

Spring Container Kabiliyetleri

2

Proxy Örüntüsü

- Bazı durumlarda **nesnelerin hemen yaratılması** uygun veya mümkün olmayabilir
 - Örneğin request veya session scope bean'ler
- Ya da asıl nesneye erişim gerçekleşmeden önce veya erişimden sonra **ilave bir takım işlemlerin yapılması** söz konusu olabilir
 - Transaction başlatılması/sonlandırılması,
 - Güvenlik kontrolü,
 - Validasyon, caching gibi

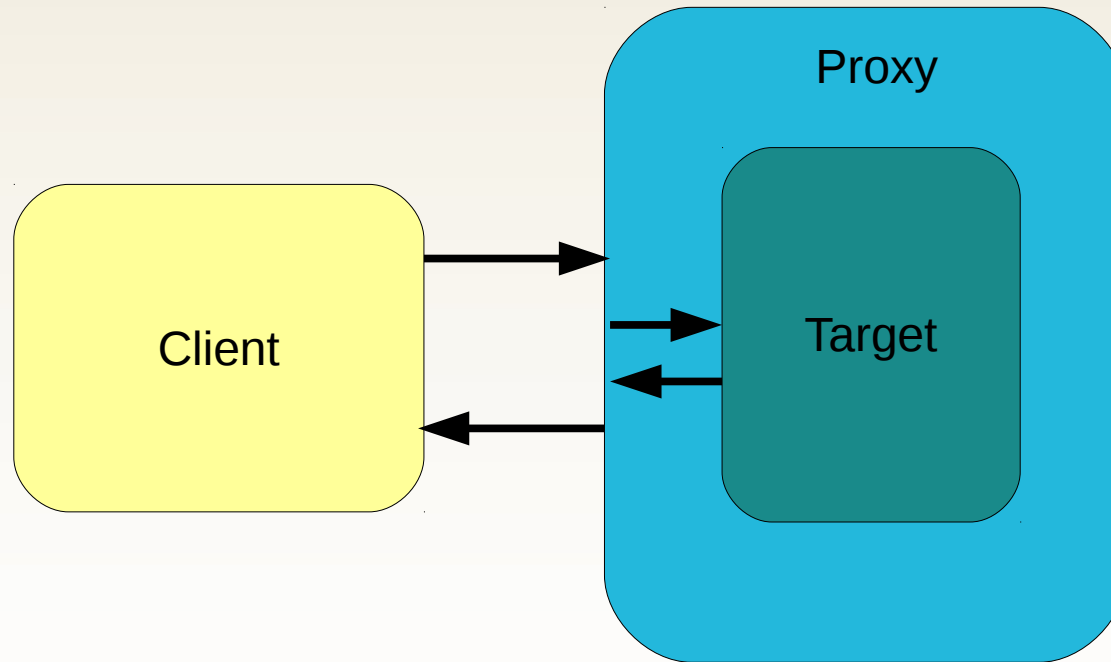
Proxy Örüntüsü

- **Asıl nesnenin yerine geçen**, yaratılmasını ihtiyaç anına veya uygun ortam oluşana kadar erteleyen, onun yerine ilave kabiliyetleri gerçekleştiren **başka bir nesne** yaratılır
- Bunun adı **proxy nesnedir**
- **Asıl nesneye erişim** proxy üzerinden dolaylı biçimde gerçekleşir
- Proxy ile target nesne **aynı türdendir**
- Dolayısı ile **istemciler** proxy'nin varlığından **habersizdir**

Proxy Örüntüsünün Yapısı

Proxy, target nesne ile aynı tipte olup, client ile target nesnenin arasına girer

