitapları dijital formatta sizlerle buluşturmak istediğim yeni bir eğitim platformu. Kitaplarım yanı sıra belli, başlı yazılım konularını kapsayan görsel öğrenim (screencast) modülleri oluşturarak, beğeninize sunmak istiyorum. Gelişmeleri http://www.pratikprogramci.com adresinden takip edebilirsiniz.

Kitapta Yer Alan Kod Örnekleri

Kitapta kullanılan kod örneklerini Eclipse projesi halinde http://www.pratikprogramci.com/?wpdmact=process&did=NS5ob3RsaW5r adresinden edinebilirsiniz.

1. Bölüm

Spring'e Giriş

Spring Java dünyasında yazılım geliştirmeyi basitleştirmek için geliştirilmiş bir yazılım çatısıdır (framework). Spring'i diğer çatılardan ayıran en büyük özellik temellerinin dependency injection, yani bağımlılıkların enjekte edilmesi prensibine ve AOP'ye (Aspect Oriented Programming) dayanmasıdır. Kitabın ikinci bölümünde dependency injection ve dokuzuncu bölümünde AOP konusunu detaylı olarak inceleyeceğiz.

İki binli yılların başlarında kullanıma sunulan J2EE (Java Enterprise Edition) ve EJB (Java Enterprise Bean) teknolojileri Java ile kurumsal bazlı uygulamaların geliştirilmesini amaçlamaktaydı. Nitekim çok kısa bir zamanda Java kurumsal projelerde kullanılan popüler bir teknoloji haline geldi. Bunun başlıca sebeplerinden birisi de IBM gibi büyük firmaların Java'ya verdiği destektir.

Java 1996 yılında ilk sürümü ile sadece nesneye yönelik bir programla diliyken, J2EE ve EJB teknolojileri ile birlikte büyük bir açık kaynaklı yazılım camiasını da (open source community) kapsayan bir teknoloji platformu haline geldi. Büyük bir yazılım ekosistemine sahip olan Java platformu günümüzde de kurumsal projelerde kullanılan en popüler teknoloji platformudur.

Aklınıza bu kadar geniş bir kapsama alanına sahip bir teknolojinin yanında nasıl olur da Spring gibi popüler bir çatı var olabilir sorusu gelebilir. Öncelikle şunu belirtelim: JEE (Java Enterprise Edition) Spring, Spring'de JEE değildir. Her ikisi de Java dilini kullanarak uygulama geliştirmek için geliştirilmiş teknoloji platformlarıdır. Spring var olma nedenini J2EE'ye borçlu. Bunu açıklayalım.

İki binli yılların başlarında EJB 1.x ve 2.x teknolojileri ile yazılım geliştirmiş olanlar çok iyi bilirler. Bu teknolojileri kullanarak yazılım yazmak kadar programcı için zahmetli bir uğraşı yoktur. EJB yazılım geliştirme modeli bir uygulama sunucusunun kullanımını gerektirmektedir. Geliştirilen EJB uygulamalarının çalışabilmeleri için JBoss, Weblogic ya da Glassfish gibi uygulama sunucusuna ihtiyaç duyulmaktadır. Bu ilk bakışta kötü bir şey değil. Uygulama sunucuları EJB uygulamaları için ihtiyaç duydukları transaksiyon yönetimi ya da güvenlik gibi her türlü servisi sunmaktadırlar. Programcı için bu servisleri bedel ödemeden almak, onun işletme mantığına konsantre olmasını sağlamaktadır. Ama uygulama sunucusundan bu hizmetleri alabilmek için işletme mantığının da belli bir yapıda kodlanmış ve konfigüre edilmiş olması gerekmektedir. Bu en azından EJB 3 öncesi programcıların başını ağrıtan bir durum idi. Bunun yanı sıra EJB modüllerini uygulama sunucusu dışında test

etmek mümkün değildi. Ben o tarihlerde çalıştığım kurumsal projelerde çok iyi hatırlıyorum. Günün hatırı sayılır bir bölümünü uygulamayı test edebilmek için uygulama sunucularını çalıştırıp, durdurmakla geçirirdim. Uygulama sunucusu çalışmıyorsa, işletme mantığını test etmek imkansızdı. Bir EJB 2 uygulamasını uygulama sunucusu içinde tam çalışır hale getirmek en kötü şartlarda on ya da on beş dakika sürebilir. Artık gerisini siz düşünün.

EJB 2 teknolojisi sağladığı imkanlarla kurumsal gereksinimlerin hakkını verebilecek uygulamalar geliştirmeyi mümkün kılsa da, hantallığından dolayı birçok programcının tabiri caizse nefret ettiği bir teknoloji olarak tarihe geçmiştir. Bu Spring çatışının ortaya çıkmasına sebep olmuştur.

Rod Johnson 2002 yılında kaleme aldığı **Expert One-on-One: J2EE Design and Development** isimli kitabında Interface 21 adını taşıyan altyapı ile nasıl daha hızlı ve kolay kurumsal projelerin geliştirilebileceğini anlatıyor. Interface 21 daha sonra açık kaynaklı yazılıma dönüştürüldü ve bugün tanıdığımız Spring çatısının temellerini oluşturdu.

Rod Johnson'un yazılım yaparken önerdiği temel prensip EJB'ler yerine sade Java nesnelerin (POJO; Plain Old Java Object) kullanımı. Buradan yola çıkarak Spring için şunu söyleyebiliriz: Zaten yapıları itibariyle karmaşık olan kurumsal projeler EJB 2 gibi teknolojiler kullanıldığında daha da karmaşık hale gelmektedir. Spring sunduğu POJO tabanlı programlama modeli ile kurumsal projelerin daha hızlı gerçekleştirilmelerini ve yazılımcıların verimliliğini artırmayı hedeflemektedir. **Kısaca Spring Java ile yapılan yazılımı basitleştirmek ve sadeleştirmek için vardır.**

Spring Filozofisi

Özellikle nesneye yönelik programlama teknikleri kullanıldığında, nesneler arasında var alan bağımlılıklar çok karmaşık bir yapının oluşmasına neden olabilmektedir. Uygulama geliştirme esnasında bağımlılıkların kontrol altına alınmasına dair bir çalışma yapılmadığı taktirde, yazılımcının verimliliği ve uygulamanın kod kalitesi düşecektir. Kaliteyi artırmanın ve yazılımcının daha verimli olmasını sağlamanın bir yöntemi, tüm bağımlılıkların ve oluşan karmaşık yapının dış bir uygulama çatışı (framework) tarafından yönetilmesini sağlamak olabilir. Bu bağımlılıkların uygulama tarafından değil, kullanılan uygulama çatışı tarafından yönetilmesi anlamana gelmektir. Bu yazılım filozofisine kontrolün tersine çevrilmesi ya da **Inversion of Control (IoC)**

ismi verilmektedir. Spring çatısının var oluşu ve çalışma prensipleri bu filozofiye dayanmaktadır.

Dependency Injection

Java gibi nesneye yönelik bir programlama dili ile geliştirilen uygulamalar ideal şartlarda kodun tekrar kullanıldığı modüler bir yapıdadır. Modüller birbirlerini kullanarak, yapmaları gereken işlemleri gerçekleştirirler. Bu modüller arası bağımlılıkların oluşmasını sağlar.

```
class RentalController {
    private RentalService service = new RentalServiceImpl();
}
```

Yukarıda yer alan RentalController sınıfı/modülü bunun güzel bir örneğini teşkil etmektedir. RentalController sınıfı RentalService interface sınıfına bağımlıdır. Böyle bir bağımlılık modüler bir yapının oluşturulması ve kodun tekrar kullanımını sağlamak açısından zaruridir. Lakin RentalController bünyesinde somut bir implementasyon sınıfı olan RentalServiceImpl sınıfından new operatörü ile yeni bir nesne oluşturulması, mevcut bağımlılığın değiştirilemez ve tek bir tipte olması gerektiği anlamına gelmektedir. Bağımlılığı yeniden yapılandırabilmek için kodu değiştirmek ve yeniden derlemek gerekmektedir. Bağımlılıklarını kendisi yöneten bir uygulamada kod kalitesini düşüren bu tür bağımlılıkların oluşturulmasıdır. Oysaki bağımlılıkların tersine çevrilmesi prensibine (DIP; Dependendy Inversion Principle) göre bağımlığın yönü somut değil, soyut sınıflara doğru olmalıdır.

```
class RentalController {
    private RentalService service = rentalServiceFactory.instance();
}
```

Somut bir sınıfa olan bağımlılığı yok etmek için rentalServiceFactory gibi bir fabrika (factory) sınıfından faydalanabiliriz. Fabrika tasarım şablonunu simgeleyen rentalServiceFactory bünyesinde hangi somut RentalService implementasyonunun kullanıldığını gizlemekte ve RentalController sınıfını bahsettiğim somut bağımlılıktan kurtarmaktadır. Lakin buradaki sorun rentalServiceFactory nesnesine olan bağımlılıktır. Bu nesnenin de bir şekilde new operatörü ile oluşturulması gerekmektedir.

```
class RentalController {

    private RentalService service;
```

```
    public void setService(RentalService service){
        this.service = service;
    }
}
```

Bağımlılıkları oluşturma işlemi ile hiç uğraşmasak, bunu başka birisi bizim için yapsa nasıl olurdu? Yukarıda yer alan kod örneğinde service değişkenine gerekli değer setService() metodu aracılığı atanmaktadır. Biran için setService() metodunun dış bir mekanizma tarafından koşturulduğunu düşünelim. Bu mekanizma setService() metodunu kullanarak herhangi bir RentalService implementasyonunu RentalController sınıfına enjekte edebilir. Bu işleme bağımlılıkların enjekte edilmesi yani dependency injection (DI) ismi verilmektedir. Bu işlemi yapan da Spring çatışıdır.

Aşağıda tipik bir Spring XML konfigürasyon örneği yer almaktadır. Yönetimi Spring'e devredilen bağımlılıklar için bu tarz konfigürasyon dosyaları oluşturulur. Spring bu konfigürayon dosyalarını kullanarak nesnelere arası gerekli bağımlılıkları oluşturur, yani bağımlılıkları enjekte eder.

```
<bean id="rentalController"
        class="com.kurumsaljava.spring.RentalController>
    <property name="service" ref="rentalService"/>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl/>
```

Bağımlılıkların enjekte edilmesi prensibi ile çok sade yapıda olan sınıflar oluşturabiliriz. Kendi bağımlılıklarını yönetmek zorunda olmayan bir sınıf asıl işi olan işletme mantığına konsantre olabilir. Tek sorumluluk prensibi açışından bakıldığında da bu bir gerekliliktir.

Hollywood Prensibi

VIP (Very Important Person) olan şahıslara erişmek zordur. Onlar genelde "bizi aramayın, biz sizi ararız" şeklinde iletişimi tercih ederler. Holywood prensibi olarak bilinen bu prensibi IoC konseptini açıklamak için kullanabiliriz.

Bağımlılıkların enjekte edilmesi Hollywood prensibine göre çalışmaktadır. RentalController sınıfı kendi başına bir konstrüktör ya da fabrika metodu koşturarak bir service nesnesi edinmeye çalışmaz. Bunu yapsaydı eğer, o zaman

bu VIP şahsı telefonda aramak ve benim service nesnesine ihtiyacım var demek gibi bir şey olurdu. Bunun yerine VIP şahıs, yani Spring RentalController sınıfında yer alan setService() metodunu kullanarak RentalController sınıfıyla iletişime geçmektedir. Spring RentalController sınıfına bir RentalService nesnesi enjekte edebilmek için setService() metodunu koşturmaktadır. Spring sınıfların set() metotlarını ya da konstrüktörlerini kullanarak gerek duyulan bağımlılıkları enjekte etmektedir. Bağımlılığı enjekte edebilmek için bu metotları koşturması, yani sınıfı araması gerekmektedir.

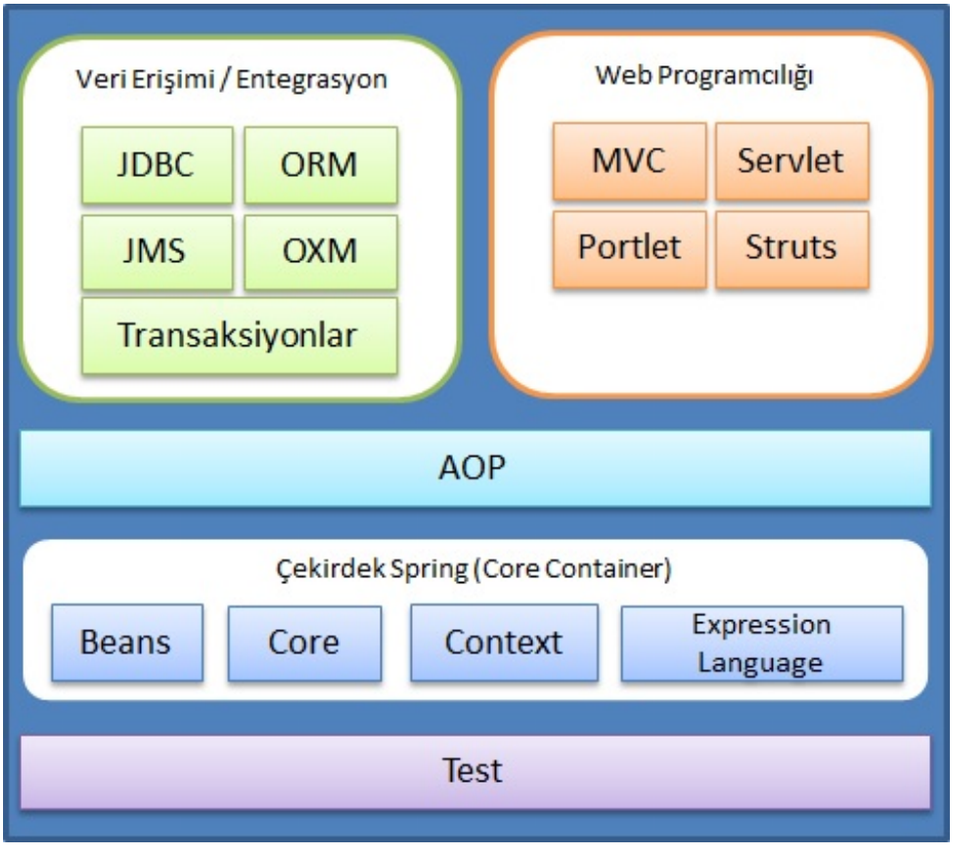
Spring koşturulacak metodun seçiminde konstrüktör ya da set() metoduyla sınırlı değildir. Hollywood prensibi sınıf bünyesinde yer alan sınıf metotları üzerinde de kullanılabilir. Bu Spring'in konfigürasyon dosyasında belirlenen herhangi bir sınıf metodunu koşturulabileceği anlamına gelmektedir. Kitabın on yedinci bölümünde inceleyeceğimiz Spring Task ve Scheduling modülünde herhangi bir POJO (Plain Old Java Object) sınıfın herhangi bir metodunu şu şekilde koşturmak mümkündür:

```
<task:scheduled ref="rentalDownloader"
        method="download" cron="*/5 * 9-17 * * MON-FRI"/>
```

Spring tarafından koşturulması gereken metodun ismi method element özelliğinde yer almaktadır. Görüldüğü gibi rentalDownloader nesnesi görevini yerine getirmek için hangi metodun koşturulması gerektiğini bilme sorumluluğundan arındırılmaktadır. Sadece konfigürasyon dosyası üzerinde değişiklik yaparak, POJO sınıfın çalışma tarzı adapte edilebilmektedir. Spring birçok modülünde bu mekanizmadan faydalanmaktadır.

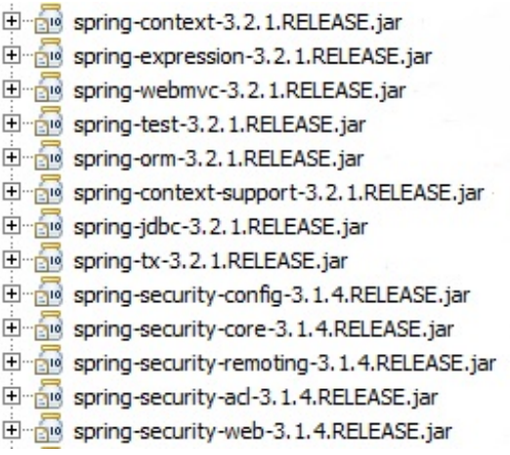
Spring Modülleri

Spring değişik modüllerden oluşan modüler bir yapıya sahiptir. Spring bir kurumsal projeyi gerçekleştirmek için gerekli her şeyi ihtiva etmekle birlikte, değişik Spring modüllerini ihtiyaçlarımız doğrultuşunda seçerek, kullanabiliriz. Spring'in modüler yapısı genel hatlarıyla resim 1.1 de görülmektedir.



Resim 1.1

Spring 3.2.4 sürümü ile yirmi değişik modülden oluşmaktadır. Her modül kendi JAR dosyasında yer almaktadır. Bu modüllerde bazıları resim 1.2 de yer almaktadır.



Resim 1.2

Spring Modülleri İle Neler Yapabiliriz?

Bu bölümde altbaşlıklar halinde Spring modüllerini tanıtmak istiyorum.

Çekirdek Sunucu (Core Container) Modülü

Spring uygulamalarının temelini çekirdek (core) sunucu (container) oluşturmaktadır. Bir Spring uygulaması çalışmaya başladığında Spring tarafından yapılan ilk işlem, içinde Spring nesnelerinin (Spring Bean) yer aldığı ve bu nesneler arasında bağımlılıkların enjekte edilmesini sağlayan bir sunucu (container) oluşturmaktır. Bağımlılıkların enjekte edilmesi için BeanFactory kullanılır. BeanFactory yapısını kitabın üçüncü bölümünde inceleyeceğiz.

Çekirdek sunucu Core, Context, Beans ve Expression Language modüllerinden oluşmaktadır. Bu modüllerin ne olduklarını ve nasıl kullanılabileceklerini kitabın ikinci bölümünden itibaren örnek kodlar üzerinde inceleyeceğiz.

Diğer tüm Spring modülleri çekirdek sunucu üzerinde inşa edilmiş modüllerdir. Her Spring uygulaması mutlaka çekirdek sunucuyu oluşturan modüllerde yer alan sınıflar kullanılarak konfigüre edilir. Spring çekirdek sunucu yapı itibari ile bir EJB uygulama sunucusuna benzer. Bünyesinde yer alan tüm nesnelere konfigürasyonları doğrultusunda ihtiyaç duydukları servisleri sunar. Bunu yaparken AOP teknolojisini kullanır.

Spring AOP Modülü

Spring uygulamalarını POJO sınıfların oluşturduğunu söylemiştik. Bu sınıflar mevcut yapıları itibari ile bir kurumsal projelerin gereksinimlerini tatmin edecek yapıda değildirler. Örneğin EJB komponentler uygulama sunucusu tarafından güvenlik ya da otomatik transaksiyon yönetimi gibi yetilerle donatılırlar. Aynı şey Spring tarafından AOP kullanılarak POJO sınıflar üzerinde gerçekleştirilir. Örneğin sadece işletme mantığını ihtiva eden bir POJO sınıfa AOP kullanılarak transaksiyonel özellik kazandırılabilir. Kitabın dokuzuncu bölümünü AOP konusuna ayırdım.

Veri Erişimi Modülü

JDBC tabanlı veri tabanı işlemleri için Spring bünyesindeki JDBC modülünü kullanabiliriz. JDBC kodu yazmış olanlar bilirler. JDBC checked exception türünü kullandığı için JDBC kodu çok kısa zamanda okunmaz bir hale gelebilir. Spring bünyesinde yer alan JDBC modülü ile çok sade JDBC kodu yazmak mümkündür. Sağladığı DAO (Data Access Object) katmanı ile, veri katmanı ile veri tabanı arasında esnek bir bağın oluşmasını destekler. Kitabın beşinci bölümü bu modülü tanıtmaktadır.

JDBC yerine Hibernate ya da EclipseLink gibi bir ORM (Object Relational

Mapping) teknolojisini tercih edenler Spring ORM modülü ile bu teknolojileri Spring uygulamalarında kullanabilirler. Spring ORM modülü bir ORM çatısı olma iddiasını gütmemektedir. Daha ziyada mevcut ORM teknolojilerini entegre ederek, tek bir programlama modeli üzerinden kullanımlarını sağlamaktadır. Kitabın yedinci bölümünde Spring ile Hibernate, sekizinci bölümünde JPA (Java Persistence API) kullanımını yakından inceleyeceğiz.

Spring OXM (Object XML Mapping) modülü Java<=>XML dönüşümünü sağlamak için kullanabileceğimiz Spring modülüdür. Yine diğer modüllerde de olduğu gibi bir OXM çatısı olma iddiası gütmemektedir. Daha ziyada JAXB, Castor, XMLBEans ve XStream gibi teknolojileri kullanarak Java<=>XML dönüşümünü sağlamaktadır. Kitabın on ikinci bölümününde yer alan REST ve on dördüncü bölümününde yer alan Web Service konularında Spring OXM modülünün kullanımını inceleyeceğiz.

Spring JMS (Java Messaging Service) JMS mesajları oluşturmak (produce) ve tüketmek (consume) için kullanılan modüldür.

Transaksiyon modülü deklaratif ve programsal transaksiyon yönetimi yapılmasını sağlamaktadır. Kitabın altıncı bölümünde Spring ile transaksiyon yönetimini inceleyeceğiz.

Spring MVC Modülü

Spring genelde entegratif bir çatıdır, yani mevcut teknolojileri aynı çatı altında toplayarak, belli bir programlama modeli sunar. Bunun bozulduğu istisnalardan bir tanesi Spring MVC çatısıdır. Bu çatı ile web tabanlı uygulamalar geliştirmek mümkündür. Spring ile Struts ya da Wicket gibi web çatılarını kullanmak mümkün iken, Spring burada kendi web çatısını geliştirmeyi tercih etmiştir. Kitabın onuncu bölümünde Spring MVC web çatısını yakından inceleyeceğiz.

Spring Remoting Modülü

Spring MVC modülü ile web tabanlı uygulamalar geliştirmek mümkün iken, Spring Remoting modülü ile POJO bazlı sınıfları servis sunucusu haline dönüştürmek mümkündür. POJO sınıflar RMI (Remote Method Invocation), Hessian, Burlap, JAX-WS (Web Service) ve HTTP invoker protokol ve teknolojileri kullanılarak servis sunucu haline getirilebilir. Ayrıca Spring Remoting ile mevcut servis sunucuları ile iletişimde kullanılabilecek kullanıcılar (client) oluşturulabilir. Kitabın on üçüncü bölümü bu modülün kullanılış tarzını tanıtmaktadır.

Spring Test Modülü

Yazılımcı olarak benim Spring'in en çok ilgimi çeken tarafı, test güdümlü yazılımı mümkün kılan bir test çatısına sahip olmasıdır. Spring Test JUnit ve TestNG çatılarını kullanarak birim ve entegrasyon testlerinin geliştirilmesini mümkün kılmaktadır. Kitabın on beşinci bölümünde Spring ile test seçeneklerini inceleyeceğiz.

Spring Uygulama Portföyü

Spring çekirdek uygulama çatısı haricinde, bu çekirdek çatı üzerine inşa edilmiş uygulamaları da ihtiva etmektedir. Bu uygulamalardan bazıları:

- ***Spring Integration*** - Spring programlama modelini kurumsal entegrasyon şablonları (Enterprise Integration Patterns) destekleyecek şekilde genişletir. Spring Integration birbirlerinden tamamen bağımsız olan yazılım modüllerinin mesajlaşma (messaging) teknikleri kullanılarak bir uygulama oluşturacak şekilde bir araya getirilmelerini amaçlamaktadır.
- ***Spring Batch*** - Kullanıcı arayüzüne ihtiyaç duymadan seri bir şekilde işlem yapmayı (batch application) mümkün kılar. Özellikle bankalar gibi müşteri işlemlerini mesai saatlerinden sonra topluca yapan kuruluşların bu yöndeki ihtiyaçlarını karşılamak amacıyla geliştirilmiştir.
- ***Spring Social*** - Spring uygulamalarını Facebook, Twitter, ve LinkedIn gibi sosyal medya platformları ile entegre etmek için geliştirilmiş uygulamadır.
- ***Spring Mobile*** - Spring MVC web yazılım çatısını genişleten ve mobil web uygulama yazılımını kolaylaştırmak için kullanılan uygulamadır.
- ***Spring Web Services*** - SOAP bazlı web servis uygulamaları geliştirmek için kullanılmaktadır.
- ***Spring Web Flow*** - Temelinde Spring MVC web çatısını kullanan Web Flow kurumsal bir işlemi birden fazla web sayfasına bölerek, işlemin adım, adım yapılmasını sağlayan uygulamadır. Web Flow sayfalar arası otomatik oturum ve durum yönetimini yapmaktadır.
- ***Spring LDAP*** - Spring bazlı uygulamaları için LDAP (Lightweight Directory Access Protocol) kullanımını basitleştirmektedir.
- ***Spring Security*** - Spring uygulamalarında güvenlik konfigürasyonunu yapmak için kullanılan uygulama modülüdür.
- ***Spring Data*** - Relasyonel veri tabanı sistemleri yanı sıra map-reduce, NOSQL, bulut gibi yeni veri tabanı ve veri erişimi teknolojilerinin

kullanımını kolaylaştırmaktadır. JPA, MongoDB, Neo4j, Redis, Hadoop, Gemfire, Rest, Solr, CouchBase ve ElasticSearch gibi teknolojileri destekleyen alt modülleri mevcuttur.
- ***Spring XD*** - Büyük veri (big data) uygulamalarında import, export, analiz ve batch işleme gibi işlemleri kolaylaştırmak için oluşturulmuş uygulamadır.
- ***Spring Roo*** - Java ile uygulama geliştiren yazılımcılar için hızlı uygulama geliştirme (rapid application development) aracıdır.

Spring 3 İle Gelen Yenilikler

Bu bölümde Spring 3.0, 3.1 ve 3.2 sürümlerinde yer alan yenilikleri sizlerle paylaşmak istiyorum. Bu sürümlerde göze çarpan yenilikler şunlardır:

Spring 3.0

- Bu sürüm ile tüm Spring çatısı Java 5 ile gelen Generics, Varargs ve diğer Java dili yeniliklerini kullanacak şekilde elden geçirilmiştir. Spring'de anotasyon desteği 2.5 sürümü ile gelmiş olsa bile, Spring 3.0 sürümü ile bu destek daha da artırılmış ve JSR-330 ile gelen standart Java anotasyonların kullanımı mümkün hale gelmiştir. Bu şekilde anotasyon bazlı konfigürasyon kullanıldığında standart Java anotasyonları kullanılarak, kod bazındaki Spring çatısına olan bağımlılık ortadan kaldırılabilmektedir.
- Spring uygulamalarında konfigürasyon XML dosyaları üzerinden yapılmaktadır. Spring 3.0 sürümü ile XML dosyası kullanmadan anotasyon bazlı konfigürasyon yapmak mümkün hale gelmiştir.
- Spring 3.0 konfigürasyon imkanlarını daha esnek hale getirmek için Spring Expression Language (SpEL) modülünü ihtiva etmektedir.
- 3.0 sürümü ile Spring MVC uygulamalarını daha kolay konfigüre etmek için yeni XML mvc isim alanı oluşturulmuştur. Yeni eklenen @CookieValue ve @RequestHeaders gibi anotasyonlarla Spring MVC uygulamalarının anotasyon bazlı konfigürasyonu genişletilmiştir.
- Web uygulamaları geliştirmek için kullanılan Spring MVC, 3.0 sürümü ile REST (Representational State Transfer) desteği sağlamaktadır. Spring MVC ile bir REST uygulaması oluşturmak için Spring MVC controller sınıfları kullanılmaktadır. RestTemplate kullanıcı (client) uygulamalar geliştirmek için kullanılmaktadır.
- jdbc isim alanında yer alan embedded-database konfigürasyon elementi ile

HSQL, H2, ve Derby gibi veri tabanı sistemlerinin kullanımı kolaylaştırılmıştır.
- Java EE 6 ile kullanıma sunulan @Asynchronous anotasyonu ile metotlar asenkron koşturulabilmektedir. Spring 3.0 sürümünde yer alan @Async anotasyonu ile bu desteği sağlamaktadır.
- Spring 3.0 JSR-303 (Bean Validation) bünyesinde yer alan anotasyonları desteklemektedir.

Spring 3.1

- Yeni bir caching modülü (Cache Abstraction) ihtiva etmektedir.
- Bu sürümle Spring bean tanımlamalarını profil bazında gruplamak (bean definition profiles) mümkün hale gelmiştir. Profiller yardımı ile uygulama değişik ortamlara göre adapte edilmiş konfigürasyon dosyalarını kullanabilmektedir.
- Oluşturulan yeni Environment isimli sınıf ile profil bazlı bilgilerin yer aldığı yeni bir alan oluşturulmuştur. Bu alan içinde profil bilgileri yanı sıra tanımlanan değişken (property) değerleri de yer almaktadır. Environment sınıfı kullanılarak bu bilgilere ulaşılabilir.
- constructor-arg elementini daha kısa yazmak için c isim alanı oluşturulmuştur. Bunun kullanımını üçüncü bölümde yakından inceleyeceğiz.
- Bu sürüm Hibernate 4.x desteği vermektedir.
- 3.1 sürümü öncesi enjeksiyon için kullanılan set metotlarının void veri tipinde bir değeri geri vermeleri gerekiyordu. Bu yeni sürümle set metotları herhangi bir yapıda olabilmektedir.
- Spring'in test çatısı olan Spring TestContext bünyesindeki @ContextConfiguration anotasyonu @Configuration anotasyonunu taşıyan konfigürasyon sınıflarını desteklemektedir. Ayrıca entegrasyon testlerinde kullanılmak üzere değişik uygulama profillerini destekleyen @ActiveProfiles anotasyonu oluşturulmuştur.
- JPA bünyesinde sınıflar META-INF/persistence.xml dosyasında tanımlanmaktadır. Spring 3.1 sürümü ile gelen LocalContainerEntityManagerFactoryBean ile classpath içinde yer alan sınıflar otomatik olarak taranarak persistence.xml kullanmayan bir JPA altyapısı oluşturulabilmektedir.
- Spring MVC controller sınıflarında kullanılan @RequestMapping anotasyon tanımlaması consumes ve produces elementleri kullanılarak genişletilmiştir. consumes controller sınıfının hangi türde verileri

işleyebileceğini belirlerken, produces kullanıcıya gönderilecek cevabın hangi formatta olması gerektiğini tanımlamaktadır. Böylece örneğin bilgileri XML formatında alan ve kullanıcıya cevabı JSON formatında gönderen controller sınıfları tanımlamak mümkün hale gelmiştir.
- Yeni oluşturulan @RedirectAttributes anotasyonu ile controller metotlarında yönlendirme (redirect) işlemi için parametre tanımlaması yapılabilmektedir.
- @RequestBody anotasyonu kullanılan bir controller metodunda @Valid anotasyonu kullanılarak otomatik validasyon işlemi yapmak mümkün hale gelmiştir.

Spring 3.2

- Spring MVC uygulamalarını uygulama sunucusuna bağımlı olmadan test edebilmek için yeni Spring MVC Test çatısı oluşturulmuştur.
- Bu sürüm Java EE 7'nin bir parçası olan JCache desteği sağlamaktadır.
- Spring MVC, Servlet 3 sürümünde tanımlanan asenkron metot koşturma (asynchronous request processing) özelliğini desteklemektedir.
- RestTemplate HTTP cevaplarında (response) yer alan verileri Java Generics kullanarak (örnegin List) edinebilmektedir.
- Spring bu sürümünde Jackson JSON 2 kütüphanesini ve bir şablon (template) yönetim çatısı olan Tiles 3 sürümünü desteklemektedir.

Spring'in Uygulama Geliştirmedeki Rolü

Spring kurumsal Java projeleri geliştirmek için geniş çaplı altyapısal destek sağlamaktadır. Entegratif yönüyle mevcut Java API (Application Programming Interface) ve çatıların (framework) kullanımını mümkün olduğu kadar tek bir programlama modelinde toplamaktadır. Değişik API ve çatıları tek bir programlama modeli ile kullanabilmek programcıların verimliliğini artıran bir durumdur.

Spring plumbing code olarak isimlendirilen, işletme mantığının mecbur kılınan programlama modeli neticesinde gereksiz kod kalabalığı ile şişmesini sunduğu konfigürayon yöntemleri ile engellemektedir. Böylece POJO sınıflar geliştirerek, sadece işletme mantığına konsantre olmak mümkün hale gelmektedir.

Spring'in çekirdeği uygulama konfigürasyonu, kurumsal entegrasyon, test etme ve veri erişimi konuları için çözümler sunmaktadır. Spring ile bir uygulamayı

değişik modülleri bir araya getirerek oluşturmak mümkündür. Modüllerin birbirlerini bulmaları gerekliliği yoktur. Her modül Spring konfigürasyonu aracılığı ile uygulamanın genel yapısına zarar vermeden başka bir modül ile yer değiştirebilir. Uygulamayı oluşturan modüllerin sessiz, sedasız değiştirilebilir yapıda olması, uygulamanın test edilebilirliğini olumlu etkilemektedir. Sunduğu test imkanları ile Spring uygulamalarını, buna Spring MVC ile oluşturulan web uygulamaları da dahildir, uygulama sunucusu olmadan test etmek mümkündür.

Kaynakları yönetmesi, veri erişimi için kullanılan API'lerin kullanımını kolaylaştıran sınıflar sunması, JDBC, Hibernate, JPA ve IBatis gibi popüler veri erişimi teknolojilerini desteklemesi ile Spring veri erişimi programcılığını kolaylaştırmaktadır.

Spring Struts, Wicket ya da JSF gibi web çatıları ile entegre edilebilmektedir. Bu entegrasyon ile bu çatılarda Spring'in sunduğu uygulama konfigürasyonu modeli kullanılabilmektedir. Bunun yani sıra ihtiva ettiği Spring MVC ve Spring Web Flow çatıları ile web uygulama geliştirmeyi desteklemektedir.

Spring Yazılım Geliştirme Ortamı

Spring STS (SpringSource ToolSuite) ismini taşıyan ve Eclipse bazlı bir yazılım geliştirme ortamına sahiptir.

STS ile tam teşekküllü bir yazılım geliştirme ortamı edinmenin yanı sıra mevcut bir Eclipse Helios (3.6), Indigo (3.7), Juno (3.8/4.2) ya da Keppler 4.3 sürümü Eclipse Marketplace üzerinde STS plugin seti yüklenerek Spring yazılım ortamına dönüştürülebilir.

Resim 1.3

Spring Jar Dosyalarını Nasıl Edinebilirim?

Spring çatısını oluşturan Jar dosyalarını edinmenin en hızlı yolu bir Maven projesi oluşturmak ve aşağıda yer alan Maven bağımlılığını projeye eklemektir. Kitabın her bölümü için oluşturduğum Maven projelerinde bu şekilde gerekli Jar dosyalarını projeye ekledim.

```
Kod 1.1 - pom.xml

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

Eğer Maven kullanmıyorsanız, Spring.io sayfasından güncel Spring sürümünü edinebilirsiniz.

Spring Hello World

Her teknolojiyi öğrenmeye klasik Hello World uygulaması ile başlanır. Bu bölümde Spring ile bu tarz bir uygulamanın nasıl geliştirilebileceğini bir örnek üzerinde göstermek istiyorum.

İlk işlem olarak HelloWorldService isminde bir interface sınıf tanımlıyoruz. Bu interface sınıf kod 1.2 de yer almaktadır. getMessage() metodu istediğimiz türde bir mesajı geri verecektir.

```
Kod 1.2 - HelloWorldService

public interface HelloWorldService {
    String getMessage();
}
```

HelloWorldService sınıfını implemente eden sınıf kod 1.3 de yer almaktadır. getMessage() metodunu Hello World kelimelerini geriye verecek şekilde yapılandırıyoruz.

```
Kod 1.3 - HelloWorldServiceImpl

public class HelloWorldServiceImpl implements HelloWorldService {
    @Override
    public String getMessage() {
        return "Hello World";
    }
}
```

Şimdi HelloWorldService sınıfını kullanan başka bir sınıf tanımlayalım. Kod 1.4 de yer alan MessageManager sınıfı bünyesinde service ismi altında HelloWorldService sınıfını kullanmaktadır. Bir Spring anotasyonu olan @Autowired ile MessageManager sınıfına, daha doğrusu bu sınıftan olan nesneye bir HelloWorldService nesnesi enjekte edilmektedir.

```
Kod 1.4 - MessageManager

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MessageManager {
```

```
    @Autowired
    HelloWorldService service;

    public void printMessage() {
        System.out.println(this.service.getMessage());
    }
}
```

Spring uygulamasını konfigüre etmek ve koşturmak için Application sınıfını (kod 1.5) oluşturuyoruz. @Configuration sınıf bazlı Spring konfigürasyonu kullandığımıza işaret etmektedir. Bu Spring uygulaması sıfır XML konfigürasyonu kullanmaktadır. @ComponentScan ile Spring'e gerekli sınıfları classpath içinde araması gerektiğini belirtiyoruz.

@Bean anotasyonu bir Spring bean tanımlamak için kullanılmaktadır. Kod 1.4 de yer alan MessageManager sınıfına HelloWorldService tipinde bir nesne enjekte edebilmek için Spring'in hangi HelloWorldService implementasyon sınıfının kullanıldığını bilmesi gerekmektedir. Bu uygulamada kullandığımız HelloWorldService implementasyonu kod 1.3 de yer alan HelloWorldServiceImpl sınıfıdır. Spring getMessageService() metodunu yeni bir HelloWorldServiceImpl nesnesi oluşturmak için kullanacak, akabinde bu nesneyi kod 1.4 de yer alan service değişkenine enjekte edecektir, çünkü bu değişken @Autowired ile işaretlenmiştir.

```
Kod 1.5 - Application

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
                AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class Application {

    @Bean
    public HelloWorldService getMessageService() {
        return new HelloWorldServiceImpl();
    }

    public static void main(final String[] args) {
        final ApplicationContext context =
             new AnnotationConfigApplicationContext(
```

```
                                Application.class);
        final MessageManager manager =
                context.getBean(MessageManager.class);
        manager.printMessage();
    }
}
```

Kod 1.5 de yer alan Application sınıfı hem uygulamayı konfigüre etmek, hem de koşturmak için kullanılmaktadır. @Configuration, @ComponentScan ve @Bean anotasyonları uygulamayı konfigüre etmek için kullanılırken, main() metodu uygulamayı koşturmaktadır. main() metodu bünyesinde yeni bir AnnotationConfigApplicationContext nesnesi oluşturulmaktadır. Bu nesne konfigüre ettiğimiz Spring uygulamasını temsil etmektedir. Bu nesne üzerinden getBean() metodu aracılığı ile bir MessageManager nesnesi edinebiliriz. manager.printMessage() metodu koşturulduğunda ekranda Hello World kelimeleri yer alacaktır.

Bu Spring ile yaptığımız çok basit bir dependency injection örneğiydi. Şimdi sır perdesini aralamaya ve Spring'i daha yakından tanımaya hazır mısınız? Öyleyse Spring trenine binip, on sekiz durağa (kitabın bölümleri) uğrayarak, Spring ile neler yapabileceğimizi birlikte görelim. Birlikte yapacağımız bu yolculukta Spring hakkında umduklarınızı bulacağınızı ümit ediyorum.

Hazırsanız tren kalkıyor...

1. Bölüm Soruları

Her bölüm sonunda, bölümle ilgili sorular bulunmaktadır. Soruların cevaplarını kitabın son bölümünde bulabilirsiniz.

1.1 Spring'in yükselişini önlemek için J2EE nasıl bir yapıda olmalıydı?

1.2 Spring'in ana amacı nedir sorusunu tek cümle ile nasıl cevaplarsınız?

1.3 Spring ilk etapta hangi yazılım prensibine dayanmaktadır?

1.4 Dependy injektion nedir?

1.5 Yazılımda nihayi amaç bağımlılıkları yok etmek midir?

1.6 Bağımlıkları kontrol edilebilir hale getirmek için kullanılan tasarım prensibi hangisidir?

1.7 Spring'i oluşturan temel modüller hangileridir?

1.8 Spring modülleri ile Spring uygulamaları arasındaki fark nedir?

1.9 İlk hangi sürüm ile Spring anotasyonları kullanmaya başlamıştır?

1.10 Java bazlı Spring konfigürasyon hangi sürüm ile gelmiştir?

2. Bölüm

Spring İle Tanışalım

Bir Program Nasıl Oluşur?

Spring'in detaylarına girmeden önce, bir programın oluşum hikayesine göz atmamızda fayda var. Bir programın oluşumundaki yön verici en önemli etken, programı kullanacak olan müşterinin iş piyasasındaki gereksinimleridir. Program müşterinin iş hayatını kolaylaştırmak, firma bünyesindeki aktiviteleri organize etmek ve kazanç sağlamak için kullanılır. Yazılım esnasında müşteri tarafından oluşturulan kriterlerin dikkate alınması gerekmektedir, aksi taktirde müşterinin isteklerini karşılayamayan ve günlük iş hayatında kullanılamaz bir program ortaya çıkar.

Yazılım süreci müşteri isteklerinin analizi ile başlar. Analiz safhasında müşterinin gereksinimleri tespit edilir ve yazılım için gerekli taban oluşturulur. Analiz ve bunu takip eden yazılım, müşteri isteklerinin transformasyona uğradığı ve netice olarak bir programın oluşturulduğu karmaşık bir süreçtir.

Resim 2.1

Transformasyon müşteri gereksinimlerinin tespiti ile başlar. Sadece müşteri ne istediğini bilebilir ve programcı olarak bizim görevimiz, müşterinin istekleri doğrultusunda programı şekillendirmektir. Şimdi bu sürecin nasıl işlediğini bir örnek üzerinde birlikte inceleyelim.

Hayali bir müşterimiz bizden sahip olduğu araç kiralama servisini yönetmek için bir yazılım sistemi oluşturmamızı istiyor. Kitabı oluşturan bundan sonraki bölümlerin hepsinde araç kiralama servisini bölüm konusuna uygun olacak şekilde Spring çatışını kullanarak geliştireceğiz. Vereceğim tüm örnekler bu uygulamadan alıntıdır ve kodlar her bölüm için hazırladığım Maven projesinde yer almaktadır. Bu projeleri Eclipse altında kullanabilirsiniz.

Araç Kiralama Servisi

Spring'in nasıl kullanıldığını göstermek amacıyla bu bölümden itibaren hayali bir araç kiralama servisi için gerekli yazılım sistemini oluşturmaya başlayacağız. Bu yazılım sistemi aracılığı ile müşteriler internet üzerinden

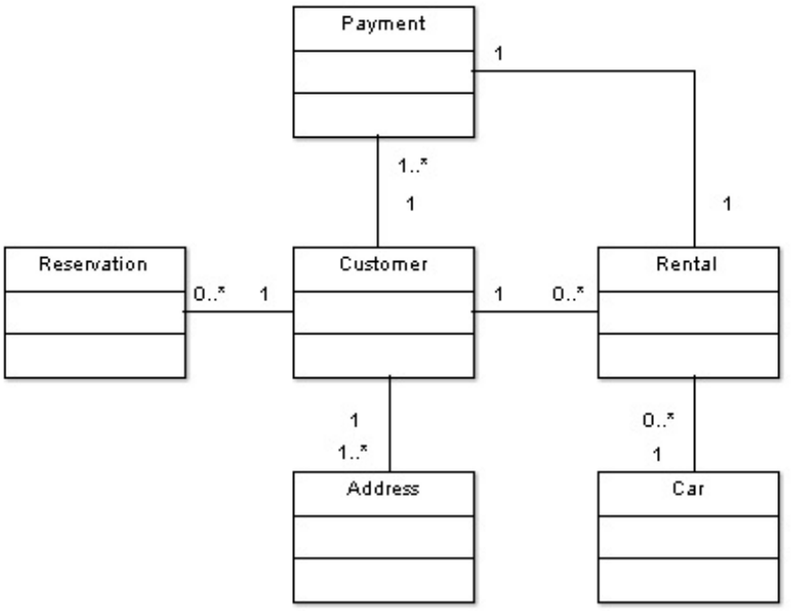
rezervasyon ve kiralama işlemlerini yapabilecekler.

Program Alan Modeli (Domain Model)

Müşteri gereksinimlerini program koduna dönüştürebilmemiz için bir model oluşturmamız gerekiyor. Bu modelden yola çıkarak neyin nasıl programlanması gerektiğini daha iyi anlayabileceğiz ve yazılım sürecine yön vereceğiz. Yazılımda bu tür modellere alan modeli (domain model) ismi verilmektedir. Alan ile, programın kapsadığı çalışma alanını kastedilmektedir. Araç kiralama servisi örneğinde içinde bulunduğumuz alan araç kiralama servisinin çalışma sahasıdır.

Alan modelinde çalışma sahasında kullanılan birimler, bu birimlerin rolleri ve birbirleriyle olan ilişkileri yer alır. Bu birimler çalışma sahasına has kullanılan terminolojiden türetilir. Bir araç kiralama servisinde çalışan personelin aralarında yaptıkları konuşmalara kulak verdiğinizde araba, müşteri, rezervasyon, ödeme, adres, rezervasyon iptali gibi bu çalışma sahasına has terimler duyarsınız. Bu terimler genelde oluşturulan alan modelinde yer alan birimlerdir. Müşterinin sahip olduğu gereksinimleri tespit edebilmek için yazılım ekibi tarafından gerekli soruların sorularak, çalışma sahasında yer alan birimlerin ortaya çıkarılması gerekmektedir.

Yazılım sistemleri için alan modelleri UML (Unified Modeling Language) olarak isimlendirilen bir modelleme dili ile yapılır. Aşağıda araba kiralama servisi için oluşturduğumuz alan modeli yer almaktadır.



Resim 2.2

Resim 2.2 de yer alan alan modelinde birimler arası bağlantılar olduğu görülmektedir. Bu bağlantılar iki birim arasındaki interaksiyonu tanımlamak için kullanılmaktadır. Birimler birbirlerini kullanarak, modelledikleri verilerin kullanım ve işlenme tarzlarını belirlerler.

Sınıflar arası ilişkilerin derecesini belirtmek için rakamlar kullanıyoruz. Kullanılan rakamlar şu şekildedir:

0,1	Sınıf karşı sınıf tarafından ya hiç kullanılmamaktadır ya da en fazla bir kez kullanılmaktadır.
0..*	Sınıf karşı sınıf tarafından ya hiç kullanılmamaktadır ya da birçok kez kullanılmaktadır.
1,1	Sınıf karşı sınıf tarafından ya en az bir kez ya da en fazla bir kere kullanılmaktadır.
1..*	Sınıf karşı sınıf tarafından ya en az bir kez ya da birçok kez kullanılmaktadır.

Şimdi bu şemayı kullanarak araç kiralama servisi alan modelinde yer alan nesne ilişkilerine bir göz atalım.

Bir müşteriyi temsil eden Customer ile bir araç kiralama işlemini temsil eden

Rental arasında 1-0,* ilişkisi vardır. Buna göre bir müşteri sıfır ya da birden fazla araç kiralamış olabilir ve her bir araç kiralama işlemi sadece bir müşteri için yapılmıştır.

Rental ve kiralanan aracı temsil eden Car arasında 0,*-1 ilişkisi vardır. Buna göre bir araç kiralama işleminde (Rental) sadece bir araç kiralık olarak verilebilir. Buna karşın bir araç sıfır ya da birden fazla müşteri için kiraya verilmiş olabilir.

Bir müşteri rezervasyonunu temsil eden Reservation ile Customer arasında 0,*-1 ilişkisi mevcuttur. Bu bir rezervasyonun sadece bir müşteriye ait olduğunu, bir müşterinin sıfır ya da birden fazla rezervasyonu olabileceği anlamına gelmektedir.

Müşteri tarafından yapılan ödemeyi temsil eden Payment ile Customer arasında 1,*-1 ilişkisi bulunmaktadır. Bu yapılan bir ödemenin bir müşteriye ait olduğu, bir müşterinin en az bir ya da daha fazla ödemesi bulunduğuna işaret etmektedir.

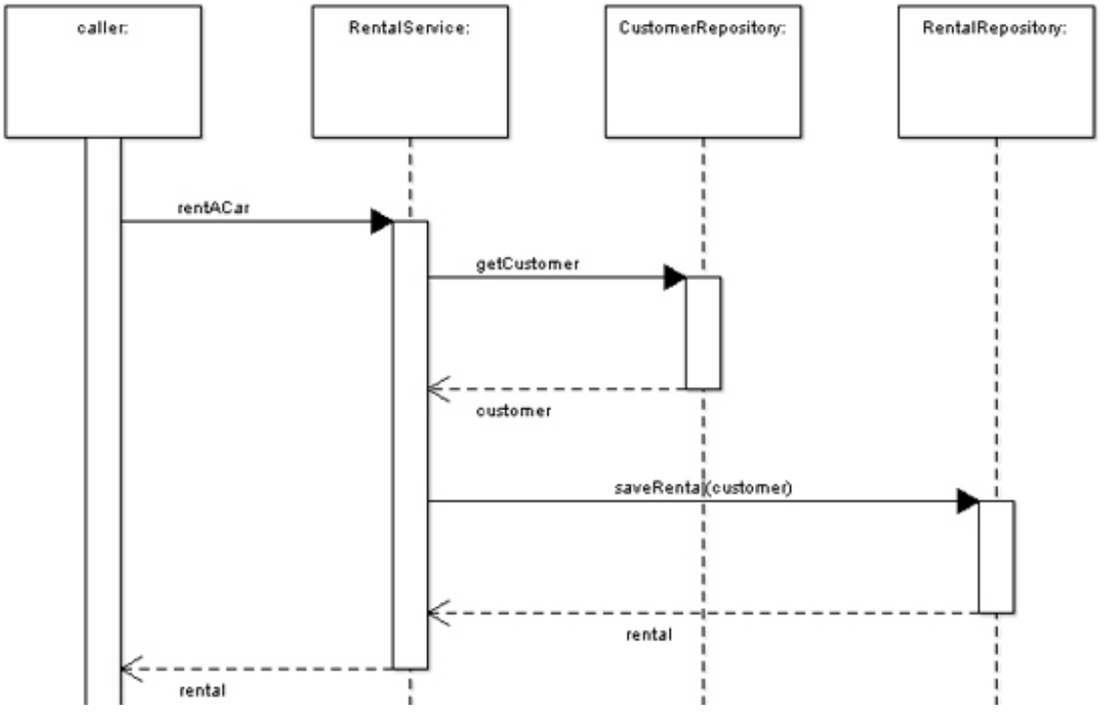
Müşterinin adresini temsil eden Address ile Customer arasında 1-1,* ilişkisi bulunmaktadır. Bu her müşterinin en az bir adresi olması gerektiği, her adresin mutlaka bir müşteriyi ait olduğu anlamına gelmektedir.

Spring ile geliştireceğimiz uygulamada alan modelinde yer alan birimlerin hepsini birer Java sınıfı olarak implemente edeceğiz. Bu tür sınıflara yazılımda entity (öğe) ismi verilmektedir. Bu öğeler aracılığı ile uygulama bünyesinde kullanılan verileri modelleyebiliriz. Bir Customer öğesi örneğin bir müşteriyi temsil eden tüm özellikleri ihtiva eder. Bu öğelerin Java gibi nesneye yönelik bir programlama dilinde sınıf olarak kodlanmaları, yazılımcının öğeler arasındaki ilişkileri daha iyi anlamasını sağlamaktadır.

Bir uygulama sadece alan modelinde yer alan entity nesnelerinden oluşmaz. Alan modeli veri yapılarını modellemek için kullanılır. Asıl uygulama bu verileri üzerinden işlem gerçekleştiren yazılım yapısıdır. Veriler üzerinde yapılan işlemleri ve program akışını daha iyi kavramak için dizge diyagramları oluşturabiliriz. Dizge diyagramları hangi kod birimlerinin hangi işten sorumlu olduğunu gösterirler. Bu şekilde uygulamanın hangi parçalardan oluştuğunu anlamak ve parçaları geliştirmek kolaylaşır.

Dizge Diyagramı (Sequence Diagram)

Kod yazmadan önce program akışını görselleştirmek için kullanabileceğimiz diğer bir araç ta UML dizge diyagramlarıdır. Resim 2.3 de bir araç kiralama işlemi için yapılması gereken işlemler yer almaktadır. RentalService, CustomerRepository ve RentalRespository diyagram bünyesinde kullandığımız Java sınıflarıdır. Bu diyagram bünyesinde hangi sınıfın hangi metodu koşturarak, işlem yaptığı yer almaktadır.



Resim 2.3

İlk Çözüm

Spring kullanmadan araç kirala servisi uygulamasını nasıl geliştirirdik? Bu sorunun cevabını aramaya koyulmadan önce bir birim testi yazalım. Bu birim testi bize uygulamanın geliştirilmesinde yol gösterici nitelikte olacaktır. Yeni bir kiralama işlemini test eden birim testi kod 2.1 de yer almaktadır.

```
Kod 2.1 - RentalServiceTest

package com.kurumsaljava.com.spring;

import static junit.framework.Assert.assertTrue;
import java.text.SimpleDateFormat;<br/>
import java.util.Date;<br/>
import org.junit.Test;<br/></code>

public class RentalServiceTest {
```

```
    @Test
    public void add_new_rental() throws Exception {
        Car car = new Car("Ford Fiesta");
        RentalService service = new RentalService();
        Date rentalBegin = new SimpleDateFormat("dd/MM/yy")
                                .parse("22/12/2013");
        Date rentalEnd = new SimpleDateFormat("dd/MM/yy")
                                .parse("29/12/2013");
        Rental rental =
            service.rentACar("Özcan Acar", car, rentalBegin,
                                    rentalEnd);
        assertTrue(rental.isRented());
    }
}
```

Resim 2.3 de yer alan dizge diyagram uygulamanın hangi modüllerden oluştuğuna dair fikir sahibi olmamızı sağlamıştı. Kod 2.1 de yer alan birim testine göz attığımızda, dizge diyagramında yer alan RentalService biriminin Java sınıfı olarak var olduğunu ve sahip olduğu sorumluluk alanına göre kullanıldığını görmekteyiz.

RentalService sınıfının rentACar() metodu ile bir aracın kiralanma işlemi gerçekleştirilmektedir. Bu metot müşteri ismini, kiralanan aracı, kiralama süresinin başlangıç ve bitiş tarihlerini parametre olarak almaktadır. rentACar() metodu bünyesinde yapılan işlemler kod 2.2 de yer almaktadır.

```
Kod 2.2 - RentalService

package com.kurumsaljava.com.spring;

import java.util.Date;

public class RentalService {

    public Rental rentACar(String customerName, Car car,
                                Date begin, Date end) {

        CustomerRepository customerRepository =
                            new CustomerRepository();
        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }
```

```
        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        RentalRepository rentRepository = new RentalRepository();
        rentRepository.save(rental);
        return rental;
    }
}
```

CustomerRepository sınıfı müşteri verilerinden sorumlu olan sınıftır. Veri tabanında yer alan bir müşteri kaydını bulmak için getCustomerByName() metodunu kullanıyoruz. Eğer getCustomerByName() metodu aradığımız müşteriyi bulamadıysa, new Customer() ile yeni bir müşteri nesnesi oluşturup, save() metodu ile bu müşteriyi veri tabanına ekleyebiliriz.

Bir aracın kiralama işlemi Rental sınıfı ile sembolize edilmektedir. Bu sınıf bünyesinde hangi aracın hangi müşteri tarafından kiralandığı yer almaktadır. setCar() ve setCustomer() metotları ile kiralanan araç ve aracı kiralayan müşteri oluşturulan rental nesnesine yerleştirilir. Akabinde RentalRepository sınıfının sahip olduğu save() metodu ile rental nesnesi veri tabanına kayıtlanır. Bu metot bünyesinde eğer kayıt esnasında bir hata oluşmadı ise, rental nesnesinin sahip olduğu rented değişkenine true değeri atanır. Bu durumda isRented() metodu kiralama işleminin olumlu şekilde tamamlandığını gösterecektir.

Bağımlılıkların Enjekte Edilmesi (Dependency Injection)

Kod 2.2 de görüldüğü gibi RentalService sınıfı bünyesinde bir kiralama işlemini gerçekleştirebilmek için kullanılan başka sınıflar mevcuttur. Örneğin müşteri verilerine ulaşmak için CustomerRepository ve bir rental nesnesini veri tabanına kayıtlamak için RentalRepository sınıfı kullanılmaktadır. Bu iki sınıf RentalService sınıfının bağımlılıkları (dependency) olarak adlandırılır. Kısaca RentalService sınıfı bağımlı olduğu sınıflar olmadan iş göremez.

RentalService sınıfını yakından incelediğimizde, rentACar() metodu bünyesinde new operatörü kullanılarak bağımlı olunan sınıflardan nesneler oluşturulduğu görülmektedir. Java'da new operatörü ile yeni nesneler oluşturulur. Lakin bu şekilde nesne üretmenin beraberinde getirdiği bir dezavantaj vardır. Kod içinde new operatörü kullandığı taktirde, hangi sınıftan bir nesne oluşturulmak

istendiği belirtilmek zorundadır. Bu katı kodlama (hard coding) yapmak anlamına gelmektedir. New her zaman derleme esnasında kullanılan sınıf tipinin sistem tarafından tanınıyor olmasını gerektirir. Ayrıca başka bir sınıf kullanabilmek için kodun değiştirilerek, yeniden derlenmesi gerekmektedir.

Yazılım yaparken karşılaşılan en büyük zorluklardan birisi, bağımlılıkların yönetilmesidir. Kodu değiştirmek zorunda kalmadan, bağımlılıkların yönetimi mümkün olmalıdır. RentalService sınıfında bağımlılıkların katı kodlanması nedeni ile, bağımlılıkların yönetimi çok güçtür. Bağımlılıkların katı kodlanmasına gerek duyulmadan, yönetilebileceği bir mekanizmaya ihtiyaç duyulmaktadır.

New operatörünü kullanmak yerine, RentalService sınıfının ihtiyaç duyduğu nesneleri dışarıdan bu sınıfa verebilseydik nasıl olurdu? Metot parametreleri gibi, sınıfın ihtiyaç duyduğu tüm bağımlılıkları dışarıdan enjekte edebiliriz. Bu yönteme bağımlılıkların enjekte edilmesi ismi verilmektedir. RentalService sınıfı için bağımlılıkların nasıl enjekte edildiği kod 2.3 de yer almaktadır.

```
Kod 2.3 - RentalService

package com.kurumsaljava.com.spring;

import java.util.Date;

public class RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentRepository;
    public RentalService(CustomerRepository customerRepository,
                                RentalRepository rentRepository) {
        this.customerRepository = customerRepository;
        this.rentRepository = rentRepository;
    }

    public Rental rentACar(String customerName, Car car,
                                    Date begin, Date end) {

        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
```

```
        rental.setCar(car);
        rental.setCustomer(customer);
        rentRepository.save(rental);
        return rental;
    }
}
```

Son yapılan değişiklik ile rentAcar() metodunda kullanılan new operatörleri kaldırmış ve bunun yerine bağımlı olunan sınıflardan oluşan sınıf değişkenleri tanımlamış olduk. Yapılan bu değişiklik ile RentalService sınıfına ihtiyaç duyduğu bağımlılıklar enjekte edilebilir hale gelmiştir. Görüldüğü gibi sınıf konstrüktörü üzerinden CustomerRepository ve RentalRepository sınıflarından oluşturulan nesneler RentalService sınıfına enjekte edilmektedir.

Bu değişikliğin ardından kod 2.1 de yer alan RentalServiceTest sınıfının aşağıdaki şekilde yeniden yapılandırılması gerekmektedir.

```
Kod 2.4 - RentalServiceTest

package com.kurumsaljava.com.spring;</code>

import static junit.framework.Assert.assertTrue;
import java.text.SimpleDateFormat;<br/>
import java.util.Date;<br/>
import org.junit.Test;<br/></code>

public class RentalServiceTest {

    @Test
    public void add_new_rental() throws Exception {
        Car car = new Car("Ford Fiesta");
        CustomerRepository customerRepository =
                    new CustomerRepository();
        RentalRepository rentalRepository =
                    new RentalRepository();
        RentalService service =
                    new RentalService(customerRepository,
                                          rentalRepository);
        Date rentalBegin =
                new SimpleDateFormat("dd/MM/yy")
                                          .parse("22/12/2013");
        Date rentalEnd =
                new SimpleDateFormat("dd/MM/yy")
                                          .parse("29/12/2013");
        Rental rental =
                service.rentACar("Özcan Acar", car,
                                     rentalBegin, rentalEnd);
```

```
        assertTrue(rental.isRented());
    }
}
```

Bir sınıfa ihtiyaç duyduğu bağımlılıkların dışarıdan enjekte edilebilmesi için doğal olarak bu bağımlılıkların sınıf dışında oluşturulmaları gerekmektedir. Kod 2.4 de görüldüğü gibi RentalService sınıfının ihtiyaç duyduğu iki bağımlılık RentalServiceTest.add*new*rental() metodunda oluşturulmakta ve bu iki bağımlılık RentalService sınıfının konstrüktörü üzerinden bu sınıftan oluşturulan nesneye enjekte edilmektedir.

RentalService sınıfı için gerek duyulan bağımlılıkların dışarıdan enjekte edilebilir hale gelmesi RentalService sınıfını daha esnek hale getirmiştir. Örneğin CustomerRepository sınıfını genişleten yeni bir alt sınıf oluşturarak, RentalService sınıfına enjekte edebiliriz. Bu yeni sınıf örneğin müşteri verilerini veri tabanından değil, başka bir kaynaktan edinmemizi sağlayabilir. RentalService sınıfını değiştirmeden sisteme böylece yeni bir davranış biçimi eklemek mümkündür. **Bağımlılıkların enjekte edilmesi metodunun ana amaçlarınından bir tanesi budur.**

Spring birçok iş için kullanılabilir, ama en güçlü olduğu saha ve tercih edilme sebeplerinin başında bağımlılıkların yönetimini ve enjekte edilmesini sağlaması gelmektedir. Bağımlılıkların enjekte edilmesi yöntemi ile kod birimleri arasında daha esnek bağ oluşturulur. Birbirini kullanan birimler kod değişikliğine gerek kalmadan değişik türde bağımlılıkları kullanabilecek şekilde yeniden yapılandırılabilir. Bu bir nevi değişik kod birimlerini bir araya getirerek, yazılım sistemi konfigüre etmek gibi bir şeydir.

Spring bağımlılıkların enjekte edilmesi yönteminde interface sınıfların kullanımına ağırlık verir. Eğer bir nesne bağımlı olduğu kod birimlerinin sadece interface sınıflarını tanırsa, Spring bu nesneye ihtiyaç duyduğu interface sınıfların herhangi bir implementasyonunu enjekte edebilir. RentalService sınıfının Spring kullanılarak nasıl yeniden yapılandırılabileceğini bir sonraki bölümde yakından inceleyelim.

Spring İle Bağımlılıkların Enjekte Edilmesi

Bu bölümde araç kiralama servisi uygulamasını Spring kullanarak yeniden yapılandıracağız. Spring bağımlılıkların tanımlanması için bir XML dosyası kullanır. Bu konfigürasyon dosyası Spring sunucusunu oluşturmak için gerekli

tüm meta bilgileri ihtiva eder. Spring sunucusu kavramını bir sonraki bölümde inceleyeceğiz. Şimdilik bunu çalışır durumda olan bir Spring uygulaması olarak düşünebilirsiniz.

Spring 3.0 ile XML dosya kullanma zorunluluğu kalkmıştır. @Configuration anotasyonu kullanılarak Java bazlı Spring konfigürasyonu yapılabilir. Yinede Spring'in temellerini daha iyi anlayabilmek için bu bölümde XML bazlı konfigürasyonu inceleyeceğiz.

Bağımlılıkların Spring tarafından yönetilebilmesi için mevcut kod birimlerinin Spring XML dosyasında tanımlanması gerekmektedir. Spring XML dosyasında tanımlanan her kod birimine Spring bean (Spring nesnesi) ismi verilmektedir. Bundan sonraki bölümlerde Spring bean yanı sıra Spring nesnesi tanımlaması ya da bean tanımlaması terimlerini kullanacağım.

Araç kiralama servisi uygulamasında Spring XML dosyasında tanımlamamız gereken Spring nesneleri RentalServiceImpl, CustomerRepositoryImpl ve RentalRepositoryImpl sınıflarıdır.

```
Kod 2.5 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
                            spring-beans-3.0.xsd">

    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
        <constructor-arg ref="customerRepository" />
        <constructor-arg ref="rentalRepository" />
    </bean>

    <bean id="customerRepository"
            class="com.kurumsaljava.spring.CustomerRepositoryImpl" />
    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl" />
</beans>
```

Herhangi bir Java sınıfı <bean/> XML elementi kullanılarak Spring nesnesi olarak tanımlanabilir. Kod 2.5 de yer alan Spring XML dosyasında rentalService, customerRepository ve rentalRepository isimlerinde üç Spring nesne tanımlaması yer almaktadır.

<bean/> bünyesinde kullanılabilecek element özellikleri (attribute) aşağıda yer almaktadır. Bu element özelliklerinin nasıl kullanıldığını kitabın bu ve diğer bölümlerinde inceleyeceğiz.

- ***class*** - Kullanılan sınıfı tanımlar.
- ***id*** - Oluşturulan Spring nesnesinin tekil tanımlayıcısıdır.
- ***name*** - Bir Spring nesnesine id haricinde birden fazla isim vermek için kullanılır. İsimler arasında virgül kullanılarak birden fazla isim tanımlanabilir. id ve name element özelliklerine sahip olmayan Spring nesne tanımlamaları anonimdir ve doğrudan başka nesnelere enjekte edilemezler.
- ***abstract*** - Nesne tanımlamasını soyutlaştırır. Bu tür Spring nesne tanımlamalarından nesneler oluşturulmaz. Soyut Spring nesne tanımlamaları başka Spring nesnelerin üst tanımlaması olarak kullanılır.
- ***parent*** - Kullanılacak üst soyut Spring nesne tanımlamasını belirlemek için kullanılır.
- ***scope*** - Varsayılan scope singleton yani tekil scopodur. Bu Spring sunucusundan bir nesne talep ettiğimizde singleton olan aynı nesneyi edindiğimiz anlamına gelmektedir. prototype scope kullanıldığında Spring her isteğe yeni bir nesne ile cevap verir.
- ***constructor-arg*** - Sınıf konstrüktörü aracılığı ile bağımlılıklar enjekte edilmek istendiğinde kullanılır.
- ***property*** - Enjeksiyon yapılacak sınıf değişkenini tanımlar.
- ***init-method*** - Nesneler enjekte edildikten sonra koşturulacak sınıf metodunu tanımlar. Bu metodun parametresiz olması gerekmektedir.
- ***destroy-method*** - Spring sunucusu son bulduğunda sunucu bünyesinde bulunan nesneleri sonlandırmak için kullanılan metotları tanımlar. Bu metodun parametresiz olması gerekmektedir. Sadece Spring sunucusu tarafından kontrol edilen nesneler üzerinde tanımlanabilir.
- ***factory-bean*** - Bu nesneyi oluşturabilecek fabrika sınıfını (Factory Bean) tanımlar. factory-bean kullanıldığı taktirde sınıf değişkenleri üzerinden enjeksiyon yapılmamalıdır.
- ***factory-method*** - Bu nesneyi oluşturmak için kullanılacak fabrika metodunu tanımlar.

Kod 2.3 de yer alan RentalService sınıfına tekrar göz attığımızda, bu sınıfın ve bağımlılık duyduğu diğer sınıfların somut sınıflar olduğunu görmekteyiz. Genel olarak bağımlılıkların soyut sınıflar yönünde olmasında fayda vardır, çünkü sadece bu durumda sınıfa istedigimiz bir implementasyon nesnesini enjekte

edebiliriz. Bunun yanı sıra Spring bean olarak tanımladığımız sınıfın da bir interface sınıf implementasyonu olması faydalıdır. Bu şekilde Spring birden fazla implementasyonu yönetebilir. Nitekim kod 2.5 de Spring bean tanımlamalarında kullandığımız sınıflar şu interface sınıfları implemente etmektedirler:

```
public interface RentalService{
    public Rental rentACar(String customerName,
             Car car, Date begin, Date end);
}

public interface RentalRepository{
    public void save(Rental rental);
}

public interface CustomerRepository{
    public Customer getCustomerByName(String name);
    public void save(Customer customer);
}
```

Spring bir nesneye ihtiyaç duyduğu bağımlılıkları sahip olduğu sınıf değişkenleri, set metotları ya da sınıf konstrüktörleri üzerinden enjekte edebilir. Kod 2.5 de yer alan örnekte rentalService ismini taşıyan Spring nesnesi bağımlılıkları constructor-arg kullanılarak sınıf konstrüktörleri üzerinden enjekte edilmektedir. RentalServiceImpl sınıfının sahip olduğu sınıf değişkenleri ve bu değişkenlerin sınıf kontrüktörü üzerinden nasıl enjekte edildiği kod 2.6 da yer almaktadır.

```
Kod 2.6 - RentalServiceImpl

package com.kurumsaljava.spring;
import java.util.Date;

public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;
    public RentalServiceImpl(CustomerRepository customerRepository,
        RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
```

```
    public Rental rentACar(String customerName, Car car, Date
                                                begin, Date end) {
        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }
        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }
}
```

Spring'in bir bağımlılığı enjekte edebilmesi için hangi Spring nesnesini nereye enjekte edeceğini bilmesi gerekir. Nereye sorusuna constructor-arg ile cevap verdik. Hangi sorusuna ise constructor-arg elementinde kullanılan ref ile cevap verebiliriz. Ref referans anlamına gelmektedir ve enjekte edilecek bağımlılığa işaret eder. rentalService örneğinde customerRepository ve rentalRepository isimlerini taşıyan iki Spring nesnesi sınıf konstrüktörü üzerinden rentalService nesnesine enjekte edilmektedir. Spring'in bu iki nesneyi rentalService nesnesine enjekte edebilmesi için customerRepository ve rentalRepository isimlerini taşıyan Spring nesnelerin XML dosyasında tanımlanmış olması gerekmektedir. Spring tanımlanan Spring nesne isimlerini kullanarak gerekli olan bağımlılıkları enjekte eder.

RentalServiceImpl sınıfının konstrüktöründe kullanılan parametrelerin belli bir sırası vardır. İlk parametre CustomerRepository sınıfından bir nesne, ikinci parametre RentalRepository sınıfından bir nesnedir. RentalServiceImpl sınıfından sahip olduğu bu iki parametreli sınıf konstrüktörü üzerinden bir nesne oluştururken, parametrelerin hangi sıra ile konstrüktöre verildiği önemlidir. Spring XML dosyasında bu parametre sırası nasıl belirtilir? Kod 2.5 de yer alan XML dosyasında rentalService için böyle bir sıra tanımlaması yapmadık. Spring ilk etapta böyle bir sıralamaya ihtiyaç duymamaktadır. Spring tanımlanan Spring nesnelerin hangi Java sınıfından oluşturulduğunu (class aracılığı ile) bildiği için bağımlılıkları kontrüktöre enjekte ederken doğru nesneyi doğru sırada enjekte eder. Bir konstrüktör aynı Java sınıfından olan birden fazla parametre almadığı sürece bir sorun oluşmaz. Eğer sınıf konstrüktörü aynı Java sınıfından olan birden fazla parametreye sahip ise, Spring'in bağımlılıkları enjekte ederken kafası karışabilir. Bunu engellemek için

aşağıdaki şekilde parametre sırası belirtilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository" index="0"/>
    <constructor-arg ref="rentalRepository" index="1"/>
</bean>
```

Spring 3.0 dan itibaren konstrüktör parametresinin ismini kullanarak enjeksiyon yapmak mümkündür. Spring'in Java byte kodunda parametre isimlerini bulabilmesi için byte kodun debug flag kullanılarak derlenmiş olması gerekmektedir. Aksi taktirde değişken isimleri byte kod içinde olmayacağından, bu yöntem kullanılarak bağımlılıklar enjekte edilemez.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg name="customerRepository"
                                ref="customerRepository"/>
    <constructor-arg name="rentalRepository"
                                ref="rentalRepository"/>
</bean>
```

JDK bünyesinde yer alan @ConstructorProperties anotasyonu ile konstrüktör parametrelerini kod içinde şu şekilde işaretleyebiliriz:

```
@ConstructorProperties({"rentalRepository", "custromerRepository"})
public RentalService(RentalRepository rentalRepository,
                CustomerRepository customerRepository ) {
    this.rentalRepository = rentalRepository;
    this.customerRepository = customerRepository;
}
```

Spring ile oluşturduğumuz bu yeni yapıyı nasıl kullanabiliriz? Bunun cevabı kod 2.7 de yer almaktadır.

```
Kod 2.7 - Main

package com.kurumsaljava.spring;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
                    ClassPathXmlApplicationContext;
public class Main {
```

```
    public static void main(String[] args) throws Exception {

        ApplicationContext ctx =
            new ClassPathXmlApplicationContext("
                                applicationContext.xml");
        RentalService rentalService =
                (RentalService) ctx.getBean("rentalService");
        Rental rental = rentalService.
                rentACar("Özcan Acar", new Car("Fiesta"),
                        getRentalBegin(), getRentalEnd());
        System.out.println("Rental status: "
                            + rental.isRented());
    }

    private static Date getRentalEnd()
                throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                parse("29/12/2013");
    }

    private static Date getRentalBegin()
                    throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                parse("22/12/2013");
    }
}
```

Idref Kullanımı

Idref ile konfigürasyon dosyasında tanımlı olan herhangi bir Spring nesnesinin ismini herhangi bir sınıf değişkenine enjekte edebiliriz.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName">
        <idref bean="customerRepository" />
    </property>
</bean>
```

Bu örnekte targetName isimli değişkene bir Spring nesnesinin ismi olan customerRepository enjekte edilmemektedir. Burada targetName isimli değişkene atanan bir String değerdir. Bu değer customerRepository String nesnesidir. Bu tanımlamayı şu şekilde de yapabilirdik:

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName" value="customerRepository"/>
</bean>
```

Idref kullanıldığı taktirde Spring uygulama çalışmaya başlamadan önce customerRepository ismini taşıyan bir Spring nesnesi tanımlaması olup, olmadığı kontrol eder. Eğer böyle bir tanımlama yoksa, Spring uygulamayı durdurur ve hata bildiriminde bulunur. Verdiğim ikinci örnekte targetName değişkenine doğrudan customerRepository değeri atandığında, Spring bu değeri kontrol etmez. Eğer yanlış bir değer kullanıldıysa, bu hata uygulama çalışırken gün ışığına çıkacaktır. Bu sebepten dolayı mevcut Spring nesne isimleri idref kullanılarak değişkenlere enjekte edilmelidir, çünkü sadece bu şekilde bu isimlerin hatalı olup, olmadıkları hemen anlaşılabilir.

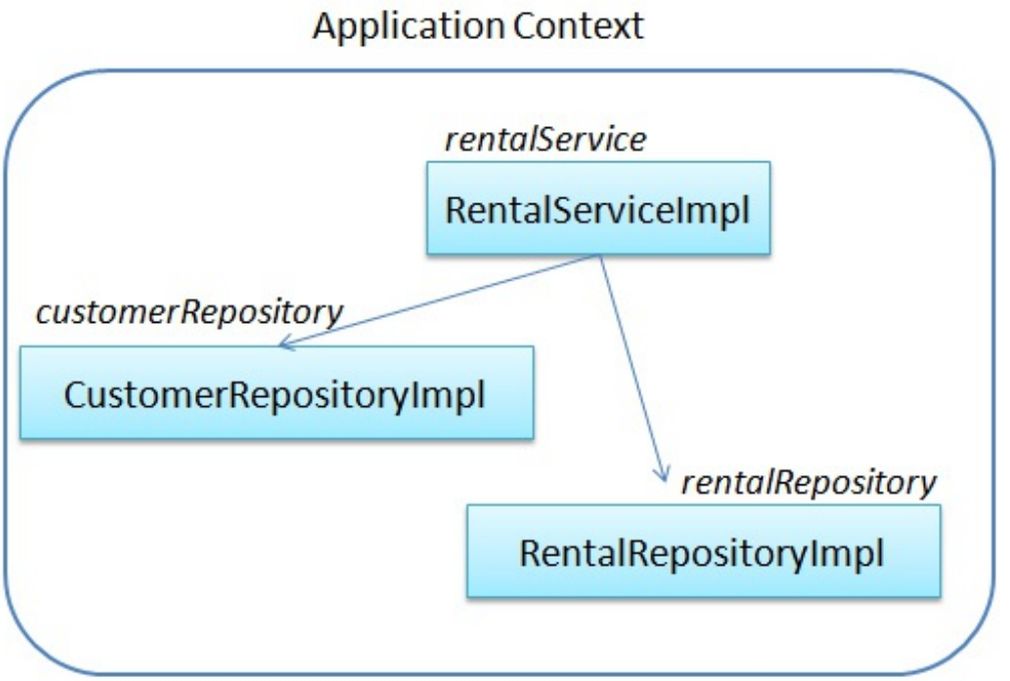
idref elementinin local element özelliği kullanıldığı taktirde, Spring bu ismi taşıyan Spring nesnesinin idref'in kullanıldığı konfigürasyon dosyasında tanımlanıp, tanımlanmadığını kontrol eder. Spring böyle bir nesne tanımlaması bulamaması durumunda uygulamayı durdurur ve hata bildiriminde bulunur.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName">
        <idref local="customerRepository" />
    </property>
</bean>
```

Spring Sunucusu ve BeanFactory

Spring uygulamasının çalışabilir hale gelmesi için Spring konfigürasyonunun yer aldığı XML dosyasının hafızaya yüklenmesi gerekmektedir. Bu amaçla org.springframework.context.ApplicationContext sınıfı ve türevleri kullanılır. ApplicationContext org.springframework.beans.factory.BeanFactory interface sınıfını genişletir. Spring sunucusunu temsil eden en üst sınıf BeanFactory sınıfıdır. Bu sınıf Spring sunucusunda yer alan nesnelerin konfigürasyonunu ve bu nesnelere erişimi tanımlar. BeanFactory Spring'in temel konfigürasyon yapısıdır.

ApplicationContext türevlerinden bir tanesi ClassPathXmlApplicationContext sınıfıdır. ClassPathXmlApplicationContext ile Java classpath içinde bulunan bir XML dosyası yüklenerek, Spring uygulaması çalışır hale getirilir.



Resim 2.5

Kod 2.7.1 de yer alan satır Spring sunucusunu oluşturur. Sunucuyu oluşturmak için gerekli tüm meta bilgiler applicationContext.xml ismini taşıyan XML dosyasında yer almaktadır.

```
Kod 2.7.1 - Main

ApplicationContext ctx =
        new ClassPathXmlApplicationContext("
                        applicationContext.xml");
```

ClassPathXmlApplicationContext sınıfı kullanılarak Spring XML dosyasının yüklenmesi hafızada Application Context ismi verilen bir yapıyı oluşturur. Bunu Spring uygulamasının hafızadaki yüklenmiş hali olarak düşünebiliriz. ApplicationContext bu yapıyı temsil eden interface sınıftır. Bu yapıya Spring Container, yani Spring sunucusu ismi verilmektedir. Spring, XML dosyasında tanımlanmış tüm Spring nesneleri tarayarak, önce hiç bağımlılığı olmayan Spring nesne tanımlamalarından nesneler oluşturur. Daha sonra bu nesneleri bağımlılık duyan diğer nesnelere enjekte eder. Bu işlem sonunda her Spring nesne tanımlaması için bir nesne oluşturularak, hafızada yer alan Application Context bünyesinde konuşlandırılmış olur.

Hafızada yer alan Application Context bünyesindeki rentalService isimli Spring nesnesine erişmek için ctx.getBean("rentalService") şeklinde bir çağrı yapmamız

yeterli olacaktır. getBean() metodu bize istediğimiz Spring nesnesini sanki new yapmışcasına geri verir. Artık new operatörünü kullanan biz değil, Spring'dir. Spring Java'nın Reflection özelliğinden faydalanarak nesneleri oluşturur.

Spring 3 ile oluşturulan yeni getBean() metodu ile cast yapmak zorunda kalmadan şu şekilde bir rentalService nesnesi edinebiliriz:

```
RentalService rentalService =
            ctx.getBean("rentalService", RentalService.class);
```

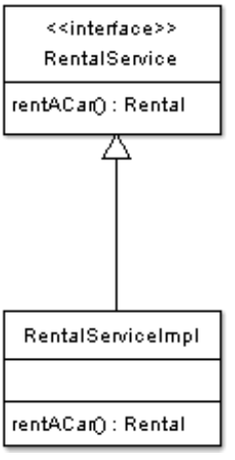
Eğer kullandığımız interface sınıfın sadece bir implementasyon sınıfını Spring nesnesi olarak tanımladıysak, aşağıdaki şekilde Spring nesnesinin ismini kullanmadan bir rentalService nesnesi edinebiliriz:

```
RentalService rentalService =
            ctx.getBean(RentalService.class);
```

Kod 2.5 de yer alan XML konfigürasyon dosyasına tekrar göz attığımızda rentalService isimli Spring nesneyi tanımlamak için RentalServiceImpl sınıfını kullandığımız görülmekte. Nasıl oluyor da kod 2.7 de yer alan main() metodunda

```
RentalService rentalService =
            (RentalService) ctx.getBean("rentalService");
```

şeklinde bir tanımlama ile rentalService nesnesini kullanıyoruz? RentalService tanımladığımız interface sınıftır.



Resim 2.6

```
public interface RentalService {
    public Rental rentACar(String customerName, Car car,
```

```
                                        Date begin, Date end);
}
```

RentalService soyut bir interface sınıfıdır. Kod içinde RentalService interface sınıfını kullanarak somut sınıflara olan bağımlılığımızı ortadan kaldırmış oluyoruz. RentalService interface sınıfını implemente eden yeni bir sınıf oluşturarak, kod 2.7 de yer alan Main sınıfını değiştirmek zorundan kalmadan, bu oluşturduğumuz yeni implementasyon sınıfını kullanabiliriz. Bunun için yapmamız gereken tek şey, Spring XML dosyasında rentalService ismini taşıyan Spring nesnesinin kullandığı sınıfı değiştirmek olacaktır.

Tekrar kod 2.2 ye göz attığımızda, Spring'in bizim için üstlendiği görevi net olarak görmek mümkündür. Spring öncesi her bağımlılığı new ile kendimiz oluştururken, Spring ile bunu yapma zorunluluğu ortadan kalkmıştır. Ayrıca oluşturduğumuz yeni uygulamanın bağımlılıkların tersine çevrilmesi tasarım prensibine DIP - Dependency Inversion Principle uygun hale geldiğini görmekteyiz. Bu tasarım prensibi uygulamanın meydana gelebilecek değişikliklere karşı daha dayanıklı hale gelmesini sağlamaktadır. Spring ayrıca interface sınıfları kullanmayı teşvik ederek, uygulamayı değiştirmeden uygulamaya yeni davranış biçimleri eklemeyi mümkün kılmaktadır. Daha fazlası can sağlığı demeyin. Spring'in uygulama geliştirirken bizim için üstlenebileceği daha birçok şey var. İlerleyen bölümlerde Spring'in bizim için yapabileceklerini daha yakından inceleyeceğiz. Kısaca Spring'in var olma nedenini tanımlamak istersek, "Spring yazılımcının hayatını kolaylaştırmak için vardır" dememiz yanlış olmaz.

Spring Nesne İsimlendirme

Application Context bünyesinde yer alan her Spring nesnesinin bir ya da birden fazla ismi vardır. Spring sunucusu bünyesinde bu isimlerin tekil olması gerekmektedir. Aynı ismi iki değişik Spring nesnesi paylaşamaz.

Şimdiye kadar kullandığımız örneklerde id element özelliği aracılığı ile oluşturduğumuz Spring nesne tanımlamalarına isimler verdik. Tanımladığımız bu isimler aracılığı ile Spring nesnelerine erişmemiz mümkün oldu.

Id element özelliği bünyesinde sadece bir isim barındırabilir, yani her nesne tanımlamasına id element özelliği ile sadece bir isim atanabilir. Bunun yanı sıra Spring 3.1 öncesi id element özelliği bünyesinde / ve : gibi işaretler kullanmak mümkün değildi. Spring 3.1 ile aşağıdaki şekilde bir nesne tanımlaması

mümkün olmuştur.

```
<bean id="clio/myclio" .../>

Car clio = ctx.getBean("clio/myclio", Car.class);
```

Bu değişiklik ne yazık ki bir Spring nesne tanımlamasına birden fazla ismi atayamama sorununu çözmemektedir. Bu sorunu çözmek için name element özelliği kullanılabilir. Örneğin yukarıda tanımlamış olduğumuz clio nesnesine aynı zamanda myclio ya da yourclio isimlerini atamak isteseydik, o zaman şu şekilde bir bean tanımlaması yapardık:

```
<bean id="clio" name="myclio, yourclio" .../>
```

Bir Spring nesnesi tanımlarken id ya da name element özelliklerini kullanma zorunluluğu bulunmamaktadır. Bu element özellikleri ile Spring nesnesine bir isim atanmadığı taktirde, Spring sunucusu bu nesneye otomatik olarak bir isim atar. İsmi olmayan Spring nesneleri anonimdir ve isimleri olmadığı için ref element özelliği kullanılarak diğer nesnelere enjekte edilemezler.

Mevcut Spring nesnelerine yeni isimler atamak için kullanılabilecek diğer bir element <alias/> elementidir. Örneğin

```
<alias name="birIsim" alias="baskaBirIsim"/>
```

ile birIsim ismini taşıyan Spring nesnesine baskaBirIsim ismini kullanarak ta erişmek mümkün olmaktadır.

Bağımlılıkları Enjekte Etme Türleri

RentalServiceImpl sınıfının ihtiyaç duyduğu bağımlılıkları sınıf konstrüktörü üzerinden constructor-arg elementi ile enjekte etmiştik. Bu bağımlılıkları enjekte etmek için kullanabileceğimiz yöntemlerden bir tanesidir. Bunun yanı sıra bağımlılıklar property elementi ile de enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository" ref="customerRepository" />
    <property name="rentalRepository" ref="rentalRepository" />
</bean>
```

Property elementi sınıf bünyesindeki bir sınıf değişkenine işaret etmektedir. Hangi sınıf değişkenine bağımlılığın enjekte edilmesi gerektiğini name element özelliği ile tanımlıyoruz. Kullandığımız örnekte (bakınız kod 2.6) sınıf değişkeninin ismi customerRepository ya da rentalRepository'dir. Eğer sınıf değişkenleri bu isim ile tanımlanmadı ise, Spring istenilen bağımlılıkları enjekte edemez. Hangi bağımlılığın sınıf değişkenine enjekte edilmesi gerektiğini ref element özelliği ile tanımlıyoruz. Kod 2.5 e tekrar göz attığımızda customerRepository ve rentalRepository ismini taşıyan iki Spring nesnesi tanımlamasının mevcut olduğunu görmekteyiz. Bu örnekte sınıf değişkenleri ve Spring nesneleri aynı ismi taşımaktadır.

Bağımlılıkların property elementi kullanılarak enjekte edilebilmesi için sınıf değişkenlerinin set() metotlarına sahip olması gerekmektedir. Bunun yanı sıra sınıf parametresiz bir sınıf konstrüktörüne ihtiyaç duymaktadır.

```
Kod 2.8 - RentalServiceImpl

package com.kurumsaljava.spring;
import java.util.Date;
public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;

    public RentalServiceImpl() {

    }

    public RentalServiceImpl(
        CustomerRepository customerRepository,
        RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Rental rentACar(String customerName, Car car,
                           Date begin, Date end) {
        Customer customer =
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }
```

```
        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }

    public CustomerRepository getCustomerRepository() {
        return customerRepository;
    }

    public void setCustomerRepository(CustomerRepository
                                      customerRepository) {
        this.customerRepository = customerRepository;
    }

    public RentalRepository getRentalRepository() {
        return rentalRepository;
    }

    public void setRentalRepository(RentalRepository
                                    rentalRepository) {
        this.rentalRepository = rentalRepository;
    }
}
```

Eğer bağımlılıkların enjekte edilmesi için property elementini kullandıysak, Spring bağımlılıkların enjekte edileceği sınıftan parametresiz sınıf konstrüktörünü kullanarak yeni bir nesne oluşturur. Akabinde Spring setCustomerRepository() ve setRentalRepository() metotlarını kullanarak tanımlanmış olan bağımlılıkları enjekte eder. Spring'in bir RentalServiceImpl nesnesini nasıl oluşturduğunu anlamak için bir sonraki kod bloğuna göz atalım.

```
RentalServiceImpl service = new RentalServiceImpl();
CustomerRepositoryImpl customerRepository =
                    new CustomerRepositoryImpl();
RentalRepositoryImpl rentalRepository =
                    new RentalRepositoryImpl();
service.setCustomerRepository(customerRepository);
service.setRentalRepository(rentalRepository);
```

Eğer kendimiz bir RentalServiceImpl nesnesi oluşturmak isteseydik, bir önceki kod bloğundaki gibi işlem yapmamız gerekirdi. Oysaki Spring aracılığı ile bir RentalServiceImpl nesnesi edinmek için yapmamız gereken tek işlem

```
RentalService rentalService =
        (RentalService) ctx.getBean("rentalService");
```

şeklindedir.

Bağımlılıkların enjekte edilmesi için hangi yöntemi kullanmalıyım sorusu aklınıza gelmiş olabilir. Seçiminizi sadece bir yönde yapmak zorunda değilsiniz. Her iki yöntemi de birlikte kullanabilirsiniz. Bir sonraki kod bloğunda hem constructor-arg hem de property elementleri beraber kullanılmaktadır.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository"/>
    <property name="rentalRepository" ref="rentalRepository" />
</bean>
```

Spring her iki enjekte metodu birlikte kullanıldığında, bir sonraki kod bloğundaki işlemleri gerçekleştirir.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
RentalServiceImpl service =
                new RentalServiceImpl(customerRepository);
RentalRepositoryImpl rentalRepository =
                new RentalRepositoryImpl();
service.setRentalRepository(rentalRepository);
```

Eğer sonradan değiştirilemeyen, sabit (immutable) nesneler oluşturmak istiyorsanız, seçiminiz constructor-arg yönünde olmalıdır. Bir sınıftan sabit bir nesne oluşturmak için set metotlarının kaldırılması ve sadece bağımlıkları parametre olarak alan bir sınıf konstrüktörü kullanılması gerekir. Örneğin RentalServiceImpl sınıfından sabit bir nesne oluşturmak için kod 2.9 da yer alan implementasyon kullanılabilir.

```
Kod 2.9 - RentalServiceImpl

package com.kurumsaljava.spring;
import java.util.Date;
public class RentalServiceImpl implements RentalService {

    private final CustomerRepository customerRepository;
    private final RentalRepository rentalRepository;

    public RentalServiceImpl(
```

```
            CustomerRepository customerRepository,
            RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Rental rentACar(String customerName, Car car,
                           Date begin, Date end) {
        Customer customer =
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }
}
```

Listelerin Enjekte Edilmesi

Spring nesneler yanı sıra konfigürasyonu tanımlanan listeleri de enjekte edebilir. Örneğin RentalServiceImpl sınıfına kiralanabilecek araç listesini enjekte etmek isteseydik, bir sonraki kod bloğundaki gibi araç listesini oluştururduk.

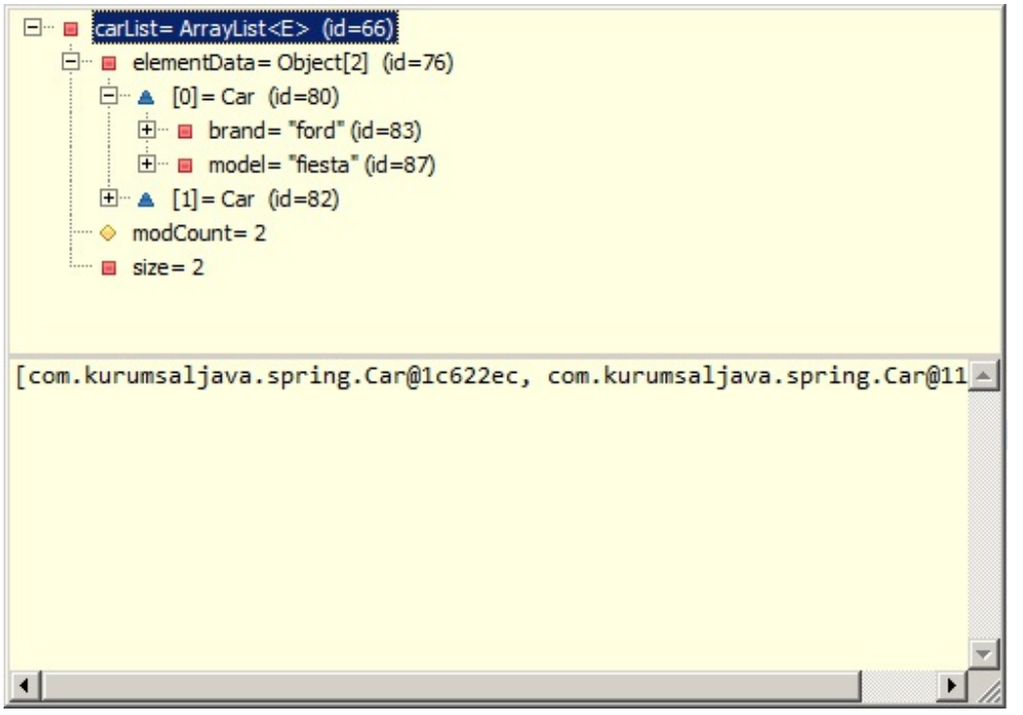
```
bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository" />
    <constructor-arg ref="rentalRepository" />
    <property name="carList">
        <list>
            <ref bean="fiesta" />
            <ref bean="clio" />
        </list>
    </property>
</bean>
<bean id="fiesta" class="com.kurumsaljava.spring.Car">
    <constructor-arg name="brand" value="ford" />
    <constructor-arg name="model" value="fiesta" />
```

```
</bean>
<bean id="clio" class="com.kurumsaljava.spring.Car">
    <constructor-arg name="brand" value="renault" />
    <constructor-arg name="model" value="clio" />
</bean>
```

Araç listesinin RentalServiceImpl sınıfından oluşturulan bir nesneye enjekte edilebilmesi için RentalService bünyesinde bir sonraki kod bloğunda görüldüğü gibi List veri tipinde bir listenin tanımlanması gerekmektedir.

```
private List<Car> carList;
public List<Car> getCarList() {
    return carList;
}

public void setCarList(List<Car> carList) {
    this.carList = carList;
}
```

Main sınıfını Eclipse Debugger ile koşturduğumuzda, Spring'in XML dosyasında tanımlandığı şekilde iki Car nesnesini oluşturarak, RentalServiceImpl sınıfında yer alan carList listesini eklediğini görmekteyiz.



Resim 2.7

Arka planda Spring'in araç listeni oluşturmak için yaptığı işlemler bir sonraki kod bloğunda yer almaktadır.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
RentalRepositoryImpl rentalRepository =
                new RentalRepositoryImpl();
RentalServiceImpl service =
            new RentalServiceImpl(customerRepository,
                                  rentalRepository);
Car fiesta = new Car("ford", "fiesta");
Car clio = new Car("renault", "clio");
List<Car> carList = new ArrayList<Car>();
carList.add(fiesta);
carList.add(clio);
service.setCarList(carList);
```

Map tipi bir koleksiyon aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carMap">
        <map>
            <entry key="fiesta">
                <ref bean="fiesta" />
            </entry>
            <entry key="clio">
                <ref bean="clio" />
            </entry>
       </map>
    </property>
</bean>
```

Set tipi bir koleksiyon aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carSet">
        <set>
            <ref bean="fiesta" />
            <ref bean="clio" />
        </set>
    </property>
</bean>
```

java.util.Properties tarzı bir özellik listesi aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carProperty">
```

```
        <props>
            <prop key="ford"> fiesta </prop>
            <prop key="renault"> clio </prop>
        </props>
    </property>
</bean>
```

Basit Değerlerin Enjekte Edilmesi

Enjekte edilmek istenen değerler int ya da String gibi basit değerler olabilir. property elementi kullanılarak bu tür enjeksiyonları gerçekleştirmek mümkündür. Örneğin CustomerRepositoryImpl sınıfında database ismini taşıyan bir String değişken olsaydı, bu değişkene aşağıdaki şekilde bir değer enjeksiyonu yapabilirdik.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="database" value="oracle" />
</bean>
```

Yukarıda yer alan örnekte CustomerRepositoryImpl sınıfının database isimli sınıf değişkenine oracle değeri enjekte edilmektedir. Bunun sağlanabilmesi için CustomerRepository isimli sınıfın database isimli bir değişkene ve bu değişkene değer atanmasında kullanılacak setDatabase() metoduna sahip olması gerekmektedir. Spring'in customerRepository isimli bir nesneyi oluşturmak için yaptığı işlem aşağıda yer almaktadır.

```
CustomerRepositoryImpl customerRepository =
                    new CustomerRepositoryImpl();
customerRepository.setDatabase("oracle");
```

Value elementini kullanarak ta değişkenlere değerler enjekte edilebilir:

```
<property name="database"><value>oracle</value></property>
```

ya da

```
<constructor-arg type=int><value>10000</value></constructor-arg>
```

Null ya da Boş String Değerinin Enjekte Edilmesi

property elementinin value element özelliği boş bırakıldığı taktirde, bu değişkene sıfır uzunluğunda bir String nesnesinin enjekte edildiği anlamına gelmektedir.

```
<bean id="customerRepository"
            class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="database" value="" />
</bean>
```

Eger bir değişkene null degerini atamak istiyorsak, <null/> elementini kullanabiliriz.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="database"><null/></property>
</bean>
```

Nesne Oluşturma Sırasının Tanımlanması

Normal şartlar altında iki nesne arasındaki bağımlılık ref elementi kullanıldığı taktirde nesnelerin oluşturulma sırası tanımlanmış olur. Spring bu durumda ilk önce enjekte edilecek nesneyi oluşturur, daha sonra bu nesnenin enjekte edileceği diğer nesneyi. Bunun yanı sıra depends-on elementi ile de nesne oluşturma sırası tayin edilebilir. Aşağıda yer alan örnekte Spring customerRepository nesnesini oluşturmadan önce, dbManager isimli nesneyi oluşturur. Bu genel olarak bir nesnenin sebep olduğu yan etkilere bağımlı olunduğu durumlarda kullanılan bir yöntemdir.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl"
                depends-on="dbManager"/>
<bean id="dbManager"
        class="com.kurumsaljava.spring.DbManager" />
```

Depends-on bünyesinde birden fazla bean ismi tanımlaması şu şekilde yapılabilir:

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl"
    depends-on="dbManager, txManager, myBean"/>
```

Uygulama son bulduğunda tanımlanan bu sıranın tam tersine göre nesneler

imha edilir.

Otomatik Veri Tipi Dönüşümü

Basit değerlerin enjeksiyonu için property elementinin value özelliği ile nasıl beraber kullanıldığını bir önceki bölümde gördük. Value özelliği içinde tanımlanan değer yapı itibari ile bir String nesnesidir. Eğer bir int veri tipine sahip bir değişkene değer enjekte etmek istersek, value özelliğinde yer alan bu değer Spring tarafından otomatik olarak bir int değerine dönüştürülür. Aşağıda yer alan kod örneğinde int veri tipinde olan port değişkenine 1234 değeri enjekte edilmektedir. Bu değer otomatik olarak Spring tarafından 1234 rakamına dönüştürülerek, port isimli int değişkenine enjekte edilmektedir.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="port" value="1234" />
</bean>
```

Float, double gibi diğer basit veri tipleri için de Spring tarafından otomatik veri tipi dönüşümü sağlanır.

Tekil Nesneler ve Bean Scope

Çoğu yazılımcı tarafından tasarım şablonu olarak görülmese de, tekillik singleton tasarım şablonu ismini taşıyan bir tasarım şablonu mevcuttur. Tekillik tasarım şablonu ile bir sınıftan sadece bir nesne oluşturulması amaçlanır.

Spring için varsayılan nesne oluşturma ayarı tekil nesnedir. Bunun ispatı için bir sonraki kod bloğunu inceleyelim.

```
System.out.println((RentalService) ctx.getBean("rentalService"));
System.out.println((RentalService) ctx.getBean("rentalService"));
```

Ekra çıktısı:

```
com.kurumsaljava.spring.RentalServiceImpl@18c908b
com.kurumsaljava.spring.RentalServiceImpl@18c908b
```

getBean() metodu ile rentalService nesnesini edindiğimizde, sahip oldukları

adres alanlarının (@18c908b) aynı olduğunu görmekteyiz. Bu Spring'in rentalService ismini taşıyan nesneyi hafızada sadece bir kez tuttuğunun kanıtıdır. Eğer getBean() metodu üzerinden her defasında yeni bir rentalService nesnesi edinmek istiyorsak, rentalService'in bean tanımlamasını bir sonraki kodda yer aldığı şekilde değiştirmemiz gerekmektedir.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
customerRepository.setDatabase("oracle");

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
                        scope="prototype">
    <constructor-arg ref="customerRepository" />
    <constructor-arg ref="rentalRepository" />
</bean>
```

Tekil nesne oluşturmayı önleyen scope="prototype" tanımlamasıdır. scope ile nesnenin ömrü tanımlanır. scope bünyesinde kullanılabilecek değerler şunlardır:

- ***singleton***: Tekil nesne tanımlamak için kullanılır. Spring tarafından varsayılan scope singletondır.
- ***prototype***: Spring sunucusu tarafından her talep için yeni bir nesne oluşturulur.
- ***request***: Web uygulamalarında bir istek (http request) boyunca geçerli olan nesne oluşturulmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***session***: Web uygulamalarında kullanıcı oturumu (http session) boyunca geçerli olan nesne oluşturmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***globalSession***: Portlet uygulamalarında tüm uygulama parçaları için geçerli olan kullanıcı oturumunda (global http session) kullanılmak üzere nesne oluşturmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***custom***: Yazılımcı tarafından oluşturulur. Spring sunucusu yazılımcının tayin ettiği şekilde nesne oluşturma sürecini yönlendirir.

Scope Bazlı Nesnelerin Enjeksiyonu

Scope bazlı nesnelerin bazı şartlar altında singleton ya da prototype olan

nesnelere enjeksiyonu sorun oluşturabilir. Aşağıda yer alan örnekte session scope olan userDetails nesnesi singleton scope olan rentalService nesnesine enjekte edilmektedir. Burada nasıl bir sorun mevcuttur?

```
<bean id="userDetails"
    class="com.kurumsaljava.spring.UserDetails"
        scope="session"/>

<bean id="rentalService"
    class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="userDetails" ref="userDetails"/>
</bean>
```

Singleton olan nesneler Application Context oluştuğunda Spring sunucusu tarafından sadece bir kere yapılandırılırlar. Yukarıda yer alan örnekte singleton olan rentalService nesnesine session scope sahibi userDetails enjekte edilmektedir. rentalService nesnesi oluşturulduğu esnada Application Context bünyesinde userDetails isminde bir nesne olamaz, çünkü böyle bir nesne kullanıcının bir web oturumu (HttpSession) açmasıyla var olabilir. Bu yüzden singleton olan rentalService nesnesine session scope olan bir nesne enjekte edilmesi mümkün değildir. Kısaca burada uygulamanın ilerleyen bir safhasında oluşturulacak bir nesne var oluşunun çok öncesinde mevcut bir nesneye enjekte edilmeye çalışılmaktadır ki bu mümkün değildir.

Bu sorunu userDetails nesnesine vekillik edecek bir nesne oluşturup, bu vekil nesneyi rentalService nesnesine enjekte ederek çözebiliriz. Vekil nesne rentalService tarafından kullanıldıgında, vekil olduğu nesneyi içinde buludugu scopedan alarak, rentalService nesnesine verecektir.

Bu gibi durumlarda kullanılmak üzere vekil nesne oluşturmak için aop isim alanında yer alan scoped-proxy elementi kullanılabilir.

```
<bean id="userDetails"
    class="com.kurumsaljava.spring.UserDetails"
            scope="session">
    <aop:scoped-proxy/>
</bean>
```

Aop:scoped-proxy elementi kullanıldığında Spring sunucusu CGLIB kütüphanesi yardımıyla bir vekil nesne oluşturur. userDetails nesnesinin enjekte edildigi her nesneye bu vekil nesne enjekte edilmiş olur, çünkü vekil nesne userDetails nesnesinin sahip olduğu sınıfı genişletmektedir. Bu vekil nesne

userDetails nesnesine yöneltilen tüm istekleri karşılar. Kullanılan scope türüne göre vekil nesne vekil olduğu nesneyi ait oldugu scopedan alarak, kullanıcı nesnenin geçerli nesne üzerinde işlem yapmış olmasını sağlar.

Scoped-proxy aşağıdaki şekilde tanımlandığında Spring sunucusu CGLIB yerine JDK interface bazlı dinamik vekil oluşturma mekanizmasını kullanır. Bu durumda mutlaka vekil olunan nesnenin bir interface sınıfı implemente etmiş olması gerekmektedir. Sadece bu durumda vekil nesneler oluşturulabilir.

```
<aop:scoped-proxy proxy-target-class="false"/>
```

Tanımlanabilir (Custom) Bean Scope

Spring tarafından kullanılan bean scope mekanizması genişletilebilir yapıdadır. Singleton ve prototype bean scopelar harici mevcut bean scopeları değiştirebilir ya da kendi bean scopelarımızı oluşturabiliriz. Yeni bir bean scope oluşturmak için org.springframework.beans.factory.config.Scope interface sınıfını implemente eden bir altsınıf oluşturmamız gerekmektedir. Yeni bir scope oluşturma işlemini bir örnek üzerinde yakından inceleyelim.

Kod 2.9.1 de ThreadScope ismini taşıyan yeni bir scope sınıfı yer almaktadır. Bean tanımlaması yaparken scope=thread kullanılarak oluşturulan nesnelerin ömürlerini bir threadin ömrü ile sınırlı tutmak istiyoruz. Bu yeni scope tanımlaması kullanıldığı taktirde oluşturulan nesneler kullanılan thread hayatta olduğu sürece kullanımda olacaktır. Threadin son bulmasıyla sahip olduğu nesneler yok edilir.

```
Kod 2.9.1 - ThreadScope

package com.kurumsaljava.spring.scope;

import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;
import org.springframework.core.NamedThreadLocal;

public class ThreadScope implements Scope {

    private final ThreadLocal<Map<String, Object>> threadScope =
            new NamedThreadLocal<Map<String, Object>>(
```

```
            "ThreadScope") {
        @Override
        protected Map<String, Object> initialValue() {
            return new HashMap<String, Object>();
        }
    };

    @Override
    public Object get(String name, ObjectFactory objectFactory) {
        Map<String, Object> scope = threadScope.get();
        Object object = scope.get(name);
        if (object == null) {
            object = objectFactory.getObject();
            scope.put(name, object);
        }
        return object;
    }

    @Override
    public Object remove(String name) {
        Map<String, Object> scope = threadScope.get();
        return scope.remove(name);
    }

    @Override
    public void registerDestructionCallback(String name,
                        Runnable callback) {

    }

    @Override
    public Object resolveContextualObject(String key) {
        return null;
    }

    @Override
    public String getConversationId() {
        return Thread.currentThread().getName();
    }

}
```

Oluşturduğumuz yeni scope sınıfını Spring'e tanıtmamız gerekiyor. Bu amaçla kod 2.9.2 de yer aldığı gibi CustomScopeConfigurer sınıfının sahip olduğu scopes isimli listeye thread ismi altında ThreadScope sınıfını ekliyoruz.

```
Kod 2.9.2 - applicationContext.xml
```

```
<bean
    class="org.springframework.beans.factory.config.
                                CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread">
                <bean
                  class="com.kurumsaljava.spring.scope.
                                    ThreadScope"/>
            </entry>
        </map>
    </property>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
        scope="thread">
        <property name="customerRepository"
                        ref="customerRepository" />
        <property name="rentalRepository"
                        ref="rentalRepository" />
</bean>
```

Spring scope-thread ile karşılaştığı yerlerde ThreadScope sınıfının get() metodu üzerinden talep edilen nesneyi oluşturup bir ThreadLocal içindeki listeye (Map) ekleyecektir. Bu liste içinde yer alan nesneler tekrar oluşturulmadan kullanıcıya geri verilir.

Metot Enjeksiyonu

Spring'in varsayılan nesne oluşturma ayarının singleton yani tekil nesne olduğunu gördük. Tekil nesneler Spring tarafından bir kez oluşturulur ve kullanıma sunulur. Bu sebepten dolayı tekil olan bir nesneye prototype tipinde olan bir nesneyi devamlı enjekte etmek mümkün değildir, çünkü bağımlılığın enjekte edilmesi işlemi nesne oluşturulurken bir kez yapılan bir işlemdir. Prototype tipinde olan nesneler Spring tarafından devamlı yeniden oluşturulur, ama böyle bir nesneye bağımlılık duyan bir tekil nesne sürekli yeni prototype nesneyi enjeksiyon yöntemiyle edinemez, çünkü Spring tekil nesnelere oluşturulma süreçlerinden sonra bağımlılıkların enjeksiyonu konusunda ilgi göstermez.

Kod 2.9.3 de yer alan örnekte Singleton nesnesine prototype nesnesi konstrüktör üzerinde enjekte edilmektedir. Singleton sınıfının Spring konfigürasyon

dosyasında tekil, Prototype sınıfının prototype nesne olarak tanımlandığını düşünelim. Application Context oluştuktan sonra Spring bir defaya mahsus olmak üzere Singleton sınıfından olan tekil nesneye prototype değişkenini enjekte edecektir. Bu işlemin ardından Singleton sınıfından olan tekil nesnenin yeni bir prototype nesnesi edinmesi imkansızdır. Bu sorunu çözmek için metot enjeksiyonu yöntemini kullanabiliriz. Spring konstrüktör ve set() metotları bazlı enjeksiyon yanı sıra, mevcut bir sınıf metodunu değiştirerek, sınıfa yeni bir davranış biçimi kazandırmak için kullanılabilecek metot enjeksiyonu yöntemini sunmaktadır.

```
Kod 2.9.3 - Singleton

public class Singleton {

    private Prototype prototype;

    public Singleton(Prototype prototype) {
        this.prototype = prototype;
    }

    public void doSomething() {
        prototype.foo();
    }
}
```

Tekil bir nesneye ihtiyaç duyduğu prototype tipinde bir nesneyi verebilmek için prototype nesne edinme işlemini soyut bir metot olarak tanımlayabiliriz. Kod 2.9.4 de yer alan örnekte Singleton sınıfına prototype tipindeki nesneye konstrüktör aracılığı ile enjekte etmek yerine, createPrototype() isminde soyut bir metot tanımlıyoruz. createPrototype() metodun soyut metot olarak tanımlanabilmesi için Singleton sınıfının da soyut olarak tanımlanması gerekmektedir. Bu şartlar altında Spring'in Singleton sınıfından yeni bir nesne oluşturamayabileceğini düşünebilirsiniz. Metot enjeksiyon yöntemiyle bu mümkün.

```
Kod 2.9.4 - Singleton

public abstract class Singleton {

    private Prototype prototype;

    public Singleton() {
    }
```

```
    public void doSomething() {
        createPrototype().foo();
    }

    protected abstract Prototype createPrototype();
}
```

Metot enjeksiyonu yöntemi kullanıldığında Spring CGLIB kütüphanesini kullanarak Singleton sınıfından olma yeni bir altsınıf oluşturur. Dinamik olarak oluşturulan bu altsınıf soyut olan createPrototype() metodunu implemente eder. Bu implementasyon tekil nesneye yeni bir prototype nesneyi kullanma fırsatı tanır. Bu şekilde tekil nesneye dolaylı olarak bir prototype nesne enjekte edilmiş olur. Bu işlemi gerçekleştirmek için kullanılan yöntem metot enjeksiyonudur. Kod 2.9.5 de metot enjeksiyonu yöntemini kullanmak için gerekli Spring konfigürasyonu yer almaktadır. lookup-method elementi ile değiştirilmesi yani dinamik olarak implemente edilmesi gereken metot ismi tanımlanmaktadır.

```
Kod 2.9.5 - app-config.xml

<bean id="prototype"
        class="com.kurumsaljava.spring.Prototype" scope="prototype"/>

<bean id="singleton"
        class="com.kurumsaljava.spring.Singleton">
    <lookup-method name="createPrototype" bean="prototype"/>
</bean>
```

Dinamik olarak bir altsınıfın oluşturulabilmesi için tekil nesnenin ait olduğu sınıfın ve implemente edilen metodun final olmaması gerekmektedir. Bunun yanı sıra tekil nesnenin ait olduğu sınıf soyut olarak tanımlandığı için bu sınıfı test edebilmek için bir altsınıfın oluşturulması gerekmektedir. Ayrıca metot enjeksiyon yöntemiyle oluşturulan altsınıflar üzerinde serialization işlemi yapılamaz. Spring 3.2 ile CGLIB kütüphanesinin classpath içinde olma zorunluluğu bulunmamaktadır, çünkü bu kütüphanede yer alan sınıflar org.springframework paketine alınmıştır.

Metot Değiştirme Yöntemi

Replaced-method konfigürasyon elementini kullanarak bir nesnenin sahip olduğu metot yerine başka bir nesnenin sahip olduğu metodu koşturabiliriz. Bu yönteme method overriding ismi verilmektedir. Nasıl altsınıflar üstsınıflarda yer alan metotları @Override anotasyonunu kullanarak yeniden

yapılandırabiliyorlarsa, Spring'de mevcut bir metodu başka bir metotla değiştirebilmekte ve nesneye yeni bir davranış biçimi katabilmektedir. Bu metot değiştirme mekanizması da Spring tarafından kullanılan metot enjeksiyon yöntemiyle sağlanmaktadır.

Kod 2.9.5 de yer alan örnekte koşturmak istediğimiz metot doA() metodudur.

```
Kod 2.9.5 - AClass

public class AClass {

    public String doA(String a) {
        System.out.println(a);
        return a;
    }
}
```

Kullanılacak yeni metodu tanımlamak için org.springframework.beans.factory.support.MethodReplacer sınıfını implemente eden yeni bir sınıf oluşturmamız gerekmektedir. Spring'in doA() yerine koşturacağı metot kod 2.9.6 da ye alan reimplement() metodudur.

```
Kod 2.9.6 - ReplacerClass

public class ReplacerClass implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args)
                            throws Throwable {
        // args[0] doA() metodunda yer alan parametredir.
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

Gerekli konfigürasyon kod 2.9.7 de yer almaktadır. Replaced-method elementi ile orijinal metot ve bu metodun yerine geçecek yeni metot tanımlanmaktadır. Arg-type elementi ile yeni metot bünyesinde kullanılacak orijinal metot parametreleri tanımlanmaktadır.

```
Kod 2.9.7 - app-config.xml

<bean id="aClass" class="com.kurumsaljava.spring.AClass">
    <replaced-method name="doA" replacer="replacer">
        <arg-type>String</arg-type>
```

```
    </replaced-method>
</bean>

<bean id="replacer" class="com.kurumsaljava.spring.ReplacerClass"/>
```

Fabrika Metodu

Her sınıfın kullanıma açık sınıf konstrüktörü olmayabilir. Bu durum genelde tekil (singleton) nesneler için geçerlidir. Bir sınıftan birden fazla nesne üretimini engellemek için sınıf konstrüktörleri private olarak işaretlenir. Bu tür sınıflardan new kullanılarak nesne üretilmesi mümkün değildir. Bean elementinin factory-method özelliği aracılığı ile tekil bir nesnenin Spring ile kullanımı mümkündür.

```
Kod 2.10 - DBSingleton

package com.kurumsaljava.spring;
import java.sql.Connection;
public class DBSingleton {

    private static final DBSingleton instance = new DBSingleton();

    private DBSingleton() {

    }

    public static final DBSingleton getInstance() {
        return instance;
    }

    public Connection getConnection() {
        return null;
    }
}
```

Normal şartlar altında kod 2.10 da yer alan DBSingleton sınıfının kullanımı

```
DBSingleton.getInstance().getConnection();
```

şeklinde olacaktır. Bu sınıfı bir Spring nesnesi olarak tanımlayıp, kullanmak istersek, factory-method özelliği bu sınıfın tekil nesnesini kullanmamızı mümkün kılacaktır.

```
<bean id="dbSingleton"
```

```
        class="com.kurumsaljava.spring.DBSingleton"
    factory-method="getInstance" />
```

Factory-method kullanılarak tanımlanan metodun static olması gerekmektedir. Bu tanımlamanın ardından

```
DBSingleton dbSingletion = (DBSingleton) ctx.getBean("dbSingleton");
Connection connection = dbSingletion.getConnection();
```

şeklinde DBSingleton'ın ihtiva ettiği tekil nesneyi kullanabiliriz.

Fabrika Sınıfları

Bir Spring nesnesi tanımlaması yaparken class element özelliğinden faydalandık. Bu Spring'e hangi sınıfı kullanarak bir nesne oluşturması gerektiği bilgisini vermektedir. Class ve factory-method element özellikleri harici factory-bean element özelliği kullanılarak ta bir Spring nesnesi tanımlaması yapılabilir. Factory-bean kullanıldığı taktirde, bir POJO (Plain Old java Object) sınıfı nesneleri üreten fabrika (factory) olarak kullanılır.

```
<bean id="customerRepositoryFactory"
        class="com.kurumsaljava.spring.CustomerRepositoryFactory" />

<bean id="customerRepository"
                factory-bean="customerRepositoryFactory"
    factory-method="getNewInstance">
</bean>
```

Bir önceki kod bloğunda görüldüğü gibi customerRepository Spring nesnesi tanımlamasında class yerine factory-bean ve factory-method element özellikleri kullanılmıştır. Böyle bir konfigürasyonda Spring factory-bean ile tanımlanan fabrika sınıfının factory-method ile tanımlanan metodu ile bir customerRepository nesnesi oluşturur. Kullanılan fabrika sınıfı CustomerRepositoryFactory aşağıda yer almaktadır. Burada nesneyi oluşturma sorumluluğu tamamen factory-bean ile tanımlanan sınıfa aittir.

```
package com.kurumsaljava.spring;

public class CustomerRepositoryFactory {

    public CustomerRepository getNewInstance() {
        return new CustomerRepositoryImpl();
```

```
    }
}
```

FactoryBean Interface Sınıfı

Spring'in nesne oluşturma ve enjekte etme metotları sadece fabrika metodu ve fabrika sınıfları ile sınırlı değildir. Spring'in bir parçası olan FactoryBean interface sınıfı kullanılarak da nesne oluşturma ve enjekte etme işlemleri yapılabilir.

```
Kod 2.10.1 CustomerRepositoryFactoryBean

package com.kurumsaljava.spring;
import org.springframework.beans.factory.FactoryBean;
public class CustomerRepositoryFactoryBean implements
        FactoryBean<CustomerRepository> {

    public CustomerRepository getNewInstance() {
        return new CustomerRepositoryImpl();
    }

    @Override
    public CustomerRepository getObject() throws Exception {
        return new CustomerRepositoryImpl();
    }

    @Override
    public Class<?> getObjectType() {
        return CustomerRepository.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

FactoryBean interface sınıfında implemente edilmesi gereken üç metot bulunmaktadır. getObject() metodu tanımlanan sınıftan bir nesne oluşturur. getObjectType() metodu hangi sınıfın nesneyi oluşturmak için kullanılacağını tayin eder. isSingleton() metodu true değerini geri verirse, Spring tarafından tekil bir nesne oluşturulur. Bu değer false ise, Spring her istekte yeni bir nesne oluşturur.

RentalServiceImpl sınıfından bir nesneye CustomerRepositoryImpl sınıfından bir nesneyi enjekte etmek için CustomerRepositoryFactoryBean (kod 2.10.1) isimli, FactoryBean interface sınıfını implemente eden bir fabrika sınıfı oluşturmuş olduk. Bu fabrika sınıfını aşağıdaki şekilde Spring XML dosyasında bir customerRepository nesnesi oluşturmak ve rentalService nesnesine enjekte etmek için kullanabiliriz.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
                    ref="customerRepository" />
    <property name="rentalRepository"
                    ref="rentalRepository" />
</bean>

<bean id="customerRepository"
    class="com.kurumsaljava.spring.CustomerRepositoryFactoryBean" />
```

Bu örnekte görüldüğü gibi customerRepository isimli Spring nesnesi tanımlamasında class olarak FactoryBean interface sınıfını implemente eden CustomerRepositoryFactoryBean sınıfını kullandık. Böyle bir konfigürasyonda Spring CustomerRepositoryFactoryBean sınıfının getObject() metodunu kullanarak yeni bir CustomerRepositoryImpl nesnesi oluşturacaktır. Eğer Spring isSingleton() metodundan true değerini geri alırsa, bu durumda sadece bir defaya mahsus getObject() metodunu kullanarak bir nesne oluşturur. Oluşturduğu nesneyi hafızada tutarak, tekil bir nesne olmasını sağlar ve bu nesneyi her defasında kullanıma sunar. isSingletion() metodunun false değerini geri vermesi durumunda Spring her defasında getObject() metodu üzerinden yeni bir nesne oluşturur.

@Component anotasyonu kullanılmadığı taktirde enjekte edilmek istenen nesnelerin XML konfigürasyon dosyasında Spring nesnesi olarak tanımlanması zorunludur. Karmaşık yapıdaki nesnelerin XML konfigürasyon dosyasında Spring nesnesi olarak tanımlanmaları zor olabilir. Bu gibi durumlarda FactoryBean interface sınıfını implemente eden bir fabrika sınıfı oluşturulabilir. Karmaşık nesne oluşturma işlemi bu şekilde getObject() metoduna taşınarak bir metot bünyesinde gerçekleştirilmiş olur. Bu işlemin bir Java metodunda yapılması, bu kod biriminin tekrar kullanılabilirlik oranını ve şansını artırır. FactoryBean interface sınıfını implemente eden sınıflar Spring tarafından nesne üretici fabrika sınıfları olarak otomatik olarak keşfedilir ve kullanılır.

CustomerRepositoryFactoryBean tarafından oluşturulan nesneye

```
context.getBean("customerRepository");
```

şeklinde ulaşabiliriz. CustomerRepositoryFactoryBean sınıfından olan, yani fabrika nesnesine erişmek için & işaretinin şu şekilde kullanılması gerekmektedir.

```
context.getBean("&customerRepository");
```

Bu bize doğrudan fabrika nesnesini verir.

Sirküler Bağımlılıklar

Aşağıda yer alan örnekte server1 ve server2 arasında sirküler (cyclic) bağımlılık bulunmaktadır. Server1 nesnesini oluşturmak için server2, server2 nesnesini oluşturmak için server1 nesnesine ihtiyaç duyulmaktadır.

```
<bean id="server1"
        class="com.kurumsaljava.spring.service.MyService">
    <constructor-arg name="service" ref="server2"/>
</bean>

<bean id="server2"
        class="com.kurumsaljava.spring.service.MyService">
    <constructor-arg name="service" ref="server1"/>
</bean>
```

Spring böyle bir sirküler bağımlılık keşfettiğinde BeanCurrentlyInCreationException hatasını fırlatır. Bu sorunu aşmanın bir yöntemi bağımlılıkları set() metotları aracılığı ile enjekte etmektir. Genel olarak sirküler bağımlılıklar bakımı zor olduğundan, bu yapıların çözülmelerinde fayda vardır.

