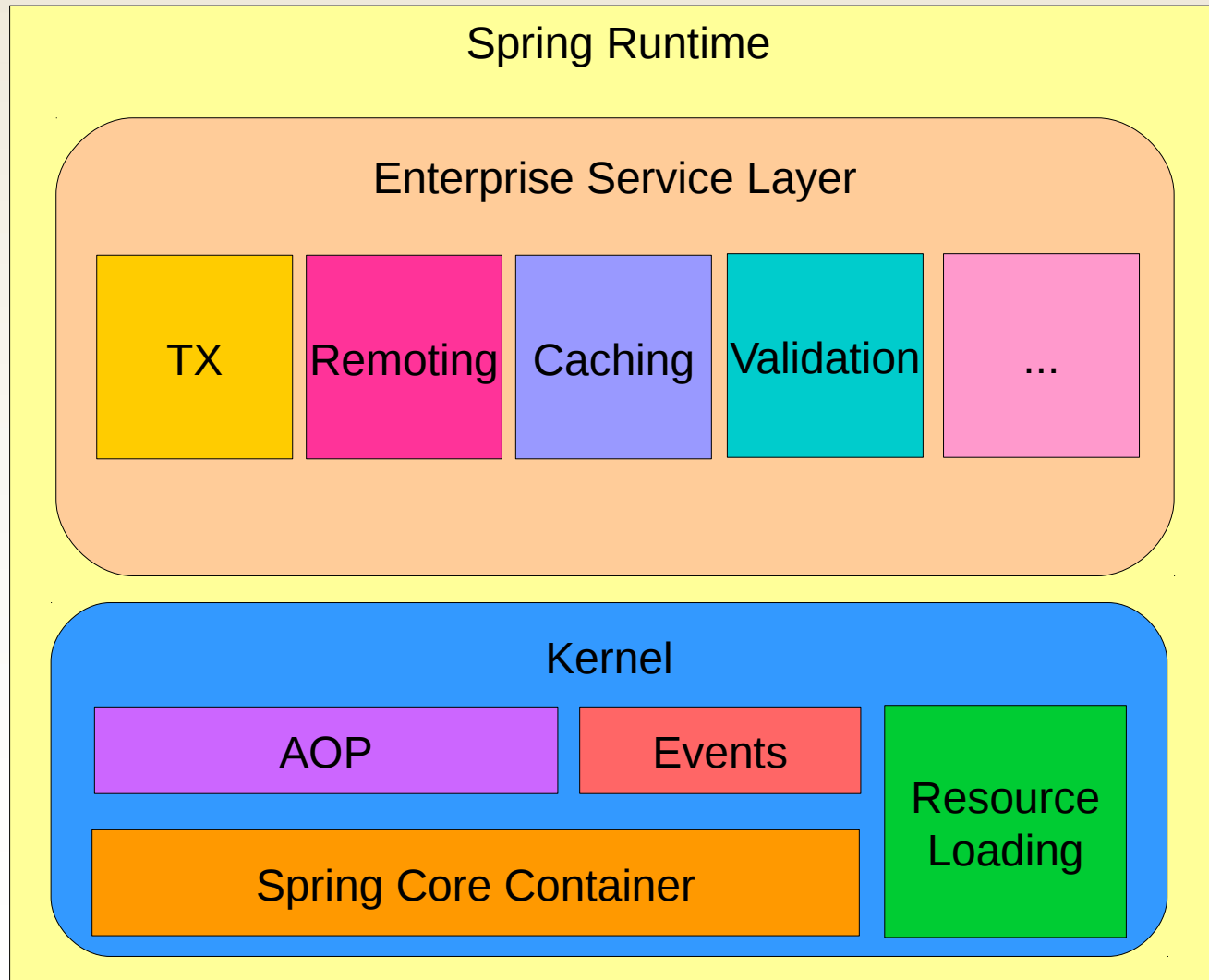


Spring Container Kabiliyetleri

1



Spring Runtime'a Genel Bakış



Spring Bootstrap Süreci

- İlk adımda **ApplicationContext'in yaratılması** gerekir
 - Konfigürasyon metadata'nın yüklenerek ApplicationContext nesnesinin oluşturulması gerekir
 - ApplicationContext programatik (standalone) veya deklaratif (web) biçimde yaratılabilir
 - Konfigürasyon metadata farklı formatlarda (xml, java, annotation) olabilir