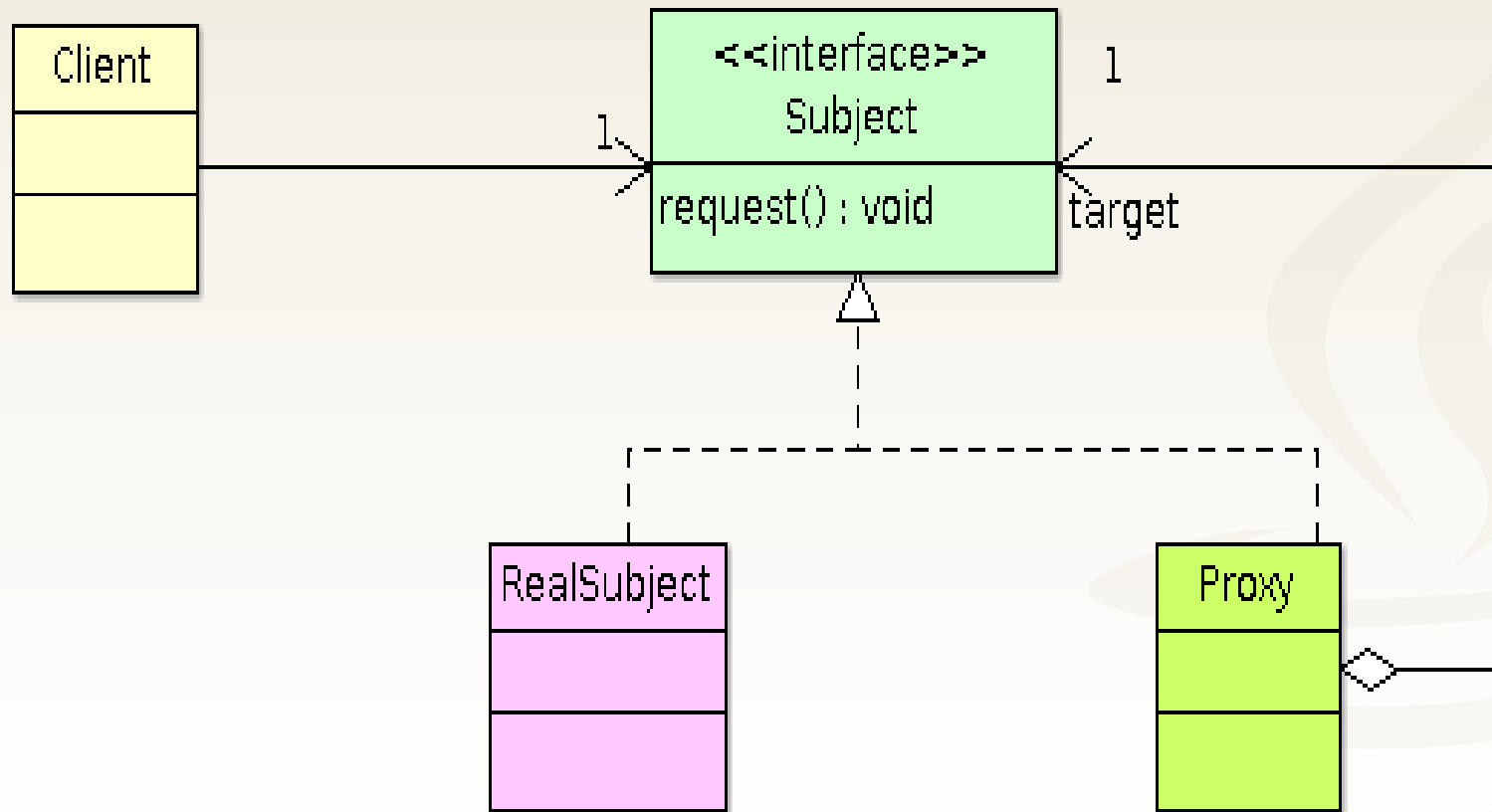
Client proxy nesne ile konuştuğunun farkında değildir