Bağımlılıkların Enjekte Edilmesi Yönteminin Avantajları

Spring çatısının sunduğu bağımlılıkların enjekte edilmesi yönetiminin yazılımcı ve uygulama için sağladığı avantajlar şunlardır:

- İnterface sınıflarının kullanımını teşvik eder. Bu yazılımcının uygulamayı bağımlılıkların tersine çevrilmesi (DIP - Dependency Inversion Principle) prensine uygun geliştirmesini sağlar.
- Uygulamanın hazır, konfigüre edilmiş nesneleri kullanmasını sağlar. Bağımlılıkların yazılımcı tarafından programlanmasını engeller.
- Kodu basitleştirir ve bakımını ve geliştirilmesini kolaylaştırır.
- Uygulamanın test edilmesini kolaylaştırır. Bağımlılıkları oluşturan nesneler yerine test amaçlı sahte nesne implementasyonları (stub ya da mock object) kullanımını mümkün kılar.
- Nesnelerin yaşam döngüsünü merkezi bir yerden kontrol etmeyi sağlar.

Bağımlılıkları Enjekte Ederken Hangi Yöntem Kullanılmalı?

Spring çatısında bağımlılıkları konstrüktör ya da set() metotları aracılığı ile enjekte edebileceğimizi gördük. Hangi yöntemi kullanmalıyız sorusu aklınıza gelmiş olabilir. Bu soruya açıklık getirmek isterim.

Her iki yöntemi de birlikte kullanmak mümkündür. Konstrüktör parametreleri zaman içinde artabileceği için konstrüktör aracılığı ile yapılan enjeksiyon karmaşık hale gelebilir. Bu yüzden benim tavsiyem set() metotlarını kullanarak bu işlemi gerçekleştirmektir. @Required anotasyonu kullanılarak, set() metotları aracılığı ile bağımlılıkların enjekte edilmesi zorunlu hale getirilebilir. Bunun yanı sıra altsınıflar otomatik olarak public ve protected olan set() metotları miras alırlar ve altsınıflara herhangi yeni bir metot tanımlama zorunluluğu olmadan nesneler enjekte edilebilir.

Yazılımcının kontrolü dışındaki sınıfların set() metotları olmayabilir. Bu durumda geriye sadece konstrüktör aracılığı ile enjeksiyon seçeneği kalmaktadır. Bunun yanı sıra konstrüktör bazlı enjeksiyonda bir nesne için tüm bağımlılıkları nesne oluşturma aşamasında doğrudan enjekte edildiğinden, kullanıcıya verilen nesne tam teşekküllü yapıda olacaktır. Final olan değişkenlere konstrüktör aracılığı ile enjeksiyon yapılabildiği için bu tür oluşturulan nesneler değiştirilemez (immutable) yapıdadır. Bunun gerekli olduğu durumlarda seçim konstrüktör bazlı enjeksiyon olmalıdır.

Spring'in Motoru Nasıl Çalıştırılır?

Spring'in iç organları sahip olduğu sınıflar ise, nefes alıp, vermesini sağlayan XML konfigürasyon dosyasıdır. Böyle bir XML dosyası Spring uygulamasının nasıl yapılandırılması gerektiğini ihtiva eder. Kullandığımız araç kiralama servisi örneğinde applicationContext.xml isminde böyle bir Spring XML konfigürasyon dosyası oluşturmuştuk. Bu dosyanın kullanılışını ve Spring uygulamasının nasıl ayağa kaldırıldığını kod 2.7 de incelemiştik.

Bir Spring uygulaması değişik ortamlarda çalışır hale getirilebilir. Bunlar:

- JUnit testleri
- Web uygulamaları
- Kurumsal Java uygulamalar (EJB)
- Tek başına (standalone) çalışan uygulamalar

Spring'in motorunun çalıştırılabilmesi için XML konfigürasyon dosyasının uygulama tarafından yüklenebilmesi gerekir. XML konfigürasyon dosyası değişik lokasyonlardan yüklenebilir. Bunlar:

- classpath
- sistemin herhangi bir dizini
- ortama ilişkin bir dizin (environment relavite path, örneğin web uygulamasının bir dizini)

Motorun çalışmasıyla birlikte Spring hafızada Application Context (resim 2.5) ismini taşıyan bir sunucu (container) oluşturur. Bu sunucuyu XML konfigürasyon dosyasının hafızadaki hali olarak düşünebiliriz. Singleton (tekil) olarak tanımlanmış tüm nesneler Spring tarafından oluşturularak sunucu içine konuşlandırılır. Singleton olmayan nesneler örneğin getBean() metodu aracılığı ile talep edildiklerinde, Spring tarafından gerekli bağımlılıklar enjekte edilerek kullanıcıya verilir.

XML Konfigürasyon Dosyasının Yüklenmesi

Bir Spring uygulamasının çalışır hale gelebilmesi için Spring nesnelerinin tanımlandığı XML dosyasının bulunması ve yüklenmesi gerekir demiştik. Kod 2.7 de kullandığımız Main sınıfı ClassPathXmlApplicationContext aracılığı ile classpath için bulunan applicationContext.xml dosyasını bularak, yüklemiş ve bir Application Context oluşturmuştu. XML konfigürasyonun classpath içinde bulunmadığı durumlarda FileSystemXmlApplicationContext kullanılabilir.

```
Kod 2.11 - FileSystemXmlApplicationContext Kullanımı

package com.kurumsaljava.spring;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
                        FileSystemXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext(
                "C:/resources/applicationContext.xml");
        RentalService rentalService =
                    (RentalService) ctx.getBean("rentalService");
        Rental rental =
                rentalService.rentACar("Özcan Acar",
                                new Car("ford", "fiesta"),
                getRentalBegin(), getRentalEnd());
        System.out.println("Rental status: " + rental.isRented());
        System.out.println((CustomerRepository)
                    ctx.getBean("customerRepository"));

    }

    private static Date getRentalEnd()
                        throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                    parse("29/12/2013");
    }

    private static Date getRentalBegin()
                        throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                    parse("22/12/2013");
    }

}
```

Kod 2.11 de bulunan Main sınıfı FileSystemXmlApplicationContext sınıfını kullanarak C:\resource\applicationContext.xml lokasyonunda bulunan applicationContext.xml dosyasını yüklemektedir.

Web uygulamalarında XML konfigürasyon dosyalarını yüklemek için XmlWebApplicationContext sınıfı kullanılabilir. Bu sınıfın nasıl kullanıldığını ilerleyen bölümlerde birlikte inceleyeceğiz.

Classpath içinde bulunan birden fazla konfigürasyon dosyasını yüklemek için joker işareti olarak bilinen * kullanılabilir.

```
ApplicationContext ctx =
        new ClassPathXmlApplicationContext("conf/*-config.xml");
```

Yukarıda yer alan tanımlama ile conf dizininde bulunan ve -config terimini ihtiva eden tüm konfigürasyon dosyaları Application Context'i oluşturmak için yüklenir.

Konfigürasyon dosyalarının hangi kaynaklardan yüklendiğini ifade etmek için classpath, file ve http gibi kısaltmalar kullanılabilir. Örneğin biri classpath içinde bulunan ve bir diğeri c:\resource dizinindeki iki konfigürasyon dosyası aşağıda yer alan kod satırında olduğu gibi yüklenebilir.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "classpath:applicationContext.xml",
        "file:C:/resources/test-datasource.xml");
```

ClassPathXmlApplicationContext classpath içinde yer alan konfigürasyon dosyalarını yüklemek için kullanılırken, file: kısaltması herhangi bir dizinde yer alan konfigürasyon dosyasını adreslemek için kullanılabilmektedir.

Eğer kullanmak istediğimiz konfigürasyon dosyası classpath ya da sistemdeki bir dizin içinde değilse, http: kısaltması kullanılarak bir konfigürasyon dosyasını bir uygulama sunucusundan edinebiliriz.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "http://ip:port/applicationContext.xml");
```

Çoklu XML Konfigürasyonu

Birden fazla XML dosyası kullanılarak bir Application Context oluşturulabilir. Bu şekilde Spring nesnelerini mantıksal guruplara ayırmak mümkündür. Çoğu zaman bu mekanizma uygulamanın servis katmanında kullanılan Spring nesneleri ile altyapı katmanında kullanılan Spring nesnelerini birbirlerinden bağımsız olarak, ayrı XML dosyalarında tanımlamak için kullanılır. Her ortama

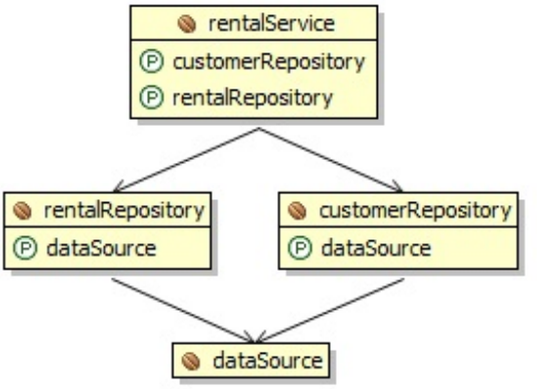
göre kullanılan altyapı komponentleri değişik olacağından, böyle bir ayrıştırma servis katmanı konfigürasyonunun değişik altyapı konfigürasyonları ile kombine edilerek kullanılmasını mümkün kılar.

Araç kiralama servisi uygulaması için böyle bir ayrıma gitseydik, konfigürasyonumuz nasıl olurdu? Bu sorunun cevabı bir sonraki kod bloklarında yer almaktadır.

```
Kod 2.12 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/
                        beans
    http://www.springframework.org/schema/beans/
                        spring-beans-3.0.xsd">

    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl"
        scope="prototype">
        <property name="customerRepository"
                        ref="customerRepository" />
        <property name="rentalRepository"
                        ref="rentalRepository" />
    </bean>

    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                        RentalRepositoryImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="customerRepository"
            class="com.kurumsaljava.spring.
                        CustomerRepositoryImpl">
        <property name="dataSource"
                    ref="dataSource" />
    </bean>

</beans>


Kod 2.13 - test-datasource.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
                                spring-beans-3.0.xsd">

    <bean id="dataSource"
        class="com.kurumsaljava.spring.DummyDataSourceImpl"/>
</beans>


Kod 2.14 - Main
public static void main(String[] args) throws Exception {
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("applicationContext.xml",
            "test-datasource.xml");
    RentalService rentalService =
            (RentalService) ctx.getBean("rentalService");
    Rental rental =
            rentalService.rentACar("Özcan Acar",
                new Car("ford", "fiesta"),
            getRentalBegin(), getRentalEnd());
    System.out.println("Rental status: " + rental.isRented());
    System.out.println((CustomerRepository)
                    ctx.getBean("customerRepository"));
}
```

Servis katmanında yer alan rentalService, customerRepository ve rentalRepository sınıfları için applicationContext.xml isminde bir konfigürasyon dosyası (kod 2.12) oluşturduk. Şimdiye kadar kullandığımız RentalRepositoryImpl ve CustomerRepositoryImpl sınıfları herhangi bir veri tabanı işlemi yapma özelliğini sahip değiller. Bu sınıfların veri tabanı işlemlerini gerçekleştirebilmeleri için DataSource ismini taşıyan bir interface sınıf tanımlıyoruz. Kod 2.12 de yer alan Spring nesne tanımlamaları ile rentalRepository ve customerRepository nesnelerine datasource ismini taşıyan bir nesne enjekte edilmektedir. Nesneler arasındaki oluşturmak istediğimiz bağlantı resim 2.8 de yer almaktadır.



Resim 2.8

Uygulamanın geliştirilmesi için kullanılan veri tabanı sistemi ile, uygulama sunucuları üzerinde çalışırken kullanılan veri tabanı sistemi değişik türde olabilir. Bu tür değişiklikleri uygulamadan saklamak amacıyla DataSource tarzı interface sınıflar kullanılır. Bu interface sınıfın değişik ortamlara göre implemente edilmesi gerekmektedir. Örneğin oluşturduğumuz birim testlerini koşturmak için DummyDataSourceImpl isminde gerçek bir veri tabanı sistemini kullanmayan bir implementasyon oluşturabiliriz. Bu implementasyon birim testlerini koşturmak için gerekli verileri veri tabanından edinmişcesine birim testine verebilir. Uygulamanın gerçek sunucularda bir Oracle veri tabanı sistemi ile çalışmasını sağlamak amacıyla OracleDataSourceImpl implementasyonu oluşturulabilir.

Eğer dataSource tanımlamasını applicationContext.xml dosyasında yapmış olsaydık, uygulamanın çalıştığı ortama göre kullanılan DataSource implementasyonunu bu dosya üzerinde değiştirmek zorunda kalırdık. Bunu önlemek amacıyla kod 2.13 de yer aldığı gibi test-datasource.xml ismini taşıyan yeni bir XML konfigürasyon dosyası oluşturuyoruz. applicationContext.xml ve test-datasource.xml dosyaları kod 2.14 de yer aldığı gibi ClassPathXmlApplicationContext aracılığı ile kombine edilebilmektedir. Bu uygulamayı test için kullanabileceğimiz bir Application Context oluşturur. Uygulamayı müşterimiz için çalışır hale getirmek istediğimizde datasource.xml isminde, OracleDataSourceImpl sınıfını kullanan yeni bir konfigürasyon dosyası oluşturabilir ve ClassPathXmlApplicationContext üzerinden applicationContext.xml dosyasını bu konfigürasyon dosyası ile kombine ederek, uygulamayı bir Oracle veri tabanını kullanacak şekilde konfigüre edebiliriz.

Çoklu XML konfigürasyon imkanı uygulamayı tanımlanmış parçalara bölerek, bu parçaları değişik ortamlarda kombine etmemizi ve ortama uygun bir Spring uygulaması oluşturmamızı mümkün kılmaktadır. Uygulamayı oluşturan bu

konfigürasyon dosyaları aynı kod birimlerinde olduğu gibi tekrar kullanımı teşvik etmekte ve uygulamanın modüler bir yapıda olmasını sağlamaktadır.

2. Bölüm Soruları

2.1 Alan modeli oluşturmanın ana amacı nedir?

2.2 Bir Spring nesnesi tanımlaması yaparken, bu nesneye birden fazla isim nasıl atanır?

2.3 Spring bağımlılıkları enjekte etmek için kaç yöntem tanımaktadır?

2.4 Metot enjeksiyonu ne amaçla kullanılır?

2.5 Bir değişkene null değerini nasıl atanır?

2.6 IdRef elementi neden kullanılır?

2.7 Tekil (singleton) nesneler Spring bean olarak tanımlanabilir mi?

2.8 Hangi Spring sınıfı kullanılarak bir sınıfın kendi istediği usulde nesne oluşturması sağlanabilir?

2.9 Sirküler bağımlılık nedir?

3. Bölüm

Spring İle Nesne Yaşam Döngüsü Yönetimi

Spring XML İsim Alanları (XML Namespaces)

Şimdiye kadar oluşturduğumuz Spring XML dosyalarında bir nevi bilgisayar dili kullandık. Bu dil <bean/> elementi ile bir Spring bean ve <property/> elementi ile enjekte edilecek bir bağımlılığı tanımlamamızı mümkün kıldı. Bu dilin kullandığı elementler isim alanı (namespace) olarak tabir edilen bir kümede bir araya getirilir. Bu kelime kümesi XML Schema (XSD - XML Schema Definition) kullanılarak tanımlanır. Bir dilin ihtiva ettiği kelimelerin tanımlandığı bu XSD dosyasını bir sözlük olarak düşünebilirsiniz. Kullanılabilecek kelimeler bu sözlükte yer alır.

Standart konfigürasyon işlemleri için Spring tarafından spring-beans-3.x.xsd dosyası kullanılmaktadır. Bu dosyada <bean/> ve <property/> gibi elementler tanımlıdır. Bir önceki bölümde oluşturduğumuz applicationContext.xml isimli konfigürasyon dosyasında spring-beans-3.0.xsd dosyasını şu şekilde kullandık:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

Spring açışından bakıldığında bir Spring uygulamasındaki her şey bir Spring nesnesidir ve <bean/> elementi kullanılarak tanımlanabilir. Her şeyin <bean/> ile tanımlanabilmesi, konfigürasyon işlemini basitleştirmektedir. Lakin yazılımcı perspektifinden bakıldığında durum böyle değildir. Yazılımcının konfigürasyon dosyasında tanımladığı her şey jenerik bir bean olmayabilir. Çoğu zaman her bean tanımlaması için spesifik konfigürasyon gerekli olabilir. Spring yazılımcının istediği türde bean tanımlama işlemini kolaylaştırmak için XSD bazlı konfigürasyonu mümkün kılmaktadır.

Yazılımcılar XSD bazlı konfigürasyonu daha çok AOP, transaction ve diğer çatılarla entegrasyonu sağlamak gibi altyapı işlemleri için kullanırlar. Spring bu tür işlemleri tanımlayan XSD dosyaları ihtiva eder. Bu XSD dosyalarında bulunan elementleri kullanabilmek için XSD dosyasının Spring XML konfigürasyon dosyası içinde bir isim alanı oluşturacak şekilde yüklenmesi (import) gerekmektedir. Bunun nasıl yapıldığını bir örnek üzerinde inceleyelim.

```
3.1 - applicationContext.xml

<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
```

```
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="dataSource"
        class="com.kurumsaljava.spring.JNDIDataSourceImpl">
    <property name="jndiName" value="jdbc/MyDataSource" />
</bean>
```

Kod 3.1 de yer alan dataSource bean tanımlamasına jdbc/MyDataSource isimli JNDI (Java Naming And Directory Inteface) kaynağı enjekte edilmektedir. Burada yapılan basit bir JEE (Java Enterprise Edition) konfigürasyonudur. Böyle bir tanımlamayı daha basit ve ifade gücü daha yüksek bir şekilde yapabilmek için spring-jee-3.0.xsd dosyasında tanımlanan elementleri kullanabiliriz. Bu XSD dosyası kullanılarak oluşturulan isim alanı kod 3.2'de yer almaktadır.

```
Kod 3.2 - jee isim alanini kullanan applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/
                                spring-jee-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                                spring-beans-3.0.xsd">

    <bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
        scope="prototype">
        <property name="customerRepository"
                    ref="customerRepository"/>
        <property name="rentalRepository"
                    ref="rentalRepository" />
    </bean>

    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
        <property name="dataSource" ref="dataSource" />
```

```
    </bean>

    <jee:jndi-lookup id="dataSource"
            jndi-name="jdbc/MyDataSource" />
</beans>
```

Konfigürasyon dosyasında xmlns:jee ile jee isim alanı oluşturduktan sonra, bu isim alanında tanımlı bulunan jndi-lookup elementini

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource" />
```

şeklinde kullanabiliriz. Eğer jee isim alanı spring-jee-3.0.xsd kullanılarak oluşturulmadı ise jndi-lookup elementini kullanmamaz mümkün değildir. <jndi-lookup/> şeklindeki bir tanımlama hata verecektir, çünkü Spring bu elementin hangi isim alanına ait olduğunu bilmemektedir. Spring'e bu isim alanını tanıtmak için kod 3.2 de görüldüğü gibi xmlns:jee isimli yeni bir isim alanı oluşturulmakta ve spring-jee-3.0.xsd dosyasında tanımlı tüm elementler konfigürasyon dosyasında kullanılmak üzere import edilmektedir.

Spring konfigürasyon dosyalarında kullanılabilecek isim alanları şunlardır:

- ***aop***: Aspekt tanımlama için kullanılan isim alanıdır.
- ***beans***: Spring'in kullandığı temel isim alanıdır; bean tanımlamaları ve bağımlılıkları enjekte etmek için kullanılan elementleri ihtiva eder.
- ***context***: Application Context ve Spring sunucusunu konfigüre etmek için kullanılan isim alanıdır.
- ***jee***: Java EE API'leri ile entegrasyonu sağlamak için kullanılan elementleri ihtiva eder.
- ***lang***: Groovy, JRuby ya da BeanShell skriptleri olarak kodlanan kod birimlerini Spring Bean olarak tanılamak için kullanılan isim alanıdır.
- ***jms***: JMS tabanlı sınıfları konfigüre etmek için kullanılan elementleri ihtiva eder.
- ***oxm***: Nesne-XML dönüşümünü sağlayan konfigürasyon elementlerini ihtiva eden isim alanıdır.
- ***tx***: Bean'lerin transaksiyonel özelliğe kavuşturulmasını sağlayan elementleri ihtiva eder.
- ***util***: Listelerin bean olarak tanımlanabilmesi gibi değişik yardımcı (utility) elementleri ihtiva eden isim alanıdır.

İsim alanlarının kullanımını özetleyecek olursak:

- İsim alanlarının kullanımı konfigürasyonu basitleştirir ve yapılan tanımlamaların ifade gücünü artırır.
- Değişik türdeki altyapı konfigürasyonları için kategoriler (aop, beans, context, jee, jms, tx, util) oluşturur. Her kategorinin kendine has XSD dosyası vardır.

Kitabın ilerleyen bölümlerinde altyapı komponentlerini konfigüre etmek için XSD dosyalarının ve isim alanlarının nasıl kullanıldığını örnekler üzerinde inceleyeceğiz.

Bir Spring Uygulamasının Yaşam Döngüsü

İkinci bölümde Application Context'in nasıl oluşturulduğuna değinmiştim. Kısaca Application Context bir Spring uygulaması çalışmaya başladığı zaman Spring tarafından oluşturulan ve hafızada konuşlandırılan bir sunucudur (container). Bu sunucu içinde Spring uygulaması ile ilgili her türlü detay yer alır. Spring Application Context bünyesindeki ayarlara göre nesneler oluşturur ve gerekli bağımlılıkları enjekte eder.

Resim 3.3

Canlılarda olduğu gibi bir Spring uygulamasının da yaşam döngüsü vardır. Bu yaşam döngüsü değişik fazlardan oluşur. Bir Spring uygulamasının yaşam döngüsünü oluşturan fazlar resim 3.3 de yer almaktadır.

İlk faz kurulum fazıdır. Bu fazda Spring Application Context'i kullanım için hazırlar. Sunucu ayarları ve hangi bean tanımlamalarından oluştuğu XML konfigürasyon dosyasında yer alır. Uygulamanın sunduğu servisler bu fazda çalışır ve hizmet verir hale getirilir. Bu faz tamamlanmadan uygulama kullanılır hale gelmez.

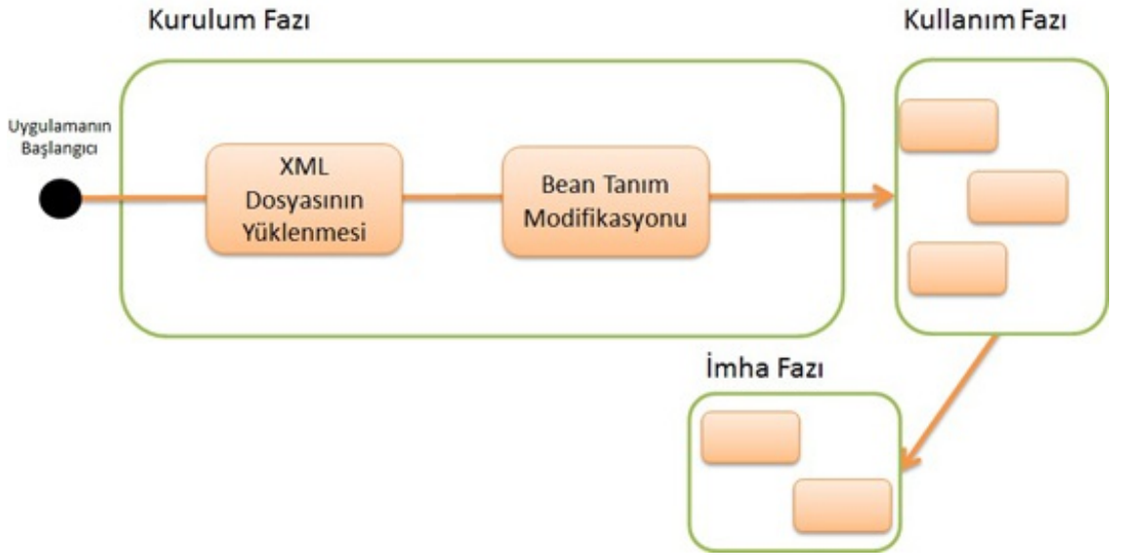
Kullanım fazında sunucu kullanıcı isteklerine cevap verir. Uygulamanın hayatının %99.99'luk bölümü bu fazdan oluşur.

İmha fazında Spring Application Context ve ihtiva ettiği her şeyi sonlandırır ve uygulamayı durdurur. Uygulama tarafından kullanılan tüm kaynaklar bırakılır (release). Uygulama Java Garbage Collector tarafından temizlenmeye hazır hale gelir.

Spring bahsettiğim yaşam döngüsünün yöneticisidir. Bu yaşam döngüsü her türlü Spring uygulaması için aynıdır. Bu fazlarda hangi detayların saklı olduğunu yakından inceleyelim.

Kurulum Fazı

Klasik bir Java uygulamasında nesneler new operatörü ile oluşturulur ve kullanılır. Kullanım sona erdikten sonra bu nesneler Garbage Collector tarafından toplanarak, ebediyete intikal ettirilir. Buna kıyasla Spring'in nesne oluşturma ve yaşam döngüsünü yönetme mekanizmaları daha sofistikedir. Spring her nesneyi öngörülen konfigürasyon doğrultusunda değişik işlemlere tabi tutarak, kullanım öncesi ihtiyaç duyulduğu şekilde yapılandırılmasını sağlar. Yazılımcının Spring tarafından yapılan işlemlere müdahil olması ve nesnelerin ihtiyaçları doğrultusunda oluşmasını sağlaması mümkündür.



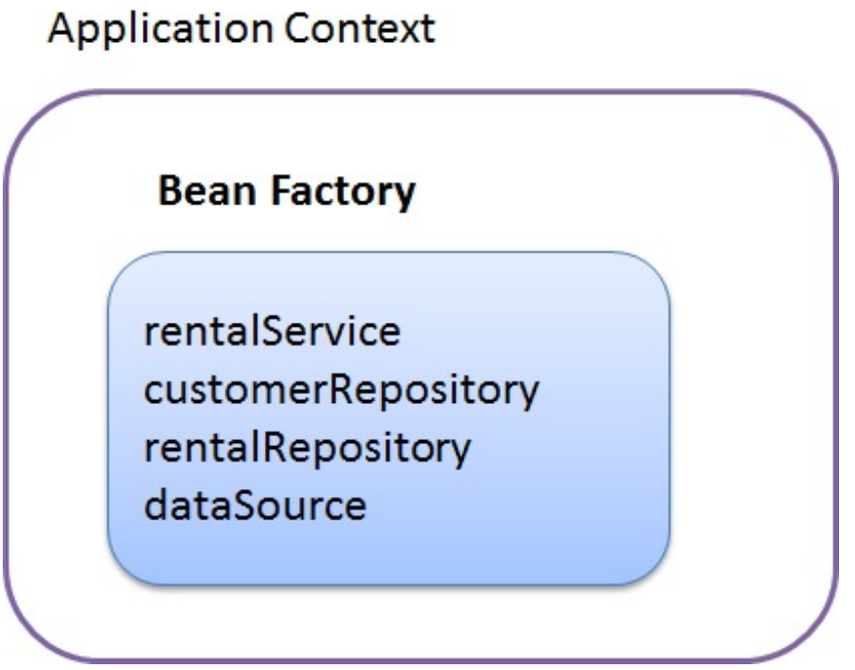
Resim 3.4

Kurulum fazı (initialization) iki safhadan (resim 3.4) oluşur. Birinci safhada Spring XML konfigürasyon dosyasını hafızaya yükler ve bean tanımlamalarını tarar. Birinci safhanın start alabilmesi için aşağıdaki şekilde Application Context'in oluşturulması gerekmektedir.

```
ApplicationContext ctx =
```

```
        new ClassPathXmlApplicationContext("
                        applicationContext.xml");
```

Bu işlem kurulum fazının ilk safhasını tetikler. Spring'in XML dosyasını yüklemesiyle birlikte bilgisayarın hafızasında resim 3.5 de yer alan Application Context oluşur. Spring akabinde tespit ettiği tüm bean tanımlamalarını Application Context içinde yer alan BeanFactory'ye yükler. Her bean sahip olduğu bean id'si ile BeanFactory içinde konuşlandırılır.



Resim 3.5

Kurulum fazının ikinci safhası Post Processing Bean Definitions, yani bean tanımlamalarının gözden geçirildiği ve gerekli durumlarda modifike edildiği safhadır. Bu safhada henüz nesneler oluşturulmamışken bean tanımlamalarını modifike ederek, nesne oluşturma sürecini şekillendirmek mümkündür. Bu amaçla BeanFactoryPostProcessor interface sınıfını implemente eden işlemciler devreye girer. Yazılımcı bu interface sınıfını implemente eden işlemciler oluşturabilir ya da mevcut işlemcileri kullanabilir.

Bu işlemcilerin bir örneğini PropertyPlaceHolderConfigurer teşkil etmektedir. Bu işlemci ${degisken} şeklinde konfigürasyon dosyalarında yer alan yer tutucuların kurulum fazının ikinci safhasında gerçek değerlerle yer değiştirmesini sağlar. Bunun bir örneği kod 3.3 de yer almaktadır.

```
Kod 3.3 - applicationContext.xml

<bean id="customerRepository"
```

```
            class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="dataSource" ref="dataSource"></property>
</bean>


<bean id="dataSource"
        class="com.kurumsaljava.spring.OracleDatasourceImpl">
    <property name="url" value="${datasource.url}"></property>
    <property name="user" value="${datasource.user}"></property>
    <property name="password" value="${datasource.pwd}"></property>
</bean>


<bean
    class="org.springframework.beans.factory.config.
                PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:database.properties"/>
</bean>
```

Kod 3.3 de yer alan dataSource bean tanımlaması ${datasource.url} şeklinde yer tutucular ihtiva etmektedir. Bu yer tutucuların kullanılmasının sebebi, konfigürasyon dosyasını spesifik değerlerden bağımsız kılma isteğidir. Kullanılan ortama göre bu değerler değişebileceği için bu değerlerin bir değer dosyasında (property file) tutulmasında fayda vardır. Kullanıcı her ortam için başka bir değer dosyası oluşturularak, mevcut konfigurasyon dosyasını değiştirmek zorunda kalınmadan bu dosyanın tekrar kullanımını sağlayabilir.

Yer tutucuların gerçek değerler ile yer değiştirebilmesi için konfigürasyon dosyasında PropertyPlaceholderConfigurer'in işlemci olarak tanımlanması gerekmektedir. Kod 3.3 de bu işlemci tanımlanmıştır. Bu işlemcinin görevini yerine getirebilmesi için değer dosyasının yerini bilmesi gerekmektedir. Bu amaçla işlemcinin location değişkenine classpath içinde yer alan database.properties değer dosyası enjekte edilmektedir. Bu değer dosyası kod 3.4 de yer almaktadır.

```
Kod 3.4 - database.properties

datasource.url=jdbc:oracle:thin:@localhost:1521:rentacar
datasource.user=rentacar
datasource.pwd=password
```

Context isim alanını kullanarak PropertyPlaceholderConfigurer işlemcisini aşağıdaki şekilde daha kısa bir şekilde tanımlamak mümkündür.

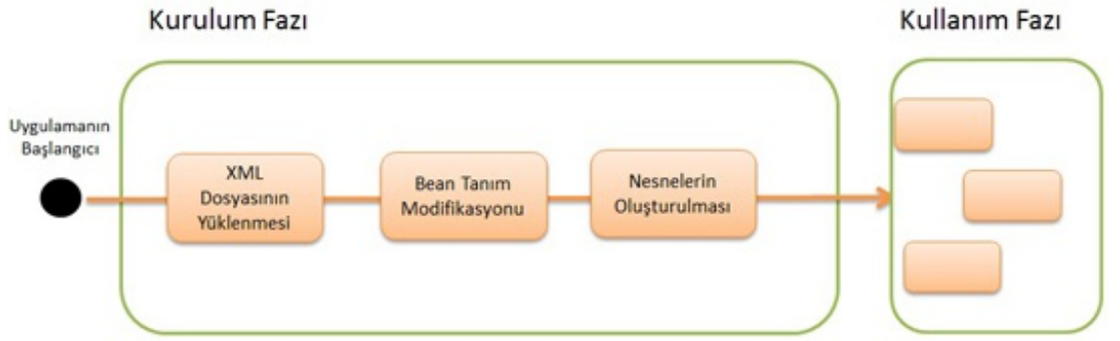
```
<context:property-placeholder
           location="classpath:database.
               properties" />
```

Kurumlum fazının ikinci safhasında BeanFactoryPostProcessor interface sınıfını implemente eden işlemcilerin Spring nesnesi tanımlamalarını gereklilikler doğrultusunda modifike edebildiklerini gördük. Böyle bakıldığında BeanFactoryPostProcessor interface sınıfı Spring sunucusu için bir genişletme noktasıdır (extention point). BeanFactoryPostProcessor interface sınıfını implemente eden kendi işlemcilerimizi geliştirerek, kurulum fazının ikinci safhasında isteklerimiz doğrultusunda Spring nesne tanımlamalarını adapte edebiliriz. BeanFactoryPostProcessor sınıfından olan işlemciler ile Spring nesneleri oluşturulup, Application Context içine yerleştirilmeden önce Spring tanımlamaları üzerinde her türlü değişiklik yapılabilir.

```
public interface BeanFactoryPostProcessor{
    public void postProcessBeanFactory
        (ConfigurableListableBeanFactory beanFactory);
}
```

Nesnelerin Oluşturulması

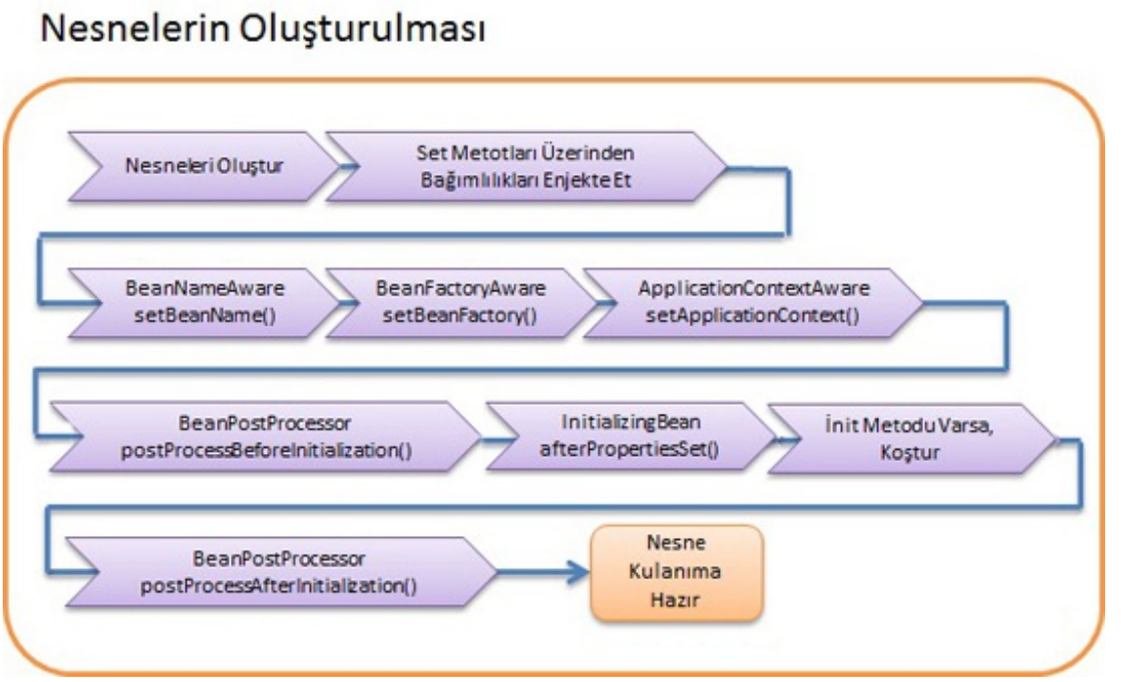
Buraya kadar yaptıklarımız bean tanımlamaları üzerindeki modifikasyonlardır. Bu modifikasyonlar nesneler oluşturulmadan yapılan bean tanımlama modifikasyonlarıdır. Nesnelerin oluşturulması ve yapılandırılması kurulum fazının henüz bahsetmediğimiz üçüncü safhasında gerçekleşir. Bu sebepten dolayı resim 3.4 de yer alan kurulum fazını resim 3.6 deki gibi genişletmemiz gerekiyor.



Resim 3.6

Kurulum fazının son safhası olan nesnelerin oluşturulması safhasını mikroskop

altında incelediğimizde, resim 3.7 de yer alan işlemlerin yapıldığını görebiliriz.



Resim 3.7

Bu safhada Spring aşağıdaki işlemleri gerçekleştirir:

- BeanFactory (resim 3.5) bünyesinde yer alan tüm bean tanımlamalarından nesneler oluşturulur ve bu nesnelere ihtiyaç duydukları bağımlılıklar enjekte edilir.
- Eğer sınıf BeanNameAware interface sınıfını implemente etti ise, sınıfın setBeanName() metodu koşturulur. Bu metot üzerinden sınıfa bean ismi enjekte edilir, örneğin rentalRepository.
- Eğer sınıf BeanFactoryAware interface sınıfını implemente etti ise, sınıfın setBeanFactory() metodu koşturulur. Bu metot üzerinden sınıfa kullanımdaki BeanFactory nesnesi enjekte edilir.
- Eğer sınıf ApplicationContextAware interface sınıfını implemente etti ise, sınıfın setApplicationContext() metodu koşturulur. Bu metot üzerinden sınıfa kullanımdaki ApplicationContext nesnesi enjekte edilir.
- Eğer sınıf BeanPostProcessor interface sınıfını implemente etti ise, sınıfın postProcessBeforeInitialization() metodu koşturulur.
- Eğer sınıf InitializingBean interface sınıfını implemente etti ise, sınıfın afterPropertiesSet() metodu koşturulur.
- Eğer bean tanımlamasında init-method kullanılarak bir metot tanımlandıysa, sınıfın bu metodu koşturulur.
- Eğer sınıf BeanPostProcessor interface sınıfını implemente etti ise, sınıfın

postProcessAfterInitialization() metodu koşturulur.

Bu noktadan itibaren tekil (singleton) nesne kullanıma hazırdır ve Application Context son bulana kadar Application Context içinde kalır. Çoğul nesneler (prototype) için de Spring bu işlemleri nesne talep edildiğinde gerçekleştirir ve kullanıma sunar.

InitializingBean interface sınıfının implemente edilmesi tavsiye edilmemektedir. Bunun yerine

```
<bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl"
                    init-method="init">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

şeklinde init-method ile bir metot tanımlanabilir. Bu metot Spring tarafından otomatik olarak koşturulacaktır. Spring'in sunduğu interface sınıfları implemente etmek kodu Spring çatışına bağımlı kılacağından, init-method örneğinde olduğu gibi Spring ile etkileşim konfigürasyon dosyası üzerinden gerçekleşmelidir. Bu kodun Spring çatışına olan bağımlılığını azaltır.

Her sınıfın init() metodunun koşturulmasını istiyorsak, bunu default-init-method kullanarak tanımlayabiliriz. Default-init-method şu şekilde tanımlanabilir:

```
<beans  default-init-method="init">
    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
</beans>
```

Default-init-method her bean için init-method tanımı yapılması gerekliliğini ortadan kaldırır. Spring böyle bir konfigürasyonda bean olarak tanımlanan her sınıfın init() isimli metodunu koşturmaya çalışacaktır. Eğer sınıf böyle bir metoda sahip ise, bu metot koşturulur. Böylece init() metodunu standart bir şekilde implemente etmek mümkün olacaktır.

BeanPostProcessor Kullanımı

Resim 3.7 ye tekrar baktığımızda bir nesne oluşturulduktan ve ihtiyaç duyduğu

bağımlılıklar enjekte edildikten sonra Spring sunucusu tarafından nesnenin InitializingBean interface sınıfından aldığı afterPropertiesSet() ve varsa init-method ile tanımlanan metodu koşturulmaktadır. Bu iki işlemin hemen öncesinde ve sonrasında Spring sunucusu BeanPostProcessor interface sınıfında yer alan şu iki metodu koşturur:

```
public interface BeanPostProcessor {

    public Object postProcessBeforeInitialization(
        Object bean, String beanName) throws BeansException;

     public Object postProcessAfterInitialization(
            Object bean, String beanName) throws BeansException;
}
```

BeanPostProcessor metotları Spring nesnesi üzerinden her türlü işlemi yapabilirler. BeanPostProcessor metotlarında yer alan Object bean Spring nesnesidir. Örneğin Spring nesnesi için bu metotlar içinde bir vekil (proxy) nesne oluşturulabilir. Nitekim bazı Spring AOP altyapı sınıfları BeanPostProcessor sınıfını implemente ederek, Spring sunucusu tarafından oluşurulan nesnelere vekil nesneleri atamaktadırlar.

BeanPostProcessor interface sınıfının iki kullanım şekli bulunmaktadır. İsteyen sınıflar bu interface sınıfı implemente ederek, Spring sunucusu tarafından gerekli yerlerde bu metotların koşturulmalarını sağlayabildikleri gibi, bu interface sınıfı implemente eden ve tüm Spring sunucu genelinde geçerli olan işlemciler oluşturulabilir. Bu işlemciler Spring sunucusu tarafından bir nesne oluşturulduğunda devreye girerek, sahip oldukları metotlar bünyesinde Spring nesnesi üzerinde işlem yapabilirler.

Tüm Spring sununcusu genelinde geçerli olan RequiredAnnotationBeanPostProcessor işlemcisini örnek olarak verebiliriz. Spring çatısı bünyesinde yer alan bu BeanPostProcessor implementasyonu @Required anotasyonunu ile sınıf değişkenlerine gerekli değerin enjekte edilip, edilmediğini kontrol eder. Bu şekilde ihtiyaç duyduğu tüm bagımlılıklar enjekte edilmeden bir Spring nesnesinin kullanıma sunulması engellenmiş olur. Yazılımcı buna benzer kullanım senaryoları için BeanPostProcessor interface sınıfını implemente eden kendi işlemcilerini oluşturabilir.

@Required ile oluşturulmak istenen kontrol mekanizmasının işleyebilmesi için RequiredAnnotationBeanPostProcessor sınıfının konfigürasyon dosyasında şu

şekilde tanımlanması gerekmektedir:

```
<bean
   class="org.springframework.beans.factory.annotation.
       RequiredAnnotationBeanPostProcessor"/>
```

Spring konfigürasyon dosyasında RequiredAnnotationBeanPostProcessor tanımlaması ile karşılaştığı taktirde, classpath içinde yer alan ve Spring bean olarak tanımlanmış tüm sınıfların @Required ile işaretlenmiş metotlarını kontrol eder. @Required sadece metot bazında kullanılabilmektedir. @Required olan bir değerin enjekte edilmemiş olması durumunda nesne oluşturma işlemi durdurulur ve bu hata olarak kullanıcıya yansıtılır.

Eğer konfigürasyon dosyasına context isim alanı yüklendi ise, RequiredAnnotationBeanPostProcessor tanımlaması şu şekilde kısaltılabilir:

```
<context:annotation-config/>
```

annotation-config elementinin kullanımı RequiredAnnotationBeanPostProcessor sınıfının aktif hale gelmesini sağlamakta ve Spring bean olarak tanımlanma gerekliliğini ortadan kaldırmaktadır.

@PostConstruct Anotasyonu

Spring'in 2.5 sürümü ile nesne yapılandırma işlemi için init-method yerine @PostConstruct anotosyonu kullanılabilir. @PostConstruct JSR 250 ile tanımlanmış standart bir Java anotasyonudur. @PostConstruct kullanımı Spring çatışına olan kod bağımlılığını tamamen ortadan kaldırır. Bu yüzden tavsiye edilen yöntemdir.

@PostConstruct anotasyonu şu şekilde kullanılabilir:

```
@PostConstruct
public void init() {
    System.out.println("init");
}
```

Spring tarafından bu anotasyonun tanınabilmesi için CommonAnnotationBeanPostProcessor sınıfının konfigürasyon dosyasında şu şekilde tanımlanması gerekmektedir:

```
<bean class="org.springframework.context.
            annotation.CommonAnnotationBeanPostProcessor"/>
```

Yukarıda yer alan örnekte CommonAnnotationBeanPostProcessor anonim bir Spring nesnesi olarak tanımlanmıştır. Anotasyonlu sınıfların aktivasyonu için böyle bir tanımlama yeterlidir. Bu tanımlama ile karşılaştığında Spring ne yapması gerektiğini bilir.

Spring konfigürasyon dosyasında CommonAnnotationBeanPostProcessor tanımlaması ile karşılaştığı taktirde, classpath içinde yer alan ve Spring bean olarak tanımlanmış tüm sınıfların @PostConstruct ile tanımlanmış metodunu koşturur. Bu işlemi set() metotları ya da konstrüktör aracılığı ile bağımlılıkları enjekte ettikten sonra yapar.

Eğer konfigürasyon dosyasına context isim alanı yüklendi ise, CommonAnnotationBeanPostProcessor tanımlaması şu şekilde kısaltılabilir:

```
<context:annotation-config/>
```

Context isim alanında yer alan annotation-config elementinin kullanımı CommonAnnotationBeanPostProcessor ve RequiredAnnotationBeanPostProcessor gibi BeanPostProcessor sınıflarının aktif hale gelerek, anotasyonlu sınıflar üzerinde gerekli işlemlerin yapılmasını sağlar. annotation-config elementi kullanıldıgı taktirde bu BeanPostProcessor sınıflarının konfigürasyon dosyasında Spring bean olarak tanımlanmaları gerekliliği ortadan kalkmaktadır.

Özetleyecek olursak, nesne oluşturma sürecine müdahil olmak için üç yöntem bulunmaktadır. Bunlar InitializingBean interface sınıfının implemente edilmesi, init-method ve @PostConstruct anotasyonunun kullanımıdır. Bu yöntemlerden hangisini kullanmalıyım sorusu aklınıza gelmiş olabilir. Daha önce de belirttiğim gibi InitializingBean interface sınıfının kullanımı kodu Spring çatısına bağımlı kılmaktadır. Bu yüzden kullanılmamalıdır. Eğer bir sınıfın kaynak koduna erişemiyorsanız init-method kullanılabilir. Kaynak koda sahipseniz @PostConstruct anotasyonunu kullanabilirsiniz.

Yaşam Döngüsü Metotlarının Kombinasyonu

InitializingBean, DisposableBean, @PostConstruct, @PreDestroy, init-method

ve destroy-method aracılığı ile nesnelerin oluşturulma ve imha edilme sürecine müdahil olabiliriz. Eğer birden fazla yaşam döngüsü metodunu implemente ettiysek, Spring bu metotları şu sıraya göre koşturacaktır:

- @PostConstruct anotasyonu ile işaretli metotlar
- InitializingBean sınıf tarafından implemente edildigi taktirde afterPropertiesSet() metodu
- init-method ile tanımlı bir sınıf metodu

Nesne imha metotları şu sıraya göre koşturulur:

- @PreDestroy anotasyonu ile işaretli metotlar
- DisposableBean sınıf tarafından implemente edildigi taktirde destroy() metodu
- destroy-method ile tanımlı bir sınıf metodu

Örneğin InitializingBean interface sınıfı implemente ettik ve aynı zamanda init-method ile afterPropertiesSet() metodunu tanımladıysak, bu metot Spring tarafından sadece bir kez koşturulur. Seçilen yaşam döngüsü yöntemine göre kullanılan metot ismi değişik olduğu taktirde Spring bu metotları yukarıda belirttiğim sıraya göre koşturacaktır.

Kullanım Fazı

Resim 3.3 e tekrar baktığımızda bir Spring uygulamasının sahip olduğu yaşam döngüsünün ikinci fazının kullanım (usage) fazı olduğunu görmekteyiz. Kurulum fazını bir önceki bölümde detaylı olarak inceledik. Bu bölümde kullanım fazını ve işlevini yakından inceleyeceğiz.

Kullanım fazı getBean() metodunun kullanılmasıyla başlar. Bir rentalService nesnesini şu şekilde edinebiliriz:

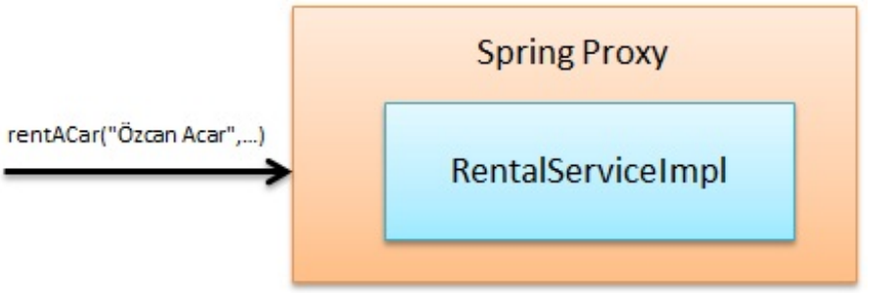
```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("
                        applicationContext.xml");
RentalService rentalService =
    ctx.getBean("rentalService", RentalService.class);
```

Daha önce bahsettiğim gibi bean tanımlamaları Application Context'in (resim 3.5) bir parçası olan BeanFactory bünyesinde yer alır. getBean() metodu aracılığı ile bir nesneyi edinmek istediğimizde, bu işleme cevap verecek olan

BeanFactory'dir. getBean() metodu aracılığı ile istediğimiz türde bir nesneyi edindikten sonra, bu nesnenin herhangi bir metodunu koşturabiliriz. Burada kullanılan nesnenin new operatörü ile oluşturulmuş bir nesneden farkı yoktur. Kullanıcı doğrudan nesnenin herhangi bir metodunu koşturabilir.

Resim 3.8

getBean() üzerinden edindiğimiz nesneye Spring sunucusu tarafından bir vekil nesne atanmış olabilir. Vekil nesne kullanılması durumunda işlem türü farklı olacaktır. Vekil (Proxy) tasarım şablonuna göre bir nesneye bir vekil atanarak, nesne ile doğrudan etkileşimin önüne geçilebilir. Hangi durumlarda bunun faydalı olabileceğini bir sonraki bölümde yakından inceleyeceğiz.



Resim 3.9

Vekil Nesne Oluşturma

Spring bünyesinde tanımlı herhangi bir nesnenin herhangi bir metodunu koşturmadan önce yapmak istediğimiz başka bir işlem olabilir. Örneğin rentalService isimli nesnenin rentACar() metodunun çalışma süresini ölçmek istediğimizi düşünelim. Bu durumda bir vekil nesne oluşturarak, bu vekilin metodun çalışma zamanını ölçmesini sağlayabiliriz. rentACar() metodu koşturulmadan önce bu vekil devreye girerek zamanı tutacak ve metodun son bulmasıyla metodun çalışma süresini ekranda görüntüleyecektir. Bu örneği nasıl kodlayabileceğimizi yakından inceleyelim.

Java'da bir dinamik vekil (dynamic proxy) oluşturabilmek için java.lang.reflect.InvocationHandler interface sınıfını implemente eden bir sınıf oluşturmamız gerekmektedir. Böyle bir implementasyon kod 3.5 de yer almaktadır.

```
Kod 3.5 - LogHandler

package com.kurumsaljava.spring.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import com.kurumsaljava.spring.RentalService;

public class LogHandler implements InvocationHandler {

    private RentalService rentalService;

    public LogHandler(RentalService service) {
        super();
        this.rentalService = service;
    }

    @Override
    public Object invoke(Object proxy,
            Method method, Object[] args)
            throws Throwable {

        long start = System.nanoTime();
        System.out.println("Invoking method " + method.getName());

        Object result = method.invoke(this.rentalService, args);
        long end = System.nanoTime();
        System.out.println("Done in " + (end - start) + " ns.");

        return result;
    }
}
```

Vekillik işlemlerini yapacak olan LogHandler sınıfının vekil olacağı sınıfı ve bu sınıfın nesnesini tanıması gerekmektedir. Bu amaçla RentalService sınıfından bir nesne, bu Spring tarafından oluşturulan rentalService nesnesi olacaktır, sınıf konstrüktörü üzerinden LogHandler sınıfının rentalService isimli sınıf değişkenine atanır. Vekillik işlemleri invoke() metodu bünyesinde gerçekleşir. Bu metot bünyesinde start değişkeninde başlangıç zamanı ve end değişkeninde bitiş zamanı tutulur. Bu iki değişken arasındaki kod bloğunda Java Reflection kullanılarak RentalService sınıfının rentACar() metodu koşturulur. invoke() metoduna giren method parametresi koşturulacak olan metodun ismini, args parametresi metot parametrelerini ihtiva eder. Result rentACar() metodunu geri verdiği değerdir, yani Result sınıfının bir nesnesi.

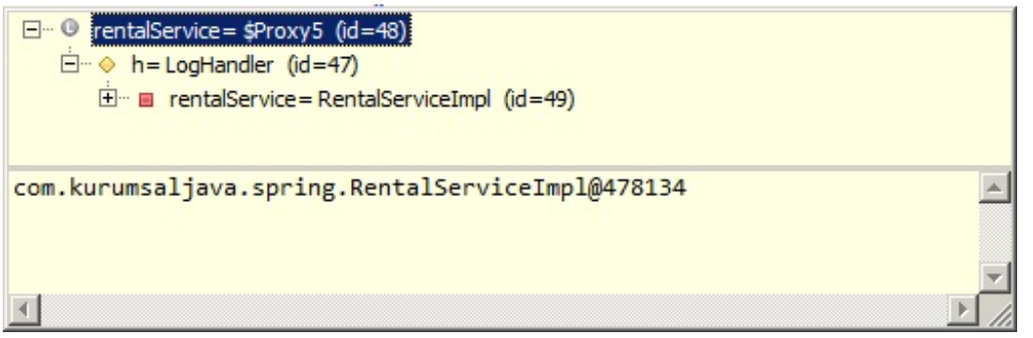
Kullanıcı olarak bizim değil, Spring'in bir vekil nesne oluşturması gerekmektedir. Kullanıcı olarak biz Spring sunucusundan (container) bir nesne talep ettiğimizde, Spring talep edilen nesne yerine, vekil nesneyi bize verebilmelidir. Bu amaçla RentalServiceImpl bünyesinde BeanPostProcessor interface sınıfını implemente etmemiz gerekmektedir. Daha öncedeki örneklerde gördüğümüz gibi BeanPostProcessor interface sınıfının postProcessBeforeInitialization() metodu nesne oluşturma işlemi tamamlandıktan ve set() metotları aracılığı ile gerekli bağımlılıklar enjekte edildikten sonra koşturulur. Bu mekanizmayı kullanıp, oluşturulan nesneye bir vekil nesne atayarak, kullanıcının vekil nesnesini kullanmasını sağlayabiliriz. Böyle bir implementasyon kod 3.6 da yer almaktadır.

```
Kod 3.6 - RentalServiceImpl

package com.kurumsaljava.spring;

import java.lang.reflect.Proxy;
import java.util.Date;
import java.util.Properties;
import org.springframework.beans.factory.config.BeanPostProcessor;
import com.kurumsaljava.spring.proxy.LogHandler;

public class RentalServiceImpl implements RentalService,
            BeanPostProcessor {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;
    private Properties carProperty;

    public Properties getCarProperty() {
        return carProperty;
    }

    public void setCarProperty(Properties carProperty) {
        this.carProperty = carProperty;
    }

    public RentalServiceImpl() {
    }

    public RentalServiceImpl(CustomerRepository customerRepository,
            RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }
```

```
    @Override
    public Rental rentACar(String customerName, Car car,
            Date begin, Date end) {
        Customer customer =
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }

    public CustomerRepository getCustomerRepository() {
        return customerRepository;
    }

    public void setCustomerRepository(CustomerRepository
            customerRepository) {
        this.customerRepository = customerRepository;
    }

    public RentalRepository getRentalRepository() {
        return rentalRepository;
    }

    public void setRentalRepository(RentalRepository
                rentalRepository) {
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Object postProcessBeforeInitialization(
                Object bean, String beanName) {

        if (bean instanceof RentalService) {
            Class[] interfaces =
                    new Class[] { RentalService.class };
            LogHandler logHandler =
                    new LogHandler((RentalService) bean);
            ClassLoader loader =
                    bean.getClass().getClassLoader();
            RentalService service =
                    (RentalService) Proxy.newProxyInstance(
```

```
                    loader, interfaces, logHandler);
            return service;
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(
                    Object bean, String beanName) {
        return bean;
    }
}
```

postProcessBeforeInitialization() metodunda yer alan kod Proxy.newProxyInstance() üzerinden bir vekil nesne oluşturarak, rentalService nesnesini yerine bu nesneyi geri verir. Resim 3.10 da görüldüğü gibi getBean() aracılığı ile edindiğimiz rentalService nesnesi aslında bir vekil nesnedir. Nesnenin veri tipi RentalServiceImpl değil $Proxy5'dir. Edindiğimiz vekil nesne bünyesinde LogHandler ve LogHandler bünyesinde rentalService nesnesi yer almaktadır. Böylece kullanıcı olarak rentalService.rentACar() metodunu koşturma isteğimiz öncelikle vekil nesneye yönlendirilr. Vekil nesne LogHandler sınıfının invoke() metodu aracılığı ile rentalService nesnesinin rentAcar() metodunu koşturur ve neticeyi kullanıcıya geri verir.



Resim 3.10

Vekil nesne oluşturma işlemini postProcessBeforeInitialization() metodu yerine postProcessAfterInitialization() metodu bünyesinde de yapabilirdik. Burada önemli olan Spring'in nesneyi kullanım için tam teşekküllü hale getirmesinin akabinde devreye girerek, nesnenin kullanıcıya geri verilmesinden bir adım önce vekil nesneyi oluşturabilmektir.

Buraya kadar yaptıklarımız aslında Aspect Oriented Programming (AOP)'dir. İşletme mantığı ile doğrudan ilişkisi olmayan logging, zaman ölçümü, transaksiyon yönetimi gibi işlemlerin gerektirdiği kod, işletme mantığının yer

aldığı metotların gereksiz yere büyümesine neden olur. Örneğin rentalService nesnesinin rentACar() metodunun ne kadar zamanda koşturulduğunu öğrenmek için kod 3.5 de yer alan zaman ölçme kodunu rentACar() metodunu eklememiz gerekirdi. Zaman ölçmek için kullandığımız kodun işletme mantığı yani bir aracın kiraya verilmesi için gerekli kod birimi ile doğrudan ilişkisi yoktur. Bu sebepten dolayı bu kodun AOP tarzı bir mekanizma ile metodun dışına çekilmesi, rentAcar() metodunun sadece işletme mantığını ihtiva etmesini ve kodun daha okunabilir ve genişletilebilir hale gelmesini sağlayacaktır.

AOP Spring çatışında önemli yeri olan bir metotdur. İlerleyen bölümlerde Spring kullanılarak AOP tarzı programlamanın nasıl yapıldığını yakından inceleyeceğiz.

İmha Fazı

Bir Spring uygulamasının sahip olduğu yaşam döngüsünün son safhası imha (destruction) fazıdır. Bu fazda sunucu bünyesinde yer alan tüm tekil (singleton) nesneler imha edilir. Akabinde Application Context sonlandırılır. Bu Spring uygulamasının son bulduğu anlamına gelmektedir. Bu süreç içinde nesnelerin imhasını kontrol edebilmek için konfigürasyon dosyasında destroy-method, sınıf bünyesinde standart bir Java anotasyonu (JSR-250) olan @PreDestroy anotasyonu kullanılabilir.

Bir Spring uygulamasını şu şekilde sonlandırabiliriz:

```
ConfigurableApplicationContext ctx =
        new ClassPathXmlApplicationContext("
                        applicationContext.xml");
...
ctx.close();
```

close() metodu sunucu içinde yer alan tüm tekil nesneleri imha etmek için harekete geçecektir. Tekil nesneler gözlerini yaşama yummadan önce son bir adımda ellerinde tuttukları kaynakları bırakmaya ya da diğer türde temizlik işlemi yapmaya zorlanabilir. Örneğin rentalService nesnesinin bir veri tabanı bağlantısını (java.sql.Connection) elinde tuttuğunu düşünecek olursak, bu nesne imha edilmeden önce bu bağlantının kapatılmasını sağlayabiliriz. Bu amaçla releaseResources() isminde gerekli temizlik işlemini yapan bir metodumuzun olduğunu düşünelim.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl"
   destroy-method="releaseResources">
    <property name="customerRepository"
                ref="customerRepository" />
    <property name="rentalRepository"
                ref="rentalRepository" />
</bean>
```

destroy-method element özelliğinden faydalanarak, nesnenin imhası öncesi koşturulması gereken metodu tanımlayabiliriz. Aynı işlemi @PreDestroy anotasyonu yardımı ile şu şekilde yapmak mümkün:

```
public class RentalServiceImpl{

    @PreDestroy
    public void releaseResources() {
        System.out.println("releaseResources() called");

    }
}
```

Temizlik işlemlerini yapan metotları sınıf bazında değil de, her sınıf için geçerli olacak şekilde tanımlamak istersek konfigürasyon dosyasında default-destroy-method element özelliğini şu şekilde kullanabiliriz:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-destroy-method="releaseResources">
...
</beans>
```

@PreDestroy ve destroy-method kullanılarak tanımlanan metotların koşturulabilmesi için oluşturulan Application Context'in close() metodu ile sonlandırılması gerekmektedir. close() metodu kullanılmadığı taktirde @PreDestroy ve destroy-method ile tanımlanan metotlar koşturulmayacaktır.

@PreDestroy ve destroy-method'a alternatif olarak DisposableBean interface sınıfı kullanılabilir. Bu interface sınıfını implemente eden sınıfların destroy isminde bir metodu bulunur. Spring imha fazında bu metotları koşturarak, nesne için gerekli temizlik işlemlerinin yapılmasını sağlar.

```
public class RentalServiceImpl implements DisposableBean{

    @Override
```

```
    public void destroy() throws Exception {
        System.out.println("destroy() called");
    }
}
```

Özetleyecek olursak, uygulamayı sonlandırırken nesnelerin ellerinde tuttukları kaynakları bırakmalarını sağlamak için üç değişik yöntem kullanabilir. Bunlar DisposableBean interface sınıfının implemente edilmesi, detroy-method ve @PreDestroy anotasyonunun kullanımıdır. Bu yöntemlerden hangisini kullanmalıyım sorusu aklınıza gelmiş olabilir. DisposableBean interface sınıfının kullanımı kodu Spring çatısına bağımlı kılmaktadır. Bu yüzden kullanılmamalıdır. Eğer bir sınıfın kaynak koduna erişemiyorsanız init-method kullanılabilir. Kaynak koda sahipseniz @PreDestroy anotasyonunu kullanabilirsiniz.

3. Bölüm Soruları

3.1 Herhangi bir isim alanında bulunan bir Spring konfigürasyon elementi konfigürasyon dosyasında nasıl kullanılır?

3.2 Spring uygulamalarında yaşam döngüsü kaç fazdan oluşur?

3.3 @PostConstruct anotasyonu ne amaçla kullanılır?

3.4 Vekil nesne ne amaçla kullanılır?

3.5 Bir Spring uygulaması son bulurken bir tekil nesne elinde tuttuğu kaynakları bırakmaya nasıl zorlanır?

3.6 İmha fazı nasıl başlatılır?

4. Bölüm

Konfigürasyon Yönetimi

Bean Tanımlama ve Kalıtım

Java'da ortak özellikleri olan sınıflar için bir soyut (abstract) sınıf oluşturularak, ortak özellikler bu sınıfta toplanır ve diğer sınıfların soyut sınıfı genişleterek (extend), ortak özellikleri kullanmaları sağlanır. Java ve diğer nesneye yönelik dillerde bu kalıtım (inheritance) olarak bilinir.

Sınıfların ortak özellikleri olacağı gibi, Spring bean tanımlamalarının da ortak özellikleri olabilir. Spring'in sunduğu bean kalıtım mekanizmasından faydalanarak, Java'da olduğu gibi ortak özellikleri merkezi bir yerde toplayabilir ve tekrar kullanabiliriz (reuse).

```
Kod 4.1 - Car

package com.kurumsaljava.spring;

public class Car {

    public static enum CarTypeEnum {
        BUS, CAR, SPORT
    }

    private CarTypeEnum type;
    private String brand;
    private String model;

    public Car() {
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public CarTypeEnum getType() {
```

```
        return type;
    }

    public void setType(CarTypeEnum type) {
        this.type = type;
    }
}
```

Kod 4.1 de yer alan Car sınıfı bir aracı temsil etmektedir. Her aracın bir çeşidi (type), bir markası (brand) ve bir modeli (model) vardır. Değişik tipte araçlar oluşturabilmek için CarTypeEnum enum sınıfı kullanılmaktadır. Binek oto CAR, otobüs BUS ve spor abalar SPORT olarak tanımlanmıştır.

Clio, Fiesta ve Audia A5 model araçları applicationContext.xml bünyesinde şu şekilde tanımlayabilirdik:

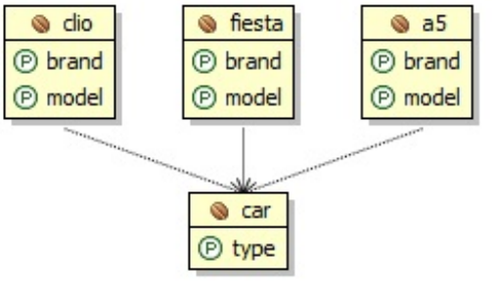
```
Kod 4.2 - applicationContext.xml

<bean id="clio" class="com.kurumsaljava.spring.Car">
    <property name="type" value="CAR"></property>
    <property name="brand" value="Renault"></property>
    <property name="model" value="clio"></property>
</bean>

<bean id="fiesta" class="com.kurumsaljava.spring.Car">
    <property name="type" value="CAR"></property>
    <property name="brand" value="Ford"></property>
    <property name="model" value="clio"></property>
</bean>

<bean id="a5" class="com.kurumsaljava.spring.Car">
    <property name="type" value="CAR"></property>
    <property name="brand" value="Audi"></property>
    <property name="model" value="a5"></property>
</bean>
```

Böyle bir tanımlamada kendisini tekrar eden birden fazla bilgi mevcuttur. Görebildiniz mi? Hangi çeşit araba nesnesi oluşturmak istediğimizi type değişkeni üzerinden belirliyoruz. Type tanımladığımız üç aracın ortak özelliği olarak görünüyor. Ayrıca üç bean tanımlaması da class element özellgi aracılığı ile Car sınıfını kullanmakta. DRY (Dont Repeat Yourself - Kendini Tekrarlama) prensibinden yola çıkarak, type değişkenini ve class element özelliğini bir kereye mahsus tanımlayıp, diğer bean tanımlamalarını sadeleştirebiliriz.

```
Kod 4.3 - applicationContext.xml
```

```
<bean id="car" class="com.kurumsaljava.spring.Car">
    <property name="type" value="CAR"></property>
</bean>

<bean id="clio" parent="car">
    <property name="brand" value="Renault"></property>
    <property name="model" value="clio"></property>
</bean>

<bean id="fiesta" parent="car">
    <property name="brand" value="Ford"></property>
    <property name="model" value="clio"></property>
</bean>

<bean id="a5" parent="car">
    <property name="brand" value="Audi"></property>
    <property name="model" value="a5"></property>
</bean>
```

Kod 4.3 de yer alan örnekte car isminde bir bean tanımlanmıştır. Bu tanımlamayı Java'da bir üstsınıf olarak düşünebiliriz. Bu bean sadece aracın çeşidine işaret eden type değişkenine bir değer atamaktadır. Bu değer CarTypeEnum enum sınıfında tanımlanmış olan değerlerden herhangi birisi olabilir. Clio ve diğer bean tanımlamaları parent element özelliği aracılığı ile car bean tanımlamasını genişletmekte ve type değişkeninin değerine car bean tanımlaması üzerinden ulaşmaktadırlar.



Resim 4.1

Parent element özelliği Java dilindeki extend kelimesine eş gelmektedir. Bu özelliği kullanan bir bean tanımlaması, mevcut bir bean tanımlamasını genişletir ve o bean tanımlamasının sahip olduğu değerleri miras olarak edinir. Bunlar bean tanımlamasında kullanılan class element özelliğindeki değer ve property elementleri ile tanımlanan değişkenlerdir.

```
Kod 4.4 - Main
```

```
package com.kurumsaljava.spring;

import org.springframework.context.
            ConfigurableApplicationContext;
import org.springframework.context.support.
            ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                    "applicationContext.xml");
        Car car = ctx.getBean("clio", Car.class);
        System.out.println(car.getBrand());
        System.out.println(car.getModel());
        System.out.println(car.getType());
    }
}
```

Kod 4.4 de yer alan main() metodunu koşturduğumuzda ekran çıktısı şu şekilde olacaktır:

```
Brand: Renault
Model: clio
Type: CAR
```

Clio bean tanımlamasında type değişkenine değer atamadığımız halde, parent tanımlaması ile car bean tanımlamasını genişlettiğimiz için System.out.println(car.getType()) ekranda CAR değerini görüntüleyecektir.

Aynı şekilde ctx.getBean("car", Car.class) ile tanımlı olan car nesnesini edinebiliriz.

```
Car car = ctx.getBean("car", Car.class);
System.out.println(car.getBrand());
System.out.println(car.getModel());
System.out.println(car.getType());
```

Ekran çıktısı şu şekilde olacaktır:

```
Brand: null
Model: null
Type: CAR
```

Car bean tanımlaması brand ve model değişkenlerine değer atamadığı için ekranda null bilgisi yer almaktadır. Bu hali ile car nesnesinin kullanımı bir şey ifade etmemektedir, çünkü nesne gerekli bilgileri ihtiva etmemektedir. Car nesnesinin uygulama bünyesinde kullanımını engellemek ve sadece bir üst bean tanımlaması olarak kullanılmasını sağlamak için bu bean tanımlamasını soyut (abstract) hale getirebiliriz.

```
<bean id="car"
        class="com.kurumsaljava.spring.Car" abstract="true">
    <property name="type" value="CAR"></property>
</bean>
```

Abstract element özelliğine true değeri atandığı taktirde, bir bean tanımlaması soyut olarak kabul edilir. Soyut olan bean tanımlamalarından nesne oluşturmak mümkün değildir. Bilindigi üzere Java'da da bir sınıf abstract kelimesi ile tanımlandığında, bu sınıftan nesneler oluşturmak mümkün değildir. Soyut olan sınıf sadece üstsınıf olarak kullanılabilir. Aynı şekilde soyut olan bir bean tanımlaması sadece üst bean tanımlaması olarak kullanilabilir.

Bu değişikliğin ardından getBean() ile bir car nesnesi edinmek istediğimizde, aşağıdaki hata mesajı ile karşılaşırız:

```
Error creating bean with name 'car': Bean definition is abstract
```

parent element özelliğini kullanarak genişlettiğimiz bean tanımlamasının özelliklerini miras almakla birlikte, bu özelliklerini hepsini yeniden tanımlayabiliriz.

```
Kod 4.5 - applicationContext.xml

<bean id="car" class="com.kurumsaljava.spring.Car">
    <property name="type" value="CAR"></property>
</bean>

<bean id="man"
        class="com.kurumsaljava.spring.ManBus" parent="car">
    <property name="brand" value="MAN"></property>
    <property name="model" value="Neoplan"></property>
    <property name="type" value="BUS"></property>
</bean>
```

Kod 4.5 de yer alan man bean tanımlaması parent ile car bean tanımlamasını genişletmektedir. man bean tanımlaması class element özelliğini ve type

değişkenini miras olarak aldığı halde, class ile yeni bir sınıf (ManBus) tanımlamaktadır ve type değişkenine başka bir değer atamaktadır. Spring bean tanımlamaları herhangi bir bean tanımlamasını genişletebilirken, miras edindikleri tüm değerleri yeniden tanımlama kabiliyetine sahiptirler.

Dahili Bean Tanımlamaları (Inner Beans)

Tanımlanan her bean ile Spring konfigürasyon dosyası biraz daha büyür. Çoğu bean tanımlaması sadece bir bean tanımlaması bünyesinde ref element özelliği aracılığı ile kullanılmak üzere tanımlanır. Böyle bir örnek kod 4.6 da yer almaktadır.

```
Kod 4.6 - applicationContext.xml

<bean id="clio" parent="car">
    <property name="brand" value="Renault"></property>
    <property name="model" value="clio"></property>
    <property name="factory" ref="clioCarFactory"></property>
</bean>

<bean id="clioCarFactory"
        class="com.kurumsaljava.spring.ClioCarFactory" />
```

Kod 4.6 da yer alan clioCarFactory bean tanımlaması clio bean tanımlaması tarafından kullanılmaktadır. Clio bean tanımlamasının factory isminde bir değişkeni mevcuttur. Bu değişken üzerinden araç ile üretildiği fabrika arasında ilişki oluşturulmaktadır.

ClioCarFactory sadece clio tarafından kullanılabilecek bir bean olarak görünüyor. Clio harici başka bir bean tanımlamasının clioCarFactory bean tanımlamasını kullanacağını zannetmiyorum. O zaman clioCarFactory bean tanımlamasını bir dahili bean tanımlaması haline getirebiliriz.

```
Kod 4.7 - applicationContext.xml

<bean id="clio" parent="car">
    <property name="brand" value="Renault"></property>
    <property name="model" value="clio"></property>
    <property name="factory">
        <bean
            class="com.kurumsaljava.spring.ClioCarFactory" />
    </property>
</bean>
```

Dahili bean tanımlamalarının id ya da name element özelliği yoktur, yani bu tür bean tanımlamalarının isimleri bulunmaz. Bu yüzden dahili bean tanımlamaları anonimdir. Anonim olduklarından dolayı dahil edildikleri bean haricinde başka bir bean tanımlaması tarafından kullanılamazlar.

Konfigürasyon Dosyalarının İmport Edilmesi

Bir Spring konfigürasyon dosyası zaman içinde yapılan yeni bean tanımlamaları ile genişler ve büyük bir hacme sahip olabilir. Bu hal konfigürasyon dosyasının bakımını ve okunabilirliğini zorlaştırabilir. Bunun önüne geçmenin bir yolu, bean tanımlamalarını mantıksal gruplara ayırarak, konfigürasyon dosyasını parçalara bölmektir. Bu parçaları daha sonra import elementi ile bir araya getirip, sadece bir konfigürasyon dosyası varmış gibi kullanmak mümkündür.

```
Kod 4.8 - applicationContext.xml

<beans>
    <bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
        <property name="customerRepository"
                    ref="customerRepository" />
        <property name="rentalRepository"
                    ref="rentalRepository" />
    </bean>

    <bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl" />

    <bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl" />

    <bean id="clio" parent="car">
        <property name="brand" value="Renault"></property>
        <property name="model" value="clio"></property>
        <property name="factory">
            <bean
            class="com.kurumsaljava.spring.ClioCarFactory" />
        </property>
    </bean>

    <bean id="fiesta" parent="car">
        <property name="brand" value="Ford"></property>
```

```
        <property name="model" value="clio"></property>
    </bean>

    <bean id="a5" parent="car">
        <property name="brand" value="Audi"></property>
        <property name="model" value="a5"></property>
    </bean>

    <bean id="car" class="com.kurumsaljava.spring.Car">
        <property name="type" value="CAR"></property>
    </bean>

    <bean id="man"
        class="com.kurumsaljava.spring.ManBus" parent="car">
        <property name="brand" value="MAN"></property>
        <property name="model" value="Neoplan"></property>
        <property name="type" value="BUS"></property>
    </bean>
</beans>
```

Kod 4.8 de yer alan konfigürasyon dosyası servis katmanı (rentalService), veri katmanı (rentalRepository, customerRepository) ve domain model sınıflarını (car, clio, fiesta, a5, man) ihtiva etmektedir. Bu konfigürasyon dosyasının zaman içinde yapılacak yeni bean tanımlamaları ile okunamaz hale geleceği aşikardır. Bu bean tanımlamalarını servis katmanı, veri katmanı ve model sınıfları şeklinde mantıksal gruplara şu şekilde ayırabiliriz:

```
Kod 4.9 - service.xml

<beans>
    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
        <property name="customerRepository"
                    ref="customerRepository" />
        <property name="rentalRepository"
                    ref="rentalRepository" />
    </bean>
</beans>

Kod 4.10 - repository.xml

<beans>
    <bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl" />

    <bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl" />
```

```
</beans>

Kod 4.11 - model.xml

<beans>
    <bean id="clio" parent="car">
        <property name="brand" value="Renault"></property>
        <property name="model" value="clio"></property>
        <property name="factory">
            <bean
                class="com.kurumsaljava.spring.ClioCarFactory" />
        </property>
    </bean>

    <bean id="fiesta" parent="car">
        <property name="brand" value="Ford"></property>
        <property name="model" value="clio"></property>
    </bean>

    <bean id="a5" parent="car">
        <property name="brand" value="Audi"></property>
        <property name="model" value="a5"></property>
    </bean>

    <bean id="car" class="com.kurumsaljava.spring.Car">
        <property name="type" value="CAR"></property>
    </bean>

    <bean id="man"
        class="com.kurumsaljava.spring.ManBus"
                        parent="car">
        <property name="brand" value="MAN"></property>
        <property name="model" value="Neoplan"></property>
        <property name="type" value="BUS"></property>
    </bean>
</beans>

Kod 4.12 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <import resource="service.xml"/>
    <import resource="repository.xml"/>
    <import resource="model.xml"/>
```

```
</beans>
```

Kod 4.12 de yer alan applicationContext.xml, oluşturduğumuz diğer konfigürasyon dosyalarını import etmektedir.

I mport elementinin kullanımı ile sağlanan avantajları şu şekilde sıralayabiliriz:

- Konfigürasyon dosyasını parçalara bölerek, bean tanımlamaları için mantıksal gruplar oluşturmayı mümkün kılar.
- Mantıksal gruplanan bean tanımlamalarının değişik modüller tarafından tekrar kullanımını sağlar.
- İkinci bölümde incelediğimiz çoklu XML konfigürasyon kullanımını tamamlayıcı niteliktedir.
- Konfigürasyon dosyalarının bakımını ve okunmasını kolaylaştırır.

P İsim Alanı

Bağımlılıkların enjekte edilmesi için şimdiye kadar property elementinden ve ref element özelliğinden faydalandık. Aşağıdaki örnekte rentalService nesnesine customerRepository ve rentalRepository nesneleri enjekte edilmektedir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
                    ref="customerRepository" />
    <property name="rentalRepository"
                    ref="rentalRepository" />
    <property name="user" value="admin">
</bean>
```

P isim alanı property elementi ve ref element özelliğinin daha kısa tutulmalarını sağlamak için oluşturulmuştur. Örneğin bu isim alanını konfigürasyon dosyamıza dahil ettikten sonra, rentalService için gerekli bean tanımlamasını şu şekilde yazabiliriz:

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
    p:customerRepository-ref="customerRepository"
    p:rentalRepository-ref="rentalRepository"
    p:user="admin"/>
```

P isim alanının kendine has bir XSD dosyası yoktur. Bu yüzden

```
xmlns:p="http://www.springframework.org/schema/p"
```

şeklinde konfigürasyon dosyasında bir tanımlama yeterli olacaktır.

C İsim Alanı

P isim alanı property elementi ve ref element özelliğini daha kısa yazmak için kullanılırken, C isim alanı constructor-arg elementini daha kısa yazmak için kullanılmaktadır. RentalService bean tanımlamasını daha önceki örneklerimizde şu şekilde yapmıştık:

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository" />
    <constructor-arg ref="rentalRepository" />
</bean>
```

Böyle bir bean tanımlamasında bir rentalService nesnesi sınıf konstrüktörüne customerRepository ve rentalRepository nesneleri enjekte edilerek oluşturulur. C isim alanını kullanarak, bu tanımlamayı aşağıdaki şekilde yazabiliriz.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
    c:customerRepository-ref="customerRepository"
    c:rentalRepository-ref="rentalRepository"/>
```

C isim alanının kendine has bir XSD dosyası yoktur. Bu yüzden

```
xmlns:c="http://www.springframework.org/schema/c"
```

şeklinde konfigürasyon dosyasında bir tanımlama yeterli olacaktır.

Util İsim Alanı

Util isim alanı aşağıda yer alan elementlerden oluşur ve set, map, list, sabit değer (constants) ve buna benzer konfigürasyon işlemlerini kolaylaştırmak için kullanılır.

<util:constant/>

Nesnelere static değişken değerleri enjekte etmek için util:constant elementi

kullanılır. Aşağıdaki örnekte rentalService nesnesinin domain isimli değişkenine DomainNames sınıfında yer alan ve sabit bir değer olan PROD_DOMAIN enjekte edilmektedir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository" ref="customerRepository" />
    <property name="rentalRepository" ref="rentalRepository" />
    <property name="domain">
        <util:constant
            static-field="com.kurumsaljava.spring.
                                DomainNames.PROD_DOMAIN" />
    </property>
</bean>


public class DomainNames {
    public static final String TEST_DOMAIN ="com.carrental.test";
    public static final String REF_DOMAIN ="com.carrental.ref";
    public static final String PROD_DOMAIN ="com.carrental";
}
```

Aynı şekilde bir enum sınıfında yer alan değerler util-constant kullanılarak nesnenin aynı enum veri tipinden olan sınıf değişkenine enjekte edilebilir.

<util:property-path/>

Spring bean olarak tanımlanmış bir nesnenin herhangi bir değişkenindeki değere util:property-path ile erişmek ve bu değeri başka bir nesneye enjekte etmek mümkündür. Aşağıda yer alan örnekte rentalService nesnesi domain isminde bir değişkene sahiptir. Bu değişkenin sahip olduğu değer (rentacar.com) util:property-path yardımı ile rentalRepository nesnesinin domain isimli değişkenine enjekte edilmektedir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
                ref="customerRepository" />
    <property name="rentalRepository"
                ref="rentalRepository" />
    <property name="domain" value="rentacar.com"/>
</bean>

<bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl"
                        scope="prototype">
```

```
    <property name="domain">
         <util:property-path path="rentalService.domain"/>
    </property>
</bean>
```

<util:properties/>

Değer dosyalarını yüklemek ve enjekte etmek için kullanılır. Aşağıda yer alan örnekte oracle.properties dosyasını temsil eden java.util.Properties nesnesi oracleDataSource sınıfının dbConfiguration isimli değişkenine enjekte edilmektedir. dbConfiguration.getProperty() metodu aracılığı ile oracle.properties dosyasında yer alan anahtar ve değerlerine ulaşılabilir.

```
# applicationContext.xml
<util:properties id="dbConfiguration"
                location="oracle.properties" />

<bean id="oracleDataSource"
            class="com.kurumsaljava.spring.OracleDataSource">
    <property name="dbConfiguration"
                     ref="dbConfiguration" />
</bean>

# oracle.properties
db.name=RENTACAR
db.user=rent
db.password=car

# OracleDataSource.java
import java.util.Properties;
public class OracleDataSource implements DataSource {

    private Properties dbConfiguration;

    public Properties getDbConfiguration() {
        return dbConfiguration;
    }

    public void setDbConfiguration(Properties dbConfiguration) {
        this.dbConfiguration = dbConfiguration;
    }
}

# Main.java
OracleDataSource oracleDataSource =
        ctx.getBean("oracleDataSource", OracleDataSource.class);
System.out.println(oracleDataSource.
```

```
            getDbConfiguration().getProperty("db.name"));
```

<util:list/>

Nesnelerin bünyelerinde taşıdıkları java.util.List tipi yapılara listeler halinde veri enjekte etmek için kullanılır. Aşağıda yer alan örnekte rentalService bünyesinde yer alan cars isimli listeye a5 ve clio nesneleri enjekte edilmektedir.

```
<util:list id="cars">
        <ref bean="a5" />
        <ref bean="clio" />
</util:list>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
    <property name="cars" ref="cars"/>
</bean>
```

Kullanılan liste tipi list-class element özelliği aracılığı ile şu şekilde tayin edilebilir:

```
<util:list id="cars" list-class="java.util.ArrayList">
        <ref bean="a5" />
        <ref bean="clio" />
</util:list>
```

List-class ile kullanılmak istenilen liste tipi belirtilmediği taktirde Spring kullanılacak liste tipini kendisi belirler.

<util:map/>

Nesnelerin bünyelerinde taşıdıkları java.util.Map tipi yapılara anahtar/kelime şeklinde veri enjekte etmek için kullanılır. Aşağıda yer alan örnekte rentalService bünyesinde yer alan cars isimli değişkene a5 anahtarı altında a5 nesnesi ve clio anahtarı altında clio nesneleri enjekte edilmektedir.

```
<util:map id="cars">
    <entry key="a5" value-ref="a5"></entry>
    <entry key="clio" value-ref="clio"></entry>
</util:map>
```

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
    <property name="cars" ref="cars"/>
</bean>
```

Kullanılan map tipi map-class element özelliği aracılığı ile şu şekilde tayin edilebilir:

```
<util:map id="cars" map-class="java.util.TreeMap">
    <entry key="a5" value-ref="a5"></entry>
    <entry key="clio" value-ref="clio"></entry>
</util:map>
```

Map-class ile kullanılmak istenilen map tipi belirtilmediği taktirde Spring kullanılacak map tipini kendisi belirler.

<util:set/>

Nesnelerin bünyelerinde taşındıkları java.util.Set tipi yapılara listeler halinde veri enjekte etmek için kullanılır. Aşağıda yer alan örnekte rentalService bünyesinde yer alan cars isimli sete a5 ve clio nesneleri enjekte edilmektedir.

```
<util:set id="cars">
    <ref bean="a5" />
    <ref bean="clio" />
</util:set>

<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
    <property name="cars" ref="cars"/>
</bean>
```

Kullanılan set tipi set-class element özelliği aracılığı ile şu şekilde tayin edilebilir:

```
<util:set id="cars" set-class="java.util.HashSet">
    <ref bean="a5" />
    <ref bean="clio" />
```

```
</util:set>
```

Set-class ile kullanılmak istenilen set tipi belirtilmediği taktirde Spring kullanılacak set tipini kendisi belirler.

Spring Expression Language (SpEL)

Bağımlılıkların enjekte edilebilmesi için bu değerlerin uygulamanın geliştirildiği esnada bilinmesi, yani bu değerlerin konfigürasyon dosyasında tanımlı olması gerekmektedir. Bunun iyi bir örneği rentalService bean tanımlamasıdır. Bu nesnenin rentalRepository ve customerRepository isimli iki bağımlılığı mevcuttur. Bu iki bağımlılığın enjekte edilebilmesi için bean olarak konfigürasyon dosyasında tanımlı olmaları gerekmektedir. Aksi taktirde bu nesneleri bağımlılık olarak enjekte etmemiz mümkün değildir.

Her zaman enjekte edilecek degerleri önceden tanımlamamız mümkün olmayabilir. Bazı değerlerin uygulamanın çalışır durumda olmasıyla ya da bazı şartlara bağımlı olarak şekil almaları söz konusu olabilir. Uygulama çalışırken konfigürasyon dosyasında değişiklik yapmamaz mümkün değildir. Ama uygulama çalışırken bazı değerlerin enjekte edilmeden önce tanımlanabilir ya da hesaplanabilir bir yapıda olması gerekebilir. Bu gibi durumlar için Spring Expression Language (kısaca SpEL) geliştirilmiştir.

SpEL'i kullanabileceğimiz alanlar şunlardır:

- Regular Expression (Regex) kullanımı
- Matematiksel ve mantıksal işlemler
- Metot koşturulması
- Nesne değişkenlerine erişim
- Nesnelere isimleri üzerinden erişim
- Liste işlemleri

SpEL'i nasıl kullanabileceğimizi birkaç örnek üzerinde yakından inceleyelim. Aşağıda yer alan ilk örnekte threadNumber değişkenine 100 değeri atanmaktadır.

```
<property name="threadNumber" value="#{100}"/>
```

Bir sonraki örnekte atanan değer 101 olacaktır, çünkü Spring bu SpEL tanımlaması ile 100+1 aritmetik işlemini gerçekleştirir. Toplanan iki değer

threadNumber değişkenine enjekte edilir.

```
<property name="threadNumber" value="#{100+1}"/>
```

SpEL ile Java'da mevcut olan tüm aritmetik operatörleri kullanmak mümkündür. SpEL ile kullanılabilecek operatörler şunlardır:

- aritmetik (+, -, *, /, %, ^)
- mantıksal (and, or, not, |)
- relasyonel (<, >, ==, <=, >=, lt (less than), gt (greater than), eq (equal), le (less than or equal), ge (greater than or equal))

Bir sonraki örnekte message değişkenine Available thread number is 101 değeri enjekte edilir.

```
<property name="message"
        value="Available thread number is #{100+1}"/>
```

Relasyonel işlem sonucu olarak değişkenlere true/false değerleri atanabilir. Bir sonraki örnekte valid değişkenine true değeri atanacaktır, çünkü 100 rakamı 1 rakamından büyüktür.

```
<property name="valid" value="#{100>1}"/>
```

Bir sonraki örnekte mantıksal operatör and kullanılarak iki relasyonel işlem sonunda oluşan değer kombine edilmektedir. Eğer seçilen aracın modeli Clio ise ve (and) aracın ağırlığı 2000'den yüksek ise, valid değişkenine true değeri atanır.

```
<property name="valid"
        value="#{car.model == 'Clio' and obj.weight gt 2000}"/>
```

Aynı şekilde bir değişkene String tipi bir değer atamak mümkündür. Aşağıdaki örnekte customerName değişkenine Özcan Acar değeri atanmaktadır.

```
<property name="customerName" value="#{'Özcan Acar'}"/>
```

String tipi değerlerle çalışırken, bu değerlerin belli bir yapıda olup, olmadıklarını kontrol etmek gerekliliği doğabilir. Bu gibi durumlarda SpEL'in sunduğu matches operatörü ile gerekli kıyaslama yapılabilir. Bir sonraki örnekte kullanıcının e-posta adresinin geçerli olup, olmadığı bir kalıp (pattern) aracılığı ile kontrol edilmektedir. Geçerli bir e-posta adresi bulunduğu taktirde

validEmail değişkenine true değeri enjekte edilir. Aksi taktirde bu değer false olacaktır.

```
<property name="validEmail"
    value="#{customer.email
        matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.
                                    [a-zA-Z]{2,4}'}"/>
```

Bean tanımlamalarına ve ihtiva ettikleri değişkenlere SpEL aracılığı ile erişerek, bu değerleri başka nesnelere enjekte edebiliriz. Bir sonraki örnekte rentalService bünyesinde tanımlı olan domain değişkeninin sahip olduğu değer #{rentalService.domain} kullanılarak rentalRepository nesnesinin domain isimli değişkenine enjekte edilmektedir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="domain" value="rentacar.com" />
</bean>

<bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl"
    scope="prototype">
    <property name="domain"
            value="#{rentalService.domain}" />
</bean>
```

Böyle bir işlemin Java dilindeki karşılığı şöyle olacaktır:

```
RentalRepositoryImpl repository = new RentalRepositoryImpl();
repository.setDomain(rentalService.getDomain());
```

SpEL'i kullanarak mevcut bir nesneyi başka bir nesneye doğrudan enjekte edebiliriz. Bir sonraki örnekte rentalService nesnesine rentalRepository isimli nesne enjekte edilmektedir. Bu ref element özelliğinin kullanımına bir alternatiftir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="rentalRepository"
            value="#{rentalRepository}" />
</bean>

<bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl"/>
```

SpEL yardımı ile herhangi bir nesnenin sahip olduğu bir metodu koşturarak, bu metottan elde edilen değeri bir değişkene enjekte etmek mümkündür. Bir sonraki örnekte rentalService nesnesinin getDomain() metodu koşturularak, bu metottan elde edilen değer rentalRepository nesnesinin domain değişkenine enjekte edilmektedir.

```
<bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl"
    scope="prototype">
    <property name="domain"
            value="#{rentalService.getDomain()}" />
</bean>
```

Eğer elde ettiğimiz değeri küçük harflere çevirmek isteseydik, şu şekilde bir tanımlama yapardık:

```
<bean id="rentalRepository"
        class="com.kurumsaljava.spring.RentalRepositoryImpl"
    scope="prototype">
    <property name="domain"
        value="#{rentalService.getDomain().toLowerCase()}" />
</bean>
```

Eğer rentalService.getDomain() null değerini geri verirse, bir önceki örnekte bir NullPointerException oluşur. Bunu önlemek için ? işareti ile null değeri şu şekilde kontrol edilebilir.

```
    <property name="domain"
        value="#{rentalService.getDomain()?.toLowerCase()}" />
```

Soru işaretinin kullanımı bir NullPointerException oluşumunu engeller, çünkü bu operatör sol tarafında yer alan ifadenin null olup olmadığını kontrol eder. Değerin null olmaması durumunda ? operatörün sağ tarafında yer alan metot koşturulur.

Statik metot ve sabit (constant) değerlere erişimi sağlamak için T() operatörü kullanılır. Bir sonraki örnekte domain isimli değişkene DomainNames sınıfında yer alan TEST_DOMAIN sabit değeri enjekte edilmektedir. Aynı şekilde bir sınıfın static metotlarını koşturmak mümkündür.

```
    <property name="domain"
            value="#{T(com.kurumsaljava.spring.DomainNames).
                                TEST_DOMAIN}"/>
```

SpEL ile Java'da bulunan ? operatörü kullanmak mümkündür. ? operatörü if/else ile yapılan bir atamanın kısaltılmış halidir. Aşağıda yer alan örnekte eğer car nesnesinin model değişkeni Clio değerine sahipse, message değişkenine Clio değeri enjekte edilmektedir. Aksi taktirde enjekte edilen değer a Car olacaktır.

```
<property name="message"
        value="#{car.model == 'Clio' ? 'Clio' : 'a Car'}"/>
```

SpEL'in sıkça kullanıldığı diğer bir alan ise liste işlemleridir. Daha önceki bir bölümde util isim alanını kullanarak şöyle bir bir liste oluşturmuştuk.

```
<util:set id="cars" set-class="java.util.HashSet">
    <ref bean="a5" />
    <ref bean="clio" />
</util:set>
```

SpEL aracılığı ile bu listenin elementlerine erişebilir ve başka nesnelere bu değerleri enjekte edebiliriz. Aşağıda yer alan örnekte cars isimli liste içinde yer alan birinci element (clio) clioCarFactory nesnesinin car isimli değişkenine enjekte edilmektedir.

```
<bean id="clioCarFactory"
        class="com.kurumsaljava.spring.ClioCarFactory">
    <property name="car" value="#{cars[1]}"/>
</bean>
```

Aynı şekilde bir java.util.Properties nesnesinde yer alan değerlere sahip oldukları anahtarlar (key) üzerinden erişmek mümkündür. Bir sonraki örnekte oracle.properties dosyası içinde yer alan db.name anahtarına sahip değer oracleDataSource nesnesinin dbName değişkenine enjekte edilmektedir.

```
<util:properties id="dbConfiguration"
                location="oracle.properties" />

<bean id="oracleDataSource"
            class="com.kurumsaljava.spring.OracleDataSource">
    <property name="dbName"
            value="#{dbConfiguration['db.name']}" />
</bean>

#oracle.properties
db.name=RENTACAR
```

```
db.user=rent
db.password=car
```

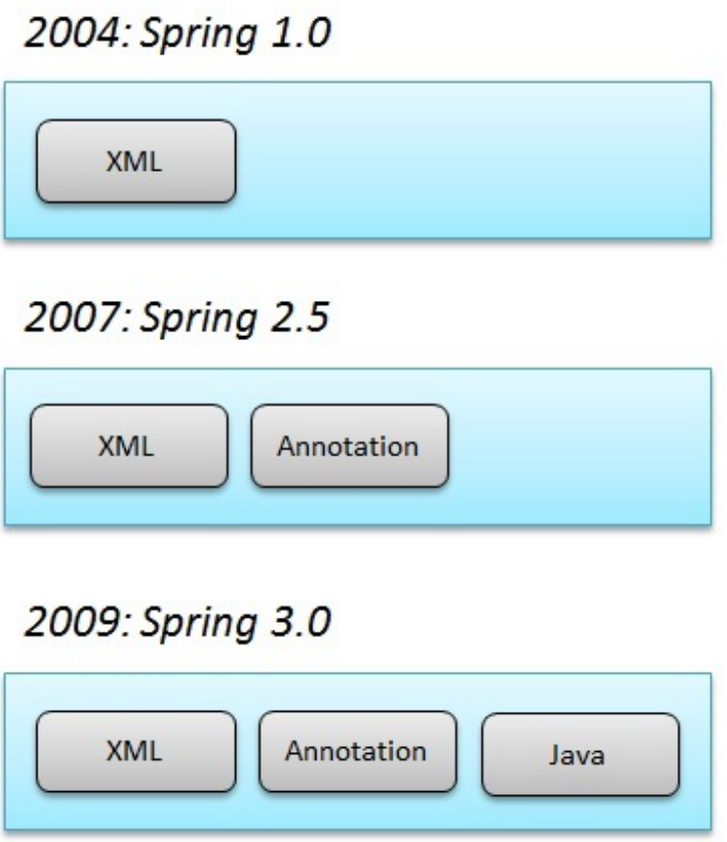
Bunun yanı sıra SpEL'in sahip olduğu systemEnvironment ve systemProperties değişkenleri üzerinde sistem değişkenlerine ve -D parametresi ile uygulamaya sunulan değerlere erişmek mümkündür.

```
<property name="path"
            value="#{systemEnvironment['JAVA_PATH']}"/>

-Ddb.name=RENTACAR
<property name="path"
            value="#{systemProperties['db.name']}"/>
```

Anotasyon Bazlı Konfigürasyon

Şimdiye kadar kullandığımız örneklerin hepsinde Spring sunucusunu (container) konfigüre etmek için bir XML dosyası kullandık. Bu konfigürasyon dosyasında yer alan bean tanımlamalarını kullanarak bağımlılıkların enjekte edilmesini sağladık. Enjekte etmek istediğimiz her bağımlılığın bean olarak bu konfigürasyon dosyasında yer alması gerekti.

Spring 2.5 sürümü ile anotasyonlar aracılığı ile uygulama konfigürasyonu oluşturma imkanı sunulmuştur. Örneğin @Autowired anotasyonu yardımı ile Java sınıf düzeyinde enjekte edilecek bağımlılıkları tanımlamak mümkün hale gelmiştir.



Resim 4.2

@Autowired anotasyonu kullanılarak sınıf tipine bağımlı olarak bir bağımlılık enjekte edilebilir. Kod 4.13 de yer alan RentalServiceImpl sınıfının customerRepository ve rentalRepository değişkenlerine gerekli bağımlılıklar Spring tarafından @Autowired anotasyonu yardımı ile enjekte edilmektedir.

```
Kod 4.13 - RentalServiceImpl

@Component("rentalService")
public class RentalServiceImpl implements RentalService {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }

    @Override
    public Rental rentACar(String customerName,
                    Car car, Date begin, Date end) {
        Customer customer =
```

```
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }
}
```

@Autowired ve @Component anotasyonlarını aynı sınıf bünyesinde birlikte kullandığımız taktirde RentalServiceImpl sınıfı için konfigürasyon dosyasında bean tanımlama gerekliliği bulunmamaktadır. Konfigürasyon dosyasında bean tanımlaması yerine bu işlemi anotasyon aracılığı ile yapmak istediğimizi konfigürasyon dosyasında belirtmemiz gerekmektedir. Bunun nasıl yapılabileceğiz kod 4.14 de yer almaktadır.

```
Kod 4.14 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
                    spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.kurumsaljava.spring"/>

</beans>
```

Anotasyon bazlı konfigürasyon yapabilmemiz için context isim alanını konfigürasyon içinde yüklüyor ve anotation-config elementini tanımlıyoruz. Spring'in @Autowired ve kullandığımız diğer anotasyonları keşfedebilmesi için mevcut sınıfları uygulama ayağa kalkmadan taraması ve anotasyonların nerede kullanıldığını keşfetmesi gerekiyor. Bu amaçla component-scan elementini kullanarak, hangi paket içinde yer alan sınıfların bu anotasyonları taşıdığını belirtiyoruz. Base-package element özelliği Spring'e hangi paketten başlayarak

anotasyonlu sınıfları taraması gerektiğini gösterir. Bu paket ve ihtiva ettiği tüm alt paketler içinde tarama gerçekleştirilir.

Base-package element özelliğini kullanarak bir sonraki örnekte görüldüğü gibi birden fazla paket tanımlaması yapabiliriz.

```
    <context:component-scan
            base-package="com.kurumsaljava.spring,
                                com.kurumsaljava.app"/>
```

Include-filter ve exclude-filter elementleri ile tarama esnasında dikkate alınacak ya da göz ardı edilecek sınıflar tanımlanabilir.

```
<context:component-scan
                base-package="com.kurumsaljava.spring">

    <context:include-filter type="regex"
        expression=".*Service, .*Repository" />

    <context:exclude-filter type="regex"
        expression="com.kurumsaljava.spring.MysqlDataSource" />
</context:component-scan>
```

Bir Java sınıfının Spring tarafından bir bean olarak algılanabilmesi için sınıfın @Component anotasyonu ile işaretlenmiş olması gerektiğini belirtmiştim. Component-scan ile bu anotasyonu ihtiva eden tüm sınıflar Spring bean olarak hafızaya yüklenir. Kod 4.13 de yer alan RentalServiceImpl sınıfı bu anotasyon ile bir Spring bean haline getirilmiştir. @Component anotasyonunun kullanımı, böyle bir sınıfın konfigürasyon dosyasında bean elementi ile tanımlanmasına eş gelmektedir. @Component anotasyonunda kullandığımız rentalService ibaresi konfigürasyon dosyasında kullanılan id ya da name element özelliği ile aynı vayifeyi görmektedir. Bu ismi kullanarak kod 4.15 de görüldüğü gibi bir rentalService nesnesi edinebiliriz.

```
Kod 4.15 - Main

public static void main(String[] args) throws Exception {
    ConfigurableApplicationContext ctx =
        new ClassPathXmlApplicationContext("annotationContext.xml");

    RentalServiceImpl rentalService = ctx.getBean("rentalService",
            RentalServiceImpl.class);
    Rental rental = rentalService.rentACar("Özcan Acar", new Car(),
            null, null);
```

```
    System.out.println(rental.isRented());
}
```

Kod 4.13 e tekrar göz attığımızda sınıf değişkenleri için get() ve set() metotlarının tanımlanmadığını görmekteyiz. @Autowired özellikle set() metodunun bağımlılığın enjekte edilebilmesi için kullanımı zorunluluğunu ortadan kaldırmaktadır. @Autowired herhangi bir sınıf değişkeni, sınıf konstrüktörü ya da metodu üzerine yerleştirilerek, bağımlılığın direk sınıf değişkenine (field injection), sınıf konstrüktörü (constructor injection) ya da herhangi bir sınıf metodu (method injection) üzerinden enjekte edilmesini mümkün kılmaktadır. Method injection yöntemi kullanıldığında metot herhangi bir ismi taşıyabilir. Böylece sadece set() metotlarının ve konstrüktörlerin bağımlılıkların enjeksiyonu için kullanılmaları zorunluluğu ortadan kalkmaktadır. Kod 4.16 da yer alan örnekte RentalServiceImpl sınıfının ihtiyaç duyduğu bağımlılıklar sınıf konstrüktörü üzerinden enjekte edilmektedir. Bunun için @Autowired anotasyonunu sınıf konstrüktörü üzerine yerleştirmek yeterlidir.

```
Kod 4.16 - RentalServiceImpl

@Component("rentalService")
public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    @Autowired
    public RentalServiceImpl(CustomerRepository customerRepository,
            RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }
}
```

Aynı şekilde @Autowired anotasyonunu herhangi bir metot üzerine yerleştirerek, bu metodun bağımlılığın enjeksiyonu için kullanılmasını sağlayabiliriz. Bunun bir örneği kod 4.17 de yer almaktadır.

```
Kod 4.17 - RentalServiceImpl

@Component("rentalService")
public class RentalServiceImpl implements RentalService {
```

```
    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }

    @Autowired
    public void injectRentalRepository(
                        RentalRepository repo) {
        this.rentalRepository = repo;
    }

    @Autowired
    public void injectCustomerRepository(
                    CustomerRepository repo) {
        this.customerRepository = repo;
    }
}
```

@Autowired ile bağımlılık sahip olduğu veri tipine göre enjekte edilir. Buna injection by type (sınıf tipine göre enjeksiyon) ismi verilmektedir. Kod 4.13 örneğinde customerRepository isimli sınıf değişkeni CustomerRepository veri tipine sahiptir. Böyle bir bağımlılığın enjekte edilebilmesi için CustomerRepository isimli interface sınıfı implemente eden bir sınıfın mevcut olması ve @Component anotasyonu ile işaretlenmiş olması gerekir. Aksi taktirde bağımlılık enjekte edilemeyeceğinden NoSuchBeanDefinitionException hatası oluşur. @Autowired anotasyonunun required değişkene false değeri atandığı taktirde bu hatanın oluşması önlenebilir.

```
@Autowired(required=false)
private CustomerRepository customerRepository;
```

Bir önceki örnekte Spring customerRepository değişkenine CustomerRepository tipinde bir nesneyi enjekte etmeye çalışacaktır. Böyle bir nesne bulunamaması durumunda customerRepository değişkeni null değeri atanır ve NoSuchBeanDefinitionException hatası oluşmaz.

Aynı sınıfın birden fazla implementasyonu olması durumunda Spring hangi implementasyonu seçmesi gerektiği konusunda tereddüte düşer ve bunu NoSuchBeanDefinitionException, no unique bean of type hatası olarak dışa vurur. İmplementasyonları birbirlerinden ayırt edebilmek için @Qualifier

anotasyonu kullanılabilir. Bir sonraki örnekte CustomerRepositoryImpl bünyesinde yer alan datasource isimli değişkene DataSource veri tipinde bir bağımlılığı enjekte ediyoruz. Hangi bağımlılığın enjekte edilmesi gerektiğini @Qualifier anotasyonu yardımı ile ifade ediyoruz. @Qualifier anotasyonu parametre olarak bir bean ismi almaktadır. MysqlDataSource sınıfında olduğu gibi bu ismi @Component anotasyonu aracılığı ile tayin edebiliriz. Spring nesnesi ismi kullanılarak yapılan enjeksiyona injection by name ismi verilmektedir.

```
@Component
public class OracleDataSource implements DataSource {

}

@Component("mysql")
public class MysqlDataSource implements DataSource {

}

@Component
public class CustomerRepositoryImpl
            implements CustomerRepository {

    @Autowired
    @Qualifier("mysql")
    private DataSource datasource;
}
```

@Component anotasyonu gibi @Qualifier anotasyonu da bir sınıfa Spring bean ismi vermek için kullanılabilir. Bir sonraki örnekte @Qualifier anotasyonunu kullanarak MysqlDataSource sınıfına mysql ismini veriyoruz.

```
@Component
@Qualifier("mysql")
public class MysqlDataSource implements DataSource {

}
```

@Qualifier kullanılmadığı taktirde iki implementasyonun mevcut olması durumunda NoSuchBeanDefinitionException, no unique bean of type hatası oluşacağını söylemiştik. Bu hatanın oluşmasını önlemenin yolu @Qualifier anotasyonunun kullanılmasıdır. @Qualifier anotasyonunu kullandığımız taktirde her implementasyon sınıfına bir isim vermemiz gerekiyor. Bu da açıkçası kodun bakımını zora sokan bir durum teşkil edebilir. İsim değişiklikleri

kodun yeniden derlenmesini zorunlu kılacaktır. Bu sebepten dolayı sadece bir implementasyon sınıfının kullanılmasında fayda vardır.

Sınıf değişkenlerine, metot ya da metot parametrelerine bir değer enjekte etmek için @Value anotasyonu kullanılır. Aşağıda yer alan örnekte domainName isimli değişkene rentcar.com değeri enjekte edilmektedir.

```
@Value("rentacar.com")
private String domainName;
```

Daha önce tanıştığımız SpEL ile @Value anotasyonunu kombine ederek, dinamik değerlerin enjekte edilmesi sağlanabilir. Bir sonraki örnekte -Ddomain=rentacar.com şeklinde uygulamaya verilen bir parametre SpEL kullanılarak dominName değişkenine enjekte edilmektedir.

```
@Value("#{systemProperties.domain}")
private String domainName;
```

@Autowired, @Required, @PostConstruct, @PreDestroy ve @Resource gibi anotasyonların Spring tarafından keşfedilebilmeleri ve görevlerini yerine getirebilmeleri için anotation-config elementinin XML dosyasında tanımlanması gerekmektedir.

```
<context:annotation-config/>
```

Bu aslında CommonAnnotationBeanPostProcessor sınıfının kısa yazılmış halidir. CommonAnnotationBeanPostProcessor sınıfı bir BeanPostProcessor işlemcisi olup, Application Context oluşturulduktan ve Spring bean tanımlamalarından nesneler oluşturulduktan sonra devreye girmekte ve örneğin @Autowired kullanılan yerlere gerekli bağımlılıkları enjekte etmektedir. Annotation-config elementi CommonAnnotationBeanPostProcessor ve diğer BeanPostProcessor işlemcilerini aktif hale getirmektedir.

Kod 4.13 de RentalServiceImpl sınıfını @Component anotasyonu ile işaretlemiştik. Bu anotaston RentalServiceImpl sınıfını bir Spring bean tanımlaması haline getirmektedir. @Component ile Spring nesnelerinin XML dosyasında tanımlanma zorunluluğu ortadan kalmıştır. Spring sunucusunun @Component ile işaretli sınıfları keşfedebilmesi için kod 4.14 de görüldüğü gibi component-scan elementinin XML konfigürasyon dosyasına eklenmesi gerekmektedir. Spring sunucusu component-scan elementi ile tanımlamış paketleri (buna altpaketlerde dahildir) tarayarak, @Component anotasyonunu

taşıyan sınıflardan Spring nesneleri oluşturur ve bunları Application Context içine konuşlandırır. Ayrıca bu nesneleri @Autowired ile işaretli değişkenlere enjekte eder.

@Autowired ile işaretli değişken ya da metotlara gerekli bağımlılıkların enjekte edilebilmesi için bu bağımlılıkların Spring bean olarak keşfedilmeleri gerekmektedir. Bunu ya sınıf bazında @Component anotasyonunu ve bu sınıfların otomatik keşifleri için XML konfigürasyonu dosyasında component-scan ile yapabiliriz ya da bağımlıkları Spring bean olarak XML dosyasında tanımlayabiliriz.

Kod 4.17.2 bağımlılıkların enjekte edilmesi için @Autowired anotasyonu kullanılmıştır. Kod 4.17.1 de yer alan konfigürasyona baktığımızda annotation-config elementinin kullanıldığını görmekteyiz. Buna karşın component-scan elementi kullanılmamıştır. Bu durumda RentalServiceImpl sınıfına ihtiyaç duydugu bağımlılıkların enjekte edilebilmesi için kod 4.17.1 de görüldüğü gibi bu bağımlılıkların Spring bean olarak konfigürasyon dosyasında tanımlanmaları gerekmektedir, çünkü component-scan olmadan @Component anotasyonunu taşıyan sınıflar otomatik olarak keşfedilez ve Application Context içine Spring bean olarak yerleştirilemezler. Bu yüzden Spring bean olarak XML konfigüyasyon dosyasında tanımlanmaları gerekir.

Kod 4.17.1 de component-scan elementi kullanılmış olsaydı bile CustomerRepositoryImpl ve RentalRepositoryImpl sınıflarından olan nesnelerin RentalServiceImpl sınıfına enjekte edilmeleri mümkün olmazdı, çünkü @Component anotasyonunu taşımamaktadırlar. Bu sebepten dolayı kod 4.17.1 de olduğu gibi Spring bean olarak XML konfigürasyon dosyasında tanımlanmaları gerekir. Spring sunucusu CommonAnnotationBeanPostProcessor aracılığı ile @Autowired ile işaretli bağımlılıkları enjekte eder.

```
Kod 4.17.1 - applicationContext.xml

<beans>
    <context:annotation-config />
    <bean id="customerRepository"
            class="...CustomerRepositoryImpl"/>

    <bean id="rentalRepository"
            class="...RentalRepositoryImpl"/>
</beans>
```

```
Kod 4.17.2 - RentalServiceImpl

public class RentalServiceImpl implements RentalService {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private RentalRepository rentalRepository;

    ...
}

public class CustomerRepositoryImpl implements CustomerRepository {
    ...
}

public class RentalRepositoryImpl implements RentalRepository {
    ...
}
```

Standart Java Anotasyonları

@Autowired ve diğer anotasyonların kullanımı konfigürasyon dosyasına olan bağımlılığı azaltmakla birlikte, kodu Spring çatışına bağımlı kılmaktadır. Bu sorunu gidermek amacıyla Google ve SpringSource'un çabalarıyla JSR 330 (Java Specification Request) oluşturulmuş ve 2009 sonunda Java'ya bağımlılıkların enjeksiyonu için kullanılmak üzere @Inject olarak bilinen anotasyon seti kazandırılmıştır. JSR 330 ile bağımlılıkların enjeksiyonu için standart bir programlama modeli oluşturulmuştur. Spring 3.0 sürümü ile bu modeli desteklemektedir. Buna göre kod bünyesinde @Autowired yerine standart Java anotasyonlarını kullanarak, kodumuzun Spring çatışına olan bağımlılığını ortadan kaldırabiliriz.

Bu yeni standartın merkezinde @Inject anotasyonu bulunmaktadır. Kod 4.18 i incelediğimizde @Component yerine @Named, @Autowired yerine @Inject anotasyonunun kullanıldığını görmekteyiz. Bu anotasyonlar Java kütüphanesine sonradan eklendikleri için javax.inject paketi içinde yer almaktadırlar. @Autowired anotasyonunda olduğu gibi @Inject anotasyonu sınıf değişkeni, konstrüktör ve metot bazında bağımlılıkların enjekte edilmesi için kullanılabilir. @Autowired ile @Inject arasındaki tek fark @Inject anotasyonunun required özelliğini tanımamasıdır. @Inject her zaman

bağımlılığın mevcut olduğu farz eder. Mevcut olmayan bir bağımlılığın enjekte edilmesi hata oluşmasına sebep olacaktır.

```
Kod 4.18 - CustomerRepositoryImpl

import javax.inject.Inject;
import javax.inject.Named;

@Named
public class CustomerRepositoryImpl
        implements CustomerRepository {

    @Inject
    private DataSource datasource;
}
```

Spring anotasyonları ve JSR-330 ile gelen yeni Java anotasyonlarının kıyaslaması tablo 4.1'de yer almaktadır.

Spring	JSR 330	JSR 330 Kısıtlamalar
@Autowired	@Inject	*@Inject'in required özelliği bulunmuyor.*
@Component	@Named	
@Scope	@Scope	
@Scope ("Singletion")	@Singleton	*jsr-330'un temel nesne oluşturma ayarı Spring'in scope=prototype özelligine eşittir.*
@Qualifier	@Named	
@Value	*karşılığı bulunmuyor*	
@Required	*karşılığı bulunmuyor*	
@Lazy	*karşılığı bulunmuyor*	

Tablo 4.1

@Inject anotasyonunun @Autowired anotasyonuna kıyasla artı bir özelliği daha mevcuttur. javax.inject.Provider kullanıldığı taktirde arzu edilen nesne yerine tedarikçi anlamına gelen bir provider nesnesi enjekte edilir. Jenerik yapıda olan bu sınıf, bağımlılığın hemen değil, belli bir metot koşturulduğunda enjekte edilmesini sağlar. Bu metot get() metodudur. Bu yönteme ise lazy

loading ismi verilmektedir. Kod 4.19 da yer alan örnekte sınıf konstrüktörü üzerinde bir Provider nesnesi enjekte edilmektedir. Döngü içinde carProvider.get() metodu yeni bir Car nesnesi geri vereceginden, konstrüktör son buldugunda cars isimli set içine üç adet car nesnesi konuşlandırılacaktır.

```
Kod 4.19 - CarFactoryImpl

package com.kurumsaljava.spring;

import java.util.HashSet;
import java.util.Set;
import javax.inject.Inject;
import javax.inject.Provider;

public class CarFactoryImpl implements CarFactory {

    private Set<Car> cars;

    @Inject
    public CarFactoryImpl(Provider<Car> carProvider) {
        cars = new HashSet<Car>();
        for (int i = 0; i < 3; i++) {
            cars.add(carProvider.get());
        }
    }

    @Override
    public Car build() {
        return null;
    }
}
```

Diğer Spring Anotasyonları

Anotasyon bazlı konfigürasyonu seçtiğimiz taktirde, Spring'in sunduğu ya da JSR 330 bünyesinde tanımlanan standart Java anotasyonları kullanmamız gerektiğinden bahsettik. Kullanabileceğimiz anotasyon seti sadece @Autowired ve @Inject ile sınıflı değildir. Aşağıdaki Spring çatısı bünyesinde yer alan anotasyon listesi bu seti tamamlayıcı niteliktedir.

- ***@Controller***: Bir sınıfı Spring'in web uygulamaları geliştirmek için kullanılan MVC çatışında bir controller sınıfı olarak işaretler.
- ***@Repository***: Bir sınıfı veri tabanı işlemleri yapan bir repository sınıfı olarak işaretler.

- ***@Service***: Bir sınıfı servis katmanında iş gören bir sınıf olarak işaretler.
- ***@Component***: Bir sınıfı Spring bean olarak işaretler.

Java Bazlı Konfigürasyon

XML ve anotasyon bazlı konfigürasyon yanı sıra Spring 3.0 ile Java bazlı Spring konfigürasyonu oluşturmak mümkün hale gelmiştir. Bu tür konfigürasyon XML bazlı konfigürasyonu tamamen ortadan kaldırmak için kullanılır. Java bazlı Spring konfigürasyonu yazılımcıya nesnelerin oluşturulması ve konfigürasyonu konusunda daha kamsaplı kontrol mekanizmaları sunmaktadır. Ayrıca bağımlılıkların tanımlanmasında ve enjeksiyonunda veri tipi kontrolü (type safety) daha iyi yapılmaktadır, çünkü bu görevi Java derleyicisi üstlenmektedir.

Bir Spring uygulamasını tamamen anotasyonlar ve bir konfigürasyon sınıfı yardımı ile konfigüre etmek mümkündür. Java bazlı konfigürasyonda @Component anotasyonu ile sınıfları işaretleyerek bir Spring bean haline getirme zorunluluğu bulunmamaktadır.

@Configuration ve @Bean anotasyonları kullanılarak bir Java sınıfı konfigürasyonun merkezi haline getirilir. Kod 4.20 de yer alan örnekte SpringJavaConf böyle bir konfigürasyon sınıfını temsil etmektedir. Bu sınıf @Configuration anotasyonu ile bir Spring konfigürasyon sınıfı haline getirilmiştir.

```
Kod 4.20 - SpringJavaConf

package com.kurumsaljava.spring.javaconfiguration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.kurumsaljava.spring.CustomerRepository;
import com.kurumsaljava.spring.RentalRepository;
import com.kurumsaljava.spring.RentalService;

@Configuration
public class SpringJavaConf {

    public SpringJavaConf{
    }

    @Bean
    public CustomerRepository createCustomerRepositoryBean() {
```

```
        return new CustomerRepositoryImpl();
    }

    @Bean
    public RentalRepository createRentalRepositoryBean() {
        return new RentalRepositoryImpl();
    }

    @Bean
    public RentalService createRentalServiceBean() {
        RentalServiceImpl rentalService =
                    new RentalServiceImpl();
        rentalService.setCustomerRepository(
                        createCustomerRepositoryBean());
        rentalService.setRentalRepository(
                        createRentalRepositoryBean());
        return rentalService;
    }
}
```

@Bean anotasyonu XML konfigürasyon dosyasından tanıdığımız bean elementiyle aynı göreve sahiptir. Bu anotasyon yardımı ile bir Spring nesnesi tanımlaması oluşturulur. @Bean anotasyonu ile işaretlenen metodun ismi Spring nesnesinin Application Context içindeki ismi olarak kullanılır.

Kod 4.20 de @Bean anotasyonu yardımı ile üç Spring bean oluşturulmaktadır. CreateRentalServiceBean() bünyesinde diğer bean tanımlamaları bağımlılık olarak set() metotları üzerinden enjekte edilmektedir. createRentalRepositoryBean(), createCustomerRepositoryBean() ve createRentalServiceBean() metotlarını Spring Application Context bünyesinde nesneleri üreten fabrika metotları olarak düşünebilirsiniz. Bu metotların koşturulması Spring tarafından tekil nesnelerin oluşturulmasını sağlayacaktır, yani varsayılan nesne oluşturma türü XML konfigürasyonunda da oldugu gibi singletondır. Peki bu nasıl gerçekleşmektedir?

Spring sunucusu çalışmaya başladığında SpringJavaConf sınıfına vekil (proxy) olan bir nesne oluşturur. Bu vekil nesneyi oluşturmak için CGLIB kütüphanesini kullanır. Örneğin sunucudan createCustomerRepositoryBean() aracılığı ile bir CustomerRepository nesnesi talep edildiginde, bu talebe cevap verecek olan vekil nesnedir. Vekil nesne önce talep edilen nesnenin Application Context bünyesinde olup, olmadığını kontrol eder. Eğer aranan nesne Application Context içinde ise, vekil nesne tarafından bu nesne geriye verilir. Aranan nesne Application Context içinde değilse, vekil nesne aranan nesneyi

oluşturarak hem Application Context içine yerleştirir hem de bu nesneyi geri verir. Böylece Application Context içinde yer alan tekil nesne kullanılmış olur. Bu davranışı değiştirmek yani prototype tipi nesneler oluşturmak için @Scope("prototype") anotasyonu kullanılabilir.

```
@Bean @Scope("prototype")
public RentalRepository createRentalRepositoryBean() {
    return new RentalRepositoryImpl();
}
```

Oluşturduğumuz bu konfigürasyon sınıfını yüklemek ve uygulamayı çalışır hale getirmek için AnnotationConfigApplicationContext sınıfından faydalanabiliriz. Kod 4.21 de yer alan main() metodu uygulamanın bir Java konfigürasyon sınıfı ile nasıl çalışır hale getirildiğini göstermektedir.