Spring Bootstrap Süreci

- Daha sonraki adımda **uygulamanın bileşen konfigürasyonu** tespit edilir
- Bu aşamada konfigürasyon metadata'daki bean tanımları, runtime'da **BeanDefinition** nesnelere dönüştürülür
- BeanDefinition nesneleri **bileşenlerin özellikleri, hangi bağımlılıklara ihtiyaç duyduğu** gibi bilgileri içerir

Spring Bootstrap Süreci

- Bir sonraki adım **bileşenlerin** (bean instance'ların) **yaratılması**dır
- Bu aşama **iki fazda** gerçekleşir
- Önce bazı **özel altyapısal bean'ler** yaratılır (BeanFactoryPostProcessor)
- Bu özel bean'lar diğer **BeanDefinition'ların üzerinde özelleştirmeler/değişiklikler** yapabilirler

Spring Bootstrap Süreci

- Daha sonra ise **başka bazı altyapısal bean'ler** (BeanPostProcessor) yaratılır
- Bu özel bean'ler ise yaratılacak olan **uygulama bean instance'larının üzerinde özelleştirme veya değişiklik yapacaklardır**
- Bu aşamada artık uygulamanın **normal bean instance'ları** yaratılmaya başlanır

Spring Bootstrap Süreci

- Her bean yaratılıp **bağımlılıkları enjekte** edilir
- Varsa **init metotları çağrılmadan önce BeanPostProcessor** tarafından özelleştirmeye tabi tutulabilir
- Ardından **init metotları çağrılır**
- Sonra **yine BeanPostProcessor** tarafından özelleştirilebilir
- Bu aşamanın ardından artık bean **kullanıma hazır** hale gelmiştir

ApplicationContextInitializer

- ConfigurableApplicationContext'in initialize edilmesi sırasında devreye giren **callback interface**'dir
- Çoğunlukla web uygulamalarında ApplicationContext üzerinde **programatik olarak bir özelleştirme veya ayar yapmak** için kullanılır
 - PropertySource eklemek veya aktif profilleri set etmek gibi

ApplicationContextInitializer Kullanımı

```
public class MyInitializer implements
ApplicationContextInitializer<ConfigurableWebApplicationContext> {

    public void initialize(ConfigurableWebApplicationContext ctx) {
        ...
    }
}
```

web.xml



```
<context-param>
    <param-name>contextInitializerClasses</param-name>
    <param-value>x.y.MyInitializer</param-value>
</context-param>
```

Container Extension Noktaları

- ApplicationContext'e ve içindeki bean'lara dinamik olarak **yeni özellikler eklemek** mümkündür
- Bunun için “**pluggable extension point**”ler vardır
- Bu extension point'ler **iki tür**lüdür
 - Namespace handler'lar
 - Post-processor bean'lar