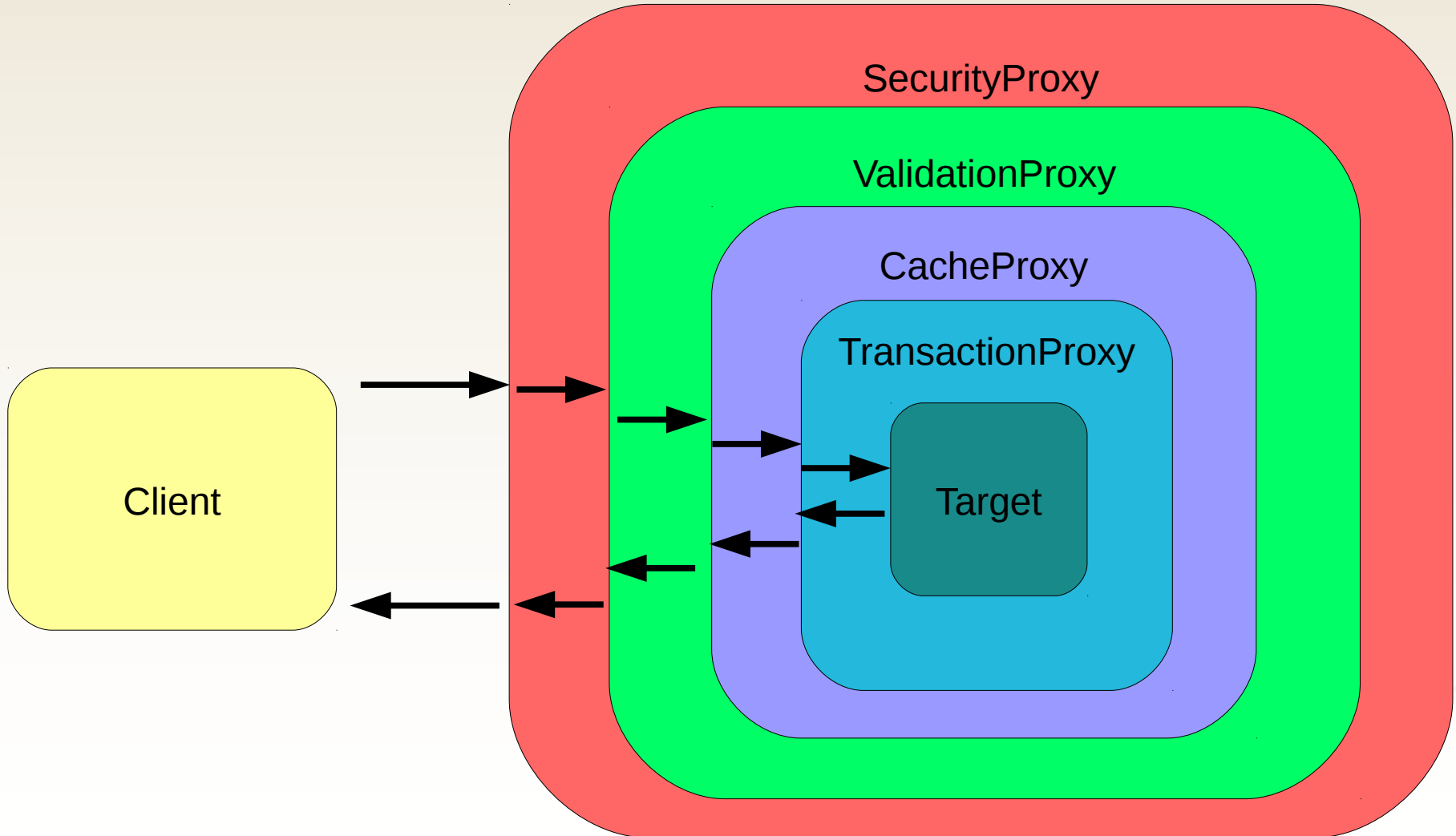
Client'in target nesne üzerindeki metot çağrıları öncelikle proxy nesneye erişir