```
Kod 4.21 - Main

package com.kurumsaljava.spring.javaconfiguration;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
                    AnnotationConfigApplicationContext;

import com.kurumsaljava.spring.Car;
import com.kurumsaljava.spring.Rental;

public class Main {

    public static void main(String[] args)
                            throws Exception {
        ApplicationContext ctx =
                new AnnotationConfigApplicationContext(
                                    SpringJavaConf.class);

        RentalServiceImpl rentalService =
                ctx.getBean(RentalServiceImpl.class);

        Rental rental = rentalService.rentACar(
                    "Özcan Acar",new Car(), null, null);

        System.out.println(rental.isRented());
    }
}
```

Bir Spring uygulaması bünyesinde bahsettiğimiz üç değişik konfigürasyon

yöntemi (XML, anotasyon, Java) birlikte kullanılabilir. Aşağıda ye alan örnekte annotation-config tanımlanmış, lakin component-scan kullanılmamıştır. Spring'in sınıfları bulabilmesi için bu sınıfların bean tanımlaması olarak konfigürasyon dosyasında yer alması gerekmektedir. Bağımlılıkların enjekte edilmesi için property ya da constructor-arg elementlerinin kullanımı gerekli değildir, çünkü annotation-config elementi kullanıldığı için bağımlılıklar sınıf bünyesinde bulunan @Autowired ya da @Inject anotasyonları aracılığı ile enjekte edilebilir.

```
<context:annotation-config />

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
</bean>

<bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl" />

<bean id="customerRepository"
            class="com.kurumsaljava.spring.CustomerRepositoryImpl" />
```

Bir sonraki örnekte XML konfigürasyon dosyası ve bir konfigürasyon sınıfı birlikte kullanılmaktadır. CustomerRepository ve rentalRepository nesneleri @Autowired ile SpringJavaConf isimli konfigürasyon sınıfını enjekte edilmektedir. @Autowired anotasyonu kullanıldığı için bu bean tanımlamalarının XML konfigürasyon dosyasında bean olarak tanımlanmaları gerekmektedir.

```
# applicationContext.xml
<context:annotation-config />

<bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                        RentalRepositoryImpl" />

<bean id="customerRepository"
            class="com.kurumsaljava.spring.
                    CustomerRepositoryImpl" />

# SpringJavaConf.java
@Configuration
```

```
public class SpringJavaConf {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private RentalRepository rentalRepository;

    @Bean
    public RentalService createRentalServiceBean() {
        RentalServiceImpl rentalService =
                    new RentalServiceImpl();
        rentalService.setCustomerRepository(
                        customerRepository);
        rentalService.setRentalRepository(
                        rentalRepository);
        return rentalService;
    }
}
```

@Import Anotasyonu Kullanımı

Daha önceki bölümlerde Spring konfigürasyonunu import konfigürasyon elementini kullanarak birden fazla konfigürasyon dosyasından bir araya getirebileceğimizi görmüştük. Aynı işlemi Java bazlı konfigürasyonda @Import anotasyonu ile yapmak mümkündür.

```
@Configuration
@Import({SecurityConf.class, AppConfig.class})
public class SpringJavaConf {
}

@Configuration
public class SecurityConf {
}

@Configuration
public class AppConfig {
}
```

Yukarı yer alan örnekte SpringJavaConf sınıfına SecurityConf ve AppConfig bünyesinde yer alan bean tanımlamaları import, yani dahil edilmektedir.

@PropertySource Anotasyonu Kullanımı

@PropertySource anotasyonu ile bir property dosyasında yer alan herhangi bir değer Environment aracılığı ile bir nesne oluştururken kullanılabilir.

Aşağıda yer alan örnekte CustomerRepositoryImpl sınıfından olan nesneye sınıf konstrüktörü aracılığı ile classpath içinde bulunan database.properties isimli dosyada yer alan database.name isimli anahtarın değeri enjekte edilmektedir.

```
@Configuration
@PropertySource("classpath:/database.properties")
public class SpringJavaConf {

    @Inject Environment env;

    @Bean
    public CustomerRepository customerRepository() {
        return new CustomerRepositoryImpl(
                env.getProperty("database.name"));
    }
}
```

Hangi Konfigürasyon Yöntemi Kullanılmalıdır?

XML bazlı konfigürasyon yapısı itibari ile değişken olmayan (static) bir uygulama konfigürasyonu için kullanılır. Tüm konfigürasyonu merkezi bir yerde toplamak mümkündür. Bu uygulamanın konfigürasyonunu ve bunun bakımını kolaylaştırır. Spring kullanan uygulama geliştiricilerinin çoğu XML konfigürasyon dosyaları ile çalıştıklarından, bir Spring uygulamasının nasıl konfigüre edildiğini bilirler. Ayrıca XML bazlı konfigürasyonda mevcut her sınıf konfigüre edilebilir. Ama ne yazık ki XML bazlı konfigürasyon sahip olduğu bazı limitlerden dolayı esnek bir uygulama konfigürasyonu oluşturulmasını tam anlamıyla desteklemez. Bunun yanı sıra XML her uygulama geliştiricisinin severek kullandığı bir teknoloji değildir.

Anotasyon bazlı konfigürasyon uygulamanın hızlı bir şekilde geliştirilmesini destekler. Konfigürasyon anotasyonlar yardımı ile sınıf bazında gerçekleşir. Bu aynı zamanda tüm konfigürasyonun mevcut sınıflara dağılmasını ve konfigürasyonun ve bakımının zorlaşmasına sebep olabilir. Anotasyon bazlı konfigürasyon sadece uygulama geliştiricisi tarafından oluşturulan sınıflar çapında mümkündür. Anotasyonlar yabancı sınıfların konfigürasyonuna izin vermez. Bunun mümkün hale gelebilmesi için uygulama geliştiricisinin

sınıfların kaynak koduna sahip olması ve koda anotasyonlar ekledikten sonra yeniden derleyebilmesi gerekmektedir. Çoğu zaman bu mümkün olmayan bir şeydir.

Java bazlı konfigürasyon ile tüm uygulama konfigürasyonu bir Java sınıfı bünyesinde gerçekleşir. Konfigüre edilmek istenen sınıflar bünyesinde herhangi bir anotasyon kullanımı mecburi olmadığı için bu durum POJO (Plain Old Java Object) sınıfların oluşmasını ve kullanılmasını destekler. Java bazlı konfigürasyonda Spring XML isim alanları olarak tanıdığımız konseptin karşılığı yoktur.

4. Bölüm Soruları

4.1 Spring nesne tanımlamalarında kalıtımı sağlamak için kullanılan konfigürasyon elementi hangisidir?

4.2 Spring nesne tanımlamalarında kalıtımın sağladığı avantajlar nelerdir?

4.3 Bir Spring nesne tanımlamasını soyutlaştıran konfigürasyon elementi hangisidir?

4.4 Spring soyut Spring nesne tanımlamalarından neden nesne oluşturamaz?

4.5 Dahili bean tanımlamaları ne zaman kullanılır?

4.6 Konfigürasyon dosyalarının hacimsel büyümelerini önlemek için hangi yöntem kullanılır?

4.7 Property ve ref elementlerinin daha kısa yazılmasını sağlayan isim alanının ismi nedir?

4.8 Bir e-posta adresinin bağımlılık olarak enjeksiyonu esnasında geçerli olup, olmadığı nasıl kontrol edilebilir?

4.9 Bir Spring uygulaması XML haricinde hangi yöntem kullanılarak konfigüre edilebilir?

4.10 Bir Spring nesnesine isim vermek için kullanılan anotasyon hangisidir

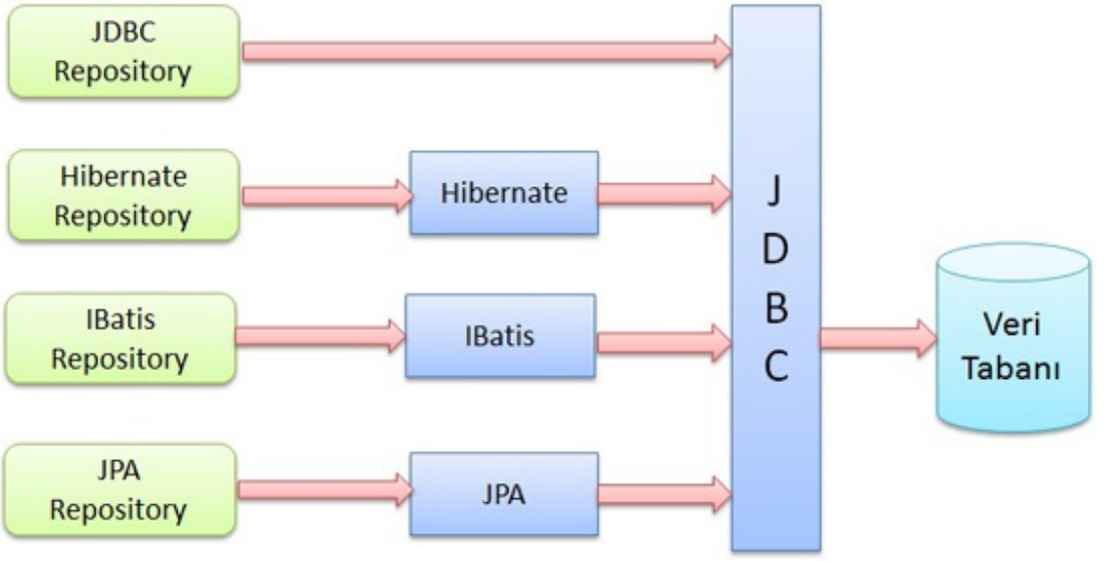
4.11 @Autowired anotasyonu yerine hangi standart Java anotasyonu kullanılabilir?

5. Bölüm

Spring İle Veri Tabanı İşlemleri

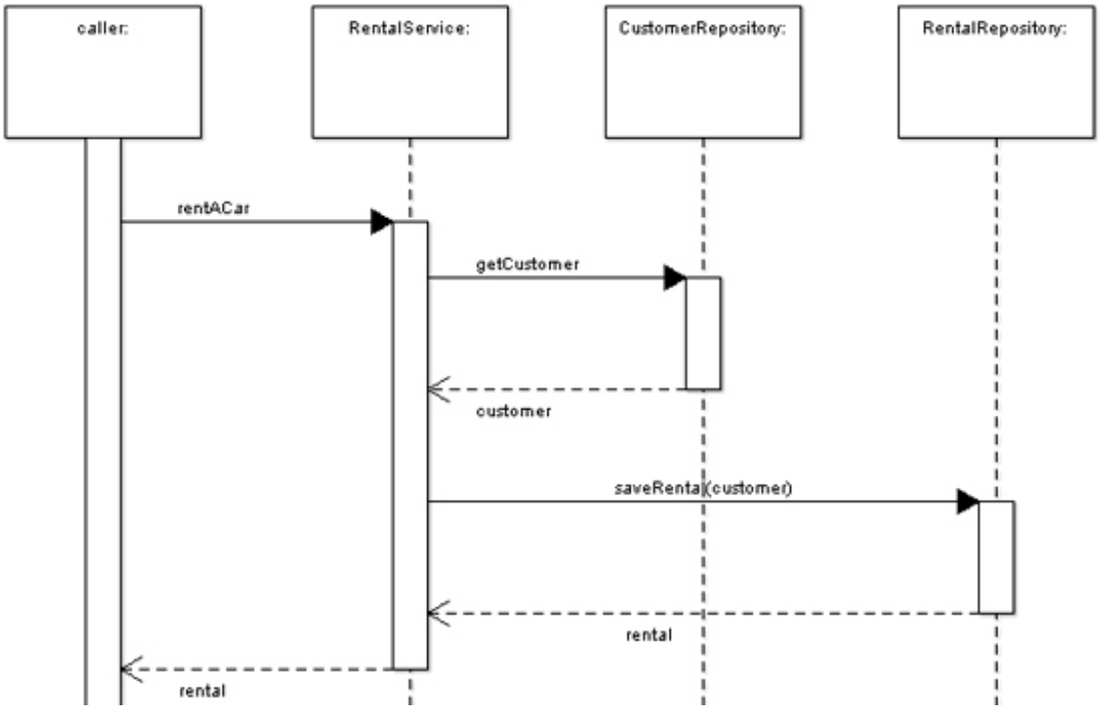
JDBC İle Veri Tabanı İşlemleri

Java dünyasında veri tabanı işlemi dendiğinde ilk akla gelen JDBC (Java Database Connectivity) dir. Zaman içinde Hibernate, IBatis ya da JPA (Java Persistence API) gibi veri tabanı sistemleri ile çalışma çatıları geliştirilmiş olsa bile, hepsinin temelinde JDBC'i vardır. Resim 5.1 de görüldüğü gibi JDBC uygulama ile veri tabanı sistemi arasında aracı rolünü üstlenmektedir. Bir Java uygulaması bünyesinde SQL (Structured Query Language) kullanılarak veri tabanı işlemleri gerçekleştirilir.



Resim 5.1

Resim 5.2 de yer alan araç kiralama servisi uygulamamızın dizge diyagramına tekrar göz attığımızda, veri tabanı işlemlerinden CustomerRepository ve RentalRepository sınıflarının sorumlu olduklarını görmekteyiz. Bu iki interface sınıf yapılabilecek veri tabanı işlemlerini tanımlamaktadırlar. Örneğin CustomerRepository sınıfının getCustomerByName(String name) isminde bir metodu mevcuttur. Bu metot verilen bir müşteri ismi aracılığı ile veri tabanında müşteri arama işlemlerini tanımlamaktadır.



Resim 5.2

İnterface sınıfları bünyelerinde sadece metot gövdeleri tanımlamış olan sınıflardır. Bir işe yarayabilmeleri için altsınıflarca implemente edilmeleri gerekir. CustomerRepository sınıfının JDBCCustomerRepositoryImpl ismini taşıyan böyle bir implementasyonu kod 5.1 de yer almaktadır.

```
Kod 5.1 - Car

package com.kurumsaljava.spring;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;

@Named
public class JDBCCustomerRepositoryImpl
            implements CustomerRepository {

    @Inject
    private DataSource dataSource;

    public JDBCCustomerRepositoryImpl() {
    }
```

```
    @Override
    public Customer getCustomerByName(String name) {
        Customer customer = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            Connection con = dataSource.getConnection();
            pstmt = con.prepareStatement("select id, name,
                    firstname, age from customer where name=?");
            pstmt.setString(1, name);
            rs = pstmt.executeQuery();
            if (rs.next()) {
                customer = new Customer();
                customer.setId(rs.getLong("id"));
                customer.setName(rs.getString("name"));
                customer.setFirstname(rs.getString("firstname"));
                customer.setAge(rs.getInt("age"));
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if (rs != null) {
                    rs.close();
                }
                if (pstmt != null) {
                    pstmt.close();
                }
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
        return customer;
    }

    @Override
    public void save(Customer customer) {
        PreparedStatement pstmt = null;
        try {
            Connection con = dataSource.getConnection();
            pstmt = con.prepareStatement("
                insert into customer
                values(name, firstname, age)
                                values(?,?,?)");
            pstmt.setString(1, customer.getName());
            pstmt.setString(2, customer.getFirstname());
            pstmt.setInt(3, customer.getAge());
            pstmt.executeUpdate();
```

```
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if (pstmt != null) {
                    pstmt.close();
                }
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

JDBCCustomerRepositoryImpl ismini taşıyan sınıfın javax.sql.DataSource tipinde bir bağımlılığı mevcuttur. Bu bağımlılığı @Inject anotasyonunu kullanarak dışardan enjekte edebiliriz. Bu amaçla XML konfigürasyon dosyamızda bir dataSource bean tanımlaması yapıyoruz. Bu bean tanımlaması kod 5.2 de yer almaktadır.

```
Kod 5.2 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
                    spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan
                base-package="com.kurumsaljava.spring" />

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.
                    embedded.EmbeddedDatabaseFactoryBean">
        <property name="databasePopulator"
                    ref="populator"/>
        <property name="databaseType" value="HSQL"/>
    </bean>

    <bean id="populator"
        class="org.springframework.jdbc.datasource.init.
                            ResourceDatabasePopulator">
```

```
        <property name="scripts">
            <list>
                <value>schema.sql</value>
                <value>data.sql</value>
            </list>
        </property>
    </bean>
</beans>
```

Kod 5.2 de yer alan konfigürasyon dosyasında iki ayrıntı dikkatimizi çekiyor. Birincisi dataSource bean tanımlaması için EmbeddedDatabaseFactoryBean sınıfının kullanılması, ikincisi populator isminde bir bean tanımlaması yapılmış olmasıdır. EmbeddedDatabaseFactoryBean sınıfını kullanarak hafızada çalışan bir H2, HSQL ya da Derby veri tabanı sistemini oluşturabilir ve databaseType değişkeni üzerinden kullanmak istediğimiz veri tabanı sistemini seçebiliriz. Hafızada yer alan bir veri tabanı sisteminin kullanımı uygulamanın geliştirilme sürecini olumlu etkiler. Eğer EmbeddedDatabaseFactoryBean kullanmasaydık, dataSource tanımlamasını aşağıdaki şekilde tanımlamak zorunda kalırdık.

```
<bean
    class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close" id="dataSource">
    <property name="driverClassName"
                value="org.hsqldb.jdbcDriver" />
    <property name="url"
                value="jdbc\:hsqldb\:mem\:spring-playground" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

Doğal olarak hafızada bir veri tabanı sisteminin oluşturulmuş olması yeterli değildir. Bu veri tabanı sistemi bünyesinde gerekli tabloların oluşturulması ve uygulamanın testi öncesi gerekli verilerin tablolara kaydedilmesi gerekmektedir. Bu amaçla kod 5.2 örneğinde ResourceDatabasePopulator sınıfını kullandık. Bu sınıf aracılığı ile tabloları oluşturmak için kullanılan create ve verileri tablolara eklemek için kullanılan insert komutlarını script adı verilen dosyalardan yükleyerek, koşturmak mümkündür. kullandığımız örnekte schema.sql dosyasında create komutları, data.sql dosyasında insert komutları yer almaktadır. Bu dosyaların içerikleri aşağıda yer almaktadır. Uygulama koşturulmadan önce Spring tarafından bu iki script yardımı ile veri tabanı sistemi oluşturulur ve kullanıma sunulur.

```
#schema.sql
```

```
drop table customer if exists;
drop table rental if exists;
create table customer (id integer identity primary key,
            name varchar(50),  firstname varchar(50), age integer);
create table rental (id integer identity primary key,
            carid integer,  customerid integer, rented boolean);

#data.sql
insert into customer (id, name, firstname, age)
            values ('100', 'Isik', 'Ayhan', 75);
insert into customer (id, name, firstname, age)
            values ('200', 'Müren', 'Zeki', 65);
insert into customer (id, name, firstname, age)
            values ('300', 'Sunal', 'Kemal', 69);
```

DataSource tanımlamasını daha kısa yapmaya ne dersiniz? jdbc isim alanında yer alan embedded-database elementi yardımı ile dataSource tanımlaması kod 5.3 deki şekilde yapılabilir.

```
Kod 5.3 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
                spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan
            base-package="com.kurumsaljava.spring." />

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql"/>
        <jdbc:script location="data.sql"/>
    </jdbc:embedded-database>

</beans>
```

Spring JdbcTemplate Kullanımı

JDBC Java'da veri tabanı işlemlerinin temelini oluşturmakla beraber, kullanılması kolay olmayan bir API (Application Programming Interface)'dir. Kod 5.1 e tekrar göz attığımızda, bir SQL komutunun hazırlanışı ve veri tabanı sistemine aktarılışı için birden fazla sınıfın (Connection, PreparedStatement, ResultSet) kullanılması zorunluluğu bulunduğunu görmekteyiz. Ayrıca finally bloğunda da görüldüğü gibi JDBC ile hata yönetimi kodun okunabilirlik seviyesini düşürmektedir. Bunun sebebi JDBC bünyesinde kontrol edilen (Checked Exceptions) hata sınıflarının kullanılmış olmasıdır. java.lang.Exception sınıfını genişleten bu hata sınıflarının mutlaka ya bir try/catch içinde işlenmeleri ya da throws komutu ile bir üst katmana delege edilmeleri gerekmektedir. Bu ne yazık ki kodun okunabilirliğine, bakılabilirliğine ve geliştirilebilirliğine zarar vermektedir.

JDBC API'si ile olan komplike interaksiyonu gizlemek amacıyla Spring bünyesindeki JdbcTemplate mekanizması kullanılabilir. Kapalı kapılar arkasında JdbcTemplate sırasıyla şu işlemleri gerçekleştirir:

- Veri tabanı bağlantısının (Connection) oluşturulması
- Bir veri tabanı transaksiyonuna dahil olma
- SQL komutlarının işletilmesi
- SQL sonuçlarının işlenmesi
- Oluşan hataların elden geçirilmesi
- Veri tabanı bağlantısının tekrar bırakılması

Bir defa oluşturulan bir JdbcTemplate nesnesi birden fazla SQL operasyonu için kullanılabilir. Ayrıca oluşturulan JdbcTemplate nesnesi threadler arası paylaşılabilir (threadsafe). JDBCCustomerRepositoryImpl sınıfının JdbcTemplate kullanan implementasyonu kod 5.4 de yer almaktadır.

```
Kod 5.4 - JDBCCustomerRepositoryImpl

package com.kurumsaljava.spring.jdbctemplate;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
```

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;

import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;

@Named
public class JDBCCustomerRepositoryImpl
        implements CustomerRepository {

    private JdbcTemplate jdbcTemplate;

    @Inject
    public JDBCCustomerRepositoryImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);

    }

    @Override
    public Customer getCustomerByName(String name) {
        ResultSetExtractor<Customer> customerExtractor =
                            new CustomerExtractor();
        String sql =
                "select id, name, firstname, age
                            from customer where name=?";
        return jdbcTemplate.query(sql,
                    customerExtractor, name);
    }

    @Override
    public void save(Customer customer) {
        String sql = "insert into customer
                    (name, firstname, age) values(?,?,?)";
        jdbcTemplate.update(sql, customer.getName(),
                customer.getFirstname(), customer.getAge());
    }

    private class CustomerExtractor
            implements ResultSetExtractor<Customer> {

        public Customer extractData(ResultSet rs)
                throws SQLException, DataAccessException {
            return mapCustomer(rs);
        }
    }

    private Customer mapCustomer(ResultSet rs)
                    throws SQLException {
        Customer customer = null;
```

```
        if (rs.next()) {
            String name = rs.getString("name");
            String firstname = rs.getString("firstname");
            int age = rs.getInt("age");
            customer = new Customer(name, firstname, age);
            customer.setId(rs.getLong("id"));
        }
        if (customer == null) {
            throw new EmptyResultDataAccessException(1);
        }
        return customer;
    }
}
```

Yeni bir JdbcTemplate nesnesi oluşturabilmek için bir DataSource nesnesi kullanılır. Kod 5.4 de dataSource nesnesi sınıf konstrüktörü üzerinden nesneye enjekte edilmektedir. getCustomerByName() ve save() metotlarında JdbcTemplate kullanılmaktadır. Kod 5.4 ü kod 5.1 ile kıyasladığımızda JdbcTemplate sınıfının kodu ne kadar daha okunur hale getirdigi aşikardır.

getCustomerByName() metodunda jdbcTemplate nesnesi customerExtractor isminde bir nesneyi kullanmaktadır. customerExtractor ResultSetExtractor tipinde olup, ResultSet aracılığı ile (mapCustomer() metodu) customer tablosunun kolonlarından bir Customer nesnesi oluşturmaktadır.

Bir JdbcTemplate nesnesini doğrudan enjekte edebiliriz. Bu amaçla aşağıdaki şekilde jdbcTemplate nesnesinin bir Spring bean olarak tanımlanması yeterli olacaktır.

```
#applicationContext.xml
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="schema.sql" />
    <jdbc:script location="data.sql" />
</jdbc:embedded-database>

<bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>


#JDBCCustomerRepositoryImpl.java
@Named
public class JDBCCustomerRepositoryImpl
        implements CustomerRepository {
```

```
    @Inject
    private JdbcTemplate jdbcTemplate;

    public JDBCCustomerRepositoryImpl() {
    }
    ...
}
```

JdbcTemplate sınıfının sahip olduğu queryForMap() metodu ile veri tabanı tablosunda yer alan bir satırı bir Java Map olarak edinmek mümkündür. Aşağıda yer alan örnek aranan müşteri bilgisini bir Map içinde geri verecektir.

```
public Map getCustomerMapByName(String name) {
    String sql = "select id, name,
            firstname, age from customer where name=?";
    return jdbcTemplate.queryForMap(sql,name);
}

#Map'in icerigi:
Map { id=1, name=Sunal, firstname=Kemal, age=69 }
```

Birden fazla tablo satırı için queryForList() metodu kullanılabilir. Aşağıda yer alan örnekte görüldüğü gibi bu metot Map elementlerinden oluşan bir listeyi geri verir.

```
public List getCustomerListByName() {
    String sql = "select id, name, firstname,
                age from customer";
    return jdbcTemplate.queryForList(sql);
}

#List'in icerigi:
List {
    0 - Map { id=1, name=Sunal, firstname=Kemal, age=69 }
    1 - Map { id=2, name=Müren, firstname=Zeki, age=78 }
    2 - Map { id=3, name=Isik, firstname=Ayhan, age=75 }
}
```

Callback Yöntemleri

Customer tablosunda yer alan bir satırı nasıl bir Customer nesnesine dönüştüreceğimizi kod 5.4 de görmüştük. CustomerExtractor sınıfı yardımı ile mevcut bir ResultSet nesnesini bir Customer nesnesine dönüştürebildik.

Spring'in ihtiva ettiği ResultSetExtractor interface sınıfını implemente eden CustomerExtractor sınıfına Spring terminolojisinde Callback sınıfı ismi verilmektedir. JdbcTemplate gerekli veri tabanı işlemlerini gerçekleştirirken bu sınıfın extractData() metodunu koşturur, yani ingilizce callback yapar.

Bunun yanı sıra tablo satırlarını Java nesnelerine dönüştürmek için RowMapper sınıfı kullanılabilir. Kod 5.5 de görüldüğü gibi RowMapper sınıfı JdbcTemplate sınıfının queryForObject() metodunun kullanımını sağlamaktadır. Veri tabanı işlemi esnasında jdbcTemplate nesnesi CustomerRowMapper sınıfının mapRow() metodunu koşturarak bir Customer nesnesi oluşturulmasını sağlayacaktır.

```
Kod 5.5 - JDBCCustomerRepositoryImpl2

package com.kurumsaljava.spring.jdbctemplate;

import java.sql.ResultSet;
import java.sql.SQLException;
import javax.inject.Inject;
import javax.inject.Named;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;

@Named
public class JDBCCustomerRepositoryImpl2
            implements CustomerRepository {

    @Inject
    private JdbcTemplate jdbcTemplate;

    public JDBCCustomerRepositoryImpl2() {
    }

    @Override
    public Customer getCustomerByName(String name) {
        String sql = "select id, name, firstname,
                age from customer where name=?";
        return jdbcTemplate.queryForObject(sql,
                new CustomerRowMapper(), name);
    }

    private class CustomerRowMapper
                implements RowMapper<Customer> {
```

```
        @Override
        public Customer mapRow(ResultSet rs, int rowNum)
                        throws SQLException {
            return mapCustomer(rs);
        }
    }

    private Customer mapCustomer(ResultSet rs)
                        throws SQLException {
        Customer customer = null;
        String name = rs.getString("name");
        String firstname = rs.getString("firstname");
        int age = rs.getInt("age");
        customer = new Customer(name, firstname, age);
        customer.setId(rs.getLong("id"));

        if (customer == null) {
            throw new EmptyResultDataAccessException(1);
        }
        return customer;
    }
}
```

Kod 5.5 de yer alan getCustomerByName() metodu sadece bir Customer nesnesini geri vermektedir. Birden fazla Customer nesnesi edinmek için aşağıdaki örnekte yer alan query() metodu kullanılabilir.

```
public List<Customer> getCustomerList() {
    String sql = "select id, name,
                firstname, age from customer";
    return jdbcTemplate.query(sql,
                new CustomerRowMapper());
}
```

ResultSetExtractor ve RowMapper callback interface sınıfları yanı sıra üçüncü alternatif olarak RowCallBackHandler kullanılabilir. Bu callback interface sınıfın processRow() isminde herhangi bir veriyi geriye vermeyen (void) bir metodu bulunmaktadır. RowCallBackHandler genelde verileri bir dosyaya yazmak, XML dosyası oluşturmak ya da verileri bir listeye eklemeden filtrelemek için kullanılır. Aşağıda yer alan örnekte customer tablosundan edinilen müşteri bilgileri bir dosyaya yazılmaktadır.

```
public void writeCustomerToFile(String name) {
    RowCallbackHandler handler =
```

```
            new RowCallbackHandler() {
        @Override
        public void processRow(ResultSet rs)
                    throws SQLException {
            StringBuilder temp =
                    new StringBuilder();
            temp.append("name=").append(rs.getString("name")).
                                    append("\n");
            temp.append("firstname=").append(
                    rs.getString("firstname")).append("\n");
            temp.append("age=").append(rs.getInt("age"));
            FileWriter.write(temp);
        }};
    String sql = "select id, name, firstname,
                    age from customer where name=?";
    jdbcTemplate.query(sql, handler, name);
}
```

Hangi Callback Yöntemi Kullanılmalı?

Yakından incelediğimizde ResultSetExtractor ve RowMapper callback sınıflarının yapı itibari ile birbirlerine çok benzediklerini görmekteyiz. Her iki sınıfta ResultSet üzerinden verilere erişmektedir. Doğal olarak aklımıza hangi sınıfı ne için kullanabiliriz sorusu gelmektedir. RowMapper bir veri tabanı tablosunda yer alan bir satırı birebir bir alan nesnesine (Domain Object) dönüştürmek için kullanılır. Örneğin aşağıda yer alan getCustomerList() metodu customer tablosunda yer alan her satır için bir Customer nesnesi oluşturur. Bu amaçla customer tablosunda yer alan kayıt adedi kadar CustomerRowMapper sınıfının mapRow() metodu koşturulur. mapCustomer() metodu bünyesinde ResultSet nesnesi rs sadece kolon bilgilerini okumak için kullanılır. Bu metot bünyesinde rs.next() metodunun kullanılması mevcut satır düzenini bozar, çünkü rs.next() mapRow() metodunu koşturan Spring çatısı tarafından yapılmaktadır.

```
public List<Customer> getCustomerList() {
    RowMapper<Customer> customerRowMapper =
            new CustomerRowMapper();
    String sql = "select id, name,
            firstname, age from customer ";
    return jdbcTemplate.query(sql,
                customerRowMapper);
}

private class CustomerRowMapper
```

```
            implements RowMapper<Customer> {
    @Override
    public Customer mapRow(ResultSet rs, int rowNum)
                        throws SQLException {
        return mapCustomer(rs);
    }
}

private Customer mapCustomer(ResultSet rs)
                    throws SQLException {
    Customer customer = null;
    String name = rs.getString("name");
    String firstname = rs.getString("firstname");
    int age = rs.getInt("age");
    customer = new Customer(name, firstname, age);
    customer.setId(rs.getLong("id"));
    return customer;
}
```

Birden fazla tablo satırını kullanarak bir alan nesnesi oluşturmamız gerektiğinde ResultSetExtractor interface sınıfını kullanabiliriz. Bu sınıfın extractData() metodunda ResultSet nesnesi üzerinde gerekli tüm işlemleri yapabilir ve alan nesnesini yapılandırabiliriz. Bu çoğu zaman komplike yapıdaki nesneleri oluşturmak için kullanılan bir yöntemdir. Ama RowMapper örneğinde olduğu gibi bu yöntemle aynı tipte olan nesnelerden oluşan bir liste oluşturmak mümkün değildir.

Calback metodundan herhangi bir geri dönüş değeri beklenilmeyen durumlarda RowCallbackHandler sınıfı kullanılabilir.

NamedParameterJdbcTemplate Kullanımı

Aşağıda yer alan koda baktığımızda, SQL sorgusunda yer alması istenilen değerlerin ? işareti ile tanımlandığını görmekteyiz. ? işaretiyle tanımlı alana herhangi bir değer atamak için JdbcTemplate sınıfında yer alan queryForObject() metodu name isminde bir metot parametresi almaktadır. SQL sorgusunda bu şekilde birden fazla ? işareti kullanılabilir ve queryForObject metodu aracılığı ile bu alanlara değer atanabilir.

```
public Customer getCustomerByName(String name) {
    String sql = "select id, name, firstname,
            age from customer where name=?";
    return jdbcTemplate.queryForObject(sql,
```

```
            new CustomerRowMapper(), name);
}
```

? ile işaretli alanlara isim vermek için isimli parametreler (named parameters) kullanılabilir. NamedParameterJdbcTemplate sınıfı isimli parametrelere sahip SQL komutlarına gerekli parametre değerlerinin atanarak JdbcTemplate sınıfı aracılığı ile koşturulmalarını sağlar. Bir sonraki kod örneği NamedParameterJdbcTemplate kullanımını göstermektedir.

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate =
        new NamedParameterJdbcTemplate(dataSource);
}

public String getCustomerName(long customerId) {
    String sql = "select name from customer where id = :id";

    SqlParameterSource namedParameters =
        new MapSqlParameterSource("id", id);

    return this.namedParameterJdbcTemplate.
        queryForObject(sql, String.class, namedParameters)
}
```

Yukarıda yer alan örnekte isimli parametrelere değer atamak için MapSqlParameterSource sınıfı kullanılmaktadır. Buna alternatif olarak parametre değerleri bir Map olarak şu şekilde doğrudan NamedParameterJdbcTemplate nesnesine verilebilir:

```
public String getCustomerName(long customerId) {
    String sql = "select name from customer where id = :id";

    Map<String, String> namedParameters =
            Collections.singletonMap("id", id);

    return this.namedParameterJdbcTemplate.
        queryForObject(sql, String.class, namedParameters)
}
```

Bir önceki örnekte isimli parametre listesini MapSqlParameterSource sınıfını kullanarak oluşturmuştuk. Bu şekilde parametre isim ve değerleri için bir map nesnesi oluşturulmaktadır. MapSqlParameterSource sınıfının en üst sınıfı

SqlParameterSource isminde bir interface sınıftır.

Diğer ilginç bir SqlParameterSource implementasyonu BeanPropertySqlParameterSource sınıfır. BeanPropertySqlParameterSource sınıfı ile JavaBean konvensiyonuna uygun herhangi bir POJO sınıfı kapsüllenerek, SQL sorgusu için parametre isimlerinin ve değerlerinin doğrudan bu POJO sınıf değişkenlerinden oluşturulması sağlanabilir. Aşağıda yer alan örnekte sql değişkeninde yer alan isimli parametrelerin değerleri customer sınıfında yer alan parametrelerle aynı isimleri paylaşan değişlenlerden edinilmektedir.

```
package com.kurumsaljava.spring.domain;
public class Customer {

    private long id;
    private String name;
    private String firstname;
    private int age;

    ... // set ve get metotları
}

public long getCustomerId(Customer customer) {

    String sql = "select id from customer
        where name = :name" and firstname = :firstname;

    SqlParameterSource namedParameters =
        new BeanPropertySqlParameterSource(customer);

    return this.namedParameterJdbcTemplate.
        queryForObject(sql, long.class, namedParameters);
}
```

SimpleJdbcInsert Kullanımı

SimpleJdbcInsert sınıfı ile veri tabanı tablolarını oluşturan meta bilgileri kullanarak çok daha sade SQL sorguları oluşturmak mümkündür.

```
create table customer (
    id integer identity primary key,
    name varchar(50),
    firstname varchar(50),
    age integer);
```

Yukarıda yer alan customer tablosuna yeni bir müşteri kaydı eklemek istediğimizi düşünelim. SimpleJdbcInsert sınıfını kullanarak veri ekleme (insert) işlemini şu şekilde kodlayabilirdik:

```
public class CustomerRepositoryImpl implements CustomerRepository{

    public void setDataSource(DataSource dataSource) {
        this.insertCustomer =
                new SimpleJdbcInsert(dataSource)
                            .withTableName("customer");
    }

    public void addCustomer(Customer customer) {
        Map<String, Object> parameters = new HashMap<String,
                                                    Object>(4);
        parameters.put("id", customer.getId());
        parameters.put("name", customer.getName());
        parameters.put("firstname", customer.getFirstName());
        parameters.put("age", customer.getLastName());
        insertCustomer.execute(parameters);
    }
}
```

Yukarıda yer alan örnekte CustomerRepositoryImpl sınıfına setDataSource() metodu üzerinden bir DataSource nesnesi enjekte edilmektedir. Bu nesne kullanılarak bir SimpleJdbcInsert nesnesi oluşturulmaktadır. Bu sınıf bünyesinde yer alan withTableName() metodu aracılığı ile kullanılacak veri tabanı tablosu tanımlanmaktadır. addCustomer() metodu aracılığı ile parameters bünyesinde yer alan müsteri bilgileri insertCustomer nesnesi aracılığı ile customer tablosuna eklenmektedir. Parameters bünyesinde yer alan anahtarlar customer tablosunun kolon isimleridir. SimpleJdbcInsert parameters bünyesindeki anahtar isimlerini customer tablosu kolon isimlerine eşitleyerek gerekli SQL sorgusu yapısının oluşmasını sağlamaktadır. SimpleJdbcInsert tablonun kolon isimlerini JDBC sürücüsü aracılığı ile veri tabanı sisteminden edinmektedir.

Daha önce tanıştığımız BeanPropertySqlParameterSource ile customer nesnesinde yer alan değişken isimlerini kolon isimleri, taşıdıkları değerleri SQL komutunda yer alan değerler olarak kullanabiliriz.

```
public void addCustomer(Customer customer) {
    SqlParameterSource parameters =
        new BeanPropertySqlParameterSource(customer);
```

```
    insertCustomer.execute(parameters);
}
```

Aynı işlemi MapSqlParameterSource sınıfını kullanarak ta yapabiliriz.

```
public void addCustomer(Customer customer) {
    SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("name", customer.getName())
            .addValue("firstname", customer.getFirstName());
    Number newId = insertActor.executeAndReturnKey(parameters);
    actor.setId(newId.longValue());
}
```

JDBC Batch İşlemleri

Şimdiye kadar gördüğümüz örneklerde tek bir SQL komutu JDBC sürücüsü (driver) aracılığı ile veri tabanı sistemine gönderilerek, koşturuldu. Her defasında tek bir SQL komutu ile veri tabanı sistemine gitmek, düşünebileceğiniz gibi performslı bir işlem değildir. Performansı artırmak için birden fazla SQL komutu topluca veri tabanı üzerinde koşturulabilir. Topluca SQL koşturma işlemine batch JDBC processing ismi verilmektedir.

Batch JDBC işlemlerini JdbcTemplate sınıfı aracılıgı ile gerçekleştirebiliriz. Aşağıda yer alan örnekte customers isimli listede yer alan her müşteri nesnesi için bir update komutu oluşturulmaktadır. Batch işlemi JdbcTemplate bünyesinde yer alan batchUpdate() metodu üzerinden yapılmaktadır.

```
public class CustomerRepositoryImpl implements CustomerRepository{

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
       this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] addCustomerInBatch(List<Customer> customers) {

        int[] updateCounts = jdbcTemplate.batchUpdate(
            "update customer set firstname = ?, name = ? where id = ?",
            new BatchPreparedStatementSetter() {
                public void setValues(PreparedStatement ps, int i)
                        throws SQLException {
                    ps.setString(1, customers.get(i).getFirstName());
                    ps.setString(2, customers.get(i).getLastName());
```

```
                    ps.setLong(3, customers.get(i).getId().longValue());
                }

                public int getBatchSize() {
                    return customers.size();
                }
            } );
        return updateCounts;
    }
}
```

batchUpdate() metoduna baktığımızda bir SQL komutu ve BatchPreparedStatementSetter sınıfından bir nesneyi parametre olarak aldığını görmekteyiz. BatchPreparedStatementSetter sınıfı batch SQL komutların oluşturulduğu sınıftır. setValues() bünyesinde sıradaki customer nesnesi için update SQL komutu oluşturulmaktadır. Her customer nesnesi için bir update SQL komutu oluşturulduktan sonra, bu komutlar topluca JDBC sürücüsü üzerinden veri tabanı sistemine aktarılmaktadır. Bu şekilde tek bir işlemde birden fazla SQL komutu koşturulmaktadır.

Hata Yönetimi

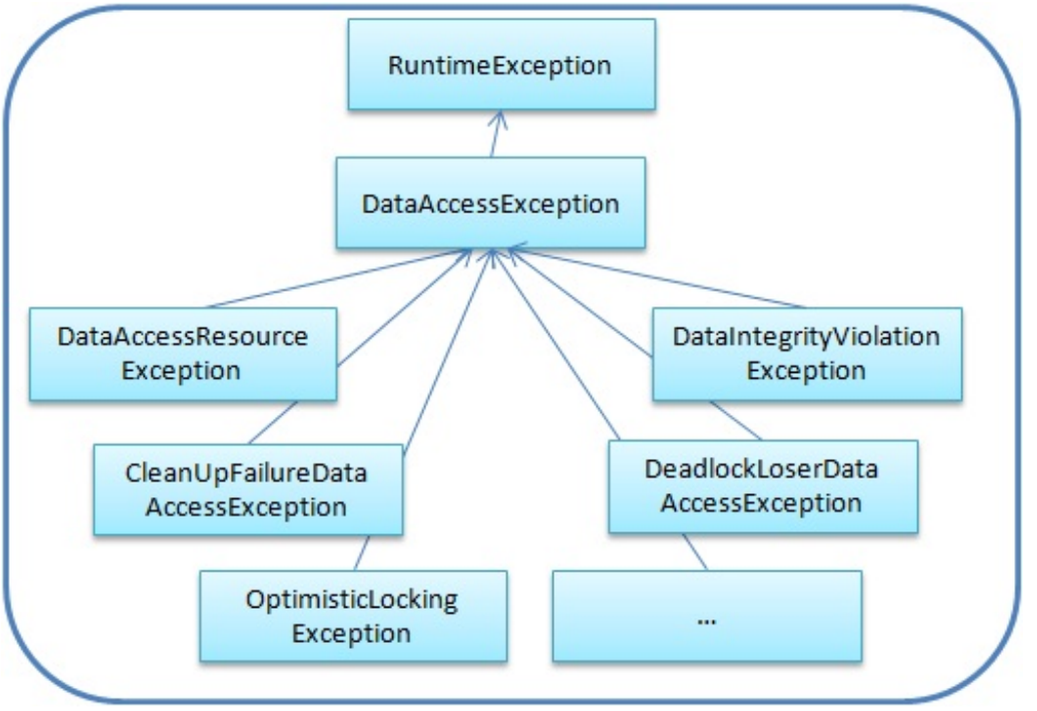
Java dünyasında hatalar (Exception) kontrol edilen (checked) ve kontrol edilmeyen (unchecked) hatalar olarak ikiye ayrılır. Kontrol edilen hataların başını java.lang.Throwable ve onun altsınıfı olan java.lang.Exception çeker. Bu tür hata sınıflarının kullanımı derleyici (compiler) tarafından kontrol edilir. Kontrol edilen hataların ya try/catch ile işlenmesi ya da throws komutu ile metot kullanıcısına delege edilmesi gerekmektedir.

Kontrol edilmeyen hatalar java.lang.RuntimeException sınıfı ve türevlerinden oluşur. Bu tür hataların try/catch içinde işlenmesi ya da throws ile bir üst katmana delege edilme zorunlulukları bulunmamaktadır. Programcı istediği seviyede oluşan hatayı yakalayarak gerekli hata yönetimini yapabilir. Bu birçok metot bünyesinde try/catch kullanılmaması anlamına gelmektedir. Böylece kodun okunabilirlik seviyesi yükselmektedir. Spring çatısı mümkün olan her yerde kontrol edilmeyen hata türlerini kullanmaktadır.

JDBC API oluşturulurken ne yazık ki fazlasıyla kontrol edilen hata türleri kullanılmıştır. Bunun neticesi olarak kod 5.1 de gördüğümüz hata yönetim kodlama tarzı mecburi hale gelmiştir. Bu kodun yazılmasını ve bakımını zorlaştıran bir durumdur.

JDBC ile geliştirilen uygulamalarda kontrol edilen SQLException hata sınıfına sıkça rastlamak mümkündür. Kod 5.1 de yer alan getCustomerByName() metodunda da bir catch bloğu içinde böyle bir hata nesnesini yakalamak ve işlemek zorunda kaldık. Jenerik olan SQLException sınıfı JDBC tarafından her türlü veri tabanı hatası için kullanılır. Çoğu zaman ilk bakışta ne türlü bir hatanın oluştuğunu anlamak mümkün değildir. Ayrıca bu hata ile karşılaşan bir uygulama modülü, alt tarafta JDBC ile çalışıldığından haberdar olmakta ve JDBC çatısına olan bağımlılık artmaktadır. Bunun yanı sıra JDBC ile edinilen tecrübeler her SQLException hatasının catch içinde işlenmesinin anlamlı olmadığını göstermektedir. Oluşan hataların büyük bir bölümü uygulamanın doğru çalışmasına engel olucu niteliktedir. Bir veri tabanı bağlantısının oluşturulamaması ya da bir SQL komutunun hatalı olması durumunda uygulama geliştiricisinin catch bloğu içinde yapabileceği fazla bir şey yoktur.

Spring bünyesinde SQLException yerine hangi veri erişim çatısı kullanılırsa kullanılsın değişmeyen, değişik hata türlerini yansıtan ve kontrol edilmeyen türde olan hata sınıflarını barındırmaktadır. Resim 5.3 de Spring'in kullandığı Exception hiyerarşisinden bir örnek yer almaktadır. Hemen hemen oluşabilecek her türlü hata için bir Exception sınıfı tanımlanmıştır. Hangi veri tabanı erişim çatısı (Hibernate, IBAtis vs.) kullanılırsa, kullanılsın Spring oluşan veri tabanı hatalarının hepsini aynı Excpetion sınıfları aracılığı ile yönetir.



Resim 5.3

Spring'in sunduğu Exception hiyerarşisinin en tepesinde DataAccessException

sınıfı yer alır. DataAccessException kontrol edilmeyen (unchecked Exception) türde bir hata sınıfıdır. Bu yüzden oluşan bu tür hataların try/catch içinde işlenme ya da throw ile bir üst katmana gönderilme zorunluluğu bulunmaktadır. Bu sebepten dolayı kod 5.4 de yer alan getCustomerByName() metodu çok sade bir yapıdadır, çünkü arka planda JdbcTemplate kontrol edilmeyen Spring hata sınıflarını kullanmaktadır. Spring JdbcTemplate kullanıldığında oluşan her SQLException hatasını yakalayarak, DataAccessException ya da türevlerinden birisine dönüştürecektir. Yazılımcı istediği taktirde bir catch blogu içinde bu hatayı yakalayarak, işleyebilir. Kod 5.6 da JdbcTemplate bünyesinde kullanılan execute() metodunda oluşabilecek bir SQLException hatasının getExceptionTranslator().translate() metodu aracılığı ile nasıl bir DataAccessException ya da türevini dönüştürüldüğünü görmekteyiz.

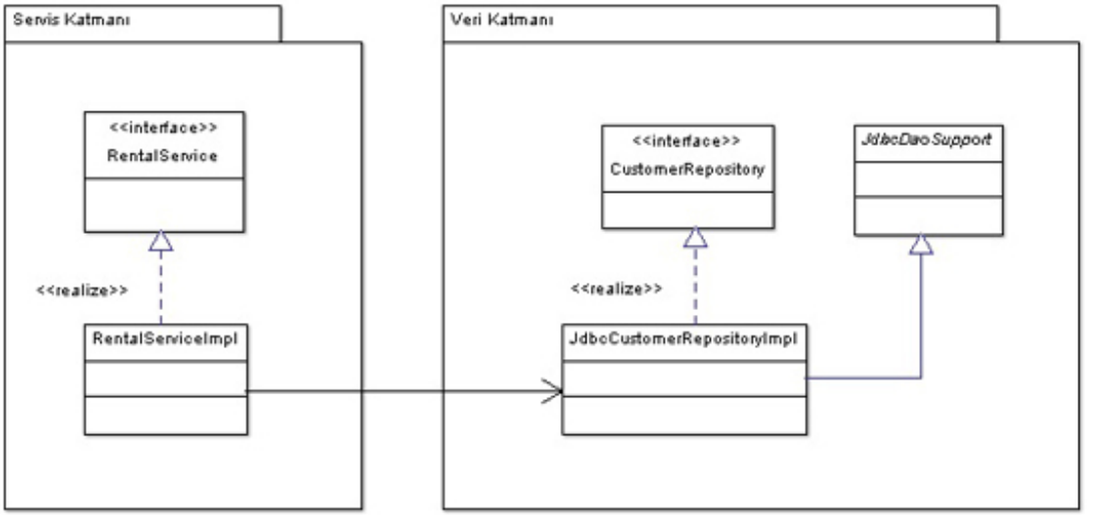
```
Kod 5.6 - org.springframework.jdbc.core.JdbcTemplate

public <T> T execute(ConnectionCallback<T> action)
                throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");

    Connection con =
            DataSourceUtils.getConnection(getDataSource());
    try {
        Connection conToUse = con;
        if (this.nativeJdbcExtractor != null) {
            conToUse =
                this.nativeJdbcExtractor.getNativeConnection(con);
        }
        else {
            conToUse = createConnectionProxy(con);
        }
        return action.doInConnection(conToUse);
    }
    catch (SQLException ex) {
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        throw getExceptionTranslator().translate("
                    ConnectionCallback", getSql(action), ex);
    }
    finally {
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}
```

JDBC ve Spring DAO Support

DAO'nun açılımı Data Access Object'tir ve veri tabanı işlemlerinin izolasyonu için kullanılan bir tasarım şablonudur.

Şimdiye kadar kullanmış olduğumuz CustomerRepository ya da RentalRepository sınıfları birer DAO sınıfıdır. Servis katmanında yer alan RentalService sınıfı veri katmanında yer alan CustomerRepository ya da RentalRepository sınıfını kullanır. Bu repository sınıfları veri tabanı işlemlerini kendi bünyelerinde gizlediklerinden, servis katmanında yer alan RentalService sınıfını veri tabanına hiç bir bağımlılığı olmadan repository sınıfları üzerinden veri tabanı işlemlerini gerçekleştirir. Buna katmanlı mimari ismi verilmektedir.

Katmanlar birbirlerini kullanacak şekilde oluşturulur. Her katman bir interface sınıfı aracılığı ile servislerini dış dünyaya sunar. Bu interface sınıfları değişik türde implemente ederek, diğer katmanları etkilemeden katmanın çalışma tarzı değiştirilebilir. Bu şekilde katmanlar arası esnek bir bağ oluşturmak ve yeri geldiğinde bir katmanın implementasyonunu diğer katmanları etkilemeden değiştirmek mümkündür.



Resim 5.4

JdbcTemplate kullanımını kolaylaştırmak amacıyla Spring JdbcDaoSupport ismini taşıyan bir sınıf ihtiva etmektedir. Bu sınıfı genişlettiğimiz taktirde Spring sınıf nesnemize JdbcDaoSupport sınıfı bünyesinde yer alan setJdbcTemplate() metodu aracılığı ile JcbdTemplate nesnesini enjekte eder. JdbcDaoSupport sınıfını genişleten repository implementasyonumuz kod 5.7 de yer almaktadır. Kod 5.8 de olduğu gibi customerRepository nesnesine bir

dataSource nesnesinin enjekte edilmesi gerekmektedir.

```
Kod 5.7-  JDBCCustomerRepositoryImpl

package com.kurumsaljava.spring.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;

public class JDBCCustomerRepositoryImpl
            extends JdbcDaoSupport implements
        CustomerRepository {

    public JDBCCustomerRepositoryImpl() {
    }

    @Override
    public Customer getCustomerByName(String name) {
        ResultSetExtractor<Customer> customerExtractor =
                        new CustomerExtractor();
        String sql = "select id, name, firstname,
                age from customer where name=?";
        return getJdbcTemplate().query(sql,
                customerExtractor, name);
    }

    @Override
    public void save(Customer customer) {
        String sql = "insert into customer (name,
                        firstname, age) values(?,?,?)";
        getJdbcTemplate().update(sql, customer.getName(),
                customer.getFirstname(), customer.getAge());
    }

    private class CustomerExtractor
        implements ResultSetExtractor<Customer> {

        public Customer extractData(ResultSet rs)
                    throws SQLException,
                DataAccessException {
            return mapCustomer(rs);
        }
```

```
    }

    private Customer mapCustomer(ResultSet rs)
                throws SQLException {
        Customer customer = null;
        while (rs.next()) {
            if (customer == null) {
                String name = rs.getString("name");
                String firstname = rs.getString("firstname");
                int age = rs.getInt("age");
                customer = new Customer(name, firstname, age);
                customer.setId(rs.getLong("id"));
            }
        }
        if (customer == null) {
            throw new EmptyResultDataAccessException(1);
        }
        return customer;
    }
}

Kod 5.8 - applicationContext.xml

<beans>
    <context:annotation-config />
    <context:component-scan
        base-package="com.kurumsaljava.spring." />

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql" />
        <jdbc:script location="data.sql" />
    </jdbc:embedded-database>

    <bean id="customerRepository"
        class="com.kurumsaljava.spring.dao.
            JDBCCustomerRepositoryImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

DataSource Konfigürasyonu

Kod 5.3 de jdbc isim alanında yer alan embedded-database elementi yardımı ile hafızada çalışan bir HSQLDB oluşturmuş ve bir dataSource nesnesi tanımlamıştık. Böyle bir konfigürasyonun kullanımı uygulama geliştirme

sürecini kısaltacağı gibi, hafif bir veri tabanı sistemi ile uygulamanın gerçek şartlara yakın bir seviyede test edilmesini mümkün kılacaktır.

embedded-database sadece uygulama geliştirilirken kullanılabilecek bir dataSource konfigürasyon türüdür. Uygulama gerçek şartlarda kullanıcı hizmetine sunulduğunda, ihtiyaç duyulan dataSource konfigürasyonu değişik olacaktır. Uygulamanın çalışma ortamına göre Spring'in sunduğu değişik türdeki dataSource konfigürasyon imkanları kullanılabilir. Genel olarak kullanılabilecek üç değişik dataSource konfigürasyonu bulunmaktadır. Bunlar:

- JDBC sürücüsü kullanılarak yapılan tanımlamalar,
- JNDI kullanılarak yapılan tanımlamalar,
- Bağlantı havuzu (Connection Pooling) oluşturmayı mümkün kılan tanımlamalar.

JDBC Sürücüsü İle DataSource Tanımlaması

Spring ihtiva ettiği DriverManagerDataSource ve SingleConnectionDataSource sınıfları ile JDBC sürücü tabanlı dataSource konfigürasyonunu mümkün kılmaktadır. Kod 5.9 da DriverManagerDataSource kullanılarak bir dataSource tanımlaması yapılmıştır. Böyle bir dataSource tanımlaması her veri tabanı bağlantı isteği için yeni bir Connection nesnesi oluşturacaktır.

```
Kod 5.9 - applicationContext.xml

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.
                DriverManagerDataSource">
    <property name="driverClassName"
                value="org.gjt.mm.mysql.Driver" />
    <property name="url"
                value="jdbc:mysql://127.0.0.1/rentacar" />
    <property name="username" value="root" />
    <property name="password" value="rentacar" />
</bean>
```

SingleConnectionDataSource ile yapılan dataSource tanımlamaları sadece bir Connection nesnesinin kullanımına izin verir. Genelde uygulamayı test etmek için bu tür bir dataSource konfigürasyonu tavsiye edilmektedir. Uygulama her zaman aynı Connection nesnesini kullanır. Kod 5.10 da SingleConnectionDataSource ile yapılan bir dataSource tanımlaması yer almaktadır.

```
Kod 5.10 - applicationContext.xml

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.
                SingleConnectionDataSource">
    <property name="driverClassName"
                value="org.gjt.mm.mysql.Driver" />
    <property name="url"
                value="jdbc:mysql://127.0.0.1/rentacar" />
    <property name="username" value="root" />
    <property name="password" value="rentacar" />
</bean>
```

Her iki alternatifin küçük çaplı uygulamalar haricinde kullanımı tavsiye edilmemektedir. DriverManagerDataSource her isteğe karşı yeni bir Connection nesnesi oluşturduğundan veri tabanı işlemleri Connection nesnesinin oluşturulma sürecine doğru orantılı olarak uzayacaktır. Ayrıca DriverManagerDataSource Connection nesnelerinin tekrar kullanımını mümkün kılmadığı için bu durum kullanıcı sayısı yüksek olan uygulamalarda Connecction nesne adedinin patlamasına ve veri tabanı sisteminin bir noktadan itibaren yeni bağlantıları kabul edemez hale gelmesine sebep verebilir. Böyle bir sorun SingleConnectionDataSource kullanıldığında ortaya çıkmamakla birlikte, tek bir Connection nesnesi ile aynı anda birden fazla kullanıcıya hizmet sunmak mümkün olmayacaktır.

JNDI DataSource Kullanımı

Tomcat, WebLogic ya da Websphere gibi uygulama sunucularında dataSource tanımlamaları JNDI (Java Naming And Directory Interface) kullanılarak yapılır. JNDI aracılığı ile veri tabanı işlemlerinin yapılması uygulamayı bunun için gerekli olan konfigürasyondan bağımsız kılar. Bir Spring uygulaması jee isim alanında bulunan jndi-lookup elementi aracılığı ile kod 5.11 de yer aldığı gibi bir dataSource tanımlaması yapabilir.

```
Kod 5.11 - applicationContext.xml

<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/RentACarDS"
    resource-ref="true" />
```

Böyle bir dataSource konfigürasyonuna sahip bir Spring uygulaması ihtiyaç duyduğu zaman uygulama sunucusundan JNDI aracılığı ile bir dataSource nesnesi edinecek ve veri tabanı işlemlerini gerçekleştirecektir. Bu işlemlerin

koordinasyonunu uygulama sunucusu üstlenecektir.

JNDI bazlı dataSource tanımlamaları genelde bağlantı havuzu (Connection Pooling) mekanizmasına sahiptir. Uygulama bir Connection nesnesi talep ettiğinde, bu istek Connection nesnelerinin yer aldığı bir havuzdan karşılanır. Connection nesnelerinin oluşturulmaları ve sonlandırılmaları zaman alıcı bir işlem olduğundan, bu isteklerin bir havuzdan karşılanması uygulamanın işlem hızını artırır. JNDI bazlı dataSource kullanan Spring uygulamaları böyle bir artı hizmete sahip olurlar. Bunun yanı sıra kullanılan JNDI ismi (jndi-name) değişmediği sürece, bu isim altında yapılan konfigürasyon ayarlarındaki değişiklikler uygulamayı etkilemeyecektir. Uygulama böylece ihtiyaç duyulan gerçek dataSource tanımlamasından tamamen bağımsız hale gelir. JNDI bazlı dataSource kullanımını internet adresleri (domain name) ile kıyaslayabiliriz. Adresin sahip olduğu IP adresinin değiştirilmesi bu adresi tanıyan kullanıcıları etkilemeyecektir.

Jndi-lookup elementi JNDI bünyesinde tanımlanmış herhangi bir nesneyi edinmek için kullanılabilir.

Bağlantı Havuzlu DataSource Tanımlaması

JNDI tabanlı dataSource konfigürasyonunun kullanımı mümkün olmadığı durumlarda bağlantı havuzu (Connection Pooling) mekanizmasına sahip bir dataSource konfigürasyonu oluşturulabilir. Sadece Spring kullanılarak böyle bir dataSource konfigürasyonu oluşturulması mümkün değildir. Çoğu zaman böyle bir konfigürasyon için DBCP ya da c3po gibi açık kaynaklı bir araç kullanılmaktadır. Kod 5.12 de yer alan örnekte DBCP'nin sahip olduğu BasicDataSource sınıfı kullanılarak bağlantı havuzu mekanizmasına sahip bir dataSource tanımlaması yapılmıştır.

```
Kod 5.12 - applicationContext.xml

<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="url"
                value="jdbc:oracle:thin:@localhost:1521:RENT_ACAR" />
    <property name="driverClassName"
                value="oracle.jdbc.driver.OracleDriver" />
    <property name="username" value="root" />
    <property name="password" value="rentacar" />
    <property name="removeAbandoned" value="true" />
    <property name="initialSize" value="20" />
```

```
    <property name="maxActive" value="30" />
</bean>
```

Konfigürasyon esnasında kullanılabilecek bazı parametreler şunlardır:

- ***initialSize*** ile başlangıçta bağlantı havuzunda kaç bağlantı nesnesinin olması gerektiği tayin edilir. Kod 5.12 de yer alan dataSource nesnesi oluşturulduğunda havuz içinde yirmi adet *Connection* nesnesi yer alır.
- ***maxActive*** en fazla kaç bağlantı nesnesinin aynı anda kullanılabileceğini belirler. Bu parametre bağlantı havuzunun üst sınırını tayin eder. MaxActive için negatif bir değerin kullanılması bu limiti kaldırır.
- ***maxIdle*** en fazla kaç bağlantı nesnesinin kullanılmadan aynı anda havuz içinde kalabileceğini belirler. maxIdle için negatif bir değerin kullanılması bu limiti kaldırır.
- ***minIdle*** en az kaç bağlantı nesnesinin kullanılmadan aynı anda havuz içinde kalabileceğini belirler. MaxIdle için negatif bir değerin kullanılması bu limiti kaldırır.
- ***maxWait*** havuz içinde bağlantı nesnesi kalmadığı taktirde bağlantı havuzunu yöneten mekanizmanın kullanımda olan bir bağlantı nesnesinin havuza tekrar geri iade edilmesi için beklemesi gereken zamanı (milisaniye) belirler. MaxWait ile tanımlanan zaman birimi tamamlandığında bağlantı havuzunu yöneten mekanizma tarafından hata (Exception) oluşturulur. Negatif bir değerin kullanılması sonsuza dek beklemeyi sağlar.
- ***validationQuery*** bağlantı nesnesinin geçerliliğini test etmek için kullanılan SQL komutu ihtiva eder. Havuzdan bir bağlantı nesnesi kullanıcıya verilmeden önce bu SQL komutu ile bağlantı nesnesinin geçerliliği test edilir.
- ***testOnBorrow*** parametresi true değerine sahip ise, bağlantı nesnesi kullanıcıya verilmeden önce validationQuery ile tanımlanmış SQL komutu ile test edilir. Hatalı olan bağlantı nesneleri bağlantı havuzundan uzaklaştırılır.
- ***testOnReturn*** parametresi true değerine sahipse bağlantı havuza iade edilmeden validationQuery ile tanımlanmış SQL komutu ile test edilir.

Diğer konfigürasyon parametreleri için bakınız.

5. Bölüm Soruları

5.1 Hafızada çalışan bir veri tabanı oluşturmak için kullanılan Spring konfigürasyon elementi hangisidir?

5.2 JDBC işlemlerini yapmak için kullanılan Spring sınıfı hangisidir?

5.3 Neden JDBC yerine JdbcTemplate kullanımı tercih edilmelidir?

5.4 Bir veri tabanı tablosunda yer alan bir satır (record) alan nesnesine nasıl dönüştürülür?

5.5 DAO tasarım şablonu JdbcTemplate kullanılarak nasıl uygulanır?

5.6 SingleConnectionDataSource kullanımında nasıl bir sorunla karşılaşabiliriz?

6. Bölüm

Spring İle Transaksiyon Yönetimi

Transaksiyon Nedir?

Veri tabanı sistemleri ile çalışırken bilinmesi gereken en önemli konseptlerden birisi, veri tabanı işlemlerinde kullanılan transaksiyon mantığıdır. Bir veri tabanı transaksiyonu kullanıcının bir görevi yerine getirmek için kullandığı birden fazla SQL komutunun bir bütün olarak işlenmesini sağlayan mekanizmadır. Tüm SQL komutlarının başarılı bir şekilde koşturulmuş olmaları transaksiyonu commit olarak ifade edilen ve yapılan tüm değişiklikleri bir bütün olarak veri tabanına kaydeden komut ile sonlandırılır. SQL komutlarından herhangi birinin başarısız olması durumunda rollback komutu ile aynı transaksiyon bünyesinde yer alan diğer SQL komutlarının yaptığı tüm değişiklikleri geri alınır. Örneğin bir işlem için dört değişik SQL komutu kullanıldıysa ve dördüncü SQL komutu koşturulurken bir hata oluştuysa, daha önce koşturulmuş olan üç SQL komutun yaptığı değişiklik geçersiz sayılır ve bu değişiklikler veri tabanı sistemi tarafından kaydedilmez. Bu şekilde veri tabanı sisteminde yer alan verilerin tutarlı kalmaları sağlanmış olur.

Veri tabanı bazlı uygulamalarda transaksiyon yönetiminin ne kadar önemli ve gerekli olduğunu gerçek hayattan bir örnek vererek vurgulamak istiyorum. Bir bankada bir müşteri hesabından diğer bir müşteri hesabına para aktarılması en çok yapılan işlemlerden bir tanesidir. Böyle bir işlem iki bölümden oluşmaktadır. Birinci bölümde bir müşterinin hesabından transfer edilecek miktar kadar alıntı yapılır. Bu bir SQL Update komutudur. İşlemin ikinci bölümünde müşterinin hesabından alınan miktar kadar diğer müşterinin hesabına eklenti yapılır. Bu da bir SQL Update komutuna denk gelmektedir. Bu iki SQL komutu mantıksal olarak tek bir işlemdir, yani belirli bir miktarın iki hesap arasında transfer edilmesi işlemi. Bu iki komutun ya hep, ya hiç mantığı ile koşturulmaları gerekmektedir. Aksi taktirde müşteri hesaplarında tutarsızlık oluşabilir. Örneğin birinci SQL komutunun başarıyla tamamlandığını düşünelim. Bu durumda müsterinin hesabından belirli bir miktar alıntı yapılmış olacaktır. Eğer ikinci SQL komutu başarısız olursa ve birinci SQL komutu ile yapılan değişilik geri alınmaz ise, hesaplar arası para transferi yapılmadığı halde birinci müşterinin hesabından transfer için düşünülen miktar alınmış olur. Bu da doğruyu yansıtmaz. Bu sebepten dolayı bu iki işlemi tek bir işlemiş gibi uygulamak için bir veri tabanı transaksiyonunun kullanılması zaruridir. Oluşan herhangi bir hata yapılan işlemlerin geriye alınmasını tetikler ve verilerin tutarsız hale gelmesini önler.

ACID Özelliği

Verilerin tutarlılıklarının korunabilmesi için transaksiyonları destekleyen bir veri tabanı sistemi birden fazla SQL komutundan oluşan bir işlemi yaparken verilere ilişkin temel dört özelliğin korunmasını garanti eder. Bunlar atomik işlem (***A***tomicity), veri tutarliligi (***Consistency), izolasyon (I***solation) ve uzun ömürlülüktür (***D***urability).

- ***A***tomicity (atomik işlem): Birden fazla SQL komutundan oluşan işlemin ya hep, ya hiç mantığı ile çalışmasını sağlar.
- ***C***onsistency (veri tutarlılığı): Her transaksiyon sonunda verilerin tutarlılığını korur.
- ***I***solation (izolasyon): Paralel koşturulan transaksiyonların birbirlerini etkilememesi için birbirlerinden izole bir şekilde çalışabilmeleri sağlar.
- ***D***urability (uzun ömürlülük): Transaksiyon bünyesinde veriler üzerinde yapılan değişikliklerin veri tabanında kaydedilmesini ve transaksiyon sonunda bu değişikliklerin görünür hale gelmesini sağlar.

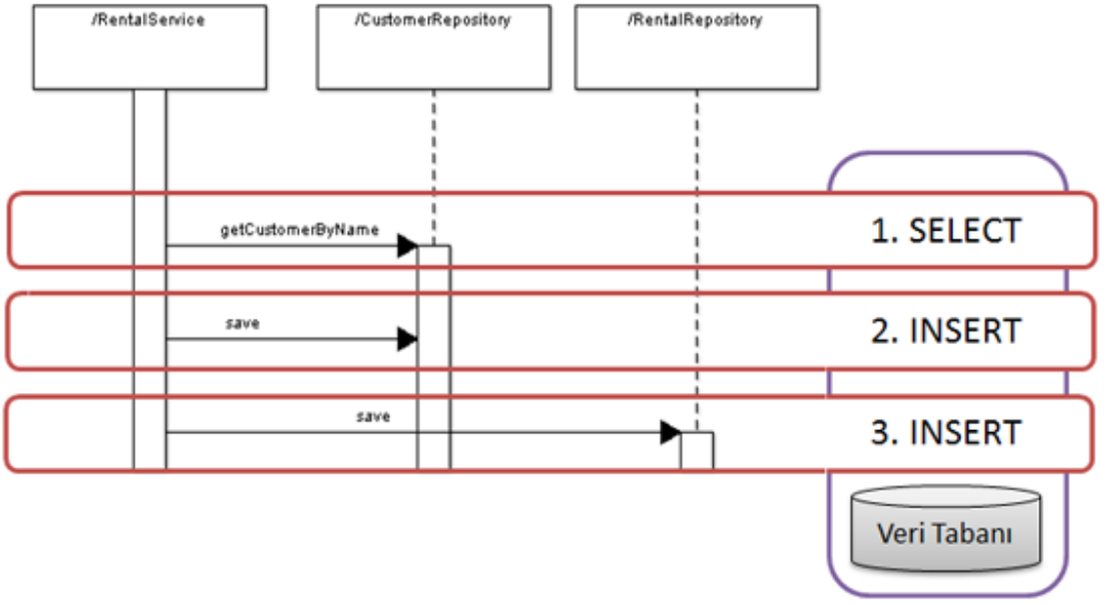
Atomik İşlem

RentalServiceImpl sınıfının rentACar() isimli metodunda müşteri mevcudiyetine bağlı olarak iki ya da üç değişik veri tabanı işlemi yapılmaktadır. Müşteri bilgilerinin veri tabanında olmaması durumunda müşteri bilgileri customerRepository.save() metodu aracılığı ile customer tablosuna kaydedilmektedir. Toplamda yapılabilecek veri tabanı işlemleri resim 6.1 de yer almaktadır.

```
Kod 6.1 - RentalServiceImpl

public Rental rentACar(Customer customer, Car car,
                Date begin, Date end) {
    Customer dbCustomer =
    customerRepository.getCustomerByName(customer.getName());

    if (dbCustomer == null) {
        customerRepository.save(customer);
    }

    Rental rental = new Rental();
    rental.setCar(car);
    rental.setCustomer(customer);
```

```
    rentalRepository.save(rental);
    return rental;
}
```

Bir aracın kiralanması için kullanılan rentACar() metodu dışarıdan bakıldığında sadece bir işlemi yapıyor olmakla birlikte, kendi bünyesinde bu işlemi üç değişik parçaya bölmektedir. Bu metot için bir transaksiyon oluşturulduğu taktirde bu parçalar bir bütünün parçası olarak görevlerini yerine getirirler. Herhangi bir parçanın görevini yerine getirememesi, daha önce yapılan işlemlerin geri alınmasını tetikleyecektir. Örneğin müşteri bilgileri customer tablosuna kaydetildikten sonra rentalRepository.save() bünyesinde bir hata oluşması durumunda, bir önceki işlemde customer tablosuna eklenen müşteri bilgileri silinir. Silme işleminin gerçekleşebilmesi için rollback komutu kullanılır. Her üç işlemin başarılı olması durumunda commit komutu ile verilerin tablolarda görünür hale gelmesi sağlanır.



Resim 6.1

RentalServiceImpl sınıfı servis katmanında yer alan tipik bir sınıftır. Bu tür sınıflar bünyelerinde rentACar() metodunda olduğu gibi birden fazla veri tabanı işlemi yapan metotlar barındırırlar. Bu tür metotların veri tabanı işlemeleri acısından atomik olmaları gerekmektedir, yani yapılan işlemlerin ya hep, ya hiç mantığı ile çalıştırılmaları gerekmektedir. Bu yüzden bu metotların bir transaksiyon içinde koşturulmaları zaruridir.

Lokal Transaksiyon Yönetimi

JDBC kullanılarak bir transaksiyon oluşturmak ve transaksiyonun akışını kontrol etmek mümkündür. Bunun bir örneği kod 6.2 de yer almaktadır.

```
Kod 6.2 - RentalServiceImpl

package com.kurumsaljava.spring;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Date;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class RentalServiceImpl extends JdbcDaoSupport
                implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }

    @Override
    public Rental rentACar(Customer customer, Car car,
                Date begin, Date end) {

        Connection con = getConnection();
        Rental rental = null;
        try {
            con.setAutoCommit(false);
            Customer dbCustomer =
            customerRepository.getCustomerByName(customer.getName());

            if (dbCustomer == null) {
                customerRepository.save(customer);
            }

            rental = new Rental();
            rental.setCar(car);
            rental.setCustomer(customer);
            rentalRepository.save(rental);
            con.commit();
            return rental;

        } catch (Exception e) {
            if (con != null) {
                try {
```

```
                    con.rollback();
                } catch (SQLException e1) {
                    throw new RuntimeException(e1);
                }
            }
            throw new RuntimeException(e);
        }
    }

    public void setCustomerRepository(
            CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(
            RentalRepository rentalRepository) {
        this.rentalRepository = rentalRepository;
    }
}
```

RentalServiceImpl sınıfı JdbcDaoSupport sınıfını genişlettiği için getConnection() metodunu kullanarak yeni bir Connection nesnesi edinebiliriz. JDBC ile bir transaksiyonu başlatabilmek için ilk önce setAutoCommit() metodunun false parametresi ile koşturulması gerekmektedir. Bu bizim commit() metodunu koşturmadığımız sürece transaksiyon bünyesinde JDBC sürücüsü tarafından otomatik olarak commit yapılmasını engeller. Metot bünyesindeki tüm işlemler başarıyla tamamlandıktan sonra con.commit() ile transaksiyon sonlandırılır. Eğer metot bünyesinde herhangi bir hata oluştu ise, kontrol catch bloğunda yer alan kod biriminin ilk satırına geçer. Bu gibi durumlarda yapılması gereken ilk işlem rollback komutunun koşturulmasıdır. Rollback komutu yapılan tüm veri tabanı işlemlerini tersine çevirir ve üzerinde işlem yapılan verilerin tutarlılıklarını devam ettirmelerini sağlar. Kod 6.2 de yer almamakla birlikte bir finally bloğu içinde con.close() ile veri tabanına olan bağlantı sonlandırılmalıdır.

Kod 6.2 de yer alan örnek tüm ACID özelliklerine sahiptir. Metot bünyesinde yer alan işlemler bir bütün olarak gerçekleştirilir ya da gerçekleştirilmez. Bunun yanı sıra gerekli tüm veriler bir bütün olarak veri tabanına kayıtlanır ya da kayıtlanmaz. Her rentACar() metoduna giriş yeni bir transaksiyon oluşturacağından, meydana gelen değişiklikler oluşturulan transaksiyonlar tarafından birbirlerinden izole edilir. En son adım olarak koşturulan commit komutu tüm değişikliklerin tablolara kaydedilmesini sağlar.

```
Kod 6.3 - applicationContext.xml

<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="schema.sql" />
    <jdbc:script location="data.sql" />
</jdbc:embedded-database>

<bean id="customerRepository"
        class="com.kurumsaljava.spring.
                JDBCCustomerRepositoryImpl">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<bean id="rentalRepository"
        class="com.kurumsaljava.spring.JDBCRentalRepositoryImpl">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository"></property>
    <property name="rentalRepository"
            ref="rentalRepository"></property>
    <property name="dataSource"
            ref="dataSource"/>
</bean>
```

Bu tür bir transaksiyon yönetimine lokal transaksiyon yönetimi ismi verilmektedir. Yazılımcı JDBC aracılığı ile transaksiyon kullanımı için gerekli tüm kodu kendisi geliştirir. Böylece transaksiyonun yönetimi tamamen programsal yapılmış olur. Bir transaksiyon oluşturabilmek için yapmamız gereken şey bir dataSource nesnesini edinmektir. Bu dataSource nesnesini kullanarak bir Connection nesnesine erişebilir ve transaksiyon yönetimi için gerekli adımları atabiliriz. Kullandığımız dataSource konfigürasyonu kod 6.3 de yer almaktadır. RentalServiceImpl sınıfı JdbcDaoSupport sınıfını genişlettiği için bu sınıfa bir dataSource nesnesini enjekte edebilmekte ve akabinde getConnection() üzerinden bir Connection nesnesi edinerek, yeni bir transaksiyon oluşturabilmekteyiz.

Ne yazık ki lokal transaksiyon yönetiminin beraberinde getirdiği bazı dezavantajlar vardır. Bunlar:

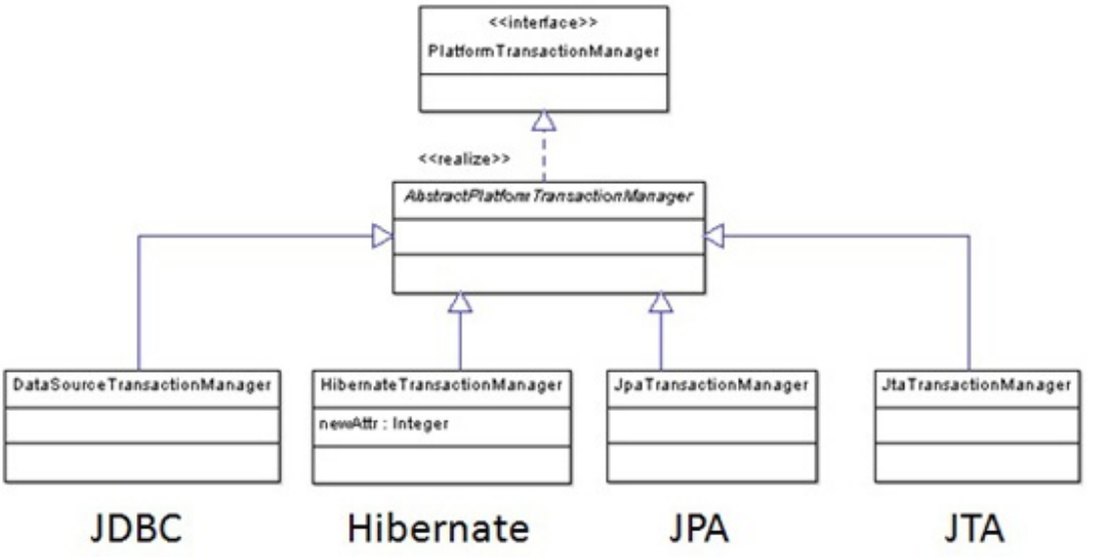
- Lokal transaksiyon yönetimi için ekstra kod yazılması gerekir. JDBC API'sinin kullanımı transaksiyon yönetimi için gerekli kodun yazılmasını

zorlaştırır. Bu tür bir kodun bakımı ve geliştirilmesi hatalara sebep verebilir.
- Transaksiyon yönetimine müdahale etmek için kod üzerinde değişiklik yapılması gerekir. Çok ufak bir değişiklik bile kodun yeniden derlenmesini mecburi kılar.
- Transaksiyon yönetimi ile iş mantığı iç içe geçer. Birbirlerinden ayırt edilmeleri zorlaşır.
- Transaksiyonlar servis katmanında yer alan metotlardan başlatılmak (transaction demarcation) zorundadır, çünkü bu metotlar genelde birden fazla veri tabanı işlemini tek bir işlem gibi gruplarlar. Bu durumda RentalServiceImpl örneğinde de gördüğümüz gibi servis katmanında yer alan bir sınıf JDBC API'si ile interaksiyona girmek zorunda kalır. Bu aslında RentalRepository ya da CustomerRepository gibi veri katmanında yer alan sınıfların görevidir. Böylece veri tabanı işlemleri için gerekli olan JDBC API soyutluk açışından daha yüksek bir seviyede olan bir katman tarafından kullanılır. Bu katmanlı mimariye aykırı bir durumdur.

Spring İle Transaksiyon Yönetimi

Spring transaksiyon template sınıfları ile programsal (programmatic), XML konfigürasyonu ve anotasyon yardımı ile deklaratif (declarative) transaksiyon yönetimi desteği sağlamaktadır. EJB teknolojisinden tanıdığımız deklaratif transaksiyon yönetimi işletme mantığının ve transaksiyon yönetimi için gerekli kodun birbirlerinden bağımsız bir şekilde oluşturularak, XML konfigürasyon üzerinden ya da anotasyonlar aracılığı ile bir araya getirilmeleri için kullanılır. Deklaratif transaksiyon yönetimi için Spring AOP (Aspect Oriented Programming) teknolojisini kullanır. AOP hakkında bilgiyi kitabın AOP başlıklı bölümünde bulabilirsiniz.

Spring transaksiyon yönetimi için bünyesinde barındırdığı transaksiyon yönetici (transaction manager) sınıflarını kullanır. Kullanılan her değişik veri erişim teknolojisi için o veri erişim teknolojisini destekleyen bir transaksiyon yönetici sınıfı mevcuttur. PlatformTransactionManager sınıfını implemente eden bu sınıflar ve mevcut hiyerarşileri resim 6.2 de yer almaktadır.



Resim 6.2

Uygulamanın doğrudan JDBC aracılığı ile veri tabanı işlemlerini gerçekleştirdiği durumlarda transaksiyonları yönetmek için DataSourceTransactionManager sınıfı kullanılır. DataSourceTransactionManager sınıfı Spring ve EJB'nin transaksiyon yönetimi için ayrı yöntemleri seçtiklerinin kanıtıdır. EJB'de doğrudan JTA API'si aracılığı ile bir transaksiyon yöneticisi ve implementasyonu kullanma zorunluluğu mevcutken, Spring'de bir JTA implementasyonunun kullanımını zorunlu değildir. Bunun yerine Spring pragmatik olarak mevcut veri tabanı erişim teknolojisinin sunduğu transaksiyon yönetimi mekanizmalarının, ihtiva ettiği transaksiyon yönetici sınıfları aracılığı ile kullanılmalarını mümkün kılar.

Deklaratif Transaksiyon Yönetimi

Transaksiyon yönetiminde tavsiye edilen yöntem deklaratif transaksiyon yönetim konfigürasyonudur. Deklaratif tarzda transaksiyon yönetimi konfigürasyonu için atılması gereken iki adım bulunmaktadır. Bunlar:

- Transaksiyon yöneticisinin tanımlanması.
- Bir transaksiyona dahil olması gereken metotların tanımlanması.

Spring hem anotasyon tabanlı hem de XML tabanlı deklaratif transaksiyon yönetimi konfigürasyonunu desteklemektedir.

Bir transaksiyon oluşturabilmek için öncelikle XML konfigürasyon dosyasında bir transaksiyon yöneticisi tanımlamamız gerekiyor. Kod 6.4 de böyle bir

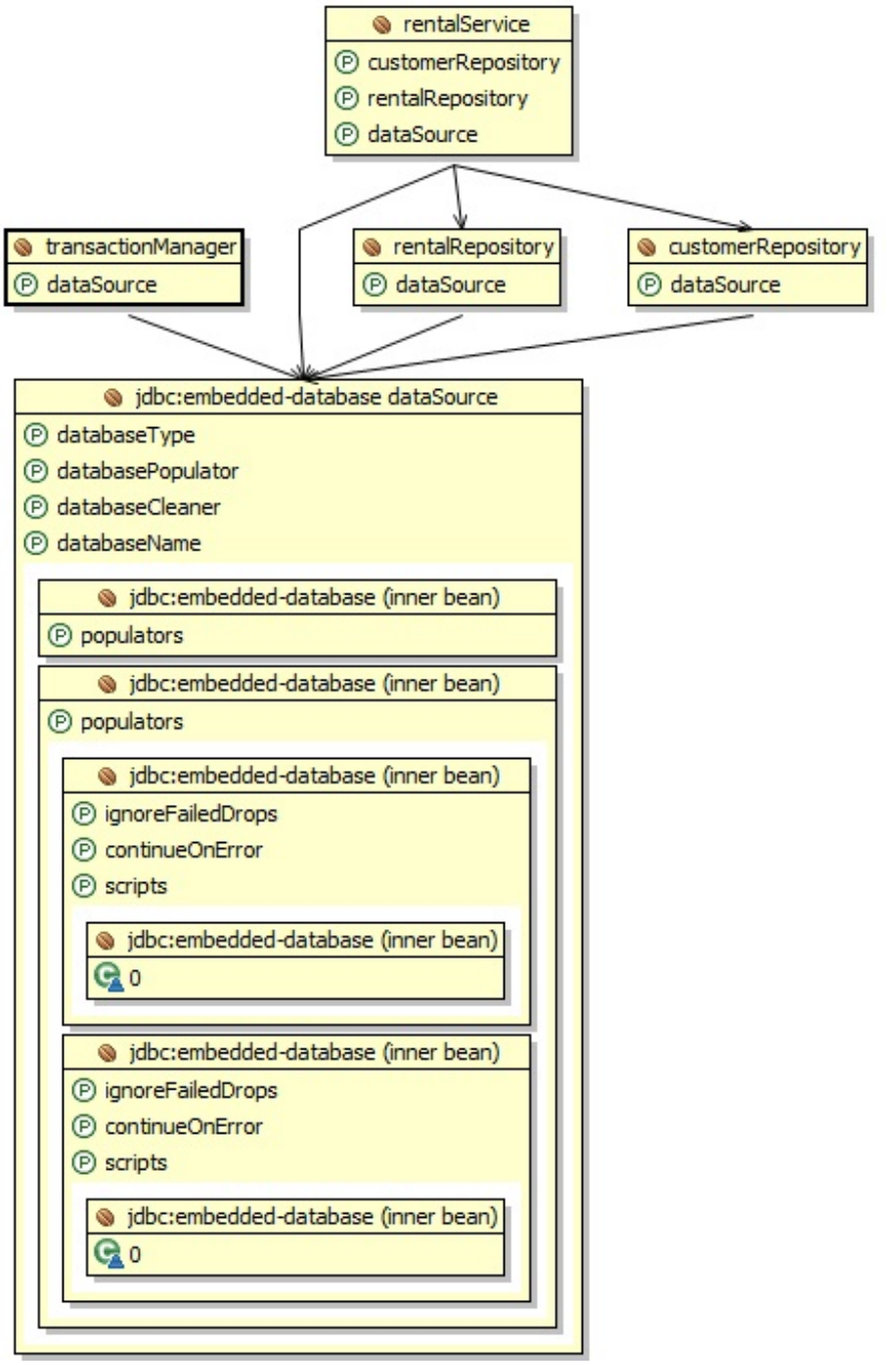
tanımlama yapılmıştır.

```
Kod 6.4 - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/
                    spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                    spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/
                    spring-tx-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                            spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan
            base-package="com.kurumsaljava.spring." />

    <tx:annotation-driven
            transaction-manager="transactionManager"/>

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.
                DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql" />
        <jdbc:script location="data.sql" />
    </jdbc:embedded-database>


    <bean id="customerRepository"
            class="com.kurumsaljava.spring.
                        JDBCCustomerRepositoryImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

```
    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                    JDBCRentalRepositoryImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
        <property name="customerRepository"
                ref="customerRepository"/>
        <property name="rentalRepository"
                ref="rentalRepository"/>
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

Araç kiralama uygulamamız JDBC yardımı ile veri tabanı işlemlerini gerçekleştirdiği için DataSourceTransactionManager sınıfını kullanarak transactionManager isminde bir transaksiyon yöneticisi tanımlıyoruz. Bu transaksiyon yöneticisine dataSource isimli nesne enjekte edilmektedir. Transaksiyona dahil olacak metotları @Transactional anotasyonu ile işaretleyebilmek için tx isim alanında yer alan annotation-driven elementini kullanmamız gerekmektedir. Transaction-manager element özelliği tanımlanan transaksiyon yöneticisinin (transactionManager) @Transactional ile işaretlenmiş metotlar bünyesinde transaksiyonu yönetmek için kullanılmasını sağlar. Spring tarafından kullanılan @Transactional ve diğer anotasyonları aktive edilebilmesi için konfigürasyon dosyasında component-scan elementini kullandık.

Konfigürasyon dosyasında oluşturduğumuz Spring tanımlamaları ve aralarındaki referanslar resim 6.3 de yer almaktadır.



Resim 6.3

Transaksiyon yöneticisini tanımladıktan sonra @Transactional anotasyonunu kullanarak transaksiyona dahil olacak metodu işaretlememiz gerekmektedir. Bu metot RentalServiceImpl sınıfında yer alan rentACar() metodudur. Bu sınıf kod 6.5 de yer almaktadır.

```
Kod 6.5  - RentalServiceImpl

package com.kurumsaljava.spring.annotation;
```

```
import java.util.Date;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.annotation.Transactional;
import com.kurumsaljava.spring.Car;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;
import com.kurumsaljava.spring.Rental;
import com.kurumsaljava.spring.RentalRepository;
import com.kurumsaljava.spring.RentalService;

public class RentalServiceImpl extends JdbcDaoSupport
                implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }

    @Transactional
    @Override
    public Rental rentACar(Customer customer, Car car,
                    Date begin, Date end) {
        Rental rental = null;
        Customer dbCustomer =
            customerRepository.getCustomerByName(
                    customer.getName());

        if (dbCustomer == null) {
            customerRepository.save(customer);
        }

        rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;

    }

    public void setCustomerRepository(CustomerRepository
                    customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(RentalRepository
            rentalRepository) {
        this.rentalRepository = rentalRepository;
```

```
    }
}
```

Kod 6.2 ile kod 6.5 de yer alan sınıfları kıyasladığımızda, Spring ile oluşturduğumuz deklaratif transaksiyon yönetim konfigürasyonunun kodu ne kadar sadeleştirdiğini görmekteyiz. Kod 6.2 de yer alan rentACar() metodu bünyesinde hem transaksiyon yönetimi için gerekli kodu, hem de iş mantığı için gerekli kodu barındırmaktadır. Kod 6.5 de yer alan rentACar() metodu sadece iş mantığına sahiptir.

```
Kod 6.6  - Main

package com.kurumsaljava.spring;
import org.springframework.context.
                    ConfigurableApplicationContext;
import org.springframework.context.support.
                    ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args)
                throws Exception {
        ConfigurableApplicationContext ctx =
                new ClassPathXmlApplicationContext(
                        "applicationContext.xml");
        RentalService rentalService =
                ctx.getBean(RentalService.class);

        Customer customer = new Customer("Sunal", "Kemal", 69);
        Rental rental =
                rentalService.rentACar(customer, new Car(),
                                    null, null);
        System.out.println(rental.isRented());
    }
}
```

Kod 6.6 da yer alan main() metodunu koşturduğumuzda aşağıda yer alan ekran çıktısı, Spring'in tanımladığımız transaksiyonun yönetimi için hangi adımları attığını ihtiva etmektedir.

```
DataSourceTransactionManager - Creating new transaction with name
            [RentalServiceImpl.rentACar]:
            PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
DataSourceTransactionManager - Acquired Connection
            [org.hsqldb.jdbc.jdbcConnection@14275d4]
            for JDBC transaction
```

```
DataSourceTransactionManager - Switching JDBC Connection
            [org.hsqldb.jdbc.jdbcConnection@14275d4] to manual
            commit
DataSourceTransactionManager - Initiating transaction commit
DataSourceTransactionManager - Committing JDBC transaction
            on Connection [org.hsqldb.jdbc.jdbcConnection@14275d4]
DataSourceTransactionManager - Releasing JDBC Connection
            [org.hsqldb.jdbc.jdbcConnection@14275d4]
            after transaction
```

Spring tarafından öncelikle RentalServiceImpl.rentACar isminde yeni bir transaksiyon oluşturulmaktadır. Akabinde transaksiyon bünyesinde kullanılmak üzere dataSource.getConnection() yardımı ile yeni bir Connection nesnesi oluşturulmakta ve iş mantığı koşturulmaktadır. İş mantığı son bulduktan sonra, eğer herhangi bir hata oluşmadı ise, yapılan değişiklikler commit ile veri tabanına kaydedilmektedir. Son adım olarak Spring sahip olduğu Connection nesnesini bırakmakta ve kaynak kullanımının dengede kalmasını sağlamaktadır.

DataSourceTransactionManager tarafından yönetilen bu transaksiyon örneğinde transaksiyon rentACar() metoduna girmeden başlatılmıştır. Transaksiyon bu metot bünyesinde kullanılan diğer metotları da kapsamaktadır. Resim 6.1 de yer aldığı gibi tanımladığımız transaksiyon RentalServiceImpl.rentACar(), JDBCCustomerRepositoryImpl.getCustomerByName(), JDBCCustomerRepositoryImpl.save(), ve JDBCRentalRepositoryImpl.save() metotlarını kapsar. rentACar() metodu return ile geri döndüğünde DataSourceTransactionManager tarafından commit komutu ile değişiklikler veri tabanına kaydedilir. Bir hata oluşması durumunda rollback komutu ile değişiklikler geriye alınır. Standart olarak oluşan RuntimeException tipi hatalar otomatik olarak rollback komutunun işletilmesini tetikler. Ama bu davranışı daha sonra göreceğimiz gibi değiştirmek mümkündür.

Sınıf Bazında Transaksiyon Yönetimi

Kod 6.5 de yer alan örnekte @Transactional anotasyonunu metot bazında bir transaksiyonu işaretlemek için kullandık. Spring tarafından oluşturulan transaksiyona sadece bu ve koşturduğu metotlar dahil edildi. Bu anotasyonu aynı zamanda sınıf bazında transaksiyon yönetimi için de kullanabiliriz. Bu anotasyon sınıf bazında kullanıldığı taktirde, sınıf bünyesinde yer alan tüm metotlar transaksiyon içinde koşturulur. Kod 6.7 de görüldüğü gibi public class

ile sınıf tanımlaması öncesi @Transactional kullanılarak bir sınıfın sahip olduğu tüm metotların transaksiyonel özelliğe kavuşması sağlanır.

```
Kod 6.7  - RentalServiceImpl

@Transactional
public class RentalServiceImpl extends JdbcDaoSupport
            implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }
    ...
}
```

Üstsınıflarda tanımlanan @Transactional anotasyonu altsınıf metotlarının da transaksiyona dahil edilmelerini sağlar.

Sınıf bazlı ve metot bazlı transaksiyon tanımlaması birlikte kullanılabilir. Kod 6.8 de yer alan RentalServiceImpl sınıfı için @Transactional kullanılarak hem sınıf bazında hem de metot bazında transaksiyon tanımlaması yapılmıştır. rentACar() metodunda kullanılan @Transactional anotasyonu ile tanımlanan transaksiyon özellikleri sınıf bazında kullanılan transaksiyon özelliklerini ezer. Bu şekilde genel olarak tanımlanmış transaksiyon özelliklerini metot bazında değiştirmek mümkündür.

```
Kod 6.8  - RentalServiceImpl

@Transactional(timeout=30)
public class RentalServiceImpl extends JdbcDaoSupport
                implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    public RentalServiceImpl() {
    }


    @Transactional(timeout=10)
    @Override
    public Rental rentACar(Customer customer, Car car,
```

```
                                Date begin, Date end) {
        Rental rental = null;
        ...
    }
}
```

Transaksiyon Özellikleri

Bir transaksiyonun uygulanış şeklini ve hangi kurallar çerçevesinde işletileceğini tanımlamak için transaksiyon özelliklerinin tanımlanması gerekmektedir. Bu özellikler:

- Transaksiyonların birbirlerinden izolasyonu (Isolation)
- Transaksiyonun yayılması (Propagation)
- Zaman aşımı (Timeout)
- Transaksiyonun sadece verileri okuyabilir ama değiştiremez özellikte olması (Read-Only)
- Yapılan değişikliklerin geri çevrilmesinde kullanılan kurallar (Rollback-Rules)

Deklaratif ya da programsal olarak tanımlanan transaksiyon yönetimi bu beş özelliğin üzerine inşa edilmiştir. Şimdi bu özellikleri yakından tanıyalım.

İzolasyon Seviyeleri (Isolation Levels)

Bir Java uygulamasında birden fazla thread aynı veriler üzerinde işlem yapacak şekilde paralel koşturulabilir. Bu threadler arası verileri edinme ve değiştirme konusunda bir rekabet ortamı oluşturur. Transaksiyon izolasyon seviyesi diğer threadlerin bir thread tarafından yapılan değişikliklerden ne oranda etkilenebileceğini tanımlar.

Çoklu thread kullanımı veri tabanında yer alan verilerin transaksiyonel olarak işlenmesi açısından şu sorunlara sebep verebilir:

- ***Kirli okuma (dirty reads)***: Bir threadin başka bir thread tarafından yapılmış ama henüz commit olmamış değişiklikleri görmesine kirli okuma ismi verilmektedir. Verinin kirliliği değişikliklerin henüz veri tabanına kaydedilmemesinden kaynaklanmaktadır. Commit olmayan bir değer, rollback olabilir anlamına gelmektedir. Rollback olması durumunda, ilk threadin okuyup, kullandığı değerler geçersiz hale gelir, çünkü veri tabanında artık o değerler yer almamaktadır.

- ***Tekrarlanamayan okuma (nonrepeatable reads)***: Bir threadin aynı sorguyu tekrarlaması neticesinde değişik sonuçlar elde etmesine tekrarlanamayan okuma ismi verilmektedir. Aynı sorgunun değişik sonuçlar vermesi, verinin başka bir thread tarafından iki sorgu arasında değişikliğe uğradığı anlamına gelmektedir.
- ***Fantom okuma (phantom reads)***: Bir threadin ikinci ve üçüncü sorgulamasında, ilk sorgulamasında bulunmayan değerleri elde etmesine fantom okuma ismi verilir. Bu genelde başka bir threadin okumalar arasında kullanılan veri tabanı tablosuna yeni değerler eklediği anlamına gelmektedir.

Bu ve bunun gibi verilerin tutarlılığını tehdit eden durumların önüne geçebilmek için bir izolasyon seviyesi tanımlanabilir. Bunlar:

- ***READ_UNCOMMITTED***: En alt izolasyon seviyesidir. Henüz commit edilmemiş verilerin okunmasını mümkün kılar. Bu izolasyon seviyesinin yan etkisi olarak kirli okuma, tekrarlanamayan okuma ve fantom okuma oluşabilir.
- ***READ_COMMITTED***: Sadece commit edilmiş verilerin okunmasını mümkün kılar. Bu şekilde kirli okumanın önüne geçilmiş olur, lakin tekrarlanamayan ve fantom okuma oluşabilir. Birçok veri tabanı sistemi tarafından kullanılan standart izolasyon seviyesidir.
- ***REPEATABLE_READ***: Bu izolasyon seviyesinde aynı verinin birden fazla okunması aynı neticeyi verir. Bu şekilde kirli ve tekrarlanamayan okuma engellenebilir. Fantom okumanın oluşma ihtimali mevcuttur.
- ***SERIALIZABLE***: Kirli, tekrarlanamayan ve fantom okumaların oluşmasını engeller. En yavaş çalışan transaksiyon türünün oluşmasına sebep olur, çünkü işlem yapan threadin üzerinde çalıştığı tabloların çalıştığı sürece kilitlenmesini sağlar. Bu diğer threadlerin kilit açılana kadar beklemeleri anlamına gelmektedir.

İzolasyon seviyesinin seçimi, uygulamanın performansını doğrudan etkileyebilir. Verilerin tutarlılığını korumak için seçilen izolasyon seviyesi ne kadar kuvvetli olursa, uygulamanın performansı o oranda düşecektir.

@Transactional anotasyonu kullanılarak izolasyon seviyesi şu şekilde tanımlanabilir:

```
@Transactional(isolation=Isolation.READ_UNCOMMITTED,
                timeout=30)
public class RentalServiceImpl
```

```
            extends JdbcDaoSupport implements RentalService {
...
}
```

Transaksiyon Yayılması (Transaction Propagation)

Bu transaksiyon özelliği transaksiyonun nerede başlayıp, nerede son bulduğunu, hangi metotların aynı ya da ayrı transaksiyonlar içinde yer aldıklarını tanımlamak için kullanılmaktadır.

Resim 6.1 deki dizge diyagramında RentalServiceImpl sınıfında yer alan rentaACar() metodu tarafından koşturulan diğer metotlar gösterilmektedir. Transaksiyon yayılma özelliğini kullanarak rentACar() ve koşturduğu diğer metotların topluca aynı transaksiyon içinde yer almaları sağlanabilir. Eğer istenirse her metodun kendi transaksiyonu başlatması sağlanabilir ya da diğer metotların transaksiyon ihtiyaç duymadığı belirtilebilir. Spring yedi değişik transaksiyon yayılım seviyesi tanımlayarak, transaksiyonun yayılış politikasının tayin edilmesini mümkün kılmaktadır. Bunlar:

- ***PROPAGATION_REQUIRED***: Metodun bir transaksiyon içinde koşturulması gerektiğine işaret eder. Eğer mevcut bir transaksiyon varsa, bu metot bu transaksiyona dahil edilir ve yeni bir transaksiyon oluşturulmaz. Mevcut bir transaksiyon yoksa yeni bir transaksiyon oluşturulur.
- ***PROPAGATION_REQUIRES_NEW***: Metodun yeni bir transaksiyon içinde koşturulması gerektigine işaret eder. Eğer mevcut bir transaksiyon varsa, bu transaksiyon durdurularak, metot için yeni bir transaksiyon oluşturulur.
- ***PROPAGATION_NEVER***: Metodun bir transaksiyon içinde koşturulmaması gerektiğine işaret eder. Eğer mevcut bir transaksiyon varsa bir hata (Exception) oluşturulur.
- ***PROPAGATION_NESTED***: Bir metodun iç, içe geçmiş bir transaksiyon (nested transaction) bünyesinde koşturulması gerektiğine işaret eder. İç transaksiyonlar en dışta yer alan transaksiyon tarafından commit ya da rollback edilebilir. Eğer dışsal bir transaksiyon mevcut değilse, davranış biçimi PROPAGATION_REQUIRED'daki gibi olacaktır.
- ***PROPAGATION_MANDATORY***: Metodun mutlaka bir transaksiyon içinde koşturulması gerektiğine işaret eder. Mevcut bir transaksiyon yoksa bir hata oluşturulur.
- ***PROPAGATION_SUPPORTS***: Metodun bir transaksiyona ihtiyaç duymadığı, lakin mevcut bir transaksiyona dahil olabileceğine işaret eder.

- *PROPAGATION_NOT_SUPPORTED*: Metodun transaksiyon mekanizmasını desteklemediğine işaret eder. Eğer mevcut bir transaksiyon varsa, bu transaksiyon metodun koşturulması esnasında durdurulur.

@Transactional anotasyonu kullanılarak transaksiyon yayılma politikası şu şekilde tanımlanabilir:

```
@Transactional(isolation=Isolation.READ_UNCOMMITTED,
            propagation=Propagation.MANDATORY)
public class RentalServiceImpl extends JdbcDaoSupport
                implements RentalService {
...
}
```

Zaman Aşımı (Timeout)

Bu transaksiyon özelliği bir transaksiyonun zaman aşımına uğramadan hangi zaman diliminde var olabileceğini gösterir.

@Transactional anotasyonu kullanılarak zaman aşımı süresi şu şekilde tanımlanabilir:

```
@Transactional(timeout=30)
public class RentalServiceImpl extends JdbcDaoSupport
            implements RentalService {
...
}
```

Transaksiyonun başlaması ile zaman aşımı için geriye sayım yapıldığı için bu transaksiyon özelliğinin PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW, ve PROPAGATION_NESTED gibi yeni bir transaksiyon başlatan transaksiyon yayılma özellikleri ile birlikte kullanılmaları mantıklı olacaktır.

Read-Only Transaksiyonlar

Read-only özelliğine sahip transaksiyonlar sadece veri okumak için oluşturulur. Bu tip transaksiyonlar bünyesinde veriler üzerinde değişiklik yapılmaz. Bu transaksiyon bünyesinde sadece SQL SELECT komutlarının kullanıldığı anlamına gelmektedir.

@Transactional anotasyonu kullanılarak read-only özelliği şu şekilde tanımlanabilir:

```
@Transactional(readOnly=true)
@Override
public Rental rentACar(Customer customer, Car car,
            Date begin, Date end) {
    Rental rental = null;
    Customer dbCustomer = customerRepository.
                getCustomerByName(customer.getName());
    ...
}
```

Burada "neden sadece SELECT ihtiva eden bir veri tabanı işlemi için bir transaksiyon oluşturulmalı?" sorusu aklınıza gelebilir. Read-only olarak tanımlanan bir transaksiyon Spring'in veri tabanı işlemleri için kullandığı kaynakları optimize etmesini sağlar. Örneğin RentalServiceImpl sınıfının rentACar() metodu eğer bir transaksiyon içinde koşturulmasaydı, getCustomerByName() ve save() metotları için iki ya da daha fazla Connection nesnesi kullanılırdı. Bu metodun bir transaksiyon içinde koşturulması, Spring tarafından sadece bir Connection nesnesinin oluşturulmasını ve diğer metotlar içinde de tekrar kullanılmasını sağlamıştır.

Bunun yanı sıra read-only transaksiyonlarda yüksek bir izolasyon seviyesinin kullanılması, okuma işlemi son bulana kadar verinin başka bir thread tarafından değiştirilmesi ve bu değişikliğin read-only transaksiyona yansıması engellenmiş olur.

Değişiklikleri Geri Alma Kuralları (Rollback-Rules)

Spring RuntimeException ya da bu sınıfı genişleten herhangi bir hata ile karşılaştığında otomatik olarak mevcut transaksiyonu rollback komutu ile sonlandırır. Rollback komutu yapılan tüm değişiklikleri geri alır ve böylece veri tabanı tablolarında herhangi bir değişiklik yapılmamış olur. Bu davranış biçimini konfigürasyon elementleri ile değiştirmek mümkündür. @Transactinal anotasyonunun rollbackFor özelliği ile istenilen türde bir hata sınıfı kullanılarak, bu hatanın oluşması durumunda rollback komutunun tetiklenmesi sağlanabilir. Kullanılan bu sınıf RuntimeException ya da Exception tipinde olabilir. noRollbackFor özelliği kullanıldığında hangi hataların rollback komutunu tetiklememesi gerektiği ifade edilmiş olur. rollbackFor ve noRollbackFor anotasyon özelliklerinin kullanımı aşağıdaki örneklerde yer almaktadır.

```
@Transactional(rollbackFor=MyDataAccessException.class)
@Override
```

```
public Rental rentACar(Customer customer, Car car,
            Date begin, Date end) {
    Rental rental = null;
    Customer dbCustomer =
        customerRepository.getCustomerByName(
                customer.getName());
    ...
}


@Transactional(noRollbackFor={MyException.class,
        MyRuntimeException.class})
@Override
public Rental rentACar(Customer customer, Car car,
            Date begin, Date end) {
    Rental rental = null;
    Customer dbCustomer =
        customerRepository.getCustomerByName(
                customer.getName());
    ...
}
```

XML İle Deklaratif Transaksiyon Yönetimi

Anotasyonların kullanımı her zaman mümkün olmayabilir. Örneğin anotasyonları kullanabilmek için JDK 5 ve üstü gereklidir. Bunun yanı sıra anotasyon kullanılmadan oluşturulan servislerin transaksiyona dahil edilebilmeleri gerekli olabilir. Bu gibi durumlarda transaksiyon yönetimi Spring XML dosyasında deklaratif olarak yapılabilir.

XML tabanlı deklaratif transaksiyon yönetimi Spring tarafından AOP teknolojisi kullanılarak yapılır. Bu amaçla bir veya birden fazla pointcut tanımlaması yapılması gereklidir. Bir pointcut bir veya birden fazla joinpoint'in seçilmesi için kullanılır. joinpoint ile program akışında belirli bir yer ifade edilir. Örneğin bir metodun koşturulması ya da bir değişkene bir değerin atanması AOP'de bir joinpoint'dir.

Kod 6.9 da yer alan örnekte rentalServiceMethods isminde bir pointcut tanımlıyoruz. aop:pointcut elementinin expression element özelliği aracılığı ile hangi sınıf ya da sınıfların joinpoint ihtiva ettiğini belirtiyoruz. aop:advisor elementi koşturmak istediğimiz koda işaret etmektedir. Bu kodu tx:advice elementini kullanarak tanımlıyoruz. Bu pointcut kullanarak tanımladığımız RentalServiceImpl sınıfının rentACar() metodudur. Spring bu metot

koşturulmadan önce bir transaksiyon oluşturarak, bu metodun bir transaksiyon içinde koşturulmasını sağlar. tx:advice elementi ayrıca transaction-manager element özelliği aracılığıyla kullanılmak istenilen transaksiyon yöneticisinin tanımlanmasını sağlar. Konfigürasyon dosyasında transactionManager bir Spring bean tanımlaması olduğu sürece transaction-manager element özelliğinin kullanımı mecburi değildir.

XML bazlı transaksiyon yönetimini konfigüre edebilmek için aop ve tx isim alanlarının konfigürasyon dosyasına yüklenmesi gerekmektedir.

AOP ile daha detaylı bilgiyi kitabın AOP bölümünde edinebilirsiniz.

```
Kod 6.9  - applicationContext-tx.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/
                                spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/
                                spring-aop-3.1.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                                spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/
                                spring-tx-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                                spring-context-3.0.xsd">


    <aop:config>
        <aop:pointcut
            expression="execution(*
                    com.kurumsaljava.spring.xml.
                            RentalServiceImpl+.*(..))"
            id="rentalServiceMethods" />
        <aop:advisor advice-ref="txAdvice"
                    pointcut-ref="rentalServiceMethods" />
    </aop:config>
```

```
    <tx:advice id="txAdvice"
            transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="rentACar" />
        </tx:attributes>
    </tx:advice>

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.
                        DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql" />
        <jdbc:script location="data.sql" />
    </jdbc:embedded-database>


    <bean id="customerRepository"
            class="com.kurumsaljava.spring.
                    JDBCCustomerRepositoryImpl">
        <property name="dataSource"
                    ref="dataSource"/>
    </bean>

    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                JDBCRentalRepositoryImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="rentalService"
                class="com.kurumsaljava.spring.xml.
                        RentalServiceImpl">
        <property name="customerRepository"
                ref="customerRepository"/>
        <property name="rentalRepository"
                ref="rentalRepository"/>
        <property name="dataSource"
                ref="dataSource"/>
    </bean>

</beans>
```

XML bazlı transaksiyon konfigürasyonu yapılması durumunda, izolasyon seviyesi *isolation* element özelliği ile şu şekilde tanımlanabilir:

```
<tx:advice id="txAdvice"
        transaction-manager="transactionManager" >
    <tx:attributes>
        <tx:method name="rentACar"
                    isolation="READ_COMMITTED"/>
    </tx:attributes>
</tx:advice>
```

Transaksiyon yayılma politikası propagation element özelliği ile şu şekilde tanımlanabilir:

```
<tx:advice id="txAdvice"
            transaction-manager="transactionManager" >
    <tx:attributes>
        <tx:method name="rentACar"
            isolation="READ_COMMITTED"
                    propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

Transaksiyon için zaman aşımı süresini timeout element özelliği ile şu şekilde tanımlayabiliriz:

```
<tx:advice id="txAdvice"
        transaction-manager="transactionManager" >
    <tx:attributes>
        <tx:method name="rentACar"
            isolation="READ_COMMITTED"
                    propagation="REQUIRED"
            timeout="30"/>
    </tx:attributes>
</tx:advice>
```

Transaksiyonun read-only özellikle olması read-only element özelliği kullanılarak şu şekilde tanımlanabilir:

```
<tx:advice id="txAdvice"
            transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="rentACar"
            read-only="true"/>
    </tx:attributes>
</tx:advice>
```

Hangi hatalar oluştuğunda transaksiyonun rollback komutu ile sonlandırılması

gerektiği rollback-for element özelliği ile şu şekilde tanımlanabilir:

```
<tx:advice id="txAdvice"
        transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="rentACar"
            rollback-for="MyDataException.class"/>
    </tx:attributes>
</tx:advice>
```

Hangi hataların transaksiyonu rollback ile sonlandırmaması gerektiği no-rollback-for element özelligi ile şu şekilde tanımlanabilir:

```
<tx:advice id="txAdvice"
        transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="rentACar"
            no-rollback-for="BusinessException.class,
                    JMXException.class"/>
    </tx:attributes>
</tx:advice>
```

Spring İle Programsal Transaksiyon Yönetimi

Deklaratif tarza transaksiyon yönetimi konfigürasyonunun sağladığı avantajları yakından inceledik. Bu tür bir konfigürasyon ile kodun okunabilirliği ve yapılan konfigürasyonunu esnekliği artmaktadır. Lakin programcının önünde transaksiyon yönetimini programsal tarzda yapma imkanı da mevcuttur.

Programsal transaksiyon yönetimi için Spring bünyesinde yer alan TransactionTemplate sınıfı kullanılır. Kod 6.10 da TransactionTemplate sınıfının kullanımını göstermektedir. Kodun transaksiyonel bir ortamda koşturulabilmesi için TransactionTemplate tipinde bir nesnenin RentalServiceImpl sınıfından olan bir nesneye enjekte edilmesi gerekmektedir. Bunun için gerekli olan konfigürasyon kod 6.11 de yer almaktadır.

```
Kod 6.10  - RentalServiceImpl

package com.kurumsaljava.spring.tx;

import java.util.Date;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.TransactionStatus;
```

```
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
import com.kurumsaljava.spring.Car;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;
import com.kurumsaljava.spring.Rental;
import com.kurumsaljava.spring.RentalRepository;
import com.kurumsaljava.spring.RentalService;

public class RentalServiceImpl extends JdbcDaoSupport
                    implements RentalService {

    private CustomerRepository customerRepository;

    private RentalRepository rentalRepository;

    private TransactionTemplate txTemplate;

    public RentalServiceImpl() {
    }

    @Override
    public Rental rentACar(final Customer customer, final Car car,
            final Date begin, final Date end) {

        return txTemplate.execute(new TransactionCallback() {

            @Override
            public Object doInTransaction(
                final TransactionStatus status) {
                Rental rental = null;
                try {
                    final Customer dbCustomer =
                        customerRepository.getCustomerByName(
                                        customer.getName());

                    if (dbCustomer == null) {
                        customerRepository.save(customer);
                    }
                    rental = new Rental();
                    rental.setCar(car);
                    rental.setCustomer(customer);
                    rentalRepository.save(rental);
                } catch (final Exception e) {
                    status.setRollbackOnly();
                }
                return rental;
            }
        });
```

```
    }

    public void setCustomerRepository(
            final CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(
            final RentalRepository rentalRepository) {
        this.rentalRepository = rentalRepository;
    }

    public void setTxTemplate(TransactionTemplate txTemplate) {
        this.txTemplate = txTemplate;
    }
}
```

TransactionTemplate sınıfı bir transaksiyon yöneticisine ihtiyaç duymaktadır. TransactionManager ismini taşıyan transaksiyon yöneticisi txTemplate ismini taşıyan TransactionTemplate nesnesine enjekte edilir. Akabinde txTemplate nesnesi rentalService nesnesine enjekte edilerek, rentACar() metodunun bir transaksiyon içinde koşturulmasını mümkün kılar.

```
Kod 6.11  - applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/
                        spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                        spring-beans.xsd">

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.
                        DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql" />
        <jdbc:script location="data.sql" />
    </jdbc:embedded-database>
```

```
    <bean id="customerRepository"
            class="com.kurumsaljava.spring.
                    JDBCCustomerRepositoryImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                    JDBCRentalRepositoryImpl">
        <property name="dataSource" ref="dataSource">
    </bean>

    <bean id="txTemplate"
        class="org.springframework.transaction.support.
                        TransactionTemplate">
        <property name="transactionManager"
            ref="transactionManager"/>
    </bean>

    <bean id="rentalService"
            class="com.kurumsaljava.spring.tx.
                    RentalServiceImpl">
        <property name="customerRepository"
                ref="customerRepository"/>
        <property name="rentalRepository"
                ref="rentalRepository"/>
        <property name="dataSource" ref="dataSource"/>
        <property name="txTemplate" ref="txTemplate"/>
    </bean>
</beans>
```

Eğer programsal transaksiyon yönetimi için gerekli nesneleri anotasyonlar aracılığı ile RentalServiceImpl sınıfından bir nesneye enjekte etmek isteseydik, RentalServiceImpl sınıfı kod 6.12 deki şekilde olurdu. İhtiyaç duyulan XML konfigürasyonu kod 6.13 de yer almaktadır.

```
Kod 6.12  - RentalServiceImpl

package com.kurumsaljava.spring.tx;

import java.util.Date;
import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.TransactionStatus;
```

```
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
import com.kurumsaljava.spring.Car;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.CustomerRepository;
import com.kurumsaljava.spring.Rental;
import com.kurumsaljava.spring.RentalRepository;
import com.kurumsaljava.spring.RentalService;

@Named
public class RentalServiceImpl extends JdbcDaoSupport
                    implements RentalService {

    @Inject
    private CustomerRepository customerRepository;

    @Inject
    private RentalRepository rentalRepository;

    @Inject
    public void injectDataSource(DataSource ds) {
        super.setDataSource(ds);
    }

    @Inject
    private TransactionTemplate txTemplate;

    public RentalServiceImpl() {
    }

    @Transactional
    @Override
    public Rental rentACar(final Customer customer, final Car car,
            final Date begin, final Date end) {

        return txTemplate.execute(new TransactionCallback() {

            @Override
            public Object doInTransaction(
                    final TransactionStatus status) {

                Rental rental = null;
                try {
                    final Customer dbCustomer =
                        customerRepository.
                            getCustomerByName(customer.getName());

                    if (dbCustomer == null) {
```

```
                        customerRepository.save(customer);
                    }
                    rental = new Rental();
                    rental.setCar(car);
                    rental.setCustomer(customer);
                    rentalRepository.save(rental);
                } catch (final Exception e) {
                    status.setRollbackOnly();
                }
                return rental;
            }
        });
    }
}
```

Kod 6.13 - applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/
                    spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                    spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/
                    spring-tx-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                            spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan
            base-package="com.kurumsaljava.spring." />

    <tx:annotation-driven
            transaction-manager="transactionManager" />

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.
                            DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
```

```
    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="schema.sql" />
        <jdbc:script location="data.sql" />
    </jdbc:embedded-database>

    <bean id="txTemplate"
        class="org.springframework.transaction.
                        support.TransactionTemplate">
        <property name="transactionManager"
                ref="transactionManager"/>
    </bean>
</beans>
```

Dağıtık (Distributed) Transaksiyonlar

Bir transaksiyon birden fazla veri kaynağını kapsayabilir. Örneğin uygulama iki değişik veri tabanını ya da bir veri tabanı ile bir JMS kuyruğunu (queue) kapsayan işlem(ler) yapabilir. Bu durumda lokal transaksiyonların kullanımı veri kaybına sebep olabilir, çünkü her veri kaynağı için bir lokal transaksiyon kullanılır ve bu lokal transaksiyonlar arasında koordinasyon yapılmaz. Birden fazla kaynak üzerinde veri güncelleme işlemini koordine etmek için dağıtık transaksiyonlar kullanılır.

Dağıtık transaksiyonu yönetmek için özel dağıtık transaksiyon yöneticileri (transaction manager) kullanılır. Bunlar tek bir transaksiyon bünyesinde ACID özelliklerini garanti edecek şekilde transaksiyonel kaynaklar üzerinde işlem yaparlar. Transaksiyon bünyesinde birden fazla transaksiyonel kaynak kullanıldığında tek adımda commit işlemini gerçekleştirmek mümkün değildir. Örneğin ilk kaynak commit işlemine olumlu cevap verdikten sonra, ikinci kaynak bünyesinde oluşan bir hata transaksiyonun ACID özelliğini olumsuz etkiler. Bu dağıtık transaksiyonlar ile yapılan commit işlemleri için birden fazla ve koordine edilmesi gereken adımların atılması anlamına gelmektedir. Bu koordinasyonu sağlamak amacıyla dağıtık transaksiyonlar bünyesinde Two Phase Commit (2PC - Iki fazli commit) protokolü kullanılır.

Dağıtık transaksiyonu yöneten transaksiyon yöneticisi 2PC nin ilk ayağında tüm kaynakların commit işlemi için hazır olup, olmadıklarını kontrol eder. Bu faza prepare (hazırlık) ismi verilmektedir. Eğer her kaynak commit işlemi için hazır ise, ikinci adımda tüm kaynaklar üzerinde commit işlemi gerçekleştirilir. Eğer kaynaklardan birisi prepare fazında olumsuz netice verirse, rollback komutu ile transaksiyon olumsuz olarak sonlandırılır.

Bir dağıtık transaksiyon bünyesinde kaynakların transaksiyon yöneticisi tarafından yönetilebilmeleri için belli bir arayüzün (interface) kullanılması gerekmektedir. Bu yöneten ve yönetilenlerin aynı dili konuşmalarını sağlar. X/Open gurubu tarafından spesifikasyonu yapılan bir arayüzün ismi XA'dır.

Dağıtık bir transaksiyon esnasında transaksiyon yöneticisi kaynaklar üzerinde şu işlemleri gerçekleştirebilir:

- ***start(xid)*** - transaksiyona dahil ol
- ***end(xid)*** - dağıtık transakyonu sonlandır
- ***prepare (xid)*** - kaynak commit için hazır mı?
- ***commit (xid)*** - asıl commit işlemini gerçekleştir

Xid dağıtık transaksiyonu ifade eden bir belirleyicidir. Tüm kaynaklar bünyesinden transaksiyon bu değer kullanılarak commit işlemi gerçekleştirilir.

Dağıtık transaksiyon oluşturabilmek için iki koşulan yerine gelmesi gerekmektedir. Bunlar:

- XA kabiliyetine sahip bir trasaksiyon yöneticisinin bulunması.
- XA kabiliyetine sahip transaksiyonel kaynakların kullanılması.

Java bünyesinde JTA (Java Transaction API) ismini taşıyan ve dağıtık transaksiyon oluşturmak için kullanılabilecek bir API mevcuttur. JTA XA olmadan da transaksiyon yönetimini sağlamak için kullanılabilecek bir API'dir. Lakin JTA olmadan XA kullanılamaz. J2EE / JEE tabanlı uygulama sunucularında JTA desteği ve implementasyonları bulunmaktadır. Bunun yanı sıra uygula sunucusu dışında kullanılmak üzere oluşturulmuş Atomikos, JTOM ve Arjuna gibi JTA implementasyonları mevcuttur.

Bir uygulama sunucusu bünyesinde

```
<tx:jta-transaction-manager/>
```

şeklinde bir tanımlama, uygulama sunucusu tarafından sunulan JTA bazlı transaksiyon yöneticisinin kullanımını mümkün kılmaktadır. Tx Spring bünyesinde yer alan bir isim alanıdır ve transaksiyon yönetimi için gerekli konfigürasyon elementlerinin kullanımını kolaylaştırır.

```
<jee:jndi-lookup id="dataSource"
        jndi-name="java:comp/env/jdbc/RentAcarDS"/>
```

```
<jee:jndi-lookup id="connectionFactory"
        jndi-name="java:comp/env/jms/RentAcarConnectionFactory"/>
```

Jee isim alanında yer alan jndi-lookup elementi ile uygulama sunucusu bünyesinde konfigüre edilmiş veri tabanı ve JMS (Java Messaging Service) gibi transaksiyonel kaynaklar edinmek ve bir dağıtık transaksiyon bünyesinde kullanmak mümkündür. Yukarıda yer alan konfigürasyon örneğinde uygulama sunucusu bünyesinde konfigüre edilmiş bir veri tabanı kaynağına (dataSource) ve bir JMS ConnectionFactory nesnesine erişim tanımlanmaktadır. Bu kaynakların sahip olduğu isimler kullanılarak uygulama sunucusunun hakimiyetinde olan transaksiyon yöneticisi aracılığı ile veri tabanı ve JMS sistemini kapsayan bir dağıtık transaksiyon oluşturulabilir. Uygulama sunucusu sahip olduğu JTA implementasyonu yardımı ile transaksiyon yönetimini otomatik olarak gerçekleştirir.