Proxy metot çağrısından önce veya sonra bir takım işlemler gerçekleştirebilir

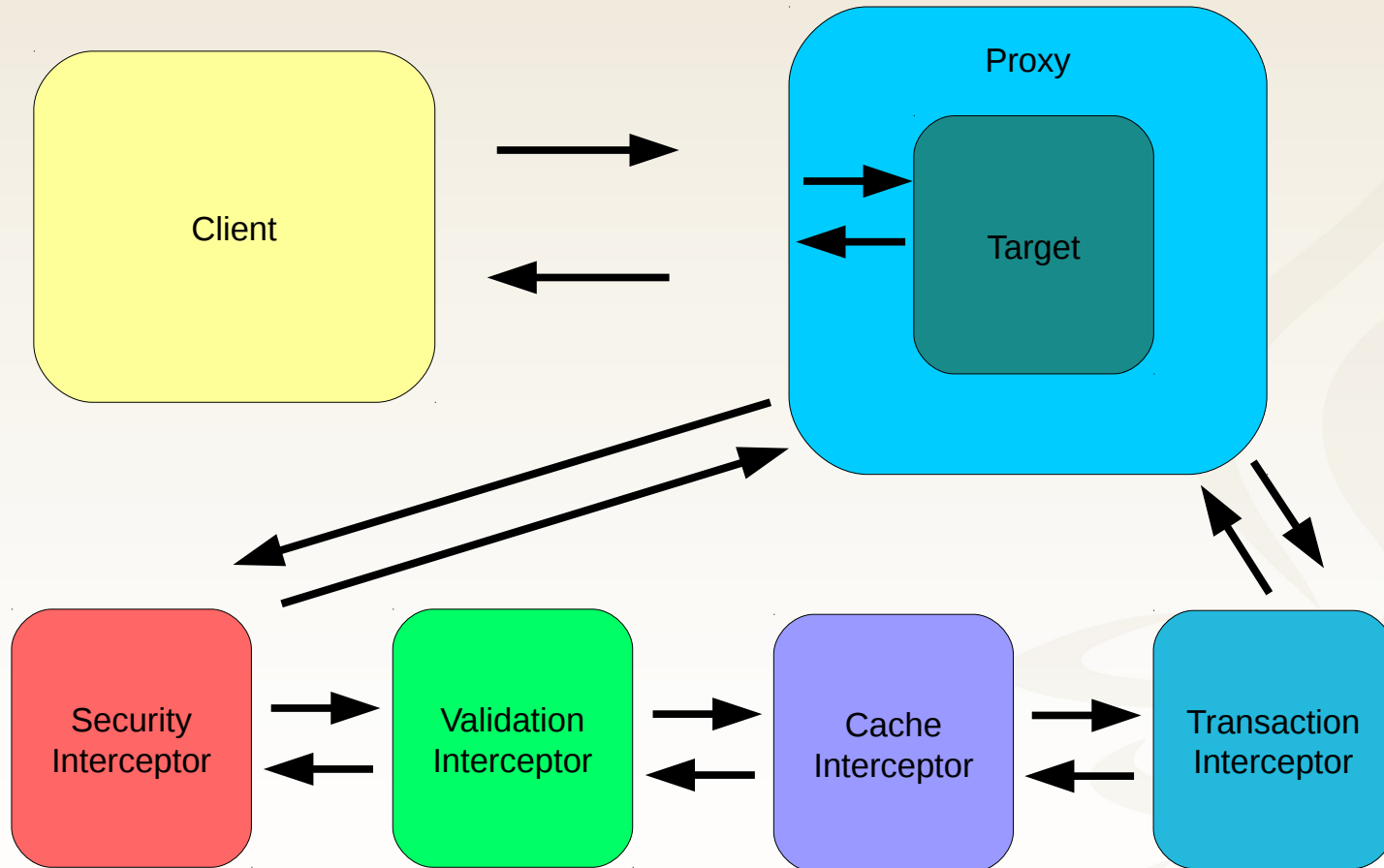
Proxy Örüntüsü Sınıf Diagramı



İç İçe Proxy Nesne Zinciri



Spring Proxy ve MethodInterceptor Zinciri



Spring İçerisinde Proxy Örüntüsünün Kullanımı

- Spring Application Framework'ün **pek çok kabiliyeti** proxy örüntüsü üzerine kuruludur
 - Transaction yönetimi
 - Bean scope kabiliyeti (request ve session scope bean'ler)
 - Aspect oriented programlama altyapısı (Spring AOP)
 - Metot düzeyinde validasyon ve caching
 - Remoting
 - Spring security'de metot düzeyinde yetkilendirme

Spring İçerisinde Proxy Örüntüsünün Kullanımı

- Spring bu kabiliyetleri hayata geçirmek için genellikle uygulama geliştiricilerden habersiz **otomatik olarak proxy oluşturma** işini gerçekleştirir
- Diğer bean'lere de **bağımlılık olarak proxy nesne enjekte** edilir
- Diğer bean'ler proxy ile çalıştıklarının farkında **değillerdir**

Proxy Oluşturma Yöntemleri

- **Interface Proxy**

- Asıl nesnenin sahip olduğu arayüzlerden birisi kullanılır
- JDK proxy olarak da bilinir

- **Class Proxy**

- Asıl nesnenin ait olduğu sınıf extend edilerek gerçekleştirilir
- CGLIB/Javassist proxy olarak da bilinir

ProxyFactoryBean

- Spring'in sunduğu scope, tx yönetimi, caching, validation, aop gibi kabiliyetlerin dışında **herhangi bir özelliği hayata geçirmek için** de proxy örüntüsü kullanılabilir
- Bu durumda “**explicit**” olarak **proxy oluşturulması** gerekecektir
- Spring bunun için **ProxyFactoryBean** sınıfını sunar

ProxyFactoryBean

- Spring **FactoryBean** implemantasyonudur
- Proxy nesnenin sınıfını interface veya sınıf tabanlı yöntemle **dinamik olarak üretir**
- Hedef bean'i sarmalayan bir **proxy yaratır**
- Proxy nesneye özel davranış
org.aopalliance.intercept.MethodInterce
ptor arayüzü ile implement edilir

ProxyFactoryBean

```
<bean id="petClinicDao"
      class="com.javaegitimleri.petclinic.dao.PetClinicDaoInMemoryImpl"/>

<bean id="petClinicServiceTarget"
      class="com.javaegitimleri.petclinic.service.PetClinicServiceImpl">
  <constructor-arg ref="petClinicDao"/>
</bean>

<bean id="petClinicServiceProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="petClinicServiceTarget"/>
  <property name="interfaces">
    <array>
      <value>
        com.javaegitimleri.petclinic.service.PetClinicService
      </value>
    </array>
  </property>
  <property name="interceptorNames">
    <value>cachingInterceptor,loggingInterceptor</value>
  </property>
</bean>
```

MethodInterceptor

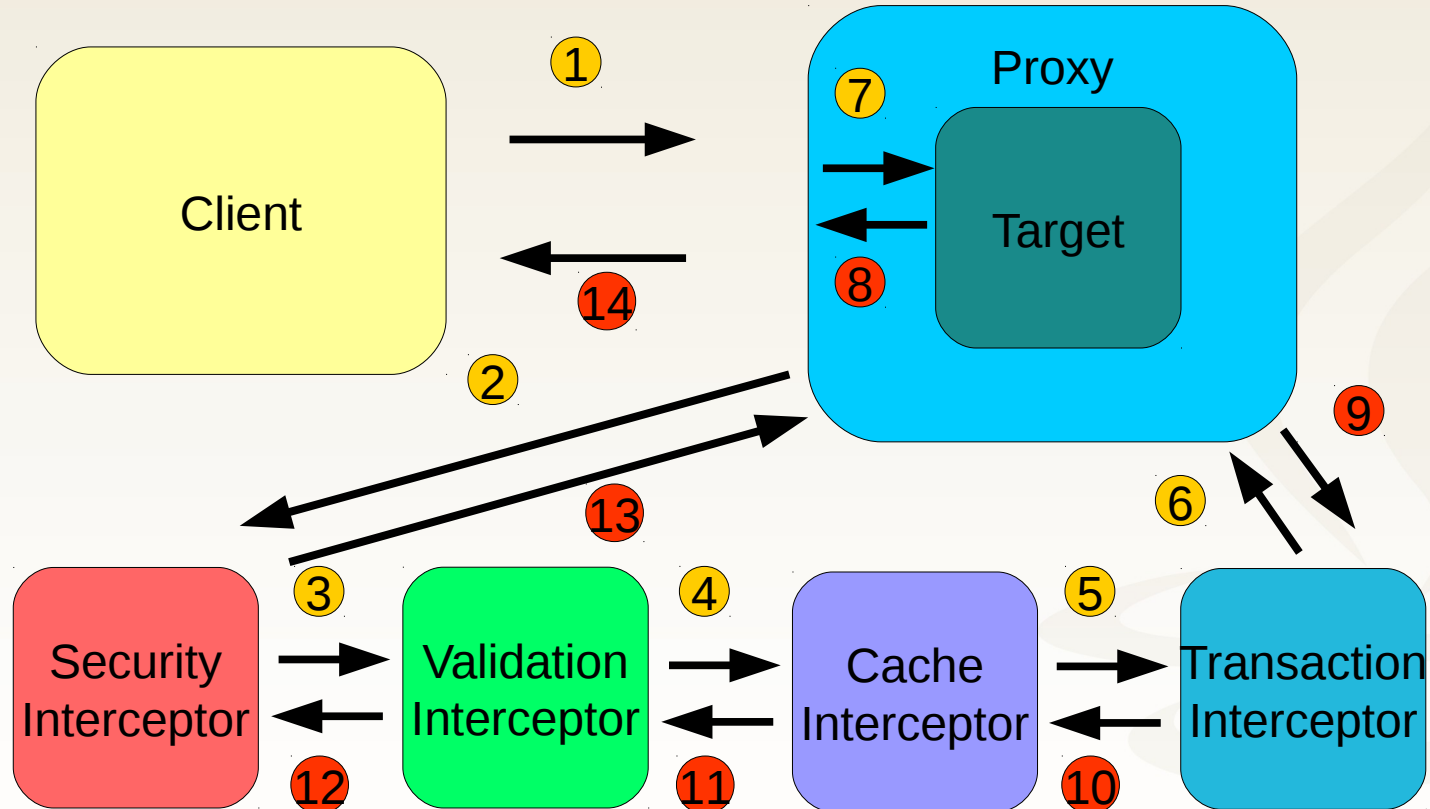
```
<bean id="cachingInterceptor"
class="com.javaegitimleri.petclinic.service.CachingInterceptor"/>
```

```
<bean id="loggingInterceptor"
class="com.javaegitimleri.petclinic.service.LoggingInterceptor"/>
```

```
public class LoggingInterceptor implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        try {
            System.out.println("method entered");
            Object result = invocation.proceed();
            System.out.println("method executed successfully, returning
result :" + result);
            return result;
        } catch (Throwable t) {
            System.out.println("exception occurred :" + t);
            throw t;
        } finally {
            System.out.println("method exited");
        }
    }
}
```

MethodInterceptor Zinciri



Dependency Hot Swap Kabiliyeti

- Bazı durumlarda bir bileşenin bağımlı olduğu bean instance'ının **çalışma zamanında dinamik olarak** başka bir bean instance ile değiştirilmesi gerekebilir
- Örneğin bakım nedeni ile uygulamanın geçici olarak **istekleri asıl veritabanı yerine yedek veritabanına yönlendirmesi** istenebilir
- Spring ile bunu **uygulamayı kapatmadan çalışma zamanında yapmak** mümkündür