Kod bünyesinde transaksiyonu başlatmak için @Transactional anotasyonu kullanılır. Bu anostasyonun kullanımı hem lokal hem de dağıtık transaksiyonlar için aynıdır. Bu şekilde kod değişikliği yapmadan kullanılan transaksiyon tipi(lokal, dağıtık) değiştirilebilir.

Spring bünyesinde bir JTA implementasyonu bulunmamaktadır. Bu sebepten dolayı bir Spring uygulaması bünyesinde dağıtık transaksiyonlar oluşturulmak istendiğinde Atomikos gibi bir JTA implementasyonunun kullanılması gerekmektedir. Spring bünyesinde yer alan JtaTransactionManager sınıfı aracılığı ile JTA implementasyonu uygulamaya entegre edilir.

Kod 6.14 de yer alan örnekte Atomikos kullanılarak bir JTA transaksiyon yöneticisi oluşturulmaktadır.

```
Kod 6.14 - applicationContext.xml

<bean id="transactionManager"
    class="org.springframework.transaction.jta.
                JtaTransactionManager">
    <property name="transactionManager">
        <bean class="com.atomikos.icatch.jta.
                UserTransactionManager"/>
    </property>
    <property name="userTransaction">
        <bean class="com.atomikos.icatch.jta.
                UserTransactionImp"/>
        </bean>
    </property>
```

```
</bean>
```

JtaTransactionManager sınıfının transactionManager ve userTransaction ismini taşıyan iki değişkeni mevcuttur. Kod 6.14 de yer alan örnekte transactionManager değişkenine com.atomikos.icatch.jta.UserTransactionManager sınıfından olan bir nesne enjekte edilmektedir. Atomikos bünyesinde yer alan bu sınıf dağıtık transaksiyonu yönetecek olan asıl transaksiyon yöneticisidir. Bu sınıf javax.transaction.TransactionManager interface sınıfını implemente ederek bir JTA implementasyonu haline gelmektedir.

UserTransaction ismini taşıyan değişkene com.atomikos.icatch.jta.UserTransactionImp sınıfından bir nesne enjekte edilmektedir. JTA API'sinde yer alan javax.transaction.UserTransaction interface sınıfı implemente eden bu sınıf transaksiyon yöneticisi tarafından transaksiyonu başlatmak (begin), geri almak (rollback) ve sonlandırmak (commit) için kullanılır.

```
Kod 6.15 applicationContext.xml

<bean id="dataSource"
        class="org.apache.derby.jdbc.EmbeddedDataSource40">
    <property name="databaseName" value="rentacar" />
    <property name="createDatabase" value="create" />
</bean>
```

Lokal bir transaksiyon oluşturmak için kod 6.15 de yer alan dataSource tanımlaması yeterlidir. Bu örnekte bir Derby veri tabanı kullanılmaktadır. Bu dataSource tanımlamasının dağıtık bir transaksiyonda kullanılabilmesi için JTA transaksiyon yöneticisi tarafından gönderilen 2PC komutlarına cevap verebilmesi gerekmektedir. Bunu sağlamak için dataSource tanımlamasının kod 6.16 da görüldüğü gibi adapte edilmesi gerekmektedir.

```
Kod 6.16 - applicationContext.xml

<bean id="dataSource"
        class="com.atomikos.jdbc.AtomikosDataSourceBean"
      init-method="init" destroy-method="close">
    <property name="uniqueResourceName" value="XADBMS"/>
    <property name="xaDataSource">
        <bean
            class="org.apache.derby.jdbc.EmbeddedXADataSource40">
            <property name="databaseName" value="rentacar" />
```

```
            <property name="createDatabase" value="create" />
        </bean>
    </property>
</bean>
```

Kod 6.16 da yer alan örnekte AtomikosDataSourceBean sınıfı aracılığı ile kullanılan veri tabanı sistemi XA arayüzüne adapte edilmektedir. Bu şekilde JTA transaksiyon yöneticisi transaksiyona dahil olan bu veri kaynağını yönetebilir hale gelmektedir. Derby veri tabanının XA komutlarını anlayabilmesi için kullanılan JDBC sürücüsünün EmbeddedXADataSource40 tipinde olması gerekmektedir.

Dağıtık bir transaksiyon en az iki veri kaynağı kullanıldığı zaman gereklidir. Böyle bir senaryoyu oluşturmak için uygulamamızın kod 6.17 de yer alan JMS connectionFactory nesnesini kullanarak veri tabanı işlemleri ardından bir JMS mesajı gönderdiğini düşünelim. Dağıtık transaksiyon oluşturmaktaki maksadımız veri tabanı ya da JMS bünyesinde oluşan bir hata durumunda transaksiyonu rollback komutu ile sonlandırmaktır. Böylece veri kaybı önlenmiş olur.

Lokal transaksiyon kullanılması durumunda veri tabanı ve JMS işlemleri birbirlerinden bağımsız olarak yönetilir. Örneğin veri tabanı işlemi commit edildikten sonra JMS mesajı gönderimi esnasında oluşan bir hata veri tabanı için yapılan commit işlemini geri almaz. Böylece başarıyla tamamlanmış bir veri tabanı işlemi ardından JMS mesajı gönderememe ihtimali doğabilir.

```
Kod 6.17 - applicationContext.xml

<bean id="connectionFactory"
     class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
        value="vm://embedded?broker.persistent=false"/>
</bean>
```

JMS kullanmak için oluşturduğumuz connectionFactory nesnesini (kod 6.17) dağıtık bir transaksiyonda yer alabilecek şekilde değiştirmemiz gerekmektedir. Böyle bir konfigürasyon kod 6.18 de yer almaktadır.

```
Kod 6.18 - applicationContext.xml

<bean id="connectionFactory"
     class="com.atomikos.jms.AtomikosConnectionFactoryBean">
    <property name="uniqueResourceName" value="QUEUE_BROKER"/>
```

```
    <property name="xaConnectionFactory">
        <bean
            class="org.apache.activemq.
                    ActiveMQXAConnectionFactory">
            <property name="brokerURL"
             value="vm:broker:(tcp://localhost:61616)?
                        persistent=false"/>
            <property name="redeliveryPolicy">
                <bean class="org.apache.activemq.RedeliveryPolicy">
                    <property name="maximumRedeliveries" value="3"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

Kod 6.18 de yer alan örnekte AtomikosConnectionFactoryBean sınıfı aracılığı ile kullanılan JMS sunucusu (provider) XA arayüzüne adapte edilmektedir. Bu şekilde JTA transaksiyon yöneticisi transaksiyona dahil olan bu JMS sunucusunu yönetebilir hale gelmektedir. JMS sunucusunun XA komutlarını anlayabilmesi için kullanılan ConnectionFactory sürücüsünün ActiveMQXAConnectionFactory tipinde olması gerekmektedir.

```
Kod 6.19 - RentalServiceImpl

@Transactional
public Object doInDistributedTransaction(
            final TransactionStatus status) {
    Rental rental = null;
    try {
        final Customer dbCustomer =
            customerRepository.getCustomerByName(
                    customer.getName());

        if (dbCustomer == null) {
            customerRepository.save(customer);
        }
        rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        Confirmation confirmation = createConfirmation(rental);
        jmsTemplate.convertAndSend(confirmation);
    } catch (final Exception e) {
        status.setRollbackOnly();
    }
    return rental;
```

```
}
```

Kod 6.19 da yer alan doInDistributedTransaction() metodu dağıtık transaksiyonun işleyişini göstermektedir. Bu metot bünyesinde öncelikle customerRepository ve rentalRepository nesneleri yardımı ie veri tabanı işlemleri yapılmakta, akabinde bu nesneye enjekte edilmiş jmsTemplate yardımı ile bir JMS mesajı gönderilmektedir. Her iki veri kaynağı da (veri tabanı, JMS sunucusu) JTA transaksiyon yöneticisi kontrolünde olduğu için dağıtık transaksiyon işlemektedir. Eğer lokal transaksiyon kullanılmış olsa idi rentalRepository.save(rental) işlemi ayrı, jmsTemplate.convertAndSend(confirmation) ayrı bir transaksiyon içinde tamamlanırdı. Tanımlanan dağıtık transaksiyon ile bu iki transaksiyon bir bütün haline getirilmektedir.

Kod 6.19 da yer alan doInDistributedTransaction() metodu Spring çatısının kod değişikliği yapmadan sadece konfigürasyon değişikliği ile uygulamaya yeni bir davranış biçimi verme kabiliyetinin iyi bir örneğini teşkil etmektedir. Lokal transaksiyon yerine dağıtık bir transaksiyon kullanmak için konfigürasyon dosyasını kod 6.16 ve 6.18 deki gibi değiştirmemiz yeterli olacaktır. Bu değişikliklerin ardından kod 6.19 da yer alan doInDistributedTransaction() metodu dağıtık bir transaksiyon bünyesinde görevini yerine getirecektir. Kod 6.15 ve 6.17 de yer alan konfigürasyon örneklerini kullandığımız taktirde her veri kaynağı için aynı bir lokal transaksiyon oluşturularak, veri kaynakları üzerinde yapılan işlemler birbirlerinden bağımsız olarak yapılacaktır. Bu belli şartlar altında veri kaybı anlamına gelebilmektedir. Bu sebepten dolayı birden fazla veri kaynağının kullanımı durumunda dağıtık transaksiyonlarin kullanımı gerekmektedir.

6. Bölüm Soruları

6.1 Veri tabanı işlemlerinde verilerin bütünlüğünü korumak için hangi mekanizma kullanılır?

6.2 Commit yapılmadığı taktirde veriler üzerinde yapılan işlemin neticesi ne olur?

6.3 Spring ile hangi türde transaksiyon yönetimi yapılabilir?

6.4 Spring deklaratif transaksiyon yönetimi için hangi yöntemleri kullanmaktadır?

6.5 Programsal transaksiyon yönetimi için kullanılan Spring sınıfı hangisidir?

6.6 Birden fazla kaynak üzerinde veri güncelleme işlemini koordine etmek için hangi transaksiyon türü kullanılır?

6.7 JTA bazlı transaksiyon yönetici hangi konfigürasyon elementi ile aktif hale getirilir?

6.8 Spring bünyesinde bir JTA implementasyonu mevcut mudur?

7. Bölüm

Spring İle Hibernate Kullanımı

Boyut Farkı

Relasyonel (RDBMS - Relational Database Management System) veri tabanlarında veriler iki boyutlu tablolarda tutulur. Her tablonun belirlenmiş adette kolonu vardır. Tablonun her satırı bu kolonların değerlerini ihtiva eder. Her satır bir öğeyi (entity) temsil eder. Bu örneğin müşteri, sipariş, ya da bir adres bilgisi olabilir. Buna göre veri tabanı tabloları öğe bazında yapılandırılır. Her satırın tekil (unique) olabilmesi için değeri yapılan her kayıtta bir artırılan bir anahtar kolon (primary key) kullanılır. Birden fazla kolon bir araya getirilerek, tabloda yer alan her kaydın tekil olması sağlanabilir. Bu anahtar kolonlar kullanılarak tablolar arasında ilişkiler oluşturulur. Bir anahtar kolonun başka bir tabloda ilişkiye işaret eden değer olarak kullanılmasına yabancı anahtar (foreign key) ismi verilir.

Relasyonel veri modelleri iş sahasında oluşan veriler ile doğrudan ilişkilidir. Bu modellerde verilerin hızlı bir şekilde ve az yer tutacak şekilde işlenmesi önceliklidir. Bunun yanı sıra çoğu veri tabanı şemalarının (database schema) mevcudiyetleri uygulamaların geliştirilmelerinin öncesine dayanır. Bu durum uygulamanın veri tabanı şemasına uyacak şekilde geliştirilmesini zorunlu kılar. Bir veri tabanı şemasını birden fazla uygulama paylaşabilir.

Java dünyasında veri taşıyıcıları nesnelerdir. Şablon vazifesi gören sınıflardan oluşturulan bu nesneler taşıdıkları veriler yanı sıra, bir kimlik ve değişik davranış (metot) biçimlerine sahiptirler. Canlı varlıklarla kıyaslayabileceğimiz nesneler bu açıdan bakıldığında üç boyutlu olarak düşünülebilir. Sınıflar uygulamaların ihtiyaçlarına göre yapılandırılırlar. Nesneler bünyelerinde taşıdıkları verileri (data hiding; data encapsulation) dış dünyadan gizlerler. En azından bunu yapma yeteneğine sahiptirler. Dış dünyaya sundukları metotlar aracılığı ile sahip oldukları verilerin kullanılmasına ve değiştirilmesine izin verirler.

Nesne/Relasyonel Eşleme

Uygulama geliştiricilerinin en çok sıkıntı çektikleri konuların başında üç boyutlu nesnelerin, iki boyutlu relasyonel veri tabanı sistemlerinde tutulması gelmektedir. Bu iki dünya arasındaki farklılığı gizlemek ve yazılımcının SQL üzeri bir soyutluk seviyesinde çalışabilmesini sağlamak için nesne/relasyonel eşleme (ORM - Object/Relational Mapping) araçları geliştirilmiştir. Java

dünyasının en popüler ORM aracı Hibernate'dir. Spring Hibernate, EclipseLink ve OpenJPA gibi ORM araçlarının kullanımını desteklemektedir. Kitabın bu bölümünde Hibernate'i daha yakından inceleyeceğiz.

Hibernate İle Eşleme (Mapping)

Ne kadar ORM araçları yazılımcının hayatını kolaylaştırsalar da, kahin olmadıkları için bir nesnenin ihtiva ettiği verilerin bir veri tabanı tablosunun kolonlarıyla nasıl eşlenmeleri gerektiğini bilemezler. Yazılımcının bu eşlemeyi ORM aracının anlayacağı dilde yapması gerekmektedir. Bu işlem için kullanılan bilgilere meta bilgisi (metadata) ismi verilmektedir. Meta bilgisi veri hakkında veri anlamına gelmektedir ve verilerin nasıl kaydedildiğini ve edinildiğini tanımlar.

Hibernate kullanıldığı taktirde meta bilgi tanımlaması anotasyonlar ya da XML eşleme dosyaları aracılığı ile yapılabilir.

Anotasyon Yardımı İle Eşleme (Annotation Mapping)

Hibernate 3.5 sürümü ile tüm JPA 2 anotasyonlarını desteklemektedir. Buna göre javax.persistence paketinde olan tüm anotasyonlar kullanılabilir. Bunun yanı sıra Hibernate'in JPA harici kendi anotasyon seti vardır. Hibernate'e özel olan bu anotasyonlar JPA harici ortamlarda Hibernate'in kullanımını sağlar. Bu anotasyonlar Hibernate kullanan uygulamaların daha performanslı olmalarını sağlar. Lakin Hibernate'in JPA 2 anotasyonlarını desteklemeyi başlaması ile bu anotasyonlar önemlerini yitirmişlerdir. Yeni uygulamalarda JPA 2 anotasyonlarının kullanımı Hibernate gibi bir ORM aracın uygulamayı etkilemeden değiştirilmesini mümkün kılar.

Kod 7.1 de yer alan Customer sınıfı JPA anotasyonları kullanılarak yapılandırılmıştır. @Entity anotasyonu bir sınıfın veri tabanında kaydedilebilir bir öge (entity) olduğuna işaret eder. @Table anotasyonu ile verilerin kaydedileceği tablonun ismi belirlenir. Kod 7.1 de yer alan Customer nesneleri kullanılan @Table anotasyonu ile customer isimli bir tabloya kaydedilir. Hibernate gerekli tabloları oluştururken bu anotasyondan faydalanır. @Table anotasyonu kullanılmadığı taktirde kullanılan ya da oluşturulan tablonun ismi sınıfın ismidir.

```
Kod 7.1 - Customer

package com.kurumsaljava.spring;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "customer")
public class Customer {

    @Id
    @GeneratedValue
    @Column(name = "id")
    private long id;

    @Column(name="name")
    private String name;

    @Column(name="firstname")
    private String firstname;

    @Column(name="age")
    private int age;

    public Customer() {
    }
}
```

Customer sınıfının id isminde bir sınıf değişkeni mevcuttur. Bu değişken @Id anotasyonu kullanılarak daha önce bahsettiğimiz anahtar kolon haline getirilmiştir. Anahtar kolonların değerleri tekildir. @GeneratedValue anotasyonu kullanılarak bu değerin her yeni kayıtta otomatik olarak bir artırılması sağlanmıştır. Buna göre yapılan ilk müşteri kaydında id değişkenin değeri 1, sonraki kayıtlarda hep n+1 olacaktır. Daha sonraki kod örneklerinde göreceğimiz gibi herhangi bir müşteri kaydını id değişkeninin sahip olduğu değere göre aramamız mümkün olacaktır.

Customer sınıfının sahip olduğu id ve name gibi değişkenlerin veri tabanında yer alacak customer tablosunun kolon başlıkları olabilmesi için @Column anotasyonu kullanılmıştır. Bu anotasyonun name özelliği aracılığı ile kolon ismi belirlenir. Bu konfigürasyondaki bir sınıf için Hibernate kod 7.2 ye yer alan

create table SQL komutunu kullanarak customer isminde bir tablo oluşturacaktır. Bu tablonun kolon isimleri @Column anotasyonu aracılığı ile değiştirebilmektedir.

```
Kod 7.2 - customer create SQL

DEBUG: org.hibernate.tool.hbm2ddl.SchemaExport -
    create table customer (
        id bigint generated by default as identity (start with 1),
        age integer,
        firstname varchar(255),
        name varchar(255),
        primary key (id)
    )
```

SessionFactory Konfigürasyonu

Hibernate ile veri tabanı işlemleri yapabilmek için bir org.hibernate.Session nesnesine ihtiyaç duyulmaktadır. Bu nesne yardımı ile nesneler aranır (find), kaydedilir (save) ve silinir (delete). Bu Session nesnesini elde edebilmek için bir org.hibernate.SessionFactory nesnesinin oluşturulması gerekir. SessionFactory tekil bir veri kaynağını temsil eder ve threadler arasında paylaşılabilir (thread-safe) yapıdadır.

Spring XML konfigürasyon dosyasında AbstractSessionFactoryBean isimli sınıfın altsınıfları olan LocalSessionFactoryBean ve AnnotationSessionFactoryBean implementasyonları kullanılarak bir SessionFactory tanımlanabilir. LocalSessionFactoryBean implementasyonu XML bazlı Hibernate eşleme (mapping) dosyaları oluşturulduğu zaman kullanılır. Bu implementasyonun nasıl kullanıldığını daha sonra inceleyeceğiz. Anotasyon bazlı Hibernate eşlemeyi seçtiğimiz için (kod 7.1 e bakınız), bir SessionFactory tanımlamak için AnnotationSessionFactoryBean implementasyonundan faydalanacağız. AnnotationSessionFactoryBean kullanarak oluşturduğumuz SessionFactory tanımlaması kod 7.3 de yer almaktadır.

```
Kod 7.3 - SessionFactory

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.
                AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
```

```
    <property name="annotatedClasses">
        <list>
            <value>com.kurumsaljava.spring.Rental</value>
            <value>com.kurumsaljava.spring.Car</value>
            <value>com.kurumsaljava.spring.Customer</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.format_sql=true
            hibernate.show_sql=true
            hibernate.hbm2ddl.auto=create
        </value>
    </property>
</bean>
```

Bir SessionFactory tanımlaması yapabilmek için bir dataSource tanımlamasına ihtiyaç duyulmaktadır. Böyle bir dataSource tanımlamasını daha önceki örneklerimizde kullanmıştık. Bunun yanı sıra AnnotationSessionFactoryBean implementasyonu JPA anotasyonlarını taşıyan sınıfların bir listesine ihtiyaç duymaktadır. Bu listeyi annotatedClasses aracılığı ile tanımlıyoruz. Kod 7.2 de yer alan örnekte Rental, Car ve Customer JPA entity sınıfları olarak tanımlanmıştır.

HibernateProperties özelliği kullanılarak Hibernate'in davranış biçimleri yönlendirilebilir. Örneğin hibernate.hbm2ddl.auto=create parametresinin kullanımı Hibernate'in gerekli veri tabanı tabloları oluşturmasını ve bu tabloları veri tabanına kaydetmesini (schema export) sağlar. Hibernate bu işlemi bir SessionFactory nesnesinin oluşturulması ile başlatır. Create-drop kullanıldığında SessionFactory sonlandırıldığında oluşturulan veri tabanı şeması silinir (drop). Hibernate.show_sql=true kullanıldığında Hibernate kullandığı tüm SQL komutlarını log dosyalarına kaydeder ya da ekranda görüntüler. Kod 7.2 böyle oluşturulmuş bir kayıttır.

AnnotatedClasses yardımı ile JPA anotasyonlarını taşıyan sınıfların bir listesini oluşturduk. Bu liste kısa zamanda çok uzun bir hale gelebilir. Böyle bir liste oluşturmak yerine packagesToScan yardımı ile JPA anotasyonlarını taşıyan sınıfların otomatik olarak taranması sağlanabilir. Bunun bir örneği kod 7.4 de yer almaktadır.

```
Kod 7.4 - applicationContext

<bean id="sessionFactory"
```

```
    class="org.springframework.orm.hibernate3.annotation.
            AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan">
        <list>
            <value>com.kurumsaljava.spring</value>
        </list>
    </property>
</bean>
```

Araç kiralama servisi uygulamasını Hibernate ile çalışır hale getirmek için gerekli tüm XML konfigürasyonu kod 7.5 de yer almaktadır.

```
Kod 7.5 - applicationContext.xml

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/
                spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
                spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/
                spring-tx-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                spring-context-3.1.xsd">

    <context:annotation-config />

    <tx:annotation-driven
        transaction-manager="transactionManager"/>

    <jdbc:embedded-database id="dataSource" type="HSQL"  />

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.annotation.
                                AnnotationSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
            <list>
                <value>com.kurumsaljava.spring.Rental</value>
                <value>com.kurumsaljava.spring.Car</value>
```

```
                <value>com.kurumsaljava.spring.Customer</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.format_sql=true
                hibernate.show_sql=true
                hibernate.hbm2ddl.auto=create
                hibernate.hbm2ddl.show=true
            </value>
        </property>
    </bean>

    <bean id="customerRepository"
        class="com.kurumsaljava.spring.
                HibernateCustomerRepositoryImpl">
        <property name="sessionFactory"
                        ref="sessionFactory"/>
    </bean>

    <bean id="rentalRepository"
        class="com.kurumsaljava.spring.
                HibernateRentalRepositoryImpl">
        <property name="sessionFactory"
                ref="sessionFactory"/>
    </bean>

    <bean id="carRepository"
        class="com.kurumsaljava.spring.
            HibernateCarRepositoryImpl">
        <property name="sessionFactory"
                ref="sessionFactory"/>
    </bean>

    <bean id="rentalService"
            class="com.kurumsaljava.spring.
                RentalServiceImpl">
        <property name="customerRepository"
                ref="customerRepository" />
        <property name="rentalRepository"
                ref="rentalRepository"/>
        <property name="carRepository"
                ref="carRepository"/>
    </bean>

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.
            HibernateTransactionManager">
        <property name="sessionFactory"
```

```
            ref="sessionFactory"/>
    </bean>
</beans>
```

Uygulamamız bünyesinde Hibernate'i kullanabilmek için gerekli konfigürasyonu oluşturduk. Kod 7.6 da Hibernate'in bir sınıf bünyesinde veri tabanı işlemleri için nasıl kullanılabileceği yer almaktadır.

```
Kod 7.6 - HibernateCustomerRepositoryImpl

package com.kurumsaljava.spring;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class HibernateCustomerRepositoryImpl
            implements CustomerRepository {

    private SessionFactory sessionFactory;

    @Override
    public Customer getCustomerByName(String name) {
        Query query = getCurrentSession().createQuery(
                "from Customer c where c.name=:name");
        query.setString("name", name);
        return (Customer) query.uniqueResult();

    }

    @Override
    public void save(Customer customer) {
        getCurrentSession().save(customer);
    }

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public void setSessionFactory(
            SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    private Session getCurrentSession() {
        return sessionFactory.getCurrentSession();
    }
}
```

Herhangi bir Hibernate işlemi yapmadan önce bir Session nesnesi oluşturmamız gerekmektedir. HibernateCustomerRepositoryImpl sınıfında yer alan getCurrentSession() metodu kod 7.5 de sınıfa enjekte edilen sessionFactory nesnesini kullanarak bir Session nesnesi oluşturmaktadır. getCustomerByName() metodu bünyesinde bu Session nesnesi kullanılarak bir Hibernate Query nesnesi oluşturulmaktadır. Query nesnesi belli bir soyisme sahip bir müşteriyi aramak için kullanılmaktadır. Sorgulamayı yapabilmek için SQL yerine HQL (Hibernate Query Language) kullanılmıştır. HQL Hibernate'e has bir veri sorgulama dilidir. HQL bünyesinde veri tabanı tablo isimleri yerine JPA anotasyonlarını taşıyan entity sınıflarının isimleri kullanılır. From Customer c where c.name= Customer sınıfında yer alan name değişkenin değeri anlamına gelmektedir. Hibernate kendi bünyesinde HQL'i bir SQL komutuna dönüştürür. GetCustomerByName() metodu işlem gördüğünde Hibernate tarafından oluşturulan SQL komutu kod 7.7 de yer almaktadır.

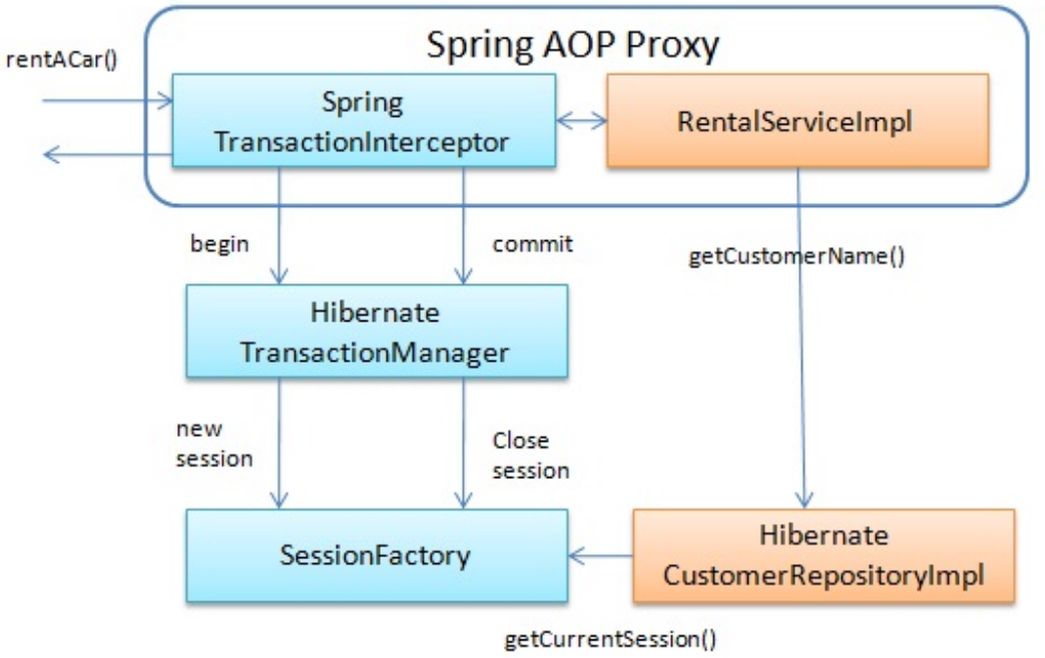
```
Kod 7.7 - Hibernate SQL Çıktısı

Hibernate:
    select
        customer0_.id as id2_,
        customer0_.age as age2_,
        customer0_.firstname as firstname2_,
        customer0_.name as name2_
    from
        customer customer0_
    where
        customer0_.name=?
```

HibernateCustomerRepositoryImpl bünyesinde yer alan save() metodu bir müşteri nesnesini veri tabanına kaydetmek için kullanılmaktadır. Bu amaçla Session nesnesinin save() metodu kullanılmaktadır. Yalın JDBC kodu ile Hibernate kodu karşılaştırıldığında, Hibernate'in veri tabanı işlemlerinden ne kadar yüksek seviyede soyutlama yaptığı görülmektedir. Yazılımcı veri tabanı işlemlerinin detayları ile kesinlikle ilgilenmeden uygulama bünyesinde yer alan işletme mantığına konsantre olabilmektedir. Hibernate'in yazılımcıya sağladığı en büyük avantaj budur.

Transaksiyon Yönetimi

Transaksiyonun Spring tarafından yönetilebilmesi için bir transaksiyon

yöneticisine (transaction manager) ihtiyaç duymaktayız. Böyle bir transactionManager tanımlaması kod 7.5 de yer almaktadır. Hibernate ile uyumlu transaksiyon yönetimi için HibernateTransactionManager sınıfını kullandık. Spring tarafından yönetilen transaksiyonun işleyiş şeması resim 7.1 de yer almaktadır.



Resim 7.1

TransactionManager nesnesi görevini yerine getirebilmek için bir sessionFactory nesnesine ihtiyaç duymaktadır. Kod 7.5 de yer alan örnekte transactionManager nesnesine sessionFactory nesnesini enjekte ettik. Yine kod 7.5 de yer alan tx:annotation-driven elementi ile transaksiyon yönetimi için hangi transactionManager nesnesini kullanmak istediğimizi tayin ettik. Bu konfigürasyon ile Spring transaksiyonu yönetmek için AOP yardımı ile bir proxy oluşturup, transaksiyon yönetimini transactionManager nesnesine delege edecektir.

Son olarak transaksiyonun nerede başlayıp, nerede son bulduğunu belirtmek için RentalServiceImpl sınıfının rentACar() metoduna @Transactional (kod 7.8) anotasyonunu ve Spring'in bu anotasyonu fark edebilmesi için konfigürasyon dosyasına tx:annotation-driven elementini konuşlandırmamız gerekmektedir (kod 7.5). Bu anotasyon yardımı ile transaksiyona dahil olacak metotları işaretlemiş olduk. Transaksiyon rentAcar() girmeden başlatılacak, rentAcar() bünyesinde koşturulan diğer metotlara yayılacak ve rentAcar() metodundan çıkılmasıyla birlikte sonlandırılacaktır.

Spring bu metodu bir transaksiyon içinde koşturabilmek için önce RentalServiceImpl sınıfına vekillik eden bir proxy nesnesi olusturur. Uygulamayı debug ettiğimiz zaman, resim 7.2 de de görüldügü gibi oluşturulan bu proxy rentACar() metoduna gelen istekleri karşılar. Bu istekler TransactionInterceptor aracılığı ile RentalServiceImpl sınıfına yönlendirilir. rentAcar() metodu son bulduğunda TransactionInterceptor.commitTransactionAfterReturning() metodunda herhangi bir hata oluşmaması durumunda commit ile transaksiyon olumlu sonlandırır. Bir hata oluşması durumunda TransactionInterceptor.completeTransactionAfterThrowing() metodunda transaksiyon rollback yapılır.

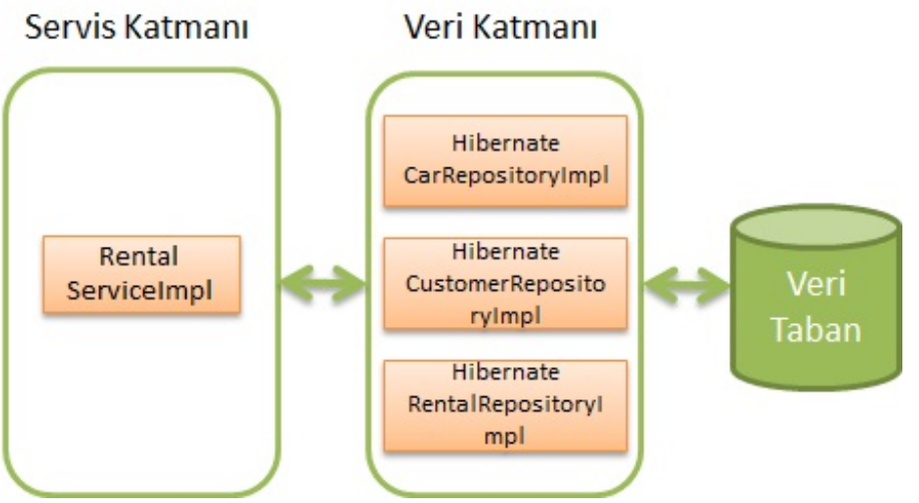
```
Kod 7.8 - RentalServiceImpl

@Transactional
@Override
public Rental rentACar(Customer customer, long carId,
            Date begin, Date end) {

    Rental rental = null;
    Customer dbCustomer =
        customerRepository.getCustomerByName(
                customer.getName());

    if (dbCustomer == null) {
        customerRepository.save(customer);
    }

    Car car = carRepository.findCarById(carId);

    rental = new Rental();
    rental.setCar(car);
    rental.setRented(true);
    rental.setCustomer(customer);
    rentalRepository.save(rental);
    return rental;
}
```

```
Main (4) [Java Application]
  com.kurumsaljava.spring.Main at localhost:62810
    Thread [main] (Suspended (breakpoint at line 20 in RentalServiceImpl))
      RentalServiceImpl.rentACar(Customer, long, Date, Date) line: 20
      NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
      NativeMethodAccessorImpl.invoke(Object, Object[]) line: 39
      DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 25
      Method.invoke(Object, Object...) line: 597
      AopUtils.invokeJoinpointUsingReflection(Object, Method, Object[]) line: 319
      ReflectiveMethodInvocation.invokeJoinpoint() line: 183
      ReflectiveMethodInvocation.proceed() line: 150
      TransactionInterceptor.invoke(MethodInvocation) line: 110
      ReflectiveMethodInvocation.proceed() line: 172
      JdkDynamicAopProxy.invoke(Object, Method, Object[]) line: 202
      $Proxy13.rentACar(Customer, long, Date, Date) line: not available
      Main.main(String[]) line: 19
  C:\Program Files\Java\jdk1.6.0_13\bin\javaw.exe (08.12.2012 13:04:35)
```

Resim 7.2

Hata Yönetimi

Herhangi bir hata oluşması durumunda Hibernate org.hibernate.HibernateException ve türevleri ile kullanıcısını haberdar edecektir. Bizim örneğimizde bu HibernateCustomerRepositoryImpl, HibernateCarRepositoryImpl ve HibernateRentalRepositoryImpl sınıflarıdır. Bu sınıflar uygulamanın veri erişim katmanında yer alan DAO (Data Access Object) sınıflarıdır.



Resim 7.3

Veri katmanında oluşan hataların birebir servis katmanına yönlendirilmesi servis katmanını kullanılan ORM teknolojisine bağımlı kılar. Bizim örneğimizde bu RentalServiceImpl sınıfının HibernateException sınıfından bir hatayı

yakalaması ve gerekli işlemleri yapması anlamına gelmektedir. Bunu engellemek için veri katmanında oluşan Hibernate hataları yakalanarak, servis katmanının kullanabileceği türde hatalara dönüştürülebilir. Bu gerekli exception sınıflarının oluşturulması, Hibernate hatalarının bu hatalara dönüştürülmesi ve yeni hataların servis katmanına yönlendirilmesi anlamına gelmektedir. Bu tür bir hata yönetimini Spring bizim için daha iyi yapabilir.

Spring oluşan Hibernate hatalarının başka hata sınıflarına dönüştürülmesini (exception translation) AOP yardımı ile mümkün kılmaktadır. Veri katmanında oluşan tüm hatalar Spring'in ihtiva ettiği org.springframework.dao.DataAccessException ve türevlerine otomatik olarak dönüştürülebilir. Bu amaçla @Repository anotasyonu kullanılabilir. @Repository anotasyonu otomatik hata dönüşümünü için gerekli mekanizmaların aktivasyonunu sağlar.

```
Kod 7.9 - HibernateCustomerRepositoryImpl

@Repository
public class HibernateCustomerRepositoryImpl
            implements CustomerRepository {

    private SessionFactory sessionFactory;

    @Override
    public Customer getCustomerByName(String name) {
        Query query = getCurrentSession().createQuery(
                "from Customer c where c.name=:name");
        query.setString("name", name);
        return (Customer) query.uniqueResult();

    }

    @Override
    public void save(Customer customer) {
        getCurrentSession().save(customer);
    }
}
```

Kod 7.9 da @Repository anotasyonunun kullanım şekli yer almaktadır. Oluşan Hibernate hatalarının otomatik olarak org.springframework.dao.DataAccessException ve türevlerine dönüştürülebilmesi için XML konfigürasyon dosyasında bir PersistenceExceptionTranslationPostProcessor tanımlaması yapılması gerekmektedir. Bu tanımlama kod 7.10 da yer almaktadır.

```
Kod 7.10 - applicationContext.xml

<bean class="org.springframework.dao.annotation.
    PersistenceExceptionTranslationPostProcessor"/>
```

@Repository anotasyonu kullanımının mümkün olmadığı durumlarda (örneğin kaynak kodun olmaması) hata dönüşümü için gerekli konfigürasyon kod 7.11 deki şekilde XML dosyasında yapılabilir.

```
Kod 7.11 - applicationContext.xml

<bean id="persistenceExceptionInterceptor"
        class="org.springframework.dao.support.
    PersistenceExceptionTranslationInterceptor"/>

<aop:config>
    <aop:advisor pointcut="execution(* *..Repository+.*(..))"
        advice-ref="persistenceExceptionInterceptor"/>
</aop:config>
```

Dönüşümden sonra oluşan org.springframework.dao.DataAccessException nesnesi meydana genel asıl hatanın tüm karakteristiklerini ihtiva etmeyebilir. Örneğin uygulama belli SQL hata kodlarını değerlendirerek kullanıcıya hata mesajları veriyor olabilir. Bu gibi bilgileri org.springframework.dao.DataAccessException türevlerinde uygulamanın üst katmanlarına taşıyabilmek için PersistenceExceptionTranslator interface sınıfının kod 7.11.1 de görüldüğü gibi yeniden implemente edilmesi gerekmektedir. Kod 7.11.1 de yer alan implementasyonda @Component anotasyonunu yer aldığı için Spring otomatik olarak hata dönüşümü için bu implementasyonu kullanmaya başlayacaktır. translateExceptionIfPossible() metodu bünyesinde orijinal hata metot parametresi olarak yer almaktadır. Bu metot bünyesinde istenilen hata bilgilerini taşıyan yeni bir DataAccessException türevi oluşturulabilir.

```
Kod 7.11.1 - CustomPersistenceExceptionTranslatorImpl

package com.kurumsaljava.spring.dao.impl;

import org.springframework.dao.DataAccessException;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.dao.support.
            PersistenceExceptionTranslator;
```

```
import org.springframework.stereotype.Component;

@Component
public class CustomPersistenceExceptionTranslatorImpl
            implements PersistenceExceptionTranslator {

    @Override
    public DataAccessException translateExceptionIfPossible(
                    final RuntimeException ex) {
        throw new DataIntegrityViolationException(
                ex.getMessage());
    }
}
```

XML Yardımı İle Eşleme (XML Mapping)

Kod 7.1 de yer alan Customer sınıfını JPA anotasyonları kullanarak bir veri tabanı ögesi (entity) haline getirmiştik. JPA anotasyonlarının kullanımı her zaman mümkün olmayabilir. Bunun yanı sıra eski uygulamaların çoğu XML bazlı eşleme türünü kullanmaktadır. Bu bölümde XML yardımı ile uygulama sınıfı, veri tabanı tablosu eşlemesinin nasıl yapılabileceğini inceleyeceğiz.

Bu tür eşlemede her sınıf için bir XML eşleme (mapping) dosyası oluşturulur. Kod 7.12 de Customer sınıfı için kullandığımız XML dosyası yer almaktadır. Sınıf, veri tabanı arasındaki eşleme işlemi hibernate-mapping elementi ile yapılır. Package element özelliği ile eşlenecek sınıfın hangi paket içinde yer aldığı belirtilir. Class elementi name element özelliği yardımı ile eşlenecek sınıfın ismini belirlemek için kullanılır. Table element özelliği Hibernate tarafından oluşturulan veri tabanı tablosunun ismini belirler.

Veri tabanı tablosundaki anahtar kolon (primary key) id elementi ile tanımlanır. Anahtar kolonun değeri tekildir ve generator elementi ile bu değerin her yeni kayıt ile bir artırılması sağlanır. Column element özelliği anahtar kolonun sınıf bünyesindeki hangi değişkene denk düştüğünü gösterir.

Property elementi ile veri tabanı tablosuna eklenecek sınıf değişkenleri belirlenir. Name element özelliği değişkenin ismini ihtiva eder. Burada kullanılacak column element özelliği ile tablodaki kolon ismi tayin edilebilir. Örneğin sınıf değişkeni name ismini taşıyorsa, bu değişken için oluşturulan kolon ismi column=customer_name olarak belirlenebilir. Sınıf değişken isimleri ile kolon isimleri aynı ismi taşımak zorunda değildirler.

```
Kod 7.12  - customer.hbm.xml

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/
                hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.kurumsaljava.spring.xml">
    <class name="Customer" table="customer">
        <id name="id" column="id">
            <generator class="sequence" />
        </id>
        <property name="name" />
        <property name="firstname" />
        <property name="age" />
    </class>
</hibernate-mapping>
```

Car sınıfı için gerekli XML eşleme dosyası kod 7.13 de yer almaktadır. Car sınıfı yapı olarak Customer sınıfı ile aynı özellikleri taşıdığı için, oluşturulan eşleme XML dosyaları birbirine benzemektedir.

```
Kod 7.13  - car.hbm.xml

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/
            hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.kurumsaljava.spring.xml">
    <class name="Car" table="car">
        <id name="id" column="id" access="field">
            <generator class="sequence" />
        </id>
        <property name="brand" />
        <property name="model" />
    </class>
</hibernate-mapping>
```

Rental sınıfı için gerekli XML eşleme dosyası kod 7.14 de yer almaktadır. Rental sınıfı bünyesinde bir Car ve bir Customer nesnesi taşıdığı için, bu ilişkiler one-to-one elementi elementi kullanılarak tanımlanmıştır.

```
Kod 7.14  - rental.hbm.xml

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/
        hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.kurumsaljava.spring.xml">
    <class name="Rental" table="rental">
        <id name="id" column="id" access="field">
            <generator class="sequence" />
        </id>
        <one-to-one name="customer"
            class="com.kurumsaljava.spring.xml.Customer"/>
        <one-to-one name="car"
            class="com.kurumsaljava.spring.xml.Car"/>
        <property name="rented" />
    </class>
</hibernate-mapping>
```

XML Eşleme İçin SessionFactory Konfigürasyonu

Uygulamanın XML bazlı eşleme tarzında çalışabilmesi için SessionFactory konfigürasyonunun kod 7.15 deki gibi adapte edilmesi gerekmektedir. MappingLocations kullanılarak XML dosyalarının lokasyonu tanımlanır. Kod 7.15 de yer alan örnekte XML dosyaları classpath içinde yer almaktadır.

```
Kod 7.15  - applicationContext.xml

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
            LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingLocations">
        <list>
            <value>classpath:customer.hbm.xml</value>
            <value>classpath:car.hbm.xml</value>
            <value>classpath:rental.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.format_sql=true
            hibernate.show_sql=true
            hibernate.hbm2ddl.auto=create
            hibernate.hbm2ddl.show=true
        </value>
    </property>
</bean>
```

SessionFactory konfigürasyonunu hem XML, hem de anotasyonu bazlı eşleme kullanılacak şekilde konfigüre etmek mümkündür. Bu genelde mevcut XML bazlı eşleme yapılan bir uygulamanın migrasyonunda kullanılan bir yöntemdir. Yeni sınıflar için anotasyon kullanılırken, eski sınıflar XML bazlı eşlemeyi kullanmaya devam ederler. Bunun yanı sıra named-query gibi yapıların XML bazlı eşlemede kullanımları daha kolaydır. Kod 7.16 da her iki tür eşlemenin beraber kullanımı yer almaktadır.

```
Kod 7.16  - applicationContext.xml

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
        annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.kurumsaljava.spring.Rental</value>
        </list>
    </property>
    <property name="mappingLocations">
        <list>
            <value>classpath:customer.hbm.xml</value>
            <value>classpath:car.hbm.xml</value>
        </list>
    </property>
</bean>
```

HibernateTemplate Kullanımı

HibernateTemplate kullanımı otomatik Hibernate Session ve transaksiyon yönetimi sağlamaktadır. Bunun yanı sıra oluşan spesifik Hibernate hataları HibernateTemplate tarafından Spring'in ihtiva ettiği DataAccessException ve türevlerine dönüştürülür. HiberneteTamplete kullanımı kod 7.17 de yer almaktadır.

```
Kod 7.17  - RentalServiceImpl

package com.kurumsaljava.spring.template;
import java.sql.SQLException;
import java.util.Date;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;
```

```
import org.springframework.orm.hibernate3.HibernateTemplate;

public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;
    private CarRepository carRepository;
    private HibernateTemplate template;

    public RentalServiceImpl() {
    }


    @Override
    public Rental rentACar(final Customer customer, final long carId,
            final Date begin, final Date end) {

        return (Rental)template.execute(
                new HibernateCallback<Object>() {
            @Override
            public Object doInHibernate(
                Session session) throws HibernateException,
                    SQLException {
                Rental rental = null;
                Customer dbCustomer =
                    customerRepository.getCustomerByName(
                        customer.getName());

                if (dbCustomer == null) {
                    customerRepository.save(customer);
                }

                Car car = carRepository.findCarById(carId);

                rental = new Rental();
                rental.setCar(car);
                rental.setRented(true);
                rental.setCustomer(customer);
                rentalRepository.save(rental);
                return rental;
            }
        });

    }
}
```

Kod 7.17 de kullanılan template nesnesi kod 7.18 deki şekilde RentalServiceImpl sınıfına enjekte edilebilir. HibernateTemplate sınıfı bir sessionFactory nesnesine

ihtiyaç duymaktadır. SessionFactory nesnesinin konfigürasyonu kod 7.16 da yer almaktadır.

```
Kod 7.18  - applicationContext.xml

<bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.
                HibernateTemplate">
    <property name="sessionFactory"
                ref="sessionFactory"/>
</bean>


<bean id="rentalService"
        class="com.kurumsaljava.spring.template.
            RentalServiceImpl">
    <property name="customerRepository"
                ref="customerRepository" />
    <property name="rentalRepository" ref="rentalRepository" />
    <property name="carRepository" ref="carRepository" />
    <property name="template" ref="hibernateTemplate" />
</bean>
```

HibernateTemplate'in sahip olduğu find(), save(), saveOrUpdate() ve delete() metotları ile doğrudan veri tabanı işlemleri yapılabilmektedir. Bunun bir örneği kod 7.19 da yer almaktadır.

Kod 7.19 da yer alan getCustomerByName() metodu bünyesinde HibernateTemplate ve HQL (Hibernatea Query Language) yardımı ile soyisim (name) kullanılarak bir müşteri nesnesi aranmaktadır. Müşteri aramak için oluşturulan HQL'in koşturulabilmesi için HibernateTemplate sınıfının sahip olduğu find() metodu kullanılmaktadır. find() metodu müşteri nesnelerinin yer aldığı bir liste geri verir. Bu listenin tek bir müşteri nesnesi ihtiva edip, etmediğini DataAccessUtils.uniqueResult ile kontrol edebiliriz. Listenin tek bir müşteri ihtiva etmesi durumunda, uniqueResult() metodu bu müşteriyi geri verecektir. Birden fazla müşteri bulunması durumunda IncorrectResultSizeDataAccessException hatası ile durumu kullanıcıya bildirilir. Listenin herhangi bir nesne ihtiva etmemesi durumunda uniqueResult() tarafından null değeri geri verilir.

```
Kod 7.19  - HibernateCustomerRepositoryImpl

package com.kurumsaljava.spring.template;
```

```
import org.springframework.dao.support.DataAccessUtils;
import org.springframework.orm.hibernate3.HibernateTemplate;

public class HibernateCustomerRepositoryImpl
            implements CustomerRepository {

    private HibernateTemplate template;

    @SuppressWarnings("unchecked")
    @Override
    public Customer getCustomerByName(String name) {
        return DataAccessUtils.uniqueResult(
            template.find("from Customer c where c.name=?",
                    name));

    }

    public HibernateTemplate getTemplate() {
        return template;
    }

    public void setTemplate(HibernateTemplate template) {
        this.template = template;
    }

    @Override
    public void save(Customer customer) {
        template.save(customer);
    }
}
```

Spring 3.0 ve Hibernate 3.0.1 ile Hibernate Session nesnesinin HibernateTemplate ile yönetimi gerekliliği ortadan kalkmıştır. Hibernate'in sunduğu kontekst bazlı oturum (contextual sessions) ile Hibernate Session nesneleri Hibernate tarafından mevcut transaksiyona uyumlu olarak yönetilmektedir. Bunun nasıl kullanıldığını daha önceki örneklerimizde görmüştük. Kod 7.6 ya baktığımızda getCurrentSession() metodu aracılığı ile Hibernate Session nesnesini edindiğimizi görmekteyiz. Bu örnekte Hibernate Session nesnelerini kendisi yönetmektedir.

HibernateTemplate sınıfının kullanımı kodu Spring çatışına bağımlı kılmaktadır. Kod 7.9 da olduğu gibi HibernateTemplate yerine org.hibernate.SessionFactory sınıfını kullanarak sadece Hibernate API'sine bağımlı bir DAO oluşturabiliriz.

HibernateTemplate sınıfının kullanılmasının diğer bir sebebi ise spesifik

Hibernate hatalarının otomatik olarak DataAccessException ve türevlerine dönüştürülmesidir. @Repository anotasyonu ile bunu sağlamanın bir alternatifi bulunmaktadır. Bu durum da HibernateTemplate sınıfının kullanılma gerekliliğini ortadan kaldırmaktadır.

Görüldüğü gibi HibernateTemplate Spring 3 öncesinden kalma bir yapıdır. Bu sebepten dolayı Spring 3 ile geliştirilen uygulamalarda kullanımı tavsiye edilmemektedir. HibernateTemplate sınıfının kullanılmaması için sunduğum argümanlar HibernateDaoSupport sınıfı için de geçerlidir.

7. Bölüm Soruları

7.1 Hibernate gibi bir ORM aracının kullanımının yazılımcıya sağladığı en büyük avantaj nedir?

7.2 Hangi Spring sınıfının kullanımı otomatik Hibernate Session ve transaksiyon yönetimi sağlar?

7.3 Bu sınıfın kullanımındaki en büyük dezavantaj nedir?

7.4 Hibernate uygulamalarında transaksiyonu yönetmek için kullanılan Spring sınıfı hangisidir?

8. Bölüm

Spring İle JPA Kullanımı

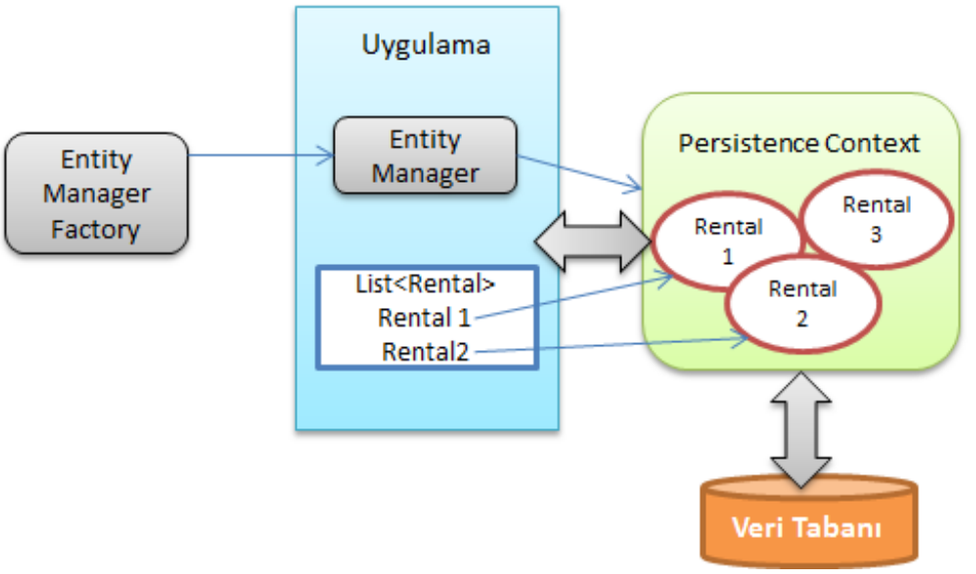
JPA Nedir?

JPA (Java Persistence API) nesne/relasyonel eşleme (ORM - Object/Relational Mapping) yapmak için kullanılan standart bir Java uygulama geliştirme arayüzüdür (Application Programming Interface). İlk sürümü 2006 yılında kullanıma sunulan JPA ile POJO (Plain Old Java Object) bazlı alan nesnelerinin (domain objects) relasyonel veri tabanlarında tutulabilmeleri öngörülmektedir. JPA Hibernate ve Toplink gibi ORM araçlarından elde edilen tecrübelerle geliştirilmiş bir uygulama geliştirme arayüzüdür. İkinci sürümü 2009 yılında kullanıma sunulan JPA doğrudan bir ORM aracı değildir. JPA'i daha ziyade uygulamayı kullanılan ORM aracından bağımsız kılmak için kullanılan bir ara katman olarak düşünmek gerekir. JPA kullanan uygulamalarda geniş çaplı değişiklik yapmadan kullanılan ORM aracı değiştirilebilir. Hibernate ve EclipseLink gibi ORM araçlarının JPA implementasyonları bulunmaktadır.

JPA bünyesinde EntityManager, EntityManagerFactory, PersistenceContext ve PersistenceUnit gibi konseptler bulunmaktadır. Bu konseptleri ve kullanılış şekillerini bu bölümde yakından inceleyeceğiz. Spring 2.0 sürümüyle birlikte JPA entegrasyonunu desteklemektedir. Spring bünyesinde JPA kullanabilmek için bir EntityManagerFactory konfigürasyonu yapılması gerekir. Bunun nasıl yapıldığını bir sonraki bölümde inceleyecegiz.

EntityManagerFactory ve EntityManager

Resim 8.1 de genel hatlarıyla JPA'in bir uygulama bünyesinde kullanılış tarzı yer almaktadır. JPA bazlı uygulamalar EntityManagerFactory yardımı ile bir EntityManager nesnesi edinirler. Veri tabanı işlemleri EntityManager aracılığı ile gerçekleştirilir. Bir transaksiyonu gerçekleştirmek için edinilen EntityManager nesnesi üzerinde işlem yaptığı nesneleri persistence context ismi verilen hafıza alanında tutar. Bu hafıza alanını veri tabanı verileri için kullanılan bir cache olarak düşünebiliriz. Örneğin bir müşteri nesnesi veri tabanında edinildikten sonra persistence context içine yerleştirilir. Aynı müşteri nesnesine ihtiyaç duyulduğunda, bu nesneyi veri tabanını tekrar sorgulamadan persistence context içinden almak mümkündür. Bu nesneler üzerinde yapılan değişiklikler için de geçerlidir. Her yapılan değişiklik tek tek değil, topluca cache boşaltıldığında (flush) gerçekleştirilir. JPA persistence context yardımı ile entity sınıfların kontrolünü ve yönetimini sağlar.



Resim 8.1

JPA bünyesinde iki değişik EntityManager tipi mevcuttur. Bunlar:

- Uygulama tarafından yönetilen EntityManager - Bu tip EntityManager doğrudan EntityManagerFactory nesnesinden edinilir (kod 8.1). Uygulama EntityManager nesnesinin edinilmesi, sonlandırılması ve transaksiyonlarda kullanımından kendisi sorumludur. Bu tip EntityManager daha çok birim testleri ve uygulama sunucusuna ihtiyaç duymayan (standalone) uygulamalarda kullanılır.
- Uygulama sunucusu (container) tarafından yönetilen EntityManager - Bu tip EntityManager uygulama sunucusu tarafından oluşturulur ve tüm yaşam döngüsü yine uygulama sunucusu tarafından yönetilir. Uygulama doğrudan EntityManagerFactory ile interaksiyona girerek bir EntityManager nesnesi edinmez. Daha ziyade bir EntityManager nesnesi @PersistenceContext anotasyonu yardımı ile uygulamaya enjekte edilir.

İki EntityManager tipi arasındaki fark, EntityManager nesnelerinin nasıl oluşturulup, yönetildikleridir. Hangi EntityManagerFactory kullanılırsa kullanılsın, Spring EntityManager nesnelerinin otomatik yönetimini sağlayacaktır. Eğer uygulama tarafından yönetilen EntityManager tipi kullanılıyorsa, Spring uygulamanın kendisiymiş gibi hareket ederek, EntityManager nesnelerini yönetir. Eğer uygulama sunucusu tarafından yönetilen EntityManager tipi seçildiyse Spring uygulama sunucusu rolünü üstlenir.

```
Kod 8.1 - Main
```

```
public static void main(String[] argv) {
    ApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
    EntityManagerFactory factory =
                (EntityManagerFactory) context.getBean("emf");
    EntityManager manager = factory.createEntityManager();

    EntityTransaction transaction = manager.getTransaction();
    transaction.begin();
    List result = manager.createQuery("
                select c.name from Customer c").getResultList();
    System.out.println(result.size());
    transaction.commit();

    manager.close();
    factory.close();
}
```

Bahsettiğim her iki EntityManager tipini oluşturmak için aşağıda yer alan Spring factory bean sınıfları kullanılır:

- LocalEntityManagerFactoryBean - uygulama tarafından yönetilen EntityManager nesnelerini sağlayan bir EntityManagerFactory nesnesi oluşturur.
- LocalContainerEntityManagerFactoryBean - uygulama sunucusu tarafından yönetilen EntityManager nesnelerini sağlayan bir EntityManagerFactory nesnesi oluşturur.

Uygulama Tarafından Yönetilen EntityManager Konfigürasyonu

Uygulama tarafından yönetilen EntityManager kullanımını sağlayan EntityFactoryManager konfigürasyonu için Spring konfigürasyon dosyasında bir LocalEntityManagerFactoryBean bean tanımlaması yapmamaz gerekmektedir. Bu konfigürasyon kod 8.2 de yer almaktadır.

```
Kod 8.2 - jpa-application-managed.xml

<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.
            LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName"
            value="rentAcar-application-managed" />
</bean>
```

EntityManagerFactory tanımlaması persistenceUnitName ismini taşıyan bir değişkene ihtiyaç duymaktadır. Bu değişken kullanılan persistene unit'e işaret etmektedir. Bir persistence unit kullanılan entity sınıfları, hangi ORM aracının ve transaksiyon tipinin (local, JTA) kullanıldığını tanımlar. Uygulama birden fazla persistence unit kullanabilir. Persistence unit'ler META-INF dizininde yer alan persistence.xml dosyasında tanımlanır. Araç kiralama servisi uygulamamız için tanımladığımız persistence unit kod 8.3 de yer almaktadır.

```
Kod 8.3 - persistence.xml

<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/
                persistence_1_0.xsd">
    <persistence-unit name="rentAcar" />

    <persistence-unit name="rentAcar-application-managed">
        <class>com.kurumsaljava.spring.Car</class>
        <class>com.kurumsaljava.spring.Customer</class>
        <class>com.kurumsaljava.spring.Rental</class>
        <properties>
            <property name="hibernate.connection.url"
                    value="jdbc:hsqldb:mem:spring-playground" />
            <property name="hibernate.connection.driver_class"
                    value="org.hsqldb.jdbcDriver" />
            <property name="hibernate.dialect"
                    value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.connection.username"
                        value="sa" />
            <property name="hibernate.connection.password"
                        value="" />
            <property name="hibernate.hbm2ddl.auto"
                        value="create" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.show"
                        value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Kod 8.4 de yer alan kod örneğinde uygulama EntityManagerFactory ve EntityManager nesnelerini kendisi edinmekte ve yine kendi imkanları ile transaksiyonu yönetmektedir. Bu örnekte JPA kullanımı tamamen uygulamanın

kendi inisiyatifindedir.

```
Kod 8.4 - Main

public static void main(String[] argv) {
    ApplicationContext context=
            new ClassPathXmlApplicationContext(
                "jpa-application-managed.xml");
    EntityManagerFactory factory = (EntityManagerFactory)
                    context.getBean("entityManagerFactory");
    EntityManager manager = factory.createEntityManager();

    EntityTransaction transaction = manager.getTransaction();
    transaction.begin();
    List result = manager.createQuery("
                select c.id from Customer c").getResultList();
    System.out.println(result.size());
    transaction.commit();
    manager.close();
    factory.close();
}
```

Uygulama Sunucusu Tarafından Yönetilen EntityManager Konfigürasyonu

Uygulama sunucusu tarafından yönetilen EntityManager kullanımını sağlayan EntityFactoryManager konfigürasyonu için Spring konfigürasyon dosyasında bir LocalContainerEntityManagerFactoryBean bean tanımlaması yapmamız gerekmektedir. Bu konfigürasyon kod 8.5 de yer almaktadır.

```
Kod 8.5 - jpa-container-managed.xml

<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.
        LocalContainerEntityManagerFactoryBean">
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.
                vendor.HibernateJpaVendorAdapter"/>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
            <prop key="hibernate.hbm2ddl.show">true</prop>
        </props>
```

```
    </property>
    <property name="dataSource" ref="dataSource" />
</bean>
```

LocalEntityManagerFactoryBean farklı olarak LocalContainerEntityManagerFactoryBean tanımlamasında bir dataSource kullanmak mümkündür. Bu bir persistence unit tanımlama gerekliliğini ortadan kaldırmaktadır. Context:annotation-config elementi kullanarak JPA anotasyonlarını taşıyan entity sınıfların Spring tarafından otomatik olarak keşfedilmelerini sağlayabiliriz.

EntityManagerFactory tanımlamasında kullanılan jpaVendorAdapter kullanılan ORM aracına işaret etmektedir. Spring sahip olduğu adapter sınıfları ile istenilen türdeki ORM aracını uygulamaya entegre etmektedir. Kullanılabilecek adapter sınıfları şunlardır:

- HibernateJpaVendorAdapter
- EclipseLinkJpaVendorAdapter
- TopLinkJpaVendorAdapter
- OpenJpaVendorAdapter

JPA Anotasyonları

JPA anotasyonları ile daha önce yedinci bölümde yer alan Hibernate örneklerinde tanışmıştık. Oluşturduğumuz Car, Customer ve Rental sınıflarını değişikliğe lüzum olmadan JPA implementasyonumuzda kullanabiliriz.

EntityManager API

Veri tabanı işlemleri için EntityManager sınıfı aşağıda yer alan metotları ihtiva etmektedir:

- ***persist(Object o)*** - Nesneyi persistence context'e ekler. SQL dilinde bu insert into table komutuna denk gelmektedir.
- ***remove (Object o)*** - Nesneyi persistence context'den siler. SQL dilinde bu delete from table komutuna denk gelmektedir.
- ***find (Class entity, Object primaryKey)*** - Anahtar alanı (primary key) kullanarak arama yapar. SQL dilinde bu select * from table where id=? komutuna denk gelmektedir.
- ***Query createQuery(String jpqlString)*** - Veri sorgulama yapmak

için bir Query nesnesi oluşturur.

- ***flush()*** - Değişikliğe uğrayan nesnelerin taşıdığı değerlerin veri tabanına aktarılmasını sağlar.
- ***merge(Object entity)*** - Nesne üzerinde yapılan değişikliklerin persistence context içinde bulunan son hali ile birleştirilmesini sağlar.
- ***refresh(Object entity)*** - Nesnenin tekrar veri tabanından yüklenmesini sağlar.
- ***clear()*** - Persistence context'i boşaltır ve içinde bulunan nesnelerin veri tabanı ile olan bağlantılarını koparır (detach).

JNDI Üzerinden EntityManagerFactory Edinme

Weblogic ya da JBoss gibi bir uygulama sunucusunda koşturulan bir uygulama mevcut tanımlanmış bir EntityManagerFactory nesnesini jee:jndi-lookup elementi şu şekilde edinebilir:

```
<jee:jndi-lookup id="entityManagerFactory"
        jndi-name="persistence/rentACar"/>
```

JPA İle DAO Kullanımı

JPA ile veri tabanı işlemi yapabilmek için bir EntityManager nesnesi edinmemiz gerekmektedir. Spring konfigürasyon dosyasında kullanmak istediğimiz EntityManagerFactory nesnesini tanımladıktan sonra, Spring tarafından veri tabanı işlemlerinin yapıldığı sınıfa @PersistenceContext anotasyonu yardımı ile bir EntityManager nesnesini enjekte ettirebiliriz. Kod 8.6 da yer alan JpaCustomerRepositoryImpl bu işlemin yapılış tarzı yer almaktadır.

```
Kod 8.6 - JpaCustomerRepositoryImpl

package com.kurumsaljava.spring;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;

@Repository
public class JpaCustomerRepositoryImpl
            implements CustomerRepository {
```

```
    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    @Override
    public Customer getCustomerByName(String name) {

        Customer result = null;
        try {
            result = (Customer) entityManager
                    .createQuery("from Customer c
                        where c.name=:name")
                    .setParameter("name", name)
                    .getSingleResult();

        } catch (NoResultException e) {
            // record bulunamadigi için null geri veriyoruz.
        }
        return result;
    }

    @Transactional
    @Override
    public void save(Customer customer) {
        entityManager.persist(customer);
    }
}
```

@PersistenceContext anotasyonunun kullanımı Spring API'sinden tamamen bağımsız bir DAO sınıfı oluşturmamızı mümkün kılmaktadır. Buna alternatif olarak Spring'in ihtiva ettiği JpaTemplate ve JpaDaoSupport sınıflarını kullanarak buna benzer DAO sınıfları oluşturmamız mümkün olacaktır. Lakin bu DAO sınıfını Spring API'sine bağımlı kılacaktır.

Yedinci bölümün sonunda neden HibernateTemplate sınıfının kullanılmaması gerektiği konusuna değinmiştim. Orada yer alanlar JpaTemplate ve JpaDaoSupport sınıfları için de geçerlidir. Spring ve JPA ile uygulama geliştirilirken bu sınıfların kullanılması tavsiye edilmemektedir.

JpaCustomerRepositoryImpl sınıfında kullandığımız @Repository anotasyonu oluşan JPA hatalarının, örneğin NoResultException otomatik olarak Spring tarafından org.springframework.dao.DataAccessException ve türevlerine dönüştürülmesini sağlamaktadır. Bu DAO sınıfını kullanan üst katmanların kullanılan ORM aracına has hata mesajları yerine org.springframework.dao.DataAccessException ve türevleri görmesini ve

böylece kullanılan ORM teknolojisinden bağımsız bir şekilde geliştirilebilmelerini mümkün kılmaktadır. Oluşan JPA hatalarının otomatik olarak org.springframework.dao.DataAccessException ve türevlerine dönüştürülebilmesi için XML konfigürasyon dosyasında bir PersistenceExceptionTranslationPostProcessor tanımlaması yapılması gerekmektedir. Bu tanımlama kod 8.7 da yer almaktadır.

```
Kod 8.7 - jpa-container-managed.xml

<bean class="org.springframework.dao.annotation.
    PersistenceExceptionTranslationPostProcessor"/>
```

Kod 8.7 de gibi bir konfigürasyonun yapılmaması oluşan tüm JPA hatalarının birebir değiştirilmeden üst katmanlara yönlendirilmelerine sebep verecektir.

JPA İle Transaksiyon Yönetimi

JpaCustomerRepositoryImpl (Kod 8.6) sınıfı bünyesinde yer alan metotlar @Transactional anotasyonu ile transaksiyonel olma özelliğine kavuşmuşlardır. Otomatik transaksiyon yönetimi için Spring konfigürasyon dosyasında bir transactionManager nesnesinin tanımlanması gerekmektedir. Böyle bir tanımlama kod 8.8 de yer almaktadır.

```
Kod 8.8 - jpa-container-managed.xml

<tx:annotation-driven
        transaction-manager="transactionManager" />

<jdbc:embedded-database id="dataSource" type="HSQL" />

<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.
            LocalContainerEntityManagerFactoryBean">
    <property name="jpaVendorAdapter">
        <bean
        class="org.springframework.orm.jpa.vendor.
            HibernateJpaVendorAdapter"/>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
            <prop key="hibernate.hbm2ddl.show">true</prop>
```

```
        </props>
    </property>
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="transactionManager"
            class="org.springframework.orm.jpa.
                JpaTransactionManager">
    <property name="entityManagerFactory"
            ref="entityManagerFactory" />
</bean>
```

org.springframework.orm.jpa.JpaTransactionManager Spring'in ihtiva ettiği JPA bazlı bir transaksiyon yöneticisidir. Oluşturulan EntityManager nesnelerinin transaksiyona dahil edilebilmeleri için bu nesneleri oluşturan EntityManagerFactory nesnesinin kullanılan JPA transaksiyon yöneticisine enjekte edilmesi gerekmektedir. Anotasyon bazlı transaksiyon yönetimini kullanabilmek için konfigürasyon dosyasında tx:annotation-driven elementini tanımlamamız gerekmektedir. Bu element tanımlanan transaksiyon yöneticisini transaction-manager element özelliği ile edinerek, @Transactional anotasyonunun kullanıldığı her yerde tanımlanan transaksiyon yöneticisinin kontrolü ele almasını mümkün kılar.

Kitabın diğer bölümlerde JDBC ve Hibernate'in kullanılış şekillerini yakından incelemiştik. JPA veri tabanı işlemleri acısından Java uygulamalarında en yüksek soyutluk seviyesini temsil etmektedir. Bu sebepten dolayı yeni geliştirilen Spring uygulamalarında JPA API'sinin kullanımını şiddetle tavsiye ediyorum. Bu API'nin kullanımı uygulamayı kullanılan veri tabanı ve ORM aracından bağımsız kılarak, uygulamanın geliştirilmesini ve bakımını kolaylaştıracaktır.

8. Bölüm Soruları

8.1 JPA bir ORM aracı mıdır?

8.2 JPA API'sinde ana veri tabanı işlemlerini gerçekleştiren yapı hangisidir?

8.3 Uygulama sunucusu bünyesindeki EntityManagerFactory nasıl edinilir?

8.4 Bir sınıfa EntityManager nesnesini enjekte etmek için hangi anotasyon kullanılır?

8.5 Bir sınıf bünyesinde JPA üzerinden veri tabanı işlemleri yapmak için hangi Spring sınıfı kullanılır?

8.6 JPA işlemlerinde oluşan hataları org.springframework.dao.DataAccessException tipine dönüştüren mekanizmalarından birisi hangisidir?

9. Bölüm

Spring İle Aspect Oriented Programming

Tipik bir uygulamaya baktığımızda işletme mantığı ile harmanlanmış, uygulamanın birçok yerinde kullanılan jenerik fonksiyonlarla karşılaşırız. Bunlardan bazıları:

- Loglama
- Transaksiyon yönetimi
- Güvenlik
- Cache kullanımı
- Hata yönetimi
- Performans ölçümleri
- İş kuralları (business rules) uygulanması

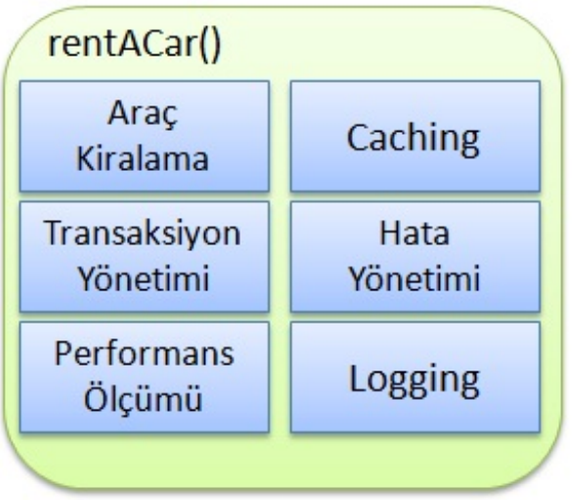
Bu tür fonksiyonlar işletme mantığının gerek duyduğu ya da duymadığı türde fonksiyonlar olabilir. Örneğin işletme mantığı transaksiyon yönetimine ihtiyaç duyabilirken, performans ölçümüne doğrudan ihtiyacı yoktur. Bu daha ziyada uygulamanın performansını ölçmek için işletme mantığına eklenmesi gereken bir fonksiyondur. İşletme mantığı eklenen bu jenerik fonksiyonlarla anlaşılmaz bir hal alabilir.

```
Kod 9.1 - JdbcRentalServiceImpl

public Rental rentACar(Customer customer, Car car,
                    Date begin, Date end) {

    logger.debug("rentACar() enter");
    long startTrace = System.currentTimeMillis();
    long endTrace = 0;

    Connection con = getConnection();
    Rental rental = null;
    try {
        con.setAutoCommit(false);

        Customer dbCustomer = (Customer) CacheManager.get(customer
                .getName());
        if (dbCustomer == null) {
            dbCustomer = customerRepository.
                getCustomerByName(customer.getName());
        }

        rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        con.commit();
```

```
    } catch (Exception e) {
        if (con != null) {
            try {
                con.rollback();
            } catch (SQLException e1) {
                throw new RuntimeException(e1);
            }
        }
        throw new RuntimeException(e);
    }
    endTrace = System.currentTimeMillis();
    logger.debug("rentACar() exit");
    logger.debug("rentACar() execution time " +
            (endTrace-startTrace) + " ms");
    return rental;
}
```

Kod 9.1 de yer alan rentACar() metodunun ilk bakışta ne yaptığını söylemek mümkün müdür? Bu metot bünyesinde olup, bitenleri anlayabilmek için metot gövdesinde yer alan kod satırlarını tek tek incelememiz gerekmektedir. Bu inceleme esnasında rentACar() metodunun aslında araç kiralama işlemi haricinde log4j yardımı ile loglama, transaksiyon yönetimi, hata yönetimi, performans ölçümü, caching gibi işlemler yaptığını görmekteyiz. Bu metotta yer alması gereken tek işletme mantığı araç kiralama işlemidir. Bu açıdan bakıldığında rentaACar() metodu birden fazla işle meşgul olmaktadır, yani birden fazla sorumluluğu vardır. Bu sebepten dolayı rentACar() metodu tek sorumluluk prensibiyle uyumlu değildir.

rentACar() metodunda araç kiralama için gerekli işletme mantığı haricinde yer alan fonksiyonlara cross cutting concern, yani işletme mantığı üzerinde yer alan ve her işletme mantığında ortak kullanım potansiyeli olan fonksiyonlar ismi verilmektedir.



Resim 9.1

Resim 9.1'de rentACar() metodu bünyesinde olup, bitenleri bir diyagram yardımı ile görsellestirmeye çalıştım. Araç kiralama iş mantığı rentAcar() bünyesinde yer alması gereken tek işlemdir. Sadece bu işletme mantığını ihtiva eden rentACar() metodu kod 9.2 de görüldüğü gibi şekillendirilebilir.

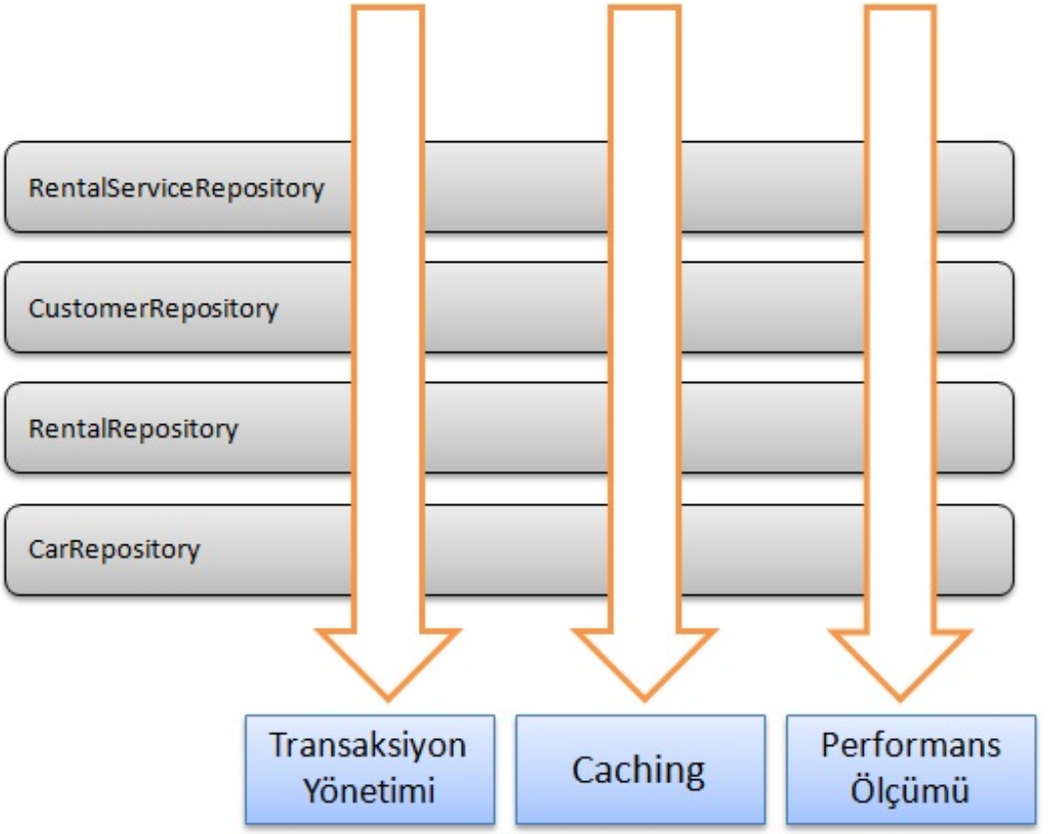
```
Kod 9.2 - JdbcRentalServiceImpl

public Rental rentACar(Customer customer, Car car,
            Date begin, Date end) {
    dbCustomer = customerRepository.getCustomerByName(
                customer.getName());
    rental = new Rental();
    rental.setCar(car);
    rental.setCustomer(customer);
    rentalRepository.save(rental);
    return rental;
}
```

Görüldüğü üzere sadece işletme mantığını ihtiva eden yeni rentACar() metodu daha sade ve anlaşılır hale gelmiştir. rentACar() metodunun yapısını bu şekilde muhafaza edip, kod 9.1 de yer alan loglama, transaksiyon yönetimi, caching ve hata yönetimi gibi işlemleri yapabilir miyiz? Teknik olarak bu nasıl mümkün olurdu?

Teknik olarak bunu yapmamız mümkün. Bu amaçla Java reflection ya da vekil (proxy) tasarım şablonunu kullanabiliriz. Yapmamız gereken tek şey rentACar() metoduna girmeden önce transaksiyonu ve performans ölçümü gibi işlemleri başlatmak ve metodun son bulmasıyla bu işlemleri sonlandırmaktır. Bu şekilde metot öncesi ve metot sonrası olmak üzere işletme mantığını şişirmek zorunda

kalmadan gerekli işlemleri yapabiliriz. Spring örneğin transaksiyon yönetimi için vekil tasarım şablonunu kullanmaktadır. Spring'in bu konuda faydalandığı başka bir teknoloji daha vardır. Bu teknoloji bu bölümün konusunu teşkil eden AOP'dir (Aspect Oriented Programming).

Resim 9.1 de yer alan işlemlerin hepsi kendi başlarına birer çalışma sahasıdır ve kodun okunabilirliğini, bakımını ve geliştirilmesini kolaylaştırmak için işletme mantığından ayrı bir yere konuşlandırılmaları gerekmektedir (resim 9.2). Bu ayrıştırmaya seperation of concerns (çalışma sahalarının ayrımı) ismi verilmektedir. AOP bunu gerçekleştirmek için vardır. AOP uygulamanın genelinde kullanılabilecek jenerik fonksiyonların modüler bir şekilde, işletme mantığı ile harmanlamaya gerek kalmadan ama birlikte çalışacak şekilde oluşturulmaları mümkün kılar. Transaksiyon yönetimi, loglama, caching, performans ölçümü ve hata yönetimi gibi işlemlere AOP dilinde aspekt ismi verilmektedir. AOP ile her bir aspekt kodu değiştirmeden bytecode seviyesinde işletme mantığı ile harmanlanır. Bu uygulamayı geliştirirken programcının işletme mantığına konsantre olmasını kolaylaştırır. AOP kullanılmaması yazılan kodda 9.1 de görüldüğü gibi işletme mantığı ile jenerik fonksiyonların iç, içe geçerek, jenerik fonksiyonların tüm kod tabanına yayılmasına (code dublication) ve kodun bakımının ve geliştirilmesinin daha zorlaşmasına sebep olacaktır.



Resim 9.2

Java dünyasında AspectJ popüler bir AOP çatısıdır. AspectJ aspektlerin bytecode seviyesinde iş mantığını taşıyan kod ile harmanlanmasını sağlamaktadır. Spring AOP AspectJ ve dinamik vekil (dynamic proxy) tasarım şablonu yardımı ile Spring uygulamalarında AOP teknolojisinin kullanımını mümkün kılmaktadır.

AOP sınıf ve metotların yapılandırılmasına getirdiği yeni bakış açışıyla OOP'yi (Object Oriented Programming) tamamlayıcı niteliktedir. OOP'nin modüler sistemler oluşturmadaki ana konsepti sınıf iken AOP'de bu jenerik fonksiyonları modüller haline getiren aspektlerdir.

AOP Konseptleri

AOP kendine has bir terminolojiye sahiptir. AOP ile ilgili kod örneklerini incelemeden önce bu terminolojiye göz atmamızda fayda vardır.

AOP kullanırken sıkça karşılaşacağımız terimler şunlardır:

- ***Advice*** - Aspekt olarak tanımladığımız jenerik fonksiyonlara verilen isimdir. Join point olarak tanımlanan yerlerde koşturulurlar. Kısaca advice'i aspektin yaptığı iş olarak tanımlayabiliriz. Advice yaptığı işin ne olduğunu bildiği gibi, ne zaman bu işi yapması gerektiğini de bilir, örneğin metod öncesi ya da metot sonrası gibi. Ne zaman sorusu join point ile tanımlanır. Değişik tipleri mevcuttur. Örneğin before advice join point öncesi, after advice join point sonrası koşturulur.
- ***Join Point*** - Program akışında bir metodun koşturulmak için gelindiği ya da bir değişkene değer atandığı bir yerdir. Buralarda advice'ın ihtiva ettiği kod koşturulur.
- ***Pointcut*** - Bir veya birden fazla join point seçiminde kullanılan ifadedir. Advice'in nerede/nerelerde koşturulacağını tanımlar.
- ***Aspect*** - Bir advice ve bir pointcut'in birleşimidir. Hangi jenerik fonksiyonun program akışının neresinde koşturulacağını tanımlar.
- ***Weaving*** - Tanımlanan aspektlerin derleme (compile) ya da uygulamanın çalışması esnasında (runtime) iş mantığını taşıyan kod ile harmanlamasına verilen isimdir.
- ***Target*** - AOP müdahelesi ile çalışma ve akış tarzı değişikliğe uğrayan nesneye verilen isimdir. Sıkça bu tür nesnelere advice verilen (adviced objects) nesneler denir.

- ***Introduction*** - Sınıfların yeni metot ya da sınıf değişkeni eklenerek yapısal değişikliğe uğradığı işleme verilen isimdir.
- ***AOP Proxy*** - AOP çatısı tarafından oluşturulan ve bir aspekte vekillik eden nesnedir. Spring çatısında bir AOP proxy JDK dynamic proxy ya da CGLIB kullanılarak oluşturulur.

İlk bakışta bu terimler kafa karıştırıcı olabilir. Şimdilik bu terimlerin kafanızı karıştırmasına izin vermeyin. Diğer bölümlerde inceleyeceğimiz AOP örnekleri bu terimlerin anlaşılmasını kolaylaştıracaktır.

AOP Harmanlama (Weaving) Türleri

Üç değişik harmanlama yöntemi mevcuttur:

- ***Derleme anında (at compile time)*** - Aspektler target olarak isimlendirilen sınıflarla derleme (compile) zamanında harmanlanır. Bunun için bir AOP derleyicisine (compiler) ihtiyaç duyulmaktadır. AspectJ bünyesinde böyle bir derleyici mevcuttur.
- ***Sınıf yükleme anında (at classload time)*** - JVM (Java Virtual Machine) bünyesinde sınıf yüklenmeden önce özel bir AOP classloader tarafından aspekt ve sınıf bytecode seviyesinde harmanlanır. Bu işleme load-time weaving - LTW (yükleme esnasında harmanlama) ismi verilmektedir. AspectJ 5 sürümü ile LTW'yi desteklemektedir.
- ***Uygulamanın çalışması esnasında (at runtime)*** - Uygulama çalışır haldeyken aspektler Java sınıfları ile harmanlanır. Bu harmanlama bytecode seviyesinde gerçekleşmez. Daha ziyade target nesnelere vekillik edecek vekil (proxy) nesneler oluşturulur. Bu vekil nesneler target nesne üzerinde yapılacak tüm işlemleri karşılayarak, target nesneye kanalize edilmelerini sağlarlar.

AOP Türleri

Statik ve dinamik olmak üzere iki AOP türü bulunmaktadır. İki tür arasındaki fark aspektlerin ne zaman iş mantığını taşıyan kod birimleri ile harmanlandığıdır.

Statik AOP türünde uygulama derlenirken aspektler mevcut kod ile doğrudan harmanlanır. Bu harmanlama bytecode seviyesinde gerçekleşir. Harmanlama öncesinde tüm Java sınıfları derlenir. Akabinde AOP çatısı derlenen bu sınıflara

aspekt kodlarını ekler. Sonuç aspektlerle harmanlanmış bytecode'dur. Herhangi bir değişiklik gerekli olduğunda kodun tekrar derlenmesi mecburidir. AspectJ çatısı sahip olduğu AOP derleyicisi ile (compiler) static AOP'yi temsil eder.

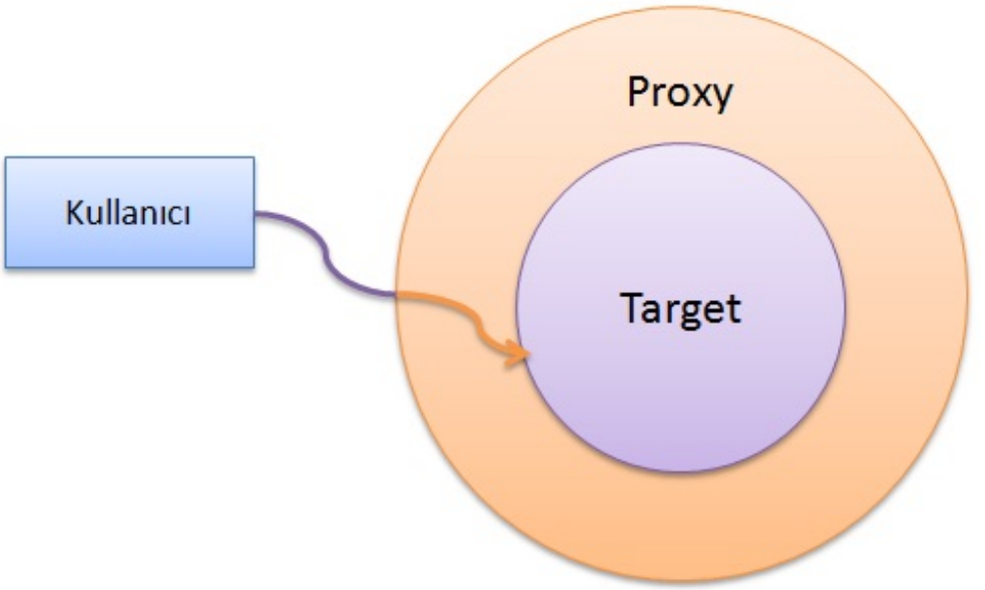
Dinamik AOP'de harmanlama işlemi uygulamanın çalışması esnasında (runtime) dinamik olarak gerçekleşir. Spring AOP bu AOP türünü kullanmaktadır.

Spring AOP

Spring AOP AspectJ çatısından bağımsız Java dilinde implemente edilmiş proxy tabanlı bir AOP çatısıdır. Spring proxy ve dekoratör tasarım şablonlarını kullanarak (resim 9.3) aspektlerin program akışının öngörülen yerlerinde koşturulmalarını sağlar. Bu öngörülen yerler target olarak isimlendirilen ve proxy nesnesi tarafından vakalet edilen nesnenin metotlarıdır.

Spring AOP'nin kullanımı AspectJ çatısına nazaran daha kolaydır, çünkü Spring AOP ile aspektlerin uygulamayı oluşturan Java sınıfları ile derlenmesi gerekli değildir. Spring dinamik AOP türünü kullandığı için sadece metot bazında pointcut'lar oluşturulabilir. Spring AOP ile konstrüktör bazında pointcut oluşturmak mümkün olmadığı için bir nesne oluştururken nesne metotlarını bir aspekt ile harmanlamak mümkün değildir. Değişken (field) ya da konstrüktör bazında pointcut oluşturma gerekliliği var ise, AspectJ gibi daha geniş AOP yelpazesine sahip bir çatı kullanılmalıdır.

Spring uygulamalarında AOP deklaratif ya da anotasyon bazlı konfigüre edilebilir. Spring AOP ile POJO sınıflardan XML konfigürasyon dosyasında deklaratif aspektler oluşturulabilir. Bunun yanı sıra Spring AOP AspectJ çatışında yer alan @Aspect anotasyonunu kullanan sınıfları aspekt olarak iş mantığı ile harmanlayabilir.



Resim 9.3

Advice Türleri

Advice'i koşturmak istediğimiz loglama, transaksiyon yönetimi ya da caching gibi jenerik fonksiyon olarak tanımlamıştık. Advice aspekt ile yapılan işlemdir. Advice yapılması gereken işlemi ve bu işlemin nerede yapılması gerektiğini bilir. Spring AOP beş değişik advice tipi tanımaktadır. Bunlar:

- ***before advice*** - Metoda girmeden koşturulan advice tipidir. Örneğin bir metodu veri tabanı transaksiyona dahil etmek için transaksiyon yönetimi advice'i before advice olarak tanımlanabilir. Bu advice koşturulduğunda metoda girilmeden önce bir veri tabanı transaksiyonu başlatılır ya da mevcut bir transaksiyona dahil olunur.
- ***after advice*** - Geri verdiği değer ne olursa olsun bir metot son bulduktan sonra koşturulan advice tipidir.
- ***after returning advice*** - Metot son bulduktan ve bir değer geri verdikten sonra koşturulan advice tipidir.
- ***after throwing advice*** - Bir metot bünyesinde hata oluştuğu zaman koşturulan advice tipidir.
- ***around advice*** - Bir metodun etrafında koşturulan advice tipidir ve yukarıda yer alan diğer advice tiplerinin kombinasyonudur.

Around Advice

Kod 9.1 de yer alan rentACar() metodunda, bu metodun hangi zaman biriminde

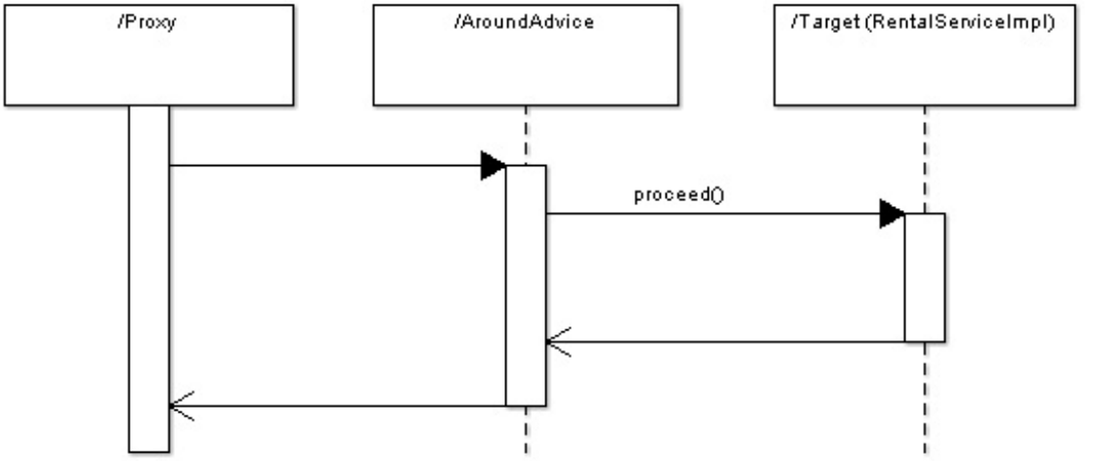
koşturulduğunu ölçen kod satırları yer almaktadır. Bu klasik bir aspekt örneğidir. Performans ölçümünü yapan kod birimini nasıl bir aspekt olarak tanımlayabilirdik? Bu sorunun cevabı kod 9.3 de yer almaktadır.

```
Kod 9.3 - PerformanceLoggerAspect

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class PerformanceLoggerAspect {

    @Around("execution(* com.kurumsaljava.spring.
            RentalServiceImpl.rentACar(..))")
    public Object profile(ProceedingJoinPoint pjp)
                throws Throwable {
        long start = System.currentTimeMillis();
        System.out.println("Metot öncesi: " + pjp.getSignature());
        Object output = pjp.proceed();
        System.out.println("Metot sonrasi");
        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Metot kosturma zamani: "
             + elapsedTime    + " milliseconds.");
        return output;
    }
}
```

Spring AOP'de hem POJO hem de @Aspect anotasyonunu taşıyan sınıflar aspekt olarak tanımlanabilir. POJO olan sınıfların aspekt olarak nasıl tanımlandığını bir sonraki örnekte inceleyeceğiz.

@Aspect AspectJ çatısında yer alan bir anotasyondur. Spring bu anotasyonu taşıyan bir sınıfa denk geldiği zaman, bu sınıfı bir aspekt olarak değerlendirip, sınıf içinde yer alan kodun iş mantığı ile harmanlanmasını sağlar. Harmanlamanın nasıl yapılacağını @Around anotasyonu belirler. Kod 9.3 de yer alan sınıfın bir aspekte dönüştürülebilmesi için Spring tarafından bir Spring bean olarak keşfedilmesi gerekmektedir. Bu amaçla @Component anotasyonunu kullandık. @Component kullanmak istediğimiz zaman Spring konfigürasyon dosyasında context:component-scan elementinin yer alması gerekir. Bu şekilde classpath içinde yer alan tüm sınıflar taranır ve Spring

anotasyonlarını taşıyan sınıflar keşfedilir.



Resim 9.4

Bir around advice oluşturmak için PerformanceLoggerAspect sınıfında @Around anotasyonunu kullandık. profile() metodu RentalServiceImpl sınıfında bulunan rentAcar() isimli metodu koşturabilmek için parametre olarak bir ProceedingJoinPoint nesnesi almaktadır. pjp.proceed() ile rentAcar() metodu koşturulmuş olur. profile() isimli metodu rentACar() metoduna girmeden önce start isimli lokal değişkede zamanı tutup, rentACar() metodu son bulduğunda geçen zamanı elapsedTime değişkenine atamaktadır. Bu bir metodun etrafında oluşturulan tipik bir around advicedır. Program akışı resim 9.4 de yer almaktadır.

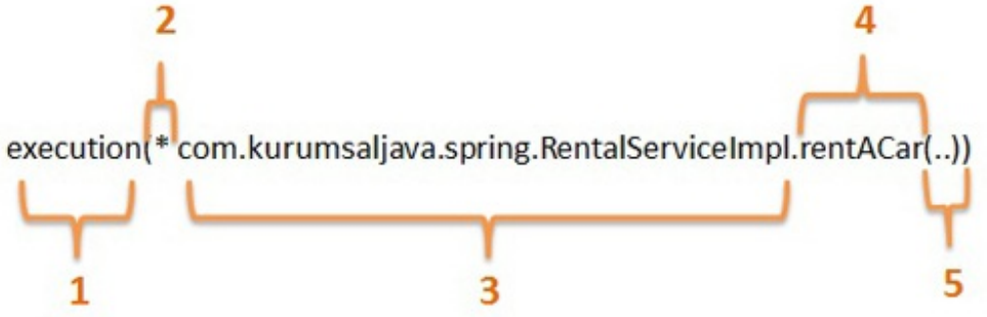
Kod 9.3 de oluşturduğumuz aspekt sınıfını Spring tarafından kullanılabilir hale getirmek için Spring konfigürasyon dosyasına aop:aspectj-autoproxy elementini eklememiz gerekmektedir. Bu direktif Spring'in aspektler ile iş mantığının yer aldığı sınıfları (target obejects) harmanlamak için proxy nesneleri oluşturmasını sağlamaktadır. Ayrıca aop:aspectj-autoproxy AspectJ anotasyonlarının kullanımını mümkün kılmaktadır. Nitekim resim 9.5 e bir göz attığımızda, alttan ikinci satırda RentalServiceImpl sınıfı için $Proxy24 isminde bir dinamik proxy nesnesinin oluşturulduğunu görmekteyiz.

```
Thread [main] (Suspended (breakpoint at line 18 in RentalServiceImpl))
    RentalServiceImpl.rentACar(Customer, long, Date, Date) line: 18
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 39
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 25
    Method.invoke(Object, Object...) line: 597
    AopUtils.invokeJoinpointUsingReflection(Object, Method, Object[]) line: 319
    ReflectiveMethodInvocation.invokeJoinpoint() line: 183
    ReflectiveMethodInvocation.proceed() line: 150
    MethodInvocationProceedingJoinPoint.proceed() line: 80
    PerformanceLoggerAspect.profile(ProceedingJoinPoint) line: 16
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 39
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 25
    Method.invoke(Object, Object...) line: 597
    AspectJAroundAdvice(AbstractAspectJAdvice).invokeAdviceMethodWithGivenArgs(Object[]) line: 621
    AspectJAroundAdvice(AbstractAspectJAdvice).invokeAdviceMethod(JoinPoint, JoinPointMatch, Object,
    AspectJAroundAdvice.invoke(MethodInvocation) line: 65
    ReflectiveMethodInvocation.proceed() line: 172
    ExposeInvocationInterceptor.invoke(MethodInvocation) line: 90
    ReflectiveMethodInvocation.proceed() line: 172
    JdkDynamicAopProxy.invoke(Object, Method, Object[]) line: 202
    $Proxy24.rentACar(Customer, long, Date, Date) line: not available
    Main.main(String[]) line: 13
```

Resim 9.5

Pointcut Tanımlamaları

@Around anotasyonunda kullandığımız ifade execution(* com.kurumsaljava.spring.RentalServiceImpl.rentACar(..)) bir pointcut tanımlamasıdır. Pointcut ile advice'in (profile() metodu) nerede koşturulacağını tayin etmekteyiz. Spring AOP'de pointcut'lar AspectJ pointcut expression language kullanılarak oluşturulur. http://eclipse.org/aspectj/ adresinden dil referansına ulaşabilirsiniz.



Resim 9.6

Resim 9.6 da oluşturduğumuz pointcut ifadesinin hangi parçalardan oluştuğu

yer almaktadır. Bunlar:

1. Metot koşturma işlemini tanımlayan bir join point'i seçmek için kullanılır.
2. İş mantığını ihtiva eden metodun (kullandığımız örnekte rentACar() metodu) geri verdiği değer. * (yıldız) geri verilen değerin önem taşımadığını ve göz ardı edildiğini ifade etmektedir.
3. İş mantığını ihtiva eden metodun bulunduğu sınıf.
4. İş mantığını ihtiva eden metot.
5. Bu metot için kullanılan parametreler. İki nokta sıfır ya da daha fazla parametre anlamına gelmektedir. Bu şekilde rentACar() ismini taşıyan tüm metotlar seçilmektedir.

Oluşturduğumuz bu pointcut ile PerformanceLoggerAspect isimli aspekt sınıfının sahip olduğu advice'i (profile() metodu) rentACar() metodunun koşturulduğu her yerde devreye girecek hale getirdik. execution() terimine AOP terminolojisinde designatör ismi verilmektedir. Spring AOP tarafından desteklenen designatörler şunlardır:

- ***execution*** - Metot koşturma işlemini tanımlayan bir join point'i seçmek için kullanılır. Spring sadece metot koşturma join point türünü desteklediği için Spring AOP ile kullanılan en sık designatördür.
- ***within*** - Joint point seçiminin verilen paketler/sınıflar arasından yapılmasını sağlar.
- ***@within*** - Join point seçiminin verilen anotasyonla işaretli sınıflar arasından yapılmasını sağlar.
- ***target*** - Join point seçiminin target nesnesinin sahip olduğu sınıftan yapılmasını sağlar.
- ***@target*** - Join point seçiminin verilen anotasyona sahip target nesneleri arasından yapılmasını saglar.
- ***this*** - Join point seçiminin this ile verilen nesne referansı sınıfından yapılmasını sağlar.
- ***args*** - Join point seçiminin verilen parametre tipine sahip metotlar arasından yapılmasını sağlar.
- ***@args*** - Join point seçiminin verilen anotasyonlarla işaretli parametrelere sahip metotlar arasından yapılmasını sağlar.
- ***@annotation*** - Join point seçiminin verilen anotasyonu taşıyan metotlar arasından yapılmasını sağlar.
- ***bean*** - Join point seçiminin Spring bean olarak tanımlanmış nesneler üzerinde yapılmasını sağlar.

Şimdi AspectJ pointcut expression language yardımı ile birkaç pointcut tanımlamasını yakından inceleyelim.

```
1. execution(void rentACar*(Rental))
```

Geri verdiği değer void olan, rentACar ile başlayan ve Rental tipinde bir parametreye sahip tüm metotların seçilmesini sağlar.

```
2. execution(* rentACar(*))
```

Herhangi bir değeri geri veren, tek parametreli ve rentACar ismini taşıyan bir metodun seçilmesini sağlar.

```
3. execution(* rentACar(Car, ..))
```

İlk parametresi Car tipinde olan (.. bu parametreden sonra 0 veya daha fazla parametre geldiğini gösterir) ve rentAcar ismini taşıyan herhangi bir metodun seçilmesini sağlar.

```
4. execution(void RentalServiceImpl.*(..))
```

RentalServiceImpl sınıfında yer alan tüm public void metotların seçilmesini sağlar.

```
5. execution(void RentalServiceImpl+.rentACar(*))
```

RentalServiceImpl veya altsınıflarında yer alan, public void şeklinde tanımlanmış, rentACar ismini taşıyan ve tek parametreli olan tüm metotların seçilmesini sağlar.

```
6. execution(@javax.annotation.security.RolesAllowed void
                rentACar*(..))
```

@RolesAllowed anotasyonunu taşıyan ve ismi rentACar ile başlayan herhangi bir void metodun seçilmesini sağlar.

```
7. execution(* kurumsaljava.*.rental.*.*(..))
```

Bu pointcut tanımlamasında kurumsaljava ve rental paketleri arasında herhangi bir paket olabilir ifadesi kullanılmaktadır.

```
8. execution(* kurumsaljava..rental.*.*(..))
```

Bu pointcut tanımlamasında kurumsaljava ve rental paketleri arasında birden fazla paket olabilir ifadesi kullanılmaktadır.

```
9. execution(* *..rental.*.*(..))
```

Bu pointcut tanımlamasında rental ismini taşıyan herhangi bir alt paket ifadesi kullanılmaktadır.

```
10. within(com.kurumsaljava.service.*)
```

Join point seçiminin com.kurumsaljava.service paketinde yer alan sınıflardan yapılmasını sağlar.

```
11. within(com.kurumsaljava.service..*)
```

Join point seçiminin com.kurumsaljava.service ve alt paketlerinde yer alan sınıflardan yapılmasını sağlar.

```
12. this(com.kurumsaljava.spring.RentalService)
```

RentalService sınıfını implemente eden target nesnenin değil, RentalService sınıfını implemente eden ve target nesnesinin vekili olan proxy nesnesinin metotlarını seçer. Bu genelde mevcut bir sınıfa kodu değiştirmeden proxy aracılığı ile yeni metot ve değişkenler eklemek için kullanılan bir yöntemdir.

```
13. target(com.kurumsaljava.spring.RentalService)
```

RentalService sınıfını implemente eden target nesnesinin sahip olduğu metotların seçilmesini sağlar.

```
14. args(Customer)
```

Tek parametreli ve Customer tipinde bir parametresi olan metotları seçmek için kullanılır. args() iş mantığını barındıran metodun sahip olduğu parametrelerin advice metodu bünyesinde kullanılmasını mümkün kılar (bknz. kod 9.17)

```
15. @target(org.springframework.transaction.annotation.Transactional)
```

@Transactional anotasyonuna sahip target nesnelerin sahip oldukları metotların seçilmesini sağlar.

```
16. @annotation(org.springframework.transaction.annotation.
         Transactional)
```

@Transactional anotasyonunu taşıyan metotların seçilmesini sağlar.

```
17. @args(com.kurumsaljava.annotation.Order)
```

Tek ve @Order tipinde bir parametresi olan metotların seçilmesini sağlar.

```
18. bean(rentalService)
```

Spring bean olarak tanımlı rentalService isimli nesnenin sahip olduğu metotların seçilmesini sağlar.

```
19. bean(*Service)
```

Spring bean olarak tanımlı ve isminde Service kelimesi geçen nesnelerin sahip oldukları metotların seçilmesini sağlar.

Pointcut tanımlamaları yapmak için ayrıca @Pointcut anotasyonu kullanılabilir. Kod 9.4 de myPointcut ismini taşıyan bir pointcut tanımlaması yapılmıştır. Bu tarz pointcut tanımlamaya named pointcut (ismi olan pointcut) ismi verilmektedir. @Pointcut anotasyonu kullanıldığı taktirde, bu pointcut'ın ismini tanımlamak için void tipinde private bir metot tanımlanır. Bu metodun ismi pointcut'ın ismidir. Bu isim daha sonra @Around anotasyonunda kullanarak, pointcut aktif hale getirilebilir.

```
Kod 9.4 - PerformanceLoggerAspect

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class PerformanceLoggerAspect2 {

    @Pointcut("execution(* com.kurumsaljava.spring.
            RentalServiceImpl.rentACar(..))")
    private void myPointcut() {}
```

```
    @Pointcut("execution(* com.kurumsaljava.spring.
                RentalServiceImpl.updateRental(..))")
    private void yourPointcut() {}


    @Around("myPointcut() || yourPointcut()")
    public Object profile(ProceedingJoinPoint pjp)
            throws Throwable {
        long start = System.currentTimeMillis();
        System.out.println("Metot öncesi: " + pjp.getSignature());
        Object output = pjp.proceed();
        System.out.println("Metot sonrasi");
        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Metot kosturma zamani: " + elapsedTime
                + " milliseconds.");
        return output;
    }
}
```

Named pointcut'lar &&', '||' ve '!' işaretleri kullanılarak kombine edilebilir. Kod 9.4 de yer alan örnekte @Around anotasyonunda iki pointcut || işareti ile kombine edilmektedir. || işareti burada veya (or) anlamına gelmektedir. Bu tür bir pointcut kombinasyonu aspektin rentACar() ya da updateRental() metotları koşturulduğunda devreye girmesini sağlayacaktır.

Kod 9.4 de görüldüğü gibi named pointcut oluşturulması komplike pointcut tanımlamalarının parçalara bölünmesini mümkün kılmaktadır. Bu şekilde yeni bir pointcut named pointcut tanımlamaları bir araya getirilerek oluşturulabilmektedir. Bunun yanı sıra named pointcut'lar pointcut'ların tekrar tanımlanmalarına gerek kalmadan tekrar kullanılmalarını mümkün kılmaktadır.

Named pointcut'ların tekrar kullanımını kolaylaştırmak için, tüm named pointcut'ları ihtiva eden bir sınıf oluşturulabilir. Bunun bir örneği kod 9.4.1 de yer almaktadır.

```
Kod 9.4.1 - SystemPointcuts

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemPointcuts {
```

```
    @Pointcut("within(com.kurumsaljava.spring.web..*)")
    private void webLayer() {
    }

    @Pointcut("within(com.kurumsaljava.spring.service..*)")
    private void serviceLayer() {
    }

    @Pointcut("within(com.kurumsaljava.spring.repository..*)")
    private void daoLayer() {
    }
}
```

Kod 9.4.2'de görüldüğü gibi SystemPointcuts sınıfında tanımlı olan herhangi bir named pointcut tanımlamasını paket+sınıf+pointcut-ismi şeklinde kullanabiliriz.

```
Kod 9.4.2 - PerformanceLoggerAspect2

@Around("com.kurumsaljava.spring.aspects.
        SystemPointcuts.serviceLayer()")
public Object profile(ProceedingJoinPoint pjp) throws Throwable {
    long start = System.currentTimeMillis();
    System.out.println("Metot öncesi: " + pjp.getSignature());
    Object output = pjp.proceed();
    System.out.println("Metot sonrasi");
    long elapsedTime = System.currentTimeMillis() - start;
    System.out.println("Metot kosturma zamani: " + elapsedTime
            + " milliseconds.");
    return output;
}
```

Around Advice İçin XML Konfigürasyonu

Kod 9.3 de yer alan örnekte @Aspect ve @Around anotasyonlarını kullanarak performans ölçümü yapan bir aspekt oluşturmuştuk. Bu bölümde anotasyon kullanmadan bir around advice'ın nasıl oluşturulacağını gösteren bir örnek sunmak istiyorum. Kod 9.5 de aspekt kodunu ihtiva eden PerformanceLoggerAspect3 sınıfı yer almaktadır. Bu bir POJO'dur (Plain Old Java Object). Bu sınıfta yer alan profile() metodunun bir around advice olabilmesi için bu aspektin Spring konfigürasyon dosyasında tanımlanması gerekmektedir. Böyle bir tanımlama kod 9.6 da yer almaktadır.

```
Kod 9.5 - PerformanceLoggerAspect3

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.ProceedingJoinPoint;

public class PerformanceLoggerAspect3 {

    public Object profile(ProceedingJoinPoint pjp)
        throws Throwable {
        long start = System.currentTimeMillis();
        System.out.println("Metot öncesi: " + pjp.getSignature());
        Object output = pjp.proceed();
        System.out.println("Metot sonrasi");
        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Metot kosturma zamani: " + elapsedTime
                + " milliseconds.");
        return output;
    }
}


Kod 9.6 - applicationContext-aop.xml

<aop:aspectj-autoproxy />

<aop:config>
    <aop:aspect ref="performanceAspect">
        <aop:around
            pointcut="execution(*
        com.kurumsaljava.spring.RentalServiceImpl.rentACar(..))"
            method="profile" />
    </aop:aspect>
</aop:config>

<bean id="performanceAspect"
    class="com.kurumsaljava.spring.aspects.
                PerformanceLoggerAspect3" />
```

Aop:aspectj-autoproxy elemeni ile daha önce tanışmıştık. Bu element yardımı ile Spring AOP hedef nesneler için proxy nesneler oluşturur. Aop:config elementi anotasyonlar aracılığı ile yaptığımız aspekt konfigürasyonu Spring XML konfigürasyon dosyasında yapmak için kullanılır. Bu elementin kullanılabilmesi için aop isim alanının konfigürasyon dosyasına yüklenmesi gerekmektedir.

Aop:aspect elemeni kullanmak istediğimiz aspekt sınıfını tanımlamaktadır. Bu sınıf kod 9.5'de yer almaktadır. Kod 9.6 da yer alan konfigürasyon örneğinde

performanceAspect isminde bir aspekt bean tanımlaması yaparak, bunu aop:aspect elementinde ref (referans) olarak kullandık.

Bir around advice tanımlamak için aop:around elementi kullanılmaktadır. Bu elementin pointcut ismini taşıyan ve pointcut tanımlamak için kullanılan bir element özelliği mevcuttur. Bu element özelliği yardımı ile rentACar() metodu koşturulduğunda devreye girecek bir advice oluşturuyoruz. Burada yer alan pointcut tanımlaması kod 9.4 de yer alan @Pointcut anotasyonu ile aynı içeriğe sahiptir. Aop:around elementinin son element özelliği olan method ile aspekt sınıfında yer alan hangi metodun advice olarak koşturulması gerektiği tanımlanmaktadır.

Before Advice

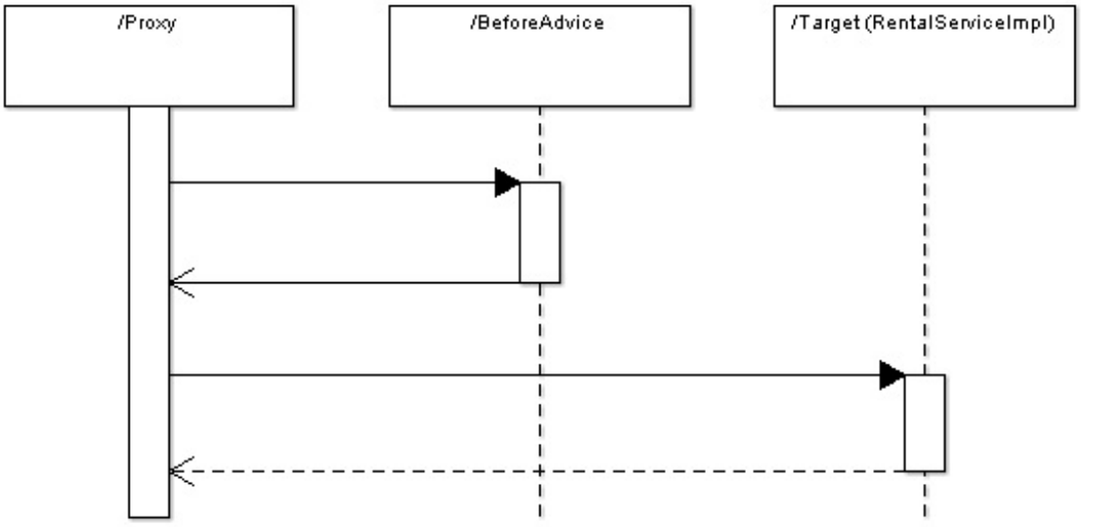
Before advice metot öncesi koşturulan advice tipidir. @Before anotasyonunu kullanarak oluşturduğumuz, @Pointcut ile tanımlanan metoda girilmeden >>Metot öncesi şeklinde bir log kaydı oluşturan aspekt kod 9.7 de yer almaktadır. BeforeMethodLoggerAspect ismini taşıyan bir aspekt kod 9.3 de yer alan aspekt ile aynı yapıdadır.

```
Kod 9.7 - BeforeMethodLoggerAspect.java

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class BeforeMethodLoggerAspect {

    @Pointcut("execution(* com.kurumsaljava.spring.
                RentalServiceImpl.rentACar(..))")
    private void beforeMethod() {}

    @Before("beforeMethod()")
    public void before(){
        System.out.println(">> Metot öncesi");
    }
}
```

Resim 9.7 de yer alan dizge akışı diyagramında program akışı yer almaktadır.

Spring ilk işlem olarak bir proxy nesne oluşturmaktadır. Bu proxy RentalServiceImpl sınıfına vekillik etmektedir. Dışarıdan bu sınıfın rentACar() isimli metodunu koşturmaya yönelik gelen her istek önce proxy nesneye yönlendirilmektedir. Bu proxy RentalServiceImpl sınıfıyla harmanlanan BeforeMethodLoggerAspect aspekt sınıfını tanıdığı için istediği BeforeMethodLoggerAspect sınıfının before() metoduna yöneltmektedir. before() metodu son bulduktan sonra program akışı RentalServiceImpl.rentACar() metoduyla devam etmektedir.

Eğer advice, yani before() metodu içinde bir hata oluşursa, target nesnesinin seçilen metodu koşturumaz.



Resim 9.7

Aynı aspekti @Aspect ve @Before anotasyonları olmadan Spring konfigürasyon dosyasında kod 9.8 deki gibi konfigüre edebiliriz.

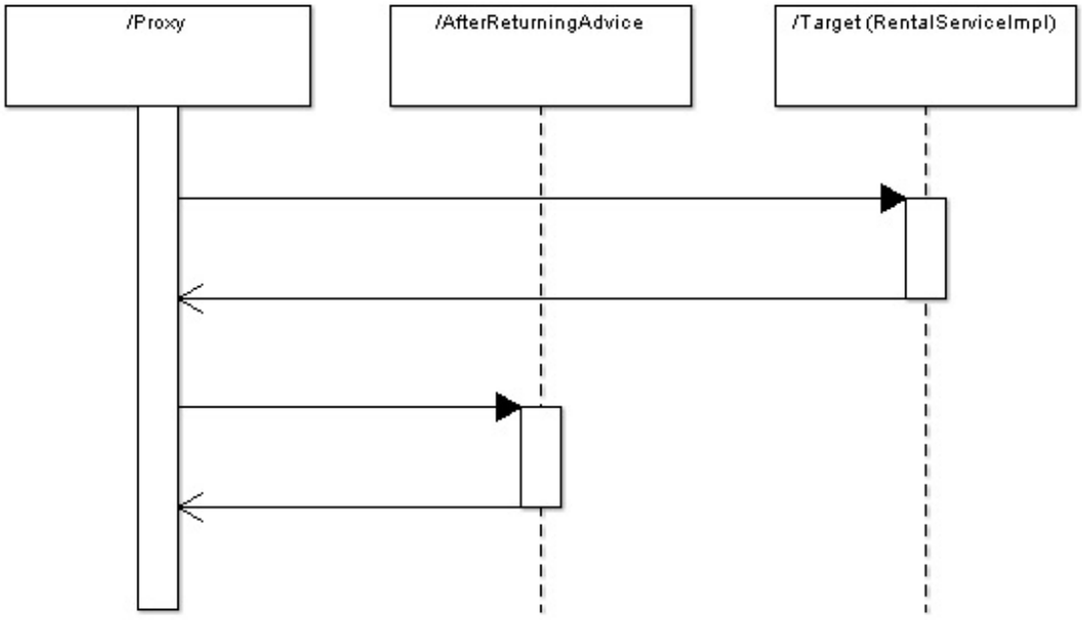
```
Kod 9.8 - applicationContext-aop.xml

<aop:config>
    <aop:aspect ref="beforeMethodLogger">
        <aop:before
            pointcut="execution(*
            com.kurumsaljava.spring.RentalServiceImpl.rentACar(..))"
            method="before" />
    </aop:aspect>
</aop:config>
```

After Returning Advice

After returning advice seçilen metot son bulduktan ve bir değer geri verdikten sonra koşturulan advice tipidir. Bu advice tipinde metodun hata oluşmadan bir değer geri vermesi gerekmektedir. Kod 9.9 da yer alan @AfterReturning anotasyonu kullanılarak oluşturduğumuz RentalLoggerAspect isimli aspekt sınıfı araç kiralama işlemi son bulduktan sonra rezervasyon numarasını ihtiva eden bir log kaydı oluşturmaktadır.

```
Kod 9.9 - RentalLoggerAspect

package com.kurumsaljava.spring.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

import com.kurumsaljava.spring.Rental;

@Aspect
@Component
public class RentalLoggerAspect {

    @AfterReturning(value = "execution(*
            com.kurumsaljava.spring.RentalServiceImpl.
                rentACar(..))",
            returning = "rental")
    public void
        after(JoinPoint jp, Rental rental) {
        System.out.println("Yapilan arac rezervasyon numarasi:"
                    + rental.getId());
    }
}
```

Resim 9.8 de yer alan dizge akış diyagramında program akışı yer almaktadır. Spring tarafından oluşturulan proxy rentACar() metodunu koşturduktan sonra RentalLoggerAspect aspektin after() metodunu koşturmaktadır. after() metoduna girilmeden önce rentACar() metodunun geri verdiği nesne (rental) tanındığı için advice içinde iken bu nesne üzerinde işlem yapmak mümkündür.



Resim 9.8

Aynı aspekti @Aspect ve @AfterReturning anotasyonları olmadan Spring konfigürasyon dosyasında kod 9.10 daki gibi konfigüre edebiliriz.

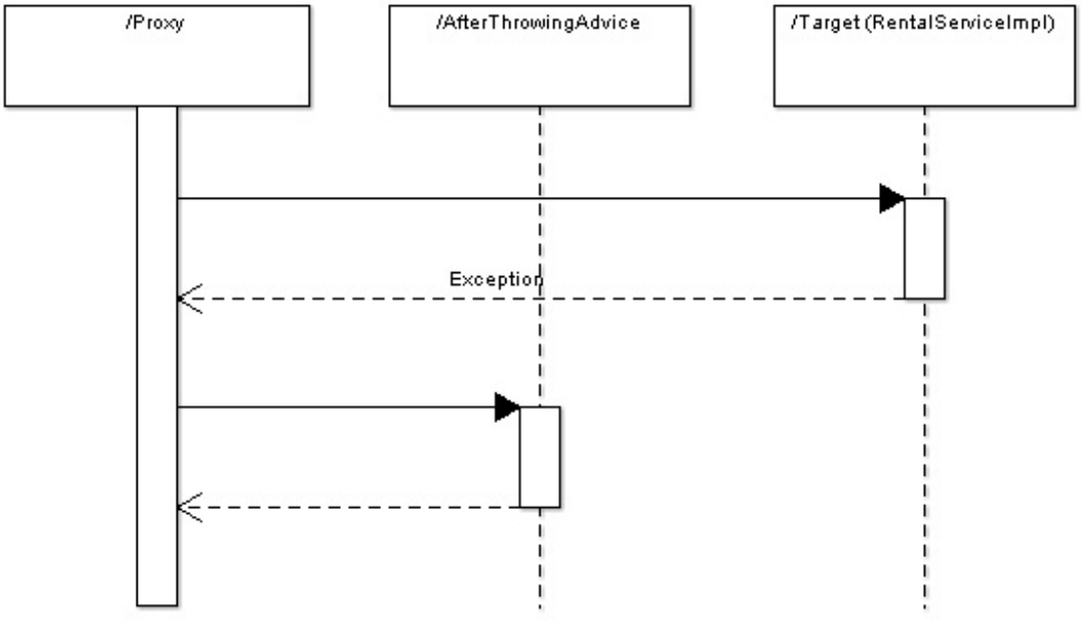
```
Kod 9.10 - applicationContext-aop.xml

<aop:config>
    <aop:aspect ref="beforeMethodLogger">
        <aop:before
            pointcut="execution(* com.kurumsaljava.spring.
                RentalServiceImpl.rentACar(..))"
            method="before" />
    </aop:aspect>
</aop:config>
```

After Throwing Advice

Bir metot bünyesinde hata oluştuğu zaman koşturulan advice tipidir. Kod 9.11 de yer alan aspekt rentACar() metodunda DataAccessException tipinde oluşan bir hatanın ardından devreye girecektir. Bu aspektin görevi oluşan hataları e-posta aracılığı ile bildirmektir. Program akışı resim 9.9 da yer almaktadır. rentACar() bünyesinde hata oluşmadığı sürece MailServiceAspect devreye girmeyecektir.

```
Kod 9.11 - MailServiceAspect

package com.kurumsaljava.spring.aspects;
```

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Component;
import com.kurumsaljava.spring.MailService;

@Aspect
@Component
public class MailServiceAspect {

    @AfterThrowing(value = "execution(* " +
            "com.kurumsaljava.spring.RentalServiceImpl.
                rentACar(..))", throwing = "e")
    public void report(JoinPoint jp, DataAccessException e) {
        MailService.emailFailure(jp, e);
    }
}
```



Resim 9.9

Bu advice tipinde oluşan hatanın üst katmanlara delege edilmesi engellenmez. Ama oluşan hata kod 9.12 de görüldüğü gibi başka bir hata tipine dönüştürülebilir. Hatanın üst katmanlara delege edilmesinin önüne geçmek için around advice tipi kullanılabilir.

```
Kod 9.12 - MailServiceAspect

@Aspect
@Component
```

```
public class MailServiceAspect {

    @AfterThrowing(value = "execution(* " +
            "com.kurumsaljava.spring.RentalServiceImpl.
                rentACar(..))", throwing = "e")
    public void report(JoinPoint jp, DataAccessException e) {
        MailService.emailFailure(jp, e);
        throw new RentalFailedException(e);
    }
}
```

Aynı aspekti @Aspect ve @AfterThrowing anotasyonları olmadan Spring konfigürasyon dosyasında kod 9.13 deki gibi konfigüre edebiliriz.

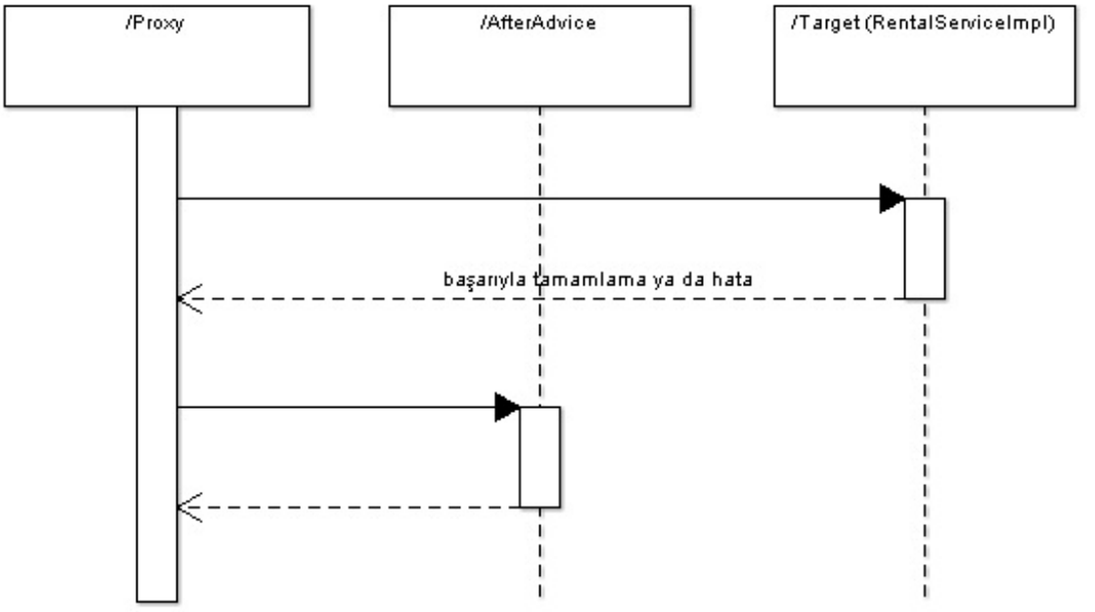
```
Kod 9.13 - applicationContext-aop.xml

<aop:config>
    <aop:aspect ref="mailServiceAspect">
        <aop:after-throwing
            pointcut="execution(* com.kurumsaljava.spring.
                    RentalServiceImpl.rentACar(..))"
            method="report" throwing="e" />
    </aop:aspect>
</aop:config>
```

After Advice

Geri verdiği değer ne olursa olsun, bu bir hata da olabilir, bir metot son bulduktan sonra koşturulan advice tipidir. Kod 9.14 de yer alan örnekte rentACar() metodundan her çıkış loglanmaktadır. Bu örneğin bu metodun kaç kez koşturulduğunu ölçmek için kullanılan bir indikatör olabilir. Resim 9.10 da program akışı yer almaktadır.

```
Kod 9.14 - TrackerAspect

@Aspect
@Component
public class TrackerAspect {

    private Logger logger = Logger.getLogger(getClass());
    @After("execution(* " +
            "com.kurumsaljava.spring.RentalServiceImpl.
            rentACar(..))")
    public void track() {
        logger.info("rentACar() metodu kosturuldu.");
```

```
    }
}
```



Resim 9.10

Aynı aspekti @Aspect ve @After anotasyonları olmadan Spring konfigürasyon dosyasında kod 9.15 deki gibi konfigüre edebiliriz.

```
Kod 9.15 - applicationContext-aop.xml

<aop:config>
    <aop:aspect ref="trackerAspect">
        <aop:after
            pointcut="execution(* com.kurumsaljava.spring.
            RentalServiceImpl.rentACar(..))"
            method="track" />
    </aop:aspect>
</aop:config>
```

XML İle Named Pointcut tanımlaması

Daha önceki bir bölümde @Pointcut anotasyonunu kullanarak named pointcut oluşturma işlemini incelemiştik. Named poincut'lar tekrar kullanılabilir yapıda olan pointcut tanımlamalarıdır. Kod 9.13, 9.15 gibi örneklerde Spring XML dosyasında pointcut tanımlama işlemini inceledik. Bu örneklerin hepsinde execution(* com.kurumsaljava.spring.RentalServiceImpl.rentACar(..)) pointcut ifadesini kullandık, yani bu ifadeyi çoğalttık, çünkü kullandığımız bir named

pointcut tanımlaması değildi. @Pointcut anotasyonunda olduğu gibi XML bazlı aspekt konfigürasyonunda da named pointcut tanımlaması yapmak ve bu tanımlamayı tekrar kullanmak mümkündür. Bunun nasıl yapıldığı kod 9.16'da yer almaktadır.

```
Kod 9.16 - applicationContext-aop.xml

<aop:config>
    <aop:pointcut id="rentAcar"
        expression="execution(* com.kurumsaljava.spring.
        RentalServiceImpl.rentACar(..))" />

    <aop:aspect ref="performanceAspect">
        <aop:around pointcut-ref="rentAcar" method="profile" />
    </aop:aspect>

    <aop:aspect ref="beforeMethodLogger">
        <aop:before pointcut-ref="rentAcar" method="before" />
    </aop:aspect>

    <aop:aspect ref="rentalLogger">
        <aop:after-returning pointcut-ref="rentAcar"
            method="after" returning="rental" />
    </aop:aspect>

    <aop:aspect ref="mailServiceAspect">
        <aop:after-throwing pointcut-ref="rentAcar"
            method="report" throwing="e" />
    </aop:aspect>
</aop:config>
```

XML dosyasında named pointcut tanımlaması aop:pointcut elementi kullanılarak yapılmaktadır. Id element özelliği bu pointcut'a verilen ismi taşımaktadır. Expression element özelliği ile pointcut ifadesi oluşturulur. Aop:aspect ile yapılan aspekt tanımlamalarında pointcut-ref element özelliği kullanılarak aop:pointcut ile oluşturulan bir named pointcut'i kullanmak mümkündür.

Aspekt Parametreleri

Aspekt bünyesinde yer alan advice metotlarında ilk metot parametresi olarak org.aspectj.lang.JoinPoint ve altsınıfları kullanılabilir. Around advice oluştururken ProceedingJoinPoint sınıfını ilk parametre olarak kullanmıştık. Bu

sınıf org.aspectj.lang.JoinPoint sınıfının bir altsınıfıdır. Diğer advice tiplerinde org.aspectj.lang.JoinPoint sınıfının ilk parametre olarak tanımlama zorunluluğu yokken, around advice tipinde advice metodunun ilk parametresinin ProceedingJoinPoint tipinde olması gerekmektedir. Sadece bu şekilde around advice ProceedingJoinPoint tipindeki ilk parametresi üzerinden iş mantığını ihtiva eden metodu koşturulabilmektedir.

org.aspectj.lang.JoinPoint bünyesinde örneğin iş mantığının yer aldığı metodun parametrelerini edinmek için kullanabileceğimiz getArgs() isminde bir metot mevcuttur. Bunun yanı sıra getTarget() ile iş mantığının yer aldığı metodun sahibi nesneyi elde edebiliriz. Bu metotların faydası çok ve kullanımları kolay iken, advice metodunda kullanılmaları kod karmaşasını artırabilir. Bunun yerine ihtiyaç duyduğumuz değerleri advice metoduna parametre olarak verebiliriz. Bunun bir örneği kod 9.17'de yer almaktadır.

```
Kod 9.17 - CustomerInspectorAspect

package com.kurumsaljava.spring.aspects;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;
import com.kurumsaljava.spring.Customer;

@Aspect
@Component
public class CustomerInspectorAspect {

    @Pointcut("execution(* com.kurumsaljava.spring.
            RentalServiceImpl.rentACar(..))")
    private void beforeMethod() {}

    @Before("beforeMethod() && args(customer,..)")
    public void before(Customer customer){
        System.out.println("Müsteri ismi: " +
                customer.getFirstname() + " " +
                customer.getName());
    }
}
```

Daha önce kod 9.7 de oluşturduğumuz @Before advice parametresizdi. Kod 9.17 de yer alan örnekte before() ismini taşıyan advice metodu Customer isminde bir parametreye sahiptir. Bu parametre RentalServiceImpl.rentACar() metodunda yer alan Customer parametresidir. rentACar() metoduna gönderilen Customer

tipindeki parametreyi yakalayarak CustomerInspectorAspect.before() metoduna enjekte edebilmek için oluşturduğumuz pointcut tanımlamasında args(customer,..) ifadesini kullanmamız gerekmektedir. @Before anotasyonu bünyesinde iki pointcut kombine edilerek rentACar() metodunda girmeden önce advice metodunu devreye sok, ayrıca rentACar() metoduna gönderilen Customer tipindeki parametreyi advice metodu olan before() metoduna parametre olarak geç ifadesi oluşturulmaktadır.

Args(customer,..) pointcut tanımlamasının iki görevi mevcuttur. İlk görevi advice metodu için Customer tipindeki parametreyi sağlamaktır. İkinci görevi ise en az bir parametreli ve ilk parametresi Customer tipinde olan bir rentACar() metodunu seçmektir. Kod 9.17 de tanımladığımız pointcut rentACar() ismini taşıyan ve herhangi sayıda parametreye sahip metotların seçimini mümkün kılmaktadır. args(customer,..) tanımlaması ile bu seçim daraltılarak sadece ilk parametresi Customer olan rentACar() metotların seçilmesi sağlanmaktadır. Aynı pointcut tanımlamasını kod 9.18 deki şekilde de yapabilirdik.

```
Kod 9.18 - CustomerInspectorAspect

@Pointcut("execution(* com.kurumsaljava.spring.RentalServiceImpl.
                            rentACar(..)) && args(customer,..)")
private void beforeMethod(Customer customer) {}


@Before("beforeMethod(customer)")
public void before(Customer customer){
    System.out.println("Müsteri ismi: " +
        customer.getFirstname() + " " + customer.getName());

}
```

Advice Sırası

Aynı join point'i (iş mantığının yer aldığı metot) koşturmak isteyen birden fazla aspektimizin olduğunu düşünelim. Hangi sıraya göre bu aspektler koşturulur? Öncelik sırası belirlenmediği sürece aspektlerin koşturulma sıralarında bir düzen yoktur. Sıra düzenini oluşturmak için Spring'in ihtiva ettiği org.springframework.core.annotation.Order anotasyonu kullanılabilir. @Order anotasyonuna sıra numarası olarak bir rakam atanabilmektedir. Spring aspekt listesini oluşturduktan sonra @Order anotasyonuna sahip aspektleri sahip

oldukları sıra numarasına göre öncelikli olarak koşturacaktır.

Kod 9.19 da üç değişik aspekt tanımlaması yer almaktadır. İlk iki aspekt bünyesinde @Order anotasyonu kullanılmıştır. Üçüncü aspekt bu anotasyona sahip değildir. Buna göre aspekt koşturma sıralaması ş şekilde olacaktır:

1. CustomerInspectorAspect
2. BeforeMethodLoggerAspect
3. MailServiceAspect

@Order.getValue() metodunun geri verdiği değer ne kadar küçük ise, aspektin koşturulma önceliği o kadar yüksek olacaktır.

```
Kod 9.19 - Aspekt Listesi

@Aspect
@Component
@Order(1)
public class CustomerInspectorAspect {

    @Pointcut("execution(* com.kurumsaljava.spring.
                    RentalServiceImpl.rentACar(..))")
    private void beforeMethod() {}

    @Before("beforeMethod() && args(customer,..)")
    public void before(Customer customer){
        System.out.println("Müsteri ismi: " +
            customer.getFirstname() + " " + customer.getName());
    }
}

@Aspect
@Component
@Order(2)
public class BeforeMethodLoggerAspect {

    @Pointcut("execution(* com.kurumsaljava.spring.
                    RentalServiceImpl.rentACar(..))")
    private void beforeMethod() {}

    @Before("beforeMethod()")
    public void before(){
        System.out.println(">> Metot öncesi");
    }
}

@Aspect
```

```
@Component
public class MailServiceAspect {

    @AfterThrowing(value = "execution(* " +
            "com.kurumsaljava.spring.RentalServiceImpl.
                            rentACar(..))", throwing = "e")
    public void report(JoinPoint jp, DataAccessException e) {
        MailService.emailFailure(jp, e);
    }

}
```

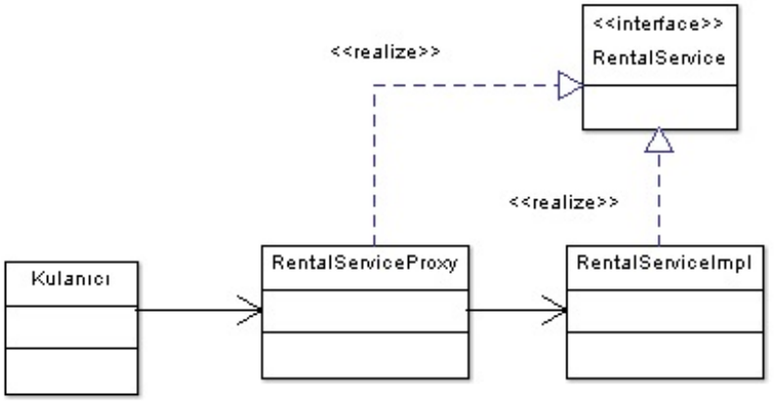
@Order ile yapılan sıralama advice tipine göre değişmektedir. @Order anotasyonuna sahip iki değişik before advice arasından en küçük order değerine sahip advice önce koşturulurken, after advice tipinde en küçük order değerine sahip advice en son sırada koşturulacaktır. Spring bu konuda AspectJ tarafından tanımlanan öncelik sırası kurallarını takip etmektedir.

Sınıfların Dinamik Olarak Genişletilmesi

Derlenen bir Java sınıfında sadece sınıfın ihtiva ettiği ve üstsınıflardan kalıtım yoluyla aldığı metotlar kullanılabilir. Derlenmiş bir Java sınıfına sonradan yeni bir metot eklemek için sınıfı değiştirip, yeniden derlemek ya da sınıfı bytecode seviyesinde manipüle etmek gerekir. Ruby ya da Groovy gibi dillerde mevcut bir sınıfa, sınıfın yapısını değiştirmeden dinamik olarak yeni bir metot eklemek mümkündür. AOP kullanarak Java sınıfları için de bu amaca ulaşılabilir.

Sınıflara dinamik olarak yeni metot ekleme işlemine AspectJ terminolojisinde inter-type declaration ve yeni bir metot kullanıma sunulduğu için sonradan eklenen bu metotlara introduction ismi verilmektedir. Spring AOP ile tanımlı Spring nesnelerine (Spring bean) dinamik olarak metot eklenebilir. Bunun nasıl yapıldığını incelemeden önce Spring AOP ile aspektlerin nasıl oluşturulduğunu tekrar hatırlayalım. Resim 9.11 de yer alan RentalServiceImpl sınıfı oluşturduğumuz aspektin hedef (target) sınıfıdır. Daha önceki örneklerde RentalServiceImpl sınıfında yer alan rentACar() metodunu join point olarak seçip, bu metot öncesi ya da sonrası aspekt aracılığı ile bir takım işlemler yapmıştık. Spring AOP aspektin görevini yerine getirebilmesi için uygulama çalışırken RentalServiceImpl sınıfından oluşturulan nesneye vekillik eden bir proxy nesnesi oluşturmaktadır. Bu proxy nesnesi RentalServiceImpl gibi RentalService interface sınıfını implemente ettiği için RentalServiceProxy RentalServiceImpl'in yerine geçerek, aspekt için gerekli işlemleri

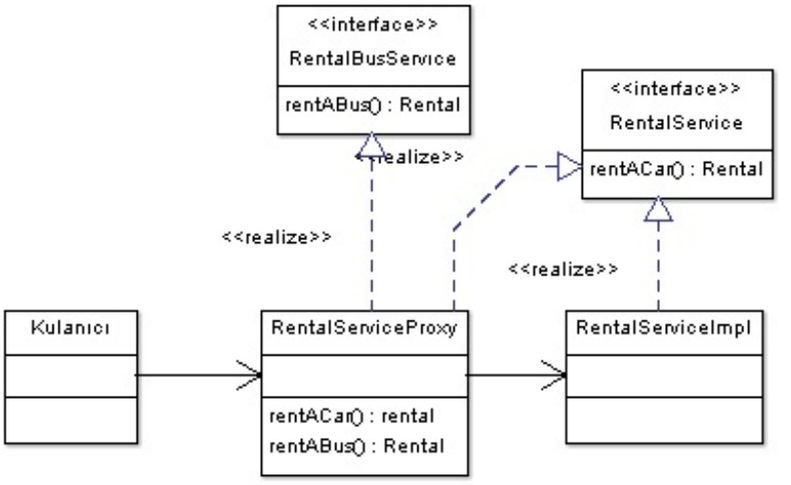
yapabilmektedir.



Resim 9.11

Bu durumda Spring tarafından bize kullanılmak üzere verilen nesne RentalServiceImpl değil, RentalServiceProxy nesnesidir. RentalServiceImpl sınıfında yer alan rentACar() metodunu koşturmak istediğimizde RentalServiceProxy.rentACar() şeklinde bir çağrı oluşturmuş oluruz. RentalServiceProxy hedef nesne olan RentalServiceImpl'i tanıdığı için bu metot çağrısını RentalServiceImpl bünyesinde yer alan rentACar() metoduna delege etmektedir. Bu işlemi yaparken de tanımladığımız advice tipine göre metot öncesi ya da sonrası aspektin sahip olduğu kodu koşturmaktadır.

RentalServiceImpl sınıfına dinamik olarak yeni bir metot eklemek için RentalServiceProxy sınıfının yeni bir interface sınıfını genişletmesi yeterli olacaktır. Resim 9.12 de yer alan sınıf diyagramında RentalServiceProxy BusRentalService sınıfını implemente etmektedir. RentalServiceProxy hem RentalService hem de BusRentalService sınıflarını implemente ettiği için rentACar() ve rentABus() metotlarına sahip olmuştur. rentABus() metodu RentalServiceImpl sınıfına dinamik olarak eklemek istediğimiz metottur.



Resim 9.12

RentalServiceImpl sınıfına dinamik olarak rentABus() metodunu kazandırmak için kod 9.20'de görüldüğü gibi ClassExtenderAspect isminde bir aspekt tanımlıyoruz. Bu aspekt bünyesinde yer alan @DeclareParents anotasyonu hangi sınıfa hangi metodun atanacağını tayin etmektedir. Kod 9.20'de yer alan örnekte @DeclareParents anotasyonu com.kurumsaljava.spring.RentalServiceImpl sınıfına BusRentalService bünyesinde yer alan metotları atamaktadır. Bu kısaca şu anlama gelmektedir:

```
com.kurumsaljava.spring.RentalServiceImpl
                    implements BusRentalService{
}
```

Bu RentalServiceProxy için de geçerlidir.

Kullanıcı rentABus() metodunu koşturmak istediğinde, RentalServiceProxy'nin bunu hangi nesne üzerinde yapacağını bilmesi gerekmektedir. Bu amaçla @DeclareParents anotasyonunda defaultImpl ile BusRentalService sınıfını implemente eden bir sınıf tanımlanmaktadır. Kod 9.20 de yer alan örnekte defaultImpl ClassExtenderAspect sınıfıdır, yani aspektin kendisidir. ClassExtenderAspect BusRentalService sınıfını implemente ettiği için defaultImpl olarak kullanılabilir. DefaultImpl olarak BusRentalService sınıfını implemente eden herhangi bir sınıf kullanılabilir. Bu aspektin kendisi olmak zorunda değildir.

```
Kod 9.20 - ClassExtenderAspect

package com.kurumsaljava.spring.aspects;
import java.util.Date;
import org.aspectj.lang.annotation.Aspect;
```

```
import org.aspectj.lang.annotation.DeclareParents;
import org.springframework.stereotype.Component;
import com.kurumsaljava.spring.Bus;
import com.kurumsaljava.spring.BusRentalService;
import com.kurumsaljava.spring.Customer;
import com.kurumsaljava.spring.Rental;

@Aspect
@Component
public class ClassExtenderAspect implements BusRentalService {


    @DeclareParents(value = "com.kurumsaljava.spring.
            RentalServiceImpl",    defaultImpl =
                    ClassExtenderAspect.class)
    public static BusRentalService mixin;

    @Override
    public Rental rentABus(Customer customer, Bus bus,
            Date begin, Date end) {
        System.out.println("rentABus() enter");
        return null;
    }

}
```

Kod 9.20 de yer alan aspekt bünyesinde herhangi bir advice tanımlamadık. ClassExtenderAspect sınıfı Spring'in bir proxy nesne oluşturup, bu proxy nesnesinin BusRentalService interface sınıfını implemente ederek, rentABus() metoduna cevap verecek hale gelmesini sağlayacaktır. Bu proxy nesne rentAcar() metoduna gelen istekleri RentalServiceImpl, rentABus() metoduna gelen istekleri ClassExtenderAspect sınıfına yönlendirecektir.

RentalServiceImpl sınıfını kullanmak için rentalService isminde bir Spring bean tanımlaması yapmıştık. Bu tanımlama kod 9.21 de yer almaktadır. Görüldüğü gibi Spring için rentalService nesnesinin veri tipi RentalServiceImpl sınıfıdır. Kod 9.20 de yer alan aspekt aktif hale geldiği zaman (RentalService)ctx.getBean("rentalService") ile edineceğimiz nesne RentalServiceProxy olacaktır. Bu nesne RentalServiceImpl gibi RentalService interface sınıfını implemente ettiği için bu nesne üzerinde rentACar() metodunu koşturabiliriz. ClassExtenderAspect aspekt aktif hale geldiğinde RentalServiceProxy BusRentalService interface sınıfını da implemente etmiş olacaktır, yani RentalServiceProxy artık rentABus() metoduna sahiptir. Bu metodu kullanabilmek için (BusRentalService)ctx.getBean("rentalService")

şeklinde rentalService nesnesini edinmemiz gerekir. Edineceğimiz yeni nesne BusRentalService veri tipinde olacağından rentABus() metodunu doğrudan bu nesne üzerinde koşturabiliriz. Burada yaptığımız işlem RentalServiceProxy üzerinde rentABus() metodunu koşturmaktır. Proxy istek yapılan metot türüne göre bu istediği ya vekil olduğu RentalServiceImpl nesnesine ya da BusRentalService interface sınıfını implemente eden ClassExtenderAspect nesnesine yönlendirecektir. Böylece RentalServiceImpl sınıfına dinamik olarak BusRentalService interface sınıfında yer alan rentABus metodunu atamak mümkün hale gelmektedir.

```
Kod 9.21 - applicationContext.xml

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
            ref="customerRepository" />
    <property name="rentalRepository" ref="rentalRepository" />
    <property name="carRepository" ref="carRepository" />
</bean>
```

@DeclareParents ile yaptığımız aspekt konfigürasyonunu Spring XML dosyasında da yapabiliriz. Böyle bir konfigürasyon kod 9.22 de yer almaktadır. Aop:declare-parents elementi ile genişletilmek istenen sınıf tanımlanmaktadır. Implement-interface genişletme işlemi için hangi interface sınıfın kullanılacağını tayin etmektedir. Default-impl ile kullanılacak implementasyon sınıfı belirlenmektedir. Bu sınıf BusRentalService interface sınıfını implemente ederek rentABus() metodunun kullanımını sağlamaktadır.

```
Kod 9.22 - applicationContext.xml

<aop:aspect>
    <aop:declare-parents
        types-matching="com.kurumsaljava.spring.RentalServiceImpl"
        implement-interface="com.kurumsaljava.spring.
                BusRentalService"
        default-impl="com.kurumsaljava.spring.aspect.
                ClassExtenderAspect"/>
</aop:aspect>
```

Default-impl ile tanımlanan sınıf bir Spring bean ise, delegate-ref ile bu nesnenin kullanılması sağlanabilir. Bunun bir örneği kod 9.23 de yer almaktadır. Kod 9.20 de kullandığımız @DeclareParents ile bu mümkün değildi, çünkü @DeclareParents bir Spring anotasyonu değildir. AspectJ bünyesinde yer

alan bu anotasyon Spring nesnelerine referans verecek yapıda değildir. Bu özelliği kullanabilmek için konfigürasyon dosyasında aop:declare-parents elementinin kullanılması gerekmektedir.

```
Kod 9.23 - applicationContext.xml

<aop:aspect>
    <aop:declare-parents
        types-matching="com.kurumsaljava.spring.RentalServiceImpl"
        implement-interface="com.kurumsaljava.spring.
                BusRentalService"
        delegate-ref="busRentalService"/>
</aop:aspect>

<bean id="busRentalService"
    class="com.kurumsaljava.spring.aspect.ClassExtenderAspect"/>
```

Spring AOP'nin Sınırları

Spring AOP kullanıldığında sadece public olan metotlar join point olarak seçilebilir. Bunun yanı sıra oluşturulan aspektler sadece Spring nesneleri ile harmanlanabilir. Spring bir interface sınıfın metodu join point olarak seçildiğinde dinamik vekil nesneler (dynamic proxy) oluşturur. Join point interface sınıfı olmayan bir sınıf metodu ise, Spring bu durumda CGLIB kütüphanesini kullanarak vekil nesne oluşturur. CGLIB kullanıldığında final olan metotlar aspektler ile harmanlanamaz, çünkü bu metotlar vekil nesne bünyesinde yeniden yapılandırılamaz (override). Vekil nesneler kullanılırken bir join point olarak seçilen metodun aynı sınıf üzerinde başka bir metodu koşturması durumunda ilk metot için koşturulan advice diğer metot için koşturulmaz.

Daha geniş çaplı join point ve pointcut seçimi için AspectJ çatısının kullanımı tavsiye edilmektedir.

9. Bölüm Soruları

9.1 İşletme mantığı haricinde uygulamanın genelinde aynı işlevi yerine getirmek için kullanılan kod birimlerine ne ad verilmektedir?

9.2 Aspekt nedir?

9.3 AOP'de kullanılan harmanlama yöntemleri nelerdir?

9.4 Spring hangi AOP türünü kullanmaktadır?

9.5 Spring AOP'yi aktif hale getiren konfigürasyon elementi hangisidir?

9.6 Spring hangi join point türünü desteklemektedir?

9.7 Advice sırası nasıl tanımlanabilir?

10. Bölüm

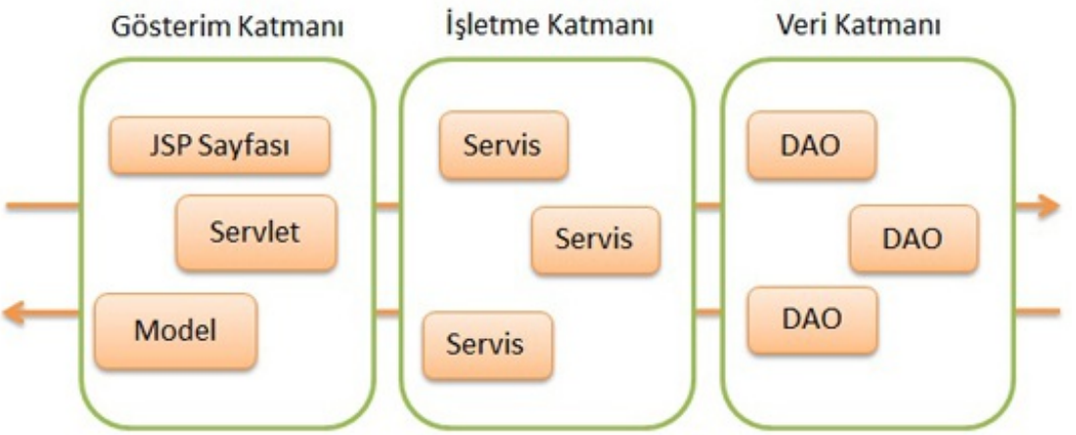
Spring MVC

Java dünyasında web uygulamaları geliştirirken kullanılabilecek en temel çatı (framework) servlet çatısıdır. Kod 10.1 HelloWorldServlet ismini taşıyan bir servlet örneği yer almaktadır. Bu servlet web tarayıcısında "Hello World" cümlesinin görünmesini sağlamaktadır. HelloWorldServlet sınıfını yakından incelediğimizde, çoğu servlet sınıfında uygulanan bir yöntemle karşılaşmaktayız: iş mantığı ile HTML kodunun (örneğin <h1>) bir servlet sınıfı bünyesinde iç içe geçmiş durumda olması.

```
Kod 10.1 - HelloWorldServlet

public class HelloWorldServlet extends HttpServlet {

    private String message;

    @Override
    public void init() throws ServletException {
        message = "Hello World";
    }

    @Override
    public void doGet(final HttpServletRequest request,
            final HttpServletResponse response)
                throws ServletException, IOException {
        response.setContentType("text/html");
        final PrintWriter out = response.getWriter();
        out.println("<h1>" + message + "</h1>");
    }
}
```

İş mantığı ile HTML kodunun bir Java sınıfı bünyesinde birlikte yer almaları, böyle bir sınıfın bakımını zorlaştıran bir durumdur. Uygulamanın web arayüzü değiştirilmek istendiğinde, HTML kodunu bünyesinde taşıyan Java sınıflarının değiştirilmesi gerekir. Bu yapıya sahip servlet tabanlı web uygulamalarının bakım ve geliştirme maliyetlerinin ne kadar yüksek olabileceklerini tahmin edebilirsiniz.

Bu sorunu gidermek ve iş mantığı ile HTML kodunu birbirlerinden ayrı yerlerde tutmak amacıyla JSP (Java Server Pages) teknolojisi geliştirilmiştir. JSP üç katmanlı bir mimaride gösterim katmanını geliştirmek için kullanılan teknolojilerden birisidir.



Resim 10.1

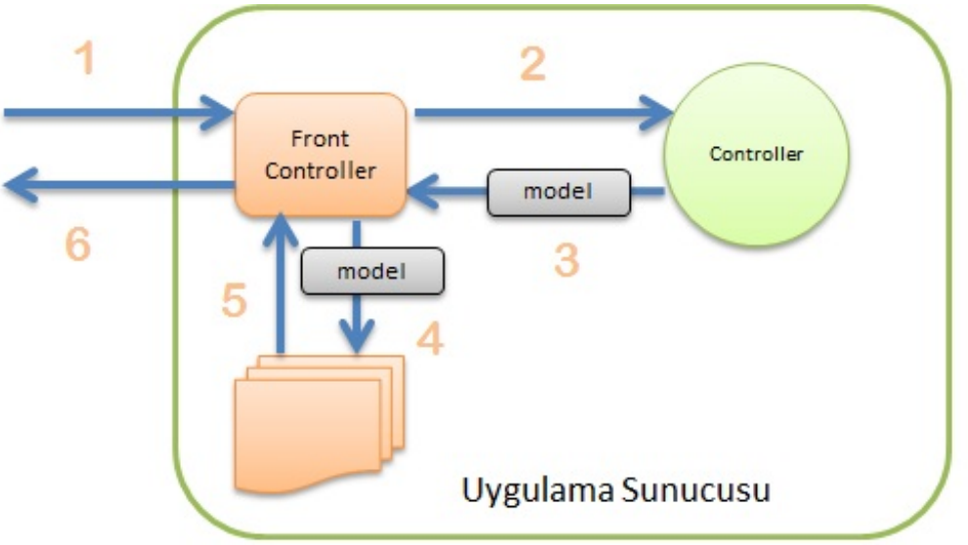
JSP ve Servlet teknolojisi kombine edilebilir. Bu ikilinin beraberliğinden Struts ve Webworks gibi web çatıları doğmuştur. JSTL (Java Standard Tag Library) ve JSP birlikte kullanılarak verilerin sadece gösteriminin yapıldığı arayüzler (view) oluşturulabilir. Servlet teknolojisi kullanılarak arayüzler arası navigasyon ve veri validasyonu (controller) yapılabilir. POJO (Plain Old Java Objects) kullanılarak arayüzlerde gösterilen verilerin yer aldığı model sınıfları oluşturulabilir. Hepsi bir araya geldiğinde MVC (Model View Controller) ismi verilen web çatıları oluşturmak mümkündür.

MVC bir tasarım şablonudur. Bu tasarım şablonuna göre gösterim katmanını oluşturan komponentler belirli bir görevi yerine getirecek şekilde yapılandırılır. MVC kısaltmasındaki her harf bir görev alanını ifade etmektedir. Ait oldukları görev alanında çalışan komponentler sadece ve sadece o görev alanının gerektirdiği görevleri yerine getirirler. Bu tek sorumluluk prensibine (Single Responsibility Principle) işaret etmektedir. Tek bir sorumluluğu olan bir komponent tek bir sebepten dolayı değiştirilebilir. Örneğin bir controller sınıfı arayüz, navigasyon, validasyon ve veri tabanı işlemlerini tek başına yapsa idi, dört değişik sebepten dolayı değişikliğe maruz kalabilirdi ki bu da bu controller sınıfının çok sık bir şekilde ve gereksiz yere değişikliğe uğraması anlamına gelirdi. Tek sorumluluk prensibine uymayan komponentler istikrarsız olduklarından uygulamanın genelde daha kırılgan olmalarını sağlarlar.

Spring herhangi bir web çatısına entegre edilebildiği gibi, Spring bazlı web programcılığını mümkün kılmak için Spring MVC, Spring Webflow ve Spring BlazeDS Integration isminde web çatılarına sahiptir. Bu bölümde Spring MVC web çatısını yakından inceleyeceğiz.

Spring MVC ile Kullanıcı İsteğinin İşlenişi

Spring'in temel web çatısı Spring MVC ismini taşımaktadır. Bu web çatısı MVC tasarım şablonu kullanılarak tasarlanmıştır. Temel işleyiş tarzı resim 10.2 de yer almaktadır.



Resim 10.2

Spring MVC kendi konfigürasyonu için Spring'i kullanmaktadır. Controller sınıfları Spring bean olarak tanımlanır. Spring 2.5 ile birlikte anotasyon bazlı konfigürasyon yapılabilmektedir. Örneğin @Controller anotasyonu herhangi bir bir Java sınıfını bir Spring MVC controller sınıfına dönüştürmektedir. Spring MVC Servlet API'si üzerine inşa edilmiş bir web çatısıdır. Bu Spring MVC'nin çekirdeğinde Servlet ya da HttpServlet sınıflarının kullanıldığı, kullanıcı isteklerinin HttpServletRequest, kullanıcıya gönderilen cevapların HttpServletResponse sınıfları aracılığı ile şekillendirildiği anlamına gelmektedir.

Spring MVC'nin merkezinde DispatcherServlet sınıfı yer almaktadır. Front Controller tasarım şablonu kullanılarak inşa edilen bu yapıda DispatcherServlet kullanıcı isteklerinin uygulama tarafından tatmin edilisini koordine etmektedir. Resim 10.2 de FrontController Spring MVC'nin DispatcherServlet sınıfını temsil etmektedir. DispatcherServlet Struts çatısında ActionServlet, JSF çatısında FacesServlet sınıfları ile aynı görevi yerine getirmektedir.

Kullanıcı web tarayıcısı aracılığı ile herhangi bir linke tıkladığı zaman, bu bir yeni HTTP isteği (request) oluşturur. Bu istek doğrudan DispatcherServlet (resim 10.2:1) tarafından karşılanır. DispatcherServlet'in görevi isteği herhangi bir controller sınıfına (resim 10.2:2) yönlendirmektir. Controller isteği işleme koyan Spring nesnesidir (Spring bean). Bir Spring MVC uygulamasında birden

fazla controller sınıfı olabilir. DispatcherServlet isteği hangi controller sınıfına yönlendireceğine karar verebileceği bir mekanizmaya ihtiyaç duymaktadır. Bu amaçla DispatcherServler handler mapping ismi verilen yapıları kullanır. Handler mapping yapıları kullanıcının isteğini taşıyan web adresi ile bu isteği isleyecek olan controller sınıfları arasındaki bağı oluşturmak için kullanılır. Kullanıcı isteği doğru controller sınıfına eriştikten sonra bu controller sınıfı tarafından işleme konur. Bu çoğu zaman gösterim katmanında yer alan controller sınıfının işletme katmanında yer alan herhangi bir servis sınıfına erişerek, gerekli iş mantığının koşturulmasını sağlaması anlamına gelmektedir. Controller tarafından yapılan işlem web tarayıcısında gösterilmek üzere bir netice doğurabilir. Bu sebeple bu neticenin tekrar kullanıcıya doğru geriye taşınması gerekir. Bu neticenin yer aldığı sınıfa model sınıfı ismi verilir. Controller görevini yerine getirdikten sonra model sınıfını gösterimi yapacak olan view elementinin ismi ile birlikte DispatcherServlet'e yönlendirir (resim 10.2:3). Model sınıfında yer alan bilgilerin kullanıcıya gösterilmeden önce HTML kullanılarak formatlanması gerekir. Bilgileri formatlamak için JSP gibi bir gösterim teknolojisi kullanılır. Formatlamayı yapmak üzere model sınıfı DispatcherServlet tarafından seçilen view elementine yönlendirir (resim 10.2:4). Controller tarafından belirlenen view ismi doğrudan bir JSP sayfasına işaret etmez. Bunu daha çok mantıksal bir isim olarak düşünmek gerekir. DispatcherServlet view resolver (view resolver komponentini daha sonra detaylı inceleyeceğiz) yardımı ile hangi JSP sayfasının gösterimi yapacağını belirler. JSP kullanılabilecek gösterim teknolojilerinden sadece bir tanesidir. View elementi model sınıfını kullanarak gösterim için gerekli HTML kodunu oluşturur (resim 10.2:5). Bu kod HttpServletResponse yardımı ile kullanıcıya taşınır ve web tarayıcısında gösterilir. Bu noktada kullanıcı isteğinin uygulama tarafından cevaplanması işlemi son bulmuştur (resim 10.2:6).

Spring MVC Kurulumu

Spring MVC kurulumu web.xml bünyesinde DispatcherServlet'in tanımlaması ile başlar. Java tabanlı web uygulamaları WEB-INF dizininde yer alan web.xml dosyası ile konfigüre edilir. DispatcherServlet konfigürasyonu kod 10.2 de yer almaktadır.

```
Kod 10.2 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
```

```
    <servlet-class>org.springframework.web.servlet.
                        DispatcherServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>rentacar</servlet-name>
    <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```

servlet-name elementi Spring MVC uygulamasını konfigüre etmek için önemli bir rol oynamaktadır. Kod 10.2 de yer alan DispatcherServlet tanımlaması uygulamayı konfigüre etmek için gerekli olan Spring MVC XML dosyasını WEB-INF dizininde arayacaktır. Konfigürasyon dosyasının bulunabilmesi için isminin [servlet-name]-servlet.xml formatında olması gerekir. Kod 10.2 yer alan örnekte servlet-name elementinde rentacar imini kullandığımız için Spring MVC XML konfigürasyon dosyasının ismi rentacar-servlet.xml olmalıdır.

Uygulama geliştiricisi kullanmak istediği Spring MVC XML konfigürasyon dosyasının ismini kendisi belirleyebilir. Bunu gerçekleştirmek için contextConfigLocation ismindeki bir parametrenin DispatcherServlet'e tanıtılması gerekmektedir. Bu parametrenin değeri kullanılmak istenen konfigürasyon dosyasının lokasyonu ve ismidir. Kod 10.3 de yer alan DispatcherServlet tanımlaması için contextConfigLocation parametresi kullanılmıştır. Bu parametrenin degeri /WEB-INF/mvc-config.xml dir. DispatcherServlet WEB-INF dizininde yer alan mvc-config.xml dosyasını uygulamayı konfigüre etmek için kullanacaktır.

```
Kod 10.3 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
    <servlet-class>org.springframework.web.servlet.
                    DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Tekrar kod 10.2 ye göz attığımızda servlet-mapping isminde bir elementin kullanıldığını görmekteyiz. Bu element kullanıcı isteği ile bu isteği işleyen

servlet arasında bağ kurmak için kullanılır. Kod 10.2 de yer alan örnekte .mvc ekini taşıyan tüm linkler rentacar ismini taşıyan servlet, yani DispatcherServlet'e yönlendirilir. Örneğin http://localhost/welcome.mvc linkine tıklandığında oluşan HTTP isteği (request) DispatcherServlet tarafından işlem görür. url-pattern elementine değişik değerler atayarak DispatcherServlet'e yönlendirilecek kullanıcı istek türlerini yönlendirebiliriz. Kullanılabilecek bazı şablonlar şunlardır:

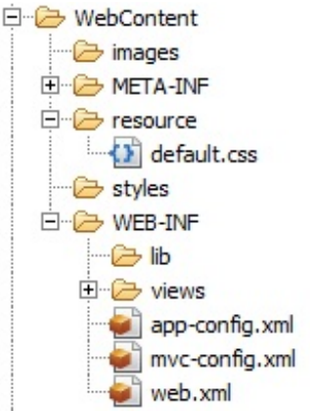
- ***<url-pattern>/</url-pattern>*** - Bir isim alanı (domain name; örneğin http://pratikprogramci.com/) altındaki tüm linkleri DispatcherServlet'e yönlendirir.
- ***<url-pattern>/app/*</url-pattern>*** - Sadece /app/ (örneğin http://pratikprogramci.com/app/welcome) altındaki tüm linkleri DispatcherServlet'e yönlendirir.
- ***<url-pattern>*.abc</url-pattern>*** - .abc ekibini (örneğin http://pratikprogramci.com/welcome.abc) taşıyan tüm linkleri DispatcherServlet'e yönlendirir.

Seçilen url-pattern / olduğu taktirde uygulamaya yöneltilen her istek DispatcherServlet tarafından cevaplanacaktır. Bu Javascript, CSS ve JPEG/GIF gibi kaynaklar için yapılan isteklerin de DispatcherServlet'e yönlendirilmesi anlamına gelmektedir. Statik olan bu kaynakların DispatcherServlet'e yönlendirilmesi gereksizdir. Uygulama sunucusu bu statik kaynakları DispatcherServlet araya girmeden doğrudan kullanıcıya sunabilir. Statik ve dinamik kaynakların Spring tarafından ayrıştırılmalarını sağlamak için mvc:resources elementi kullanılabilir.

```
10.4 - mvc-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/
                    context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/
                    schema/beans
            http://www.springframework.org/schema/beans/
                    spring-beans-3.0.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/
                    spring-mvc-3.0.xsd">
```

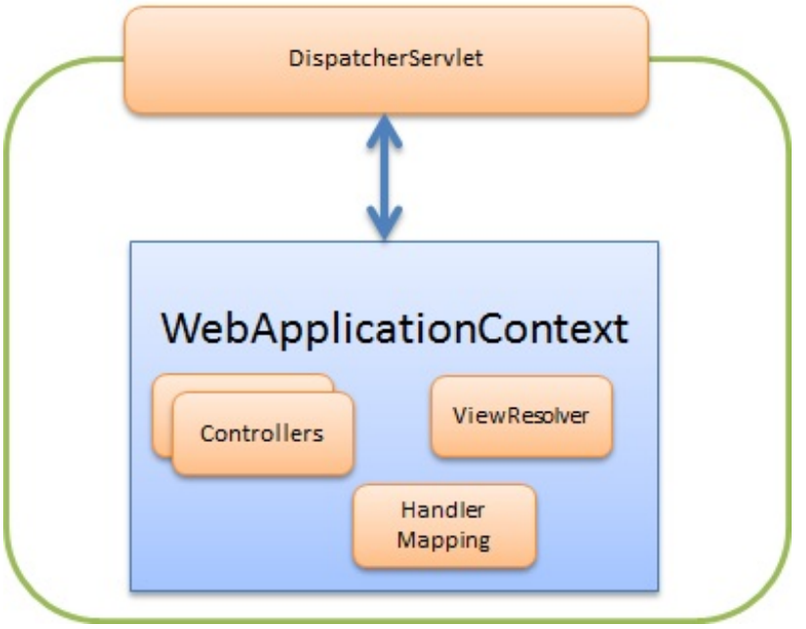
```
    <mvc:resources mapping="/static/**" location="/resource/"/>
</beans>
```

Kod 10.4 de yer alan örnekte mvc:resources elementi yardımı ile statik kaynakların doğrudan sunumunu gerçekleştirmek için bir konfigürasyon oluşturulmaktadır. Mapping element özelliği statik kaynakların web üzerinden hangi adres altında erişilebilir olduklarını, location element özelliği uygulama sunucusunun fiziksel olarak bu kaynaklara nasıl ulaşabileceğini tanımlamaktadır. Bu konfigürasyona göre /static/ konumuna sahip tüm kaynaklar (örneğin http://localhost/static/default.css) /resource/ isimli dizinden alınarak kullanıcıya sunulacaktır. Dizin yapısı resim 10.3 de yer almaktadır.



Resim 10.3

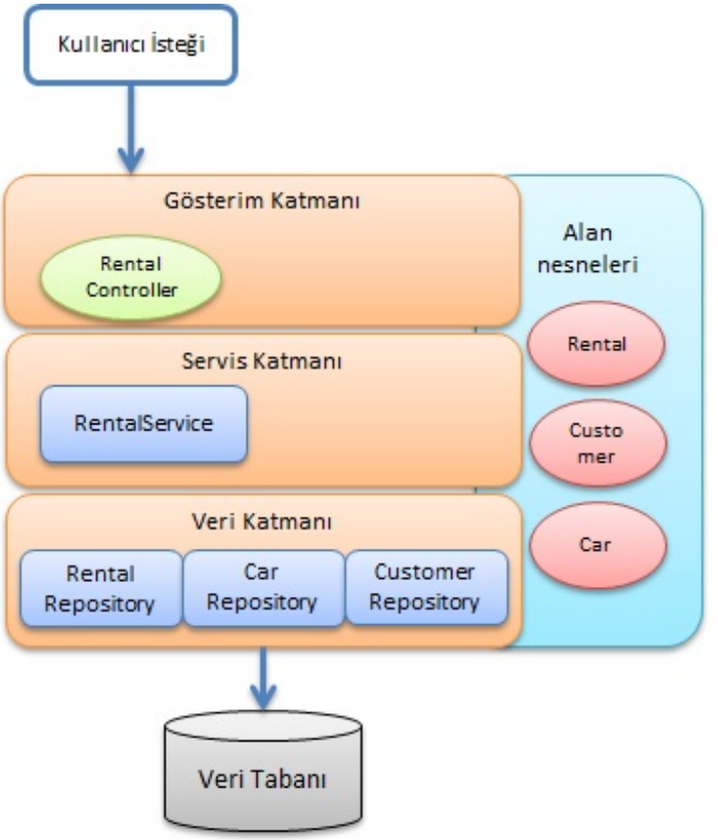
Spring MVC uygulamamızı konfigüre etmek için çıkış noktamız kod 10.4 de yer alan mvc-config.xml ve kod 10.3 de yer alan web.xml dosyalarıdır. Spring MVC çatısında her DispatcherServlet kendi WebApplicationContext nesnesine sahiptir. WebApplicationContext Controller, HandlerMapping, ViewResolver ve diğer Spring nesnelerini ihtiva eder. Spring MVC uygulamasının herhangi bir yerinden WebApplicationContext aracılığı ile bu nesnelere erişmek ve kullanmak mümkündür.



Resim 10.4

Spring MVC ve Uygulama Mimarisi

Diğer bölümlerde Spring'in kullanımını hayali olan bir araç kiralama servisi uygulaması üzerinde örneklemeye çalıştım. Bu bölümde bu uygulamayı web tabanlı bir Spring MVC uygulaması haline getireceğiz. Uygulamayı geliştirmeden önce nasıl bir mimariye sahip olacağını yakından inceleyelim. Resim 10.5 de web tabanlı araç kiralama servisi uygulamasının mimarisi yer almaktadır.

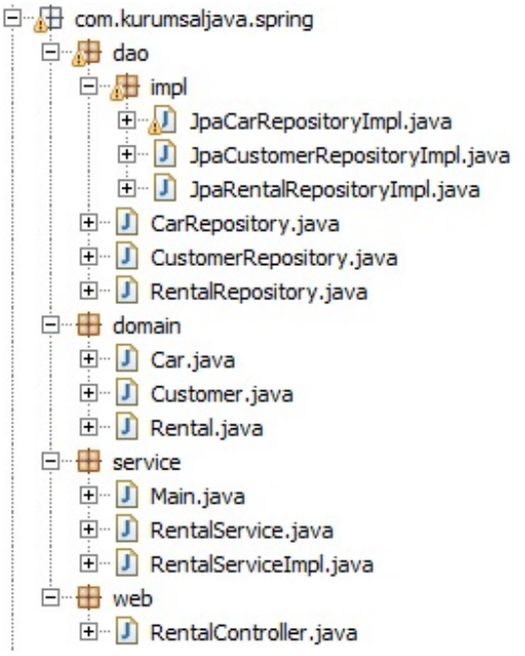


Resim 10.5

Uygulama mimarisi üç katmandan (layer; tier) oluşmaktadır. En üst katman gösterim katmanıdır. Bu katman Spring MVC'yi kullanarak oluşturduğumuz katmandır. Tüm controller, model ve view elementleri bu katmanda yer alır. Araç kiralama sürecini yönetmek üzere oluşturacağımız RentalController sınıfını bu katmanda konuşlandıracağız.

Gelen kullanıcı istekleri gösterim katmanında yer alan controller sınıfları tarafından servis katmanında yer alan servis sınıflarına yönlendirilir. Araç kiralama uygulaması örneğinde RentalController sınıfı araç kiralama işlemini gerçekleştirmek için servis katmanında yer alan RentalService sınıfını kullanmaktadır. RentalService veri tabanı işlemlerini yerine getirmek için veri katmanında yer alan CustomerRepository, RentalRepository ve CarRepository sınıflarını kullanmaktadır. Veri katmanında yer alan repository sınıfları JPA aracılığı ile veri tabanı işlemlerini gerçekleştirmektedir.

Alan (domain) nesneleri olan Car, Customer ve Rental sınıfları tüm katmanlar tarafından ortak kullanılmaktadır. Alan nesneleri servis ve veri katmanında klasik alan nesnesi vazifesini görürken, gösterim katmanında model nesneleri olarak kullanılmaktadırlar.



Resim 10.6

Resim 10.6 da uygulama sınıflarının yer aldığı paket yapısı görülmektedir. Her katman için bir paket oluşturulmuştur. Gösterim katmanını oluşturan sınıflar web, servis katmanını oluşturan sınıflar service, veri katmanını oluşturan sınıflar dao paketinde yer almaktadır. Kullanılan alan nesneleri domain isimli paket içinde yer almaktadır.

Controller Tanımlaması

Her web uygulamasının bir giriş (home; index) sayfası bulunur. Giriş sayfasında yer alan linkler aracılığı ile kullanıcı uygulama ile interaksiyona girer. Bu bölümde araç kiralama servisi uygulaması için giriş ve kiralama işleminin yapıldığı sayfaları oluşturacağız.

Resim 10.7

Resim 10.7 de uygulamanın giriş sayfası yer almaktadır. Böyle bir sayfanın kullanıcıya gösterilmesini sağlamak için HTML kodunun yer aldığı JSP sayfasını ve içeriğin oluşmasını sağlayan IndexController sınıfını oluşturmamız gerekiyor. Kod 10.5 de IndexController sınıfının kodu yer almaktadır.

```
Kod 10.5 - IndexController

package com.kurumsaljava.spring.web;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class IndexController {

    @RequestMapping(value = "/")
    public String index(final ModelMap model) {
        model.addAttribute("titel", "Araç Kiralama Servisi");
        return "index";
    }
}
```

IndexController sınıfı tipik bir Spring MVC controller sınıfında olması gereken özelliklere sahiptir. IndexController sınıfında ilk göze çarpan @Controller anotasyonudur. Bu anotasyon bir Java sınıfını bir Spring MVC controller sınıfı olarak işaretler. Anotasyon bazlı konfigürasyonu aktif hale getirmek için konfigürasyon dosyasına context:component-scan ve mvc:annotation-driven

elementlerini eklememiz gerekmektedir. Kod 10.6 de bu iki elementini kullanılış şekli yer almaktadır. Spring classpath içinde yer alan tüm Java sınıflarını taradıktan sonra @Controller anotasyonunu taşıyan tüm sınıfları Spring MVC controller sınıfı olarak kullanıma hazır tutacaktır.

Kod 10.5 de yer alan IndexController sınıfının herhangi bir üstsınıfının olmadığını görmekteyiz. Bunun yanı sıra bu sınıfın doğrundan servlet çatısına bağımlılığı bulunmamaktadır. Bu controller sınıflarının test edilebilirliğini olumlu etkileyen bir durumdur. Bir Spring MVC uygulamasının nasıl test edildiğini ilerleyen bölümlerde yakından inceleyeceğiz.

```
10.6 - mvc-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
        http://www.springframework.org/schema/beans/
                spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                spring-context-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/
                spring-mvc-3.0.xsd">

    <context:component-scan
                base-package="com.kurumsaljava"/>

    <mvc:annotation-driven/>

    <mvc:resources mapping="/static/**" location="/static/"/>
</beans>
```

Kullanıcı ve uygulama geliştirici olarak beklentimiz http://localhost:8080/ adresine istek yapıldığında resim 10.7 de yer alan sayfanın görünmesidir. Burada uygulamaya yapılan istek / adresidir. DispatcherServlet'in bu isteği IndexController sınıfına yönlendirebilmesi için bu adres ile IndexController sınıfı arasında bir bağ oluşturmamız gerekmektedir. Bu bağı oluşturmak için @RequestMapping anotasyonu kullanılır. Adres ile controller arasında oluşturulan bu bağa handler mapping ismi verilmektedir. Kod 10.5 de yer alan

örnekte @RequestMapping anotasyonu / adresine gelen tüm kullanıcı isteklerinin IndexController sınıfında yer alan index() metoduna yönlendirilmesini sağlar. @RequestMapping anotasyonun yer aldığı metot herhangi bir ismi taşıyabilir.

Spring MVC bünyesinde değişik türde handler mapping implementasyonları mevcuttur. Bunlardan bazıları:

- ***BeanNameUrlHandlerMapping*** - Talep edilen web adresi ile (örneğin /rental) aynı Spring bean ismini taşıyan controller sınıfları arasında bağ (mapping) kurmak için kullanılır.
- ***DefaultAnnotationHandlerMapping*** - @RequestMapping anotasyonunu taşıyan controller sınıfları ile kullanıcı isteğini arasında bağ kurmak için kullanılan handler mapping türüdür.
- ***ControllerClassNameHandlerMapping*** - Talep edilen web adresi ile (örneğin /rental) aynı ismi taşıyan controller sınıfları arasında bağ (mapping) kurmak için kullanılır.
- ***SimpleUrlHandlerMapping*** - Talep edilen web adresinin istek anında bir controller sınıfı ile dinamik olarak eşlenme işlemini yapmak için kullanılan handler mapping türüdür.

Kod 10.6.1 de yer alan örnekte /index=indexController şeklinde bir eşleme yapılmaktadır.

```
Kod 10.6.1 - applicationContext.xml
<bean
    class="org.springframework.web.servlet.handler.
                            SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/index">indexController</prop>
        </props>
    </property>
</bean>

<bean id="indexController"
    class="com.kurumsaljava.spring.web.IndexController"/>
```

Arkasında bir controller sınıfı olmayan view elementlerini göstermek için kod 10.6.2 de yer alan UrlFilenameViewController sınıfı kullanılabilir. Bu controller implementasyonu istek yapılan sayfa ismini view ismine dönüştürerek, bu view elementinin kullanıcıya gösterilmesini sağlamaktadır. Örneğin kullanıcı

/secure/index şeklinde bir sayfa talebinde bulunduysa, UrlFilenameViewController /WEB-INF/views/index.jsp (bknz. Kod 10.7) JSP sayfasının gösterilmesini sağlar.

```
Kod 10.6.2 - applicationContext.xml

<bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.
                SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/**">urlFilenameViewController</prop>
        </props>
    </property>
</bean>

<bean id="urlFilenameViewController"
    class="org.springframework.web.servlet.mvc.
        UrlFilenameViewController" />
```

Handler mapping sınıfları HandlerMapping ismini taşıyan interface sınıfı implemente ederler. Yazılımcının bu interface sınıfını implemente ederek yeni handler mapping sınıfları oluşturması mümkündür. Uygulama bünyesinde bir handler mapping tanımlanmadığı taktirde Spring otomatik olarak BeanNameUrlHandlerMapping ya da DefaultAnnotationHandler handler mapping implementasyonunu kullanır. Anotasyon bazlı controller sınıfları kullandığımız için bu bizim örneğimizde DefaultAnnotationHandler sınıfıdır.

Model Taşıyıcı ModelMap

Kod 10.5 de yer alan index() metodunun ModelMap tipinde bir parametresi mevcuttur. Bu nesne conroller ile view arasında veri taşımak için kullanılan bir yapıdır. Verileri taşıyan model nesneleri ModelMap aracılığı ile view elementlerinin kullanımına sunulur. Bu yapıya eklenen her model nesnesinin bir anahtarı mevcuttur. Bu anahtar kullanılarak view içinde iken verileri tekrar elde etmek mümkündür. Kod 10.5 de yer alan örnekte index ismini taşıyan view elementi için titel anahtarına sahip String veri tipinde bir model nesnesi ModelMap'e eklenmektedir.

View Resolver Tanımlaması

Index() metodu index değerini taşıyan bir String nesnesini geri vermektedir. Bu gösterimi yapacak olan JSP sayfasının ismidir. DispatcherServlet IndexController sınıfında yer alan index() metodundan bu değeri edindikten sonra, gerekli JSP sayfasına yönlendirmeyi gerçekleştirir. Bu ismi taşıyan bir JSP sayfasını bulabilmesi için mvc-config.xml dosyasında bir view resolver'in tanımlanması gerekmektedir. Bunun bir örneği kod 10.7 da yer almaktadır.

```
Kod 10.7 - mvc-config.xml

<bean class="org.springframework.web.servlet.view.
                InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

DispatcherServlet kod 10.7 de kullandığımız InternalResourceViewResolver yardımı ile index.jsp ismini taşıyan JSP dosyasını /WEB-INF/views/index.jsp lokasyonunda bulmaya çalışacaktır. DispacherServlet gösterimden sorumlu JSP sayfasını tespit ettikten sonra kontrolü bu JSP sayfasına bırakarak, JSP sayfasının model sınıfları yardımı ile HTML içeriği oluşturmasını sağlayacak, bu işlem tamamlandıktan sonra kontrolü tekrar üstlenerek bu içeriğin HttpServletResponse nesnesi olarak web tarayıcısına gönderilmesini sağlayacaktır. Bu işlemin ardından kullanıcı ile uygulama arasındaki istek döngüsü son bulmaktadır.

WEB-INF dizininde yer alan JSP sayfalarına web tarayıcısı üzerinden doğrudan erişmek mümkün değildir. WEB-INF dizinde yer alan tüm kaynaklar uygulama sunucusu tarafından korunur. Bu açıdan bakıldığında InternalResourceViewResolver JSP sayfalarının kaynak kodunun korunması için iyi bir seçenektir.

```
Kod 10.8 - index.jsp

<html>
<body>
    <h1>${titel}</h1>

    <a href="/rental">Araç kiralama formu </a>
</body>
</html>
```

Kod 10.8 de index.jsp sayfasının kaynağı yer almaktadır. Dolar işareti ile

başlayan tanımlama (${titel}) bir yer tutucudur (placeholder). Titel kelimesi IndexController.index() metodunda oluşturarak ModelMap nesnesine eklediğimiz String tipindeki modelin anahtarıdır. JSP sayfası içinde bu modelin içeriğine erişebilmek için bu anahtarı kullanmamız gerekmektedir. Spring MVC çatısı JSP sayfalarında dolar işareti ile karşılaştığı zaman bunun bir model element anahtarı olabileceğini düşünerek, bu yer tutucunun modelin değeri ile yer değiştirmesini sağlar. Bu şekilde JSP sayfalarında dinamik içerik oluşmuş olur.

View Resolver Türleri

Gösterimi yapan view elementlerinin MVC çatısı tarafından lokalize edilerek model sınıflarında yer alan bilgilerin kullanıcıya gösterilmesi (model rendering) gerekmektedir. Bu görevi Spring MVC bünyesinde view resolver sınıfları üstlenmektedir. Kod 10.7 de InternalResourceViewResolver ismini taşıyan ilk view resolver sınıfı ile tanıştık.

Spring MVC view elementlerinin bulunma işleminde kullanılmak üzere ViewResolver ve View isminde iki interface sınıf ihtiva etmektedir. ViewResolver view element isimleri ile view elementleri arasında eşleme (mapping) yapmak için kullanılır. Örneğin kod 10.7 de kullandığımız InternalResourceViewResolver sınıfı kod 10.5 deki index() metodunun son satırında yer alan index ismini /WEB-INF/views/index.jsp ile eşlemektedir. Eğer kod 10.5 de yer alan index() metodu abc değerini geri vermiş olsaydı, InternalResourceViewResolver /WEB-INF/views/abc.jsp şeklinde bir eşleme yaparak abc.jsp sayfasının gösterimi yapmasını sağlardı.

ViewResolver interface sınıfı yanı sıra bahsettiğim View interface sınıfı kullanıcı isteğinin (HttpServletRequest) seçilen view elementine aktarılmasını sağlamak için kullanılmaktadır.

ViewResolver interface sınıfını implemente eden bazı Spring MVC view resolver sınıfları şunlardır:

- ***AbstractCachingViewResolver*** - Bazı view elementleri için gösterim öncesi hazırlık yapmak gerekli olabilir. Her gösterim öncesinde bu hazırlık işlemlerinin tekrarını önlemek için view elementleri önbelleğe (cache) alınabilir. AbstractCachingViewResolver sınıfı view elementlerinin önbelleğe alınmalarını sağlamak için genişletilebilecek view resolver sınıfıdır.

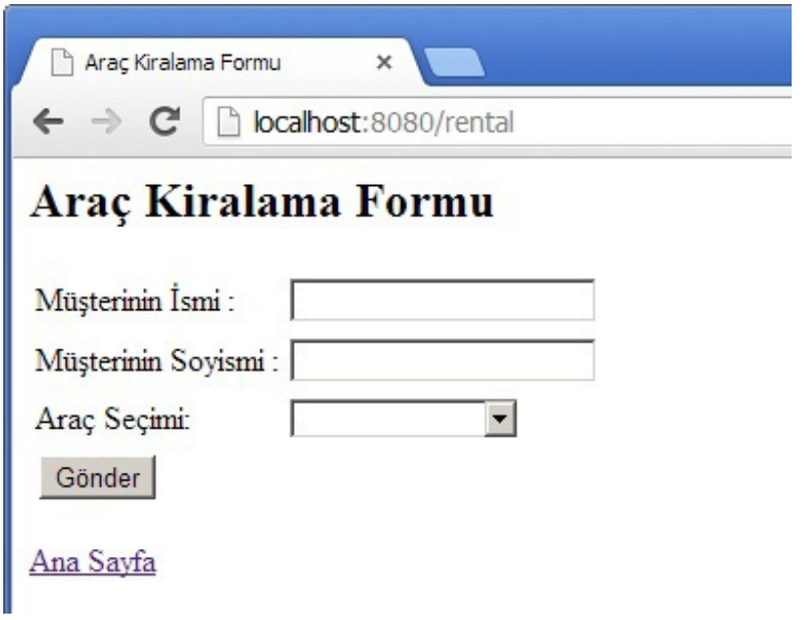
- ***XmlViewResolver*** - Bir XML dosyasında yer alan ve bir Spring bean gibi tanımlı olan view elementlerini bulmak için kullanılan view resolver türüdür. Varsayılan XML /WEB-INF/views.xml isimli dosyasıdır.
- ***ResourceBundleViewResolver*** - Property dosyalarında yer alan view tanımlamalarını kullanarak istenilen view elementini bulmak için kullanılan view resolver türüdür. Varsayılan property dosyası ismi views.properties'dir. Bu dosyasının classpath içinde olması gerekir.
- ***UrlBasedViewResolver*** - InternalResourceViewResolver gibi sınıfların üstsınıfı olan view resolver sınıfıdır. Kullanılan web adresi ile view elementi arasında ilişki kurmak için kullanılır. Örneğin talep edilen adres /index ise bu view resolver uygulama sunucusunun ana dizininde (doc root) yer alan index.jsp sayfasını seçer.
- ***VelocityViewResolver / FreeMarkerViewResolver*** - UrlBasedViewResolver sınıfının bir altsınıfı olan bu view resolver JSP gibi bir gösterim teknolojisi olan Velocity ve Freemarker ile yapılmış view elementlerinin seçimi için kullanılır.
- ***ContentNegotiatingViewResolver*** - Kullanıcı isteği (HttpServletRequest) bünyesinde yer alan "request file name" ya da "Accept header" elementlerinin değerine göre view element seçimi yapan view resolver türüdür. "request file name" ya da "Accept header" kullanıcının talep ettiği içerik tipini belirler.
- ***JasperReportsViewResolver*** - View elementinin Jasper Reports olarak hazırlanmış bir dosyalar arasında seçimi için kullanılan view resolver sınıfıdır.
- ***TilesViewResolver*** - Tiles template olarak hazırlanmış bir view elementinin seçimi için kullanılır.
- ***XsltViewResolver*** - XSLT tabanlı bir view elementinin seçimi için kullanılan view resolver türüdür.

Görüldüğü gibi Spring MVC uygulamalarında gösterim için kullanılan teknoloji JSP ile sınırlı degildir. Freemarker ya da Velocity gibi şablon (template) mekanizmaları sunan gösterim teknolojilerini de kullanmak mümkündür. Gerekli view resolver sınıfının konfigürasyonu kullanılan gösterim teknolojisinin Spring MVC ile entegrasyonunu sağlamaktadır.

Araç Kiralama Formu

Araç kiralama işlemini web tabanlı hale getirebilmek için bir HTML form

oluşturmamız gerekmektedir. Böyle bir form prototipi resim 10.8 de yer almaktadır.



Resim 10.8

Böyle bir formu oluşturabilmek için form elementlerini ihtiva eden bir JSP sayfasına, forma kaydedilen bilgileri işleyen bir servis sınıfına, navigasyon ve veri validasyonundan sorumlu bir controller sınıfına, hem kullanıcı verilerini hem de işlem sonuçlarını bünyesinde taşıyan bir model sınıfına ve veri tabanı işlemlerini gerçekleştiren DAO sınıflarına ihtiyacımız bulunmaktadır.

Controller sınıfının içeriğini oluşturarak başlamamız doğru olacaktır, çünkü hangi sayfanın gösterileceğini tayin eden controller sınıfıdır. RentalController ismini taşıyan controller sınıfı kod 10.9 da yer almaktadır.

```
Kod 10.9 - RentalController

package com.kurumsaljava.spring.web;

import java.util.LinkedHashMap;
import java.util.Map;
import javax.inject.Inject;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import com.kurumsaljava.spring.domain.Rental;
import com.kurumsaljava.spring.service.RentalService;

@Controller
@RequestMapping("/rental")
public class RentalController {

    private static final String RENTAL_FORM = "rentalForm";

     @Inject
     private RentalService service;


    private static final String DONE_VIEW = "done";
    private static final String RENTAL_VIEW = "rental";

    @RequestMapping(method = RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }

    private void buildCarList(final ModelMap model) {
        final Map<String,String> cars =
            new LinkedHashMap<String,String>();
         cars.put("", "");
         cars.put("1", "Ford Fiesta");
         cars.put("2", "Renault Twingo");
         cars.put("3","Audi A4");
         model.put("cars", cars);
    }

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit( final ModelMap model,
            @ModelAttribute(RENTAL_FORM) @Valid
                final Rental rental,
            final BindingResult result) {
        if (result.hasErrors()) {
            buildCarList(model);
            return RENTAL_VIEW;
        } else {
            processRental(rental);
            return DONE_VIEW;
        }
    }

    private void processRental(final Rental rental) {
        service.rentACar(rental);
```

```
    }
}
```

@Controller anotasyonu ile daha önce tanışmıştık. @Controller anotasyonu sınıfın rolünü tanımlamaktadır. Bu anotasyon yardımı ile RentalController sınıfı bir Spring MVC controller sınıfı haline gelmektedir.

@RequestMapping anotasyonuyla da daha önce tanışmıştık. Kod 10.5 de yer alan örnekte @RequestMapping anotasyonunu metot bazında kullanmıştık. Bu anotasyonu hem metot hem de sınıf bazında kullanmak mümkündür. Sınıf bazında ve metot bazında aynı anda kullanıldığı taktirde, sınıf bazında kullanım şekli controller sınıfının sorumlu olduğu ana uygulama adresini tayin eder. Metot bazında kullanılan anotasyonlar bu adresi tamamlayıcı nitelikte olur. Örneğin araç kiralama formunu kullanıcıya göstermek için kullanılan adres kod 10.9 da http://localhost/rental iken, kod 10.10 da http://localhost/rental/new şeklindedir.

```
Kod 10.10 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/new", method =
                    RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Kullanıcı isteği ile gönderilen parametreleri (request parameter) @RequestMapping anotasyonu yardımı ile değerlendirmek mümkündür. Kod 10.11 de yer alan örnekte kullanıcı isteği /rental/form?new olduğu taktirde getForm() metodu devreye girecektir.

```
Kod 10.11 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/form", method = RequestMethod.GET,
```

```
                 params="new")
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Kod 10.11 de yer alan örnekte new parametresi bir değer taşımamaktadır. Bu parametreye bir değer atayarak kullanıcı isteğini uygulamaya göndermek mümkündür. Örneğin /rental/form?new=1 şeklinde bir isteği karşılamak için @RequestMapping anotasyonunun params="new=1" olarak şekillendirilmesi yeterli olacaktır. Eğer bir parametrenin kullanıcı isteği bünyesinde yer almasını istemiyorsak, bu isteğimizi ünlem işareti kullanarak ifade edebiliriz. params=!new seklindeki bir tanımlama istek bünyesinde new parametresinin olmaması durumunda işlem görecektir.

@RequestMapping anotasyonunda yer alan headers elementi ile kullanıcı isteği ile gönderilen HTTP başlık (header) parametrelerini değerlendirmek mümkündür. Bunun bir örneği kod 10.12 de yer almaktadır.

```
Kod 10.12 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/form", method = RequestMethod.GET,
            headers="key=value")
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Tekrar kod 10.9 da yer alan RentalController sınıfına göz atalım. getForm() metodu üzerinde yer alan RequestMapping anotasyonunda method = RequestMethod.GET ibaresi yer almaktadır. Eğer gönderilen kullanıcı isteği HTTP GET özelliğine sahip ise, bu durumda RentalController bünyesinde yer alan getForm() metodu devreye girecektir. Bu metodun görevi resim 10.8 de yer alan araç kiralama formunun kullanıcıya gösterilmesini sağlamaktır. RequestMapping anotasyonunu kullanarak ifade etmek istediğimiz şey şudur:

Eğer kullanıcı http://localhost/rental şeklinde bir istekte bulunuyorsa ve gönderilen bu isteğin türü HTTP GET ise, bu durumda kullanıcıya araç kiralama formunu göster.

Resim 10.8 der yer alan formun kullanıcıya gösterildiğini farz edelim. Kullanıcı gerekli form alanlarını doldurmuş ve Gönder butonuna tıklamış olsun. Kullanıcının girdiği bu bilgiler uygulamaya bir HttpServletRequest nesnesi olarak erişir. HttpServletRequest.getParameter() metodunu kullanarak kullanıcının girmiş olduğu değerlere erişebiliriz. Bu değerler Rental sınıfından bir nesne olarak elimize ulaşsaydı ve HttpServletRequest ile uğraşmak zorunda kalmasaydık daha iyi olmaz mıydı? Spring MVC'yi kullanmamızın ana nedenlerinden bir tanesi de, servlet çatısına derin dalış yapıp, HttpServletRequest gibi sınıflarla uğraşmak zorunda kalmak istemeyişimizdir. Kullanıcıya gösterilen formu ve ihtiva ettiği bilgileri bir POJO sınıf olarak tasarlayıp, Spring MVC tarafından kullanıcı bilgileri ile otomatik olarak donatılmasını sağlayabiliriz. Bu amaçla kod 19.9 da yer alan getForm() metodunda model nesnesine yeni bir Rental nesnesi eklemektedir. Bu Rental nesnesi Gönder tuşuna tıklandıktan sonra kullanıcının girmiş olduğu tüm bilgileri ihtiva edecektir. Peki Spring MVC formda yer alan verileri Rental sınıfından olan nesneye nasıl eklemektedir? Bunu anlayabilmek için formu oluştururken kullandığımız JSP sayfasına bir göz atmamız gerekmektedir. Formu kullanıcıya göstermek için oluşturduğumuz rental.jsp kod 10.13 de yer almaktadır.

```
Kod 10.13 - rental.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>
<%@taglib uri="http://www.springframework.org/tags/form"
            prefix="form"%>
<html>
<head>
<title>Ara&ccedil; Kiralama Formu</title>
</head>
<body>
    <h2>Ara&ccedil; Kiralama Formu</h2>

    <form:form method="post" action="/rental"
            commandName="rentalForm">
        <table>
            <tr>
                <td>Müşterinin İsmi :</td>
                <td><form:input path="customer.firstname" />
```

```
                    <font color="Red">
                      <form:errors path="customer.firstname"
                             delimiter=", "/>
                    </font>
                </td>
            </tr>
            <tr>
                <td>Müşterinin Soyismi :</td>
                <td><form:input path="customer.name" />
                  <font color="Red">
                    <form:errors path="customer.name"
                             delimiter=", "/>
                  </font>
                </td>
            </tr>
            <tr>
                <td>Araç Seçimi:</td>
                <td>
                    <form:select path="car.userSelection">
                        <form:options items="${cars}" />
                    </form:select>
                    <font color="Red">
                      <form:errors path="car.userSelection"
                             delimiter=", "/>
                    </font>
                </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit"
                             value="G&ouml;nder" /></td>
            </tr>
        </table>

    </form:form>

<p><a href="/rental">Ana Sayfa</a>
</body>
</html>
```

Kod 10.13 der yer alan JSP sayfasında HTML elementleri ile Spring MVC uygulaması arasında bağ oluşturmak için Spring MVC'nin ihtiva ettiği form ismini taşıyan JSP Custom Tag Library (taglib) kullanılmaktadır. Taglib elementlerini HTML elementleri olarak düşünebiliriz. Taglib'ler JSP sayfalarında Java kodu ile program yazmak yerine, HTML benzeri elementler kullanarak dinamik içerik oluşturmak için kullanılmaktadır. Örneğin form:form taglib elementi ile bir HTML formu oluşturulmaktadır. Bu form ile kullanıcının

form aracılığı ile girdiği verileri tutmak için kullanılacak olan Rental nesnesi arasında bağ oluşturmak için commandName elementi kullanılmaktadır. Bu elementin değeri olan rentalForm RentalController.getForm() metodunda model nesnesine eklenen Rental nesnesinin anahtarıdır.

Kullanıcının girmiş olduğu bilgileri Rental nesnesinde yer alan değişkenlere atamak için form:input elementi kullanılmaktadır. Form:input path="customer.firstname" direktifi form üzerinden girilen müşteri ismini doğrudan Rental nesnesinde yer alan Customer nesnesinin name ismini taşıyan değişkenine atamaktadır. Rental sınıfının yapısı kod 10.14 de yer almaktadır. Görüldüğü gibi bir JPA entity sınıfını form nesnesi olarak kullanmak mümkündür. Bu nesneye gösterim katmanında gerekli veriler atandıktan sonra veri katmanında JPA kullanarak bu nesne ve ihtiva ettiği veriler veri tabanına eklenebilir.

```
Kod 10.14 - Rental

package com.kurumsaljava.spring.domain;

@Entity
@Table(name = "rental")
public class Rental {

    @Id
    @GeneratedValue
    @Column(name = "id")
    private long id;

    @Valid
    @OneToOne
    private Car car;

    @Valid
    @OneToOne
    private Customer customer;

    @Column(name="rented")
    private boolean rented;
}
```

Resim 10.8 de yer alan formda müşteri isim ve soyismi için iki alan, kiralanacak aracın türünü belirlemek için bir araç listesi bulunmaktadır. Bu listenin içeriği dinamik olarak kod 10.9 da yer alan getForm() metodunda oluşturulmaktadır. buildCarList() metodu araç türlerini ihtiva eden bir map oluşturarak, bu map

nesnesini cars ismi altında model nesnesine eklemektedir. Bu listenin resim 10.8 de yer alan forma gösterilmesi için kod 10.13 de yer alan JSP sayfasında form:select kullanılmaktadır. Bu element bünyesinde form:options items="${cars} yardımı ile araç türlerinin yer aldığı map nesnesi edinilerek, HTML Select bileşeni oluşturulmaktadır.

Kullanıcı forma gerekli bilgileri girip, Gönder butonuna tıkladığında bu istek uygulama sunucusuna bir HTTP POST isteği olarak gönderilir. Bunun sebebi kod 10.13 de yer alan form:form method="post" action="/rental" tanımlamasıdır. Bu istek /rental adresini taşıdığı için DispatcherServlet tarafından tekrar RentalController sınıfına yönlendirilir. İsteğin controller bünyesindeki hangi metot tarafından cevaplanacağını @RequestMapping anotasyonu belirler. Bu metot processSubmit() ismini taşımaktadır, çünkü bu metodun sahip olduğu @RequestMapping anotasyon method = RequestMethod.POST şeklinde konfigüre edilmiştir.

Formu işleyen processSubmit() metodunun ModelMap, Rental ve BindingResult tipinde üç metot parametresi bulunmaktadır. Bu parametreler haricinde @ModelAttribute ve @Valid anotasyonları kullanılmıştır.

@ModelAttribute anotasyonu ile model içinde yer alan bir nesneye erişmek mümkündür. Kod 10.9 da yer alan processSubmit() metodunda @ModelAttribute anotasyonu model içinde yer alan Rental nesnesini edinmek için kullanılmaktadır. Bu nesne kullanıcının form üzerinde girdiği verileri ihtiva etmektedir. Spring MVC otomatik olarak form alanlarına girilen verileri Rental nesnesine yerleştirir. Bu işleme data binding ismi verilmektedir.

Spring MVC data binding işlemi için WebDataBinder sınıfını kullanmaktadır. WebDataBinder kullanıcı isteği içinde yer alan HTTP parametreleri ile form verilerini tutan nesne arasında değişken ismi bazında eşitleme yapar. Örneğin kod 10.13 de yer alan JSP sayfasında müşteri ismini tutmak için form:input path="customer.firstname şeklinde bir tanımlama yapılmıştır. Kullanıcı Gönder tuşuna tıkladığında WebDataBinder rental.getCustomer().setFirstname() çağrısı ile kullanıcının girmiş olduğu müşteri ismini rental nesnesinde yer alan customer nesnesinin firstname isimli değişkenine yerleştirir. Bu işlem mevcut tüm form elementleri için gerçekleştirilir.

@Valid anotasyonu otomatik veri kontrolü (validation) yapmak için kullanılmaktadır. Bu anotasyon ile Spring MVC JSR (Java Specification Requests) 303 ile Java EE 6'nın bir parçası haline gelen Bean Validation

çatısının kullanımını mümkün kılmaktadır. Kullanıcı formu doldurup, Gönder tuşuna tıkladığında Spring MVC önce WebDataBinder yardımı ile rental nesnesini yapılandıracak ve akabinde @Valid anotasyonuna denk geldiği taktirde otomatik veri kontrol işlemini gerçekleştirecektir. Spring MVC veri kontrolünü Bean Validation çatısında yer alan anotasyonlar yardımı ile yapmaktadır. Kod 10.15 de yer alan Customer sınıfında müşteri ismi (firstname) ve soyismi için veri kontrolü @NotBlank ve @Length anotasyonları kullanılarak yapılmaktadır. @NotBlank form elementine mutlaka bir değerin girilmesi gerektiğini ifade ederken, @Length ile girilen verinin kaç harften oluşabileceği tayin edilmektedir. Kod 10.15 de yer alan örnekte müşteri soyisminin en az bir, en fazla on harften oluşması gerekmektedir. Aksi taktirde veri kontrolü neticesi olarak bir hata oluşacaktır. Data binding ve veri kontrolü sırasında oluşan tüm hatalar BindingResult içinde yer almaktadır. Result.hasErrors() ile hata mevcudiyeti kontrol edilebilir. Hata durumunda formun tekrar kullanıcıya oluşan hatalar ile birlikte gösterilmesi gerekmektedir. return RENTAL_VIEW; tekrar form sayfasına dönüş yapılmasını sağlar.

```
Kod 10.15 - Customer

public class Customer {

    private long id;

    @NotBlank
    @Length(min=1, max=10)
    private String name;

    @NotBlank
    private String firstname;

    private int age;
}
```

Bean Validation çatısında veri kontrolü yapmak için kullanılabilecek anotasyonlar şunlardır:

- ***@AssertFalse*** - Değişkenin false değerinde olmasını bekler.
- ***@AssertTrue*** - Değişkenin true değerinde olmasını bekler.
- ***@DecimalMin*** - Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden yüksek olmasını bekler.
- ***@DecimalMax*** - Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden düşük olmasını bekler.

- ***@Digits*** - Değişkenin bu anotasyon ile belirlenen virgül öncesi ve sonrası basamaklardan oluşmasını bekler.
- ***@Past*** - java.util.Date ve java.util.Calendar için kullanılabilir. Değişken değerinin geçmiş bir zaman diliminde olmasını bekler.
- ***@Future*** - java.util.Date ve java.util.Calendar için kullanılabilir. Değişken değerinin gelecek bir zaman diliminde olmasını bekler.
- ***@Min*** - Sadece integer değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden yüksek olmasını bekler.
- ***@Max*** - Sadece integer değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden düşük olması bekler.
- ***@NotNull*** - Değişkenin null değerine sahip olmamasını bekler.
- ***@Null*** - Değişkenin null değerine sahip olmasını bekler.
- ***@Pattern*** - Sadece String veri tipine sahip değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen regex ifadesi ile uyuşmasını bekler.
- ***@Size*** - String, Collection, Map ve Array veri tiplerine sahip değişkenler için kullanılabilir. Değişkenin sahip olduğu uzunluğun bu anotasyon ile belirlenen min ve max değerleri içinde olmasını bekler.
- ***@Valid*** - Veri kontrolünün rekursif olarak yapılmasını sağlar. Bunun bir örneği kod 10.14 de yer almaktadır. Eğer Rental sınıfı bünyesinde car ve customer nesneleri için @Valid anotasyonu kullanılmamış olsaydı, Customer sınıfında yer alan (kod 10.15) firstname ve name değişkenlerinin veri kontrolü yapılması mümkün olmazdı, çünkü veri kontrolü giriş noktası processSubmit() (kod 10.9) metot parametrelerinden birisi olan ve @Valid ile işaretlenmiş rental nesnesidir.

Veri kontrolü ve data binding işlemlerinde oluşan hataların ekranda kullanıcıya gösterilebilmesi için JSP sayfasında form:errors (kod 10.13) elementi kullanılmaktadır. Resim 10.9 da form bünyesinde oluşabilecek hatalar yer almaktadır. Bu hataların kullanıcıya gösterilebilmesi için formun tekrar yüklenmesi gerekmektedir. Bu işlem processSubmit() metodunda BindingResult nesnesinde yer alan hataların (result.hasErrors()) kontrolü ve return RENTAL_VIEW; komutuyla gerçekleşmektedir.

Resim 10.9

Resim 10.9 da yer alan hata mesajları nerede tutulmaktadır? Bu mesajları bir değer (property) dosyasından edinmek mümkündür. Data binding ve veri kontrolü sırasında oluşan her hatanın bir anahtarı bulunmaktadır. Eğer bir değer dosyasında bu anahtarlara kullanıcıya gösterilmek istenen hata mesajı atanırsa, Spring MVC oluşan hatalar için bu değer dosyasını kullanır. Araç kiralama servisi için kullanılan hata mesajları dosyası kod 10.16 da yer almaktadır.

```
Kod 10.16 - messages.properties

Length.rentalForm.customer.firstname =
        M\u00fc\u015fteri ismi on harften olu\u015fabilir
NotBlank.rentalForm.customer.firstname =
        L\u00fctfen m\u00fc\u015fteri ismini giriniz
Length.rentalForm.customer.name =
        M\u00fc\u015fteri soyismi on harften olu\u015fabilir
NotBlank.rentalForm.customer.name =
        L\u00fctfen m\u00fc\u015fteri soyismini giriniz
NotBlank.rentalForm.car.userSelection =
        L\u00fctfen bir ara\u00e7 se\u00e7iniz
```

Örneğin müşteri ismi girilmeden Gönder butonuna tıklandığında, veri kontrolü esnasında oluşan hatanın anahtarı NotBlank.rentalForm.customer.firstname olacaktır. Görüldüğü gibi bu anahtar veri kontrolü için kullanılan anotasyon (NotBlank), form (rentalForm), değişkenin yer aldığı nesne (customer) ve değişken isminden (firstname) oluşmaktadır. Bu şekilde tüm oluşabilecek hatalar için hata anahtarlarını kestirmek ve kod 10.16 da yer alan messages.properties ismini taşıyan dil dosyasını oluşturmak mümkündür. Oluşturulan bu dosyaya resource bundle ismi verilmektedir. Resource bundle dosyaları çok dilli uygulamalarda kullanılan yapılardır.

Örneğin hata mesajlarının ingilizce dilinde görünmeleri isteniyorsa messagesenEN.properties, almanca dilinde görünmeleri isteniyorsa messagesdeDE.properties ismini taşıyan dil dosyaları oluşturulması ve gerekli tercümenin yapılması yeterli olacaktır. Spring MVC kullanılan dil türüne göre (Locale) gerekli dil dosyasını seçerek, hata mesajlarının bu dilde gösterilmesini sağlayacaktır. Spring MVC'nin dil dosyasını bulabilmesi için konfigürasyon dosyasında dil dosyasının ismi ve lokasyonunun ResourceBundleMessageSource yardımı ile tanımlanması gerekmektedir. Bunun bir örneği kod 10.17 de yer almaktadır.

```
Kod 10.17 - mvc-config.xml

<bean
    class="org.springframework.context.support.
        ResourceBundleMessageSource"
    id="messageSource">
    <property name="basename" value="messages" />
</bean>
```

Kod 10.17 de yer alan örnekte Spring MVC dil dosyalarını classpath içinde arayacaktır. Spring MVC'nin beklediği dil dosya ismi messages'dir. basename parametresi dil dosyasının kök ismini tayin eder. Çok dilli uygulamalarda dil dosya ismi kök dosya ismi artı kullanılan dil türevidir (örneğin ingilizce için en_EN).

Controller Sınıfları ve Bağımlılıkların Enjekte Edilmesi

Resim 10.9 da yer alan araç kiralama formunun kullanıcı tarafından eksiksiz olarak doldurulup, işlenmek üzere uygulamaya gönderildiğini farz edelim. Veri kontrolü başarıyla yapıldıktan sonra kod 10.9 da yer alan processSubmit() metodundaki else kod bloğu devreye girecektir. Bu kod bloğunda processRental() metodu yer almaktadır. Bu metot bünyesinde de service.rentACar(rental) şeklinde bir satır yer almaktadır. Service @Inject anotasyonu yardımı ile RentalController sınıfına enjekte edilmiş RentalService veri tipinde bir nesnedir. RentalServiceImpl ismini taşıyan ve RentalService interface sınıfını implemente eden sınıf kod 10.18 de yer almaktadır.

```
Kod 10.18 - RentalServiceImpl
```

```
package com.kurumsaljava.spring.service;

import javax.inject.Inject;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.
            Transactional;
import com.kurumsaljava.spring.dao.CarRepository;
import com.kurumsaljava.spring.dao.CustomerRepository;
import com.kurumsaljava.spring.dao.RentalRepository;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;

@Service
public class RentalServiceImpl implements RentalService {

    @Inject
    private CustomerRepository customerRepository;
    @Inject
    private RentalRepository rentalRepository;
    @Inject
    private CarRepository carRepository;

    public RentalServiceImpl() {
    }

    @Transactional
    @Override
    public Rental rentACar(final Rental rental) {

        final Customer dbCustomer =
            customerRepository.getCustomerByName(
                    rental.getCustomer().getName());

        if (dbCustomer == null) {
            customerRepository.save(rental.getCustomer());
        }

        final Car car = carRepository.findCarById(1);

        rental.setCar(car);
        rental.setRented(true);
        rental.setCustomer(rental.getCustomer());
        rentalRepository.save(rental);
        return rental;
    }

    public void setCustomerRepository(
            final CustomerRepository customerRepository) {
```

```
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(
            final RentalRepository rentalRepository) {
        this.rentalRepository = rentalRepository;
    }

    public void setCarRepository(
            final CarRepository carRepository) {
        this.carRepository = carRepository;
    }
}
```

RentalServiceImpl uygulamanın servis katmanında yer alan bir sınıftır (bknz. resim 10.5). @Service anotasyonu aracılığı ile bir Spring bean haline gelir. Spring konfigürasyon dosyasında context:component-scan konfigürasyon elementi kullanıldığı taktirde Service, @Component ve @Repository gibi anotasyonu taşıyan sınıflar Spring tarafından @Inject anotasyonu ile talep edilen sınıflara enjekte edilir. Gösterim katmanında yer alan RentalController sınıfı kendisine enjekte edilen RentalService nesnesi aracılığı ile araç kiralama işlemini gerçekleştirmektedir. Görüldüğü gibi Spring MVC kendi bünyesinde Spring dependency injection mekanizmalarını kullanabilmektedir.

Kod 10.18 de yer alan RentalServiceImpl sınıfına yine @Inject anotasyonu kullanılarak veri katmanında yer alan repository sınıfları enjekte edilmektedir. Kullanılan repository sınıflarından birisi olan CustomerRepository sınıfının JPA implementasyonu kod 10.19 da yer almaktadır.

```
Kod 10.19 - JpaCustomerRepositoryImpl

package com.kurumsaljava.spring.dao.impl;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
import com.kurumsaljava.spring.dao.CustomerRepository;
import com.kurumsaljava.spring.domain.Customer;

@Repository
public class JpaCustomerRepositoryImpl implements
                CustomerRepository {

    @PersistenceContext
    private EntityManager entityManager;
```

```
    @Override
    public Customer getCustomerByName(final String name) {

        Customer result = null;
        try {
            result = (Customer) entityManager.
            createQuery("from Customer c where c.name=:name")
                    .setParameter("name", name)
                    .getSingleResult();

        } catch (final NoResultException e) {
        }
        return result;
    }

    @Override
    public void save(final Customer customer) {
        entityManager.persist(customer);
    }
}
```

Spring MVC ile Çoklu Konfigürasyon Kullanımı

Bir Spring MVC uygulaması için gerekli Spring konfigürasyonu birden fazla konfigürasyon dosyasına dağıtılabilir. Bu uygulama için gerekli konfigürasyonun sadeleşmesini ve daha anlaşılır hale gelmesini sağlayacaktır. Örneğin uygulamanın tüm konfigürasyonu gösterim katmanı için mvc-config.xml, servis katmanı için service-config.xml, veri katmanı için persistence-config.xml ismini taşıyan, birbirinden bağımsız konfigürasyon dosyalarından oluşabilir.

Gösterim katmanını konfigüre etmek için oluşturduğumuz mvc-config.xml isimli konfigürasyon dosyasının DispatcherServlet tarafından yüklenmesini sağlamak için kod 10.20 de yer alan servlet tanımlamasını yapmıştık.

```
Kod 10.20 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
    <servlet-class>org.springframework.web.servlet.
            DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
```

```
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Birden fazla Spring konfigürasyon dosyasını yüklemek için ContextLoaderListener sınıfı kullanılmaktadır. web.xml dosyasında bu sınıfı ihtiva eden bir listener tanımlaması kod 10.21 de yer almaktadır.

```
Kod 10.21 - web.xml

<listener>
    <listener-class>
        org.springframework.web.context.
                    ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener sınıfının istenilen konfigürasyon dosyalarını yükleyebilmesi için context-param elementi ile bu dosyalardan oluşan bir listenin tanımlanması gerekmektedir. Böyle bir tanımlama kod 10.22 de yer almaktadır.

```
Kod 10.22 - web.xml

<context-param>
    <param-name>
        contextConfigLocation
    </param-name>
    <param-value>
        /WEB-INF/service-config.xml
        /WEB-INF/persistence-config.xml
        classpath:other-config.xml
    </param-value>
</context-param>
```

Kod 10.22 de yer alan tanımlama ile WEB-INF dizinde yer alan service-config.xml ve persistence-config.xml konfigürasyon dosyaları, ayrıca classpath içinde yer alan other-config.xml dosyası ContextLoaderListener tarafından yüklenecektir. Context-param ile bir konfigürasyon dosyası listesi oluşturulmadığı taktirde ContextLoaderListener /WEB-INF/applicationContext.xml lokasyonundaki konfigürasyon dosyasını yüklemeye çalışacaktır. /WEB-INF/applicationContext.xml konfigürasyon dosyasında import komutu kullanılarak ta adı geçen diğer konfigürasyon

dosyaları yüklenebilir.

@RequestParam Anotasyonu Kullanımı

@RequestParam anotasyonu kullanıcı isteği ile gelen parametrelerin metot parametrelerine atanması için kullanılmaktadır. Kod 10.23 de @RequestParam anotasyonunu kullanan redirectToPage() metodu yer almaktadır. @RequestParam("page") final String page ile pageName değişkenine bir kullanıcı isteği parametresi olan (request parameter) page'in degeri atanmaktadır. redirectToPage() metodu kullanıcının page parametresi ile tayin ettiği sayfanın gösterilmesini sağlamaktadır.

```
Kod 10.23 - RentalController

@RequestMapping(value = "/redirect", method =
            RequestMethod.GET)
public String redirectToPage(@RequestParam("page")
        final String pageName, final ModelMap model) {
    return pageName;
}
```

Kod 10.23 de yer alan metodun işlem yapabilmesi için uygulamanın http://localhost/redirect?page=list şeklinde çağrılması gerekmektedir. ?page=list şeklindeki ifade page ismini taşıyan bir kullanıcı isteği parametresi oluşturur. Bu parametrenin degeri list dir. WEB-INF dizininde list.jsp ismini taşıyan bir JSP sayfası bulunması durumunda, uygulama redirectToPage() metodu aracılığı ile bu sayfanın gösterilmesini sağlayacaktır.

redirectToPage() metodu kullanıcı isteği bünyesinde page parametresinin mevcudiyeti durumunda işlem görür. Eğer page parametresinin eksik olduğu kullanıcı isteklerinde de redirectToPage() metodunun işlem görmesi bekleniyorsa, @RequestParam(value = "page", required = false) şeklinde bir tanımlama yapılması gerekmektedir.

@PathVariable Anotasyonu Kullanımı

@PathVariable anotasyonu aracılığı ile kullanıcı isteğini temsil eden web adresin belli parçalarına erişmek ve bunları parametre olarak değerlendirmek mümkündür. Kod 10.24 de yer alan örnekte web adresin bir parçası olan userid parametresi yardımı ile bir kullanıcının kiraladığı araçların listesi

oluşturulmaktadır. Bu web adresi http://localhost/list/1000 ya da http://localhost/list/200 şeklinde olabilir. 1000 ya da 200 rakamları kullanıcının veri tabanındaki kayıt anahtarıdır (primary key). @PathVariable anotasyonu bu değerin web adresinden edinilerek userid isimli metot parametresine eşitlendikten sonra metot gövdesinde kullanılmasını mümkün kılmaktadır.

```
Kod 10.24 - RentalController

@RequestMapping(value="/list/{userid}", method =
        RequestMethod.GET)
public String listRentals(@PathVariable("userid") String userid,
            final ModelMap model) {
    List<Rental> rentanList = rentalService.listRentalsOfUser(
                    userid);
    model.addAttribute("rentalList", rentalList);
    return "list";
}
```

Uygulama tarafından web adresin bir parçasının parametre olarak algılanabilmesi için bu adresin hangi kısmının parametre olduğunun @RequestMapping anotasyonu ile tanımlanması gerekmektedir. Kod 10.24 de yer alan örnekte /list/{userid} ile bu tanımlama yapılmaktadır.

@RequestMapping(value="/list/{userid}" şeklinde bir tanımlama yapıldığı taktirde @PathVariable anotasyonunun parametre ismini taşıması gerekli değildir. Bu sebepten dolayı kod 10.24 de yer alan listRentals(@PathVariable("userid") String userid, final ModelMap model) metot tanımlaması listRentals(@PathVariable String userid, final ModelMap model) olarak yazılabilir.

```
Kod 10.25 - RentalController

@RequestMapping(value="/user/{userid}/rental/{rentalid}", method =
            RequestMethod.GET)
public String listRentals(@PathVariable("userid") String userid,
                    @PathVariable("rentalid") String rentalid,
            final ModelMap model) {
    ...
}
```

Kod 10.25 de görüldüğü gibi metot parametrelerini oluştururken birden fazla @PathVariable kullanmak mümkündür. Örneğin 50 numaralı müşterinin 1 numarasını taşıyan araç kiralama işlemine ulaşmak için /user/50/rental/1

şeklinde bir web adresi kullanılabilir. listRentals() metodu bünyesinde userid (50) ve rentalid (1) değişkenleri ile web adresinde yer alan parametre değerlerine erişmek mümkündür.

@PathVariable ile int, long, Date gibi herhangi bir veri tipi kullanılabilir. Spring otomatik olarak veri tipi dönüşümünü sağlamaktadır.

Spring MVC Tarafından Tüketilebilecek Veri Türleri

Bir Spring MVC uygulaması String, XML, JSON, Byte gibi değişik türdeki verileri işleyecek şekilde yapılandırılabilir. Uygulamanın kullanılış tarzına ve kullanıcı tipine göre işlenen veri türü değişiklik gösterebilir. İncelediğimiz araç kiralama formu örneğinde kullandığımız veri türü String veri tipinde idi. Bu veriler HttpServletRequest sınıfı aracılığı ile uygulamaya taşındı.

Seçilen veri türünü tespit etmek için uygulama HttpServletRequest nesnesinde yer alan Content-Type başlık (request header) parametresinin değerini inceler. Form örneğinde bu parametrenin sahip olduğu değer application/x-www-form-urlencoded şeklindedir. XML veri türü için bu değer application/xml ya da text/xml, Json için application/json şeklindedir. Görüldüğü gibi veri türünü kullanıcı (client) tayin etmektedir.

Bir Spring MVC uygulamasında @RequestMapping anotasyonu kullanılarak işlenmek istenen veri türü tayin edilmektedir. Bunun yanı sıra RequestBody anotasyonu yardımı ile kullanıcı tarafından uygulamaya gönderilen verinin bir metot parametresine eşlenme işlemi gerçekleştirilir. Kod 10.26 da yer alan örnekte JSON olarak gönderilen veri @RequestBody anotasyonu ile bir Rental nesnesine dönüştürülmektedir. Sadece Content-Type parametresinin application/json değerini taşıdığı durumlarda kod 10.26 da yer alan addRental() metodu devreye girmektedir. Bunun kontrolü @RequestMapping anotasyonunun consumes elementi ile yapılmaktadır.

```
Kod 10.26 - RentalController

@RequestMapping(value="/rental", method =
        RequestMethod.POST consumes="application/json")
public String addRental(final @RequestBody Rental rental,
            final ModelMap model) {
    ...
}
```

HttpServletRequest nesnesinde yer alan verilerin istenilen türde bir alan nenesine dönüştürülebilmesi için HttpMessageConverter sınıfı kullanılmaktadır. Bu sınıf kullanıcı verilerinin bir alan nesnesine, alan nesnelerinin de kullanıcıya gönderilen cevaba (HttpServletResponse) dönüştürülmesini mümkün kılmaktadır. Kullanılabilecek bazı HttpMessageConverters implementasyonları şöyledir:

- ***ByteArrayHttpMessageConverter*** - byte array dönüşümü
- ***StringHttpMessageConverter*** - string dönüşümü
- ***FormHttpMessageConverter*** - form <-> MultiValueMap<String, String> dönüşümü
- ***MarshallingHttpMessageConverter*** - org.springframework.oxm paketi yardımı ile Java<->XML dönüşümü

RequestBody tarafından gerekli dönüşümün yapılabilmesi için konfigürasyon dosyasında RequestMappingHandlerAdapter sınıfının Spring bean olarak tanımlanması gerekmektedir. Kod 10.27 de yer alan örnekte String ve XML bazlı veri türlerinin dönüşümü için gerekli RequestMappingHandlerAdapter konfigürasyonu yer almaktadır.

```
Kod 10.27 - applicationContext.xml

<bean
     class="org.springframework.web.servlet.mvc.method.
                annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
          <util:list id="beanList">
            <ref bean="stringConverter"/>
            <ref bean="xmlConverter"/>
          </util:list>
    </property
</bean>

<bean id="stringConverter"
   class="org.springframework.http.converter.
            StringHttpMessageConverter"/>

<bean id="xmlConverter"
   class="org.springframework.http.converter.xml.
        MarshallingHttpMessageConverter">
      <property name="marshaller" ref="castorMarshaller" />
      <property name="unmarshaller" ref="castorMarshaller" />
</bean>
```

```
<bean id="castorMarshaller" class="org.springframework.oxm.castor.
            CastorMarshaller"/>
```

Spring MVC Tarafından Oluşturulabilecek Veri Türleri

Bir Spring MVC uygulaması bir önceki bölümde ismi geçen veri türlerini tüketebildiği gibi, kullanıcılarına bu veri türlerinden oluşan cevaplar da sunabilir. Kod 10.28 de yer alan örnekte kullanıcıya bir Rental nesnesi Json formatında sunulmaktadır. Sunulan veri türünü belirlemek için @RequestMapping anotasyonunun produces elementi kullanılmaktadır.

```
Kod 10.28 - RentalController

@RequestMapping(value="/user/{userid}/rental/{rentalid}",
          method = RequestMethod.GET produces="application/json")
public Rental listRentals(@PathVariable("userid") String userid,
                      @PathVariable("rentalid") String rentalid,
                                      final ModelMap model) {
    ...
}
```

İç ve Dış Yönlendirme

JSP sayfalarına (view) yönlendirme işlemi controller sınıflarının bir JSP sayfasının ismini geri vermesi ile sağlanmaktadır. Kod 10.24 yer alan örnekte metodun geri verdiği değer list şeklindedir. Spring MVC InternalResourceViewResolver yardımı ile list.jsp ismini taşıyan JSP sayfasını lokalize ederek akış kontrolünü bu sayfaya bırakmaktadır. Bu yönlendirme Servlet API'sinde yer alan RequestDispatcher.forward() ya da RequestDispatcher.include() kullanılarak yapılmaktadır. Bu işlem iç yönlendirme (internal redirect) olarak isimlendirilmektedir, çünkü yönlendirme uygulama bünyesinde yapılmaktadır.

Bazı şartlar altında dış yönlendirme (external direct) gerekli olabilir. Örneğin bir controller sınıfından başka bir controller sınıfına iç yönlendirme yapıldığı zaman, birinci controller sınıfına kullanıcı tarafından gönderilmiş tüm veriler ikinci controller sınıfına da yönlendirilir. Bu beklentilerin dışında bir veri akışı olduğunda ikinci controller sınıfı için kafa karıştırıcı olabilir. Bunu engellemek

için ikinci controller sınıfına dışsal yönlendirme yapılabilir.

Kod 10.29 da yer alan örnekte form için gerekli işlem yapıldıktan sonra redirect:rental/done ile bu adresten sorumlu controller sınıfı için dış yönlendirme yapılmaktadır.

```
Kod 10.29 - RentalController

@RequestMapping(method = RequestMethod.POST)
public String processSubmit(final ModelMap model,
        @ModelAttribute(RENTAL_FORM) @Valid final Rental rental,
        final BindingResult result) {
    if (result.hasErrors()) {
        buildCarList(model);
        return RENTAL_VIEW;
    } else {
        processRental(rental);
        return "redirect:rental/done";
    }
}
```

rental/done adresinden DoneController sınıfı sorumlu olduğundan, done() metodu bünyesinde done.jsp sayfasına iç yönlendirme yapılmaktadır. Dış ve iç yönlendirme gerçekleştikten sonra kullanıcının web tarayıcısının adres çubuğunda http://localhost/rental/done adresi yer alır.

```
Kod 10.30 - DoneController

@Controller
public class DoneController {

    @RequestMapping(value = "/rental/done")
    public String done(final ModelMap model) {
        return "done";
    }
}
```

Dış yönlendirme için ayrıca RedirectView sınıfı kullanılabilir. Kod 10.30.1 de yer alan örnekte /rental/list sayfasına dış yönlendirme yapılmaktadır. DispatcherServlet RedirectView ile karşılaştığı durumlarda HttpServletResponse.sendRedirect() aracılığı ile ismi geçen sayfaya dış yönlendirme yapar.

```
Kod 10.30.1 - RentalController
```

```
@RequestMapping(method = RequestMethod.GET)
public RedirectView redirect(final ModelMap model)
            throws IOException {
        RedirectView redirectView =
            new RedirectView("rental/list");
        redirectView.addStaticAttribute("errorMessage",
                    "error1");
        return redirectView;
}
```

addStaticAttribute() metodu ile RedirectView nesnesine eklenen tüm parametreler dış yönlendirme yapılan adresin parametreleri hale gelir. Kod 10.30.1 de yer alan örnekte dış yönlendirme adresi /rental/list?errorMessage?error1 şeklindedir. /rental/list adresinden sorumlu olan controller sınıfı metodu @RequestParam anotasyonu ile bu parametrelere erişebilir. Kod 10.30.2 de yer alan getList() metodu rental/list adresinden sorumlu olduğu için @RequestParam("errorMessage") aracılığı ile errorMessage parametresini metot parametresi olarak tanımlamaktadır. Böylece dış yönlendirme ile gönderilen errorMessage parametresi getList() bünyesinde kullanılır hale gelmektedir.

```
Kod 10.30.2 - RentalController

@RequestMapping(value = "rental/list")
public String getList(
                 @RequestParam("errorMessage") String
                           errorMessage, Model model) {
    model.addAttribute("errorMessage", errorMessage);
    return "list";
}
```

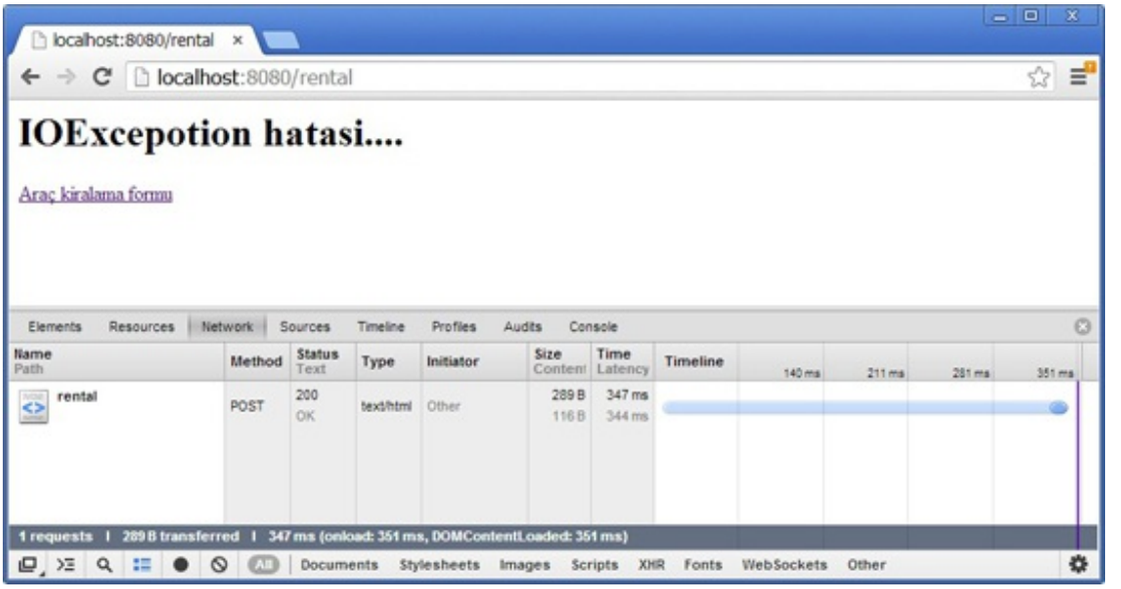
Hata Yönetimi

Bir Spring MVC uygulaması bünyesinde işlem esnasında çeşitli hatalar (exception) oluşabilir. SimpleMappingExceptionResolver sınıfı kullanılarak her hata için bir gösterim sayfası tanımlanabilir.

```
Kod 10.31 - applicationContext.xml

<bean id="exceptionResolver"
      class="org.springframework.web.servlet.handler.
                        SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
      <map>
```

```
        <entry key="java.io.IOException"
               value="io-exception" />
        <entry key="java.lang.Exception"
               value="generic-error" />
      </map>
    </property>
</bean>
```

Kod 10.31 de yer alan örnekte IOException oluşması durumunda io-exception (resim 10.10) view elementine yönlendirme yapılmaktadır. Oluşan diğer hatalar için yönlendirme adresi generic-error şeklinde olacaktır.



Resim 10.10

Resim 10.10 da görüldüğü gibi SimpleMappingExceptionResolver oluşan hata türüne göre gerekli sayfaya yönlendirme yapmaktadır, lakin HTTP statü kodu oluşan hata türünü yansıtmamaktadır. Statü kodunun 200 değerine sahip olması, kullanıcı isteğinin uygulama tarafından hatasız cevaplandığı anlamına gelmektedir. Ama biliyoruz ki bir IOException hatası oluşmuştur.

Bazı kullanıcılar (client) HTTP statü kodlarını inceleyerek oluşan hatalardan haberdar olma isteğinde olabilir. Bu durumda uygulama sunucusunun belli bir hata için belli bir sayfaya yönlendirme yapması yeterli olmayacaktır. Oluşan hata türüne göre kullanıcıya gönderilen cevap içinde HTTP statü kodunun tanımlanmış olması gerekir. Oluşan hata türüne göre HTTP statü kodunu atamak için DefaultHandlerExceptionResolver sınıfı kullanılmaktadır.

DefaultHandlerExceptionResolver sınıfı uygulama bünyesinde oluşan hata türüne göre statü kodunu atar. Bu sınıf mvc isim alanında yer aldığı için

konfigürasyon dosyasında tanımlanmasına gerek yoktur ve mvc isim alanının yüklenmesiyle aktif hale gelir. Aşağıda yer alan listede bazı hata türleri ve DefaultHandlerExceptionResolver tarafından atanan statü kodları yer almaktadır.

```
Liste 10.1:

BindException - 400 (Bad Request)
ConversionNotSupportedException - 500 (Internal Server Error)
HttpMediaTypeNotAcceptableException - 406 (Not Acceptable)
HttpMediaTypeNotSupportedException - 415 (Unsupported Media Type)
HttpMessageNotReadableException - 400 (Bad Request)
HttpMessageNotWritableException - 500 (Internal Server Error)
HttpRequestMethodNotSupportedException - 405 (Method Not Allowed)
MethodArgumentNotValidException - 400 (Bad Request)
MissingServletRequestParameterException - 400 (Bad Request)
MissingServletRequestPartException - 400 (Bad Request)
NoSuchRequestHandlingMethodException - 404 (Not Found)
TypeMismatchException - 400 (Bad Request)
```

DefaultHandlerExceptionResolver sınıfı hata durumunda sadece statü kodunu atamakla yetinir. Statü kodu haricinde kullanıcıya herhangi bir bilgi gönderilmez. Kullanıcının oluşan hatadan detaylı olarak haberdar olabilmesi için sadece statü kodunun atanması yeterli değildir. Oluşan hatanın çıktısı (stacktrace) kullanıcıya hata mesajı (error response) olarak gönderilmelidir. Bu amaçla @ExceptionHandler anotasyonu kullanılabilir.

Kod 10.32 yer alan örnekte processSubmit bünyesinde IOException tipinde bir hata oluşması durumunda handleIOException() metodu devreye girerek bir RedirectView nesnesi oluşturur. Bu nesne bünyesinde dış yönlendirme yapılacak sayfanın adresi (rental/ioexception) ve hata mesajı (errorMessage) yer almaktadır. Spring MVC /rental/ioexception için gerekli yönlendirmeyi gerçekleştirir. /rental/ioexception sayfası için bir istek gerçekleştiği için bu istek tekrar RentalController sınıfına yönlendirilir, çünkü RentalController bünyesinde @RequestMapping anotasyonu kullanılarak ioexception adresinden sorumlu errorRedirectPage() isimli bir metot tanımlanmıştır. Bu metot bünyesinde model nesnesine istek parametresi (request parameter) olarak gelen hata mesajı (errorMessage) yerleştirilir ve io-exception yani io-exception.jsp sayfasına hata gösterimi için iç yönlendirme yapılır.

```
Kod 10.32 - RentalController

@Controller
```

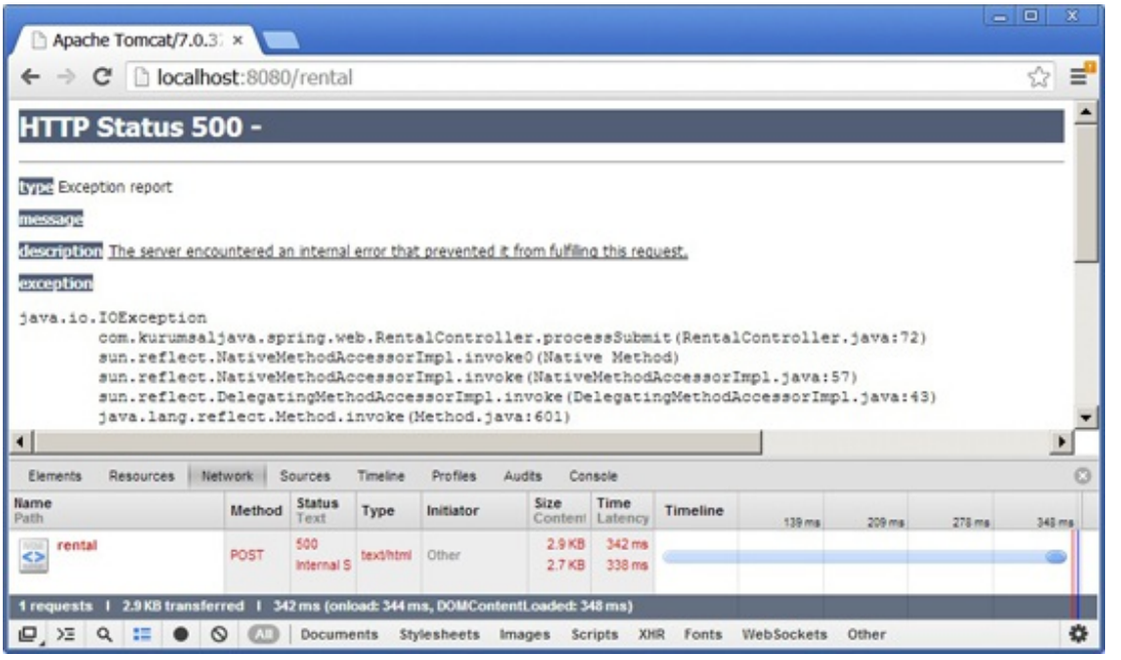
```
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(final ModelMap model,
        @ModelAttribute(RENTAL_FORM) @Valid final Rental rental,
            final BindingResult result) {
        ...
    }

    @ExceptionHandler(IOException.class)
    public RedirectView handleIOException(IOException ex)
            throws IOException {
        RedirectView redirectView =
            new RedirectView("rental/ioexception");
        redirectView.addStaticAttribute("errorMessage",
                ex.getMessage());
        return redirectView;
    }

    @RequestMapping(value = "ioexception")
    public String errorRedirectPage(
                     @RequestParam("errorMessage")
                        String errorMessage, Model model) {
        model.addAttribute("errorMessage", errorMessage);
        return "io-exception";
    }
}
```

@ExceptionHandler anotasyonuna ek olarak HTTP statü kodu atanmak istendiğinde @ResponseStatus anotasyonu kullanılabilir. Kod 10.32.1 de yer alan örnekte IOException türünde bir hatanın oluşması durumunda rental/ioexception adresine dış yönlendirme yapılmakta ve HTTP statü kodu 500 (INTERNAL*SERVER*ERROR) olarak atanmaktadır.

```
Kod 10.32.1 - RentalController

@ExceptionHandler(IOException.class)
@ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
public RedirectView handleIOException(IOException ex)
            throws IOException {
    RedirectView redirectView =
            new RedirectView("rental/ioexception");
    redirectView.addStaticAttribute("errorMessage",
            ex.getMessage());
    return redirectView;
}
```

Genel Hata Sayfası Konfigürasyonu

Uygulama tarafından aktif olarak herhangi bir hata yönetimi uygulanmıyorsa, oluşan bir hata kullanıcıya resim 10.11 deki gibi yansıyacaktır.



Resim 10.11

Resim 10.11 de yer alan hata sayfası geliştiriciler için önemli bilgiler ihtiva etmekle birlikte son kullanıcılar için kafa karıştırıcı türdedir. Son kullanıcılar için genel bir hata sayfasının tanımlanması ve oluşan hata hakkında bu hata sayfasında bilgi verilmesi kullanıcının daha az irite olmasını sağlayacaktır.

Oluşan herhangi bir hataya işaret etmek için error-page elementi kullanılarak web.xml bünyesinde genel bir hata sayfası tanımlanabilir. Uygulama sunucusu statü kodunun bir hata değerini ihtiva etmesi durumunda error-page ile tanımlı olan sayfaya yönlendirme yapacaktır. Kod 10.33 de genel bir hata sayfasının tanımlanması yer almaktadır.

```
Kod 10.33 - web.xml

<error-page>
    <location>/error</location>
</error-page>
```

Kod 10.34 de yer alan ErrorController sınıfı /error adresinden sorumludur ve uygulama sunucusunun /error adresine yönlendirme yapmasıyla birlikte aktif

hale gelir. handle() metodu bünyesinde hata verilerini ihtiva eden bir model nesnesi oluşturulduktan sonra, bu verilerin gösterimi amacıyla errorPage.jsp sayfasına iç yönlendirme yapılır. @ResponseBody anotasyonununu on ikinci bölümde inceleyeceğiz. @ResponseBody handle() metodunun geriye verdiği değerin HTTP cevap gövdesine (response body) eklenmesini sağlamaktadır.