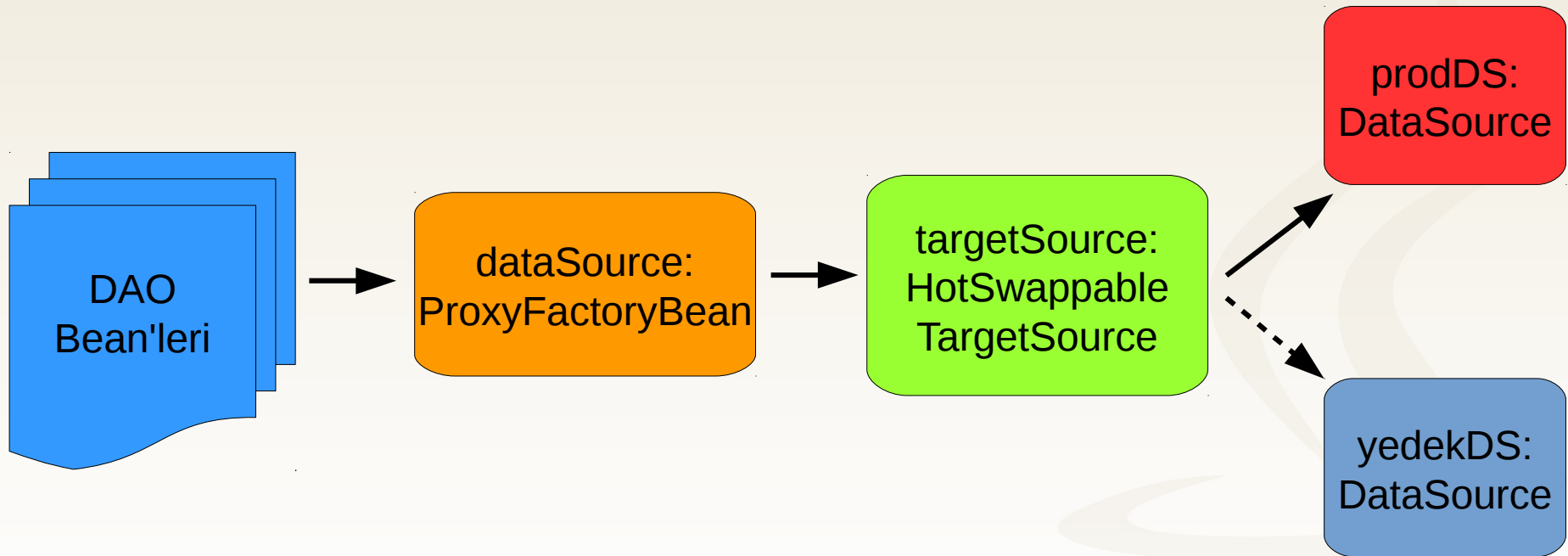
Dependency Hot Swap Kabiliyeti

- Proxy örüntüsü üzerine kurulu bu kabiliyeti hayata geçirmek için **TargetSource** arayüzü kullanılır
- **ProxyFactoryBean**'e doğrudan target bean'i enjekte etmek yerine **TargetSource** arayüzünü implement eden başka bir bean enjekte edilir
- Proxy nesneye gelen istekler targetSource üzerinden **hedef bean'e** iletilir




Dependency Hot Swap Kabiliyeti

- **HotSwappableTargetSource** özel implemantasyonu çalışma zamanında target bean'i swap etmeyi sağlayan bir metoda sahiptir
- Gerektiğinde uygulama içerisinde **swap metodu** ile hedef bean instance'ı değiştirilir
- Böylece müteakip metot çağrıları **yeni hedef bean tarafından** karşılanacaktır

Dependency Hot Swap Kabiliyeti



Dependency Hot Swap Kabiliyeti

```
<bean id="petClinicDao"  
    class="com.javaegitimleri.petclinic.dao.PetClinicDaoJdbcImpl">  
    <property name="dataSource" ref="dataSource"/>  
</bean>  
  
<bean id="dataSource"   
    class="org.springframework.aop.framework.ProxyFactoryBean">  
    <property name="targetSource" ref="targetSource"/>  
    <property name="interfaces">  
        <array>  
            <value>javax.sql.DataSource</value>  
        </array>  
    </property>  
</bean>  
  
<bean id="targetSource"   
    class="org.springframework.aop.target.HotSwappableTargetSource">  
    <constructor-arg ref="prodDS"/>  
</bean>  
  
<jee:jndi-lookup jndi-name="java:comp/env/jdbc/prodDS" id="prodDS" >  
<jee:jndi-lookup jndi-name="java:comp/env/jdbc/yedekDS" id="yedekDS"/>
```

Dependency Hot Swap Kabiliyeti

```
HotSwappableTargetSource targetSource =  
    applicationContext.getBean(HotSwappableTargetSource.class);  
  
DataSource yedekDS =  
    applicationContext.getBean("yedekDS", DataSource.class);  
  
targetSource.swap(yedekDS);
```

Java Tabanlı Container Konfigürasyonu

- Spring **bean tanımlarının** Java sınıflarında yapılmasını sağlar
- XML tabanlı konfigürasyona **birebir karşılık** gelmektedir
- Avantajı “**type safety**” dir
- Konfigürasyon metadata'nın yazıldığı Java sınıfları **@Configuration** anotasyonu ile işaretlenmelidir

XML vs Java Konfigürasyonları

appContextConfig.xml

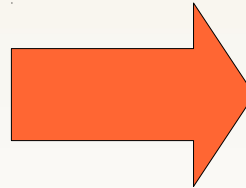
```
<beans...>
```

```
<bean id="foo" class="x.y.Foo">
  <property name="bar">
    <ref bean="bar"/>
  </property>
  <property name="baz"
ref="baz"/>
</bean>
```

```
<bean id="bar"
class="x.y.Bar"/>
```

```
<bean id="baz"
class="x.y.Baz"/>
```

```
</beans>
```



@Configuration

```
public class AppConfig {
```

@Bean

```
public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());
    foo.setBaz(baz());
    return foo;
}
```