```
Kod 10.34 - ErrorController

@Controller
public class ErrorController {

    @RequestMapping(value="/error")
    @ResponseBody
    public String handle(ModelMap model,
            HttpServletRequest request) {
        Map<String, Object> map =
            new HashMap<String, Object>();
        map.put("status",
            request.getAttribute("
                    javax.servlet.error.status_code"));
        map.put("reason", request.getAttribute("
                    javax.servlet.error.message"));
        model.put("errorMap", map);
        return "errorPage";
    }
}
```

Spring MVC uygulamalarında hata yönetimi için daha detaylı bilgiyi kitabın on ikinci, REST uygulamalarında hata yönetimi bölümünde bulabilirsiniz. REST uygulamaları Spring MVC çatısı kullanılarak geliştirildiğinden, kullanılan hata yönetimi mekanizmaları aynıdır.

10. Bölüm Soruları

10.1 Spring MVC'nin dayandığı standart Java API hangisidir?

10.2 MVC'yi oluşturan harflerin karşılığı nedir?

10.3 Handler Mapping ne amaçla kullanılır?

10.4 Spring MVC üç katmanlı mimaride hangi katmanı oluşturmak için kullanılır?

10.5 Controller ile view arasında veri taşımak için kullanılan yapı hangisidir?

10.6 Gösterimi yapan view elementlerini lokalize etmek için kullanılan yapının ismi nedir?

10.7 @RequestParam ne amaçla kullanılır?

11. Bölüm

Spring Security

İnternetin gelişmesi ve web uygulamalarının artması ile karşı karşıya kaldığımız yeni bir tehlike bulunuyor: verilerin kanunsuz işler yapmaya meyilli şahıslar tarafından veri sahiplerinin izni olmadan temin edilerek, kullanılmaları. Hırlı olmayan bu şahısların ilgi alanlarına kredi kartı ve banka hesap numaraları, bilumum şifreler ve kullanıcı profili bilgileri gibi veriler girmektedir. Bu tür aktiviteleri kısaca dijital hırsızlık olarak isimlendirebiliriz. Bu çağımızın en büyük problemlerinden biri haline gelmiştir, çünkü interneti kullanan insan sayısı arttıkça, bu insanların internette bıraktıkları dijital izlerin sayısı her geçen gün katlanarak artmakta ve buna doğru orantılı olarak bahsettiğim karanlık şahısların dijital verilere olan ilgileri de büyümektedir.

Bahsettiğim problemin önüne geçmek için yazılım sistemlerinin geniş kapsamlı güvenlik özellikleri ile donatılmaları gerekmektedir. Bunun için mevcut konseptler bulunmaktadır. Örneğin uygulamayı şifre sahibi belli bir kullanıcı kesiminin kullanımına açmak, belli rollere sahip kullanıcıların belli aksiyonları yapmalarına izin vermek vb. Bu konseptler yıllardan beli bilinen ve uygulanan konseptler olmakla birlikte, yer, yer yeterli olmadıklarına, uygulamaların güvenlik yetersizliklerinden dolayı açık verdiklerine şahit olmaktayız. Bu bize uygulamanın sahip olması gereken güvenlik özelliğinin uygulamanın tasarlama ve geliştirme süreçlerinde devamlı göz önünde bulundurularak, yapılan her değişikliğe ayak uydurabilecek şekilde adapte ve kontrol edilmesi gerekliliğini göstermektedir. Bunun yanı sıra güvenlik konseptlerini her yönüyle destekleme kabiliyetine sahip güvenlik çatılarına ihtiyaç duyulmaktadır. Spring bünyesinde Spring Security ismini taşıyan böyle bir güvenlik çatısı mevcuttur. Bu bölümde Spring Security çatısı ve Spring uygulamalarının bu güvenlik çatısı yardımı ile nasıl korunabileceğini yakından inceleyeceğiz.

Güvenlik Terminolojisi

Spring Security çatısını incelemeden önce, bu çatı bünyesinde kullanılan bazı terimlere göz atmamızda fayda var. Bu terimlerin tanınması, güvenlik çatısının kullanımını kolaylaştıracaktır. Bu konuda önce çıkan terimleri şu şekilde tanımlayabiliriz:

- ***Principal*** - İşlem yapan kullanıcı, aygıt (device) ya da bir sistem.
- ***Authentication*** - Principal tarafından kullanılan kimlik bilgilerini doğrulama işlemi.
- ***Authentication Interface*** - Uygulama sunucusunun sunduğu authentication mekanizması implementasyonunu oluşturmak için

kullanılan arayüz.

- ***Authentication Provider*** - Veri tabanı, Ldap ya da başka bir veri tabanını türünü kullanarak authentication işlemini gerçekleştiren implementasyon.
- ***Authorization*** - Principal'in belli bir işlem yapma izninin olup, olmadığının kontrolü.
- ***Secured Item*** - Korunun kaynak, örneğin veri tabanındaki müşteri bilgileri.

Bir Uygulamanın Güvenlik İhtiyaçları

Tipik bir web uygulamasında güvenlik ihtiyacının doğduğu dört alan bulunmaktadır. Bunlar kimlik bilgilerinin doğrulanması (authentication), web sayfalarının güvenliği (web request security), servis katmanındaki sınıfların ve alan nesnelerinin güvenliğidir (domain object security).

Kimlik bilgileri doğrulama işlemi (authentication) için değişik mekanizmalar mevcuttur. Bunların başında basic, digest, form ve x.509 tabanlı kimlik doğrulama mekanizmaları gelir. Çoğu uygulama bünyesinde kimlik doğrulama işlemi için kullanıcı ismi ve şifresi yeterli olmaktadır. Bu bilgilere user credential ismi verilmektedir. Bu bilgiler veri tabanı, LDAP ya da hafıza bünyesinde (in-memory) tutulur ve authentication işlemi esnasında kullanılır.

Kimlik doğrulama işlemi olumlu olarak yapıldıktan sonra, koruma altında bulunan kaynaklara erişimin belli kriterler doğrultusunda kontrol edilmesi gerekmektedir. Bu amaçla kullanıcılar için roller ve haklar tanımlanır. Authorization işlemi authentication işleminin üzerine inşa edilmiş olan ve bu rol ve hakların kontrol edildiği mekanizmadır.

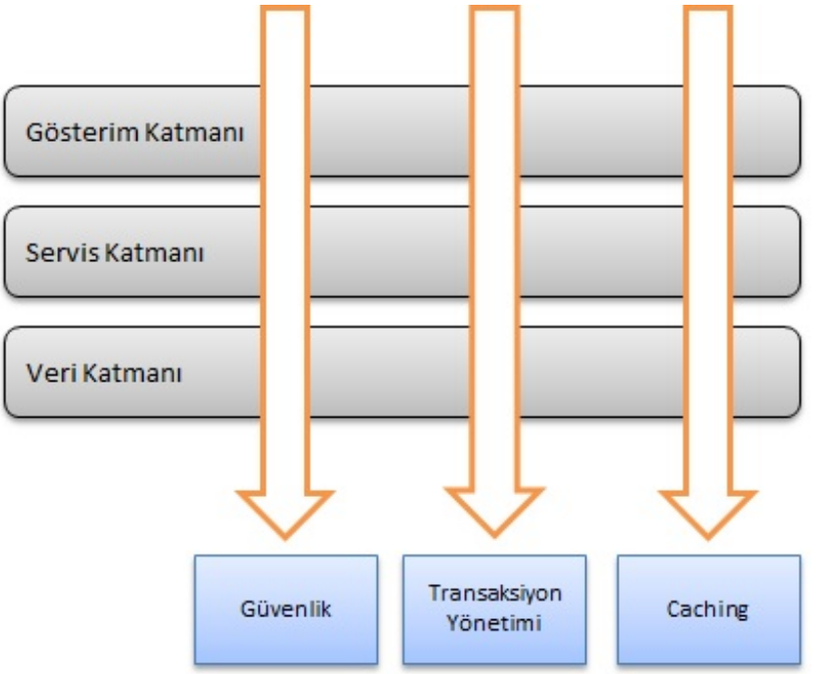
Koruma altında yer alan kaynakların başında veri tabanında yer alan veriler gelir. Bu verilere erişim katmanlı bir mimaride servis ve veri katmanı üzerinde gerçekleşir. Verilerin korunabilmesi için servis ya da veri katmanında yer alan metotların kullanıcının sahip olduğu rollere göre veri erişimini düzenlemesi gerekir. Bu işleme metot güvenliği ismi verilmektedir. Kullanıcı sadece sahip olduğu rollerin izin verildiği metotları koşturabilir.

Metot güvenliği belli bir seviyeye kadar verilerin korumak için kullanılabilecek bir yöntemdir. Sadece rollere bağımlı olarak veri erişimini sağlamak, alan nesnelerinin (domain objects) güvenliği için yeterli olmayabilir. Veri tabanında yer alan veriler alan nesneleri (domain objects) olarak uygulama bünyesinde

kullanılır. Örneğin Order (Siparis) ismini taşıyan bir nesne setini yönetici (admin) rolündeki bir kullanıcı bir engelleme olmadan edinebilirken, sipariş vermiş bir kullanıcının (customer) sadece kendi siparişini ihtiva eden bir Order nesnesine erişmesi doğrudur. Bunu sağlamak amacıyla her alan nesnesi için bir erişim listesinin (ACL - Access Control List) tanımlanması gerekir. Nesnenin erişim listesi kullanılarak hangi kullanıcının bu nesnede yer alan verileri edinebileceği tanımlanır.

Güvenlik Bir Aspekttir

Dokuzuncu bölümde AOP (Aspect Oriented Programming) konseptleri ile tanışmıştık. Resim 11.1 de görüldüğü gibi uygulama güvenliğini bir aspekt olarak düşünebiliriz. Aspekt olarak implemente edilmiş ve uygulamadan soyutlanmış bir güvenlik mekanizması uygulamayı oluşturan parçaların bakımını, geliştirilmesini ve test edilmesini kolaylaştırmaktadır.



Resim 11.1

Spring Security servlet filtreleri ve AOP tekniklerini kullanılarak uygulama güvenlinini sağlamaktadır. Böylece iş mantığı ile güvenlik mekanizmaları birbirlerinden bağımsız olarak geliştirilebilmekte ve birlikte kullanılabilmektedir. Bunun yanı sıra authentication ve authorization mekanizmaları birbirlerinden bağımsızdır. Authentication mekanizmasında yapılan bir değişiklik authorization mekanizmasını etkilememektedir.

Neden Spring Security?

Servlet spesifikasyonunda authentication işleminin nasıl olması gerektiği tanımlanmıştır. Authentication mekanizmasını kullanabilmek için uygulama sunucusuna has ayarların (realms) yapılması ve authentication interface sınıflarının implemente edilmedi gerekmektedir. Bu authentication mekanızmasını kullanılan uygulama sunucusuna bağımlı kılmakta ve uygulamanın başka bir uygulama sunucusuna taşınmasını engellemektedir. Demek oluyor ki Servlet teknolojisi kullanılarak oluşturulan authentication mekanizması taşınabilir (portable) bir çözüm değildir. Spring Security ile bir uygulama için hazırlanan authentication mekanizması değişik uygulama sunucularına taşınabilmektedir. Spring Security bünyesinde yer alan değişik authentication provider implementasyonları uygulamanın değişik ortamlara adaptasyonunu kolaylaştırmaktadır.

Servlet spesifikasyonunda servis katmanı güvenliği konusunda bir tanımlama bulunmamaktadır. Klasik bir Java web projesinde her kullanıcı servis katmanında yer alan her metodu koşturabilir. Bu yazılımcıları bu metotları korumak ve erişimi sınırlamak için controller sınıfları ya da view elementleri içinde kod yazmaya yönlendirmektedir. Böylece iş mantığı ile güvenlik kodu iç, içe geçmekte ve uygulamanın bakımı ve gelişimini zorlaştırmaktadır. Uygulama güvenliği bir aspekttir ve bu şekilde implemente edilmelidir. Spring Security bunu desteklemekte ve servis katmanı güvenliğini metot bazında mümkün kılmaktadır.

Servlet spesifikasyonu alan nesnelerinin (domain object) güvenliği konusunda bir tanımlama ihtiva etmemektedir. Bunun eksikliği yazılımcı için değişik katmanlarda alan nesnesi güvenliğini sağlamak için kod yazma gerekliliğini doğurmaktadır. Spring Security ile alan nesnelerinin sahip olduğu erişim listeleri (ACL - Access List) sayesinde veriye erişim regüle edilebilmektedir.

Basit web uygulamaları için Servlet spesifikasyonu yeterli derecede güvenlik mekanizmaları ihtiva ederken, yazılımcılar bahsetmiş olduğum sebeplerden dolayı alternatif arayışı içindedirler. Spring Security deklaratif güvenlik tanımlama modeli, bağımlılıkların enjekte edilmesi ve AOP tekniklerinin kullanışı gibi nedenlerden dolayı Spring uygulamaları için ideal bir çözümdür.

Spring Security Modülleri

Spring 3 ile Spring Security modülü değişik Jar dosyalarından oluşmaktadır. Şimdi bu modülleri ve işlevlerini tanıyalım.

Core - spring-security-core.jar

Temel Spring Security paketidir ve bünyesinde authentication, metot güvenliği v e Spring Remoting desteği sağlayan sınıflar ihtiva eder. Spring Security modülünü kullanılmak istendiğinde bu paketin projeye dahil edilmesi gerekmektedir.

Remoting - spring-security-remoting.jar

Spring Remoting ile entegrasyonu sağlar. Spring Remoting için kullanıcı sınıf yazılmadığı sürece kullanımı zorunlu değildir.

Web - spring-security-web.jar

Bünyesinde servlet filtreler ve web bazlı güvenlik için gerekli altyapı sınıflarını barındırır. Web bazlı authentication ve URL tabanlı güvenlik için kullanılır.

Config - spring-security-config.jar

Spring konfigürasyon dosyasında security isim alanı kullanıldığında proje dahil edilmesi gereken modüldür. Security isim alanını oluşturmak için kullanılan sınıfları ihtiva eder. Bu sınıflar Spring uygulaması bünyesinde doğrudan kullanılmaz.

LDAP - spring-security-ldap.jar

LDAP authentication için kullanılan pakettir.

ACL - spring-security-acl.jar

Alan nesnelerine erişimi kontrol etmek için kullanılan security sınıflarını ihtiva eder.

CAS - spring-security-cas.jar

CAS (Central Authentication Service) entegrasyonu için kullanılan sınıfları ihtiva eder.

OpenID - spring-security-openid.jar

OpenID bazlı authentication işlemlerini gerçekleştirmek için kullanılan sınıfları ihtiva eder.

Spring Security XML İsim Alanı

Spring'in konfigürasyon işlemlerini kolaylaştırmak için XML isim alanları (namespace) ve bu isim alanlarında tanımlı olan konfigürasyon elementlerinden yararlandığını daha önceki bölümlerde incelemiştik. Spring Security modülü için de böyle bir XML isim alanı bulunmaktadır. Bu isim alanı ve konfigürasyon dosyasına dahil ediliş şekli kod 11.1 de yer almaktadır.

```
Kod 11.1 - security-config.xml

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:security="http://www.springframework.org/schema
                /security"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
        http://www.springframework.org/schema/beans/
                spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/
                spring-security-3.0.xsd">
</beans>
```

Konfigürasyon dosyası bünyesinde xmlns:security elementi ile Spring Security XML isim alanını oluşturduktan sonra, örneğin bu isim alanına ait olan ldap-server elementi ile lokal çalışacak bir LDAP sunucusunu şu şekilde tanımlayabiliriz:

```
<security:ldap-server />
```

Spring Security isim alanında bulunan elementlere security eki olmadan doğrudan isimleri aracılığı ile erişmek için Spring Security isim alanını kullanılan ana (default) isim alanı haline getirmemiz gerekmektedir. Böyle bir tanımlama tarzı kod 11.2 de yer almaktadır.

```
Kod 11.2 - security-config.xml

<beans:beans xmlns="http://www.springframework.org/schema/security"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/
```

```
                spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/
                spring-security-3.0.xsd">
</beans:beans>
```

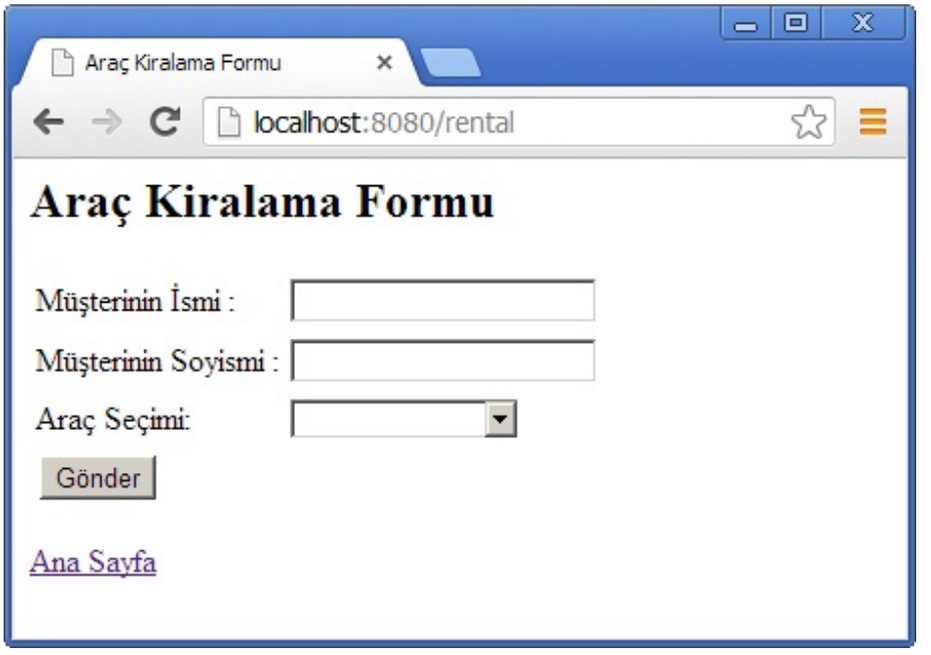
Kod 11.de yer alan tanımlama ile LDAP sunucusunu şu şekilde oluşturabiliriz:

```
<ldap-server />
```

Bu isim alanını kullanabilmek için spring-security-config.jar paketinin projeye dahil edilmiş olması gerekmektedir. Bu Jar paketi bünyesinde XML isim alanının kullanımını mümkün kılan Spring sınıfları bulunmaktadır.

Spring Security İle Web Sayfası Güvenliği

Resim 11.2 de araç kiralama işleminin yapıldığı web sayfası yer almaktadır. Bu sayfa araç kiralama işlemlerini yapan servis personeli için hazırlanmış bir sayfadır, yani sadece servis personeli tarafından kullanılabilir. Bunu sağlayabilmek için sayfaya erişimin kontrol edilmesi gerekmektedir.



Resim 11.2

Web uygulamalarında web sayfalarına olan erişimi kontrol etmek için login mekanizması kullanılabilir. Kullanıcı korunun sayfaya erişmek istediğinde isim ve şifresini gireceği bir login sayfası ile karşılaşır. Kullanıcının girmiş olduğu isim ve şifre kullanılan authentication yöntemine göre sistem tarafından

kontrol edilir. Kullanılan authentication mekanizması olumlu sonuç verdiğinde kullanıcıya erişmek istediği sayfa gösterilir. Aksi taktirde erişim yasaklanır.

Bu tür bir erişim kontrolünü kod 11.3 de yer alan http elementi ile gerçekleştirebiliriz. intercept-url kullanıcılar tarafından erişilmek istenen sayfaları tanımlayan bir şablon (pattern) ihtiva etmektedir. Kod 11.3 de yer alan örnekte kullanılan Ant tarzı şablon / web uygulaması bünyesinde yer alan tüm sayfaları kapsamaktadır. Güvenlik kapsamına giren sayfaları tanımlamak için regex (Regular Expression) şablonları da kullanılabilir.

Intercept-url elementi bünyesinde yer alan access ile kullanıcı rolleri tanımlanmaktadır. Kod 11.3 de yer alan örnekte kullanıcının web uygulaması bünyesinde yer alan herhangi bir sayfaya erişebilmek için ROLE_USER rolüne sahip olması gerekmektedir. Virgül kullanılarak birden fazla rol tanımlanabilmektedir.

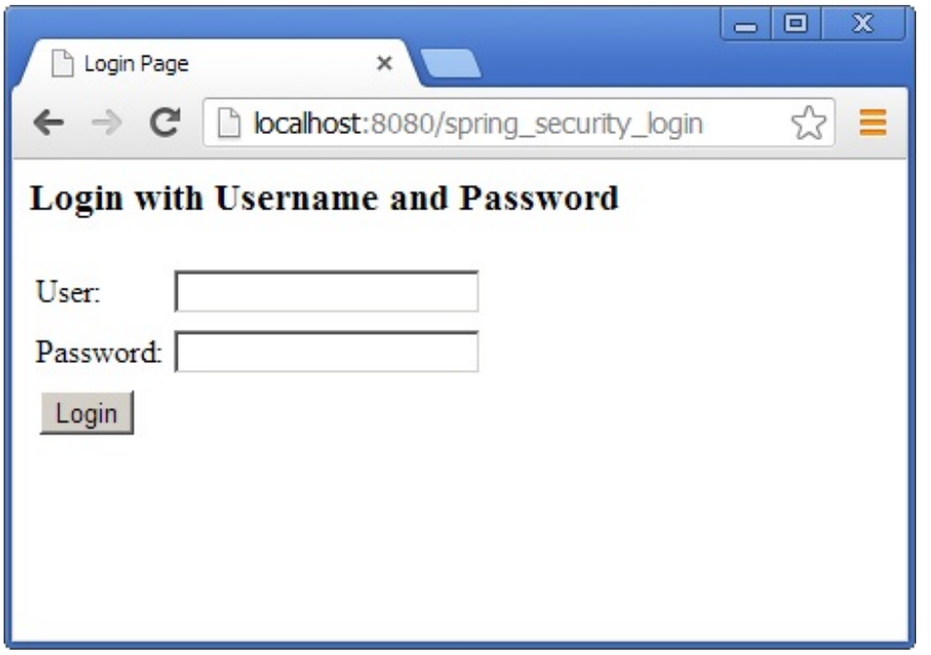
```
Kod 11.3 - security-config.xml

<http>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login default-target-url="/rental" />
</http>
```

Bu konfigürasyon ile http://localhost:8080/rental adresine erişmek istediğimizde, resim 11.3 de yer alan login sayfası ile karşılaşırız. Bu login sayfasını oluşturan kod 11.3 de kullandığımız form-login elementidir. Bu element aracılığı ile bir login sayfası tanımlanmadığında, Spring otomatik olarak resim 11.3 de yer alan login sayfasını oluşturur. Bu sayfanın HTML kodu kod 11.3.1 de yer almaktadır.

```
Kod 11.3.1 - login sayfasi

<html>
<head>
<title>Login</title>
</head>
<body onload='document.f.j_username.focus();'>
    <h3>Login with Username and Password</h3>
    <form name='f' action='/j_spring_security_check'
                method='POST'>
        <table>
            <tr>
                <td>User:</td>
                <td><input type='text' name='j_username'
```

```
                        value='')>
                </td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><input type='password'
                        name='j_password' />
                </td>
            </tr>
            <tr>
                <td colspan='2'>
                <input name="submit" type="submit"
                    value="Login" />
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

form-login bünyesinde kullanılan default-target-url ile login işlemi sonrasında kullanıcıya gösterilecek olan sayfa tanımlanmaktadır. Kod 11.3 de yer alan örnekte olumlu bir authentication işlemi ardından /rental sayfasına yönlendirme yapılacaktır.



Resim 11.3

Kullanıcının isim, şifre ve sahip olduğu rolleri kontrol etmek amacıyla bir authentication yöneticisine ihtiyaç duyulmaktadır. Spring Security isim alanında yer alan authentication-manager elementi ile böyle bir

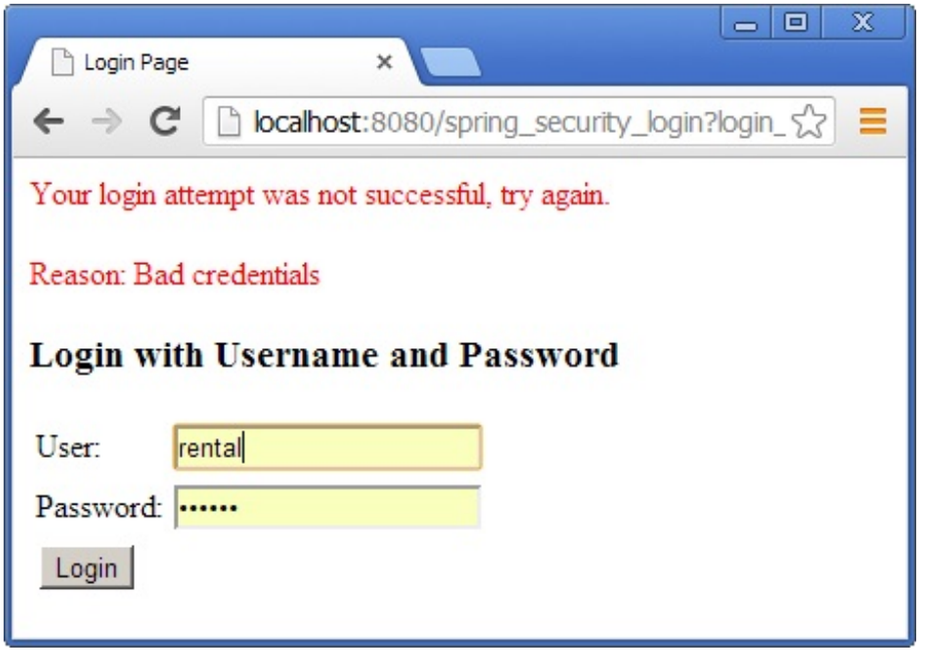
konfigürasyonu yapabiliriz.

```
Kod 11.4 - security-config.xml

<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="rental" password="rental"
                    authorities="ROLE_USER,
                    ROLE_ADMIN" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

Kod 11.4 de yer alan authentication-manager elementi ile bir authentication yöneticisi tanımlanmaktadır. Bu authentication yöneticisi kullanıcı bilgilerine erişmek için bir authentication sunucusu (authentication-provider) kullanmaktadır. Bu authentication sunucusu bünyesinde user-service elementi ile kullanıcı bilgileri tanımlanmaktadır. Kod 11.4 de yer alan örnekte rental isminde bir kullanıcı tanımlanmaktadır. Bu kullanıcının şifresi rental ve sahip olduğu roller ROLE_USER ve ROLE_ADMIN dir.

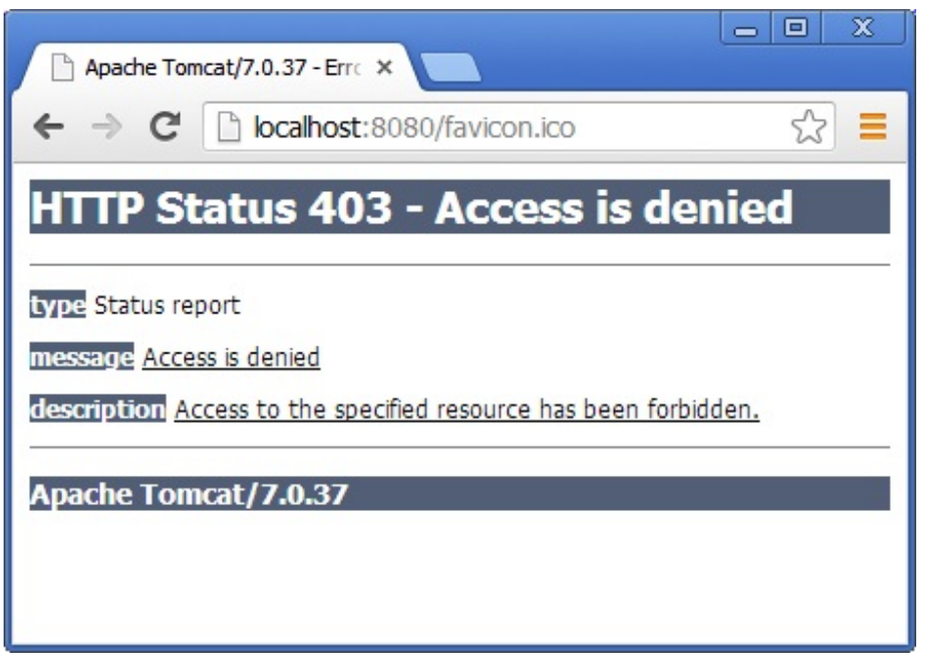
Kod 11.3 e tekrar baktığımızda herhangi bir sayfaya erişim için access=ROLE_USER tanımlaması yapıldığını görmekteyiz. Kullanıcının herhangi bir sayfaya erişebilmesi için ROLE_USER rolüne sahip olması gerekmektedir. Resim 11.3 de yer alan login sayfasına isim ve şifre olarak rental değerini girdiğimizde, uygulama authentication işlemini gerçekleştirdikten sonra /rental sayfasına yönlendirme yapmadan önce authorization işlemini gerçekleştirecektir. Buradaki maksat kullanıcının sahip olduğu rollerin korunan sayfanın ihtiyaç duyduğu roller ile kıyaslamaktır. rental isimli kullanıcı beklenen role sahip olduğu için login işleminin ardından resim 11.2 de yer alan araç kiralama formuna yönlendirilir. Bu andan itibaren authentication ve authorization işlemini kapsayan güvenlik kontrolü tamamlanmıştır ve kullanıcı istediği işlemi gerçekleştirebilir.

Kullanıcı ismi ya da şifrenin hatalı olması durumunda ekranda resim 11.4 de yer alan hata mesajları görünür. Bu authentication işleminin olumsuz sonuçlandığı anlamına gelmektedir.



Resim 11.4

Olumlu sonuçlanan authentication işlemi sonrasında, kullanıcı beklenen role sahip değilse, resim 11.5 de yer alan access denied (erişim reddedildi) hatası oluşur.



Resim 11.5

Yaptığımız Spring Security konfigürasyonunun web uygulaması bünyesinde aktif hale gelebilmesi için, kod 11.5 de görüldüğü gibi web.xml dosyasına DelegatingFilterProxy sınıfını servlet filter olarak eklememiz gerekmektedir. Kod 11.5 de tanımladığımız springSecurityFilterChain Spring Security isim alanının oluşturulmasıyla otomatik olarak hayat bulan bir Spring

nesnesidir. Spring Security Modülü web uygulamalarında servlet filter mekanizmasını kullanarak, kullanıcı isteği web sayfasına erişmeden önce, tabiri caiz ise kullanıcı ile talep ettiği web sayfası arasına girerek, gerekli kontrolleri ve yönlendirmeleri yapar.

```
Kod 11.5 - web.xml

<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.
                DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Spring Security Uyumlu Login Sayfası

Kod 11.3 de kullandığımız form-login elementi resim 11.3 de görülen login sayfasının oluşmasını sağladı. Basit olan bu login sayfası birçok uygulama için tasarım ve işlev acısından yetersizdir. Bu bölümde uygulamaya has bir login sayfasının oluşturulmasını ve kullanımını inceleyeceğiz.

Kendi login sayfamızı kullanabilmek için form-login elementini kod 11.6 da görüldüğü gibi değiştirmemiz gerekmektedir.

```
Kod 11.6 - security-config.xml

<http>
    <intercept-url pattern="/secure/**"
            access="ROLE_USER" />
    <form-login login-page="/login"
            default-target-url="/secure/rental"
        login-processing-url="/j_spring_security_check"
        authentication-failure-url="/login_error"
        always-use-default-target="true" />
</http>
```

Pattern="/*" kullanıldığı taktirde, web uygulamasının sahip olduğu her sayfa güvenlik kapsamına gireceği için, login-page="/login" ile login sayfasına yapılan yönlendirme, login sayfası da güvenlik kapsamına girdiğinden sonsuz bir

döngünün oluşmasına sebep olacaktır. Bunu önlemek amacıyla uygulama sayfaları güvenlik kapsamına girenler ve güvenli kapsamına girmeyenler şeklinde iki sınıfa ayrılabilir. Kod 11.6 da yer alan örnekte güvenlik kapsamına giren tüm sayfalar /secure dizini içinde yer almaktadır. Araç kiralama işlemi için kullanılan web sayfasının adresi /secure/rental şeklinde olacaktır.

Form-login elementinde kullanılan element özellikleri şunlardır:

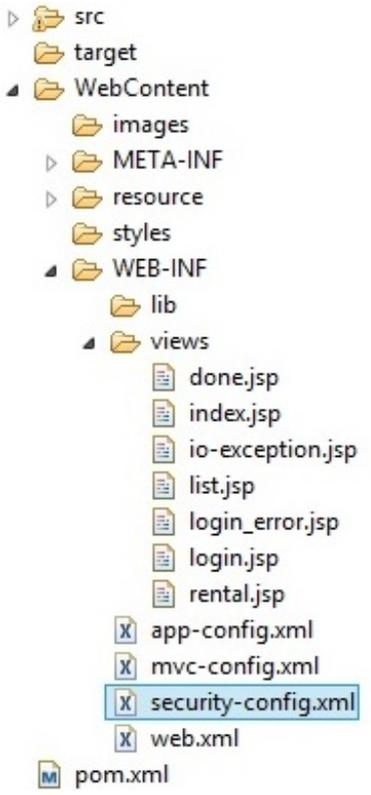
- ***default-target-url*** - Kullanıcı login yaptıktan sonra yönlendirme yapılacak sayfayı tayin etmek için kullanılır.
- ***login-processing-url*** - HTML form action elementinde login işlemini gerçekleştirmek için kullanılan web adresini tayin etmek için kullanılır (bknz. kod 11.7).
- ***authentication-failure-url*** - Login işlemi hatalı sonuç verdiğinde gösterilecek sayfayı tayin etmek için kullanılır.
- ***always-use-default-target*** - True olan bu değer her şartta kullanıcıyı default-target-url ile tanımlı sayfaya yönlendirir (redirect).

Olusturdugumuz login sayfasının HTML kodu kod 11.7 de yer almaktadır. Login.jsp uygulamanın WEB-INF/views (resim 11.6) dizininde yer almaktadır.

```
Kod 11.7 - login.jsp

<head>
<title>Login</title>
</head>
<body>
    <form action="/j_spring_security_check" method="POST">
        <label for="username">Kullan&#305;c&#305;
                    &#304;smi:</label>
        <input id="username" name="j_username"
                    type="text" />
        <label for="password">&#350;ifre:</label>
        <input id="password" name="j_password"
                type="password" />
        <inpu type="submit" value="Login" />
    </form>
</body>
</html>
```

Kod 11.6 da tanımladığımız gibi login sayfasının http://localhost:8080/login adresinde erişilir olması gerekmektedir. Bunu sağlamak için yeni bir controller sınıfı oluşturmamız ya da SimpleUrlHandlerMapping ile controller sınıfına

ihtiyaç duymayan sayfaları erişilebilir kılan bir konfigürasyon yapmamaz gerekmektedir. Login işlemi Spring Security modülü sorumluluğunda gerçekleştiği için, login.jsp için bir controller sınıfı tanımlama zorunluluğumuz bulunmamaktadır. Bu yüzden kod 11.8 de yer alan SimpleUrlHandlerMapping tanımlamasını kullanıyoruz.



Resim 11.6

Kod 11.8 de yer alan SimpleUrlHandlerMapping tanımlaması ile arkasında bir controller sınıfı olmayan sayfalar için UrlFilenameViewController yardımı ile sanal controller yapıları tanımlanmaktadır. Spring MVC UrlFilenameViewController kullanıldığı taktirde istek yapılan sayfanın isminden yola çıkarak, yine kod 11.8 de tanımlı olan InternalResourceViewResolver yardımı ile fiziksel sayfanın lokasyonunu tespit edip, kullanıcıya sunar. Bu istek yaptığımız http://localhost:8080/login sayfası için WEB-INF dizininde yer alan login.jsp sayfadır.

```
Kod 11.8

<bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.
            SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/**">urlFilenameViewController</prop>
```

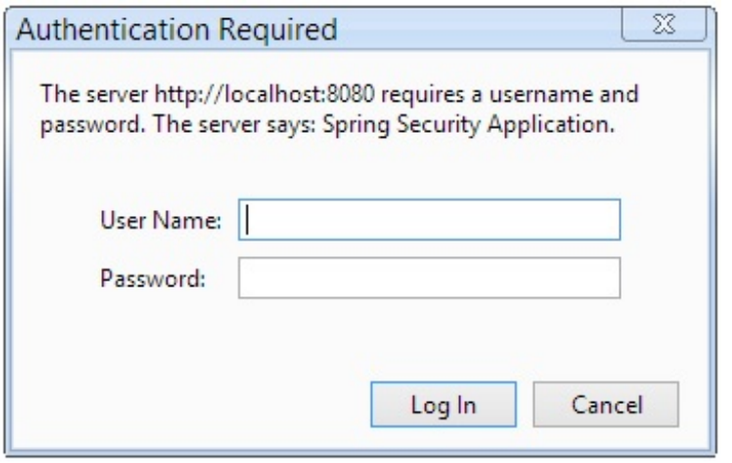
```
        </props>
    </property>
</bean>

<bean id="urlFilenameViewController"
    class="org.springframework.web.servlet.mvc.
            UrlFilenameViewController" />

<bean
   class="org.springframework.web.servlet.view.
            InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

Basic Authentication

Şimdiye kadar incelediğimiz örneklerde form-login elementini kullanarak form bazlı login mekanizmasından faydalandık. Bu login türünde bir login formunun kullanılması gerekmektedir ve hedef kitle uygulama kullanıcılardır. Bir program parçasından bu şekilde login işlemini gerçekleştirmesi beklenemez. Bu gibi durumlarda HTTP protokolünün bir parçası olan basic authentication mekanizması kullanılabilir.

```
Kod 11.9

<http>
    <intercept-url pattern="/secure/**"
            access="ROLE_USER" />
    <http-basic />
</http>
```

Basic authentication kullanıldığı taktirde resim 11.7 de görüldüğü gibi web tarayıcısı tarafından isim ve şifrenin girilebileceği bir pencere görüntülenir. Bu web tarayıcısının isim ve şifreyi edinmek için kullandığı bir yöntemdir. Uygulamaya programsal olarak (örneğin bir HTTP client aracılığı ile) erişmek istediğimizde, uygulamaya isim ve şifreyi isteğin (HTTP request) bir parçası olarak gönderebiliriz.



Resim 11.7

Spring Security bünyesinde basic authentication kod 11.9 da görüldüğü gibi http-basic elementi ile aktif hale getirilmektedir.

Logout İşlemi

Bir uygulama bünyesinde login işlemi ile kullanıcı oturumu (session) oluşturuldu ise, kullanıcının logout işlemi ile oturumu sonlandırabilmesi mümkün olmalıdır. Kullanıcı logout linkine tıklayarak bu işlemi başlatabilir. Spring Security'de logout işlemini yönetmek için logout elementi kullanılır. Aşağıda yer alan logout örneğinde logout işlemi kullanıcının /secure/logout linkine tıklamasıyla gerçekleşir. Logout işlemi ile birlikte kullanıcının otorumu sonlandırılır. Akabinde kullanıcı logout-success-url element özelliği ile tanımlanan sayfaya yönlendirir.

```
<logout logout-url="/secure/logout" logout-success-url="/" />
```

logout-url tanımlanmadığı taktirde, Spring Security /j*spring*security_logout adresini logout linki olarak kullanır.

Veri Tabanı Bazlı Authentication Provider Kullanımı

Kullanıcı isim ve şifrelerinin uygulama bünyesindeki bir kaynakta tutulması gerekmektedir. Kod 11.4 de yer alan örnekte bu hafıza bazlı bir authentication provider'dır. Kullanıcı isim ve şifreleri konfigürasyon dosyasında tutulmaktadır. Bunun çok esnek bir çözüm olmadığı ortadadır. Geniş bir kullanıcı kitlesine

sahip olan uygulamalarda kullanıcı bilgileri veri tabanı ya da LDAP (Lightweight Directory Access Protocol) gibi veri kaynaklarında tutulur. Bu bölümde veri tabanı bazlı authentication provider kullanımını inceleyeceğiz.

Kullanıcı bilgilerinin veri tabanından çekilebilmesi için bu bilgilerin belli bir yapıda olması gerekmektedir. Böyle bir veri tabanı şemasını kod 11.10 da görmekteyiz.

```
Kod 11.10 - schema.sql

create table users(
  username varchar_ignorecase(50) not null primary key,
  password varchar_ignorecase(50) not null,
  enabled boolean not null);

create table authorities (
      username varchar_ignorecase(50) not null,
      authority varchar_ignorecase(50) not null,
      constraint fk_authorities_users foreign key(username)
            references users(username));
      create unique index ix_auth_username on authorities
                (username,authority);

create table groups (
      id bigint generated by default as identity(start with 0)
        primary key,
      group_name varchar_ignorecase(50) not null);

create table group_authorities (
      group_id bigint not null,
      authority varchar(50) not null,
      constraint fk_group_authorities_group foreign key(group_id)
            references groups(id));

create table group_members (
      id bigint generated by default as identity(start with 0)
        primary key,
     username varchar(50) not null,
      group_id bigint not null,
      constraint fk_group_members_group foreign key(group_id)
        references groups(id));
```

Kullanıcı isim ve şifreyi users isimli tabloda, bu kullanıcının sahip olduğu roller authorities, kullanıcıların dahil edilebilecekler gruplar groups, bu grupların sahip oldukları roller group_authorities ve grup üyeleri group_members tablolarında tutulmaktadır. Somut bir kullanıcı örneği

kod 11.11 de yer almaktadır.

```
Kod 11.11 - data.sql

insert into users (username, password, enabled)
        values ('rental', 'rental', true);
insert into authorities (username, authority)
        values ('rental', 'ROLE_USER');
insert into groups (id, group_name)
        values (1, 'ADMIN');
insert into group_authorities (group_id,authority)
        values (1, 'ROLE_ADMIN');
insert into group_members (id, username, group_id)
        values (1, 'rental', 1);
```

rental isimli kullanıcının şifresi rental olup, bu kullanıcı ROLE_USER rolüne sahiptir. Bu kullanıcı ayrıca ADMIN grubunda olup, bu grup üzerinden ROLE_ADMIN rolüne sahiptir.

```
Kod 11.12 - security-config.xml

<authentication-manager>
    <authentication-provider
        user-service-ref='jdbcUserDetailsService' />
</authentication-manager>

<beans:bean id="jdbcUserDetailsService"
    class="org.springframework.security.core.userdetails.
        jdbc.JdbcDaoImpl">
    <beans:property name="dataSource" ref="dataSource" />
</beans:bean>
```

Kod 11.12 der alan authentication-provider tanımlaması JdbcDaoImpl sınıfı yardımı ile kullanıcı bilgilerini veri tabanından almaktadır.

Kod 11.10 da yer alan veri tabanı şemasının kullanımı mümkün olmadığı durumlarda, jdbc-user-service elementi kullanılarak, kullanıcı bilgilerini edinmek için kullanılan sql komutları tanımlanabilir. Böyle bir örnek kod 11.13 de yer almaktadır.

```
Kod 11.13 - security-config.xml

<authentication-manager>
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
```

```
            users-by-username-query="select username,password,
            enabled from users where username=?"
            authorities-by-username-query="select username,
            authority from authorities where username=?" />
    </authentication-provider>
</authentication-manager>
```

Jdbc-user-service bünyesinde kullanılan element özellikleri şunlardır:

- ***users-by-username-query*** - Kullanıcının isim, şifre ve hesabın aktiflik durumu gösteren bilgiyi edinmek için kullanılan sql komutunu tanımlamak için kullanılır.
- ***authorities-by-username-query*** - Kullanıcının sahip olduğu rolleri edinmek için kullanılan sql komutunu tanımlamak için kullanılır.
- ***group-authorities-by-username-query*** - Kullanıcının dahil olduğu gurupları edinmek için kullanılan sql komutunu tanımlamak için kullanılır.

Beni Hatırla (Remember-Me Authentication)

Login işlemi esnasında kullanıcıya "Beni Hatırla" (Remember Me) seçeneği sunularak, bir sonraki login işleminin otomatik olarak yapılması sağlanabilir. Spring Security bünyesinde iki değişik "Beni Hatırla" implementasyonu bulunmaktadır. Kod 11.14 der alan örnekte remember-me elementi ile hash fonksiyonu bazlı "Beni Hatırla" konfigürasyonu görülmektedir. Token olarak isimlendirilen değerin oluşturulabilmesi için key element özelliğinde yer alan anahtar kullanılmaktadır. Oluşturulan token'in geçerlilik süresi token-validity-seconds element özelliği ile tanımlanabilir. Kod 11.14 de yer alan örnekte oluşturulan token üç gün boyunca geçerli kalacaktır. Bu zaman diliminde kullanıcı, kullanıcı bilgisayarında oluşturulan çerez (cookie) yardımı ile uygulamaya otomatik olarak giriş yapabilir.

```
Kod 11.14 - security-config.xml

<http>
    ...
    <remember-me key="myKey"
        token-validity-seconds="259200" />
</http>
```

Bunun yanı sıra Spring Security bünyesinde veri tabanı bazlı "Beni Hatırla" implementasyonu kullanılabilir.

```
Kod 11.15 - security-config.xml

<http>
    ...
    <remember-me data-source-ref="dataSource"/>
</http>
```

Data-source-ref element özelliğinde kullanılan değer uygulama bünyesinde veri tabanı erişimi için oluşturduğumuz dataSource nesnesidir. Bu "Beni Hatırla" implementasyonu veri tabanında persistent_logins isminde bir tabloya ihtiyaç duymaktadır. Bu tablonun yapısı kod 11.16 da yer almaktadır.

```
Kod 11.16 - schema.sql

create table persistent_logins (
    username varchar(64) not null,
      series varchar(64) primary key,
      token varchar(64) not null,
      last_used timestamp not null);
```

HTTPS Kullanımı

Uygulama HTTP yanı sıra ya da yerine HTTPS (Secure Hypertext Transfe Protocol) aracılığı ile erişilebilir olmalıysa, requires-channel element özelliği ile kod 11.17 de görüldüğü gibi uygulamaya bu özellik eklenebilir.

```
Kod 11.17 - security-config.xml

<http>
    <intercept-url pattern="/secure/**" access="ROLE_USER"
            requires-channel="https"/>
    <intercept-url pattern="/**" access="ROLE_USER"
            requires-channel="any"/>
 </http>
```

Kod 11.17 de yer alan konfigürasyon örneginde /secure adresi altında yer alan tüm sayfalar sadece HTTPS (requires-channel="https") üzerinden erişilebilir durumdadır. / adresi altında yer alan diğer sayfalar hem HTTP hem de HTTPS (requires-channel="any") aracılığı ile erişilebilir. Standart olmayan HTTP (standart port 80 dir) ya da HTTPS (standart port 443 dür) port konfigürasyonu kod 11.18 de görüldüğü gibi port-mappings elementi ile yapılabilir.

```
 Kod 11.18 - security-config.xml
```

```
<http>
   ...
   <port-mappings>
         <port-mapping http="9080" https="9443"/>
   </port-mappings>
 </http>
```

SpEL İle Güvenlik Konfigürasyonu

Dördüncü bölümde Spring Expression Language (SpEL) ile tanışmıştık. Spring Security 3.0 sürümü ile güvenlik konfigürasyonlarında SpEL kullanımını mümkün hale gelmiştir. Bu özelliği kullanabilmek için http elementi bünyesinde use-expressions element özelliğinin true değerini taşıması gerekmektedir (kod 11.19).

```
Kod 11.19 - security-config.xml

<http use-expressions="true">
..
</http>
```

Kullanılabilecek SpEL güvenlik ifadeleri (expressions) şunlardır:

- ***authentication*** - Kullanıcının authentication nesnesine doğrudan erişimi sağlar.
- ***hasIpAddress*** - Kullanıcının IP adresi olup, olmadığını sorgular.
- ***isAnonymous*** - Anonim kullanıcı için true değerini geri verir.
- ***isAuthenticated*** - Anonim olmayan kullanıcı için true değerini geri verir.
- ***permitAll*** - Her zaman true değerini geri verir.
- ***denyAll*** - Her zaman false değerini geri verir.
- ***hasAnyRole(rol1, rol2)*** - Kullanıcı tanımlı rol listesinden herhangi bir role sahip ise true değerini geri verir.
- ***hasRole(rol)*** - Kullanıcı tanımlı role sahip ise true değerini geri verir.
- ***isRememberMe*** - Kullanıcı "Beni Hatırla" fonksiyonu aracılığı ile login işlemini yaptı ise, true değerini geri verir.
- ***principal*** - Kullanıcının principal nesnesine doğrudan erişimi sağlar. Principal nesnesi kullanıcıyı temsil eden nesnedir.
- ***isFullyAuthenticated*** - Kullanıcı anonim ya da "Beni Hatırla" kullanıcısı değil ise, true değerini geri verir.

Kod 11.20 de yer alan örnekte kullanıcı ROLE_USER ismini taşıyan bir role sahip olmak zorundadır. Bunun yanı sıra hasIpAddress ile kullanıcının IP adresi kontrol edilmektedir. Eğer bu IP adresi 213.221.93.251 değil ve kullanıcı ROLE_USER rolüne sahip değilse, kullanıcı /secure adresi altında yer alan sayfalara erişemez.

```
Kod 11.20 - security-config.xml

<http use-expressions="true">
    <intercept-url pattern="/secure/**"
        access="hasRole('ROLE_USER') and
            hasIpAddress('213.221.93.251')"/>
    ...
  </http>
```

Spring Security JSP Tag Kullanımı

Kod 11.16 da yer alan güvenlik konfigürasyonu kullanıcının başarılı bir login ismi ardından /secure adresi altında olan sayfalara ersimini mümkün kılacaktır. Bu sayfalarda rol ya da hak bazında veri gösterimi yapılmak istendiğinde Spring Security JSP taglib kullanılabilir. Kod 11.21 der alan kod örneğinde taglib JSP sayfasına eklenmektedir. sec:authorize elementi bünyesinde yer alan metni sadece ROLE_ADMIN rolüne sahip kullanıcılar görebilir.

```
Kod 11.21 - index.jsp

<%@ taglib prefix="sec"
        uri="http://www.springframework.org/security/tags" %>

<sec:authorize access="hasRole('ROLE_ADMIN')">
    Bu bölümü sadece ROLE_ADMIN rolüne sahip olanlar görebilir.
</sec:authorize>
```

Spring Security Taglib bünyesinde yer alan elementler şunlardır:

- ***authorize*** - Kod bloğunun tayin edilen şartların yerine gelmesi durumunda koşturulmasını sağlar.
- ***authentication*** - Login işlemi ardından kullanıcı için oluşturulan authentication nesnesine erişimi mümkün kılar.
- ***accesscontrollist*** - Kod bloğunun kullanıcının kullanılan alan nesnesi haklarına sahip olması durumunda koşturulmasını sağlar.

authorize elementinin access element özelliğinde bir önceki bölümde tanımladığımız SpEL güvenlik ifadelerinin hepsi kullanılabilir. Bu elementin kullanımı veriye rol bazlı erişimin kontrolünü sağlamaktadır. Kod 11.22 de yer alan örnekte authorize elementinin url element özelliği kullanılmıştır. Kod 11.9 tekrar göz attığımızda /secure altındaki sayfalara sadece ROLE_USER rolüne sahip olan kullanıcıların erişileceğini görmekteyiz. Bu durumda url element özelliği ile dolaylı olarak rol kontrolü yapılmakta ve sadece ROLE_USER rolüne sahip kullanıcılar /secure/listuser linkini görebilmektedirler.

```
Kod 11.22 - index.jsp

<%@ taglib prefix="sec"
    uri="http://www.springframework.org/security/tags" %>

<sec:authorize url="/secure/**">
    <a href="/secure/listuser">List User</a>
</sec:authorize>
```

Authorize elementinin var isminde bir element özelliği daha bulunmaktadır. Kod 11.23 de var ile JSP sayfasında kullanılmak üzere isAdmin ismini taşıyan bir değişken tanımlanmaktadır. Eğer kullanıcı ROLE_ADMIN rolüne sahip ise, isAdmin true değerine sahip olacaktır.

```
Kod 11.23 - index.jsp

<%@ taglib prefix="sec"
    uri="http://www.springframework.org/security/tags" %>

<sec:authorize access="hasRole('ROLE_ADMIN')" var="isAdmin">
    Bu bölümü sadece ROLE_ADMIN rolüne sahip olanlar görebilir.
</sec:authorize>
<c:if test="${isAdmin}">
      // Admin olan kullanici görebilir.
</c:if>
```

Kod 11.24 de yer alan örnekte security:authentication elementi ile kullanıcının ismine erişim sağlanmaktadır. authentication elementi ile erişilebilecek nesneler şunlardır:

- ***authorities*** - Kullanıcının sahip olduğu rol listesi.
- ***credentials*** - Kullanıcının şifresi.
- ***details*** - Authentication işlemi için ek bilgiler (ip adresi, oturum numarası vs).

- ***principal*** - Kullanıcının principal nesnesi (isim gibi kullanıcı bilgilerini ihtiva eden ve login işlemi ardından kullanıcıyı temsil eden nesne.).

Kod 11.24 de authentication elementinin property element özelliği principal nesnesinde yer alan username isimli değişkene erişmek için kullanılmıştır. Bu değişkenin değeri var element özelliği yardımı ile yeni bir değişkene atanabilir. Kod 11.24 de yer alan örnekte var element özelliği yardımı ile username isminde, kullanıcının ismini taşıyan bir değişken oluşturulmaktadır. Scope element özelliği var ile oluşturulan değişkenin geçerlilik alanını tayin etmektedir.

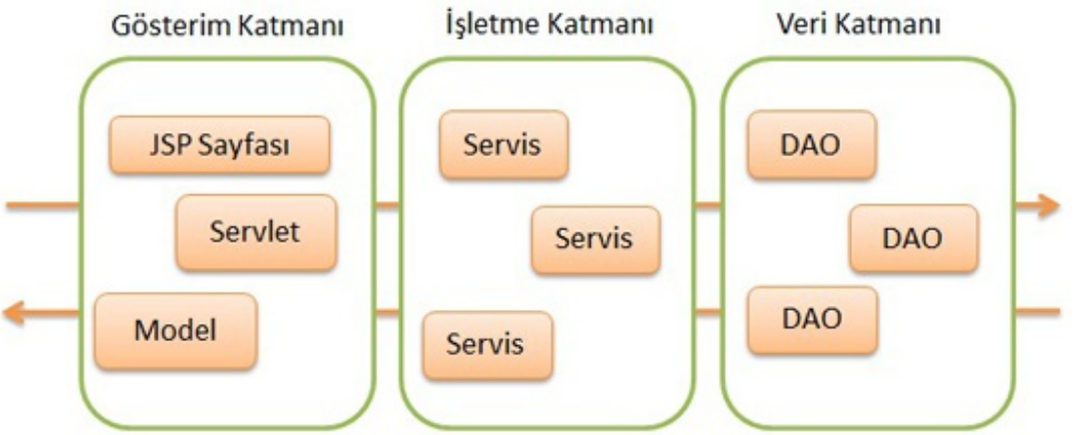
Kod 11.24 de yer alan örnekte bu request geçerlilik (scope) alanıdır. username ismini taşıyan değişken sadece kullanıcı isteği boyunca geçerliliğini koruyacaktır. Geçerlilik alanı olarak javax.servlet.jsp.PageContext (request,page, session, application) bünyesinde yer alan herhangi bir scope kullanılabilir .

```
Kod 11.24 - index.jsp

Merhaba <security:authentication
        property="principal.username"
        var="username" scope="request"/>, Hoşgeldiniz!

Sifreniz: <sec:authentication property="credentials"/>

<sec:authentication property="authorities"
        var="roles" scope="page" />
Rolleriniz:
<ul>
    <c:forEach var="role" items="${roles}">
    <li>${role}</li>
    </c:forEach>
</ul>
```

Metot Bazında Güvenlik

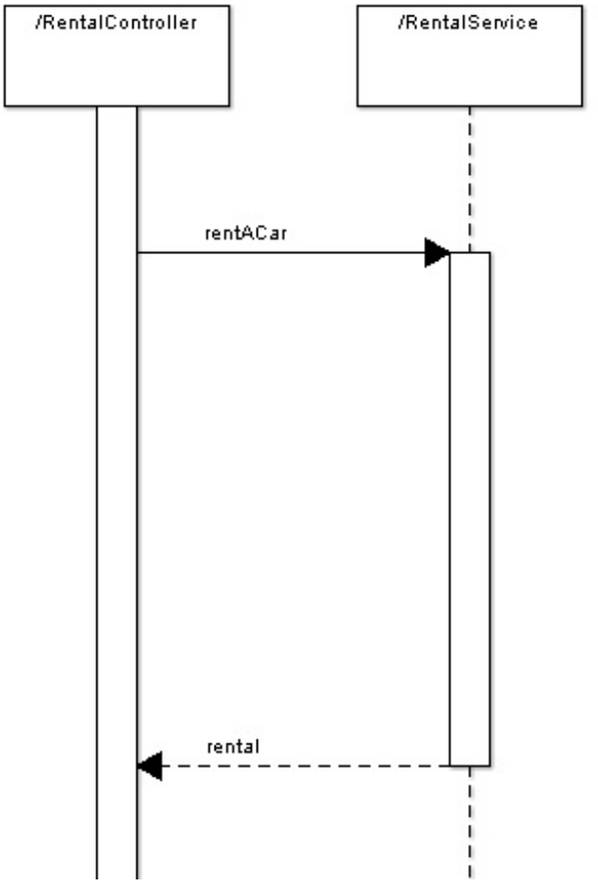
Buraya kadar incelediğimiz konular web sayfalarının güvenliği ile ilgiliydi. Tipik üç katmanlı bir mimaride web sayfaları uygulamanın gösterim katmanında yer alırlar. Birçok uygulama için web sayfalarında uygulanan güvenlik konseptleri yeterlidir. Servis katmanına olan erişim web sayfalarında geçerli olan güvenlik kriterleri aracılığı ile kontrol edilmiş olur. Bu ilk bakışta servis katmanı için güvenlik tedbirlerinin gerekli olmadığı intibaını uyandırabilir. Aynı servis katmanı birden fazla gösterim katmanı ya da

uygulama tarafından kullanılıyorsa, servis katmanını kullanan her uygulamadan servis katmanını koruması beklenemez. Bu sebepten dolayı servis katmanı için ihtiyaç duyduğu güvenlik tedbirlerinin alınarak, güvenlik çemberinin oluşturulması gerekmektedir. Bu amaçla Spring Security bünyesindeki metot bazında güvenlik mekanizması kullanılabilir.



Resim 11.8

Servis katmanını kullanan bir uygulama istediği işlemin yapılması için servis katmanında yer alan bir sınıfın herhangi bir metodunu koşturacaktır. Araç kiralama uygulamasında gösterim katmanı servis katmanını resim 11.9 daki gibi kullanmaktadır. Kullanılan metot RentalService bünyesinde yer alan rentACar() metodudur. Bu metoda olan erişimi rol bazında kontrol etmek için @Secured anotasyonu kullanılabilir.



Resim 11.9

Kod 11.25 de yer alan konfigürasyon örneğinde global-method-security elementi ile @Secured anotasyonu kullanıma hazırlanmaktadır.

```
Kod 11.25 - security-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/
                context"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
            http://www.springframework.org/schema/beans/
                spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/
                spring-context-3.0.xsd
            http://www.springframework.org/schema/security
            http://www.springframework.org/schema/security/
            spring-security-3.1.xsd">
```

```
    <context:component-scan base-package="com.kurumsaljava" />

    <sec:global-method-security secured-annotations="enabled" />
</beans>
```

Kod 11.25.1 de rentACar() metodu @Secured anotasyonu ile güvenlik kapsamına alınmaktadır. Bu rentACar() metodu ROLE_ADMIN ya da ROLE_USER rolüne sahip kullanıcılar tarafından koşturulabilir anlamına gelmektedir. Burada Spring AOP (Aspect Oriented Programming) teknolojisini kullanarak @Secured anotasyonunun geçtiği yerde bir pointcut oluşturmakta ve Spring Security aspekt nesneleri bu metot koşturulmadan önce araya girerek kullanıcı rollerini kontrol etmektedirler.

```
Kod 11.25.1 - RentalServiceImpl

@Secured("ROLE_ADMIN", "ROLE_USER")
public Rental rentACar(final Rental rental) {

    final Customer dbCustomer =
        customerRepository.getCustomerByName(
            rental.getCustomer().getName());

    if (dbCustomer == null) {
        customerRepository.save(rental.getCustomer());
    }

    final Car car = carRepository.findCarById(1);

    rental.setCar(car);
    rental.setRented(true);
    rental.setCustomer(rental.getCustomer());
    rentalRepository.save(rental);
    return rental;
}
```

Kullanıcı beklenen rollere sahip olması durumunda rentACar() metodu bir engel olmadan koşturulur. Aksi durumda kod 11.26 da yer alan AccessDeniedException hatası oluşacaktır.

```
Kod 11.26 - Stacktrace

org.springframework.security.access.AccessDeniedException:
            Access is denied
at org.springframework.security.access.vote.
            AbstractAccessDecisionManager.
```

```
checkAllowIfAllAbstainDecisions(
            AbstractAccessDecisionManager.java:70)
at org.springframework.security.access.vote.
            AffirmativeBased.decide(AffirmativeBased.java:88)
at org.springframework.security.access.intercept.
            AbstractSecurityInterceptor.beforeInvocation
(AbstractSecurityInterceptor.java:206)
at org.springframework.security.access.intercept.
            aopalliance.MethodSecurityInterceptor.invoke
(MethodSecurityInterceptor.java:60)
at org.springframework.aop.framework.
            ReflectiveMethodInvocation.proceed
(ReflectiveMethodInvocation.java:172)
at org.springframework.aop.framework.JdkDynamicAopProxy.
            invoke(JdkDynamicAopProxy.java:202)
at com.sun.proxy.$Proxy62.rentACar(Unknown Source)
at com.kurumsaljava.spring.web.RentalController.
            processRental(RentalController.java:61)
at com.kurumsaljava.spring.web.RentalController.
            processSubmit(RentalController.java:54)
```

@Secured bir Spring anotasyonudur. Kullanıldığı yerde kodu doğal olarak Spring çatısına bağımlı kılmaktadır. Bu bağımlılığı ortadan kaldırmak için JSR 250 (Common Annotations for Java) ile tanımlanan genel Java anotasyonları kullanılabilir. JSR 250 ile gelen Java anotasyonlarını kullanabilmek için jsr250-annotations element özelliğinin kod 11.27 de görüldüğü gibi enabled değerini taşıması gerekmektedir. Bu işlemin ardından @Secured yerine bir JSR 250 anotasyonu olan @RolesAllowed kullanılabilir. Her iki anotasyonunuda gerekli konfigürasyonu yaparak aynı anda kullanmak mümkündür.

```
Kod 11.27 - security-config.xml

<sec:global-method-security
        jsr250-annotations="enabled" />
```

@Pre ve @Post Anotasyonları

@RolesAllowed ya da @Secured anotasyonu ile sadece bir metoda giriş kontrol edilebilir. Kullanıcı beklenen rollere sahip değilse, metot koşturulmaz. Bazen rollerin yanı sıra bazı şartların da kontrolü gerekli olabilir. Örneğin bir metoda erişim ROLE_ADMIN rolünü gerektirirken, kullanıcının hangi IP adresini kullanarak ta girdiği güvenlik sorgulamasının bir parçası olmak zorunda olabilir. Bu amaçla Spring 3 ile gelen @Pre ve @Post anotasyonları kullanılabilir. Bu anotasyonlar daha önce tanıştığımız güvenlik ifadelerinin

(SpEL Security Expressions) kullanımını mümkün kılmaktadırlar. Bu anotasyonları kullanabilmek için kod 11.28 deki gibi aktif hale getirilmeleri gerekmektedir.

```
Kod 11.28 - security-config.xml

<sec:global-method-security
        pre-post-annotations="enabled"/>
```

Spring 3 ile gelen yeni güvenlik anotasyonları şunlardır:

- ***@PreAuthorize*** - Güvenlik ifadesinin neticesine göre metoda erişimi kontrol eder.
- ***@PostAuthorize*** - Metodun koşturulmasına ilk etapta izin vermekle birlikte güvenlik ifadesinin olumsuz olması durumunda güvenlik hatası oluşturur.
- ***@PreFilter*** - Kullanılan güvenlik ifadesine göre metot parametrelerinin filtrelenmesini sağlar.
- ***@PostFilter*** - Kullanılan güvenlik ifadesine göre metodun geriye verdiği değerlerin filtrelenmesini sağlar.

Şimdi bu anotasyonları ihtiva eden birkaç örneği yakından inceleyelim. Aşağıdaki örnekte sadece ROLE_ADMIN rolüne sahip olanlar rentACar() metodunu koşturabilir.

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
public Rental rentACar(final Rental rental);
```

Aşağıdaki örnekte ROLE_ADMIN rolüne sahip olanlar rentACar() metodunu koşturabilir. Bunun yanı sıra rental nesnesinin sahip olduğu car nesnesinin brand isimli değişkeninin değeri Ford olmak zorundadır. Aksi taktirde kullanıcı ROLE_ADMIN rolüne sahip olsa bile metodun koşturulmasına izin verilmez.

```
@PreAuthorize("hasRole('ROLE_ADMIN')
            and #rental.car.brand == 'Ford'")
public Rental rentACar(final Rental rental);
```

@PostAuthorize anotasyonunda güvenlik kontrolleri metodun geriye verdiği değer üzerinde yapılmaktadır. Doğal olarak bu değeri elde edebilmek için metodun koşturulması gerekmektedir. Aşağıdaki örnekte getRental() metodu bir Rental nesnesi geri vermektedir. @PostAuthorize anotasyonu ile bu

nesnesinin kullanıcıya ait olup, olmadığı kontrol edilmektedir. Kullanıcı login işlemini gerçekleştirdikten sonra principal nesnesinin username değişkeninde kullanıcı ismi yer alır. Eğer getRental() metodu ile elde edilen Rental nesnesi (returnObject) kullanıcıya ait değilse, @PostAuthorize AccessDeniedException hatası oluşturur ve elde edilen Rental nesnesinin kullanımını engeller. @PostAuthorize ile metot her seferine koşturulduğu için yan etkilerinin olmadığına dikkat edilmesi gerekmektedir.

```
@PostAuthorize("#returnObject.username ==
                principal.username")
public Rental getRental(long userId);
```

@PostFilter anotasyonu metodun geri verdiği değerler üzerinde filtreleme işlemlerini gerçekleştirmek için kullanılır. Aşağıdaki örnekte getRentals() metodu bir kullanıcıya ait Rental nesnelerini listelemektedir. @PostFilter kullanıcıya ait olmayan nesneleri

```
filterObject.rental.username == principal.name
```

ifadesi ile liste dışı bırakmaktadır. FilterObject metodun geriye verdiği liste içindeki nesnelere erişmek için kullanılmaktadır. @PreFilter anotasyonu @PostFilter anotasyonuna analog olarak metot parametrelerini filtrelemek için kullanılır.

```
@PostFilter("filterObject.rental.username ==
            principal.name")
public List<Rental> getRentals(long userId){

}
```

Pointcut Örneği

Bir taşla iki kuş vurma misali protect-pointcut elementi ile birden fazla sınıfı güvenlik çemberine almak için pointcut oluşturulabilir. Kod 11.29 da yer alan örnekte com.kurumsaljava paketinde yer alan ve isminde ServiceImpl ibaresini taşıyan tüm sınıfların tüm metotları güvenlik çemberine dahil edilmektedir. Bu güvenlik çemberine giriş sadece ROLE_USER rolüne sahip olan kullanıcılar için mümkündür. Bu şekilde zahmetli olan sınıf ya da metot bazında güvenlik çemberi oluşturma işlemi merkezi bir yerden kontrol edilebilir hale gelmektedir.

```
Kod 11.29 - security-config.xml
```

```
<global-method-security>
    <protect-pointcut expression="execution(* com.
            kurumsaljava.*ServiceImpl.*(..))"
         access="ROLE_USER"/>
 </global-method-security>
```

Alan Nesnesi Bazında Güvenlik

Müşteri, sipariş, ürün gibi bilgileri uygulamada temsil eden nesneler alan (domain) nesneleridir. Alan nesnelerinin ihtiva ettiği bilgiler çoğu zaman veri tabanında tutulur. Bu bilgilere erişim üç katmanlı bir mimaride (resim 11.8) veri katmanı üzerinden gerçekleşir. Alan nesneleri çoğu zaman servis katmanı üzerinden gösterim katmanına iletilir ve böylece ihtiva ettikleri bilgilerin web sayfalarında gösterimi sağlanır. Bu sebepten dolayı bu bilgilere gerek duyulduğunda erişimin kontrol edilebilmesi gerekmektedir.

Spring Security ile alan nesneleri erişim kontrol listeleri (ACL - Access Control List) aracılığı ile güvenlik çemberine alınabilir. Bu amaçla alan nesneleri üzerinde yapılacak işlemlerin ve kimlerin bu işlemleri yapma yetkisinin olduğunun tanımlanması gerekmektedir.

Bir alan nesnesi üzerinde temelde dört işlem yapılabilir. Bunlar:

- ***Gösterim (View)*** - Alan nesnesinin sahip olduğu verilerin gösterimi.
- ***Olusturma (Add)*** - Yeni bir alan nesnesinin oluşturulması.
- ***Imha (Delete)*** - Bir alan nesnesinin ve ihtiva ettiği bilgilerin veri tabanından silinmesi.
- ***Degistirme (Edit)*** - Bir alan nesnesinin ve ihtiva ettiği bilgilerin veri tabanında değiştirilmesi.

Alan nesneleri bazında güvenliği nasıl sağlayabileceğimizi bir örnek üzerinde inceleyelim. Uygulamamız bünyesinde Customer, Rental ve Car isminde üç alan nesnesi barındırıyor. Bunun yanı sıra ROLE_ADMIN ve ROLE_CUSTOMER isminde iki rolümüz bulunuyor. Tablo 11.1 de ROLE_ADMIN rolüne sahip kullanıcıların, tablo 11.2 de ROLE_CUSTOMER rolüne sahip kullanıcıların alan nesneleri üzerinde yapabileceği işlemler yer almaktadir. ROLE_ADMIN alan nesneleri üzerinden her türlü hakka sahip iken, ROLE_CUSTOMER rolüne sahip bir kullanıcı sadece kendine ait Customer ve Rental alan nesnelerine bakabilmekte (view), yeni bir Rental nesnesi oluşturabilmekte (add), kendi müşteri bilgilerini

(Customer edit) değiştirebilmekte ve Car nesnelerine bakabilmektedir (view). Bunun haricindeki işlemlere ROLE_CUSTOMER rolü için kısıtlama getirilmiştir.

ROLE_ADMIN	View	Add	Edit	Delete
Customer	x	x	x	x
Rental	x	x	x	x
Car	x	x	x	x

Tablo 11.1

ROLE_CUSTOMER	View	Add	Edit	Delete
Customer	x		x	
Rental	x	x		
Car	x			

Tablo 11.2

Spring Security ile alan nesnesi bazında güvenlik çemberini oluşturabilmek için veri tabanında aşağıda yer alan tabloların oluşturulması gerekmektedir. Bu tablolar:

- ***acl_class*** - Alan sınıflarının tanımlandığı tablodur.
- ***acl_sid*** - kullanıcı isimlerinin ya da rollerin tanımlandığı tablodur.
- ***acl_object_identity*** - Alan nesnelerinin kimliklerini ihtiva eden tablodur.
- ***acl_entry*** - Hangi kullanıcının hangi alan nesnesi üzerinde hangi işlemi yapabildiğinin tanımlandığı tablodur.

acl_object_identity tablosunda veri tabanında kayıtlı olan her alan nesnesi için bir kayıt yer almaktadır. Örneğin veri tabanında araçlar kod 11.30 da görüldüğü gibi car isimli tabloda tutulmaktadır. Kod 11.30 da yer alan iki insert komutu ile car isimli tabloda iki araç kaydı oluşturulmaktadır. Uygulama bünyesinde Car isimli alan sınıfı car isimli veri tabanı tablosunda yer alan verileri temsil etmektedir. Kod 11.30 a göre uygulama bünyesinde iki Car alan

nesnesi mevcuttur. Bu alan nesnelerine erişimi kontrol etmek amacıyla her birisi için acl_object_identity tablosunda kayıt oluşturulması gerekmektedir. acl_entry tablosunda bu bilgiler kullanılarak hangi kullanıcının hangi alan nesnesine erişebileceği tanımlanabilmektedir. İlerleyen satırlarda acl_object_identity ve diğer tabloların içeriğini somut örnekler üzerinde yakından inceleyeceğiz.

```
Kod 11.30 - schmema.sql

create table car (
    id BIGINT GENERATED BY DEFAULT AS IDENTITY,
    brand varchar(50),
    model varchar(50));

insert into car (id, brand, model)
        values ('1000', 'Ford', 'Fiesta');
insert into car (id, brand, model)
        values ('1001', 'Audi', 'A1');
```

Alan nesnesi bazlı güvenlik için gerekli konfigürasyon kod 11.31 de yer almaktadır. Şimdi adım adım konfigürasyonda kullanılan elementlerin hangi vazifeleri üstlendiklerini inceleyelim.

Alan nesnesi bazlı güvenliğin aktivasyonu global-method-security elementi ile yapılmaktadır. Spring Security alan nesneleri için gerekli kontrolleri metot girişi öncesi ya da sonrası yapmaktadır. Bunu sağlamak için kullanabileceğimiz @Pre ve @Post anotasyonları ile daha önce tanışmıştık.

global-method-security elementi bünyesinde kullandığımız pre-post-annotations element özelliği yardımı ile güvenlik ifadeleri kullanabilir hale gelmektedir. Başka bir konfigürasyon yapılmadığı taktirde Spring Security güvenlik ifadelerini analiz edip, koşturmak için DefaultMethodSecurityExpressionHandler sınıfını kullanmaktadır. Bu implementasyonda ACL desteği yoktur. AclPermissionEvaluator sınıfı güvenlik ifadelerinin koşturulmasında ACL desteği sağlamaktadır. Bu yüzden permissionEvaluator ismi altında bir Spring nesnesi tanımlayarak, bu nesneyi expressionHandler nesnesinin permissionEvaluator değişkenine enjekte ediyoruz. Bu şekilde DefaultMethodSecurityExpressionHandler sınıfı ACL destekli güvenlik ifadelerini koşturabilir hale gelmektedir.

JdbcMutableAclService JdbcTemplate yardımı ile birkaç paragraf önce tanıdığımız ACL veri tablolarında (acl_class, acl_sid vs.) gerekli sorgulamaları

yapmak için kullanılan AclService implementasyondur. Bu tabloları daha sonra detaylı olarak inceleyeceğiz.

```
Kod 11.31 - acl-config.xml

<global-method-security
        pre-post-annotations="enabled" >
    <expression-handler ref="expressionHandler" />
<global-method-security>

<bean id="expressionHandler"
    class="org.springframework.security.access.
        expression.method.
            DefaultMethodSecurityExpressionHandler">
    <property name="permissionEvaluator"
            ref="permissionEvaluator" />
    <property name="roleHierarchy"
                ref="roleHierarchy" />
</bean>

<bean class="org.springframework.security.acls.
            AclPermissionEvaluator"
    id="permissionEvaluator">
    <constructor-arg ref="aclService" />
</bean>

<bean class="org.springframework.security.acls.jdbc.
            JdbcMutableAclService"
    id="aclService">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="lookupStrategy" />
    <constructor-arg ref="aclCache" />
</bean>

<bean id="lookupStrategy"
    class="org.springframework.security.acls.jdbc.
            BasicLookupStrategy">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="aclCache" />
    <constructor-arg ref="aclAuthorizationStrategy" />
    <constructor-arg ref="auditLogger" />
</bean>

<bean id="aclCache"
    class="org.springframework.security.acls.domain.
            EhCacheBasedAclCache">
    <constructor-arg>
        <bean class="org.springframework.cache.ehcache.
```

```
                    EhCacheFactoryBean">
            <property name="cacheManager">
                <bean
                    class="org.springframework.cache.
                        ehcache.EhCacheManagerFactoryBean" />
            </property>
            <property name="cacheName" value="aclCache" />
        </bean>
    </constructor-arg>
</bean>

<bean id="aclAuthorizationStrategy"
    class="org.springframework.security.acls.domain.
            AclAuthorizationStrategyImpl">
    <constructor-arg>
        <list>
            <bean
                class="org.springframework.security.
                core.authority.GrantedAuthorityImpl">
                <constructor-arg value="ROLE_ADMIN" />
            </bean>
        </list>
    </constructor-arg>
</bean>

<bean id="auditLogger"
    class="org.springframework.security.acls.domain.
            ConsoleAuditLogger" />

<bean id="roleHierarchy"
    class="org.springframework.security.access.
            hierarchicalroles.RoleHierarchyImpl">
    <property name="hierarchy">
        <value>
            ROLE_ADMIN > ROLE_CUSTOMER > ROLE_VISITOR
     </value>
    </property>
</bean>
```

RoleHierarchy roller arasındaki hiyerarşiyi ifade etmek için kullanılmaktadır. Kod 11.31 de yer alan örnekte hiyerarşinin en üstünde yer alan rol ROLE_ADMIN rolüdür. Onu ROLE_CUSTOMER rolü takip etmektedir. En altta ROLE_VISITOR rolü yer almaktadır. ROLE_ADMIN rolüne sahip olan bir kullanıcı aynı zamanda ROLE_CUSTOMER ve ROLE_VISITOR rollerine ve bu rollerin sahip olduğu haklara da dolaylı olarak sahiptir. Aynı şey ROLE_CUSTOMER rolü için de geçerlidir. ROLE_CUSTOMER rolüne sahip

kullanıcı aynı zamanda ROLE_VISITOR rolüne de sahiptir. Rol hiyerarşileri alan nesneleri kontrol listelerine erişimi kolaylaştırmaktadır.

LookupStrategy tanımlaması ACL tabloları üzerinde yapılan SQL işlemlerini optimize etmek için kullanılmaktadır. ACL veri taban tablolarına erişimi sağlamak için lookupStrategy nesnesine dataSource nesnesi enjekte edilmektedir.

LookupStrategy nesnesine enjekte edilen aclCache ACL veri tabanı tabloları üzerinde koşturulan SQL komutlarını sınırlamak için kullanılmaktadır. SQL komutları ile veri tabanından edinilen bilgiler önbelleğe (cache) alınarak, bu verilere erişim hızlandırılmakta ve veri tabanı kullanımı azaltılmaktadır. Kod 11.31 de görüldüğü gibi EhCacheBasedAclCache sınıfı aracılığı ile EhCache ürünü kullanılmaktadır. EhCache açık kaynaklı bir caching çatısıdır.

Son olarak dikkatimizi aclAuthorizationStrategy nesnesi çekmektedir. AclAuthorizationStrategyImpl sınıfı ACL nesneleri üzerinde administratif değişiklikler (yeni ACL oluşturma, silme, değiştirme vs.) yapılmak istendiğinde gerekli güvenlik kontrollerini yapmak için kullanılmaktadır. Kullanıcı (principal) sadece erişim hakkına sahip olduğu ACL listeleri üzerinde administratif işlem yapabilir. Bunun yanı sıra sistem genelinde geçerli roller tanımlanarak, bu rollere sahip kullanıcıların her türlü ACL listesi üzerinde değişiklik yapması sağlanabilir. Bu amaçla AclAuthorizationStrategyImpl implementasyonuna rol isminin konstrüktör parametresi olarak verilmesi gerekmektedir. Kod 11.31 de yer alan örnekte ROLE_ADMIN ACL listeleri üzerinde her türlü administratif değişikliği yapma yetkisine sahiptir.

Konfigürasyon işlemi ardından ACL veri tabanı tablolarının yapılarını yakından inceleyelim. Adı gecen tabloların yapıları kod 11.30 da yer almaktadır.

```
Kod 11.32 - schema.sql

create table acl_sid (
  id bigint generated by default as identity(start with 100)
                not null primary key,
  principal boolean not null,
  sid varchar_ignorecase(100) not null,
  constraint unique_uk_1 unique(sid,principal) );

create table acl_class (
```

```
      id bigint generated by default as
            identity(start with 100)
            not null primary key,
      class varchar_ignorecase(100) not null,
      constraint unique_uk_2 unique(class) );

create table acl_object_identity (
      id bigint generated by default as
            identity(start with 100) not null
                primary key,
      object_id_class bigint not null,
      object_id_identity bigint not null,
      parent_object bigint,
      owner_sid bigint not null,
      entries_inheriting boolean not null,
      constraint unique_uk_3 unique(
                object_id_class,object_id_identity),
      constraint foreign_fk_1 foreign key(
                parent_object)references
                    acl_object_identity(id),
      constraint foreign_fk_2 foreign key(
                object_id_class)references acl_class(id),
      constraint foreign_fk_3 foreign key(
                owner_sid)references acl_sid(id) );

create table acl_entry (
      id bigint generated by default as identity(
                start with 100) not null primary key,
      acl_object_identity bigint not null,ace_order
                int not null,sid bigint not null,
      mask integer not null,granting boolean not null,
                audit_success boolean not null,
      audit_failure boolean not null,
      constraint unique_uk_4 unique(
            acl_object_identity,ace_order),
      constraint foreign_fk_4 foreign key(
            acl_object_identity)
    references acl_object_identity(id),
      constraint foreign_fk_5 foreign key(sid)
            references acl_sid(id) );
```

İlk adımda Rental alan nesnelerinin temsil ettikleri kayıtları oluşturulalım:

```
insert into rental (id, car_id, customer_id, rented)
        values ('100', 1000, 100, true);
insert into rental (id, car_id, customer_id, rented)
        values ('200', 1001, 100, true);
insert into rental (id, car_id, customer_id, rented)
```

```
        values ('300', 1000, 100, true);
```

İkici adımda admin1 ve customer1 isminde iki yeni kullanıcı oluşturalım:

```
insert into users (username, password, enabled)
            values ('admin1', 'admin1', true);
insert into users (username, password, enabled)
            values ('customer1', 'customer1', true);
```

Kullanıcılara rollerini atayalım:

```
insert into authorities (username, authority)
            values ('admin1', 'ROLE_ADMIN');
insert into authorities (username, authority)
            values ('customer1', 'ROLE_CUSTOMER');
```

Hangi kullanıcıların ACL işlemlerinde yer alacağını tanımlayalım:

```
insert into acl_sid (id, principal, sid)
            values (1,1,'admin1')
insert into acl_sid (id, principal, sid)
            values (2,1,'customer1')
```

ACL listeleri belirlenecek alan sınıflarını tanımlayalım:

```
insert into acl_class (id, class)
            values (1, 'com.kurumsaljava.spring.domain.Car')
insert into acl_class (id, class)
            values (2, 'com.kurumsaljava.spring.domain.Rental')
insert into acl_class (id, class)
            values (3, 'com.kurumsaljava.spring.domain.Customer')
```

Alan nesneleri ve kullanıcılar arasındaki ilişkiyi kuralım. object_id_class kullanılan alan sınıfı (aşağıdaki örnekte 2 değeri ile acl_class tablosunda yer alan Rental sınıfı kastedilmektedir.), object*id*identity alan nesnesinin sahip olduğu id değeri (aşağıdaki örnekte 100 ile rental tablosunda kayıtlı 100 id değerine sahip kayıt kastedilmektedir), owner_sid ile alan nesnesinin sahibi tanımlanmaktadır.

```
insert into acl_object_identity
    (id, object_id_class, object_id_identity,owner_sid )
    values (1,2,100,1);

insert into acl_object_identity
    (id, object_id_class, object_id_identity, owner_sid )
```

```
    values (2,2,200,1);
```

Kullanıcıların alan nesneleri üzerinde yapabilecekleri işlemleri tanımlayalım. sid kullanıcıyı tanımlarken acl_object_identity alan nesnesini, mask ise yapılacak işlem türünü tanımlamaktadır.

```
insert into acl_entry
    (id, acl_object_identity, mask,sid)
    values (1,1,16,1);

insert into acl_entry
    (id, acl_object_identity, mask,sid)
    values (2,2,1,2);
```

Alan nesneleri üzerinde yapılabilecek işlemler org.springframework.security.acls.domain.BasePermission sınıfında şu şekilde tanımlanmaktadır:

```
public static final Permission READ =
        new BasePermission(1 << 0, 'R'); // 1
public static final Permission WRITE =
        new BasePermission(1 << 1, 'W'); // 2
public static final Permission CREATE =
        new BasePermission(1 << 2, 'C'); // 4
public static final Permission DELETE =
        new BasePermission(1 << 3, 'D'); // 8
public static final Permission ADMINISTRATION =
        new BasePermission(1 << 4, 'A'); // 16
```

acl_entry tablosunda mask değeri olarak kullanılacak 1 rakamı okuma (READ), 2 rakamı yazma ya da değiştirme (WRITE), 4 rakamı oluşturma (CREATE), 8 rakamı silme (DELETE) ve 16 rakamı alan nesnesi üzerinde her türlü administrasyon (ADMINISTRATION) işlemi yapabilme anlamana gelmektedir. acl_entry tablosunda yer alan ilk kayıt ile admin1 isimli kullanıcıya 100 id değerini taşıyan Rental nesnesi üzerinde ADMINISTRATION hakkı tanınmaktadır. İkinci kayıt ile customer1 isimli kullanıcıya 200 id değerini taşıyan Rental nesnesi üzerinde READ hakkı tanınmaktadır.

Şimdi customer isimli kullanıcının bir Rental nesnesini edinmek istediğini hayal edelim. Kod 11.33 de yer alan örnekte hasPermission güvenlik ifadesi ile kullanıcının getRental() metodu tarafından geriye verilen (returnObject) nesne üzerinde WRITE işlemini yapma hakkının olup, olmadığı kontrol edilmektedir. Bu amaçla Spring Security getRental() bünyesinde

oluşan Rental nesnesinin id değişkenine bakmakta ve bu değeri kullanıcıya atanan ACL kayıtları ile kıyaslamaktadır. customer1 sadece 200 id değerine sahip Rental nesnesi için READ hakkına sahiptir. Bu sebepten dolayı kod 11.33 de yer alan getRental() metodu bu nesneyi geri vermiş olsa bile hasPermission ile WRITE hakkı kontrol edildiğinden kod 11.26 yer alan hata oluşacaktır. Kısaca kullanıcının bu alan nesnesi üzerinde WRITE işlem hakkı bulunmamaktadır ve getRental() metodunda oluşan Rental nesnesini kullanması engellenir.
Kullanılan @PostAuthorize anotasyonu getRental() metodunun koşturulmasını ve güvenlik kontrollerinin bu sayede oluşan Rental nesnesi üzerinde yapılmasını mümkün kılmaktadır.

```
Kod 11.33 - RentalServiceImpl

@PostAuthorize("hasPermission(returnObject, 'WRITE')")
public Rental getRental(long id) {
    return this.dao.get(id);
}
```

Kod 11.34 de yer alan örnekte saveRental() metodu kullanıcı ROLE_CUSTOMER rolüne sahip ise ve metot parametresi olan rental nesnesi üzerinde CREATE hakkına sahip ise koşturulur. Aksi taktirde kod 11.26 da yer alan hata oluşur. Böylece kullanıcının bu işlemi yapması engellenmiş olur.

```
Kod 11.34 - RentalServiceImpl

@PreAuthorize("hasRole('ROLE_CUSTOMER') and
            hasPermission(#rental, 'CREATE')")
public Rental saveRental(Rental rental) {
    this.dao.save(rental);
}
```

Kod 11.35 de yer alan örnekte getAll() metodu tarafından bünyesinde Rental nesnelerinin yer aldığı bir liste oluşturulmaktadır. @PostFilter bu liste içinde sadece kullanıcının READ hakkına sahip olduğu Rental nesnelerinin kalmasını sağlamaktadır. Bu durumda kullanıcı sadece üzerinde READ hakkına sahip olduğu Rental nesnelerini edinebilmektedir. Diğer Rental nesneleri listeden çıkartılmaktadır.

```
Kod 11.35 - RentalServiceImpl

@PostFilter("hasPermission(filterObject, 'READ')")
```

```
public List<Rental> getAll() {
    return this.dao.getList();
}
```

SecurityContext ve SecurityContextHolder

Login işlemi ardından kullanıcıyı temsil eden org.springframework.security.core.Authentication nesnesini bünyesinde ihtiva eden bir org.springframework.security.core.context.SecurityContext nesnesi oluşturulmaktadır. Bu nesneye uygulamanın herhangi bir yerinden ThreadLocal tipinde olduğundan SecurityContextHolder.getContext() ile erişmek mümkündür. Kod 11.36 da yer alan kod örneğinde kullanıcının sahip olduğu kullanıcı ismi Authentication nesnesinden edinilmektedir. Bu şekilde Authentication nesnesi bünyesinde yer alan metotlar kullanılarak güvenlik kontrolleri yapılabilir.

```
Kod 11.36 - IndexController

@RequestMapping(value="/login", method = RequestMethod.GET)
public String index(ModelMap model) {
    Authentication auth = SecurityContextHolder.
            getContext().getAuthentication();
    String name = auth.getName();

    model.addAttribute("username", name);
    return "index";
}
```

11. Bölüm Soruları

11.1 Bir web uygulamasında güvenlik ihtiyacının doğduğu alanlar hangileridir?

11.2 Otomatik login işlemi için kullanılan yöntem hangisidir?

11.3 Metot güvenliği sağlamak için kullanılan Spring anotasyonu hangisidir?

11.4 @Secured anotasyonunun işleyiş tarzı nasıldır?

11.5 Alan nesnelerini nasıl korunabilir?

12. Bölüm

Spring REST

Çoğu zaman web servis ve RPC (Remote Procedure Calp) ile karıştırılan REST (Representational State Transfer) HTTP spesifikasyonunun yazarı Roy Fielding tarafından oluşturulmuş, verilerin (resource) tanımlı bir metot kümesi (örneğin HTTP bünyesinde yer alan GET, POST, DELETE) aracılığı ile işlenip, oluşan veri şekillerinin (state) değişik formatlarda (XML, JSON, HTML; değişik veri formatlarının kullanılması REST'in R harfi olan representational kelimesi ile ifade edilmektedir) HTTP protokolü aracılığı ile transfer edilmesi için kullanılan bir yazılım mimari tarzıdır. REST bazlı uygulamalar için RESTful ismi de kullanılmaktadır.

REST bir API ya da çatı (framework) değildir. Sadece bir yazılım mimari tarzıdır. REST HTTP protokolüne bağımlı değildir. Başka protokoller kullanarak REST uygulamaları geliştirmek mümkündür. REST SOAP (Simple Object Access Protocol) tabanlı web servis ile doğrudan karşılaştırılamaz, çünkü kullanım alanları ve şekilleri farklıdır. REST çok kapsamlı ve yer yer karmaşık bir altyapıya sahip olabilen SOAP'a kolay kullanılabilir bir alternatif teşkil etmektedir.

REST mimarisinin çekirdek konseptlerini şu şekilde sıralayabiliriz:

- Tanımlanmış kaynaklar (Identifiable Resources)
- Tanımlı metot kümesi (Uniform Interface)
- Oturumsuz iletişim (Stateless Conversation)
- Değişik Veri formatları (Resource Representations)
- Linkler (Hypermedia)

REST bünyesinde veriler doğrudan web adres şemaları kullanılarak adreslenir. Bu sebepten dolayı REST terminolojisinde bu verilere identifiable resources (tanımlanmış ya da adreslenebilir veriler) ismi verilmektedir. Bir veri örneğin bir müşteri (Customer), bir araç (Car) ya da bir araç kiramalar işlemini temsil eden Rental nesnesi olabilir. Bir uygulama bünyesinde bu veriler alan nesnesi (domain object) iken, REST bünyesinde veri ya da kaynak olarak isimlendirilir. REST ile oluşturulan mimari türünde bu veriler üzerinde HTTP protokolü ve metotları kullanılarak işlem yapılır. Yapılan işlemlerin sonuçları kullanıcının isteği ve ihtiyaçları doğrultusunda değişik veri formatları kullanılarak transfer edilir.

REST uygulamalarında veriler üzerindeki işlemler uniform interface olarak isimlendirilen metot kümesi aracılığı ile gerçekleştirilir. Bu komutlar HTTP bünyesinde yer alan GET, POST, PUT, DELETE, HEAD ve OPTIONS

metotlarına eş gelmektedir. Kullanıcı (client) ve REST sunucusu (server) arasındaki iletişim her zaman bağımsızdır (stateless) ve oturum (session) ihtiva etmez. REST sunucusu bir kullanıcı için yaptığı işlemleri bir oturum aracılığıyla takip etmezken, kullanıcı REST sunucusundan aldığı HTML linkleri aracılığı ile durum takibi yapabilir. Örneğin POST komutu ile yeni bir müşteri nesnesi oluşturulduğunda, REST sunucusu yeni verinin lokasyonunu link olarak kullanıcıya transfer eder. kullanıcı bu link aracılığı ile yeni müşterinin bilgilerine erişebilir.

Java ekosisteminde REST tabanlı uygulamalar geliştirmek için JAX-RS 1.1 ismini taşıyan bir spesifikasyon bulunmaktadır. Bu spesifikasyonun Restlet, RESTEasy ve Jersey gibi implementasyonları mevcuttur. Spring 3 ile gelen anotasyon bazlı REST desteği bir JAX-RS implementasyonu değildir. Spring bünyesinde yer alan REST desteği daha ziyada Spring MVC kullanılarak oluşturulan bir yapıdır. JAX-RS bünyesinde kullanıcı (client) oluşturma desteği yokken, Spring REST RestTemplate sınıfı aracılığı ile kullanıcı oluşturma desteği sağlamaktadır.

REST mimarisinde verilere kaynak ismi verildiği için, bundan sonraki bölümlerde bu iki terimi eş anlamlı olarak kullanacağım.

Kaynakların REST ile Adreslenmesi

URL (Unified Resource Locator) bir kaynağı belli bir protokol (HTTP ya da FTP gibi) yardımı ile adreslemek için kullanılan tekdüze bir göstergeçtir. İnternet ya da web adresi olarak bilinir. Ama kullanımı sadece web adresleri ile sınırlı değildir.

```
http://www.rentacar.com/rental.html?id=100
```

Yukarıda yer alan URL örneginde rentacar.com alan adresi altında yer alan rental.html isimli HTML sayfası adreslenmektedir. rental.html URL aracılığı ile adreslenen kaynaktır. Soru işaretinden sonra bu kaynağa gönderilen parametreler yer almaktadır. rental.html isimli kaynağa gönderilen parametrenin ismi id ve sahip olduğu değer 100 dür. Büyük bir ihtimalle kullanılan bu URL ile 100 numarasını taşıyan araç kiralama (rental) nesnesinin içeriğe erişim tanımlanmaktadır. Ama bundan tam olarak emin değiliz. Bu URL HTTP protokolü, alan ismi, kaynak ismi ve parametre yardımı ile verinin hangi lokasyonda olduğunu tayin etmekle birlikte, hangi tür bir veriye erişimin

sağlandığını ifade etme gücünden yoksundur. Ne tür bir veriye erişimin sağlandığını kestirmenin tek ipucu id parametresidir. Bu tür bir URL verinin sahip olduğu türü belirleyici nitelikte değildir.

REST bünyesinde kullanılan URL nesnelerinin veriyi lokalize etme gücü yanı sıra, ne tür bir verinin kullanıldığını tanımlama yetenekleri de mevcuttur. Bunun bir örneğini aşağıda yer alan URL de görmekteyiz.

```
http://www.rentacar.com/rental/id/100
```

REST bazlı bir URL nesnesinde kullanılan parametre ve değeri URL nesnesinin bir parçası haline gelmektedir. Yukarıda yer alan REST URL nesnesi aracılığı ile 100 numaralı rental nesnesi adreslenmektedir. Bu URL hem verinin nerede olduğunu gösterebilmektedir hem de ne tür bir veriye erişim sağlandığını ifade edebilmektedir.

Ne tür bir işlemin yapıldığını URL nesnesine bakarak anlamamız mümkün değildir. URL nesnesi bu bilgiyi bünyesinde taşımamaktadır. Bir REST uygulamasında yapılabilecek işlemler REST metotları ile tanımlanmaktadır.

HTTP Metotları

Tipik bir REST uygulamasında bir veriyi edinmek için GET, bir veriyi değiştirmek için PUT, yeni bir veri oluşturmak için POST ve bir veriyi silmek için DELETE komutları kullanılır. Bu komutlar HTTP protokolünde yer alan metotlarla örtüşmektedir.

HTTP metotları güvenli olanlar (safe) ve tekrarlanabilir olanlar (idempotent) olarak iki guruba ayrılır. Kaynak üzerinde değişiklik yapmayan metotlar güvenlidir. İdempotent metotlar ilk kullanımda güvenli olmayabilir, yani veriyi değiştirebilirler. Lakin sonraki kullanımlarında aynı neticeyi verirler. Güvenli olan her metot idempotent iken, her idempotent metot güvenli olmayabilir.

GET

Bir verinin (kaynağın) belli bir formattaki şeklini edinmek için kullanılır. GET işlemi güvenli ve idempotenttir.

GET ile yapılan işlem kullanıcı tarafından önbellege (cache) alınabilir. Bu amaçla REST uygulama sunucusu kullanıcıya kaynağın önbellege alınabileceğini göstermek amacıyla bir ETag başlığı (header) gönderebilir.

Kullanıcı sonraki GET isteklerinde bu başlığı kullanarak, sunucudan istekte bulunabilir. Eğer kaynak değişmedi ise, sunucu 304 (Not Modified) HTTP statü kodunu geri verir. Bu değer kullanıcının daha önce GET ile edindiği kaynağı tekrar kullanabileceği anlamına gelmektedir. Bir GET istek-cevap örneği aşağıda yer almaktadır.

```
Kullanıcı isteği
------------------
GET /rental/100
HOST: www.rental.com
Accept: application/xml
...


Sunucu cevabı
---------------
HTTP/1.1 200 OK
Date: ...
Content-Length: 4546
Content-Type: application/xml

<rental>
...
</rental>
```

Bu GET örneğinde kullanıcı Accept HTTP başlığı (header) ile veriyi XML formatında edinmek istediğini belirtmektedir. Kullanıcının edinmek istediği kaynak /rental/100 adresinde yer almaktadır. Sunucu tarafından kullanıcıya transfer edilen cevapta Content-Type verinin formatını belirlemektedir. Transfer edilen veri XML formatında olup, kullanıcının isteği ile birebir örtüşmektedir. Sunucu 200 HTTP statü kodunu kullanarak kullanıcıya verinin lokalize edildiğinin sinyalini vermektedir. Bunun yanı sıra veri XML formatında cevabın gövdesinde (response body) kullanıcıya transfer edilmektedir.

POST

Yeni bir verinin (kaynağın) oluşturulması için kullanılır. Sunucu kullanıcıya gönderdiği cevapta yer alan Location HTTP başlığı (header) oluşturulan yeni verinin lokasyonunu ihtiva eder . Bir POST istek-cevap örneği aşağıda yer almaktadır.

```
Kullanıcı isteği
------------------
POST /rental
```

```
HOST: www.rental.com
Accept: application/xml

<rental>
...
</rental>

Sunucu cevabı
---------------
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Content-Type: application/xml
Location: http://rental.com/rental/101
...
```

POST yan etkileri olan bir metottur, yani güvenli değildir. POST aynı zamanda idempotent değildir.

PUT

Belli bir adresteki veriyi güncellemek ya da oluşturmak için kullanılır. Bir PUT istek-cevap örneği aşağıda yer almaktadır.

```
Kullanıcı isteği
------------------
PUT /rental/101
HOST: www.rental.com
Accept: application/xml

<rental>
...
</rental>

Sunucu cevabı
---------------
HTTP/1.1 204 No Content
Date: ...
 ...
```

PUT yan etkileri olan bir metottur, yani güvenli değildir. İlk kullanıldığında güvenli olmamasına rağmen, tekrarlandığında idempotent davranışı gösterir.

DELETE

Belli bir adresteki veriyi silmek için kullanılır. Bir DELETE istek-cevap örneği aşağıda yer almaktadır.

```
Kullanıcı isteği
------------------
DELETE /rental/101
HOST: www.rental.com
...

Sunucu cevabı
---------------
HTTP/1.1 204 No Content
Date: ...
...
```

DELETE yan etkileri olan bir metottur, yani güvenli değildir. İlk kullanıldığında güvenli olmamasına rağmen, tekrarlandığında idempotent davranışı gösterir.

Spring MVC ile REST Uygulaması

Spring dünyasında REST uygulamaları geliştirmek için Spring MVC web çatısı kullanılmaktadır. Bir REST uygulaması geliştirmek için Spring MVC bünyesinde gerekli tüm araçlar yer almaktadır. Spring MVC'nin REST uygulamaları geliştirmek için sağladığı desteği şu şekilde sıralayabiliriz:

- Controller sınıfları bünyesinde @RequestMapping anotasyonu aracılığı ile controller metodunun hangi HTTP metoduna cevap vereceği tanımlanabilir.
- @PathVariable anotasyonu kaynak adresleme şemalarında kullanılan parametre değerleri edinmek için kullanılabilir. /rental/100 örneğinde 100 rakamını @PathVariable ile bir metot parametresine dönüştürmek mümkündür.
- @RequestBody anotasyonu yardımıyla kullanıcı isteğinde yer alan veriler doğrudan Java nesnelerine dönüştürülebilir.
- Spring MVC view resolver mekanizmasıyla kullanıcıya gönderilen cevaplar XML, JSON ya da RSS gibi herhangi bir formatta olabilir.
- ContentNegotiatingViewResolver yardımıyla kullanıcı için en uygun veri tipi secimi otomatik olarak yapılabilir.
- @ResponseBody anotasyonu ile bir Java nesnesi view elementleri kullanılmadan kullanıcının talep ettiği formatta gönderilebilir. HttpMessageConverter gerekli dönüşümü yapmaktadır.
- RestTemplate sınıfı REST uygulamaları için kullanıcı (client) oluşturmak için kullanılabilir.

Spring MVC bünyesinde yer alan bu özellikleri REST uygulamaları oluşturmak için nasıl kullanabileceğimizi aşağıda yer alan dört örnek üzerinde sizlerle paylaşmak istiyorum. İlk örneğimiz bir REST kaynağını edinmek için gerekli kodu ihtiva etmektedir.

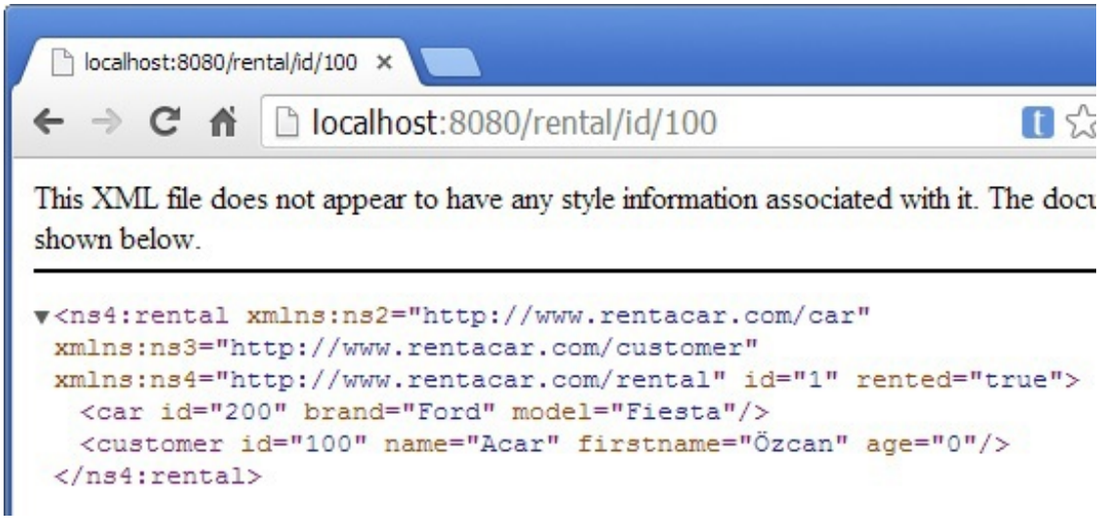
Kaynak Edinme (GET)

Şimdiye kadar geliştirmiş olduğumuz uygulama bünyesinde bir araç kiralama işlemini ifade etmek için Rental alan sınıfını kullandık. Bu sınıfı temsil eden nesneler REST dünyasında adreslemek istediğimiz kaynaklardır. Bu bölümde bir Rental nesnesinin, yani bu REST kaynağının içeriğini XML formatında edinmek için kullanılabilecek bir controller sınıfı kodunu sizinle paylaşmak istiyorum. RESTRentalController ismini taşıyan bu sınıfın kodu kod 12.1 de yer almaktadır.

```
Kod 12.1 - RESTRentalController

package com.kurumsaljava.spring.web;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;

@Controller
@RequestMapping("/rental")
public class RESTRentalController {

    @RequestMapping(value="/id/{id}", method = RequestMethod.GET,
                                  produces = "application/xml")
    public @ResponseBody Rental getRentalAsXML(@PathVariable
                          final long id) {

        Rental rental = new Rental();
        rental.setId(1);
        Customer customer = new Customer();
        customer.setId(id);
        customer.setFirstname("Özcan");
        customer.setName("Acar");
        rental.setCustomer(customer);
        Car car = new Car();
        car.setId(200);
```

```
        car.setBrand("Ford");
        car.setModel("Fiesta");
        rental.setCar(car);
        rental.setRented(true);
        return rental;
    }
}
```

@Controller ve @RequestMapping anotasyonları ile onuncu bölümde tanışmıştık. RESTRentalController sınıfı /rental adresinde yer alan işlemler için sorumlu olan controller sınıfıdır. getRentalAsXML() metodu bünyesinde bir Rental nesnesi oluşturulmaktadır. Bu nesne RentalService sınıfı aracılığı ile veri tabanında da edinilmiş bir nesne olabilirdi. Resim 12.1 de görüldüğü gibi bu nesneyi edinmek için http://localhost:8080/rental/id/100 adresine bir HTTP isteği göndermemiz gerekmektedir.



Resim 12.1

/rental/id/ ibaresinden sonra kullanılan rakam veri tabanında yer alan rental kaydının (record) id alanında taşıdığı değerdir. Bu değeri kullanarak belirli bir rental nesnesini veri tabanından çekebiliriz. Controller bünyesinde kullanıcı tarafından girilen bu değere erişebilmek için @RequestMapping anotasyonunun value elementi kullanılmaktadır. Value elementinin taşıdığı değer (/{id}) /rental/id/ adresi altındaki herhangi bir ibare anlamına gelmektedir. {id} bir URI template yani bir nevi şablondur. Spring MVC PathVariable anotasyonu yardımı ile bu değeri bir metot parametresine atamamızı mümkün kılmaktadır. Eğer kullanıcı /rental/id/100 şeklinde bir adres kullandı ise, metot parametresi olan id değişkeninin değeri 100, adres /rental/id/101 şeklinde ise, bu değer 101 olacaktır. Spring MVC değer dönüşümünü otomatik olarak gerçekleştirmektedir. Bu şekilde metot parametresi olan id değişkeni bir long

olarak tanımlanabilmektedir.

getRentalAsXML() metodunun hangi HTTP metodundan sorumlu olacağını @RequestMapping anotasyonunun metot elementi tayin etmektedir. Kod 12.1 de yer alan getRentalAsXML() metodu sadece HTTP GET kullanıldığında işlem yapmaktadır. Bunu sağlayan method=RequestMethod.GET atamasıdır. Bunun yanı sıra @RequestMapping anotasyonu bünyesinde produces = "application/xml" ibaresi kullanılmıştır. Resim 12.1 de görüldüğü gibi kullanıcı bir Rental nesnesini XML formatında edinmektedir. Bunun gerçekleşmesini sağlayan @RequestMapping anotasyonunun produces elementidir. Bu element metodun oluşturacağı Rental nesnesinin hangi formatta kullanıcıya gönderileceğini tayin eder. Kod 12.1 de produces elementi için application/xml degeri kullanıldığı için resim 12.1 de yer alan netice oluşmaktadır. Lakin produces elementi tek başına bu neticeden sorumlu değildir.

Bir Spring controller sınıfının herhangi bir nesneyi XML formatına dönüştürebilmesi için Java kodu ile XML arasında dönüşümü sağlayan bir çatıya ihtiyaç duyulmaktadır. Kod 12.1 de kullanılan örnekte bu çatı JAXB 2 (Java Architecture for XML Binding) dir. Spring konfigürasyon dosyasında <mvc:annotation-driven/> elementi kullanıldığı taktirde Spring MVC otomatik olarak Java<->XML dönüşümü için JAXB 2 çatısını kullanır. Java<->XML dönüşümü için dönüşümün yapılacağı sınıflarda JAXB anotasyonları kullanılır. Kod 12.2 de JAXB anotasyonlarını taşıyan Rental sınıfı yer almaktadır. @XmlAttribute basit sınıf değişkenlerini, @XmlElement ise sınıf tipinde olan sınıf değişkenlerini dönüştürmek için kullanılan JAXB anotasyonlarıdır.

```
Kod 12.2 - Rental

package com.kurumsaljava.spring.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.validation.Valid;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "")
@XmlRootElement(name = "rental", namespace =
                "http://www.rentacar.com/rental")
@Entity
@Table(name = "rental")
public class Rental {

    @XmlAttribute
    @Id
    @GeneratedValue
    @Column(name = "id")
    private long id;

    @XmlElement
    @Valid
    @OneToOne
    private Car car;

    @XmlElement
    @Valid
    @OneToOne
    private Customer customer;

    @XmlAttribute
    @Column(name = "rented")
    private boolean rented;
}
```

Rental nesnesinin XML formatında kullanıcıya ulaşabilmesi için kod 12.1 de @ResponseBody anotasyonu kullanılmaktadır. Yapı olarak @RequestBody anotasyonuna benzeyen @ResponseBody getRentalAsXML() metodunun geriye verdiği değerin HTTP cevap gövdesine (response body) eklenmesini sağlamaktadır. @RequestBody anotasyonunda da olduğu gibi Spring HttpMessageConverter implementasyonları yardımı gerekli dönüşümü sağlamaktadır. Kod 12.1 de XML dönüşümü için kullanılan HttpMessageConverter implementasyonu Jaxb2RootElementHttpMesssageConverter sınıfıdır.

Kaynak Oluşturma (POST)

Yeni bir REST kaynağı oluşturmak için HTTP POST metodu kullanılır. Kod 12.3 de yer alan @RequestMapping anotasyonu method değişkeni aracılığı ile kullanılan HTTP metoduna işaret etmektedir. Bu örnekte yeni bir kaynak, yani veri tabanında yeni bir kayıt oluşturulduğu için newRental() metodunun

transaksiyonel yapıda olması gerekmektedir, aksi taktirde bir hata oluşması durumunda veri bütünlüğü zarar görebilir. Bu amaçla @Transactional anotasyonu kullanılmaktadır. Kaynağı oluşturmak için kullanılan sunucu adresi http://localhost:8080/rental/create şeklindedir. Sunucu tarafında bu adresi tanımlamak için @RequestMapping anotasyonu kullanılmaktadır.

```
Kod 12.3 - RESTRentalController

@Transactional
@RequestMapping(value = "/create", method = RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void newRental(@RequestBody final Rental rental,
               final HttpServletRequest request,
               final HttpServletResponse response) {

    final Rental rentCreated = this.service.rentACar(rental);

    final String rewardPathInfo = "/rental/id/" + rentCreated.getId();
    response.addHeader("Location", this.determineLocationUri(
                           request, rewardPathInfo));
}

String determineLocationUri(final HttpServletRequest request,
               final String newPathInfo) {
    String encodedPath;
    try {
        encodedPath = UriUtils.encodePath(newPathInfo, "UTF-8");
    } catch (final UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
    final StringBuffer url = request.getRequestURL();
    url.replace(url.indexOf(request.getServletPath()),
                   url.length(), encodedPath);
    return url.toString();
}
```

Kullanıcı tarafından REST uygulamasına gönderilen verileri otomatik olarak bir Java nesnesine dönüştürmek için @RequestBody anotasyonu kullanılmaktadır. Bu kullanıcı verilerinin rental ismini taşıyan değişken olarak metot bünyesinde kullanılmasını mümkün kılmaktadır.

POST metodu ile yeni bir kaynak oluşturulduğunda, bu kaynağı temsil eden veriler kullanıcıya doğrudan gönderilmez. Bunun yerine kullanıcıya kaynağın hangi adreste erişilebilir olduğu bilgisi aktarılır. Bunu sağlamak için bir HTTP başlık (header) değişkeni olan Location degiskeni kullanılır. Kod 12.3 de yer

alan örnekte response.addHeader() ile Location degişkeni oluşturulmakta ve bu değişkene determineLocationUri() metodu ile gerekli değer ataması yapılmaktadır. determineLocationUri() tarafından oluşturulan tipik bir değer

```
http://localhost:8080/rental/id/301
```

şeklinde olabilir. 301 oluşturulan yeni kaynağın id değeridir. Veri tabanında yer alan rental tablosunda baktığımızda 301 id değerini taşıyan bir kayıt bulabiliriz. Kullanıcı POST ile kaynağı oluşturduktan sonra GET metodunu kullanarak

```
http://localhost:8080/rental/id/301
```

adresindeki POST ile oluşturulmuş bu kaynağa erişebilir.

Kaynağı GET ile edinmek için tekrar resim 12.1 de görüldüğü gibi web tarayıcısını kullanabiliriz. Programsal olarak kaynağı edinmek, oluşturmak ya da değiştirmek istiyorsak RestTemplate sınıfını kullanmamız daha yerinde olacaktır.

Kaynak Silme (DELETE)

DELETE metodu kullanılarak bir kaynağın yok edilmesi sağlanabilir. Kod 12.3.1 de yer alan örnekte delete() metodu HTTP DELETE metodu kullanıldığında devreye girmektedir. Kullanıcı tarafından gönderilen id parametresi @PathVariable ile bir metot parametresine dönüştürülmekte, bu değer ile deleteRental() isimli servis metodu aracılığı ile veri yok edilmektedir. Görüldüğü gibi delete() metodu void tipinde olup, herhangi bir değer geri vermemektedir, çünkü geriye verilecek bir veri kalmamıştır. Bunu kullanıcıya bildirmek için @ResponseStatus aracılığı ile NO_CONTENT statü kodu oluşturulmaktadır.

Silinmesini istediğimiz bir kaynağı DELETE metodunu kullanarak silebiliriz.

```
Kod 12.3.1 - RESTRentalController

@RequestMapping(value = "/id/{id}", method =
               RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable final long id) {
       this.service.deleteRental(id);
}
```

Kaynak Güncelleme (PUT)

Bir kaynağı güncellemek için PUT metodu kullanılır. PUT yapı olarak DELETE ile benzerlik göstermektedir. DELETE örneği için yazdıklarım PUT için de geçerliliğini korumaktadır.

```
Kod 12.3.2. - RESTRentalController

@RequestMapping(value = "/id/{id}", method =
               RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void update(@PathVariable final long id) {
       this.service.updateRental(id);
}
```

Statü Kodları

Web uygulamaları 200 OK, 404 Not Found (bulunamadı), 302/303 yönlendirme ve 500 hata gibi statü kodlarını kullanarak kullanıcıya işlem sonucunu bildirirler. REST bazlı uygulamalar da bu ve buna benzer statü kodları aracılığı ile kullanıcılarla iletişim sağlayabilirler. Java kodu içinde doğrudan statü kodunu belirlemek için @ResponseStatus anotasyonu kullanılabilir. Kod 12.3 örneğinde HttpStatus.CREATED 201 statü kodudur ve kullanıcıya kaynağın oluşturulduğu mesajını vermektedir. Web ve REST bazlı uygulamalarda kullanılabilecek bazı statü kodları şunlardır:

- ***200***: Başarıyla tamamlanmış bir GET işlemi için kullanılır.
- ***201***: POST ya da PUT ile yeni bir kaynak oluşturulduğunda kullanılır. REST uygulamalarında yeni kaynağın adresi Location HTTP başlığında yer alır.
- ***204***: Cevabın (response) boş olduğunu gösterir. Bunu Java metotlarında kullanılan void olarak düşünebiliriz. PUT ya da DELETE yapıldığında bir değer geri verilmez. Bunun yerine statü kodu 204 olarak atanır.
- ***404***: Talep edilen kaynak tanımlı adreste bulunamadığı zaman kullanılır.
- ***405***: Eğer seçilen kaynak kullanılan HTTP metodunu desteklemiyorsa kullanıcıya 405 ile durum bildirilir.
- ***409***: Kaynak üzerinde değişiklik yaparken bir hata oluştuğunu bildirir.
- ***500***: Uygulama bünyesinde genel bir hata oluşması durumunda kullanılan statü kodudur.

org.springframework.http.HttpStatus sınıfı bünyesinde statü kodları tanımlıdır.

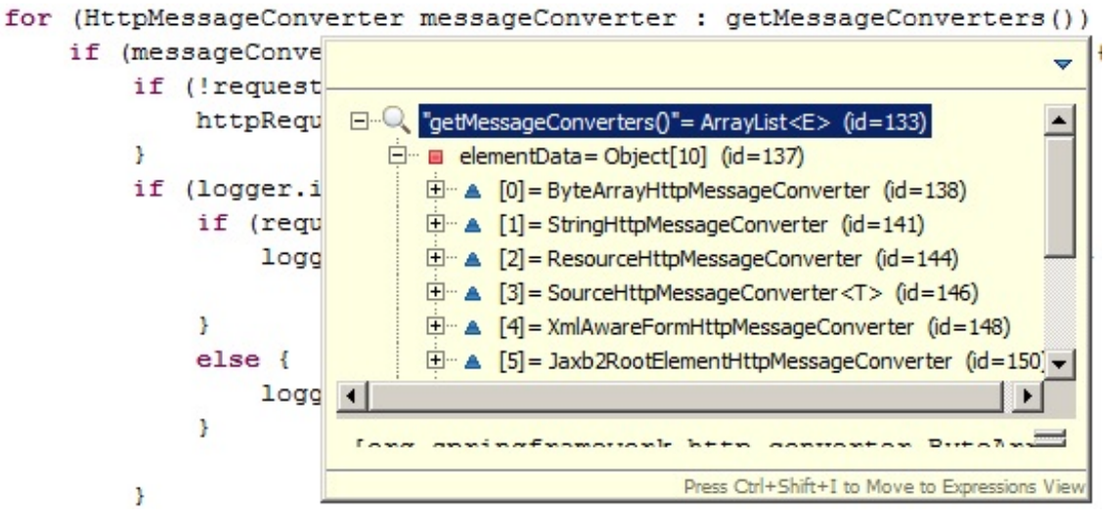
RestTemplate Kullanımı

Yeni bir REST kaynağı oluşturmak için sunucuya bir Rental nesnesini XML formatında göndermemiz gerekmektedir. Bu amaçla kod 12.4 de yer alan post() metodunu kullanabiliriz. post() bünyesinde yeni bir Rental nesnesi oluşturulmakta ve bu nesne RestTemplate üzerinden postForLocation() metodu aracılığı ile sunucuya gönderilmektedir.

```
Kod 12.4 - RestClient

package com.kurumsaljava.spring.client;
import java.net.URI;
import org.springframework.web.client.RestTemplate;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;

public class RestClient {

    final static RestTemplate client = new RestTemplate();

    public static void main(final String[] args) {
        post();
    }

    private static void post() {
        final Rental rental = createRentalObject();
        final URI uri = client.postForLocation("
        http://localhost:8080/rental/create", rental);
        get(uri.toString());
    }

    private static Rental createRentalObject() {
        final Rental rental = new Rental();
        final Customer customer = new Customer();
        customer.setFirstname("Özcan");
        customer.setName("Acara");
        rental.setCustomer(customer);
        final Car car = new Car();
        car.setId(200);
        car.setBrand("Ford");
        car.setModel("Fiesta");
        rental.setCar(car);
        rental.setRented(true);
        return rental;
    }

    private static void get(final String uri) {
```

```
        final Rental rental = client.getForObject(uri,
                Rental.class);
        System.out.println(rental.getCustomer().getName());
    }
}
```

Yeni bir kaynak oluşturmak için kullanılan adres http://localhost:8080/rental/create şeklindedir. RestTemplate bir Rental sınıfı tarafından temsil edilen verileri otomatik olarak XML formatına dönüştürür. Bunu JAXB çatısı aracılığıyla gerçekleştirir. Bu dönüşümü yapmak için kullandığı implementasyon Jaxb2RootElementHttpMessageConverter sınıfıdır. Resim 12.3 de RestTemplate sınıfının veri dönüşümünü sağlamak için mevcut tüm HttpMessageConverter implementasyonlarını denediği görülmektedir. Rental sınıfı JAXB anotasyonlarını taşıdığı için, RestTemplate Jaxb2RootElementHttpMessageConverter implementasyonunu seçerek, dönüşümü gerçekleştirmekte ve verileri REST sunucusuna aktarmaktadır.



Resim 12.3

postForLocation() metodu URI tipinde bir değer geri vermektedir. Bu yeni oluşturulan kaynağın adresini temsil etmektedir. RestTemplate bu değeri sunucu tarafından gönderilen cevap (response) bünyesinde yer alan Location isimli HTTP başlığından edinmektedir. uri.toString() ile yeni kaynağın adresini edinebiliriz. get() metodu bünyesinde bulunan client.getForObject() ile daha önce oluşturduğumuz yeni kaynağı edinebiliriz. Sunucu tarafından Location değişkeni ile geri gönderilen değerin http://localhost:8080/rental/id/301 olduğunu düşündüğümüzde, client.getForObject() bu adreste yer alan veriyi edinmek için REST uygulamasına istekte bulunacaktır. Uygulama gelen bu isteği kod 12.5 de yer alan getRental() controller metoduna yönlendirir. Bu metot 301 değerini taşıyan id değişkeni üzerinden gerekli Rental nesnesini veri

tabanından edinir ve geriye verir. Spring MVC çatısı yine JAXB aracılığı ile bu nesneyi XML formatına dönüştürerek, kullanıcıya geri gönderir.

```
Kod 12.5 - RESTRentalController

@RequestMapping(value = "/id/{id}",
        method = RequestMethod.GET, produces = "application/xml")
 public @ResponseBody Rental getRental(
                @PathVariable final long id) {
       return this.service.getRentalById(id);
}
```

Bir REST kaynağını değiştirmek için kullanılabilecek metot 12.6 da yer almaktadır. Bu örnekte bir Java nesnesi olan rental PUT metodu kullanılarak REST sunucusuna aktarılmaktadır. Sunucu tarafında bulunan RESTRentalController sınıfı PUT metoduna cevap verecek bir metoda sahip ise bu metot bünyesinde rental nesnesinin ihtiva ettiği verileri üzerinde gerekli değişiklik yapılabilir.

```
Kod 12.6 - RestClient

private static void put(Rental rental) {
    client.put("http://localhost:8080/rental/id/"
                            +rental.getId(), rental);
}
```

Bunun yanı sıra kaynağı bulmak için gerekli parametreleri bir Map içinde sunucuya aktarmak mümkündür. Kod 12.7 de bunun bir örneği yer almaktadır. Birçok parametreli kullanıcı isteklerinde tercih edilmesi gereken yöntem bu olmalıdır.

```
Kod 12.7 - RestClient

private static void put(Rental rental) {
    Map<String,String> params=new HashMap<String,String>();
    params.put("id", rental.getId());
    client.put("http://localhost:8080/rental/id/{id},
           rental, params);
}
```

Bir kaynağı silmek için 12.8 de yer alan delete() metotlarından birisi kullanılabilir.

```
Kod 12.8 - RestClient
```

```
private static void put(long customerId, long rentalId) {
    client.delete("http://localhost:8080/rental/id/"+rentalId);
    // ya da
    client.delete("http://localhost:8080/rental/id/{id}, rentalId);
    // ya da
    client.delete("http://localhost:8080/customer/{customerId}/
                rental/{rentalId}, customerId, rentalId);
}
```

Kaynak Formatını Belirleme Stratejisi

Tipik bir Spring MVC ya REST uygulamasında controller metotları bünyesinde verileri tutmak için alan nesneleri kullanılır. Örneğin kod 12.3 de yer alan newRental() metodu bünyesinde kullanıcı tarafından gönderilen verilere metot parametresi olarak rental değişkeni üzerinden erişmekteyiz. Kullanıcı tarafından gönderilen veriyi bir Java nesnesine dönüştürme işlemini Spring MVC HttpMessageConverter türevleri ile gerçekleştirmektedir. Controller sınıfı yapılan bu işlemden habersizdir ve kendisine sadece gerekli veriler Spring tarafından enjekte edilmektedir. Aynı şekilde bir controller metodu kullanıcıya göndermek istediği verileri çoğunlukla bir Java nesnesi olarak geri verecektir. Bunun bir örneğini kod 12.1 de görmekteyiz. Metot son bulduktan sonra Spring yine devreye girerek gerekli dönüşümü yapacak ve verileri kullanıcıya aktaracaktır.

Kullanıcı kaynağı hangi formatta edinmek istiyorsa, bu formatı sağlayan controller metodunu adreslemesi gerekmektedir. http://localhost:8080/rental/id/100 örneginde Spring MVC çatısı veriyi kullanıcıya kod 12.5 de yer alan getRental() metodu aracılığı XML formatında göndermektedir. Eğer kullanıcı kaynağı JSON formatında edinmek isterse, controller bünyesinde verinin JSON formatına dönüştürülmesi gerektiğini belirten yeni bir metot oluşturulabilir. Örneğin

```
http://localhost:8080/rental.json/id/100
```

şeklindeki bir istek kod 12.9 da yer alan getRentalAsJson() tarafından işlem görecek ve Spring MVC metot bünyesinde oluşturulan rental nesnesini JSON formatında kullanıcıya aktaracaktır.

```
Kod 12.9 - RESTRentalController
```

```
    @RequestMapping(value="/rental.json/id/{id}",
                 method = RequestMethod.GET,
                                produces = "application/json")
    public @ResponseBody Rental getRentalAsJson(
                  @PathVariable final long id) {
            ...
}
```

Kullanıcının istediği kaynağı belli bir formatta edinmeye çalışması, hem kullanıcı kodunu daha karmaşıklaştırmaktadır, hem de kaynaklara olan erişim adreslerinin yönetimini gerektirmektedir. Bundan daha iyi bir yöntem, kullanıcı isteğini analize ederek, kullanılan adres özelliklerinden faydalanıp, kullanıcının ne tür bir formatı tercih ettiğini kestirmeye çalışmaktır. Spring'in kullanıcının talep ettiği veri formatını belirlemek için takip edeceği sıra şu şekilde olacaktır:

- Eğer kullanıcı isteğinde belirli bir ek kullanıldıysa, bu ek verinin formatını tayin eder. http://localhost:8080/rental.html örneğinde kullanılan .html eki, verinin HTML formatında geri gönderilmesini sağlayacaktır.
- Kullanıcı isteğinde format isminde bir parametre kullanıldığında, kullanılan veri formatını bu parametrenin değeri belirler. http://localhost:8080/rental?format=xml örneğinde sunucu veriyi XML formatına dönüştürür ve kullanıcıya aktarır. Standart ayarlarda bu parametre kullanım dışıdır, aktif hale getirildiğinde veri formatını tespit etmek için kullanılan ikinci yöntemdir.
- Son seçenek olarak bir HTTP başlığı (header) olan Accept parametresinin değeri kontrol edilir. Accept: application/json örneğinde kullanıcı veriyi JSON formatında edinmeyi arzulamaktadır.

Spring'in saydığımız sırada veri formatını belirleme işlemini yapabilmesi için Spring 3.2 ile gelen ContentNegotiationManagerFactoryBean sınıfını kod 12.10 daki gibi konfigüre etmemiz gerekmektedir. DefaultContentType değişkeni kullanıcının talep ettiği veri formatı belirlenemediği taktirde sunucu tarafından seçilen veri formatıdır. Bu konfigürasyonun tüm Spring MVC uygulaması bünyesinde geçerli olması için annotation-driven elementinin content-negotiation-manager element özelliğini oluşturduğumuz contentNegotiationManager nesnesini atamamız yeterli olacaktır.

```
Kod 12.10 - mvc-config.xml

<bean id="contentNegotiationManager"
       class="org.springframework.web.accept.
               ContentNegotiationManagerFactoryBean">
```

```
      <property name="defaultContentType"
               value="application/xml" />
    </bean>

<mvc:annotation-driven
       content-negotiation-manager="
                       contentNegotiationManager" />
```

ContentNegotiationManagerFactoryBean kod 12.11 de görüldüğü şekilde detaylı konfigürasyon imkanları sunmaktadır. FavorPathExtension false olduğu taktirde veri formatını belirleme işleminde kullanıcı isteğinde yer alan herhangi bir ek (örnegin .html) dikkate alınmaz. FavorParameter true olduğu taktirde kullanıcı isteğinde format parametresinin değeri değerlendirilir. ParameterName kullanılan parametrenin ismini tanımlar. Kod 12.11 de yer alan örnekte format parametresinin ismi mediaType olarak belirlenmektedir. Buna göre

```
http://localhost:8080/rental?mediaType=json
```

şeklinde bir istek, verinin JSON formatında geri gönderilmesini sağlayacaktır. Accept HTTP başlığının sahip olduğu değeri göz ardı etmek için ignoreAcceptHeader değişkenini kullanılabilir. UseJaf değişkeninin true olması veri formatını belirlemek için JAF (JavaBeans Activation Framework) çatısının kullanımını sağlar. Bu durumda activation.jar dosyasının classpath içinde olması gerekmektedir. Eğer useJaf false ise, mediaTypes değişkeni altında kabul edilebilecek veri formatları (json ve xml) yer almaktadır. DefaultContentType değişkeni hiçbir eşleme yapılamadığı taktirde kullanılacak veri formatını belirler. Bu JSON formatıdır.

```
Kod 12.11 - mvc-config.xml

<bean id="contentNegotiationManager"
            class="org.springframework.web.accept.
              ContentNegotiationManagerFactoryBean">
    <property name="favorPathExtension" value="false" />
    <property name="favorParameter" value="true" />
    <property name="parameterName" value="mediaType" />
    <property name="ignoreAcceptHeader" value="true"/>
    <property name="useJaf" value="true"/>
    <property name="defaultContentType" value="application/json" />

    <property name="mediaTypes">
            <map>
                    <entry key="json" value="application/json" />
```

```
                    <entry key="xml" value="application/xml" />
          </map>
    </property>
</bean>
```

REST Uygulamalarında Hata Yönetimi

Spring 3.2 öncesinde Spring MVC uygulamalarında hata yönetimi HandlerExceptionResolver ve @ExceptionHandler ile yapılmaktadır. Spring 3.2 ile oluşturulan @ControllerAdvice anotasyonu hata yönetiminin tüm uygulama genelinde geçerli olacak şekilde yapılmasını mümkün kılmaktadır. Bu bölümde her iki yönteminde kullanılış tarzını inceleyeceğiz.

@ExceptionHandler İle Controller Bazlı Hata Yönetimi

@ExceptionHandler controller bazında hata yönetimi yapmak için kullanılır. Kod 12.12 de yer alan örnekte IOException oluşması durumunda hata yönetimini yapacak olan metot handleIOException() metodudur. Bu metot bünyesinde hata sayfasına yönlendirme yapılmaktadır. Bunun yanı sıra @ResponseStatus anotasyonu ile 500 değerini taşıyan statü kodu oluşturulmaktadır.

```
Kod 12.12 - mvc-config.xml

public class RESTRentalController{
    ...
    @ExceptionHandler(IOException.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
     public RedirectView handleIOException(IOException ex)
                          throws IOException {
           RedirectView redirectView =
                   new RedirectView("rental/ioexception");
           redirectView.addStaticAttribute("errorMessage",
                          ex.getMessage());
           return redirectView;
    }
}
```

Bu tür bir hata yönetimi yapmanın dezavantajı, @ExceptionHandler ile tanımlanan metodun bulunduğu controller sınıfı bünyesinde işlem yapabilmesidir. handleException() tüm uygulama bünyesinde hata yönetimi için kullanılamaz. Bu sorunu aşmak için handleException() metodu bir üst controller sınıfında tanımlanır. Bu şekilde tüm alt controller sınıflar bu metodu

kullanabilirler. Üstsınıfı genişletmek mümkün olmadığı taktirde, bu tür hata yönetiminden faydalanmak yine mümkün olmayacaktır. Bu bizim daha global bir hata yönetimi mekanizmasına ihtiyaç duyduğumuz anlamına gelmektedir. Global, yani tüm uygulama genelinde hata yönetimi yapmak için HandlerExceptionResolver mekanizmasını kullanabiliriz.

HandlerExceptionResolver Bazlı Hata Yönetimi

HandlerExceptionResolver uygulama genelinde oluşan tüm hataların (Exception) yakalanarak, işlenmesini sağlayan bir mekanizmadır. Aşağıda yer alan implementasyonları mevcuttur:

- ***ExceptionHandlerExceptionResolver*** - Spring 3.1 ile gelen ve DispatcherServlet bünyesinde varsayılan HandlerExceptionResolver implementasyonudur. Controller bazlı hata yönetimi sağlayan @ExceptionHandler kullanıldığında devreye girer.
- ***DefaultHandlerExceptionResolver*** - Spring 3.0 ile gelen bu HandlerExceptionResolver implementasyonu uygulama bünyesinde oluşan hata türüne göre statü kodunu atar. Bu sınıf MVC isim alanında yer aldığı için konfigürasyon dosyasında tanımlanmasına gerek yoktur. Liste 12.1 deki listede bazı hata türleri ve DefaultHandlerExceptionResolver tarafından atanan statü kodları yer almaktadır. DefaultHandlerExceptionResolversınıfı hata durumunda sadece statü kodunu atamakla yetinir. Statü kodu haricinde kullanıcıya herhangi bir bilgi gönderilmez. Kullanıcının oluşan hatadan detaylı olarak haberdar olabilmesi için sadece statü kodunun atanması yeterli değildir. Oluşan hatanın çıktısı (stacktrace) kullanıcıya hata mesajı (error response) olarak gönderilmelidir. Bu amaçla kod 12.12 de yer alan yapı kullanılabilir.
- ***ResponseStatusExceptionResolver*** - @ResponseStatus ile tanımlanan değer statü kodu olarak atar. DefaultHandlerExceptionResolver implementasyonunda olduğu gibi hata mesajını kullanıcıya gönderilen cevaba (response) eklemez.
- ***SimpleMappingExceptionResolver*** - SimpleMappingExceptionResolver hatayı tanımlayan sınıfın ismini kullanarak hata gösterimi yapmak için view elementini seçer (bknz kod 10.31). Bu REST uygulamaları için uygun olmayan bir hata yönetimi türüdür.

Liste 12.1:

- BindException - 400 (Bad Request)
- ConversionNotSupportedException - 500 (Internal Server Error)
- HttpMediaTypeNotAcceptableException - 406 (Not Acceptable)
- HttpMediaTypeNotSupportedException - 415 (Unsupported Media Type)
- HttpMessageNotReadableException - 400 (Bad Request)
- HttpMessageNotWritableException - 500 (Internal Server Error)
- HttpRequestMethodNotSupportedException - 405 (Method Not Allowed)
- MethodArgumentNotValidException - 400 (Bad Request)
- MissingServletRequestParameterException - 400 (Bad Request)
- MissingServletRequestPartException - 400 (Bad Request)
- NoSuchRequestHandlingMethodException - 404 (Not Found)
- TypeMismatchException - 400 (Bad Request)

@ControllerAdvice Bazlı Hata Yönetimi

Spring 3.2 ile gelen @ControllerAdvice ile merkezi bir noktadan ve yukarıda yer alan eksiklikleri giderecek şekilde hata yönetimi yapmak mümkün hale gelmiştir. @ControllerAdvice @ExceptionHandler anotasyonu kullanılarak yapılan tanımlamaların tüm uygulama bünyesinde geçerli olmasını sağlamaktadır.

```
Kod 12.13 - RentalExceptionHandler

@ControllerAdvice
public class RentalExceptionHandler extends
               ResponseEntityExceptionHandler {

    @ExceptionHandler(value = { IllegalArgumentException.class,
                   IllegalStateException.class })
     ResponseEntity<Object> handleException(RuntimeException ex,
                   WebRequest request) {
           String response = "Bu bir hata mesajidir.";
           return handleExceptionInternal(ex, response,
               new HttpHeaders(), HttpStatus.CONFLICT, request);
    }
}
```

Kod 12.13 de yer alan handleException() metodu uygulama bünyesinde oluşan tüm IllegalArgumentException ve IllegalStateException türündeki hatalardan sorumludur. @ExceptionHandler bünyesinde bu iki exception sınıfı haricinde başka hata sınıfları da tanımlanabilir. @ControllerAdvice anotasyonu RentalExceptionHandler sınıfını hata yönetiminin merkezinde olan sınıf olarak

tayin etmektedir. Bir HttpEntity türevi olan ResponseEntity ile kullanıcıya hem oluşan hata mesajı, hem de hatayı tanımlayan statü kodu taşınmaktadır.

12. Bölüm Soruları

12.1 REST uygulamalarında yapılacak işlemler nasıl tanımlanır?

12.2 İdempotent olan bir REST metotları hangi özelliğe sahiptir?

12.3 REST uygulamaları geliştirilirken kullanılan Spring modülü hangisidir?

12.4 Bir REST uygulamasında kullanıcı oluşturduğu veriyi nasıl edinebilir?

12.5 HTTP Statü kodu oluşturmak için kullanılan anotasyon hangisidir?

13. Bölüm

Spring Remoting

İki değişik uygulamanın veri alışverişi yapabilmeleri için belirli arayüzler aracılığı ile entegre edilmeleri gerekir. Bu entegrasyon için kullanılabilecek değişik teknolojiler mevcuttur. Bunlardan bazıları:

- Remote Method Invocation (RMI)
- HTTP
- JMS (Java Messaging Service)
- WS (Web Service)

Bu teknolojileri kullanarak entegrasyon yapmış programcılar, bu tür entegrasyonların kullanılan entegrasyon teknolojisine bağımlı olarak ne kadar güç bir hal alabileceğini bilirler. Çoğu zaman entegrasyonu gerçekleştirmek için fazladan kod yazılması ve konfigürasyon yapılması gerekir. İşletme mantığı haricinde entegrasyonu sağlamak için yazılması gereken koda plumbing code ismi verilmektedir. Plumbing code entegrasyon detaylarını gün ışığına çıkardığı için anlaşılması zor ve karmaşık bir yapıda olabilir.

Her entegrasyon teknolojisi beraberinde kendine has entegrasyon şablonları (integration pattern) getirir. Bu çoğu zaman entegrasyon kodunun belli bir programlama modeline uygun olarak geliştirilmesi anlamına gelmektedir. Örneğin RMI kullanıldığında servis interface sınıflarının java.rmi.Remote interface sınıfını genişletmeleri (extend) gerekmektedir. Tek bir programlama modeline sadık kalarak bilumum entegrasyon teknolojilerini kullanmak programcının hayatını çok daha kolaylaştırdı.

Programcı olarak biz plumbing code yazılmasını önleyen, servisleri deklaratif bir şekilde konfigüre edip, kullanıma sunan ve tek bir programlama modeli aracılığı ile entegrasyon teknolojilerinin kullanımını mümkün kılan bir yapıya ihtiyaç duymaktayız. Böyle bir yapı Spring'in bir parçası olan Spring Remoting modülüdür.

Spring Remoting tamamen konfigürasyon bazlı bir yaklaşım kullanmaktadır. Mevcut POJO servis sınıfları sadece konfigüre edilerek, kodu değiştirmek zorunda kalmadan sunucuya (server) dönüştürülebilir. Bu yaklaşım plumbing code oluşmasını önlemektedir. Spring arka planda kullanılan entegrasyon teknolojinin gerekli kıldığı programlama modeline uyan altyapıyı oluşturmaktadır. Bu programcının sadece işletme mantığının geliştirilmesine konsantre olmasını kolaylaştırmaktadır.

Spring Remoting kullanıcı (client) tarafında da entegrasyon teknolojilerinin kullanımını kolaylaştıran elementler ihtiva etmektedir. Mevcut FactoryBean

sınıfları kullanılarak sunucularla olan iletişim sağlayan proxy nesneler oluşturulabilmektedir.

Spring Remoting tarafından desteklenen entegrasyon teknolojileri şunlardır:

- RmiProxyFactoryBean ve RmiServiceExporter aracılığı ile RMI (Remote Invocation Interface) teknolojisi.
- HessianProxyFactoryBean ve HessianServiceExporter aracılığı ile Hessian protokolü.
- BurlapProxyFactoryBean ve BurlapServiceExporter aracılığı ile Burlap protokolü.
- HttpInvokerProxyFactoryBean ve HttpInvokerServiceExporter aracılığı ile HTTP protokolü.
- JAX-RPC bazlı Webservice teknolojisi.
- JAX-WS bazlı Webservice teknolojisi.
- JmsInvokerProxyFactoryBean ve JmsInvokerServiceExporter aracılığı ile JMS (Java Messaging System) teknolojisi.

Kitabın bu bölümünde Spring Remoting ile RMI, Hessian, Burlap, JAX-WS ve HTTP teknolojilerinin kullanımını hem sunucu hem de kullanıcı tarafında örnekler ile yakından inceleyeceğiz.

RMI Service Exporter Kullanımı

Spring dekoratör tasarım şablonunu kullanarak, mevcut POJO sınıflarını RMI sunucusu haline dönüştürebilmektedir. Bu bölümde araç kiralama uygulamasının servis katmanında yer alan RentalService interface sınıfını implemente eden RentalServiceImpl sınıfını Spring RMI exporter yardımı ile bir RMI sunucusuna dönüştüreceğiz.

RentalService interface sınıfı kod 13.1, RentalServiceImpl sınıfı kod 13.2 de yer almaktadır.

```
Kod 13.1 - RentalService

public interface RentalService {

    public Rental rentACar(Rental rental);

    public Rental getRentalById(long id);
}
```

```
Kod 13.2 - RentalServiceImpl

package com.kurumsaljava.spring.service;
import org.springframework.transaction.annotation.Transactional;
import com.kurumsaljava.spring.dao.CarRepository;
import com.kurumsaljava.spring.dao.CustomerRepository;
import com.kurumsaljava.spring.dao.RentalRepository;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;

public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;
    private CarRepository carRepository;

    public RentalServiceImpl() {
    }

    @Override
    @Transactional
    public Rental rentACar(final Rental rental) {

        final Customer dbCustomer = customerRepository
                .getCustomerByName(rental.getCustomer()
                        .getName());

        if (dbCustomer == null) {
            customerRepository.save(rental.getCustomer());
        }

        final Car car = carRepository.findCarById(1);

        rental.setCar(car);
        rental.setRented(true);
        rental.setCustomer(rental.getCustomer());
        rentalRepository.save(rental);
        return rental;
    }

    @Override
    @Transactional
    public Rental getRentalById(final long id) {

        return rentalRepository.get(id);
```

```
    }

    public void setCustomerRepository(
        final CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(
        final RentalRepository rentalRepository) {
        this.rentalRepository = rentalRepository;
    }

    public void setCarRepository(final CarRepository carRepository) {
        this.carRepository = carRepository;
    }
}
```

Eğer Spring olmadan RMI teknolojisini kullanmak isteseydik RentalService interface sınıfının java.rmi.Remote interface sınıfını extend ederek bir remote interface haline gelmesi gerekirdi. Görüldüğü gibi RMI teknolojisi programcıyı sahip olduğu programlama modeline uymaya zorlamaktadır. Bunun yanı sıra RentalService interface sınıfında yer alan her metot throws ile java.rmi.RemoteException tipinde bir hatayı fırlatabileceğini belirtmek zorundadır. Spring Remoting ile java.rmi.Remote ve java.rmi.RemoteException sınıflarını kullanma zorunluluğu ortadan kalkmaktadır. Spring Remoting bünyesinde yer alan RmiServiceExporter sınıfı yardımı ile RentalService ve implementasyonu olan RentalServiceImpl sınıflarını kod değişikliği yapmadan RMI sunucusu olarak kod 13.13 de ki gibi tanımlayabiliriz.

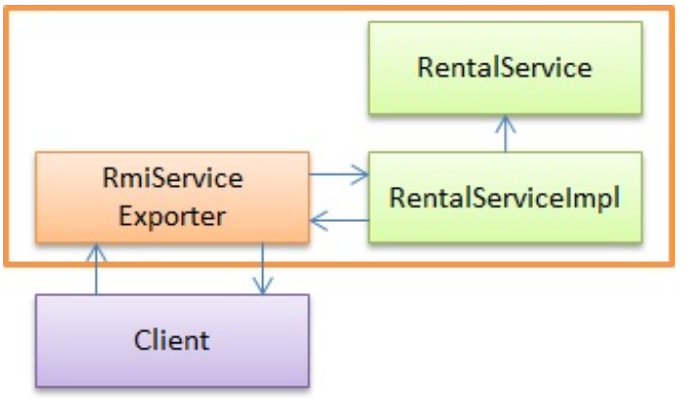
```
Kod 13.3 - rmi-server-config.xml

<bean
    class="org.springframework.remoting.rmi.
        RmiServiceExporter">
    <property name="service" ref="rentalService"/>
    <property name="serviceInterface" value="com.kurumsaljava.
            spring.service.RentalService"/>
    <property name="serviceName" value="rentalService"/>
    <property name="alwaysCreateRegistry" value="true"/>
</bean>
```

RmiServiceExporter sınıfı kendisine service aracılığı ile verilen bir POJO sınıfı RMI sunucuna dönüştürmektedir. Kod 13.3 de yer alan örnekte bu RentalServiceImpl sınıfıdır. Bunu gerçekleştirirken kullandığı interface sınıfı

serviceInterface ile tanımlanmış olan RentalService sınıfıdır. ServiceName element özelliği oluşturulan RMI nesnesine verilen isimdir. Kullanıcılar bu ismi kullanarak RMI nesnesine erişim sağlayabilirler. Kod 13.3 de yer alan örnekte RmiServiceExporter aracılığı ile oluşturulan RMI nesnesine erişim rmi://localhost:1099/rentalService adresi üzerinden olacaktır. AlwaysCreateRegistry element özelliği true değerine sahip olduğu için yeni bir RMI registry oluşmasını sağlayacaktır. Eğer çalışan mevcut bir RMI registry varsa, port already in use: 1099; hatası oluşur. RMI registry fabrika ayarlarına göre 1099 numaralı port üzerinde çalışır. Bu portu değiştirmek için

```
<property name="registryPort" value="2000"/>
```

registryPort element özelliği kullanılabilir.



Resim 13.1

Oluşturduğumuz, daha doğrusu konfigüre ettiğimiz bu RMI sunucusunu kod 13.4 de görüldüğü gibi çalışır hale getirebiliriz.

```
kod 13.4 - RmiServerMain

public class RmiServerMain {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext(
            "rmi-server-config.xml");
    }
}
```

Aşağıda yer alan ekran çıktısı rentalService ismini taşıyan bir RMI nesnesinin 1099 numaralı port üzerinde aktif olan RMI registry bünyesinde yer aldığını göstermektedir.

```
INFO : RmiServiceExporter - Creating new RMI registry
```

```
INFO : RmiServiceExporter - Binding service 'rentalService'
            to RMI registry:
RegistryImpl[UnicastServerRef [liveRef:
            [endpoint:[192.168.1.101:1099](local),
            objID:[0:0:0, 0]]]]
```

Spring RMI exporter özelliklerini şu şekilde özetleyebiliriz:

- Herhangi bir POJO sınıfı kod değişikliği yapmadan RMI sunucu haline dönüştürebilir.
- Kodun RMI modeline uygun olma zorunluluğu yoktur, yani servis interface sınıfı java.rmi.Remote sınıfını genişletmek zorunda değildir. Bunun sebebi resim 13.1 de görüldüğü gibi tüm RMI trafiğini RmiServiceExporter sınıfının yönetmesidir.
- RMI compiler (rmic) kullanarak client stub ve server skeleton sınıfları oluşturma zorunluluğu bulunmamaktadır.
- Otomatik olarak mevcut RMI registry nesnesini keşfeder ya da bulamadı ise yeni bir RMI registry oluşturur.
- RmiServiceExporter ile oluşturulan RMI nesnesine mevcut güvenlik ve transaksiyon ögelerini (security context, transaction context propagation) delege edecek standart bir yöntem bulunmamaktadır. Lakin bu Spring'in sunduğu entegrasyon noktaları (hooks) kullanılarak saglanabilir. Bu Spring'e has kod yazılması ya da konfigürasyon yapılması anlamına gelmektedir.

RMI Client Kullanımı

Spring Remoting bünyesinde kullanıcı tarafında proxy (client-side proxy) oluşturarak mevcut RMI nesnelerine erişimi sağlayabilen RmiProxyFactoryBean isminde bir sınıf bulunmaktır. Bir önceki bölümde RmiServiceExporter aracılığı ile oluşturduğumuz rmi nesnesine kod 13.5 de yer alan RmiProxyFactoryBean konfigürasyonu ile erişebiliriz.

```
kod 13.5 - rmi-client-config.xml

<bean id="rentalRmiProxy"
        class="org.springframework.remoting.rmi.
                RmiProxyFactoryBean">
    <property name="serviceInterface"
            value="com.kurumsaljava.spring.service.RentalService"/>
    <property name="serviceUrl"
```

```
                value="rmi://127.0.0.1:1099/rentalService"/>
    <property name="refreshStubOnConnectFailure"
        value="true"/>
</bean>
```

RmiProxyFactoryBean parametre olarak serviceInterface aracılığı ile tanımlanan service interface sınıfının ismine ve serviceUrl aracılığı ile RMI nesnesinin lokasyonuna ihtiyaç duymaktadır. True olan refreshStubOnConnectFailure parametre RMI nesnesine bağlantı koptuğunda, bağlantının tekrar oluşturulmasını sağlamaktadır. RmiProxyFactoryBean oluşan tüm RemoteException türündeki hataları Spring'in ihtiva ettiği RemoteAccessException hiyerarşisindeki hatalara dönüştürür.

rmi-client-config.xml dosyasında tanımladığımız rentalRmiProxy Spring nesnesini kullanarak bir proxy nesne oluşturan RmiClient sınıfı kod 13.5.1 de yer almaktadır. Bu örnekte görüldüğü gibi context nesnesi aracılığı ile edindiğimiz rentalService nesnesi Spring tarafından oluşturulmuş proxy nesnesidir. Bu nesne üzerinde yapılan tüm işlemler kod 13.5 de RmiProxyFactoryBean ile tanımlanmış olan ve

```
rmi://127.0.0.1:1099/rentalService
```

adresinde yer alan RmiServiceExporter (kod 13.3) nesnesine yönlendirilir.

```
Kod 13.5.1 - RmiClient

public class RmiClient {

    public static void main(String[] args)
            throws IOException {
        ApplicationContext context =
                new ClassPathXmlApplicationContext(
                    "rmi-client-config.xml");
        RentalService rentalService =
            context.getBean(RentalService.class);
        Rental rental = new Rental();
        Customer customer = new Customer("Zeki", "Alasya", 68);
        Car car = new Car("Ford", "Fiesta");
        rental.setCustomer(customer);
        rental.setCar(car);
        Rental result = rentalService.rentACar(rental);
        logger.debug(result.isRented());
    }
}
```

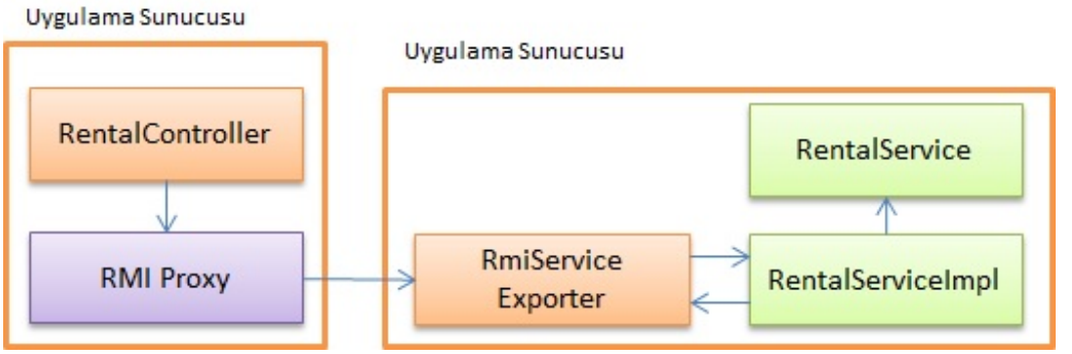
RmiProxyFactoryBean kullanımı RmiClient gibi sınıfların oluşturulmasını şu şekilde kolaylaştırmaktadır:

- Service interface sınıfı (RentalService) kullanıldığı sürece herhangi bir proxy implementasyonu enjekte etmek mümkündür. Bu kullanıcı sınıfın (RmiClient) farkında olmadan uzak bir nesne üzerinde işlem yapmasını sağlar.
- Proxy nesne oluşturma ve RMI bağlantısını yönetme görevini RmiProxyFactoryBean üstlenmektedir. Bu RmiClient kodunun daha sade ve sadece işletme mantığına odaklı olmasını sağlar.
- Kullanılan Spring konfigürasyon tipi değiştirilerek RmiCLient sınıfının lokal bir RentalService implementasyonu kullanması sağlanabilir. Bu sadece konfigürasyon değişikliği gerektirir, kod değişikliği değil.

RmiProxyFactoryBean tarafından oluşturulan proxy nesnesi service interface sınıfını dinamik olarak implemente ettiği için bu proxy nesnesi servis interface sınıfı üzerinden işlem yapan her yere enjekte edilebilir. Kod 13.6 da yer alan konfigürasyon örneğinde gösterim katmanında yer alan RentalController sınıfına kod 13.5 de oluşturduğumuz rentalRmiProxy nesnesi enjekte edilmektedir. rentalController nesnesi farkında bile olmadan rmi proxy nesnesi aracılığı ile bir RMI nesnesi kullanmaktadır. RMI nesnesi ile olan tüm iletişim rentalRmiProxy (kod 13.5) aracılığı ile sağlanmaktadır.

```
kod 13.6 - applicationContext.xml

<bean id="rentalController"
        class="com.kurumsaljava.spring.web.RentalController">
    <property name="rentalService"
        ref="rentalRmiProxy"/>
</bean>
```

Resim 13.2 de görüldüğü gibi rmi proxy nesnesi iki değişik uygulama sunucusu arasında bağlantı kurma ve veri trafiğini yönetme vazifesini üstlenmektedir.

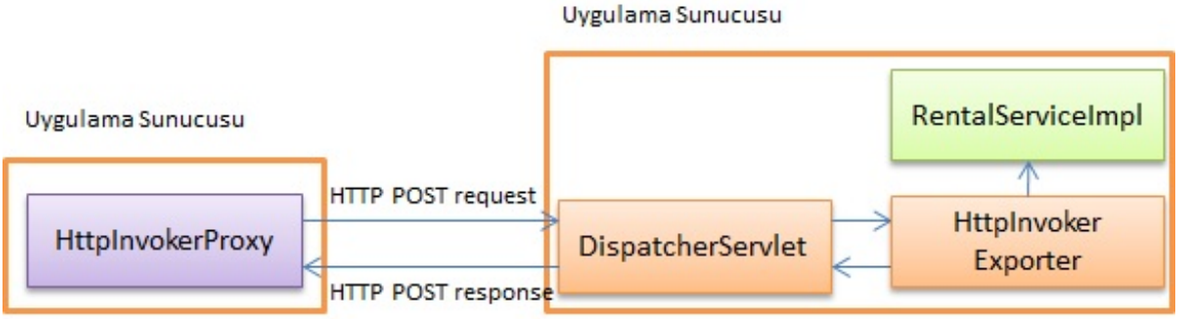


Resim 13.2

HttpInvoker Kullanımı

Spring Remoting bünyesinde yer alan HttpInvoker yardımı ile HTTP protokolünü kullanarak POJO sınıflar sunucu servis haline dönüştürülebilir. Bu tür sunuculara istek göndermek için HTTP POST metodu, verileri transfer etmek için standart Java serialization mekanizması kullanılmaktadır.

HttpInvoker kullanımını gerekli kılan başlıca nedenlerden birisi RMI teknolojisinin firewall sistemleri ile birlikte kullanılmasının sorunlu olmasıdır. Çalıştığım bir projede RMI ile geliştirdiğim sunucuyu AWS (Amazon Web Services) sunucularında çalıştırmak istediğimde bu sorunla karşılaşmış ve bu sorunu HttpInvoker konfigürasyonu aracılığı ile çözmüştüm.



Resim 13.3

Kod 13.7 de yer alan konfigürasyon örneğinde HttpInvokerServiceExporter sınıfı kullanılarak RentalService implementasyonu olan RentalServiceImpl sınıfı sunucu olarak tanımlanmaktadır. ServiceInterface element özelliği servis interface sınıfını tanımlamak için kullanılmaktadır. Kullanıcı bu interface sınıfı aracılığı ile hangi metotları kullanabileceğini bilir, sunucu ise bu interface sınıf bünyesinde tanımlanan metotları bir implementasyon sınıfı aracılığı ile kullanıma sunar.

```
Kod 13.7 - rental-httpinvoker-servlet.xml

<bean name="/RentalService"
        class="org.springframework.remoting.httpinvoker.
            HttpInvokerServiceExporter">
    <property name="service" ref="rentalService"/>
    <property name="serviceInterface"
        value="com.kurumsaljava.spring.service.RentalService"/>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.service.
            RentalServiceImpl">
    <property name="customerRepository"
                ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
    <property name="carRepository" ref="carRepository" />
</bean>
```

Bu tür bir entegrasyon çözümünde HTTP protokolü kullanıldığı için sunucu tarafında uygulamanın bir web uygulaması olarak çalışır hale getirilmesi gerekmektedir. Bu amaçla web.xml bünyesinde kod 13.8 de yer alan servlet konfigürasyonunun yapılması zorunludur.

```
Kod 13.8 - web.xml

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/rental-httpinvoker-servlet.xml
    </param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.
            ContextLoaderListener
    </listener-class>
</listener>

<servlet>
    <servlet-name>rental-httpinvoker</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>2</load-on-startup>
```

```
</servlet>

<servlet-mapping>
    <servlet-name>rental-httpinvoker</servlet-name>
    <url-pattern>/RentalService</url-pattern>
</servlet-mapping>
```

ContextLoaderListener sınıfı contextConfigLocation parametresine atanan Spring konfigürasyon dosyası ile bir WebApplicationContext oluşturmaktadır. Gelen kullanıcı isteklerinin Spring web uygulaması bünyesinde yönlendirilmesi için DispatcherServlet kullanılmaktadır. http://localhost/RentalService adresine gelen tüm istekler DispatcherServlet tarafından HttpInvokerServiceExporter aracılığı ile tanımlanan /RentalService Spring nesnesine, yani servis sınıfına yönlendirilir.

HttpInvoker Client Kullanımı

HttpInvoker ile tanımlanan bir sunucuya kullanıcı olarak HttpInvokerProxyFactoryBean aracılığı ile erişim sağlanabilir. Böyle bir kullanıcı konfigürasyonu kod 13.9 da yer almaktadır.

```
Kod 13.9 - httpinvoker-client-config.xml

<bean id="rentalServiceClient"
    class="org.springframework.remoting.httpinvoker.
            HttpInvokerProxyFactoryBean">
    <property name="serviceUrl"
        value="http://localhost:8080/RentalService" />
    <property name="serviceInterface"
        value="com.kurumsaljava.spring.service.RentalService" />
</bean>
```

HttpInvokerProxyFactoryBean konfigürasyonu için serviceUrl ve serviceInterface isminde iki parametreye ihtiyaç duyulmaktadır. Bu iki parametre ile servisin bulunduğu sunucunun web adresi ve kullanılan servis interface sınıfı tanımlanmaktadır. Kod 13.10 da yer alan HttpInvokerClient sınıfı kod 13.9 da tanımlı olan konfigürasyon dosyası aracılığı ile HTTP sunucusu üzerinde işlem yapmaktadır. Eğer HttpInvokerClient kodunu kod 13.5.1 de yer alan RmiClient kodu ile kıyaslarsanız, aralarında çok fazla bir farklılık olmadığını görebilirsiniz. Tek farklılık, kullanılan konfigürasyon dosyasının ismidir. Spring bu iki örnek ile çok değişik entegrasyon teknolojilerinin kullanımı için tek bir programlama modeline sahip olduğunun

ispatını sunmaktadır.

```
Kod 13.10 - HttpInvokerClient

public class HttpInvokerClient {

    public static void main(String[] args) {

        ApplicationContext context =
                new ClassPathXmlApplicationContext(
            "httpinvoker/httpinvoker" + "-client-config.xml");
        RentalService rentalService =
            context.getBean(RentalService.class);
        Rental rental = new Rental();
        Customer customer = new Customer("Zeki", "Alasya", 68);
        Car car = new Car("Ford", "Fiesta");
        rental.setCustomer(customer);
        rental.setCar(car);
        Rental result = rentalService.rentACar(rental);
        System.out.println(result.isRented());
    }
}
```

HppInvoker ile oluşturduğumuz uygulama bir web uygulamasıdır ve bir WAR arşivi olarak uygulama sunucusunda koşturulur. HttpInvoker Tomcat gibi bir uygulama sunucusu olmadan da kullanılabilir. Bunun için SimpleHttpInvokerServiceExporter, SimpleHttpServerFactoryBean ve Sun Java 6 kullanımı gerekmektedir. Sun Java 6 bünyesinde yer alan HTTP uygulama sunucusu implementasyonu ile uygulama koşturulur. Böyle bir konfigürasyon örneği kod 13.11 de yer almaktadır.

```
Kod 13.11 - standalone-rental-httpinvoker-servlet.xml

<bean id="rentalHttpServer"
    class="org.springframework.remoting.support.
        SimpleHttpServerFactoryBean">
    <property name="contexts">
        <util:map>
            <entry key="/RentalService" value
                    ref="rentalServiceExporter"/>
        </util:map>
    </property>
    <property name="port" value="8080" />
</bean>

<bean name="rentalServiceExporter"
```

```
    class="org.springframework.remoting.
        httpinvoker.SimpleHttpInvokerServiceExporter">
    <property name="service" ref="rentalService"/>
    <property name="serviceInterface"
            value="com.kurumsaljava.spring.service.
                    RentalService"/>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.service.
                RentalServiceImpl">
    <property name="customerRepository"
                ref="customerRepository" />
    <property name="rentalRepository"
            ref="rentalRepository" />
    <property name="carRepository" ref="carRepository" />
</bean>
```

SimpleHttpServerFactoryBean sınıfının contexts isminde bir parametresi bulunmaktadır. Bu parametre yardımı ile oluşturulan web uygulamasının gelen kullanıcı isteklerine hangi adres altında cevap vereceği tanımlanmaktadır. Kod 13.11 de yer alan örnekte kullanıcı istekleri için

```
http://localhost:8080/RentalService
```

adresi tanımlanmaktadır. Bu adrese gelen istekler rentalServiceExporter isimli nesneye yönlendirir. rentalServiceExporter SimpleHttpInvokerServiceExporter kullanılarak tanımlanmış bir Spring nesnesidir. HttpInvokerServiceExporter sınıfında olduğu gibi SimpleHttpInvokerServiceExporter ile HttpInkover nesnesi tanımlanmaktadır.

Kod 13.11 de yer alan HttpInvoker nesnesine erişimi sağlamak için kod 13.10 da yer alan HttpInvokerCLient sınıfı kullanılabilir.

Hessian ve Burlap Kullanımı

Hessian ve Burlap Caucho firması tarafından geliştirilmiş ve HTTP protokolünü kullanan iki entegrasyon teknolojisidir. Hessian RMI teknolojinde olduğu gibi bir binary protokol kullanmaktadır. RMI'ın aksine sunucu ve kullanıcı arasında taşınan binary mesajlar Java harici PHP, Python, C++ ve C# gibi başka dillerde de işlenebilmektedir. Burlap bünyesinde mesajlar XML olarak taşınmaktadır. Burlap'ın bu özelliği kullanıcının değişik dillerde

geliştirilebilmesini mümkün kılmaktadır, çünkü kullanılan dilin sunması gereken tek özellik XML mesajları üzerinde işlem yapabilmektir. Burlap bünyesinde kullanılan XML mesaj yapısı çok basit olduğu için, örneğin Web Service teknolojisinde mesaj formatlarını tanımlamak için kullanılan WSDL (Web Service Definition Language) gibi bir format tanımlama dili kullanma gerekliliği bulunmamaktadır.

Konfügrasyon işlemleri HttpInvoker ve Hessian/Burlap için kullanılan exporter sınıfları haricinde aynı şemayı takip etmektedir. Hessian ile bir servis tanımlamak için HessianServiceExporter sınıfı kullanılmaktadır. Kod 13.12 de bir Hessian exporter örneği yer almaktadır.

```
Kod 13.12 - rental-hessian-servlet.xml

<bean name="/RentalService"
        class="org.springframework.remoting.caucho.
        HessianServiceExporter">
    <property name="service" ref="rentalService"/>
    <property name="serviceInterface"
        value="com.kurumsaljava.spring.service.
                RentalService"/>
</bean>
```

Hessian ile oluşturulan bir sunucuya erişim HessianProxyFactoryBean sınıfı ile sağlanabilir. Böyle bir kullanıcı tanımlaması kod 13.13 de yer almaktadır.

```
Kod 13.13 - hessian-client-config.xml

<bean id="rentalServiceClient"
    class="org.springframework.remoting.caucho.
            HessianProxyFactoryBean">
    <property name="serviceUrl"
        value="http://localhost:8080/RentalService" />
    <property name="serviceInterface"
        value="com.kurumsaljava.spring.service.
            RentalService" />
</bean>
```

Analog olarak Burlap ile bir servis oluşturmak için BurlapServiceExporter ve servis kullanıcısı oluşturmak için BurlapProxyFactoryBean sınıfı kullanılabilir.

Hessian/Burlap HTTP protokolünü kullandığı için Spring HttpInvoker çözümünde olduğu gibi oluşturulan uygulamanın bir web uygulaması olarak koşturulması gerekmektedir. Hessian/Burlap uygulaması için yapılacak

web.xml ayarları HttpInvoker ile aynıdır.

SimpleJaxWsServiceExporter İle Web Servis Kullanımı

Java dünyasında web servis uygulamları geliştirmek için JAX-WS ismini taşıyan standart bir API bulunmaktadır. Bu API, sahip olduğu anotasyonlar ve Spring'in sunduğu SimpleJaxWsServiceExporter aracılığı ile sıradan bir Java sınıfını web servis uygulaması haline getirebiliriz.

Kod 13.15 de yer alan JaxWSRentalServiceEndPointImpl sınıfı kod 13.14 de yer alan JaxWSRentalServiceEndPoint interface sınıfın bir implementasyonudur. JaxWSRentalServiceEndPointImpl web servis sunucu sınıfı temsil etmektedir. JaxWSRentalServiceEndPointImpl bir JAX-WS anotasyonu olan @WebService ile bir web servis sunucu sınıf, @WebMethod ile rentACar() metodu bir web servis metodu haline gelmektedir.

JaxWSRentalServiceEndPointImpl bünyesinde RentalService tipinde olan service değişkenini barındırmaktadır. Spring sunucusu uygulama çalışmaya başladıgı zaman JaxWSRentalServiceEndPointImpl sınıfına kod 13.2 de yer alan RentalServiceImpl sınıfından olma bir nesne enjekte edecektir. Bu şekilde JaxWSRentalServiceEndPointImpl sınıfı kullanıcı isteklerini service değişkeni aracılığı ile servis katmanında yer alan RentalServiceImpl sınıfına delege etmektedir. Bu açıdan bakıldığında JaxWSRentalServiceEndPointImpl sınıfını servis katmanını gölgeleyen bir cephe (facade) olarak görebiliriz.

```
Kod 13.14 - JaxWSRentalServiceEndPoint

@WebService
public interface JaxWSRentalServiceEndPoint {

    public Rental rentACar(Rental rental);

}

Kod 13.15 - JaxWSRentalServiceEndPointImpl

@Component
@WebService(serviceName="RentalWebService")
public class JaxWSRentalServiceEndPointImpl
        implements JaxWSRentalServiceEndPoint {
```

```
    @Autowired
    RentalService service;

    @WebMethod
    public Rental rentACar(Rental rental) {
        return service.rentACar(rental);
    }
}
```

SimpleJaxWsServiceExporter @WebService ile işaretli sınıfları web servis sunucusu haline getirmek için JDK 1.6 bünyesinde yer alan JAX-WS provider altyapısını ve HTTP web sunucusunu kullanmaktadır. Böylece herhangi bir uygulama sunucusu kullanma zorunluluğu olmadan sadece Spring çatısı ve JDK 1.6 ile HTTP SOAP bazlı bir web servis uygulaması geliştirilebilmektedir. Bunun için gerekli Spring konfigürasyonu kod 13.14 de yer almaktadır.

```
Kod 13.16 - jaxws-server-config.xml

<bean class="org.springframework.remoting.
            jaxws.SimpleJaxWsServiceExporter">
    <property name="baseAddress"
            value="http://localhost:8080/"/>
</bean>
```

SimpleJaxWsServiceExporter aktif hale geldiği zaman, classpath içinde yer alan ve @WebService anotasyonunu taşıyan sınıfları http://localhost:8080/ adresi altında web servis sunucusu haline getirir. Web servis sunucusunu aktif hale getirmek için kod 13.16.1 de yer alan JaxWSServerMain sınıfını koşturmak yeterli olacaktır.

```
Kod 13.16.1 - JaxWSServerMain

public class JaxWSServerMain {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("
        jaxws/jaxws-server-config.xml");
    }
}
```

Yapı olarak incelediğimizde SimpleJaxWsServiceExporter sınıfının gerçek bir JAX-WS uygulaması oluşturmadığını görmekteyiz. Daha ziyade SimpleJaxWsServiceExporter JDK 1.6 bünyesindeki JAX-WS provider ve HTTP web sunucusunu kullanarak bir web servis uygulaması oluşturmaktadır.

Oluşturulan web servis uygulamasının yaşam döngüsünü Spring kontrol etmektedir. Kod 13.15 de yer alan JaxWSRentalServiceEndPointImpl bir Spring nesnesidir. Gerçek bir JAX-WS uygulamasında uygulamanın yaşam döngüsünü JAX-WS motoru (engine) ya da implementasyonu kontrol eder.

Mevcut bir uygulama sunucusu bünyesinde SimpleJaxWsServiceExporter sınıfını kullanmak mümkün değildir, çünkü bu exporter sınıfı JDK 1.6 bünyesindeki HTTP web sunucusunu kullanmaktadır.

JAX-WS Client Kullanımı

Kod 13.15 de oluşturduğumuz web servis sunucusuna bağlantı kurabilmek için bir kullanıcı sınıfa ihtiyaç duymaktayız. JaxWsPortProxyFactoryBean sınıfını kullanarak böyle bir kullanıcı sınıfı oluşturabiliriz.

```
Kod 13.17 - jaxws-client-config.xml

<bean id="rentalWebService"
    class="org.springframework.remoting.jaxws.
                        JaxWsPortProxyFactoryBean">
    <property name="serviceInterface"
        value="com.kurumsaljava.spring.jaxws.
                    JaxWSRentalServiceEndPoint" />
    <property name="wsdlDocumentUrl"
        value="http://localhost:8080/RentalWebService?wsdl" />
    <property name="namespaceUri"
        value="http://jaxws.spring.kurumsaljava.com/" />
    <property name="serviceName"
        value="RentalWebService" />
    <property name="portName"
        value="JaxWSRentalServiceEndPointImplPort" />
</bean>
```

JaxWsPortProxyFactoryBean isminden de anlaşıldığı gibi web servis sunucusuna vekillik eden bir proxy nesne oluşturmaktadır. Hangi interface sınıf kullanılarak proxy nesnenin oluşturulacağı serviceInterface değişkeni ile tanımlanmaktadır. Kod 13.17 de yer alan örnekte serviceInterface olarak kod 13.14 de yer alan JaxWSRentalServiceEndPoint tanımlamıştır. Böylece proxy nesne kod 13.15 de yer alan JaxWSRentalServiceEndPointImpl sınıfı ile aynı interface sınıfı implemente ederek, onun yerine geçebilmektedir.

WsdlDocumentUrl değişkeni web servisin WSDL (Web Service Definiton Language) dosyasına işaret etmektedir. Bu Spring tarafından otomatik olarak

oluşturulan ve web servisi ve kullanım şeklini tanımlayan bir yapıdır. NamespaceUri, serviceName ve portName WSDL dosyasında edinebileceğimiz değerlerdir. Şu şekilde WSDL dosyasında yer alırlar:

```
<xsd:schema>
    <xsd:import
        namespace="http://jaxws.spring.kurumsaljava.com/"
        schemaLocation="
          http://localhost:8080/RentalWebService?xsd=1"/>
</xsd:schema>
...

<service name="RentalWebService">
    <port name="JaxWSRentalServiceEndPointImplPort"
        binding="tns:JaxWSRentalServiceEndPointImplPortBinding">
        <soap:address
            location="http://localhost:8080/RentalWebService"/>
    </port>
</service>
```

Kod 13.18 de yer alan JaxWSClient sınıfı kod 13.17 de yer alan Spring konfigürasyonunu kullanmaktadır. Bu örnekte görüldüğü gibi web servis kullanıcı sınıf konumunda olan JaxWSClient sınıfı JaxWsPortProxyFactoryBean tarafından oluşturulan proxy nesneyi kullanmaktadır. Bu proxy nesne web servis sunucusu ile olan tüm iletişimi yönetmektedir.

```
Kod 13.18 - jaxws-client-config.xml

public class JaxWSClient {

    public static void main(String[] args) {

        ApplicationContext context =
                new ClassPathXmlApplicationContext("jaxws/jaxws" +
                        "-client-config.xml");
        JaxWSRentalServiceEndPoint rentalService =
                context.getBean(JaxWSRentalServiceEndPoint.class);
        Rental rental = new Rental();
        Customer customer = new Customer("Zeki", "Alasya", 68);
        Car car = new Car("Ford", "Fiesta");
        rental.setCustomer(customer);
        rental.setCar(car);
        Rental result = rentalService.rentACar(rental);
        System.out.println(result.isRented());
    }
```

```
}
```

Kitabın bir sonraki bölümünde Spring ile web servis kullanımını daha detaylı olarak inceleyecegiz.

Hangi Çözümü Kullanmalıyım?

- Sunucu ve kullanıcı Spring çatısını kullanıyorsa, HttpInvoker çözümü seçilebilir.
- Sunucu ve kullanıcı Java ortamında çalışıyor ve uygulama sunucusu mevcut değilse, RMI çözümü kullanılabilir. Sunucu firewall arkadaşına görev yapacaksa RMI yerine HttpInvoker tercih edilmelidir.
- HTTP kullanılıyor ve kullanıcıların değişik programlama dillerinde programlanması isteniyorsa, Hessian çözümü kullanılabilir.
- Kontrol dışında olan kullanıcılar için en uygun çözüm Web Service ya da JMS çözümü olacaktır.

Bahsettiğimiz çözümlerin çoğu RPC (Remote Procedure Call) yöntemiyle çalışmaktadır. RCP'yi uzak bir sunucuda bulunan bir sınıfın bir metodunu koşturma olarak tercüme edebiliriz. Herhangi bir metodu koşturmak için o metodun isminin, parametrelerinin ve geriye verdiği değerin bilinmesi gerekmektedir. Bu yüzden RPC kullanıldığında da bu bilgilerin sunucu ve kullanıcı arasında taşınması gerekmektedir. Bu sunucu ile kullanıcıyı bir anlamda birbirine bağımlı kılmaktadır.

Eğer seçilen çözüm bünyesinde Java serialization mekanizması kullanılıyorsa, kullanılan servis sınıflarının kullanıcıda da mevcut olması gerekmektedir. Bunun yanı sıra kullanılan sınıf versiyonlarının da uyuşması mecburidir.

13. Bölüm Soruları

13.1 Entegrasyon teknolojilerini kullanırken programcıların karşılaştığı sorunların başında ne gelmektedir?

13.2 Spring Remoting modülünün sağladığı en belirgin avantaj nedir?

13.3 Spring Remoting HTTP protokolünü kullanan hangi entegrasyon yapısına sahiptir?

13.4 Sunucu ve kullanıcı Spring çatısını kullanıyorsa, hangi entegrasyon yöntemi kullanılmalıdır?

14. Bölüm

Spring Web Service

Yazılımın savaş verdiği en büyük cephelerden birisi değişen müşteri gereksinimlere ayak uydurmaktır. Bu savaşı kazanmada kullanılabilecek en güçlü silah Extreme Programming gibi çevik bir sürecin kullanılmasıdır. Bunun yanı sıra modüler bir yapının oluşturulması ve modüller arası esnek bağların oluşturulması değişen müşteri gereksinimlerine adaptasyonu kolaylaştıracaktır.

Esnek bağların oluşturulması özellikle kolay değişme eğilimi gösteren müşteri gereksinimlerinin koda dönüştürülmesinde önemli rol oynamaktadır. Esnek bağlar uygulamanın değişikliklere karşı dayanıklılığını artırmaktadır. Ana maksat yapılan değişikliklerin modüller arası uyumsuzluğa neden olmasını engellemektir. Bu tarz uyumsuzluklara değişik teknolojiler kullanılarak oluşturulan modüllerin birbirlerini kullanmalarında daha sıkça rastlanmaktadır. Özellikle böyle durumlarda esnek bağın saydığım avantajlarıyla birlikte kullanılmasını mümkün kılacak bir teknoloji ile oluşturulması önem taşımaktadır. Böyle bir teknoloji XML (Extensible Markup Language) dir.

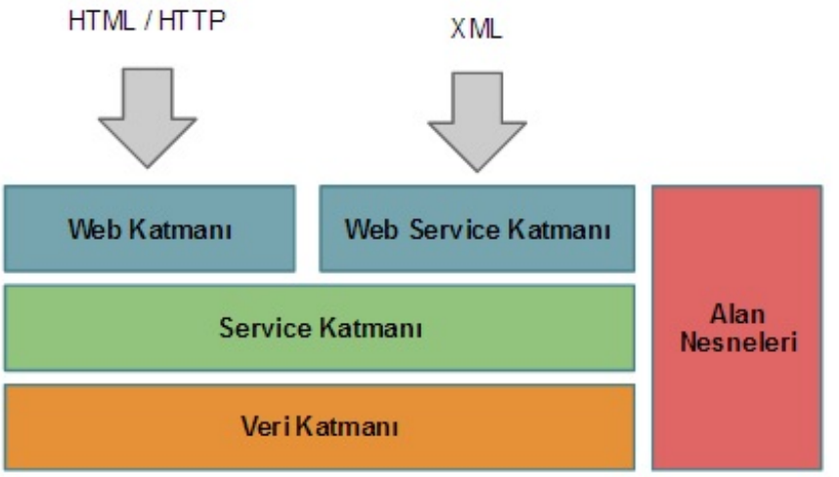
XML (Extensible Markup Language) günümüzde değişik sistemler arasında veri taşımak için kullanılan bir teknolojidir. Java dünyasında Dom, Stax ve Sax, .Net dünyasında System.XML, Ruby dünyasında REXML gibi implementasyonları mevcuttur. XML'i değişik türdeki sistemlerin birlikte çalışmalarını mümkün kılan ortak bir dil olarak düşünebiliriz.

Sistem entegrasyonunda XML gibi ortak bir dilin tanımlanması ve kullanımı temel altyapıyı oluşturmaktadır. Bunun yanı sıra sistemler arası entegrasyon yöntemlerini tanımlayan, veri alışverişini standartlaştıran ve hata yönetimi gibi konulara cevap veren bir entegrasyon teknolojisine ihtiyaç duyulmaktadır. Böyle bir teknoloji web servis (Web Service) ismini taşımaktadır. Bu bölümde Spring ile web servis teknolojisinin kullanımını örnekler üzerinde yakından inceleyeceğiz.

Web Servis Uygulama Mimarisi

Araç kiralama işlemlerini gerçekleştirmek için oluşturduğumuz uygulama bu hali ile bir web arayüzü üzerinden işlem yapılmasını mümkün kılmaktadır. Müşterimiz mevcut uygulamanın herhangi bir sistemden araç kiralama talebi alacak şekilde genişletilmesi isteğine sahiptir. Bunu gerçekleştirmek için kendisine web servis teknolojisinin kullanımını tavsiye ediyoruz. Resim 14.1 de görüldüğü gibi uygulamanın web katmanına paralel olarak web servis

katmanını oluşturacağız. Web servis katmanı mevcut servis katmanını kullanarak gerekli işlemlerin web servis kanalı üzerinde gerçekleştirilmesini sağlayacak.



Resim 14.1

Resimde görüldüğü gibi web katmanı HTTP protokolü ile işleyip, veri alışverişi için HTML kullanırken, web servis katmanı XML ile veri alışverişini yönetmektedir. Web servis için herhangi bir veri taşıyıcı protokol kullanılabilir. Bu çoğu zaman HTTP'dir, lakin HTTP kullanımı mecburi değildir. Örneğin XML formatındaki verileri taşımak için JMS (Java Messaging Service) ya da SMTP (Simple Message Transfer Protocol) kullanılabilir. Web servis teknolojisinde XML formatının kullanımı da mecburi değildir. Örneğin XML yerine CSV (Comma Seperated Value) kullanılabilir. Web servis teknolojilerinde çoğunlukla veri taşımak için HTTP, verinin formatını tanımlamak için XML ve verinin transfer ediliş şeklini, iletişim noktalarının tanımlanması ve kullanılmasını belirlemek için SOAP (Simple Object Access Protocol) protokolü kullanılır. Bu bölümde yer alacak web servis örneklerinde HTTP/SOAP/XML kombinasyonunu kullanacağız.

Ortak Dilin Tanımlanması

Web servis aracılığı ile entegre edilmek istenen sistemlerin aynı dili konuşabilmeleri için ortak bir kelime hazinesinin tanımlanması gerekmektedir. Bu ortak dil XSD (XML Schema Definition) kullanılarak oluşturulur. XSD dosyalarını Java sınıfları gibi düşünebiliriz. Birer şablon olan XSD dosyalarından XML nesneleri, yani transfer edilecek veriyi tanımlayan mesajlar oluşturulur. Kod 14.1 de yabancı bir sistemin araç kiralama işlemi yapmak için web servis arayüzü üzerinden uygulamamıza gönderebileceği XML formatında

bir mesaj yer almaktadır.

```
Kod 14.1 - rentacar-request.xml

<?xml version="1.0" encoding="UTF-8"?>
<rentalRequest xmlns="http://www.rentacar.com/request">
    <car id="1"/>
    <customer>
        <name>Akin</name>
        <firstname>Filiz</firstname>
        <age>60</age>
    </customer>
    <date>2014-11-11</date>
</rentalRequest>
```

Bu mesajdan yola çıkarak, Trang ismini taşıyan şema konverter yardımı ile gerekli XSD dosyasını oluşturacağız. Trang'in kullanımı kod 14.2 de yer almaktadır.

```
Kod 14.2 - Trang Kullanimi

java -jar trang.jar rentacar-request.xml rentacar-schema.xsd
```

rentacar-request.xml dosyasını parametre olarak alan Trang rentacar-schema.xsd ismi altında gerekli XSD dosyasını oluşturmaktadır. Oluşturulan bu dosyanın içeriği kod 14.3 de yer almaktadır.

```
Kod 14.3 - rentacar-request-schema.xsd

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
        targetNamespace="http://www.rentacar.com/request"
      xmlns:request="http://www.rentacar.com/request">
  <xs:element name="rentalRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="request:car"/>
        <xs:element ref="request:customer"/>
        <xs:element ref="request:date"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="car">
    <xs:complexType>
      <xs:attribute name="id" use="required" type="xs:integer"/>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="customer">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="request:name"/>
        <xs:element ref="request:firstname"/>
        <xs:element ref="request:age"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:NCName"/>
  <xs:element name="firstname" type="xs:NCName"/>
  <xs:element name="age" type="xs:integer"/>
  <xs:element name="date" type="xs:NMTOKEN"/>
</xs:schema>
```

Mevcut bir sistem mesajından yola çıkarak bir XSD dosyası oluşturma işlemini daha kolay bulduğum için Trang ile bu işlemi gerçekleştirdim. XML Spy gibi araçlar kullanara ta böyle XSD dosyaları oluşturmak mümkün.

Veri Tipi Sözleşmesi - Message Contract

Web servis genel olarak değişik teknolojiler kullanılarak oluşturulan kullanıcı ve sunucular arasında entegrasyonu sağlayan bir entegrasyon teknolojisidir. Kullanıcı ve sunucunun veri alışverişini sağlamak için belli noktalarda anlaşmaları, yani bir nevi sözleşme yapmaları gerekmektedir. Kullanılan veri tiplerini belirleyen bu sözleşmeye web servis dilinde **Message Contract** ismi verilmektedir. Bir önceki bölümde oluşturduğumuz rentacar-schema.xsd böyle bir mesaj sözleşmesidir.

rentacar-schema.xsd kullanıcı tarafından sunucuya gönderilen mesajlarda yer alan verilerin veri tiplerini tanımlamaktadır. Aynı şekilde sunucudan kullanıcıya gönderilen mesajların da bir XSD dosyasında tanımlanması gerekmektedir. Böyle bir XSD dosyasını oluşturmak amacıyla kod 14.4 deki rentalResponse ismini taşıyan mesajı tanımlıyoruz.

```
Kod 14.4 - rentacar-response.xml

<?xml version="1.0" encoding="UTF-8"?>
<rentalResponse xmlns="http://www.rentacar.com/response">*
    <id>1</id>
</rentalResponse>
```

Trang ve rentacar-response.xml dosyasını kullanarak kod 14.5 de yer alan ve sunucunun kullanıcıya gönderdiği mesajları tanımlayan rentacar-response-schema.xsd isimli dosyası oluşturuyoruz.

```
Kod 14.5 - rentacar-response-schema.xsd

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
        targetNamespace="http://www.rentacar.com/response"
    xmlns:response="http://www.rentacar.com/response">
  <xs:element name="rentalResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="response:id"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="id" type="xs:integer"/>
</xs:schema>
```

Geldigimiz noktayı tekrar gözden geçirelim:

- rentalRequest elementi ile başlayan ve kullanıcının talebini (request) sunucuya ileten bir XML yapısı tanımladık.
- rentacar-request-schema.xsd dosyasında rentalRequest XML elementinin yapısını tanımladık.
- rentalResponse elementi ile başlayan ve sunucunun işlem sonucunu kullanıcıya aktardığı (response) bir XML yapısı tanımladık.
- rentacar-response-schema.xsd dosyasında rentalResponse XML elementinin yapısını tanımladık.

rentacar-request-schema.xsd ve rentacar-response-schema.xsd dosyaları kullanıcı ve sunucu arasında kullanılacak ortak dili tanımlayan XSD dosyalarıdır. Bu her iki XSD dosyasını rental.xsd kod 14.6 da görüldüğü şekilde bir araya getiriyoruz.

```
Kod 14.6 - rental.xsd

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.rentacar.com/rental"
        xmlns:r="http://www.rentacar.com/rental"
    elementFormDefault="qualified">
```

```
    <xs:element name="rentalRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="r:car" />
                <xs:element ref="r:customer" />
                <xs:element ref="r:date" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="car">
        <xs:complexType>
            <xs:attribute name="id" use="required"
                type="xs:integer" />
        </xs:complexType>
    </xs:element>

    <xs:element name="customer">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="r:name" />
                <xs:element ref="r:firstname" />
                <xs:element ref="r:age" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="name" type="xs:NCName" />
    <xs:element name="firstname" type="xs:NCName" />
    <xs:element name="age" type="xs:integer" />
    <xs:element name="date" type="xs:NMTOKEN" />

    <xs:element name="rentalResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="r:id" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="id" type="xs:integer" />
</xs:schema>
```

JAXB ile XML/Java Dönüşümü

Kullanıcı tarafından gönderilen bir XML mesajın içinde yer alan verilere erişmek için ya XML araçları kullanabiliriz (Dom ya da Sax parseri ya da

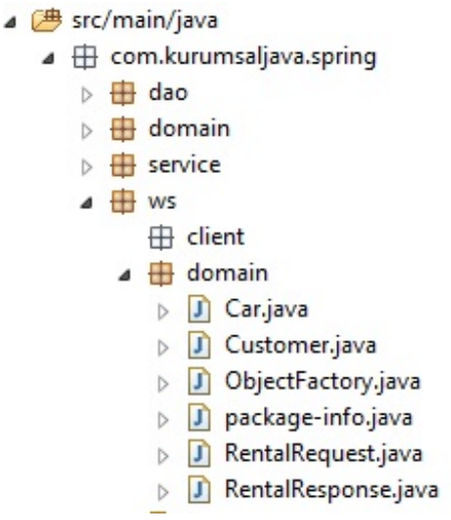
XPath gibi araçlar) ya da verileri birebir Java nesnelerine dönüştürebiliriz. Java dünyasında işlem yaptığımız için XML verileri Java nesnelerine dönüştürmek en mantıklı seçenek olacaktır.

Java dünyasında bu işlemi gerçekleştirmek için JAXB (Java Architecture for XML Binding), Castor ya da XMLBeans gibi çatılar mevcuttur. Bu bölümdeki örneklerde JAXB 2 (2. sürüm) çatısını kullanacağız. JAXB Java EE 5 ve JDK 6 nın bir parçasıdır ve XML/Java arası dönüşüm için anotasyonlardan faydalanmaktadır. JAXB ile XSD şemalardan sınıflar ya da sınıflardan XSD şemalar oluşturabilmektedir. Spring OXM modülü bünyesinde JAXB desteği sağlamaktadır.

Bu amaçla JAXB 2 çatısı yardımı ile öncelikle rental.xsd dosyasında tanımladığımız veri tiplerine es gelen Java sınıflarını oluşturmamız gerekmektedir. Kod 14.7 de yer alan Ant dosyası JDK'in bir parçası olan xjc komutu ile gerekli Java sınıflarını oluşturmaktadır.

```
Kod 14.7 - create-classes.xml

<?xml version="1.0"?>
<project name="rentacar" default="generate-sources">

    <property name="src.dir" value="src/main/java"/>

    <target name="generate-sources">
        <exec executable="xjc">
            <arg line=" -d ${src.dir}
                -p com.kurumsaljava.spring.ws.domain
                src/main/resources/schema/rental.xsd"/>
        </exec>
    </target>
</project>
```

Ant dosyasında yer alan generate-sources hedefini koşturduğumuz taktirde resim 14.2 de yer alan Java sınıfları oluşur.



Resim 14.2

Resim 14.2 de görüldüğü gibi daha önce rental-request.xml ve rental-response.xml dosyalarında kullandığımız her XML elementi için bir Java sınıfı oluşturuldu.

JAXB ile XML/Java arasında dönüşüm otomatik olarak yapılmaktadır. Spring WS JAXB'yi kullanarak kullanıcılar tarafından gönderilen XML verilerin oluşturulan bu Java nesnelerine dönüştürülmesini sağlamaktadır.

Spring WS Konfigürasyonu

Oluşturacağımız web servis katmanı XML dosyalarını kullanıcı ile sunucu arasında taşımak için HTTP protokolünü kullanmaktadır. Bu bir web uygulaması geliştirdiğimiz anlamına gelmektedir. Web servis katmanı belli bir HTTP adresi altında hizmet vermektedir. Bu adres ve kullanılacak olan Spring WS konfigürasyonu kod 14.8 da yer almaktadır.

```
Kod 14.8 - web.xml

<servlet>
    <servlet-name>rentals</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.
            MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/ws-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>rentals</servlet-name>
    <url-pattern>/rentals/*</url-pattern>
</servlet-mapping>
```

MessageDispatcherServlet XML olarak gelen web servis mesajlarını uygulamaya yönlendirmek için kullanılan Spring sınıfıdır. ContextConfigLocation parametresi ile kullanılacak Spring WS konfigürasyon dosyası tanımlanmaktadır. ws-config.xml dosyası kod 14.9 da yer almaktadır.

```
Kod 14.9 - ws-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:ws="http://www.springframework.org/schema/web-services"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/
                    spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/
                    spring-context-3.0.xsd
            http://www.springframework.org/schema/web-services
            http://www.springframework.org/schema/web-services/
                web-services-2.0.xsd">

    <context:component-scan base-package="com.kurumsaljava" />

    <ws:annotation-driven />

    <ws:dynamic-wsdl id="rentalDefinition" portTypeName="Rentals"
        locationUri="http://localhost:8080/rentals">
        <ws:xsd location="classpath:/schema/rental.xsd" />
    </ws:dynamic-wsdl>

</beans>
```

Spring MVC controller sınıflarında olduğu gibi Spring WS uygulaması bünyesinde de konfigürasyon işlemleri için anotasyonlardan faydalanacağız. Bu sınıfların Spring tarafından bulunabilmesi ve Spring nesnesi olarak oluşturulabilmesi için component-scan elementini kullanmaktayız.

Ws isim alanında yer alan annotation-driven elementi Spring WS uygulamasının anotasyon kullanılarak konfigüre edildiğine işaret etmektedir. Bunun yanı sıra bu element yardımı ile XML/Java dönüşümü için gerekli

JAXB2 marshaller, unmarshaller sınıfları konfigüre edilmektedir. Bunun haricinde JAXB2 kullanımı için başka bir konfigürasyon yapma zorunluluğu bulunmamaktadır.

Kod 14.9 da yer alan konfigürasyon ile web servis uygulamamız çalışır hale getirebiliriz.

Endpoint Tanımlaması

Kullanıcının belli bir HTTP adresine gönderdiği isteğin (request) uygulamamız tarafından web servis istediği olarak karşılanabilmesi ve işlenebilmesi için Endpoint ismini taşıyan bir Java sınıfı tanımlamamız gerekmektedir. Kod 14.8 de tanımladığımız MessageDispatcherServlet web servis isteklerini bu Endpoint sınıfına yönlendirmektedir. Bir WS Endpoint olan RentalEndpoint kod 14.10 da yer almaktadır.

```
Kod 14.10 - RentalEndpoint

package com.kurumsaljava.spring.ws;

import java.math.BigInteger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.
                Endpoint;
import org.springframework.ws.server.endpoint.annotation.
                PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.
                RequestPayload;
import org.springframework.ws.server.endpoint.annotation.
                ResponsePayload;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;
import com.kurumsaljava.spring.service.RentalService;
import com.kurumsaljava.spring.ws.domain.RentalRequest;
import com.kurumsaljava.spring.ws.domain.RentalResponse;

@Endpoint
public class RentalEndpoint {

    private static final String NAMESPACE_URI =
                "http://www.rentacar.com/rental";

    private final RentalService service;
```

```
    @Autowired
    public RentalEndpoint(final RentalService service) {
        this.service = service;
    }

    @PayloadRoot(namespace = NAMESPACE_URI,
        localPart = "rentalRequest")
    @ResponsePayload
    public RentalResponse rentACar(
            @RequestPayload final RentalRequest req) {

        final Rental rental = new Rental();
        final Car car = new Car();
        car.setId(req.getCar().getId().longValue());
        rental.setCar(car);
        final Customer customer = new Customer();
        customer.setAge(req.getCustomer().getAge().intValue());
        customer.setFirstname(req.getCustomer().getFirstname());
        customer.setName(req.getCustomer().getName());
        rental.setCustomer(customer);

        this.service.rentACar(rental);

        final RentalResponse response = new RentalResponse();
        response.setId(BigInteger.valueOf(
            System.currentTimeMillis()));
        return response;
    }
}
```

Kod 14.10 da yer alan RentalEndpoint sınıfı @Endpoint anotasyonunu taşımaktadır. Bu anotasyon RentalEndpoint sınıfının Spring WS tarafından bir WS Endpoint olarak görülmesini sağlamaktadır. Kod 14.9 da kullandığımız component-scan elementi Spring'in tüm Java sınıflarını tarayarak RentalEndpoint gibi belli anotasyonları taşıyan sınıfların görevlerini yerine getirecek şekilde konfigüre edilmelerini sağlamaktadır.

NAMESPACE_URI değişmeyeni web servis uygulamamız için tanımladığımız ortak kelime hazinesinin isim alanını tanımlayan ibareyi ihtiva etmektedir. Bunu bir Java paket ismi olarak düşünebiliriz. Kullanıcı ve sunucu arasında değiş-tokuş edilen XML mesajlarında kullanılan elementler bu isim alanında tanımladığımız elementlerdir.

Service RentalEndpoint sınıfına sınıf konstrüktörü üzerinden enjekte edilen servis katmanı nesnesidir. Görüldüğü gibi web servis katmanına ait olan

RentalEndpoint sınıfı servis katmanına erişmek için servis nesnesini kullanmaktadır. Daha önce incelediğimiz Spring MVC controller örneklerinde de servis katmanına erişmek için bu nesneyi kullanmıştık. Bu demek oluyor ki gösterim ve web servis katmanı uygulamamız için değişik erişim kanalları şeklinde kullanılmaktadır.

@PayloadRoot ile XML mesajı işlenmek üzere kabul edecek metot tanımlanmaktadır. Kod 14.10 da yer alan örnekte rentalRequest elementi ile başlayan XML mesajlarından rentACar() metodu sorumludur.

@RequestPayload anotasyonu ile kullanıcının gönderdiği XML JAXB yardımı ile RentalRequest tipinde bir nesneye dönüştürülmektedir. Böylece req ismindeki metot parametresi kullanıcının gönderdiği XML'de yer alan tüm verileri ihtiva etmektedir.

@ResponsePayload anotasyonu ile rentACar() metodunda oluşturulan RentalResponse nesnesi XML'e dönüştürülerek kullanıcıya gönderilmektedir.

WSDL ve Contract First

WSDL (Web Services Definition Language) web servis katmanında kullanılabilecek fonksiyon, veri tipi ve veri alışveriş protokol formatlarını tanımlamak için kullanılan bir meta dildir. WSDL dosyasını kullanıcı ve sunucu arasında bir sözleşme metni olarak düşünebiliriz. Tanımladığımız XSD dosyası (rental.xsd) veri tipi sözleşmesi (Message Contract) iken, WSDL sunulan servisin genel hatlarını tanımlayan sözleşmedir. Bu yüzden WSDL dosyasına servis sözleşmesi ya da Service Contract ismi verilmektedir. Sadece WSDL aracılığı ile bir web servis kullanıcısı sunucuyu nasıl kullanabileceğini bilebilir. Bu yüzden web servis katmanlarında sunulan servisi tanımlayan WSDL dosyalarının tanımlanması zorunludur.

Klasik web servis uygulamalarında servis tanımlama işlemine WSDL dosyası oluşturularak başlanır. Bu yönteme contract first, yani sözleşmenin önceden tanımlanması ismi verilmektedir. Bizim oluşturduğumuz web servis uygulaması da contract first yöntemini kullanmaktadır, çünkü uygulamayı geliştirmeden önce XSD dosyalarını oluşturduk ve bu dosyalardan yola çıkarak kullanıcı ve sunucu arasında veri alışverişi için gerekli veri tiplerini tanımladık. Web servis uygulamaları geliştirirken contract first yönetimi en iyi yöntem (best practice) olarak kabul edilmektedir. Mevcut Java sınıfların yola çıkarak ta WSDL

dosyaları oluşturmak mümkündür, lakin bu servisin tanımlanış şeklini implementasyon detaylarına bağladığı için, servis tanımlaması üzerinde değişiklik yapılmasını zor kılmaktadır. Spring WS contract first yönetimi destekleyen bir web servis çatıdır.

Peki WSDL dosyasını nasıl tanımlamamız gerekiyor? Bir WSDL dosyası tanımlamamız gerekmiyor, çünkü Spring bu işi bizim için kod 14.9 da yer alan dynamic-wsdl elementi ile otomatik olarak yapmaktadır. Spring rental.xsd dosyasından yola çıkarak dinamik olarak WSDL dosyasını oluşturmaktadır. Oluşturulan WSDL dosyasını görmek için uygulamanın şu adresine gitmeniz yetmektedir:

```
http://localhost:8080/rentals/rentalDefinition.wsdl
```

Spring tarafından dinamik olarak oluşturulan WSDL dosyası kod 14.11 de görülmektedir.

```
Kod 14.11 - rentalDefinition.wsdl

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:sch="http://www.rentacar.com/rental"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.rentacar.com/rental"
    targetNamespace="http://www.rentacar.com/rental">
    <wsdl:types>
        <xs:schema xmlns:r="http://www.rentacar.com/rental"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            targetNamespace="http://www.rentacar.com/rental">

            <xs:element name="rentalRequest">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="r:car" />
                        <xs:element ref="r:customer" />
                        <xs:element ref="r:date" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>

            <xs:element name="car">
                <xs:complexType>
                    <xs:attribute name="id" type="xs:integer"
                use="required" />
```

```
                </xs:complexType>
            </xs:element>

            <xs:element name="customer">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="r:name" />
                        <xs:element ref="r:firstname" />
                        <xs:element ref="r:age" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="name" type="xs:NCName" />
            <xs:element name="firstname" type="xs:NCName" />
            <xs:element name="age" type="xs:integer" />
            <xs:element name="date" type="xs:NMTOKEN" />

            <xs:element name="rentalResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="r:id" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="id" type="xs:integer" />

        </xs:schema>
    </wsdl:types>
    <wsdl:message name="rentalRequest">
        <wsdl:part element="tns:rentalRequest"
        name="rentalRequest">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="rentalResponse">
        <wsdl:part element="tns:rentalResponse"
        name="rentalResponse">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="Rentals">
        <wsdl:operation name="rental">
            <wsdl:input message="tns:rentalRequest"
        name="rentalRequest">
            </wsdl:input>
            <wsdl:output message="tns:rentalResponse"
        name="rentalResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="RentalsSoap11" type="tns:Rentals">
```

```
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="rental">
            <soap:operation soapAction="" />
            <wsdl:input name="rentalRequest">
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="rentalResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="RentalsService">
        <wsdl:port binding="tns:RentalsSoap11"
        name="RentalsSoap11">
            <soap:address
            location="http://localhost:8080/rentals" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Kod 14.11 de yer alan WSDL dosyasını herhangi bir kullanıcı vererek, uygulamamıza web servis katmanı üzerinden erişmesini sağlayabiliriz. Bu WSDL dosyasını alan kullanıcı ilk etapta eğer Java platformunu kullanıyorsa JAXB yardımı ile gerekli sınıfları oluşturur. Kullanıcı eğer Java platformunu kullanmayorsa, kullandığı platformun sunduğu araçlar yardımı ile sunucunun anlayacağı dile XML mesajları oluşturarak, sunucu ile iletişime geçebilir. Görüldüğü gibi kullanıcı ile sunucunun aynı dili konuşabilmeleri için WSDL dosyası çok önemli bir araçtır.

Web Servis Kullanımı

Web servis sunucusuna erişmek için bir kullanıcı sınıf oluşturmamız gerekmektedir. WsClientTest ismini taşıyan böyle bir sınıf kod 14.11.1 de yer almaktadır.

```
Kod 14.11.1 - WsClientTest

package com.kurumsaljava.spring.ws.client;

import static java.math.BigInteger.valueOf;
import static org.junit.Assert.assertNotNull;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
            SpringJUnit4ClassRunner;
import org.springframework.ws.client.core.
            WebServiceTemplate;
import com.kurumsaljava.spring.ws.domain.Car;
import com.kurumsaljava.spring.ws.domain.Customer;
import com.kurumsaljava.spring.ws.domain.RentalRequest;
import com.kurumsaljava.spring.ws.domain.RentalResponse;

@ContextConfiguration("classpath:client-config.xml")
@RunWith(SpringJUnit4ClassRunner.class)
public class WsClientTest {

    @Autowired
    WebServiceTemplate webServiceTemplate;

    @Test
    public void testServiceWithJaxb() {

        final RentalRequest request = new RentalRequest();
        final Customer customer = new Customer();
        customer.setFirstname("Tarik");
        customer.setName("Akan");
        customer.setAge(valueOf(68));
        request.setCustomer(customer);

        final Car car = new Car();
        car.setId(valueOf(1));
        request.setCar(car);

        request.setDate("2014-11-11");
        final RentalResponse response =
            (RentalResponse) this.webServiceTemplate.
            marshalSendAndReceive(request);
        assertNotNull(response);
    }
}
```

WsClientTest aslında bir JUnit test sınıfı. Kitabın 15. bölümünde Spring ile test konseptlerini daha yakından inceleyeceğiz.

Kod 14.11.1 görüldüğü gibi testServiceWithJaxb() test metodu bünyesinde daha önce JAXB ile oluşturduğumuz Java sınıfları yardımı ile bir RentalRequest nesnesi oluşturulmaktadır. Bu Java nesnesi webServiceTemplate nesnesi aracılığı ile XML formatına dönüştürülmekte ve web servis sunucusuna

gönderilmektedir. WebServiceTemplate web servis sunucuları ile interaksiyona girmek için kullanılabilecek bir Spring sınıfıdır.

@ContextConfiguration anotasyonu ile kullanılacak Spring konfigürasyonu tayin edilmektedir. Kullanılan bu konfigürasyon dosyasının içeriği kod 14.12 de yer almaktadır.

```
Kod 14.12 - client-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oxm="http://www.springframework.org/schema/oxm"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
      http://www.springframework.org/schema/beans/
                spring-beans.xsd
      http://www.springframework.org/schema/oxm
      http://www.springframework.org/schema/oxm/
                spring-oxm-3.0.xsd">

    <oxm:jaxb2-marshaller id="marshaller"
        contextPath="com.kurumsaljava.spring.ws.domain" />

    <bean
        class="org.springframework.ws.client.core.
                WebServiceTemplate">
        <property name="marshaller" ref="marshaller" />
        <property name="unmarshaller" ref="marshaller" />
        <property name="defaultUri"
        value="http://localhost:8080/rentals" />
    </bean>
</beans>
```

WebServiceTemplate sınıfının Java nesnelerini XML formatına dönüştürebilmesi için kullanacağı dönüşüm aracını bilmesi gerekmektedir. Kod 14.12 de yer alan örnekte oxm isim alanında yer alan jaxb2-marshaller elementi ile JAXB çatısı konfigüre edilmektedir. WebServiceTemplate tanımlanan marshaller nesnesini kullanarak XML/Java dönüşümünü gerçekleştirmektedir. Bu dönüşüm her iki yöne doğru yapılmaktadır. WebServiceTemplate RentalRequest nesnesini XML'e dönüştürmekte ve oluşan XML mesajı sunucuya göndermekte ve sunucudan gelen XML dosyasını bir RentalResponse nesnesine dönüştürmektedir.

SOAP Mesaj Yapısı

Kullanıcı ve sunucu arasındaki XML trafiğini SOAP protokolü şekillendirmektedir. Kod 14.13 de WsClientTest sınıfı aracılığı ile web servis sunucusuna gönderdiğimiz bir XML mesajı yer almaktadır. Görüldüğü gibi mesaj SOAP protokolü kullanılarak formatlanmıştır.

```
Kod 14.13 - sample-soap-request-xml

<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
                soap/envelope/">
    <SOAP-ENV:Header />
    <SOAP-ENV:Body>
        <ns2:rentalRequest
        xmlns:ns2="http://www.rentacar.com/rental">
            <ns2:car id="1" />
            <ns2:customer>
                <ns2:name>Akan</ns2:name>
                <ns2:firstname>Tarik</ns2:firstname>
                <ns2:age>68</ns2:age>
            </ns2:customer>
            <ns2:date>2014-11-11</ns2:date>
        </ns2:rentalRequest>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Kod 14.13 de yer alan SOAP mesajı üç bölümden oluşmaktadır. Bunlar Envelope, Header ve Body bölümleridir. Tüm mesaj Envelope elementi ile bir zarfa konmaktadır. Bu zarf içinde Header ve Body bölümleri yer almaktadır. Body zarfa konulan mektubun kendisi iken, Header alıcının adresi gibi verileri ihtiva eden bölümdür. Body bölümünde daha önce rental.xsd dosyasında tanımladığımız rentalRequest elementini görmekteyiz. rentalRequest ile sunucuya gönderilen verinin veri tipi tanımlanmaktadır. Bu veri tipinin tanımlandığı isim alanı http://www.rentacar.com/rental dır.

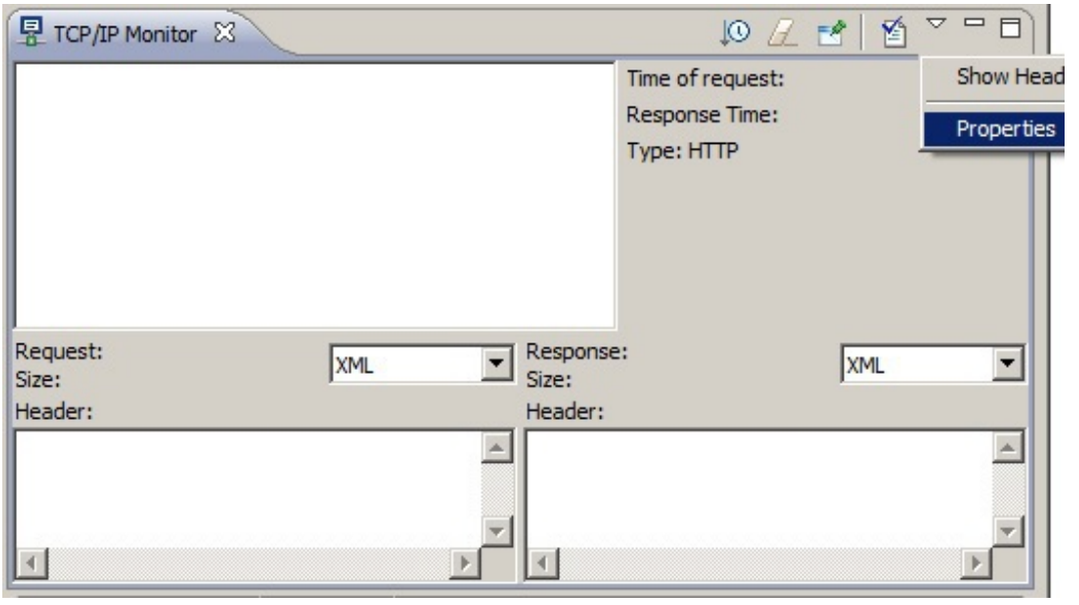
```
Kod 14.14 - sample-soap-response.xml

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.
            org/soap/envelope/">
    <SOAP-ENV:Header />
    <SOAP-ENV:Body>
        <ns2:rentalResponse xmlns:ns2="http://www.rentacar.com/
                    rental">
```

```
            <ns2:id>1382096899129</ns2:id>
        </ns2:rentalResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

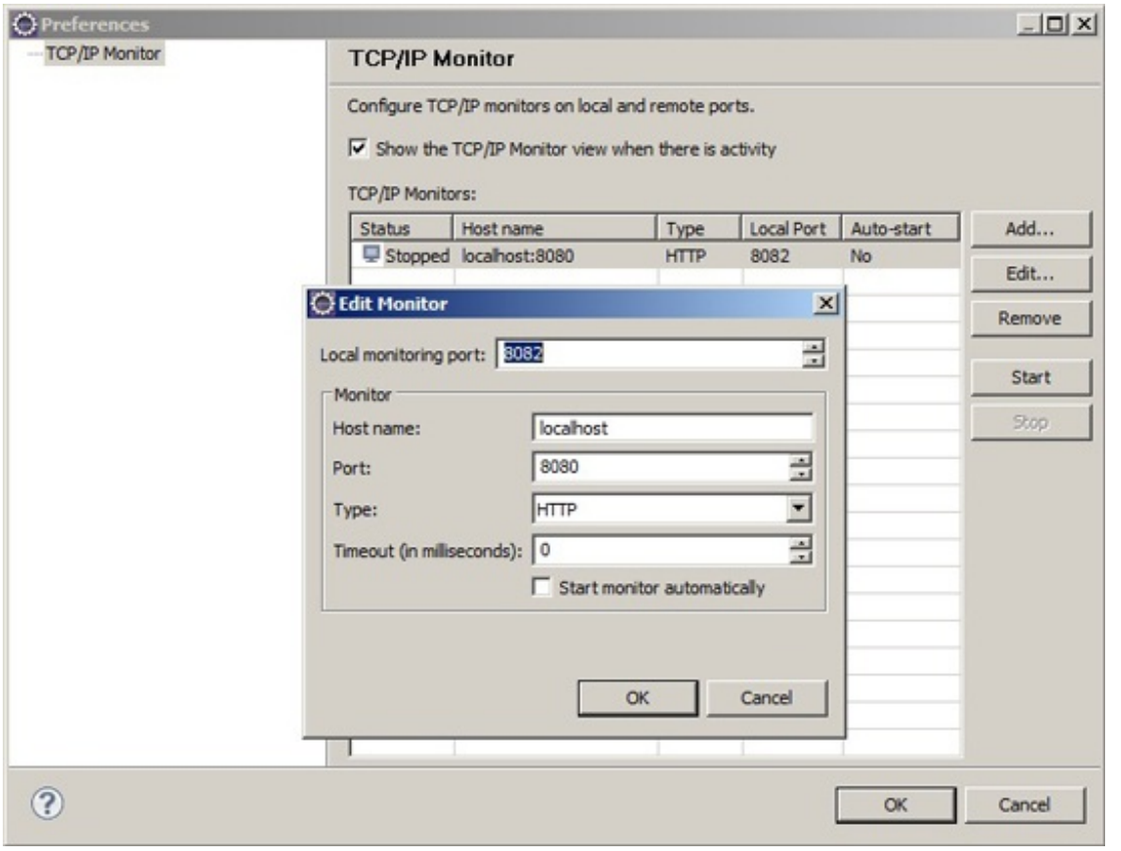
Kod 14.14 de sunucunun kullanıcıya gönderdiği cevap yer almaktadır. Bu mesajda SOAP formatındadır ve daha önce bahsettiğim bölümlerden oluşmaktadır. Body bölümünde rentalResponse elementi ile kullanıcı için önem taşıyan veri tanımlanmaktadır.

TCP Monitor Kullanımı

Eğer Eclipse kullanıcısıysanız TCP/IP Monitor ismini taşıyan plugin yardımıyla kullanıcı ve sunucu arasındaki trafiği takip edebilirsiniz. Bu aracın konfigürasyonu resim 14.3 ve 14.4 de görülmektedir.

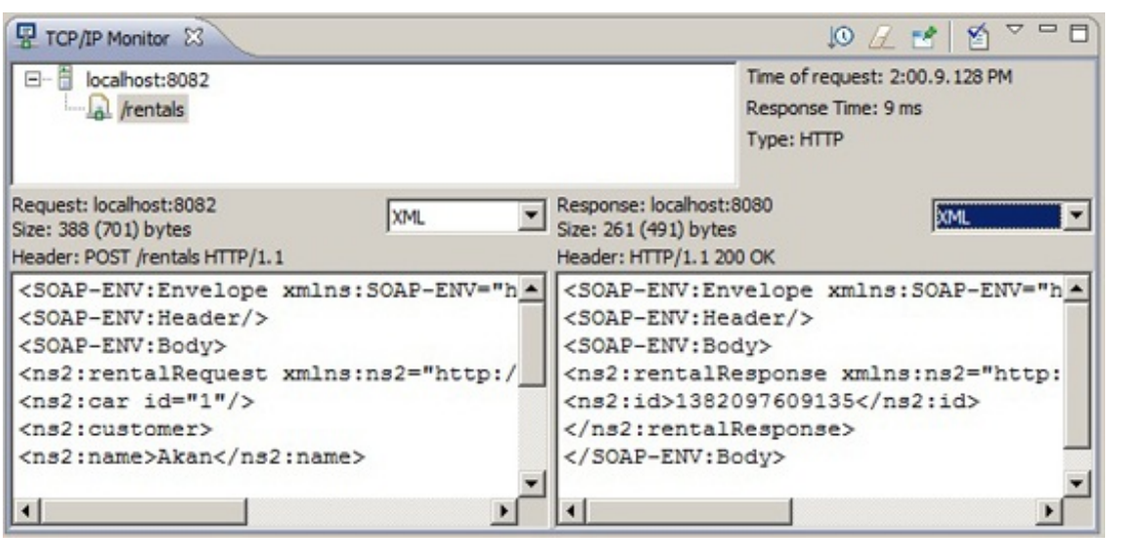


Resim 14.3



Resim 14.4

Web servis sunucusu 8080 numaralı port üzerinde çalışmaktadır. TCP/IP Monitor kullanıcı ile sunucu arasına girerek, gönderilen mesajların kopyasını almaktadır. Bu amaçla TCP/IP Monitor'un 8082 gibi bir port üzerinde çalıştırılması ve kullanıcının 8080 yerine bu portu kullanması gerekmektedir. Kullanıcının 8082 numaralı portu kullanabilmesi için gerekli değişikliği kod 14.12 de yer alan client-config.xml dosyasında yapabiliriz. Sunucu ve kullanıcıyı koşturduktan sonra gönderilen XML dosyalarını kod 14.14 deki gibi TCP/IP Monitor bünyesinde görebiliriz.



Resim 14.4

POX - Plain Old XML

SOAP yerine gönderilen mesajların basit XML (POX) formatında olmasını istiyorsak, kod 14.15 de yer alan konfigürasyon işimizi görecektir.

```
Kod 14.15 - ws-pox-config.xml

<bean id="messageFactory"
        class="org.springframework.ws.pox.dom.
                DomPoxMessageFactory"/>

<bean id="messageDispatcher"
        class="org.springframework.ws.server.
            MessageDispatcher"/>
```

Spring SOAP XML mesajlarını oluşturmak için standart olarak SaajSoapMessageFactory sınıfını kullanılır. POX oluşturmak için bu sınıfı DomPoxMessageFactory olarak değiştirmemiz gerekmektedir.

Sunucuya ulaşan SOAP mesajlar SoapMessageDispatcher sınıfı aracılığı ile Endpoint sınıfına iletilir. Gelen POX mesajları EndPoint sınıfına aktarmak için MessageDispatcher sınıfını tanımlamamız gerekmektedir. Bu değişikliğin ardından TCP/IP Monitor'e bakıldığında kullanıcı ve sunucunun POX formatında veri alışverişi yaptığını görmek mümkün olacaktır.

İnterseptör Kullanımı

İnterseptörler (Interceptor) verinin işlenme sürecinde belli noktalarda kontrolü devralarak, veriyi değerlendirmek için kullanılan bir tekniktir. Örneğin interseptörler kullanılarak loglama, veri kontrolü (validation) ve güvenlik kontrolleri gibi işlemler yapılabilir. Spring WS çatısında kullanıcı ve sunucu arasındaki veri trafiğinde kullanılan XML dosyaları üzerinde işlem yapmak üzere interseptörler mevcuttur. EndpointInterceptor interface sınıfını implemente bu interseptörler:

Loglama interseptörleri

- SoapEnvelopLoggingInterceptor - Request ve response mesajlarını loglamak için kullanılan interseptör sınıfıdır.

- PayloadLoggingInterceptor - Mesajlarda payload olarak isimlendirilen ve transfer edilen verileri loglamak için kullanılan interseptör sınıfıdır.

Mesaj validasyonu interseptörü

- PayloadValidationInterceptor - Request ve response mesajlarının kullanılan XSD şemasıyla uyumluluğunu kontrol eden interseptör sınıfıdır. Geçerli olmayan mesajları için Soap hatası (fault) oluşturur.

Mesaj transformasyon interseptörü

- PayloadTransformingInterceptor - Request ya da response mesajlarının XSLT kullanarak yapısal değiştirilmesini mümkün kılan interseptör sınıfıdır.

Kod 14.15 de PayloadLoggingInterceptor ve PayloadValidatingInterceptor interseptörlerinin konfigürasyonu yer almaktadır. ws isim alanında yer alan interceptors elementi ile interseptörler uygulamada aktif hale getirilmektedirler.

```
Kod 14.15 - ws-interceptor-configuratin.xml

<ws:interceptors>
    <bean class="org.springframework.ws.server.endpoint.
             interceptor.PayloadLoggingInterceptor"/>
    <bean class="org.springframework.ws.soap.server.endpoint.
            interceptor.PayloadValidatingInterceptor">
        <property name="schema"
            value="classpath:schema/rental.xsd"/>
    </bean>
</ws:interceptors>
```

EndpointInterceptor sınıfını implemente ederek yeni interseptör sınıfları oluşturmak mümkündür.

Adı geçen bu interseptörler uygulamanın sunucu tarafında kullanılan interseptörlerdir. Kllanıcı tarafında da interseptörlerin kullanımı mümkündür. Kod 14.16 da PayloadValidatingInterceptor kullanımı yer almaktadır. Sunucu interseptörlerin aksine kullanıcı interseptörleri org.springframework.ws.client paketinde yer almaktadır.

```
Kod 14.16 - client-config.xml

<bean
        class="org.springframework.ws.client.core.
```

```
            WebServiceTemplate">
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
    <property name="defaultUri"
            value="http://localhost:8080/rentals" />
    <property name="interceptors">
        <bean class="org.springframework.ws.client.
              support.interceptor.
            PayloadValidatingInterceptor">
            <property name="schema" value="rental.xsd"/>
        </bean>
    </property>
</bean>
```

Hata Yönetimi

Web servis katmanında oluşan hatalar SOAP Fault olarak kullanıcıya aktarılır. Spring WS çatısında oluşan hataları kullanıcıya bildirmek için EndpointExceptionResolver implementasyonları kullanılır. Herhangi bir konfigürasyon yapılmadığı taktirde bir hata oluşması durumunda SimpleSoapExceptionResolve Server tipinde bir SOAP Fault oluşturur. Hata mesajına (Fault String) oluşan hatanın içeriği atanır. Bir SOAP Fault örneği kod 14.17 de görülmektedir.

```
Kod 14.17 - SOAP Fault

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
            schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header />
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Server</faultcode>
            <faultstring xml:lang="en">bu bir hata
        mesajidir</faultstring>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SoapFaultMappingExceptionResolver kullanılarak kullanıcı için daha anlamlı hata mesajları oluşturulabilir. Kod 14.18 de SoapFaultMappingExceptionResolver konfigürasyonu yer almaktadır. ExceptionMappings bölümünde oluşan uygulama bünyesinde oluşan EmptyResultDataAccessException tipindeki bir hata için SOAP Fault türü

(Server ya da Client) ve hata mesajı tanımlanmaktadır. ExceptionMappings bölümünde birden fazla SOAP Fault tanımlaması yapılabilmektedir.

```
Kod 14.18 - ws-config.xml

<bean
    class="org.springframework.ws.soap.server.endpoint.
                     SoapFaultMappingExceptionResolver">
    <property name="exceptionMappings">
        <value>
            org.springframework.dao.
             EmptyResultDataAccessException=
                    CLIENT,Entity not found
        </value>
    </property>
    <property name="defaultFault" value="SERVER" />
</bean>
```

Anotasyon bazlı hata yönetimi kullanmak için SoapFaultAnnotationExceptionResolver sınıfı geliştirilmiştir. java.lang.Exception türünden olan herhangi bir sınıf @SoapFault anotasyonunu taşıyorsa, SOAP Fault hata mesajı oluşturma işlemi bu sınıf tarafından gerçekleştirilir. Kod 14.19 da yer alan BusinessException sınıfı @SoapFault anotasyonunu taşıdığı için uygulama bünyesinde BusinessException tipinde bir hata oluşması durumunda Spring WS tarafından gerekli SOAP Fault'un oluşturulması için kullanılır.

```
Kod 14.19 - BusinessException

package com.kurumsaljava.spring.ws;

import org.springframework.ws.soap.server.
        endpoint.annotation.FaultCode;
import org.springframework.ws.soap.server.
            endpoint.annotation.SoapFault;

@SoapFault(faultCode = FaultCode.SERVER)
public class BusinessException extends Exception {

    public BusinessException(String message) {
        super(message);
    }
}
```

Spring'in anotasyon bazlı hata yönetimi yapabilmesi için

SoapFaultAnnotationExceptionResolver sınıfının Spring konfigürasyon dosyasında şu şekilde tanımlanması gerekmektedir.

```
<bean
    class="org.springframework.ws.soap.server.
         endpoint.SoapFaultAnnotationExceptionResolver">
</bean>
```

Web Servis Katmanının Test Edilmesi

Kod 14.11.1 de yer alan sınıf web servis uygulamasını test etmek için kullandığımız birim testidir. Bu testin çalışabilmesi için uygulama sunucusunun çalışıyor halde olması gerekmektedir. Bu tür testlere entegrasyon testleri ismi verilmektedir. Testleri koşturabilmek için uygulama sunucusunun ayakta olması gerekliliği, bu testlerin daha kırılgan olmalarına sebep olmaktadır. Bunun yanı sıra entegrasyon testleri tüm sistemin ayakta oluşunu gerektirdiği için koşturma zamanları (execution time) düz birim testlerine nazaran daha uzundur. Uygulama sunucusunun ayakta olma zorunluluğunu ortadan kaldırmak ve test güdümlü yazılım (TDD - Test Driven Development) yapılmasını sağlamak amacıyla Spring ekibi Spring WS çatısına MockServiceClient ve MockWebServiceServer sınıflarını eklemişlerdir.

MockServiceClient

MockServiceClient sınıfı fake (gerçek olmayan) bir web servis kullanıcısı (client) oluşturmak ve mevcut Spring WS konfigürasyonunu test etmek amacıyla kullanılmaktadır. Kod 14.20 de MockClientTest test sınıfı bünyesinde @ContextConfiguration anotasyonu ile uygulamayı oluşturan Spring konfigürasyonu yüklenmektedir. Bu uygulama sunucusuna gerek kalmadan tüm Spring uygulamasının çalışır hale gelmesi anlamına gelmektedir. Bu hali ile Spring uygulamasına dış dünyadan erişmek mümkün olmada da, uygulamayı içeriden MockWebServiceClient ile test etmek mümkündür. validRequest() test metodu uygulamaya geçerli bir mesaj gönderildiğinde beklenen cevabı test ederken, notValidRequest() kasıtlı olarak hatalı hazırlanan bir mesaja uygulamanın SOAP Fault ile cevap verisini test etmektedir.

```
Kod 14.20 - MockClientTest

package com.kurumsaljava.spring.ws.client;

import static org.springframework.ws.test.server.RequestCreators.
```

```
            withPayload;
import static org.springframework.ws.test.server.ResponseMatchers.
            serverOrReceiverFault;
import static org.springframework.ws.test.server.ResponseMatchers.
            payload;
import javax.xml.transform.Source;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
            SpringJUnit4ClassRunner;
import org.springframework.ws.test.server.MockWebServiceClient;
import org.springframework.xml.transform.StringSource;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "file:WebContent/WEB-INF/ws-config.xml",
        "file:WebContent/WEB-INF/db-config.xml" })

public class MockClientTest {
    @Autowired
    ApplicationContext applicationContext;
    MockWebServiceClient mockClient;

    @Before
    public void createClient() {
        mockClient = MockWebServiceClient.createClient(
                applicationContext);
    }

    @Test
    public void validRequest() {
        Source requestPayload = new StringSource(
                "<rentalRequest xmlns=\"http://www.rentacar.com/
                        rental\">"
                + "<car id=\"1\"/><customer><name>Akin</name>"
                + "<firstname>Filiz</firstname>"
                + "<age>60</age></customer>"
                + "<date>2014-11-11</date></rentalRequest>");

        Source responsePayload = new StringSource(
                "<rentalResponse xmlns=\"http://www.rentacar.
                com/rental\">"
                + "<id>0</id></rentalResponse>");
        mockClient.sendRequest(withPayload(requestPayload)).
                    andExpect(
                payload(responsePayload));
```

```
    }

    @Test
    public void notValidRequest() {
        Source requestPayload = new StringSource(
                "<rentalRequest xmlns=\"
            http://www.rentacar.com/rental\">"
                + "<car id=\"0\"/><customer>"
                + "<name>Akin</name>"
                + "<firstname>Filiz</firstname>"
                + "<age>60</age>"
                + "</customer>  <date>2014-11-11</date>
                            </rentalRequest>");

        mockClient.sendRequest(withPayload(requestPayload)).
            andExpect(serverOrReceiverFault(
                    "car id not found"));
    }
}
```

Kod 14.20 de yer alan MockClientTest test sınıfı her ne kadar çalışan bir uygulama sunucusuna ihtiyaç duymasa da, yapısı itibariyle sunucuyu (server side) test eden bir entegrasyon testidir, çünkü @ContextConfiguration anotasyonu tüm Spring uygulamasını çalışır hale getirmektedir. Buna db-config.xml ile veri tabanı bağlantısının konfigürasyonu da dahildir.

Testi çalıştırmadan önce RentalEndpoint sınıfında yer alan rentACar() metodunun herhangi bir satırına bir break point yerleştirerek, web servis uygulamasını debug etmek mümkündür. Bu ayrıca uygulama sunucusuna gerek kalmadan, web servis uygulamasının çalışır halde olduğunun bir kanıtıdır.

MockWebServiceServer

Kod 14.11.1 de WsClientTest ismini taşıyan ve sunucuya bağlantı kurmak için WebServiceTemplate sınıfını kullanan bir test sınıfı yer almaktadır. Bu sınıf bir JUnit birim testi olarak inşa edilmiş olsa da, WebServiceTemplate sınıfını kullandığı için tipik bir web servis kullanıcısıdır (client). WebServiceTemplate için gerekli konfigürasyon kod 14.12 de (client-config.xml) yer almaktadır.

WebServiceTemplate sınıfını kullanarak bir web servis kullanıcısı (client) haline gelen bir sınıfı herhangi bir web servis sunucusu olmadan test etmek için MockWebServiceServer sınıfından faydalanabiliriz. MockWebServiceServer bizim için fake bir web servis sunucusu oluşturarak, kullanıcı sınıfını test etmemizi mümkün kılacaktır. Kod 14.21 de MockWebServiceServer yardımı ile

bir fake sunucu oluşturan MockServerTest test sınıfı yer almaktadır.

```
Kod 14.21 - MockServerTest.java

package com.kurumsaljava.spring.ws.client;

import static org.junit.Assert.assertNotNull;
import static org.springframework.ws.test.client.RequestMatchers.
            payload;
import static org.springframework.ws.test.client.ResponseCreators.
            withPayload;
import javax.xml.transform.Source;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
            SpringJUnit4ClassRunner;
import org.springframework.ws.test.client.MockWebServiceServer;
import org.springframework.xml.transform.StringSource;
import com.kurumsaljava.spring.ws.domain.RentalResponse;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:mock-server-config.xml")
public class MockServerTest {

    @Autowired
    ServiceClient client;

    private MockWebServiceServer mockServer;

    @Before
    public void createServer() throws Exception {
        mockServer = MockWebServiceServer.
                createServer(client.webServiceTemplate);
    }

    @Test
    public void rentalClient() throws Exception {

        Source requestPayload = new StringSource(
                "<rentalRequest xmlns=\"
                http://www.rentacar.com/rental\">"
                + "<car id=\"1\"/><customer><name>Akin</name>"
                + "<firstname>Filiz</firstname>"
                + "<age>60</age></customer>"
                + "<date>2014-11-11</date></rentalRequest>");
```

```
        Source responsePayload = new StringSource(
                "<rentalResponse xmlns=\"
            http://www.rentacar.com/rental\">"
                + "<id>0</id></rentalResponse>");

        mockServer.expect(payload(requestPayload)).andRespond(
                withPayload(responsePayload));

        RentalResponse response = client.sendRequest();
        assertNotNull(response);
        mockServer.verify();
    }
}
```

Burada MockServerTest sınıfının neden bir JUnit birim testi olarak yapılandırıldığı sorusu aklınıza gelebilir. Bu yöntemi seçtim, çünkü @ContextConfiguration anotasyonunu kullanarak, mevcut bir Spring konfigürasyonunda yer alan nesneleri MockServerTest sınıfına enjekte etmek mümkündür. Kullandığım konfigürasyon dosyası kod 14.22 de yer almaktadır.

```
Kod 14.22 - mock-server-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oxm="http://www.springframework.org/schema/oxm"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            spring-beans-3.0.xsd
        http://www.springframework.org/schema/oxm
        http://www.springframework.org/schema/oxm/
                    spring-oxm-3.0.xsd">

    <oxm:jaxb2-marshaller id="marshaller"
        contextPath="com.kurumsaljava.spring.ws.domain" />

    <bean id="client"
            class="com.kurumsaljava.spring.ws.client.
            ServiceClient">
        <property name="webServiceTemplate"
            ref="webServiceTemplate" />
    </bean>

    <bean id="webServiceTemplate"
        class="org.springframework.ws.client.core.
```

```
                                WebServiceTemplate">
        <property name="defaultUri"
        value="http://www.rentacar.com/rental" />
        <property name="marshaller" ref="marshaller" />
        <property name="unmarshaller" ref="marshaller" />
    </bean>

</beans>
```

Kod 14.22 yer alan konfigürasyon dosyasında kullanıcı isteklerini (request) sunucuya iletmek için kullanılan webServiceTemplate nesnesi yer almaktadır. Böyle bir konfigürasyonu 14.12 de yer alan client-config.xml konfigürasyon dosyasında da kullanmıştık. Orada kullandığımız defaultUri değişkeni web servis sunucusunun adresini ihtiva etmekteydi (http://localhost:8080/rentals). MockWebServiceServer ile fake bir web servis sunucusu oluşturduğumuz için 8080 numaralı port üzerinde çalışan bir uygulama sunucumuz bulunmamaktadır. Bu sebeple defaultUri değişkenine http://www.rentacar.com/rental şeklinde web servis uygulamamızın isim alanını oluşturan değeri atamamız gerekmektedir. Kullanıcı ve fake sunucu arasında kullanılan mesajlar bu isim alanına göre yapılandırılmış olacaktır/olmalıdır.

Test etmek istediğimiz kullanıcı kodu kod 14.23 de yer almaktadır. Kod 14.21 de yer alan MockServerTest sınıfına tekrar baktığımızda

```
MockWebServiceServer.createServer(client.webServiceTemplate)
```

ile fake sunucunun oluşturulduğunu görmekteyiz. Fake sunucuyu oluşturmak için ServiceClient sınıfında yer alan ve web servis kullanıcı mesajlarını sunucuya iletmek için kullanılan webServiceTemplate nesnesi gerekmektedir. MockServerTest.rentalClient() metodu bünyesinde kullanıcıdan geleceği tahmin edilen istek (request) ve kullanıcının beklediği cevap (response) oluşturulmakta ve fake sunucu bu mesajlarla (mockServer.expect) aktif hale getirilmektedir. Burada herhangi bir port üzerinde herhangi bir sunucu çalışır hale getirilmemektedir. Fake sunucuya sadece web servis trafiğinde kullanılacak istek-cevap mesajları tanıtılmaktadır. ServiceClient sınıfında yer alan sendRequest() metodu kosturuldugunda, bu metot fake sunucunun beklediği türde bir istek olusturacak, bu isteği webServiceTemplate üzerinden alan fake sunucu sendRequest() metodunun beklediği cevabı webServiceTemplate üzerinden kullanıcıya iletecektir. Buradaki maksat kod 14.23 de yer alan kullanıcının doğru çalışıp, çalışmadığını test etmektir. Bunun için çalışır

durumda olan bir uygulama sunucusu da kullanılabilir, lakin uygulamayı daha hızlı geliştirebilmek ve test edebilmek için fake bir uygulama sunucusu oluşturan MockWebServiceServer sınıfını kullanmak daha mantıklıdır.

```
Kod 14.23 - ServiceClient.java

package com.kurumsaljava.spring.ws.client;

import static java.math.BigInteger.valueOf;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.ws.client.core.WebServiceTemplate;

import com.kurumsaljava.spring.ws.domain.Car;
import com.kurumsaljava.spring.ws.domain.Customer;
import com.kurumsaljava.spring.ws.domain.RentalRequest;
import com.kurumsaljava.spring.ws.domain.RentalResponse;

@Component
public class ServiceClient  {

    @Autowired
    WebServiceTemplate webServiceTemplate;

    public RentalResponse sendRequest() {

        final RentalRequest request = new RentalRequest();
        final Customer customer = new Customer();
        customer.setFirstname("Filiz");
        customer.setName("Akin");
        customer.setAge(valueOf(60));
        request.setCustomer(customer);

        final Car car = new Car();
        car.setId(valueOf(1));
        request.setCar(car);

        request.setDate("2014-11-11");
        return (RentalResponse) this.webServiceTemplate.
                marshalSendAndReceive(request);
    }

    public void setWebServiceTemplate(WebServiceTemplate
        webServiceTemplate) {
        this.webServiceTemplate = webServiceTemplate;
    }
}
```

Bir sonraki bölümde bu konuları daha detaylı inceleyeceğiz için, bu bölümde yer alan test kodlarını detaya girmeden, tanıtmaya çalıştım.

14. Bölüm Soruları

14.1 Web servis terminolojsinde kullanıcı ile sunucu arasındaki sözleşmenin ismi nedir?

14.2 Web servis uygulamaları geliştirirken Spring hangi yöntemi tercih etmektedir?

14.3 Bir SOAP mesajı hangi bölümlerden oluşmaktadır?

14.4 Request ve response mesajlarını loglamak için kullanılan interseptör hangisidir?

14.5 Web servis kullanıcıları oluşturmak için kullanılan Spring sınıfı hangisidir?

14.6 POX nedir?

14.7 Bir web servis uygulamasını test etmek için uygulama sunucusu gerekli midir?

15. Bölüm

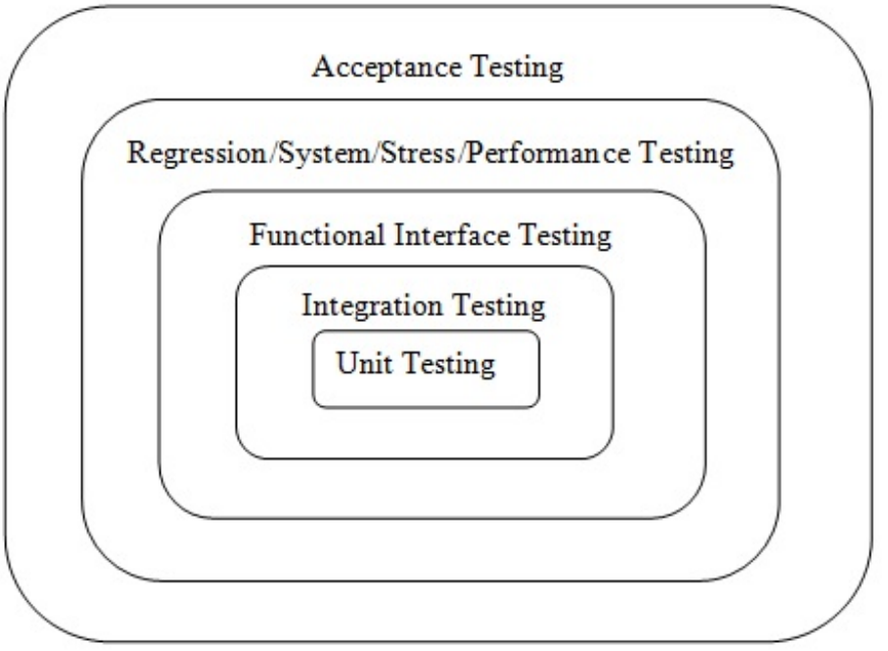
Spring Testing

Yazılım sürecinde oluşturulan sistemin kalite kontrolü değişik türde testlerle yapılır. Bu bölümde Spring ile bu testlerin nasıl hazırlandığını yakından inceleyeceğiz.

En temel test türü birim (unit) testleridir. Bu tür testleri oluşturmak için JUnit test çatısı kullanılır. Spring uygulamalarını test etmek için JUnit çatısından faydalanacağız.

Değişik türde uygulama testleri oluşturmak mümkündür. Bunlar:

- Birim testleri (Unit Testing)
- Entegrasyon testleri (Integration testing)
- Arayüz (interface) testleri (Functional Interface Testing)
- Regresyon testleri (Regression Testing)
- Onay/Kabul testleri (Acceptance Testing)
- Sistem testleri (System Testing)
- Stres testleri (Stress Testing)
- Performans testleri (Performance Testing)



Resim 15.1

Birim testleri ile kendi içinde bütün olan bir kod birimi test edilir. Bu kod biriminin diğer kod parçalarından izole bir şekilde test edilmesi anlamına gelmektedir. Test edilen kod birimleri sınıflarda yer alan metotlardır. Sınıf üzerinde yapılan her değişiklik ardından o sınıf için hazırlanan birim testleri çalıştırılarak, yapılan değişikliğin yan etkileri olup, olmadığı kontrol edilir.

Testlerin olumlu netice vermesi durumda, sınıfın görevini hatasız yerine getirdiği ispat edilmiş olur. Bu açıdan bakıldığında birim testleri programcılar için kodun kalitesini korumak ve daha ileri götürebilmek için vazgeçilmezdir. Uygulama bünyesinde yapılan her değişiklik yan etkilere sebep olabileceği için sistemin her zaman çalışır durumda olduğunu sadece testler ile kontrol edebiliriz. Onlarca ya da yüzlerce sınıfın olduğu bir uygulamayı, her değişikliğin ardından elden ve gelişi güzel test etmek imkansız olduğu için otomatik çalışabilen testlere ihtiyacımız vardır. Daha sonra yakından inceleyeceğimiz JUnit çatısı ile otomatik test sürümü gerçekleştirilir.

Birçok modülden oluşan bir sistemde, modüller arası entegrasyonu test etmek için **entegrasyon testleri** oluşturulur. Entegrasyon testleri hakkında detaya girmeden önce, birim testleri hakkında bir açıklama daha yapma gereği duyuyorum. Birim testleri, test edilen sınıfları kullandıkları diğer sınıflardan bağımsız olarak test ederler. Örneğin bir sınıf veri tabanından veri edinmek için bir servis sınıfını kullanıyorsa, birim testinde bu servis sınıfı kullanılmaz, çünkü bu veri tabanının çalışır durumda olmasını gerektirir. Eğer birim testi içinde veri tabanı kullanılıyorsa, bu birim testi değil, entegrasyon testidir, çünkü test edilen sınıf ile veri tabanı arasındaki ilişki dolaylı olarak test edilmektedir. Daha öncede belirttiğim gibi birim testleri metot bazında ve sınıfın dış dünyaya olan bağımlılıklarından bağımsız olarak gerçekleştirilir. Amacımız bir metot içinde bulunan kod satırlarının işlevini test etmektir. Bunun için veri tabanına bağlantı oluşturulması gerekmez. Test edilen sınıfın bağımlılıklarını ortadan kaldırabilmek için Mock nesneler kullanılır. Bir Mock nesne ile servis sınıfı işlevini yerine getiriyormuşcasına simüle edilir. Birim testinde, test edilen sınıf servis sınıfı yerine onun yerine geçmiş olan Mock nesnesini kullanılır. Entegrasyon testlerinde Mock nesneleri kullanılmaz. Entegrasyon testlerindeki ana amaç sistemin değişik bölümlerinin (sub system) entegre edilerek işlevlerinin kontrol edilmesidir. Entegrasyon testlerinde test edilen sınıflar için gerekli tüm altyapı (veri tabanı, e-posta sunucusu vs.) çalışır duruma getirilir ve entegrasyon test edilir.

Uygulamayı oluşturan modüller bir veya birden fazla sınıftan oluşabilir. Modül kullanımını kolaylaştırmak için interface sınıflar tanımlanır. Modülü kullanmak isteyen diğer modüller bu interface sınıfına karşı programlanır. Modüller arası interaksiyon **arayüz (interface) testleri** ile test edilir. Bu testlere **fonksiyon testleri** adı da verilir.

Regresyon bir adım geri atmak anlamına gelmektir. Regresyon yazılım esnasında, programın yapılan değişiklikler sonucu çalışır durumdan, çalışmaz

bir duruma geçtiği anlamına gelir. **Regresyon testleri** ile sistemde yapılan değişikliklerin bozulmaları neden olup, olmadığı kontrol edilir. Sistem üzerinde yapılan her değişiklik istenmeyen yan etkiler doğurabilir. Her değişikliğin ardından regresyon testleri yapılarak, sistemin bütünlüğü test edilir. Regresyon testlerinde sistem için kullanılan altyapı tanımlanmış bir duruma getirildikten sonra testler uygulanır. Örneğin test öncesi veri tabanı silinerek, test için gerekli veriler tekrar yüklenir. Her test başlangıcında aynı veriler kullanılarak, sistemin nasıl reaksiyon gösterdiği test edilir. Regresyon testlerinin uygulanabilmesi için test öncesinde tüm altyapının başlangıç noktası olarak tanımlanan bir duruma getirilmesi gerekmektedir.

Onay/Kabul testleri ile uygulamanın bütünü kullanıcı gözüyle test edilir. Bu tür testlerde uygulama kara kutu olarak düşünülür. Bu yüzden onay/kabul testlerinin diğer bir ismi kara kutu testleridir (black box test). Kullanıcının uygulamanın içinde ne olup bittiğine dair bir bilgisi yoktur. Onun uygulamadan belirli beklentileri vardır. Bu amaçla uygulama ile interaksiyona girer. Onay/Kabul testlerinde sistemden beklenen geri dönüm test edilir.

Stres testleri tüm sistemin davranışını olağanüstü şartlar altında test eden testlerdir. Örneğin bir web tabanlı uygulamanın eşli zamanlı yüz kullanıcı ile gösterdiği davranış, bu rakam iki yüze çıktığında aynı olmayabilir. Stres testleri ile sistemin belirli kaynaklar ile (donanım, işletim sistemi) stres altındaki davranışı test edilmiş olur.

Performans testleri ile uygulamanın, tanımlanmış kaynaklar (donanım, işletim sistemi) yardımıyla beklenen performansı ölçülür. Test öncesi sistemden beklenen davranış biçimi tayin edilir. Test sonrası beklentiler test sonuçlarıyla kıyaslanır ve kullanılan kaynakların beklenen davranış için yeterli olup olmadığı incelenir. Uygulamanın davranış biçiminin kullanılan kaynakların yetersiz olduğunu göstermesi durumunda, ya uygulama üzerinde değişikliğe gidilir ve uygulamanın mevcut kaynaklar ile yetinmesi sağlanır ya da kaynaklar genişletilerek uygulamanın istenilen davranışı edinmesi sağlanır.

Çevik süreçlerde birim testleri büyük önem taşımaktadır. Extreme Programming bir adım daha ileri giderek, test güdümlü yazılım (Test Driven Development – TDD) konseptini geliştirmiştir. Test güdümlü yazılımda baş rolü birim testleri oynar. Sınıflar oluşturulmadan önce test sınıfları oluşturulur. Bu belki ilk bakışta çok tuhaf bir yaklaşım gibi görünebilir. Var olmayan bir sınıf için nasıl birim testleri yazılabilir diye bir soru akla gelebilir. TDD uygulandığı taktirde, oluşturulan testler sadece sistemde olması gereken fonksiyonların

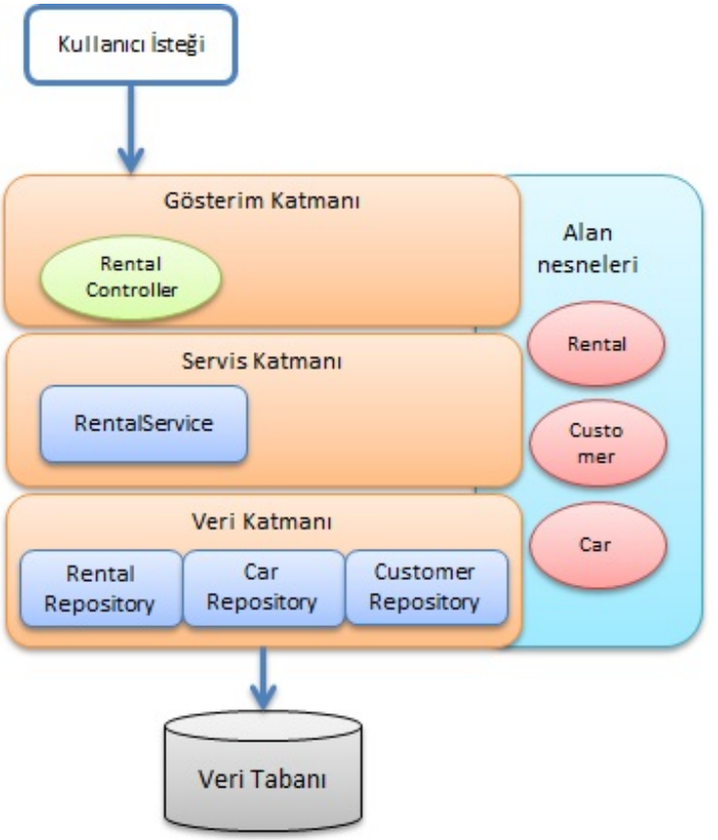
programlanmasını sağlar.

Programcılar kafalarında oluşan modelleri program koduna çevirirler. Bu süreçte çoğu zaman uygulamayı kullanıcı gözlüğüyle değil, programcı gözlüğüyle görürler. Buda belki ilk etapta gerekli olmayan davranışların uygulamaya eklenmesine sebep verebilir. Birim testlerinde bu durum farklıdır. Örneğin onay/kabul testlerinde tüm sistem bir kullanıcının perspektifinden test edilir. Bu testlerde uygulamanın mutlaka sahip olması gerektiği davranışlar test edilmiş olur. Eğer onay/kabul testleri oluşturarak yazılım sürecine başlarsak, testin öngördüğü fonksiyonları implemente ederiz. Böylece gereksiz ve belki bir zaman sonra kullanılabileceğini düşünerek oluşturduğumuz fonksiyonlar programlanmaz. TDD ile testler oluşturulan uygulamaya paralel olarak oluşur. Sonradan bir uygulama için onlarca ya da yüzlerce birim testi oluşturmak çok zor olacağı için birim testlerini oluşturarak yazılama başlamak çok daha mantıklıdır.