@Bean

```
public Bar bar() {
    return new Bar();
}
```

@Bean

```
public Baz baz() {
    return new Baz();
}
```

```
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Autowired
    private Bar bar;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```


Bağımlılıkların Enjekte Edilmesi

```
@Configuration  
public class AConfig {
```

```
    @Autowired
```

```
    private BConfig bConfig;
```

```
    @Bean
```

```
    public Foo foo() {  
        Foo foo = new Foo();  
        foo.setBar(bConfig.bar());  
        return foo;  
    }
```

```
}
```

```
@Configuration
```

```
public class BConfig {
```

```
    @Bean
```

```
    public Bar bar() {  
        return new Bar();  
    }
```

```
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Bean
    @Autowired
    public Foo foo() {
        Foo foo = new Foo();
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Bean
    public Foo foo(Bar bar) {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

@ImportResource ve @Import

@ImportResource herhangi bir XML spring bean definition dosyasının yüklenmesini sağlar

```
@Configuration
@ImportResource("classpath:/appcontext/beans-config.xml")
public class AConfig {

}
```

```
@Configuration
@Import(AConfig.class)
public class BConfig {

}
```

@Import herhangi bir başka configuration sınıfının diğer bir configuration sınıfı tarafından yüklenmesini sağlar

@ComponentScan

@Component ve türevi
anotasyonların scan edileceği
paketleri tanımlar

```
@Configuration
@ComponentScan("com.javaegitimleri.petclinic")
public class AConfig {

}
```

@PropertySource

@PropertySource anotasyonu ile belirtilen resource'lar Environment'a PropertySource olarak eklenirler


```
@Configuration
@PropertySource("classpath:/application.properties")
public class AppConfig {

    @Autowired
    Environment env;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setName(env.getProperty("foo.name"));
        return foo;
    }
}
```

@PropertySource

targetPlatform değişkeni halihazırda bu aşamaya kadar Environment'e register olmuş PropertySource nesneleri arasından resolve edilmeye çalışılır.



```

@Configuration
@PropertySource("classpath:/application_${targetPlatform}.properties")
public class AppConfig {

    @Autowired
    private Environment env;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setName(env.getProperty("foo.name"));
        return foo;
    }
}
    
```

Java Tabanlı Konfigürasyon ve Bean Profil Kabiliyeti

Konfigürasyon sınıflarının hangi profillerde yükleneceği sınıf düzeyinde @Profile anotasyonu ile tanımlanır

```
@Configuration
```

```
@Profile("dev")
```

```
public class DevConfig {
```

```
    @Bean
```

```
    public DataSource dataSource() {
```

```
        return new EmbeddedDatabaseBuilder()
```

```
            .setType(EmbeddedDatabaseType.HSQL) Exception {
```

```
            .addScript("classpath:/schema.sql")
```

```
            .addScript("classpath:/data.sql")
```

```
            .build();
```

```
    }
```

```
}
```

```
@Configuration
```

```
@Profile("prod")
```

```
public class ProdConfig {
```

```
    @Bean
```

```
    public DataSource dataSource() throws
```

```
        Exception {
```

```
        Context ctx = new InitialContext();
```

```
        return (DataSource)
```

```
            ctx.lookup("java:comp/env/jdbc/DS");
```

```
    }
```

```
}
```

```
AnnotationConfigApplicationContext applicationContext = new  
AnnotationConfigApplicationContext();
```

```
applicationContext.getEnvironment().setActiveProfiles("dev");
```

```
applicationContext.scan("com.javaegitimleri");
```