Spring bünyesinde özellikle uygulamayı sunucu dışında (out of container) test etmek birçok araç bulunmaktadır. Bu Spring ile test güdümlü yazılım yapmayı kolaylaştıran bir durumdur. Bu bölümde araç kiralama uygulamasını test güdümlü yazılım yöntemleri ve Spring'in ihtiva ettiği Spring TestContext ve Spring MVC Test test çatılarıyla sıfırdan geliştireceğiz.

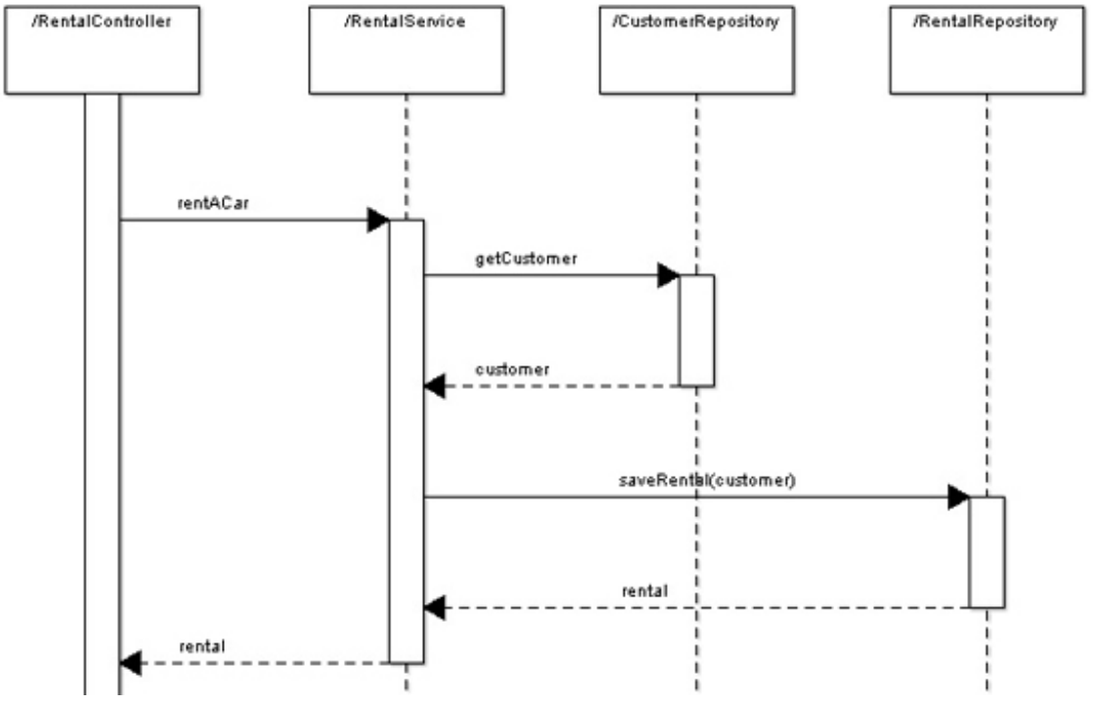
TDD ile Gösterim Katmanı

Araç kiralama uygulamasının mimari yapısı resim 15.2 de yer almaktadır. Resim 15.2 de yer alan sınıfların var olmadıklarını var sayarak, böyle bir uygulamayı test güdümlü nasıl geliştirebileceğimizi inceleyelim.



Resim 15.2

Normal şartlar altında bir uygulamanın gösterim katmanını test etmek için onay/kabul testleri kullanılır. Bu tür testler doğaları itibariyle entegrasyon testleridir. Doğal olarak testler tüm uygulamanın bir uygulama sunucusu içinde çalışır bir hale olmasını bekler. Bu uygulamanın test edilmesini güçleştiren bir durumdur. Bu şartlar altında test güdümlü yazılım yapmak mümkün değildir ya da çok zordur. Spring ile durum biraz farklı. Test güdümlü yazılım yapma niyetine sahip olduğumuzu söylemiştik. Bu durumda resim 15.2 de yer alan araç kiralama uygulamasını test güdümlü nasıl geliştirebiliriz? Elimizde uygulamanın hiçbir parçasının olmadığını unutmayalım. Uygulamayı gerçek anlamda testler yazarak sıfırdan geliştireceğiz. Bunu nasıl yapabileceğimizi bir örnek üzerinden inceleyelim.



Resim 15.3

Resim 15.3 de araç kiralama uygulamasının dizge akış diyagramı yer almaktadır. RentalController sınıfı gösterim katmanında yer alan Spring MVC controller sınıfıdır ve işlevini yerine getirebilmek için servis katmanında yer alan RentalService sınıfına ihtiyaç duymaktadır. Bu durumda RentalController sınıfını oluşturabilmek ve çalışır hale getirmek için RentalService sınıfının var olması gerekmektedir. Lakin böyle bir sınıfa sahip değiliz. Ama bu bir sorun teşkil etmiyor. Bu sınıfı bir interface sınıfı olarak tanımlayıp, bir mock implementasyonunu kullanabiliriz. Kod yazmaya RentalControllerTest sınıfını oluşturarak başlıyoruz. Bu sınıf kod 15.1 de yer almaktadır.

```
Kod 15.1 - RentalControllerTest.java

package com.kurumsaljava.spring.web;

import static org.hamcrest.Matchers.hasKey;
import static org.hamcrest.Matchers.hasValue;
import static org.springframework.test.web.servlet.request.
            MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
            MockMvcResultMatchers.forwardedUrl;
import static org.springframework.test.web.servlet.result.
            MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.
            MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.
            MockMvcResultMatchers.view;
```

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
            SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:test-config.xml" })
public class RentalControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.
                webAppContextSetup(this.wac).build();
    }

    @Test
    public void rental_form_should_be_loaded()
        throws Exception {

        this.mockMvc.perform(get("/rental"))
            .andExpect(status().isOk())
            .andExpect(view().name("rental"))
            .andExpect(forwardedUrl(
                    "/WEB-INF/views/rental.jsp"))
            .andExpect(model().attributeExists("cars"))
            .andExpect(model().attribute("cars",
                    hasKey("")))
            .andExpect(model().attribute("cars",
                    hasValue("")))
            .andExpect(model().attribute("cars",
                    hasKey("1")))
            .andExpect(model().attribute("cars",
                    hasValue("Ford Fiesta")))
            .andExpect(model().attribute("cars",
                    hasKey("2")))
```

```
            .andExpect(model().attribute("cars",
                    hasValue("Renault Twingo")))
            .andExpect(model().attribute("cars",
                    hasKey("3")))
            .andExpect(model().attribute("cars",
                hasValue("Audi A4")));
    }
}
```

Spring 3.2 öncesi Spring MVC uygulamalarını sunucu dışında test etmek için MockHttpServletRequest ve MockHttpServletResponse gibi sınıfları kullanılırdı. Bu sınıflar isimlerinden de anlaşıldığı gibi kullanıcı ile sunucu arasındaki iletişimi modelleyen ve Servlet API'sinde yer alan HttpServletRequest ve HttpServletResponse sınıflarının mock implementasyonlarıdır. Bu sınıflar aracılığı ile bir Spring MVC uygulaması her yönü ile test etmek mümkün olmadığı için, Spring 3.2 ile birlikte MockMvc ismini taşıyan ve Spring MVC Test çatısının parçası olan sınıf geliştirilmiştir. Şimdi gelin kod 15.1 de yer alan RentalControllerTest sınıfını satır, satır inceleyelim.

@RunWith anotasyonu ile bir JUnit koşturucu sınıfı tanımlanmaktır. Bu Spring TestContext çatısının merkezi sınıflarından birisidir. @WebAppConfiguration ile test bünyesinde oluşturacağımız Spring ApplicationContext nesnesinin WebApplicationContext türünde olmasını sağlıyoruz, çünkü test ettiğimiz uygulama bir Spring MVC uygulamasıdır.

@ContextConfiguration kullandığımız Spring konfigürasyon dosyasına işaret etmektedir. @ContextConfiguration bünyesinde herhangi bir konfigürasyon dosyası yer almadığı taktirde, Spring <sınıf-ismi>-context.xml isminde bir dosyayı arayacaktır. test-config.xml dosyasının içeriği kod 15.2 de yer almaktadır.

```
Kod 15.2 - test-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
            http://www.springframework.org/schema/beans/
                spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
```

```
            http://www.springframework.org/schema/context/
                spring-context-3.0.xsd
            http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/
                spring-tx-3.0.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/
                spring-mvc-3.0.xsd">


    <context:component-scan base-package="com.kurumsaljava" />

    <mvc:annotation-driven />

    <bean id="rentalService"
            class="com.kurumsaljava.spring.web.
                    MockitoFactoryBean">
        <constructor-arg name="classToBeMocked"
            value="com.kurumsaljava.spring.service.
                RentalService" />
    </bean>

    <bean
        class="org.springframework.web.servlet.view.
            InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

test-config.xml tipik bir Spring MVC konfigürasyonudur. rentalService isimli Spring nesnesinin ne olduğunu daha sonra yakından inceleyeceğiz.

Uygulamayı test edebilmek için mockMvc nesnesini oluşturmamız gerekiyor. Bunu sanal bir sunucu olarak düşünebilirsiniz. Daha sonra oluşturacağımız kullanıcı isteğini bu sanal sunucuya göndereceğiz. rental_form_should_be_loaded() metodu ile RentalController sınıfının getForm() metodunu test ediyoruz. RentalController sınıfı kod 15.3 de yer almaktadır.

this.mockMvc.perform(get("/rental")) şeklinde bir tanımlama, RentalController sınıfının getForm() metodunu koşturacaktır, çünkü RentalController bünyesinde bu metot @RequestMapping(method = RequestMethod.GET) ile HTTP GET metodu kullanıldığında koşturulacak şekilde tanımlanmıştır.

```
Kod 15.3 - RentalController.java

@Controller
@RequestMapping("/rental")
public class RentalController {

    private static final String RENTAL_VIEW = "rental";
    private static final String RENTAL_FORM = "rentalForm";

    @Inject
    private RentalService service;

    @RequestMapping(method = RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        this.buildCarList(model);
        return RENTAL_VIEW;
    }

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(final ModelMap model,
            @ModelAttribute(RENTAL_FORM) @Valid final Rental rental,
            final BindingResult result) throws IOException {
        if (result.hasErrors()) {
            this.buildCarList(model);
            return RENTAL_VIEW;
        } else {
            this.processRental(rental);
            return "redirect:rental/done";
        }
    }

    private void buildCarList(final ModelMap model) {
        final Map<String, String> cars = new LinkedHashMap<String,
                String>();
        cars.put("", "");
        cars.put("1", "Ford Fiesta");
        cars.put("2", "Renault Twingo");
        cars.put("3", "Audi A4");
        model.put("cars", cars);
    }

    private void processRental(final Rental rental) {
        this.service.rentACar(rental);
    }
}
```

Şimdi rental_form_should_be_loaded() metodunda kullanılan yapıyı satır, satır

inceleyelim:

```
this.mockMvc.perform(get("/rental"))
```

perform() metoduyla kullanıcı istediğini tetikler. RentalController.getForm() metoduna bir break point koyduğunuz taktirde mockMvc.perform() metodunun çalışmasıyla birlikte debuggerın getForm() metodundaki break pointte durduğunu görürsünüz. Bu Spring MVC uygulamasının kod 15.2 de kullandığımız konfigürasyonla çalışmaya başladığına işaret eden bir durumdur.

```
andExpect(status().isOk())
```

andExpect() bir beklentiyi ifade etmek için kullanılan metottur. Burada kullanıcı isteğinin RentalController.getForm() metodunda işlenmesinden sonra HTTP statü kodunun 200 olması beklenmektedir. Bu kullanıcı isteğinin hatasız bir şekilde işlenmiş olması gerektiği anlamına gelmektedir.

```
andExpect(view().name("rental"))
```

Burada controller sınıfının rental isimli view elementine yönlendirme yapmış olması beklentisi ifade edilmektedir. Kod 15.3 de yer alan getForm() metoduna göz attığımızda, bir hata oluşmaması durumunda bu metodun geri dönüş değerinin rental olduğunu görmekteyiz.

```
andExpect(forwardedUrl("/WEB-INF/views/rental.jsp"))
```

Burada uygulamanın hangi view elementine yönlendirme yaptığı kontrol edilmektedir.

```
andExpect(model().attributeExists("cars"))
```

Controller sınıfında yer alan getForm() metodu yaptığı işlemi tamamladığında model nesnesi içinde cars ismini taşıyan bir map oluştur. Bu map içinde kiralanabilecek araçların listesi yer almaktadır.

```
andExpect(model().attribute("cars", hasKey("")))
andExpect(model().attribute("cars", hasValue("")))
andExpect(model().attribute("cars", hasKey("1")))
andExpect(model().attribute("cars", hasValue("Ford Fiesta")))
andExpect(model().attribute("cars", hasKey("2")))
andExpect(model().attribute("cars", hasValue("Renault Twingo")))
andExpect(model().attribute("cars", hasKey("3")))
```

```
andExpect(model().attribute("cars", hasValue("Audi A4")));
```

Son olarak map içinde hangi değerlerin olduğunu kontrol ediyoruz. Bu şekilde sunucu olmadan RentalController sınıfını test etmek mümkün oldu. Lakin testin ve RentalController sınıfının çalışabilmesi için RentalService sınıfını tanımlamamız gerekiyor. Kod 15.3 e baktığımızda şu satırı görmekteyiz:

```
@Inject
private RentalService service;
```

RentalController sınıfının vazifesini yerine getirebilmesi için servis katmanında yer alan RentalService sınıfına ihtiyaç duymaktadır. Yukarıda yer alan satırda RentalService sınıfından bir nesne RentalController sınıfından olan nesneye enjekte edilmektedir. Bu henüz implementasyonu olmayan bir interface sınıftır. Kodu 15.4 de yer almaktadır.

```
Kod 15.4 - RentalService.java

package com.kurumsaljava.spring.service;

import com.kurumsaljava.spring.domain.Rental;

public interface RentalService {

    Rental rentACar(Rental rental);
}
```

Peki implemenetasyonu olmayan bir sınıfı somut bir nesneye nasıl enjekte edebiliriz? Bunu Mockito mock çatısını kullanarak yapabiliriz. Bu amaçla test-config.xml bünyesinde şu tanımlamayı yapıyoruz:

```
<bean id="rentalService"
    class="com.kurumsaljava.spring.web.MockitoFactoryBean">
    <constructor-arg name="classToBeMocked"
        value="com.kurumsaljava.spring.service.RentalService" />
</bean>
```

MockitoFactoryBean kod 15.5 de yer almaktadır. Bu tanımlama ile Spring RentalController nesnesini oluştururken sahip olduğu rentalService değişkenine Mockito ile oluşturulan mock nesneyi enjekte edecektir.

```
Kod 15.5 - MockitoFactoryBean.java

package com.kurumsaljava.spring.web;
```

```
import org.mockito.Mockito;
import org.springframework.beans.factory.FactoryBean;

public class MockitoFactoryBean<T> implements FactoryBean<T> {

    private final Class<T> classToBeMocked;

    public MockitoFactoryBean(final Class<T> classToBeMocked) {
        this.classToBeMocked = classToBeMocked;
    }

    @Override
    public T getObject() throws Exception {
        return Mockito.mock(this.classToBeMocked);
    }

    @Override
    public Class<?> getObjectType() {
        return this.classToBeMocked;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

Şimdi controller sınıfının HTTP Post metoduyla çalışan processSubmit() metodunu test edelim. Oluşturacağımız ikinci test veriler üzerinden yapılan kontrolleri (validation) test etmektedir. Kullanıcı ekranda yer alan araç kiralama formunu doldurup, gönder tuşuna tıkladıktan sonra, bu veriler sunucuya, oradan da controller sınıfını aktarılır. Kod 15.6 da yer alan test metodu özellikle sunucuya gönderilen verilerin eksik olmasını sağlayarak, validation işlemini test etmektedir.

```
Kod 15.6 - RentalControllerTest.java

@Test
public void validation_check_should_work() throws Exception {

    this.mockMvc.perform(post("/rental")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("customer.name", "acar"))
            .andExpect(status().isOk())
            .andExpect(model().hasErrors());
```

```
}
```

Kod 15.3 de yer alan processSubmit metodu metot parametlerinden olan Rental değişkeni için @Valid anotasyonunu taşımaktadır. Bu anotasyon Rental sınıfında tanımlanan veri kontrol mekanizmalarını tetikler. Rental sınıfı kod 15.7 de yer almaktadır. validation_check_should_work() bünyesinde bilinçli olarak sunucuya eksik veri gönderdiğimiz için, veri kontrolünün tetiklenmiş ve hata tespiti yapılmış olmasını beklemekteyiz.

```
Kod 15.7 - Rental.java

public class Rental {

    @Id
    @GeneratedValue
    @Column(name = "id")
    private long id;

    @Valid
    @OneToOne
    private Car car;

    @Valid
    @OneToOne
    private Customer customer;

    @Column(name = "rented")
    private boolean rented;
}
```

Oluşturacağımız son test metodu servis katmanının kullanımını test edecektir. Bu test submit_should_work ismini taşımaktadır ve kod 15.8 de kodu yer almaktadır.

```
Kod 15.8 - RentalControllerTest.java

@Test
public void submit_should_work() throws Exception {

    this.mockMvc.perform(post("/rental")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("customer.firstname", "Filiz")
            .param("customer.name", "Akin")
            .param("car.userSelection", "1"))
            .andExpect(status().is(302))
            .andExpect(model().hasNoErrors());
```

```
}
```

submit_should_work() test metodu bünyesinde servis katmanının kullanımı için gerekli veriler yer almaktadır. Burada araç kiralama formunda yer alan bilgileri kullanarak bir kullanıcı isteği (request) oluşturduk. Bu test metodunu koşturduğumuzda, kullanıcı isteği RentalController sınıfının processSubmit() metoduna yönlendirilir. Spring MVC processSubmit() metodunu koşturmadan önce gerekli veri kontrollerini (validation) yapar. Sunucuya gerekli olan tüm verileri gönderdiğimiz için processSubmit() metodu else bloğunda yer alan işlemleri gerçekleştirir, yani servis katmanında yer alan RentalService sınıfının rentACar() metodunu koşturur. Bu bir Mockito mock nesnesi olduğu için rentACar() nesnesi sessiz, sedasız, işlem yapmış gibi normal geri dönüş yapar. processSubmit() metodu en son işlem yaparak rental/done view elementine yönlendirme yapar ki, bunu da submit_should_work() metodunda .andExpect(status().is(302)) ile test etmekteyiz.

TDD ile Servis Katmanı

Kitabın diğer bölümlerinde RentalService ve implementasyonu olan RentalServiceImpl sınıfları ile karşılaşmıştık. Bu bölümde servis katmanınında yer alan RentalServiceImpl sınıfının nasıl test güdümlü implemente edilebileceğini inceleyeceğiz. Amacımız yine mock nesneler kullanarak, veri katmanıyla entegrasyona gitmek zorunda kalmadan RentalServiceImpl sınıfını implemente ve test edebilmek. Bu amaçla veri katmanında yer alan repository interface sınıflarından mock nesneler oluşturmamız gerekiyor. Ama önce test sınıfını görelim. Servis katmanını test eden RentalServiceImplTest kod 15.9 da yer almaktadır.

```
Kod 15.9 - RentalServiceImplTest.java

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "
            classpath:test-service-config.xml" })
public class RentalServiceImplTest {

    @Inject
    RentalService service;

    @Test
    public void test_rental() {
```

```
        final Rental rental = this.getDummyRental();
        final Rental result =
            this.service.rentACar(rental);
        assertEquals(result.getId(), 1);
    }
}
```

RentalServiceImpl sınıfını kod 15.10 da görüldüğü gibi implemente ediyoruz.

```
Kod 15.10 - RentalServiceImpl.java

@Service
public class RentalServiceImpl implements RentalService {

    @Inject
    private CustomerRepository customerRepository;

    @Inject
    private RentalRepository rentalRepository;

    @Inject
    private CarRepository carRepository;

    public RentalServiceImpl() {
    }

    @Override
    public Rental rentACar(final Rental rental) {

        final Customer dbCustomer = this.customerRepository.
        getCustomerByName(rental.getCustomer().getName());

        if (dbCustomer == null) {
            this.customerRepository.save(rental.getCustomer());
        }

        final Car car = this.carRepository.findCarById(1);

        rental.setCar(car);
        rental.setRented(true);
        rental.setCustomer(rental.getCustomer());
        this.rentalRepository.save(rental);
        return rental;
    }
}
```

RentalServiceImpl sınıfı görevini yerine getirebilmek için veri katmanında yer

alan CustomerRepository, RentalRepository ve CarRepository sınıflarını kullanmaktadır. Bu sınıfların henüz bir implementasyonu olmadığı ve zaten RentalServiceImplTest bünyesinde entegrasyon testi değil, birim testi yaptığımız için, bu sınıfları kullanarak Mockito yardımı ile şu şekilde mock nesneler tanımlıyoruz:

```
Kod 15.11 - test-service-config.xml

<bean id="customerRepository"
        class="com.kurumsaljava.spring.web.MockitoFactoryBean">
    <constructor-arg name="classToBeMocked"
        value="com.kurumsaljava.spring.dao.CustomerRepository" />
</bean>

<bean id="carRepository"
            class="com.kurumsaljava.spring.web.MockitoFactoryBean">
    <constructor-arg name="classToBeMocked"
        value="com.kurumsaljava.spring.dao.CarRepository" />
</bean>

<bean id="rentalRepository"
        class="com.kurumsaljava.spring.web.MockitoFactoryBean">
    <constructor-arg name="classToBeMocked"
        value="com.kurumsaljava.spring.dao.RentalRepository" />
</bean>
```

Spring kod 15.9 da yer alan @ContextConfiguration yardımı ile bu mock nesneleri kod 15.10 da yer alan RentalServiceImpl sınıfına enjekte edecektir. Kod 15.9 da yer alan test_rental() test metodunun olumlu sonuç verebilmesi için, test sınıfına Spring tarafından service nesnesi enjekte edilir, çünkü kod 15.10 da yer alan RentalServiceImpl sınıfı @Service anotasyonunu taşımaktadır ve Spring otomatik olarak @Inject ile kullanılarak oluşturulan RentalService tipindeki nesnelere RentalServiceImpl sınıfından bir nesneyi enjekte eder. Bu şekilde test esnasında da bağımlılıkların enjekte edilmesi prensibinden yararlanarak, tüm gerekli konfigürasyonun Spring tarafından yapılmasını sağlamaktayız. Bu oluşturduğumuz testlerin daha sade bir yapıda olmalarını sağlamaktadır.

Kod 15.9 da yer alan test_rental() metodu Spring ve Mockito'nun sağladığı imkanları göstermek için hazırladığım düz bir testtir. Mockito kullanılarak oluşturulan mock nesnelere değişik davranış biçimleri kazandırmak mümkündür. Bu şekilde kod 15.10 da yer alan rentACar() metodunu daha değişik yönlerden test etmek mümkün olacaktır. Bu amaçla Mockito ile mock

nesneleri oluşturma ve programlama konusunu incelemenizi tavsiye ediyorum.

Veri Katmanı İçin Entegrasyon Testleri

Veri katmanı en altta yer alan ve doğrudan veri tabanı ile iletişim içinde olan bir katmandır. Mock nesneler kullanarak bu katmanı test etmeyi mantıklı bulmuyorum, çünkü doğrudan verilere erişmemiz gerekiyor. Mock nesneler ile edineceğimiz veriler bizi yanıltabilir. Entegrasyon testleri oluşturarak bu katmanı test etmek daha yerinde olacaktır. Veri katmanının bir parçası olan CarRepositoryImpl sınıfını test etmek amacıyla kod 15.12 de yer alan test sınıfını oluşturuyoruz.

```
Kod 15.12 - CarRepositoryImplTest.java

package com.kurumsaljava.spring.dao;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;
import javax.inject.Inject;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
        SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;

import com.kurumsaljava.spring.domain.Car;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:test-data-config.xml" })
public class CarRepositoryImplTest {

    @Inject
    CarRepository repository;

    @Test
    public void test_rental_repository() {
        final Car car = this.repository.findCarById(1000);
        assertThat(car.getBrand(), equalTo("Ford"));
    }
}
```

test-data-config.xml konfigürasyon dosyasının içeriği şu şekildedir:

```
Kod 15.13 - test-data-config.xml

<context:component-scan base-package="com.kurumsaljava" />

<bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>

<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:data.sql" />
</jdbc:embedded-database>
```

Test ettiğimiz CarRepositoryImpl implementasyonu kod 15.14 de yer almaktadır. CarRepositoryImpl veri tabanı işlemleri için JdbcTemplate sınıfını kullanmaktadır.

```
Kod 15.14 - CarRepositoryImpl.java

@Repository
public class CarRepositoryImpl implements CarRepository {

    @Inject
    private JdbcTemplate jdbcTemplate;

    @Override
    public Car findCarById(final long id) {
        final ResultSetExtractor<Car> carExtractor =
            new CarExtractor();
        final String sql = "select id, brand, model
                    from car where id=?";
        return this.getJdbcTemplate().query(sql,
            carExtractor, id);
    }

    public void setJdbcTemplate(
            final JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    private class CarExtractor implements ResultSetExtractor<Car> {

        @Override
        public Car extractData(final ResultSet rs) throws
                        SQLException, DataAccessException {
            return CarRepositoryImpl.this.mapCar(rs);
```

```
        }
    }

    private Car mapCar(final ResultSet rs) throws SQLException {
        Car car = null;
        while (rs.next()) {
            if (car == null) {
                final String brand = rs.getString("brand");
                final String model = rs.getString("model");
                car = new Car(brand, model);
                car.setId(rs.getLong("id"));
            }
        }
        if (car == null) {
            throw new EmptyResultDataAccessException(1);
        }
        return car;
    }
}
```

Kod 15.13 de yer alan embedded-database elementini daha önceki bölümlerde kullanmıştık. Bu elementi kullanarak hafızada çalışan bir HSQL veri tabanı oluşturuyoruz. Her defasında testler koşturulduğunda tanımlı bir veri setinin veri tabanında yer alması gerekmektedir, aksi taktirde entegrasyon testleri istenilen neticeyi vermeyebilir. Bu amaçla veri tabanı şemasını schema.sql, veri tabanında olmasını istediğimiz verileri data.sql dosyalarında tanımlayarak script elementine bu dosya isimlerini parametre olarak veriyoruz. Spring testleri koşturmadan önce HSQL veri tabanı gerekli yapıya getirerek, testlerin tanımlı bir ortamda koşmalarını sağlayacaktır. Kod 15.13 de yer alan dataSource tanımlamasını değiştirerek, başka bir veri tabanı sistemini kullanabiliriz. CarRepositoryImplTest klasik bir entegrasyon testidir, çünkü dış bir kaynak olan veri tabanı ile entegre edilen uygulamayı test etmektedir.

EmbeddedDatabaseBuilder Kullanımı

Kod 15.13 de yer alan embedded-database yerine EmbeddedDatabaseBuilder sınıfını kullanabiliriz. Kod 15.15 de yer alan EmbeddedCarRepositoryImplTest sınıfı EmbeddedDatabaseBuilder yardımı ile hafızada oluşturulan HSQLDB veri tabanını kullanan bir entegrasyon testidir.

```
Kod 15.15 - EmbeddedCarRepositoryImplTest.java

package com.kurumsaljava.spring.dao;
```

```
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

import javax.sql.DataSource;

import org.junit.Before;
import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.
            EmbeddedDatabaseBuilder;

import com.kurumsaljava.spring.domain.Car;

public class EmbeddedCarRepositoryImplTest {

    CarRepository repository;

    @Before
    public void setup() {
        final JdbcTemplate template =
            new JdbcTemplate(this.createTestDataSource());
        this.repository = new CarRepositoryImpl(template);

    }

    @Test
    public void test_rental_repository() {
        final Car car = this.repository.findCarById(1000);
        assertThat(car.getBrand(), equalTo("Ford"));
    }

    private DataSource createTestDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setName("rental")
            .addScript("schema.sql")
            .addScript("data.sql")
            .build();
    }
}
```

Testlerde Kullanılabilecek Anotasyonlar

@ContextConfiguration - Bir Spring ApplicationContext oluşturmak için kullanılır. ApplicationContext oluşturmak için XML ya da Java sınıfı bazlı konfigürasyon kullanılabilir.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:test-data-config.xml" })
public class CarRepositoryImplTest{
}

@ContextConfiguration(classes = TestConfig.class)
public class CarRepositoryImplTest{
}
```

@WebAppConfiguration - @ContextConfiguration tarafından oluşturulan ApplicationContext nesnesinin bir WebApplicationContext nesnesine dönüştürülmesini sağlar. Web uygulamalarının testi edilebilmesi için gerekli olan bir anotasyondur.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:test-config.xml" })
public class RentalControllerTest {
}
```

@ContextHierarchy - Konfigürasyon dosyaları arasında hiyerarşi oluşturmak için kullanılan anotasyondur. XML ya da Java sınıfı bazlı konfigürasyon kullanılabilir

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")})
public class RentalControllerTest {
}


@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = AppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class RentalControllerTest {
}
```

@DirtiesContext - Test esnasında mevcut ApplicationContext nesnesinin tahribata uğradığı ve test sonunda yok edilmesini gerektiğini ifade eder.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:test-config.xml" })
@DirtiesContext
public class RentalControllerTest {
}
```

@Timed - Metodun tanımlanan zaman biriminde tamamlanması gerektiğini ifade eder. Bu zaman birimi aşıldığı taktirde test başarısız olur. Belirlen zaman birimi testin kendisinin, setup() ve teardown() metotlarının koşturulma zamanlarını kapsamaktadır.

```
@Timed(millis = 1000)
@Test
public void test_rental_repository() {
    final Car car = this.repository.findCarById(1000);
}
```

@Repeat - Metodun kaç sefer tekrarlanacağını ifade eder. Metodun tekrarı, setup() ve teardown() metotların da her tekrarda koşturulmasını gerektirir. Aşağıda yer alan test metodu test_rental_repository() 10 kere ard, arda koşturulur.

```
@Repeat(10)
@Test
public void test_rental_repository() {
    final Car car = this.repository.findCarById(1000);
}
```

@IfProfileValue - Testlerin belli bir test ortamında koşturulmaları gerektiğini ifade eder. Aşağıda yer alan örnekte testler Sun firmasının sahip olduğu JDK/JVM kullanıldığı taktirde koşturulur.

```
@IfProfileValue(name="java.vendor", value="Sun Microsystems Inc.")
@Test
public void test_rental_repository() {
    final Car car = this.repository.findCarById(1000);
}
```

@TransactionConfiguration - Testlerin veri tabanı ile ilişkili bir transaksiyon içinde koşmaları gerektiğini ifade eden anostasyondur. Anotasyon bünyesinde kullanılan TransactionManager nesnesi yer alır.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:test-data-config.xml" })
@TransactionConfiguration(transactionManager="txMgr",
            defaultRollback=false)
public class CarRepositoryImplTes
```

@Rollback - Bu anotasyon true değerini taşıdığı taktirde, test sonunda transaksiyon rollback komutuyla geriye alınır.

```
@Rollback(false)
@Test
public void test_rental_repository() {
    final Car car = this.repository.findCarById(1000);
    assertThat(car.getBrand(), equalTo("Ford"));
}
```

@BeforeTransaction - Transaksiyonu tetikleyen test koşturulmadan önce koşturulması gereken metodu tanımlar.

```
@BeforeTransaction
@Test
public void test_rental_repository() {
    final Car car = this.repository.findCarById(1000);
    assertThat(car.getBrand(), equalTo("Ford"));
}
```

@ActiveProfiles - Kullanılması gereken Spring bean tanımlaması profiline işaret eden anotasyondur.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:test-data-config.xml" })
@ActiveProfiles("dev")
public class CarRepositoryImplTest{
}

<beans profile="dev">
    <jdbc:embedded-database id="dataSource">
        <jdbc:script
            location="classpath:schema.sql"/>
        <jdbc:script
            location="classpath:data.sql"/>
    </jdbc:embedded-database>
</beans>
```

15. Bölüm Soruları

15.1 İmplemenetasyonu olmayan bir sınıfı somut bir nesneye nasıl enjekte edebiliriz?

15.2 Spring 3.2 öncesi Spring MVC uygulamalarını test etmek için kullanılan yapılar hangileridir?

15.3 Spring 3.2 gelen ve web uygulamalarını test etmek için kullanılan sınıfın adı nedir?

15.4 Spring Test çatısının sağladığı en büyük avantaj nedir?

15.5 Veri katmanını test etmek için hangi tür testler oluşturulmalıdır?

16. Bölüm

Spring JMX

Bu bölümde Spring ile JMX konusunu inceleyeceğiz. JMX (Java Management Extentions) çalışır durumda olan uygulamaların konfigürasyon yönetimi ve izlenimi (monitoring) için kullanılan bir Java API'dir.

JMX ile şu işlemler yapılabilir:

- Uygulama hakkında çalışırken (runtime) bilgi toplanabilir.
- Uygulama konfigürasyonu dinamik olarak değiştirilebilir.
- Uygulama bünyesinde işlemler tetiklenebilir.
- İşletme mantığı gereksinimler doğrultusunda adapte edilebilir.

Uygulamada belli davranış biçimleri oluşturabilmek için değişkenlere sabit değerler atanır. Bu değerler çoğu zaman uygulamayı konfigüre etmek için oluşturulan konfigürasyon dosyalarından (property file) alınır. Değişkenlere atanan bu değerleri sadece uygulama içinde değiştirmek mümkündür. Değerleri değiştirmek için harici bir müdahale mümkün değildir, çünkü bir Java sınıfında yer alan bir değişkene uygulama dışından bir değer atamak kolay ya da mümkün değildir. Bu konfigürasyon dosyalarından edinilen sabit değerlerle çalıştırılan bir uygulamanın sahip olduğu ve bazı şartlar altında değiştirilmesi gerekli olabilecek davranış biçimlerinin sabit ve değiştirilemez olduğu anlamına gelmektedir. Bu tür değişiklikleri JMX yardımıyla yapmak mümkündür.

JMX'in pratikte nasıl kullanıldığını bir örnek üzerinde inceleyelim. Kod 16.1 de IndexController sınıfı yer almaktadır. log ismini taşıyan değişken aracılığı ile uygulamanın loglama davranışı biçimi yönetilmektedir. index() metoduna baktığımızda isLog() metodu yardımıyla logmala yapmaya izin olup, olmadığı kontrol edilmektedir. Eğer log değişkeni true değerine sahip ise, o zaman loglama işlemi yapılmaktadır. Bu tip bir uygulamada genellikle yazılım sürecinde log değişkeni true değerini taşır. Bu şekilde yazılımcı metotların ne yaptığını daha iyi takip edebilir. Uygulama test edildikten ve sürümü oluşturulduktan sonra log değişkene false delerini atamak için gerekli konfigürasyon yapılır. Uygulama sunucusu üzerinde çalışan bu uygulamada index() metodunda yer alan loglar log dosyalarında görünmez, çünkü loglama işlemi durdurulmuştur.

```
Kod 16.1 - IndexController.java

@Controller
@RequestMapping("/")
public class IndexController {
```

```
    private boolean log = false;

    public boolean isLog() {
        return this.log;
    }

    public void setLog(final boolean log) {
        this.log = log;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String index(final ModelMap model) {

        if (this.isLog()) {
            System.out.println("enter index()");
        }

        model.put("message", "Hello World");

        if (this.isLog()) {
            System.out.println("exit index()");
        }
        return "index";
    }
}
```

Peki uygulama beklendiği şekilde çalışmadığı taktirde ne yapılabilir? Bu durumda yazılımcı olup, bitenleri anlamak için loglama davranış biçimine tekrar ihtiyaç duyacaktır. Bu özelliği çalışır durumda olan bir uygulamada aktif hale getiremeyeceği için, uygulamayı durdurur ve log değişkeni true değerine sahip olacak şekilde rekonfigüre eder ya da JMX kullanır.

JMX kullanıldığı taktirde dışarıdan değiştirilmelerine izin verilmeyen değişkenlere uygulama çalışır durumda olsa bile yeni değerlerin atanması mümkündür. Simdi bunun IndexController sınıfında yer alan log değişkeni üzerinde nasıl yapılacağını görelim. Burada amacımız çalışır durumda olan bir uygulamada kapatılmış olan loglama işleminin tekrar aktif hale getirilmesidir. Bu amaçla log değişkenine true değerini atamamız gerekiyor.

Mevcut bir Java sınıfını JMX tarafından yönetilebilir hale getirmek için Spring bünyesinde yer alan MBeanExporter sınıfı kullanılır. IndexController sınıfında yer alan log değişkeninin değerini JMX üzerinden değiştirebilmek için gerekli Spring konfigürasyonu kod 16.2 de yer almaktadır. Dışarıya açmak istediğimiz, yani export etmek istediğimiz Java sınıfının bir Java Bean olması, yani her

değişke için bir get, bir de set metodunun olması gerekmektedir. Export edilen Java Bean sınıfına JMX terminolojisinde MBean, yani Managed Bean ismi verilmektedir. MBean bünyesinde konfigürasyon yönetimi için gerekli meta bilgileri taşıyan bir nesnedir. Bu meta bilgiler değişkenler (atttibutes) ve metotlar (operations) gibi bilgileri kapsamaktadır. Bu meta bilgileri statik olarak bir Java sınıfı bünyesinde tanımlamak mümkündür. Bunun yanı sıra dinamik türde MBean nesneleri oluşturup, meta bilgileri uygulama çalışırken oluşturmak mümkündür.

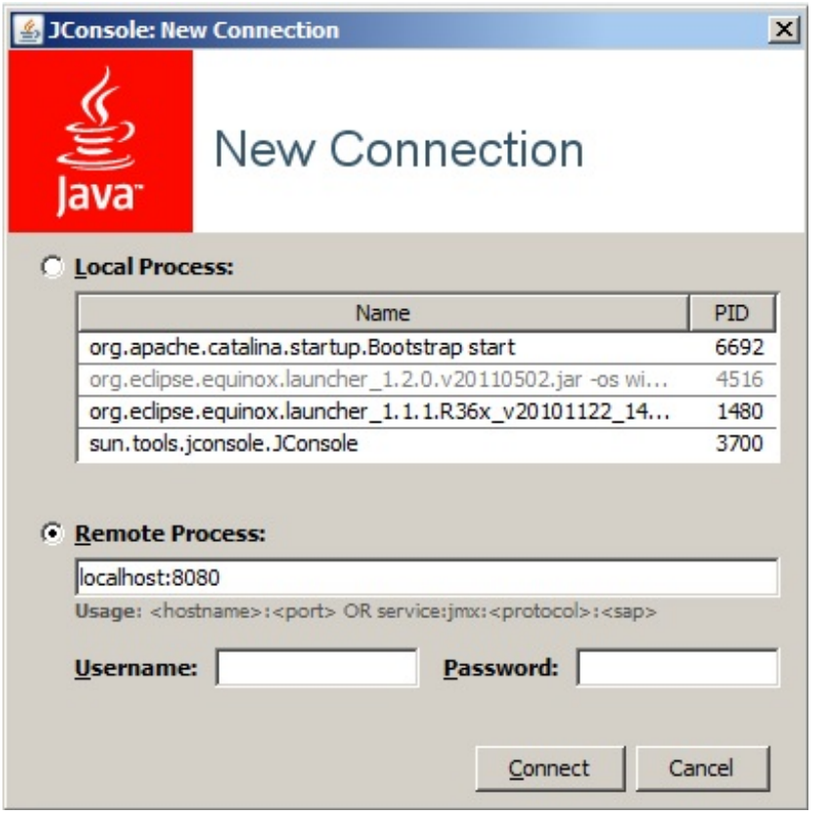
```
Kod 16.2 - jmx-config.xml

<bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=RentalController"
        value-ref="indexController" />
        </map>
    </property>
</bean>
```

Örnek verdiğim uygulama bir web uygulamasıdır. Bu yüzden uygulamayı Tomcat gibi bir uygulama sunucusundan çalışır hale getirmemiz gerekmektedir. Uygulamayı kod 16.2 de yer alan jmx-config.xml konfigürasyon dosyasıyla çalıştırdığımızda pek bir değişiklik olmayacaktır, çünkü kod 16.2 de yer alan konfigürasyon uygulama sunucusunun sunduğu JMX MBean sunucuyu (MBean Server) kullanmaktadır. MBean sunucusu lokal MBean nesneleri ya da MBean nesneleri kullanan uzak kullanıcılarla (remote client) kullanmak istedikleri MBean nesneler arasında aracılık yapmaktadır.

Spring bulunduğu ortamdaki MBean sunucusunu tespit ederek, MBean nesneleri bu sunucu üzerinden kullanıma sunar. Eğer Tomcat bünyesinde JMX Bean sunucu aktif değilse, indexController isimli MBean nesnesine erişmemiz mümkün değildir. Bu yüzden ilk önce Tomcat bünyesindeki MBean sunucusunu aktif hale getirmemiz gerekmektedir. Bunun sağlamak için Tomcat'i çalıştırırken aşağıda yer alan JVM parametrelerinin kullanilması gerekmektedir.

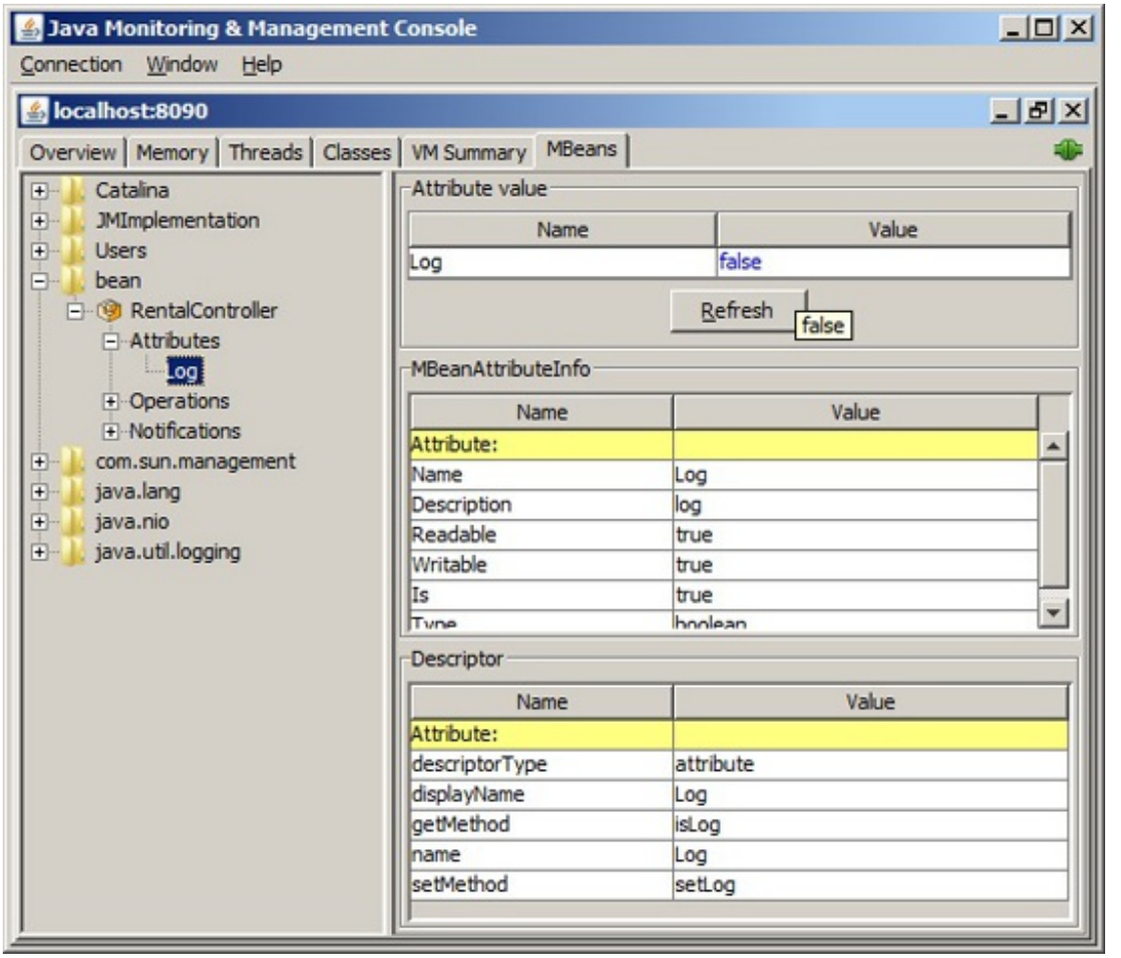
```
-Dcom.sun.management.jmxremote.port=8090
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

jmxremote.port MBean sunucunun port adresini göstermektedir. MBean sunucuya bağlanmak için JDK/bin dizininde yer alan jconsole programından faydalanabiliriz. Uygulama sunucunu çalıştırdıktan sonra resim 16.1 de görüldüğü gibi jconsole ile MBean sunucusuna bağlantı gerçekleştiriyoruz.



Resim 16.1

Resim 16.2 de oluşturduğumuz RentalController MBean görülmektedir. Attributes menüsü altında log değişkeni ve taşıdığı değer yer almaktadır. Bu değişkene true değerini atadığımız taktirde loglama işlemi aktif hale gelecek ve uygulama log yazmaya başlayacaktır. Aynı şekilde false değeri ile loglama işlemini durdurabiliriz.



Resim 16.2

MBean Sunucusu

Daha öncede belittiğim gibi MBean sunucusu MBean nesneleri ve kullanıcıları arasında aracılık yapan sunucudur. Bünyesinde kayıtlı olan her MBean nesnesi için bir referans (kod 16.3 - bean:name) tutar. Bu referans üzerinden sunucu aracılığı ile MBean nesnesini edinmek mümkündür.

Eğer uygulamamız bir MBean sunucu ihtiva eden bir uygulama sunucusu içinde çalışmıyorsa, ya yeni bir MBean sunucu oluşturmamız ya da mevcut bir MBean sunucuyu kullanmamız gerekmektedir. Yeni bir MBean sunucu oluşturmak için context isim alanında yer alan mbean-server elementi kullanılabilir. Kod 16.3 de yeni bir MBean sunucu oluşturulmaktadır. Oluşturulan bu MBean sunucu nesnesinin ismi mbeanServer şeklindedir. Bu nesnenin exporter nesnesinin server isimli değişkenine enjekte edilmesi gerekmektedir. Bu şekilde oluşturulan yeni MBean sunucu üzerinde indexController MBean olarak kullanıma sunulmuş olur.

```
Kod 16.3 - mbean-server-config.xml

<context:mbean-server/>

<bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=RentalController"
            value-ref="indexController" />
        </map>
    </property>
    <property name="server" ref="mbeanServer"/>
</bean>
```

context:mbean-server yerine

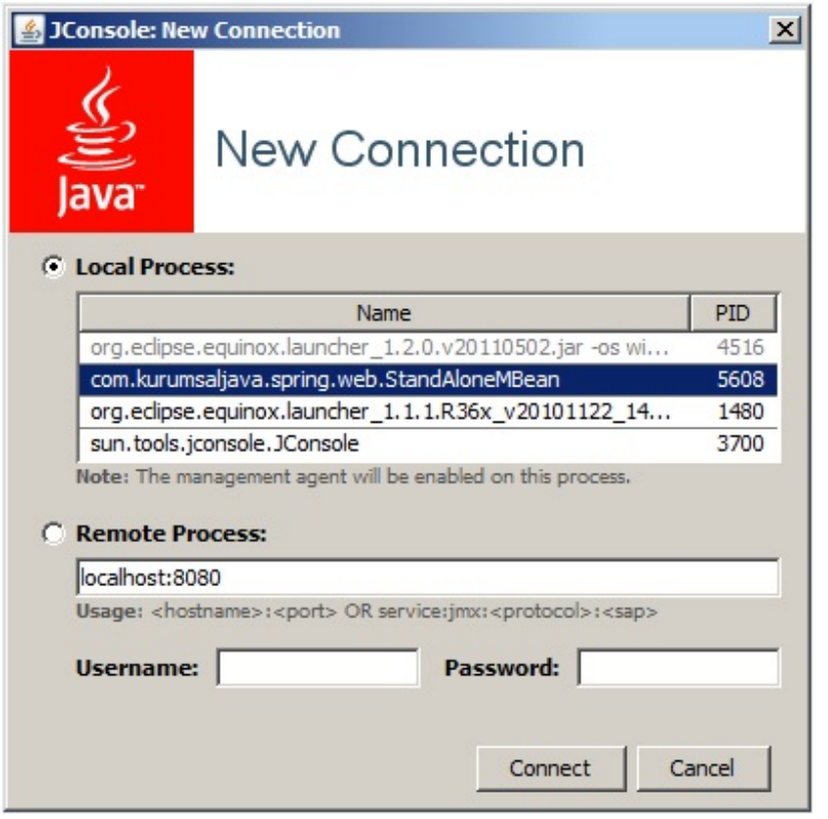
```
<bean id="mbeanServer" class="org.springframework.jmx.support.
        MBeanServerFactoryBean"/>
```

şeklinde de bir MBean sunucu tanımlayabilirdik.

MBean sunucuyu aktif hale getirmek için mbean-server-config.xml konfigürasyon dosyasını yüklemek yeterli olacaktır. Kod 16.4 de yer alan StandAloneMBean bu görevi yerine getirmektedir.

```
KOd 16.4 - StandAloneMBean.java

public class StandAloneMBean {

    public static void main(final String[] args) {
        System.out.println("starting...");
        new ClassPathXmlApplicationContext(
            "mbean-server-config.xml");
        endlessLoop();
    }

    private static void endlessLoop() {
        while (true) {
            try {
                Thread.sleep(10000);
            } catch (final InterruptedException e) {
            }
        }
    }
}
```

Bu işlem sonunda çalışmaya başlayan MBean sunucuya resim 16.3 de görüldüğü gibi bağlantı kurabiliriz.



Resim 16.3

Mevcut bir MBean sunucu şu şekilde kullanılabilir:

```
<bean id="mbeanServer" class="org.springframework.jmx.support.
            MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible"
            value="true"/>
    <property name="agentId" value="<MBeanServer
            instance agentId>"/>
</bean>
```

MBean Kayıt Denetimi

Bir MBean nesnesi kendisini MBean sunucusuna, MBean sunucusu bünyesindeki mevcut bir MBean ismi ile tanıtırsa InstanceAlreadyExistsException hatası oluşur, çünkü iki MBean nesnesi aynı MBean sunucusu bünyesinde aynı ismi paylaşamazlar. Spring MBean nesnelerin kaydı için üç yöntem sunmaktadır. Bunlar:

- ***REGISTRATION_FAIL_ON_EXISTING*** - Yeni bir MBean nesnesi

MBean sunucusunda kayıtlı bir MBean nesnesinin ismini kullanamaz. Kullanırsa InstanceAlreadyExistsException hatası oluşur. Kayıtlı olan MBean nesnesi bu durumdan etkilenmez.

- ***REGISTRATION_IGNORE_EXISTING*** - Yanı MBean ismi kullanıldığında InstanceAlreadyExistsException hatası oluşmaz, lakin yeni MBean için kayıt oluşturulmaz. Kayıtlı olan MBean nesnesi bu durumdan etkilenmez.
- ***REGISTRATION_REPLACE_EXISTING*** - Yanı MBean ismi kullanıldığında bu ismi taşıyan eski MBean nesnesi silinir ve yerine yeni MBean nesnesi kayıtlanır. Böylece yeni MBean nesnesi eski MBean nesnesinin yerine geçmiş olur.

Bu değerleri MBeanRegistrationSupport sınıfında bulmak mümkündür. Varsayılan (REGISTRATION_FAIL_ON_EXISTING) kayıt yöntemini değiştirmek için MBeanExporter sınıfında bulunan registrationBehaviorName değişkenine yukarıda yer alan değerlerden birisi şu şekilde atanabilir:

```
<bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=RentalController"
            value-ref="indexController" />
        </map>
    </property>
    <property name="registrationBehaviorName"
            value="REGISTRATION_REPLACE_EXISTING"/>
    <property name="server" ref="mbeanServer"/>
</bean>
```

Anotasyon Bazlı Spring JMX

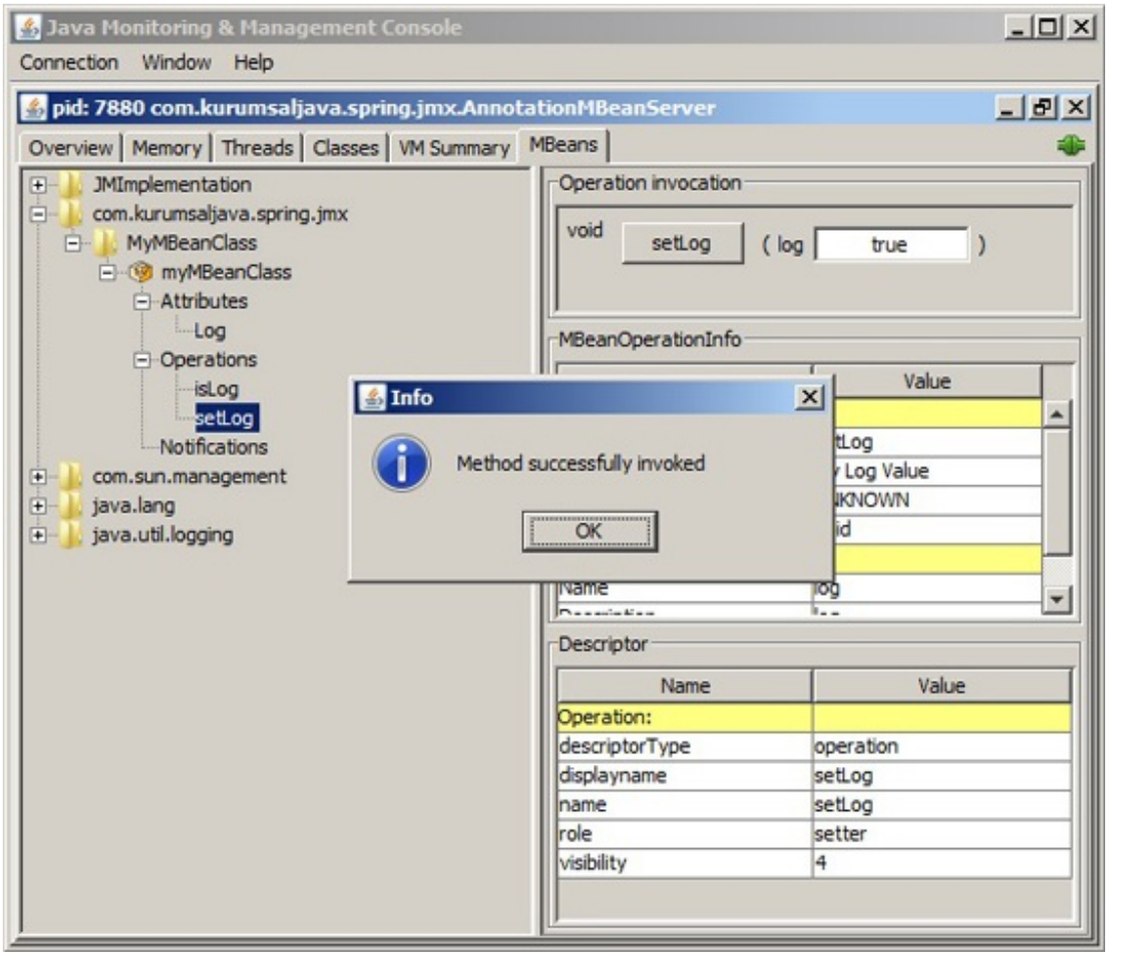
Anotasyon bazlı MBean sunucu ve nesne konfigürasyonu için gerekli elementler kod 16.5 de yer almaktadır. Mbean-server elementi yeni bir MBean sunucu oluştururken, mbean-export @ManagedResource anotasyonunu taşıyan tüm sınıfları MBean olarak export etmektedir.

```
Kod 16.5 - annotation-mbean-server-config.xml

<context:component-scan base-package="com.kurumsaljava" />
<context:mbean-server/>
<context:mbean-export/>
```

@ManagedResource anotasyonunu taşıyan MyMBeanClass sınıfı kod 16.6 da yer almaktadır. @ManagedResource anotasyonunu taşıyan her sınıf bir MBean nesnesi haline gelir. Değişken ve metotların export edilebilmesi için @ManagedAttribute anotasyonu ile işaretlenmeleri gerekmektedir.

```
Kod 16.6 - MyMBeanClass.java

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Component;

@Component
@ManagedResource
public class MyMBeanClass {

    private boolean log;

    @ManagedAttribute
    public boolean isLog() {
        return this.log;
    }

    @ManagedAttribute
    public void setLog(final boolean log) {
        this.log = log;
    }
}
```

Kod 16.6 da yer alan örnekte log değişkeni ve sahip olduğu get ve set metotları export edilmektedir. Bu şekilde hem log değişkeninin değerini değiştirmek hem de jconsole üzerinden get ve set metotlarını koşturmak mümkün olmaktadır. Resim 16.4 de setLog() metodunun koşturulması ve alınan netice yer almaktadır.


Java Monitoring & Management Console
Connection Window Help
pid: 7880 com.kurumsaljava.spring.jmx.AnnotationMBeanServer
Overview Memory Threads Classes VM Summary MBeans
JMImplementation
com.kurumsaljava.spring.jmx
MyMBeanClass
myMBeanClass
Attributes
Log
Operations
isLog
setLog
Notifications
com.sun.management
java.lang
java.util.logging
Operation invocation
void
setLog
(log
true
)
MBeanOperationInfo
Value
Info
Method successfully invoked
OK
Descriptor
Name
Value
Operation:
descriptorType
operation
displayname
setLog
name
setLog
role
setter
visibility
4

Resim 16.4

16. Bölüm Soruları

16.1 MBean nedir?

16.2 MBean sunucusu oluşturmak için kullanılan Spring konfigürasyon elementi hangisidir?

16.3 Bu konfigürasyon elementi hangi Spring sınıfını temsil etmektedir?

16.4 @ManagedResource ne için kullanılır?

16.5 Değişken ve metotların export edilebilmesi için kullanılan anotasyon hangisidir?

17. Bölüm

Spring Task ve Scheduling

Java'da paralel işlem (task) yapmak için kullanılan sınıflardan bir tanesi java.util.concurrent.Executor interface sınıfıdır. Bu sınıf thread yönetimi detaylarını gizlediği için paralel program yazılımını kolaylaştırmaktadır.

```
Kod 17.1 - Executor

public interface Executor{
    public abstract void execute(java.lang.Runnable task);
}
```

java.util.concurrent.ExecutorService Executor sınıfını genişleten ve paralel işlem yapmaya yaşam döngüsü (lifecycle) metotları ekleyen bir diğer interface sınıftır. java.util.concurrent.ThreadPoolExecutor isminde, sıkça kullanılan bir implementasyonu mevcuttur.

```
Kod 17.2 - ExecutorService

public interface ExecutorService extends Executor{
    void shutdown();
    boolean awaitTermination(long amount, TimeUnit unit)
            throws InterruptedException;
    // ...
}
```

ExecutorService interface sınıfında yer alan shutdown() paralel işlemleri sonlandırıp yeni işlem başlatılmasına izin vermezken, awaitTermination() tüm işlemler tamamlanana kadar program akışını bloke etmektedir.

java.util.concurrent.Executors bünyesinde değişik ExecutorService implementasyonlarını edinmek için fabrika (factory) metotları bulunur.

```
Kod 17.3 - Executors

public class Executors{
    public static ExecutorService newFixedThreadPool(int size);
    public static ExecutorService newSingleThreadExecutor();
    public static ExecutorService newCachedThreadPool();
    // ...
}
```

newFixedThreadPool() belli sayda thread ihtiva eden bir thread havuzu (pool) oluştururken, newSingleThreadExecutor() işlemleri tek bir thread ile koşturmaktadır. newCachedThreadPool() yeni bir thread havuzu oluşturmakta, havuzda thread kalmadığında havuza yeni threadler eklemekte, thread varsa bu

threadlari işlem yapmak için kullanmaktadır.

Kod 17.4 de yukarıda bahsettiğim Java sınıflarını kullanarak paralel işlem yaptıran RentalDownloader sınıfı yer almaktadır. IdList isimli listede id bilgisi yer alan Rental nesneleri bir web service kullanıcısını temsil eden RentalWSClient aracılığı ile sunucudan XML formatında alınmaktadır. Bu işlem DownloadTask sınıfı kullanılarak yapılmaktadır. DownloadTask bir thread nesnesidir ve ExecutorService tarafından diğer DownloadTask nesnelerine paralel koşturulmaktadır.

```
Kod 17.4 - RentalDownloader

public class RentalDownloader{

    @Inject
    RentalWSClient client;

    List<Integer> idList;

    public RentalDownloader(List<Integer> ids){
        this.idList = ids;
    }

    public void download() {

        ExecutorService service = getExecutorService();
        for(Integer id: idList){
            service.execute(new DownloadTask(client));
        }

        service.shutdown();
        service.awaitTermination(Long.MAX_VALUE,
            TImeUnit.SECONDS);
    }

    ExecutorService getExecutorService(){
        return Executors.newCachedThreadPool();
    }
}
```

Java API'si ile yaptığımız bu işlemleri Spring'in sunduğu Tasks and Scheduling çatısıyla da yapabiliriz.

Spring TaskExecutor

java.util.concurrent.Executor ve implementasyonları ile Java 5 de tanıştık. Onun öncesinde paralel işlem yaptırmak için Java 1.4 bünyesinde sadece thread kullanımı mümkündü. Bunun yanı sıra Java EE ortamlarında thread yönetimi uygulama sunucusu tarafından yapılmaktadır. Görüldüğü gibi bu üç değişik ortamda paralel işlem yaptırmak için değişik yöntemler kullanılabilmektedir. Spring paralel işlem görevlerini aynı paydada toplamak için TaskExecutor sınıfını sunmaktadır. TaskExecutor implementasyon detaylarını gizlemekte ve kullanıcıya tek bir programlama modeli sunmaktadır. Aslında Spring'in her konuda nihayi amacı bu değil midir? Spring'in paralel işlemler için de böyle bir modeli sunmasına şaşırmamak lazım :)

Spring TaskExecutor sınıfı java.util.concurrent.Executor ile aynı yapıdadır ve Spring bünyesinde şu implementasyonlara sahiptir:

SimpleAsyncTaskExecutor

Gerçek bir thread havuzu oluşturmaz. Her yeni görev (task) için yeni bir thread koşturur. Thread sayısı limitlenebilir. Boş thread kalmadığında thread oluşturmak isteyen uygulamayı bloke eder. Bir görevin son bulmasıyla görevi koşturan thread de sonlandırılır.

SyncTaskExecutor

Görevleri (task) SyncTaskExecutor sınıfını koşturan sınıfın sahip olduğu thread içinde senkron olarak koşturur. Paralel işlem yapma gerekliliği olmadığı durumlarda kullanılmalıdır.

ConcurrentTaskExecutor

Bir java.util.concurrent.Executor implementasyonu kapsülleyerek (wrapper) TaskExecutor olarak kullanılmasını sağlar.

SimpleThreadPoolTaskExecutor

Quartz çatısında yer alan SimpleThreadPool sınıfının bir altsınıfıdır. InitializingBean ve DisposableBean Spring lifecycle (hayat döngüsü) inteface sınıfları implemente eder. Quartz kullanan ve kullanmayan uygulama modülleri arasında thread havuzu paylaşımı için kullanılır.

ThreadPoolTaskExecutor

Java 5 ve sonrası ortamlarda kullanılabilecek TaskExecutor implementasyonudur. Bir java.util.concurrent.ThreadPoolExecutor nesnesini

konfigüre edebilmek için konfigürasyon değişkenlerine dışarıdan değer atanmasını mümkün kılar.

TaskExecutor Kullanımı

Kod 17.5 de yer alan MyTaskExecutor bünyesinde bir TaskExecutor ihtiva etmektedir. printMessages() metodunda taskExecutor kullanılarak MyTask tipinde görevler oluşturulmaktadır.

```
Kod 17.5 - MyTaskExecutor

package com.kurumsaljava.spring.task;

import org.springframework.core.task.TaskExecutor;

public class MyTaskExecutor {

    private final TaskExecutor taskExecutor;

    private class MyTask implements Runnable {

        private final String message;

        public MyTask(final String message) {
            this.message = message;
        }

        public void run() {
            System.out.println(this.message);
        }
    }

    public MyTaskExecutor(final TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public void printMessages() {
        for (int i = 0; i < 25; i++) {
            this.taskExecutor.execute(
            new MyTask("Message" + i));
        }
    }
}
```

MyTask tipindeki görevlerin nasıl koşturulacağını kullanılan TaskExecutor

implementasyonu tayin etmektedir. Örneğin ThreadPoolTaskExecutor kullanarak kod 17.6 daki gibi bir konfigürasyon oluşturabiliriz.

```
Kod 17.6 - task-config.xml

<bean id="taskExecutor"
    class="org.springframework.scheduling.concurrent.
        ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="5" />
    <property name="maxPoolSize" value="10" />
    <property name="queueCapacity" value="25" />
</bean>

<bean id="myTaskExecutor"
        class="com.kurumsaljava.spring.task.MyTaskExecutor">
    <constructor-arg ref="taskExecutor" />
</bean>
```

Kod 17.6 da yer alan taskExecutor içinde en fazla 10 aktif threadin yer alacağı bir thread havuzu oluşturmaktadır. Thread havuzu içinde 5 thread ile start alırken, zaman içinde thread sayısı en fazla 10 olabilir. QueueCapacity bünyesinde işlem için sırada bekleyebilecek görev adedi yer almaktadır. Görevleri tutmak için BlockingQueue tipinde bir kuyruk (Queue) kullanılmaktadır.

Uygulamayı koşturmak için kod 17.8 de yer alan Main sınıfını kullanabiliriz.

```
Kod 17.8 - Main

package com.kurumsaljava.spring.task;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
        ClassPathXmlApplicationContext;

public class Main {

    public static void main(final String[] args) {
        System.out.println("starting...");
        final ApplicationContext ctx =
                new ClassPathXmlApplicationContext(
            "task-config.xml");
        final MyTaskExecutor bean =
            (MyTaskExecutor) ctx.getBean(
                "myTaskExecutor");
```

```
        bean.printMessages();

        System.out.println("ending...");
    }

}
```

Kod 17.8 de yer alan Main sınıfının ekran çıktısına baktığımızda, Main.main() metodunu koşturan threadin 8. satırda son bulduğunu, havuzda yer alan diğer threadlerin paralel olarak görevleri işlediğini görmekteyiz. MyTaskExecutor.printMessages() metodunda mesajlar sırayla gönderilmiş olmalarına rağmen, threadler paralel koştuğundan, ekran çıktısında yer alan mesaj sıralaması bu paralelliği göstermektedir.

```
Kod 17.8 - Ekran çıktısı

starting...
Message0
Message1
Message3
Message5
Message6
Message7
ending...
Message9
Message11
Message8
Message12
Message2
Message4
Message10
Message16
Message15
Message14
Message13
Message20
Message19
Message18
Message17
Message24
Message23
Message22
Message21
```

TaskScheduler Kullanımı

TaskScheduler belli zaman birimlerinde görev koşturmak için oluşturulmuş, Spring 3.0 ile kullanıma sunulan bir interface sınıftır. İhtiva ettiği metotlar kod 17.9 da yer almaktadır.

```
Kod 17.9 - TaskScheduler

public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task,
        Trigger trigger);
    ScheduledFuture schedule(Runnable task,
        Date startTime);
    ScheduledFuture scheduleAtFixedRate(Runnable task,
        Date startTime, long period);
    ScheduledFuture scheduleAtFixedRate(Runnable task,
        long period);
    ScheduledFuture scheduleWithFixedDelay(Runnable task,
            Date startTime, long delay);
    ScheduledFuture scheduleWithFixedDelay(Runnable task,
            long delay);
}

ThreadPoolTaskScheduler scheduler =
        new ThreadPoolTaskScheduler();
// ...
scheduler.scheduleWithFixedDelay(task, 5000);
scheudler.shutdown();
```

Kod 17.9 da yer alan ThreadPoolTaskScheduler bir TaskScheduler implementasyonudur. scheduleWithFixedDelay() metodu ile görev 5 saniye (5000 ms) beklettikten sonra koşturulur.

Trigger Kullanımı

Tanımlı bir görevin (task) belli aralıklarla koşturulma işlemi tetiklemek için Trigger sınıfı kullanılır. CronTrigger ve PeriodicTrigger isminde implementasyonları bulunmaktadır.

```
Kod 17.9 - Trigger

public interface Trigger {
    Date nextExecutionTime(
        TriggerContext triggerContext);
}
```

Görevlerin tekrar edilmesinde görevler arası bilgi paylaşımı ve koordinasyon için TriggerContext kullanılır. Kod 17.11 de yer alan CronTrigger implementasyonu task ismini taşıyan görevi hafta içi her gün 9-17 saatleri arası her 5 saniyede bir koşturmaktadır.

```
Kod 17.10 - CronTrigger

ThreadPoolTaskScheduler scheduler =
        new ThreadPoolTaskScheduler();
// ...
scheduler.schedule(task,
        new CronTrigger("*/5 * 9-17 * * MON-FRI"));
scheudler.shutdown();
```

Task İsim Alanı

Spring 3.0 ile TaskExecutor ve TaskScheduler nesnelerini konfigüre etmek ve kullanımlarını kolaylaştırmak için task ismini taşıyan yeni bir isim alanı oluşturulmuştur.

task:executor

task:executor elementi bir ThreadPoolTaskExecutor oluşturmak için kullanılmaktadır. Kod 17.6 de yer alan ThreadPoolTaskExecutor tanımlamasını task:executor elementi ile kod 17.11 de ki gibi yapabiliriz.

```
Kod 17.11 - task-config.xml

<task:executor id="taskExecutor" pool-size="10"/>

<bean id="myTaskExecutor"
        class="com.kurumsaljava.spring.task.
        MyTaskExecutor">
    <constructor-arg ref="taskExecutor" />
</bean>
```

task:scheduler

Bir ThreadPoolTaskScheduler nesnesini bünyesinde 10 thread ihtiva eden bir thread havuzu ile şu başka oluşturabiliriz:

```
<task:scheduler id="scheduler" pool-size="10"/>
```

pool-size ile havuzdaki thread adedi belirlenmediğinde, havuzda sadece 1 thread yer alır.

task:scheduled-tasks

task isim alanında yer alan scheduled-tasks elementi ile bir Spring nesnesinin sahip olduğu bir metot belli aralıklarla koşturulabilir. Kod 17.12 de yer alan scheduled-tasks konfigürasyonu ile rentalDownloader nesnesinin download metodu koşturulmaktadır. Fixed-delay element özelliği ile bir önceki görevin tamamlanmasından sonra yeni görevin başlatılması için geçmesi gereken süre tanımlanmaktadır.

```
Kod 17.12 - task-config.xml

<task:scheduled-tasks scheduler="scheduler">
    <task:scheduled ref="rentalDownloader"
        method="download"
        fixed-delay="1000"/>
</task:scheduled-tasks>
```

Aşağıda yer alan task:scheduled konfigürasyonunda kullanılan fixed-rate ile bir önceki görevin son bulması beklenmeden yeni bir görev başlatılır. Bu her beş saniyede bir yeni görevin başlatılması anlamına gelmektedir. Fixed-delay ve fixed-rate initial-delay ile zenginleştirilerek. görevin ilk koşturulmasından sonra beklenecek süre tayin edilebilir.

```
<task:scheduled ref="rentalDownloader"
    method="download"
        fixed-rate="5000"/>
```

Kod 17.10 da yer alan CronTrigger yapısını cron element özelliğini kullanarak şu şekilde tanımlayabiliriz:

```
<task:scheduled ref="rentalDownloader"
    method="download"
        cron="*/5 * 9-17 * * MON-FRI"/>
```

Anotasyon Bazlı Task Konfigürasyonu

Kambersiz düğün, anotasyonsuz konfigürasyon olmaz diyerek, bu bölümde paralel görev koşturma işlemleri için kullanılabilecek anotasyonları tanıtmak

istiyorum.

@Scheduled

@Scheduled atandığı metodun tanımlı aralıklarla koşturulması gerektiğini ifade eder. Atandığı metodun metot parametresiz olması gerekmektedir.

```
Kod 17.13 - RentalDownloader

@Component
public class RentalDownloader {

    @Scheduled(fixedRate = 5000)
    public void download() {
        System.out.println("downloding now.... "
            + System.currentTimeMillis());
    }
}
```

@Scheduled anotasyonunun çalışabilmesi için konfigürasyon dosyasında kod 17.14 de yer alan elementlerin yer alması gerekmektedir.

```
Kod 17.14 - task-config.xml

<context:component-scan base-package="com.kurumsaljava" />

<task:annotation-driven scheduler="scheduler"/>

<task:scheduler id="scheduler" pool-size="10" />
```

@Scheduled kod 17.15 de görüldüğü gibi cron tanımlamaları yapmak için de kullanılabilir. Kod 17.15 de yer alan download() metodu her 5 saniyede bir koşturulur.

```
Kod 17.15 - RentalDownloader

@Scheduled(cron = "*/5 * * * * ?")
public void download() {
    System.out.println("downloding now.... " +
        System.currentTimeMillis());
}
```

@Scheduled bünyesindeki kullanılacak cron tanımlamasını bir dosyadan edinmek mümkündür. Kod 17.16 da yer alan cron.expression bir yer tutucudur (placeholder) ve sahip olduğu değer kod 17.17 de yer alan application.properties

dosyasından gelmektedir. Bu dosya içinde yer alan değerler context:property-placeholder ile uygulama tarafından erişilebilir hale gelmektedir.

```
Kod 17.16 - RentalDownloader

@Scheduled(cron = "${cron.expression}")
public void download() {
    System.out.println("downloding now.... " +
        System.currentTimeMillis());
}

Kod 17.17 - task-config.xml

<util:properties id="applicationProps"
        location="application.properties" />
<context:property-placeholder
        properties-ref="applicationProps" />
```

@Async

Bu anotasyonu taşıyan metotlar asenkron olarak koşturulur. Bu yüzden bu metotlar bloke olmazlar ve hemen geri dönerler, çünkü metot Spring TaskExecutor bünyesinde ayrı bir thread içinde koşturulur. Geri dönüş değerinin void ya da Future tipinde olması gerekmektedir.

```
Kod 17.18 - RentalDownloader

@Async
public void download() {
    System.out.println("downloding now.... " +
    System.currentTimeMillis());
}
```

@Async ile işaretli metodun bir değer vermesi durumunda, bu değerin Future tipinde olması gerekmektedir. Kullanıcı sınıf bloke olmadan başka işlemleri yaptıktan sonra @Async ile işaretli metodun yaptığı işlem sonucu programın ilerleyen akışında Future.get() ile edinebilir.

```
Kod 17.18 - RentalDownloader

@Async
public Future<File> download() {
    System.out.println("downloding now.... " +
        System.currentTimeMillis());
    // ....
```

```
    return new AsyncResult(file);
}
```

Uygulama Sunucu Entegrasyonu

Özellikle Java EE uygulamaları için kullanılan uygulama sunucularında thread yönetimi uygulama sunucusu (container managed) tarafından yapılmaktadır. Bu yönetime thread havuzları oluşturarak müdahale etmek istenmedik sonuçlar doğurabilir. Bu yüzden CommonJ gibi bir API üzerinden paralel işlem yapmak isteyen uygulama modülleri için threadler uygulama sunucusundan talep edilebilir. Bir TaskExecutor implementasyonu olan org.springframework.jca.work.WorkManagerTaskExecutor bu tür işlemler için kullanılabilir.

17. Bölüm Soruları

17.1 Spring'de paralel görev (task) yaptırmak için kullanılan soyut sınıf hangisidir?

17.2 Belli zaman birimlerinde görev koşturmak için kullanılan Spring sınıfı hangisidir?

17.3 Trigger sınıfının görevi nedir?

17.4 Paralel görev tanımlamada kullanılan konfigürasyon elementlerinin bulunduğu isim alanının ismi nedir?

17.5 Hangi konfigürasyon elementi ile belli bir zaman biriminde koşturulan paralel görev tanımlaması yapılır?

17.6 @Scheduled ne için kullanılır?

17.7 @Scheduled ile cron tanımlamaları yapılabilir mi?

17.7 @Async ile işaretli metotlar nasıl koşturulur?

18. Bölüm

Spring E-Mail

Spring bünyesinde e-posta gönderme işlemleri için kullanılan merkezi sınıf org.springframework.mail.MailSender sınıfıdır. Bu interface sınıf org.springframework.mail.SimpleMailMessage tipinde olan e-posta iletileri göndermek için gerekli metotları tanımlamaktadır. Bu sınıfın org.springframework.mail.javamail.JavaMailSender ismini taşıyan bir alt sınıfı bulunmaktadır. JavaMailSender JavaMail API'yi kullanmak üzere tasarlanmış bir interface sınıftır.

Bir JavaMailSender implementasyonu olan org.springframework.mail.javamail.JavaMailSenderImpl JavaMail API'yi kullanarak, e-posta gönderme işlemlerini gerçekleştirmektedir. Bu bölümde Spring MailSender API'yi kullanarak e-posta gönderme işlemlerini bir örnek üzerinde inceleyeceğiz.

Kod 18.1 de uygulamanın e-posta gönderme ihtiyacını karşılayan MailService sınıfı yer almaktadır. Bu sınıfa e-posta iletilerini göndermek üzere bir MailSender nesnesi ve bir e-posta iletisini temsil eden SimpleMailMessage nesnesi enjekte edilmektedir. sendMail() metodu metot parametrelerini kullanarak bir SimpleMailMessage nesnesi oluşturduktan sonra mesajı gönderirken, sendPreConfiguredMail() hazır bir e-posta iletisini (enjekte edilen message nesnesi) ileti olarak gönderilecek içerik ile tamamladıktan sonra e-posta olarak göndermektedir.

```
Kod 18.1 - MailService

package com.kurumsaljava.spring.mail;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.stereotype.Component;

@Component("mailService")
public class MailService {

    @Autowired
    private MailSender mailSender;

    @Autowired
    private SimpleMailMessage message;

    public void sendMail(final String to, final String
                            subject, final String body) {
        final SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(to);
```

```
        message.setSubject(subject);
        message.setText(body);
        this.mailSender.send(message);
    }

    public void sendPreConfiguredMail(final String message) {
        final SimpleMailMessage mailMessage =
                    new SimpleMailMessage(this.message);
        mailMessage.setText(message);
        this.mailSender.send(mailMessage);
    }
}
```

E-posta gönderme işlemi için gerekli konfigürasyon kod 18.2 de yer almaktadır. Gerçek e-posta gönderim işlemini JavaMailSenderImpl sınıfı yapmaktadır. Bu sınıf e-postayı göndermek için JavaMail API'yi kullanmaktaktadır.

```
Kod 18.2 - mail-config.xml

<context:component-scan base-package="com.kurumsaljava" />

<bean id="mailSender"
        class="org.springframework.mail.javamail.
                JavaMailSenderImpl">
    <property name="host" value="smtp.gmail.com" />
    <property name="port" value="25" />
    <property name="username" value="acar@agilementor.com" />
    <property name="password" value="password" />
    <property name="javaMailProperties">
        <props>
            <prop key="mail.transport.protocol">smtp</prop>
            <prop key="mail.smtp.auth">true</prop>
            <prop key="mail.debug">true</prop>
        </props>
    </property>
</bean>

<bean id="preConfiguredMessage"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="to" value="acar@gmail.com"></property>
    <property name="from" value="acar@agilementor.com"/>
    <property name="subject" value="Test the West" />
</bean>
```

mailSender nesnesini konfigüre etmek için kullanılan parametreler şunlardır:

- ***host*** - E-postanın gönderilecek olduğu e-posta sunucusunun adresidir.

- ***port*** - E-posta sunucusunun port adresidir.
- ***username*** - E-posta sunucusuna bağlantı yapmak için kullanılacak kullanıcı hesabını tanımlar.
- ***password*** - E-posta sunucusuna bağlantı yapmak için kullanılacak şifreyi tanımlar.

javaMailProperties e-posta sunucusuna bağlantı yapılırken kullanılacak özellikleri tanımlamak için kullanılmaktadır. Kullanılan özellikler şunlardır:

- ***mail.transport.protocol*** - E-posta sunucusuyla iletişimde kullanılacak protokolün ismini belirler.
- ***mail.smtp.auth*** - True değerini taşıyorsa e-posta sunucusuna bağlantı kurup, işlem yapabilmek için kullanıcı ismi ve şifresi gereklidir.
- ***mail.debug*** - Hata tespitini kolaylaştırmak için e-posta sunucusuyla yapılan iletişimi detaylı olarak loglar.

Kod 18.3 de e-posta göndermek için kullanılan Main sınıfı yer almaktadır. main() metodunda konfigürasyonda tanımlanan mailSender nesnesi edinilmekte ve sendPreConfiguredMail() metodu üzerinden e-posta, alıcısına iletilmek üzere e-posta sunucusuna gönderilmektedir. sendPreConfiguredMail() metot parametresi olarak e-posta iletinin içeriğini almaktadır. Kod 18.2 ye baktığımızda preConfiguredMessage isminde içerik haricinde tüm özellikler ile donatılmış bir SimpleMailMessage nesnesini görmekteyiz. Bu nesne message ismi altında MailService sınıfına enjekte edilmekte ve endPreConfiguredMail() bu nesneye setText() ile içeriği ekleyerek e-posta iletisini sunucuya göndermektedir.

```
Kod 18.3 - Main

package com.kurumsaljava.spring.mail;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
            ClassPathXmlApplicationContext;

public class Main {
    public static void main(final String[] args) {
        final ApplicationContext ctx =
                new ClassPathXmlApplicationContext(
            "mail-config.xml");
        final MailService service = (MailService) ctx.getBean(
            "mailService");
        service.sendPreConfiguredMail("Icerik...");
```

```
    }
}
```

MIME Tipi Mesaj Oluşturma

Multipurpose Internet Mail Extensions (MIME) e-posta formatını tanımlayan RFC 822 standardı genişleten ve ASCII haricinde karakterler kullanmayı ve e-posta iletilerinde multimedya içeriği destekleyen bir tanimlamadır.

MIME tipinde e-posta göndermek için org.springframework.mail.javamail.MimeMessageHelper sınıfından faydalanabiliriz. MIME tipi e-posta iletilerini göndermek için kullanabileceğimiz metotlar JavaMailSender interface sınıfında tanımlıdır. Bu sebepten dolayı kod 18.3.1 de yer alan MailService sınıfındaki mailSender değişkeninin JavaMailSender tipinde olması gerekmektedir. Kod 18.2 de yer alan konfigürasyonla mailSender nesnesine bir JavaMailSender altsınıfı olan JavaMailSenderImpl sınıfından bir nesne enjekte edilmektedir.

```
Kod 18.3.1 - MailService

@Inject
private JavaMailSender mailSender;

public void sendMIMEMail(final String to,
            final String subject) throws MessagingException {

    final MimeMessage message =
        (this.mailSender).createMimeMessage();
    final MimeMessageHelper helper =
            new MimeMessageHelper(message);
    helper.setTo(to);
    helper.setSubject(subject);
    helper.setText("Bir bir icerik.",
        "<html><body><h3>Bu bir HTML icerik </h3></body></html>");
    this.mailSender.send(message);
}
```

Kod 18.3.1 de içeriği hem düz metin, hem de HTML tipinde tanımlayan setText() metodu kullanılmaktadır. Alıcının kullandığı e-posta okuyucusunun (e-mail client) özelliklerine göre içerik ya düz metin, ya da HTML formatlı olarak gösterilir.

Dosya Gönderme

Bir e-posta iletisine herhangi bir dosyayı ekleyerek (attachment) alıcıya bu dosyanın transfer edilmesini sağlayabiliriz. Kod 18.4 de MimeMessageHelper sınıfında yer alan addAttachment() metodu kullanılarak bu işlem yapılmaktadır.

```
Kod 18.4 - MailService

public void sendEMailWithAttachment(final String to,
            final String subject, final String filename)
        throws MessagingException {

    final MimeMessage message =
        (this.mailSender).createMimeMessage();
    final MimeMessageHelper helper =
        new MimeMessageHelper(message, true);
    helper.setTo(to);
    helper.setSubject(subject);
    helper.setText("Bir bir icerik.",
        "<html><body><h3>Bu bir HTML icerik </h3></body></html>");
    final FileSystemResource file =
            new FileSystemResource(new File(filename));
    helper.addAttachment("myimage.jpg", file);

    this.mailSender.send(message);
}
```

İçerik Formatlama

MimeMessageHelper sınıfında yer alan addAttachment() metodu seçilen dosyayı e-posta iletisine doğrudan eklemektedir. Bu dosyanın bir dosya olarak alıcıya iletilmesi anlamına gelmektedir. Ama bazı durumlarda dosyanın iletiye eklenmesini değil de, iletinin gösterimi yapılırken bir parçası olarak eklenmesini isteyebiliriz. Örneğin aklımıza alıcı e-posta iletisini okurken seçtiğimiz resimleri görünmesi ya da iletiyi CSS (Cascading Style Sheet) kullanarak formatlama fikri gelebilir. Bu gibi durumlarda resimleri ya da CSS dosyasını addAttachment() metoduyla e-posta iletisine eklemek, bu dosyaların e-posta iletisi ile transfer edilmesini sağlar, yani istediğimiz formatlama neticesini alamayız. Eğer iletiyi formatlamak için dosya kullanmak istiyorsak, bu dosyaların doğrudan iletinin içine eklenmesini (inline) sağlamamız gerekmektedir.

Kod 18.5 de yer alan addInline() metodu ismi filename değişkeninde yer alan resmi iletiye eklemektedir. Eğer alıcının kullandığı e-posta okuyucusu MIME tipinde mesajları destekliyorsa, iletiyi okurken e-posta iletisine inline yaparak gönderdiğimiz resmi iletinin bir parçası olarak görecektir.

```
Kod 18.5 - MailService

public void sendEMailWithAttachment(final String to,
        final String subject, final String filename)
        throws MessagingException {

    final MimeMessage message =
        (this.mailSender).createMimeMessage();
    final MimeMessageHelper helper =
            new MimeMessageHelper(message);
    helper.setTo(to);
    helper.setSubject(subject);
    helper.setText("Bir bir icerik.",
            "<html><body><h3>Bu bir HTML icerik </h3>
                    </body></html>");
    FileSystemResource res =
            new FileSystemResource(new File(filename));
    helper.addInline("ident1234", res);

    this.mailSender.send(message);
}
```

Şablon Kullanımı

Kod 18.5 de yaptığımız gibi bir Java sınıfı içinde HTML kodu oluşturmak, kodun bakımını zorlaştırmaktadır. HTML kodu üzerinde yapılan her değişiklik kodun yeniden derlenmesini gerektirmektedir. Halbuki HTML kodunu şablonlar kullanarak Java kodunun dışına çekebilir ve bu bağımlılığı ortadan kaldırabiliriz. Bu Java kodunu değiştirmeden, e-posta iletilerin formatlanma şekillerini değiştirebilmemizi mümkün kılacaktır.

Kod 18.6 da Apache Velocity ile oluşturduğum şablon yer almaktadır. Şablon bünyesinde ${customer.name} gibi yer tutucular (place holder) bulunmaktadır. E-posta gönderme işlemi esnasında bu yer tutucular gerçek veri ile yer değiştirerek, e-posta iletisinin oluşmasını sağlarlar.

```
Kod 18.6 - email-sablon.vm
```

```
<html>
    <body>
        <h3>Merhaba ${customer.firstname} ${customer.name},
            </h3>
        <div>
            Adres bilgileriniz <br/>
            ${customer.address} - ${customer.city}
        </div>
    </body>
</html>
```

Kod 18.6 da yer alan Velocity şablonu Java Bean tarzı değişken erişimini kullanmaktadır. ${customer.firstname} aslında customer.getFirstname() anlamına gelmektedir. Kod 18.7 de e-posta iletisinin kod 18.6 da yer alan şablon ile hazırlanış şekli yer almaktadır. velocityEngine değişkeni şablon kullanımını yöneten sınıftır ve MailService sınıfına Spring tarafından enjekte edilmektedir. Şablon içindeki yer tutucuların gerçek verilerle yer değiştirebilmesi için, bu verilerin bir Map içine yerleştirilmesi gerekmektedir. Bu amaçla sendMailWithTemplate() metoduna parametre olarak verilen customer nesnesi model ismini taşıyan bir HashMap nesnesine yerleştirilmektedir. Akabinde model, velocityEngine ve kullanılan şablon VelocityEngineUtils sınıfının mergeTemplateIntoString() metoduna parametre olarak verilmektedir. mergeTemplateIntoString() bu değişkenleri kullanarak kod 18.9 de yer alan içeriği oluşturmaktadır.

```
Kod 18.7 - MailService

@Autowired
private VelocityEngine velocityEngine;

public void sendMailWithTemplate(final Customer customer) {

    final MimeMessagePreparator preparator =
        new MimeMessagePreparator() {
        @Override
        public void prepare(final MimeMessage mimeMessage)
            throws Exception {
            final MimeMessageHelper message =
                new MimeMessageHelper(mimeMessage);
            message.setTo(customer.email);
            final Map model = new HashMap();
            model.put("customer", customer);
            final String text = VelocityEngineUtils.
                    mergeTemplateIntoString(
                MailService.this.velocityEngine,
```

```
                    "email-template.vm", "UTF-8", model);
            message.setText(text, true);
        }
    };
    this.mailSender.send(preparator);
}
```

Kod 18.8 de gerekli olan Spring konfigürasyonu yer almaktadır. Kod 18.7 de yer alan velocityEngine değişkenine enjekte edilen nesne org.springframework.ui.velocity.VelocityEngineFactoryBean tipinde bir nesnedir. Bu sınıf bünyesinde VelocityEngine tipinde bir değişken yer almaktadır. Kod 18.8 de kullanılan velocityProperties velocityEngine nesnesinin çalışma tarzını yönetmek için gerekli özellikleri tanımlamakta kullanılmaktadır. resource.loader şablonların hangi yöntemle yükleneceğini tanımlanmaktadır. Kod 18.8 de yer alan örnekte şablonların classpath içinde olmaları gerektiği ve ClasspathResourceLoader kullanılarak yükleneceği tanımlaması yapılmaktadır.

```
Kod 18.8 - mail-config.xml

<bean id="velocityEngine"
    class="org.springframework.ui.velocity.
                VelocityEngineFactoryBean">
    <property name="velocityProperties">
        <value>
            resource.loader=class
            class.resource.loader.class=org.apache.velocity.
                runtime.resource.loader.ClasspathResourceLoader
        </value>
    </property>
</bean>

Kod 18.9 - e-posta icerigi

<html>
    <body>
        <h3>Merhaba Ahmet Yildirim,
            </h3>
        <div>
            Adres bilgileriniz <br/>
            Yildirim sokak, no 13 - 4. Levent / Istanbul
        </div>
    </body>
</html>
```

18. Bölüm Soruları

18.1 Spring çatısında e-posta gönderme işlemlerini temsil eden merkezi interface sınıf hangisidir?

18.2 Spring hangi standart Java API'yi kullanarak e-posta gönderme işlemlerini yapmaktadır?

18.3 E-posta içeriği HTML kullanılarak şekillendirilebilir mi?

18.4 Bir dosyayı e-posta ile göndermek için kullanılan Spring sınıfı hangisidir?

18.5 E-posta içeriğini oluştururken şablon kullanmanın faydası nedir?

Cevaplar

1. Bölüm Cevapları

1.1 Spring'in yükselişini önlemek için J2EE nasıl bir yapıda olmalıydı?

Bugün kullandığımız JEE 6 yapısında olması gerekirdi. JEE 6 ile POJO bazlı sınıflar ve bu sınıfların standart Java anotasyonları kullanarak konfigürasyonu mümkün olmuştur. JEE uygulamalarını sunucu dışında, sunucu olmadan test etmek mümkündür. Böylece J2EE den tanıdığımız sunucuya bağımlı olma zorunluluğu ortadan kalmış ve JEE 6 ile yazılım geliştirme süreci hem kısaltılmış hem de yazılımcı için daha eğlenceli hale gelmiştir.

1.2 Spring'in ana amacı nedir sorusunu tek cümle ile nasıl cevaplarsınız?

Spring Java ile yapılan yazılımı basitleştirmek ve programcının hayatını kolaylaştırmak için vardır.

1.3 Spring ilk etapta hangi yazılım prensibine dayanmaktadır?

Bu inversion of control (IoC) yani kontrolün tersine çevrilme prensibidir. Uygulama kendi bağımlılıklarını yönetmek yerine, bu görevi uygulama harici olan bir uygulama çatısı yapmaktadır.

1.4 Dependy injektion nedir?

Bir sınıfın ihtiyaç duyduğu bir bağımlılığın bir sınıf metodu üzerinden sınıfa verilmesine, dependency injection, yani bağımlılığın enjekte edilmesi ismi verilmektedir.

1.5 Yazılımda nihayi amaç bağımlılıkları yok etmek midir?

Hayır! Eğer öyle olsaydı sınıflar arasında hiçbir ilişki kurulamaz ve sınıflar mantıklı bir şeyler yapamazlardı. Yazılımda ana amaç, mevcut bağımlılıkları kontrol altında tutabilmek, yani onları yönetebilme yetisi geliştirmektir. Spring çatısı bir sınıfın ihtiyaç duyduğu tüm bağımlılıkları yöneterek, bu sınıfın sadece iş mantığına yoğunlaşmasını sağlamaktadır.

1.6 Bağımlıkları kontrol edilebilir hale getirmek için kullanılan tasarım prensibi hangisidir?

DIP (Dependency Inversion Principle)

1.7 Spring'i oluşturan temel modüller hangileridir?

Core, Context, Beans ve Expression Language modülleri.

1.8 Spring modülleri ile Spring uygulamaları arasındaki fark nedir?

Spring modülleri Spring'in iç organlarını oluştururken, Spring uygulamaları Spring modülleri üzerine inşa edilmiş, belli bir alanda kullanılan uygulamalardır.

1.9 İlk hangi sürüm ile Spring anotasyonları kullanmaya başlamıştır?

Spring 2.5

1.10 Java bazlı Spring konfigürasyon hangi sürüm ile gelmiştir?

Spring 3.0

2. Bölüm Cevapları

2.1 Alan modeli oluşturmanın ana amacı nedir?

Alan modeli iş sahasında kullanılan öğelerin birbirleriyle olan ilişkilerini gösterir. Bu öğeler iş sahasına has terminolojide kullanılan isimlerdir (substantive).

2.2 Bir Spring nesnesi tanımlaması yaparken, bu nesneye birden fazla isim nasıl atanır?

Bean elementinin name ismindeki element özelliği kullanılarak bir Spring nesnesine birden fazla isim atanabilir.

```
<bean id="clio" name="myclio, yourclio" .../>
```

2.3 Spring bağımlılıkları enjekte etmek için kaç yöntem tanımaktadır?

Spring bağımlılıkları sınıf konstrüktörü ya da set() metotları aracılığı ile enjekte edebilir.

2.4 Metot enjeksiyonu ne amaçla kullanılır?

Metot enjeksiyonu ilk etapta singleton alan Spring nesnelerine prototype Spring nesneleri enjekte etmek için kullanılır. Bunun yanı sıra bir sınıfın herhangi bir metodunu yeniden yapılandırmak ve sınıfa yeni bir davranış biçimi kazandırmak için kullanılır.

2.5 Bir değişkene null değerini nasıl atanır?

null elementi kullanılarak bir değişkene null değeri atanabilir. Bunun yanı sıra @Required anotasyonunu taşımayan değişkenlere Spring tarafından bağımlılık enjekte edilmediği taktirde taşıyacakları değer null olacaktır.

```
<property name="database"><null/></property>
```

2.6 IdRef elementi neden kullanılır?

idRef elementi Application Context bünyesinde var olmak zorunda olan bir Spring nesnesinin ismine işaret eder. Bu ismi idRef ile herhangi bir değişkene enjekte ettiğimizde, Spring otomatik olarak Application Context içinde böyle bir nesne olup, olmadığını kontrol eder. Böylece sadece mevcut olan Spring nesnelerin isimlerini uygulama bünyesinde kullanmış oluruz.

```
<property name="targetName">
    <idref local="customerRepository" />
</property>
```

2.7 Tekil (singleton) nesneler Spring bean olarak tanımlanabilir mi?

Tekil bir nesne için Spring bean tanımlaması yapılırken factory-method elementi kullanılarak tekil nesneye erişim tanımlanabilir. Spring için bir fabrika metodu olan bu static metot, tekil nesneyi edinmek için kullanılan metottur.

```
<bean id="dbSingleton"
    class="com.kurumsaljava.spring.DBSingleton"
        factory-method="getInstance" />
```

2.8 Hangi Spring sınıfı kullanılarak bir sınıfın kendi istediği usulde nesne oluşturması sağlanabilir?

FactoryBean sınıfını implemente eden sınıflar, nesne oluşturma süreçlerini implemente ettikleri FactoryBean metotları ile kendileri tayin ederler.

2.9 Sirküler bağımlılık nedir?

İki nesne karşılıklı olarak birbirlerine enjekte ediliyorsa, bu iki nesne arasında sirküler bağımlılık mevcuttur. Spring sirküler bağımlılıkların sınıf konstrüktörleri aracılığı ile enjekte edilmesine izin vermez. Bu sorunu çözmek için set() metotları üzerinden bağımlılıklar enjekte edilebilir.

3. Bölüm Cevapları

3.1 Herhangi bir isim alanında bulunan bir Spring konfigürasyon elementi konfigürasyon dosyasında nasıl kullanılır?

Örneğin jee isim alanında bulunan jndi-lookup konfigürasyon elementini kullanmak istediğimizi düşünelim. Bu elementi kullanabilmek için ait olduğu isim alanının

```
xmlns:jee="http://www.springframework.org/schema/jee"
```

şeklinde tanımlanması gerekmektedir. Bu işlem ardından XML dosyası içinde jndi-lookup elementini aşağıdaki şekilde kullanabiliriz.

```
<jee:jndi-lookup id="dataSource"
            jndi-name="jdbc/MyDataSource" />
```

3.2 Spring uygulamalarında yaşam döngüsü kaç fazdan oluşur?

Spring uygulamalarında yaşam döngüsü üç fazdan oluşur. Bunlar:

- Kurulum fazı
- Kullanım fazı
- İmha fazı

3.3 @PostConstruct anotasyonu ne amaçla kullanılır?

Spring'in 2.5 sürümü ile nesne yapılandırma işlemi için init-method yerine @PostConstruct anotasyonu kullanılabilir. @PostConstruct JSR 250 ile tanımlanmış standart bir Java anotasyonudur. @PostConstruct kullanımı Spring çatışına olan kod bağımlılığını tamamen ortadan kaldırır.

3.4 Vekil nesne ne amaçla kullanılır?

Bir nesnenin sahip olduğu bir metodu koşturmadan önce örneğin ön işlem yapmak amacıyla vekil nesne kullanılabilir. Vekil nesne, vekil olduğu nesnenin yerine geçerek, ona gelen istekleri cevaplar. Bu esnada vekil olduğu nesnenin metotlarını koşturmadan önce istenilen türde metot öncesi ya da metot sonrası işlemler yapabilir.

3.5 Bir Spring uygulaması son bulurken bir tekil nesne elinde tuttuğu kaynakları bırakmaya nasıl zorlanır?

Kaynak bırakma (release) işlemi bir metot bünyesinde yapılabilir. Bu metodu kaynak bırakma metodu olarak atamak için destroy-method konfigürasyon elementi ya da @PreDestroy anotasyonu kullanılabilir.

3.6 İmha fazı nasıl başlatılır?

```
applicationContext.close();
```

4. Bölüm Cevapları

4.1 Spring nesne tanımlamalarında kalıtımı sağlamak için kullanılan konfigürasyon elementi hangisidir?

Parent konfigürasyon elementidir.

```
 <bean id="man"
       class="com.kurumsaljava.spring.ManBus" parent="car"/>
```

4.2 Spring nesne tanımlamalarında kalıtımın sağladığı avantajlar nelerdir?

Bir nesne tanımlamasını sadece bir kere yapılarak, bu tanımlama bünyesinde kullanılan özelliklerin parent konfigürasyon elementi aracılığı ile diğer nesne tanımlamalarında kullanılmaları sağlanabilir.

4.3 Bir Spring nesne tanımlamasını soyutlaştıran konfigürasyon elementi hangisidir?

Abstract konfigürasyon elementidir.

```
<bean id="car"
       class="com.kurumsaljava.spring.Car" abstract="true">
```

4.4 Spring soyut Spring nesne tanımlamalarından neden nesne oluşturamaz?

Soyut Spring nesneleri soyut Java sınıfları ile eş değerdedir. Soyut Java sınıflarından nesne oluşturulamaz.

4.5 Dahili bean tanımlamaları ne zaman kullanılır?

Diğer nesnelere ref konfigürasyon elementi üzerinden enjekte edilme gerekliliği olmayan nesneler dahili bean olarak tanımlanabilir. Bu tür bean

tanımlamaların isimleri yoktur ve anonimdirler.

4.6 Konfigürasyon dosyalarının hacimsel büyümelerini önlemek için hangi yöntem kullanılır?

Uygulama mantıksal guruplara bölünerek, her gurup için bir Spring XML dosyası oluşturulabilir. Bu parçaları daha sonra import elementi ile bir araya getirip, sadece bir konfigürasyon dosyası varmış gibi kullanmak mümkündür.

4.7 Property ve ref elementlerinin daha kısa yazılmasını sağlayan isim alanının ismi nedir?

P isim alanıdır.

```
<bean id="rentalService"
       class="com.kurumsaljava.spring.RentalServiceImpl"
   p:customerRepository-ref="customerRepository"
   p:rentalRepository-ref="rentalRepository"
   p:user="admin"/>
```

4.8 Bir e-posta adresinin bağımlılık olarak enjeksiyonu esnasında geçerli olup, olmadığı nasıl kontrol edilebilir?

SpEL'in sunduğu matches operatörü ve regex ifadeleri yardımıyla bir e-posta adresinin geçerliliği kontrol edilebilir.

```
<property name="validEmail"
   value="#{customer.email
       matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.
                                   [a-zA-Z]{2,4}'}"/>
```

4.9 Bir Spring uygulaması XML haricinde hangi yöntem kullanılarak konfigüre edilebilir?

XML dosyası yerine bir Java sınıfı ve anotasyon kullanılarak bir Spring uygulaması konfigüre edilebilir.

4.10 Bir Spring nesnesine isim vermek için kullanılan anotasyon hangisidir

@Qualifier anotasyonu.

4.11 @Autowired anotasyonu yerine hangi standart Java anotasyonu kullanılabilir?

JSR 330 ile gelen @Inject anotasyonudur.

5. Bölüm Cevapları

5.1 Hafızada çalışan bir veri tabanı oluşturmak için kullanılan Spring konfigürasyon elementi hangisidir?

jdbc isim alanında bulunan embedded-database.

5.2 JDBC işlemlerini yapmak için kullanılan Spring sınıfı hangisidir?

org.springframework.jdbc.core.JdbcTemplate

5.3 Neden JDBC yerine JdbcTemplate kullanımı tercih edilmelidir?

JDBC API oluşturulurken ne yazık ki fazlasıyla kontrol edilen (checked exception) hata türleri kullanılmıştır. Bu gereğinden fazla kodun yazılmasına neden olmakta ve kodun bakımını zorlaştırmaktadır. JdbcTemplate kontrol edilmeyen (unchecked exception) hata türlerini kullandığı için, oluşan kod daha sade yapıdadır.

5.4 Bir veri tabanı tablosunda yer alan bir satır (record) alan nesnesine nasıl dönüştürülür?

RowMapper bir veri tabanı tablosunda yer alan bir satırı birebir bir alan nesnesine (Domain Object) dönüştürmek için kullanılır.

5.5 DAO tasarım şablonu JdbcTemplate kullanılarak nasıl uygulanır?

JdbcDaoSupport sınıfını genişleten sınıflar oluşturularak DAO tasarım şablonu uygulanabilir.

5.6 SingleConnectionDataSource kullanımında nasıl bir sorunla karşılaşabiliriz?

SingleConnectionDataSource sadece bir Connection nesnesinin kullanımına izin verir. Aynı anda birden fazla kullanıcısı olan uygulamalarda bu sadece bir kullanıcının veri tabanı işlemlerini yapabileceği anlamına gelmektedir. Yeni bir Connection nesnesi oluşturulamadığından, diğer kullanıcılar beklemek zorunda kalırlar.

6. Bölüm Cevapları

6.1 Veri tabanı işlemlerinde verilerin bütünlüğünü korumak için hangi mekanizma kullanılır?

Veri bütünlüğünü korumak için veri tabanı sistemlerinin sunduğu transaksiyon mekanizması kullanılır. Uygulamaya programsal ya da deklaratif olarak transaksiyon yönetimi desteğinin sağlanması gerekmektedir.

6.2 Commit yapılmadığı taktirde veriler üzerinde yapılan işlemin neticesi ne olur?

Rollback komutuyla yapılan tüm işlemler geri alınır.

6.3 Spring ile hangi türde transaksiyon yönetimi yapılabilir?

Spring transaksiyon template sınıfları ile programsal (programmatic), XML konfigürasyonu ve anotasyon yardımı ile deklaratif (declarative) transaksiyon yönetimi desteği sağlamaktadır.

6.4 Spring deklaratif transaksiyon yönetimi için hangi yöntemleri kullanmaktadır?

Spring hem anotasyon tabanlı hem de XML tabanlı deklaratif transaksiyon yönetimi konfigürasyonunu desteklemektedir.

6.5 Programsal transaksiyon yönetimi için kullanılan Spring sınıfı hangisidir?

org.springframework.transaction.support.TransactionTemplate

6.6 Birden fazla kaynak üzerinde veri güncelleme işlemini koordine etmek için hangi transaksiyon türü kullanılır?

Birden fazla kaynak üzerinde veri güncelleme işlemini koordine etmek için dağıtık (distributed) transaksiyonlar kullanılır.

6.7 JTA bazlı transaksiyon yönetici hangi konfigürasyon elementi ile aktif hale getirilir?

```
<tx:jta-transaction-manager/>
```

6.8 Spring bünyesinde bir JTA implementasyonu mevcut mudur?

Spring bünyesinde bir JTA implementasyonu bulunmamaktadır. Bu sebepten dolayı bir Spring uygulaması bünyesinde global transaksiyonlar oluşturulmak

istendiğinde Atomikos gibi bir JTA implementasyonunun kullanılması gerekmektedir. Spring bünyesinde yer alan JtaTransactionManager sınıfı aracılığı ile JTA implementasyonu uygulamaya entegre edilir.

7. Bölüm Cevapları

7.1 Hibernate gibi bir ORM aracının kullanımının yazılımcıya sağladığı en büyük avantaj nedir?

Yalın JDBC kodu ile Hibernate kodu karşılaştırıldığında, Hibernate'in veri tabanı işlemlerinden ne kadar yüksek seviyede soyutlama yaptığı görülmektedir. Yazılımcı veri tabanı işlemlerinin detayları ile kesinlikle ilgilenmeden uygulama bünyesinde yer alan işletme mantığına konsantre olabilmektedir. Hibernate'in yazılımcıya sağladığı en büyük avantaj budur.

7.2 Hangi Spring sınıfının kullanımı otomatik Hibernate Session ve transaksiyon yönetimi sağlar?

org.springframework.orm.hibernate3.HibernateTemplate

7.3 Bu sınıfın kullanımındaki en büyük dezavantaj nedir?

Kodu Spring çatısına bağımlı kılmasıdır.

7.4 Hibernate uygulamalarında transaksiyonu yönetmek için kullanılan Spring sınıfı hangisidir?

org.springframework.orm.hibernate3.HibernateTransactionManager

8. Bölüm Cevapları

8.1 JPA bir ORM aracı mıdır?

Hayır, değildir. JPA bir standart Java API'sidir. Uygulamayı mevcut ORM araçlarından bağımsız kılmak ve veri tabanı işlemlerini standart hale etmek için oluşturulmuştur.

8.2 JPA API'sinde ana veri tabanı işlemlerini gerçekleştiren yapı hangisidir?

javax.persistence.EntityManager aracılığı ile veri tabanı işlemleri gerçekleştirilir.

8.3 Uygulama sunucusu bünyesindeki EntityManagerFactory nasıl edinilir?

```
<jee:jndi-lookup id="entityManagerFactory"
             jndi-name="persistence/rentACar"/>
```

8.4 Bir sınıfa EntityManager nesnesini enjekte etmek için hangi anotasyon kullanılır?

@PersistenceContext.

8.5 Bir sınıf bünyesinde JPA üzerinden veri tabanı işlemleri yapmak için hangi Spring sınıfı kullanılır?

JpaTemplate ve JpaDaoSupport.

8.6 JPA işlemlerinde oluşan hataları org.springframework.dao.DataAccessException tipine dönüştüren mekanizmalarından birisi hangisidir?

@Repository anotasyonu oluşan JPA hatalarının, örneğin NoResultException otomatik olarak Spring tarafından org.springframework.dao.DataAccessException ve türevlerine dönüştürülmesini sağlamaktadır.

9. Bölüm Cevapları

9.1 İşletme mantığı haricinde uygulamanın genelinde aynı işlevi yerine getirmek için kullanılan kod birimlerine ne ad verilmektedir?

Bu tür kod birimlerine cross cutting concern ismi verilmektedir.

9.2 Aspekt nedir?

Transaksiyon yönetimi, loglama, caching, performans ölçümü ve hata yönetimi gibi iş mantığıyla doğrudan ilişkili olmayan kod birimlerine AOP dilinde aspekt ismi verilmektedir.

9.3 AOP'de kullanılan harmanlama yöntemleri nelerdir?

Üç değişik harmanlama yöntemi mevcuttur:

- Derleme anında (compile)
- Sınıf yükleme anında (classloading)

- Uygulamanın çalışması esnasında (runtime)

9.4 Spring hangi AOP türünü kullanmaktadır?

Spring Proxy bazlı dinamik AOP kullanmaktadır.

9.5 Spring AOP'yi aktif hale getiren konfigürasyon elementi hangisidir?

aop:aspectj-autoproxy

9.6 Spring hangi join point türünü desteklemektedir?

Spring sadece metot koşturma joint point'lerini desteklemektedir.

9.7 Advice sırası nasıl tanımlanabilir?

Öncelik sırası belirlenmediği sürece aspektlerin koşturulma sıralarında bir düzen yoktur. Sıra düzenini oluşturmak için Spring'in ihtiva ettiği org.springframework.core.annotation.Order anotasyonu kullanılabilir.

10. Bölüm Cevapları

10.1 Spring MVC'nin dayandığı standart Java API hangisidir?

Java Servlet

10.2 MVC'yi oluşturan harflerin karşılığı nedir?

MVC bir tasarım şablonudur ve harfler Model, View, Controller anlamına gelmektedir.

10.3 Handler Mapping ne amaçla kullanılır?

Handler mapping yapıları kullanıcının isteğini taşıyan web adresi ile bu isteği işleyecek olan controller sınıfları arasındaki bağı oluşturmak için kullanılır.

10.4 Spring MVC üç katmanlı mimaride hangi katmanı oluşturmak için kullanılır?

Spring MVC gösterim katmanını oluşturmak için kullanılan web uygulama çatısıdır.

10.5 Controller ile view arasında veri taşımak için kullanılan yapı hangisidir?

ModelMap conroller ile view arasında veri taşımak için kullanılan bir yapıdır.

10.6 Gösterimi yapan view elementlerini lokalize etmek için kullanılan yapının ismi nedir?

Gösterimi yapan view elementlerinin MVC çatısı tarafından lokalize edilerek model sınıflarında yer alan bilgilerin kullanıcıya gösterilmesi (model rendering) gerekmektedir. Bu görevi Spring MVC bünyesinde view resolver sınıfları üstlenmektedir.

10.7 @RequestParam ne amaçla kullanılır?

@RequestParam anotasyonu kullanıcı isteği ile gelen parametrelerin metot parametrelerine atanması için kullanılmaktadır.

11. Bölüm Cevapları

11.1 Bir web uygulamasında güvenlik ihtiyacının doğduğu alanlar hangileridir?

Tipik bir web uygulamasında güvenlik ihtiyacının doğduğu dört alan bulunmaktadır. Bunlar kimlik bilgilerinin doğrulanması (authentication), web sayfalarının güvenliği (web request security), servis katmanındaki sınıfların ve alan nesnelerinin güvenliğidir (domain object security).

11.2 Otomatik login işlemi için kullanılan yöntem hangisidir?

Login işlemi esnasında kullanıcıya "Beni Hatırla" (Remember Me) seçeneği sunularak, bir sonraki login işleminin otomatik olarak yapılması sağlanabilir.

11.3 Metot güvenliği sağlamak için kullanılan Spring anotasyonu hangisidir?

@Secured

11.4 @Secured anotasyonunun işleyiş tarzı nasıldır?

@Secured anotasyonu ile sadece bir metoda giriş kontrol edilebilir. Kullanıcı beklenen rollere sahip değilse, metot koşturulmaz.

11.5 Alan nesnelerini nasıl korunabilir?

Spring Security ile alan nesneleri erişim kontrol listeleri (ACL - Access Control List) aracılığı ile güvenlik çemberine alınabilir.

12. Bölüm Cevapları

12.1 REST uygulamalarında yapılacak işlemler nasıl tanımlanır?

Bir REST uygulamasında yapılabilecek işlemler REST komutları ile tanımlanmaktadır. Bunlar HTTP GET. DELETE, PUT ve POST metotlarına denk gelmektedir.

12.2 İdempotent olan bir REST metotları hangi özelliğe sahiptir?

İdempotent metotlar ilk kullanımda güvenli olmayabilir, yani veriyi değiştirebilirler. Lakin sonraki kullanımlarında aynı neticeyi verirler.

12.3 REST uygulamaları geliştirilirken kullanılan Spring modülü hangisidir?

Spring dünyasında REST uygulamaları geliştirmek için Spring MVC web çatısı kullanılmaktadır.

12.4 Bir REST uygulamasında kullanıcı oluşturduğu veriyi nasıl edinebilir?

POST metodu ile yeni bir kaynak oluşturulduğunda, bu kaynağı temsil eden veriler kullanıcıya doğrudan gönderilmez. Bunun yerine kullanıcıya kaynağın hangi adreste erişilebilir olduğu bilgisi aktarılır. Bunu sağlamak için bir HTTP başlık (header) değişkeni olan Location değişkeni kulllanılır.

12.5 HTTP Statü kodu oluşturmak için kullanılan anotasyon hangisidir?

@ResponseStatus

13. Bölüm Cevapları

13.1 Entegrasyon teknolojilerini kullanırken programcıların karşılaştığı sorunların başında ne gelmektedir?

Her entegrasyon teknolojisi programcıyı kendi sahip olduğu programlama modelini kullanmaya zorlar. Örneğin RMI kullanıldığında servis sınıflarının java.rmi.Remote interface sınıfını genişletmesi gerekir. Bu kodu kullanılan entegrasyon teknolojisine bağımlı kılmaktadır.

13.2 Spring Remoting modülünün sağladığı en belirgin avantaj nedir?

Birden fazla entegrasyon teknolojisini hem sunucu hem de kullanıcı tarafında aynı programlama modeli dahilinde kullanmak mümkündür.

13.3 Spring Remoting HTTP protokolünü kullanan hangi entegrasyon yapısına sahiptir?

HttpInvoker

13.4 Sunucu ve kullanıcı Spring çatısını kullanıyorsa, hangi entegrasyon yöntemi kullanılmalıdır?

Sunucu ve kullanıcı Spring çatısını kullanıyorsa, HttpInvoker çözümü seçilebilir.

14. Bölüm Cevapları

14.1 Web servis terminolojsinde kullanıcı ile sunucu arasındaki sözleşmenin ismi nedir?

Bu sözleşmenin ismi WSDL dir. WSDL (Web Services Definition Language) web servis katmanında kullanılabilecek fonksiyon, veri tipi ve veri alışveriş protokol formatlarını tanımlamak için kullanılan bir meta dildir.

14.2 Web servis uygulamaları geliştirirken Spring hangi yöntemi tercih etmektedir?

Web servis uygulamaları geliştirirken contract first yönetimi en iyi yöntem (best practice) olarak kabul edilmektedir. Mevcut Java sınıfların yola çıkarak ta WSDL dosyaları oluşturmak mümkündür, lakin bu servisin tanımlanış şeklini implementasyon detaylarına bağladığı için servis tanımlaması üzerinde değişiklik yapılmasını zor kılmaktadır. Spring WS contract first yönetimi destekleyen bir web servis çatıdır.

14.3 Bir SOAP mesajı hangi bölümlerden oluşmaktadır?

Bir SOAP mesajı Envelope, Header ve Body bölümlerinden oluşur.

14.4 Request ve response mesajlarını loglamak için kullanılan interseptör hangisidir?

SoapEnvelopLoggingInterceptor

14.5 Web servis kullanıcıları oluşturmak için kullanılan Spring sınıfı

hangisidir?

WebServiceTemplate

14.6 POX nedir?

POX plain old XML için kullanılan terimdir. Spring web servis uygulamaları SOAP yerine POX kullanarak mesaj alışverişinde bulunabilirler.

14.7 Bir web servis uygulamasını test etmek için uygulama sunucusu gerekli midir?

Hayır. MockWebServiceServer kullanılarak web servis uygulaması uygulama sunucusu olmadan test edilebilir.

15. Bölüm Cevapları

15.1 İmplemenetasyonu olmayan bir sınıfı somut bir nesneye nasıl enjekte edebiliriz?

Bunu Mockito ya da EasyMock gibi bir mock çatısı kullanarak yapabiliriz

15.2 Spring 3.2 öncesi Spring MVC uygulamalarını test etmek için kullanılan yapılar hangileridir?

MockHttpServletRequest ve MockHttpServletResponse

15.3 Spring 3.2 gelen ve web uygulamalarını test etmek için kullanılan sınıfın adı nedir?

Spring MVC Test çatısında yer alan MockMvc sınıfıdır.

15.4 Spring Test çatısının sağladığı en büyük avantaj nedir?

Spring uygulamalarını uygulama sunucusu dışında test etmek mümkündür. Uygulama sunucusuna bağımlı olmayan bu testler daha hızlı test neticesi almayı sağlarlar.

15.5 Veri katmanını test etmek için hangi tür testler oluşturulmalıdır?

Veri katmanını test etmek için en uygun test türü entegrasyon testleridir.

16. Bölüm Cevapları

16.1 MBean nedir?

Export edilen Java Bean sınıfına JMX terminolojisinde MBean, yani Managed Bean ismi verilmektedir.

16.2 MBean sunucusu oluşturmak için kullanılan Spring konfigürasyon elementi hangisidir?

mbean-server

16.3 Bu konfigürasyon elementi hangi Spring sınıfını temsil etmektedir?

org.springframework.jmx.support.MBeanServerFactoryBean

16.4 @ManagedResource ne için kullanılır?

Bu anotasyonu taşıyan her sınıf MBean olarak export edilir

16.5 Değişken ve metotların export edilebilmesi için kullanılan anotasyon hangisidir?

@ManagedAttribute

17. Bölüm Cevapları

17.1 Spring'de paralel görev (task) yaptırmak için kullanılan soyut sınıf hangisidir?

org.springframework.core.task.TaskExecutor

17.2 Belli zaman birimlerinde görev koşturmak için kullanılan Spring sınıfı hangisidir?

org.springframework.scheduling.TaskScheduler

17.3 Trigger sınıfının görevi nedir?

Tanımlı bir görevin (task) belli aralıklarla koşturulma işlemi tetiklemek için Trigger sınıfı kullanılır. CronTrigger ve PeriodicTrigger isminde implementasyonları bulunmaktadır.

17.4 Paralel görev tanımlamada kullanılan konfigürasyon elementlerinin bulunduğu isim alanının ismi nedir?

task

17.5 Hangi konfigürasyon elementi ile belli bir zaman biriminde koşturulan paralel görev tanımlaması yapılır?

task:scheduled-tasks

17.6 @Scheduled ne için kullanılır?

@Scheduled atandığı metodun tanımlı aralıklarla koşturulması gerektiğini ifade eder. Atandığı metodun metot parametresiz olması gerekmektedir.

17.7 @Scheduled ile cron tanımlamaları yapılabilir mi?

Evet.

```
@Scheduled(cron = "*/5 * * * * ?")
```

17.7 @Async ile işaretli metotlar nasıl koşturulur?

@Async anotasyonu ile işaretli metotlar koşturulduğunda, koşturucu sınıf bloke olmaz. Bu tarz metot koşturma yöntemine asenkron metot koşturma ismi verilmektedir.

18. Bölüm Cevapları

18.1 Spring çatısında e-posta gönderme işlemlerini temsil eden merkezi interface sınıf hangisidir?

org.springframework.mail.MailSender sınıfıdır.

18.2 Spring hangi standart Java API'yi kullanarak e-posta gönderme işlemlerini yapmaktadır?

JavaMail API.

18.3 E-posta içeriği HTML kullanılarak şekillendirilebilir mi?

Evet. E-posta içeriğinin HTML kullanılarak şekillendirilebilmesi için içeriğin MIME formatında olması gerekmektedir. MIME formatında içerik oluşturmak için org.springframework.mail.javamail.MimeMessageHelper sınıfı kullanılır.

18.4 Bir dosyayı e-posta ile göndermek için kullanılan Spring sınıfı hangisidir?

org.springframework.mail.javamail.MimeMessageHelper sınıfıdır.

18.5 E-posta içeriğini oluştururken şablon kullanmanın faydası nedir?

Şablon kullanımı içeriği şekillendirmek için kullanılan kodun (örneğin HTML) Java kodu dışına çekilmesini ve böylece Java kodunu değiştirmeden e-posta içeriğini yeniden yapılandırmayı mümkün kılmaktadır.

BTSoru.com

BTSoru.com yazılımcıları bir araya getirmek için kurduğum bir soru-cevap platformu. Bu kitap hakkındaki sorularınızı BTSoru.com üzerinden bana yöneltebilirsiniz.

KurumsalJava.com

KurumsalJava.com adresinde blog yazılarım yer almaktadır. Java ve Spring konularındaki yazılarımı buradan takip edebilirsiniz.

KurumsalJava.com'da yer alan yazılarımı bir kitap haline getirdim. PDF formatındaki bu kitabı buradan edinebilirsiniz.

EOF (End Of Fun)

Yolculuğumuzun sonuna geldik. Sizin için burası son durak değil. Spring çatısı hakkında internette birçok faydalı kaynak bulabilirsiniz. Özellikle Spring referans dokümentasyonu aklınıza gelen sorulara cevap bulmanızda faydalı olacaktır.

Spring çatısına hakim olabilmek için bol, bol pratik yapılmalıdır. Kitapla birlikte gelen Maven bazlı Spring projeleri pratik yapmak için kullanabilirsiniz. Bu projeleri temel alıp, kendi projelerinizi geliştirebilirsiniz. Aklınıza takılan soruları BTSoru.com üzerinden benimle ve diğer bilişimci arkadaşlarla paylaşabilirsiniz.

Sağlıkcakla kalın. Her şey gönlünüzce olsun.

Özcan Acar