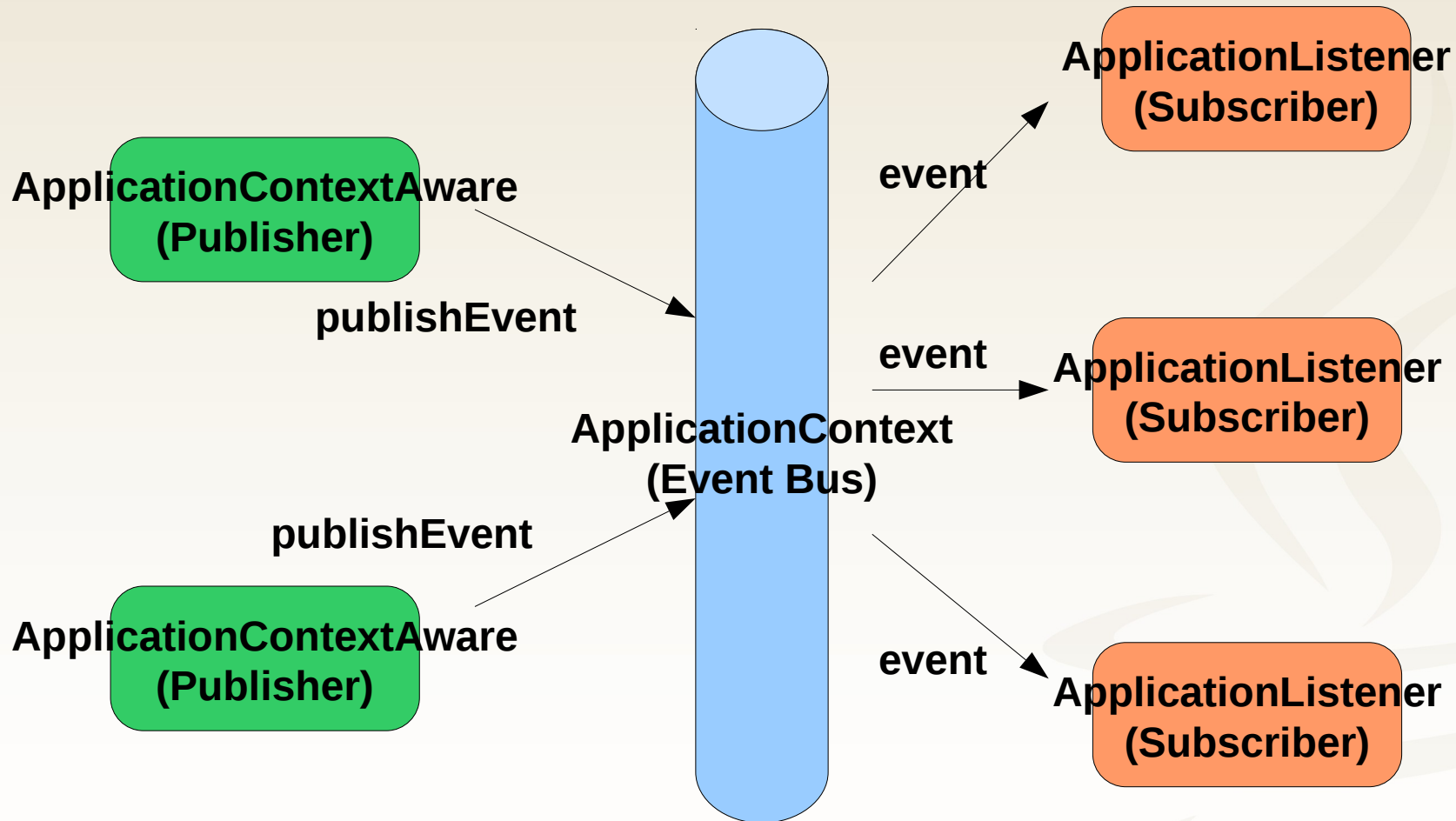
```
applicationContext.refresh();
```

ApplicationContext yaratılırken aktif profillerin hangileri olacağı belirtilir

ApplicationContext Yaratılması

```
AnnotationConfigApplicationContext applicationContext =  
new AnnotationConfigApplicationContext();  
  
applicationContext.register(AConfig.class, BConfig.class);  
  
applicationContext.refresh();  
  
Foo foo = applicationContext.getBean(Foo.class);
```

ApplicationContext Event Mimarisi



Burada **Observer** ve **Mediator** örüntüleri birlikte kullanılmaktadır

Annotasyon Tabanlı Event Listener Tanımı

- Spring 4.2 ile birlikte **ApplicationListener** arayüzünü **implement etmeden** de event handler tanımlamak mümkün hale gelmiştir
- Bunun için **metot düzeyinde @EventListener anotasyonu** kullanılır
- Bu sayede bir bean içerisinde **birden fazla farklı türde event'i yakalayan metotlar** yazmak mümkündür

Annotasyon Tabanlı Event Listener Tanımı

```
@Component
public class MyApplicationEventListener {

    @EventListener
    public void handle(ContextRefreshedEvent event) {
        System.out.println("ApplicationContext is ready to serve...");
    }

    @EventListener
    public void handle(ContextClosedEvent event) {
        System.out.println("ApplicationContext shutdown...");
    }
}
```

Transactional Event Listener Kabiliyeti

- **@TransactionalEventListener** anotasyonu event'lerin sadece **transactional bir metot içerisinde** fırlatıldıklarında yakalanmalarını sağlar
- Eğer event transactional bir metot içerisinde fırlatılmamış ise **göz ardı** edilir
- Metodun çağırılma zamanı da **TransactionPhase** ile belirlenebilir
- Böylece metodun TX **commit öncesi** veya **sonrası**, yada sadece **rollback** olduğunda invoke edilmesi sağlanabilir

Transactional Event Listener Kabiliyeti

```
@Component
public class MyApplicationEventListener {

    @TransactionalEventListener(
        phase=TransactionPhase.AFTER_COMMIT)
    public void handle(PetCreatedEvent event) {
        //handle pet created event here...
    }
}
```

BEFORE_COMMIT
AFTER_COMMIT
AFTER_ROLLBACK
AFTER_COMPLETION

Ayrıca **fallbackExecution=true** ifadesi ile transactional context dışında fırlatılan bir event'in de yakalanması sağlanabilir

İletişim

- **Harezmi** Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- <http://www.java-egitimleri.com>
- info@java-egitimleri.com