Post-Processor Bean Kabiliyeti

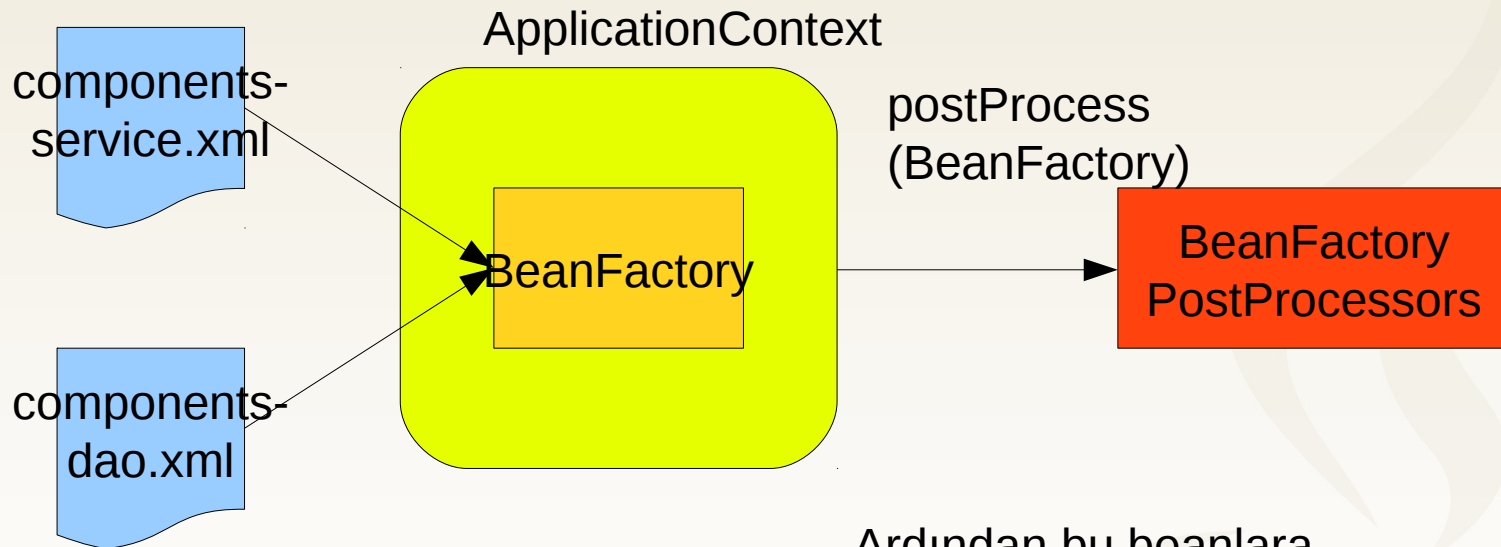
- ApplicationContext'e ve içindeki bean'lara dinamik olarak **yeni özellikler eklemek** sağlayan özel bean'lardır
- Kendi içinde ikiye ayrılırlar
 - **BeanFactoryPostProcessor**
 - **BeanPostProcessor**

BeanFactory

PostProcessor

- Bean tanımları (**BeanDefinition**) üzerinde işlem yaparlar
- Özel bean'lerdir
- Container tarafından otomatik tanınırlar
- Diğer **bütün bean'lerden önce** yaratılırlar
- Normal bean tanımları application context tarafından işlenip bean'lar yaratılmadan önce (!) devreye girerek **bean tanımlarını değiştirmeye yararlar**

PostProcessor Nasıl Çalışır?



ApplicationContext,
BeanFactoryPostProcessor
arayüzüne sahip bean'ları **startup**
aşamasında
tespit eder

Ardından bu beanlara
konfigürasyon metadata'yı
okuma ve değiştirme izini verir

Container diğer bean'ları yaratmaya
bu aşamadan sonra başlar

Örnek: Property Placeholder Kabiliyeti

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

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

```
</beans>
```



Bu namespace tanımı sayesinde ApplicationContext'e PropertySourcesPlaceholder isimli BeanFactoryPostProcessor'ü eklenir.

Bu post-processor'de diğer bean tanımları üzerindeki placeholder değişkenlerini resolve eder.

Örnek: Property Placeholder Kabiliyeti

```
<bean id="dataSource"  
class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}"/>  
    <property name="url" value="${jdbc.url}"/>  
    <property name="username" value="${jdbc.username}"/>  
    <property name="password" value="${jdbc.password}"/>  
</bean>
```

ApplicationContext

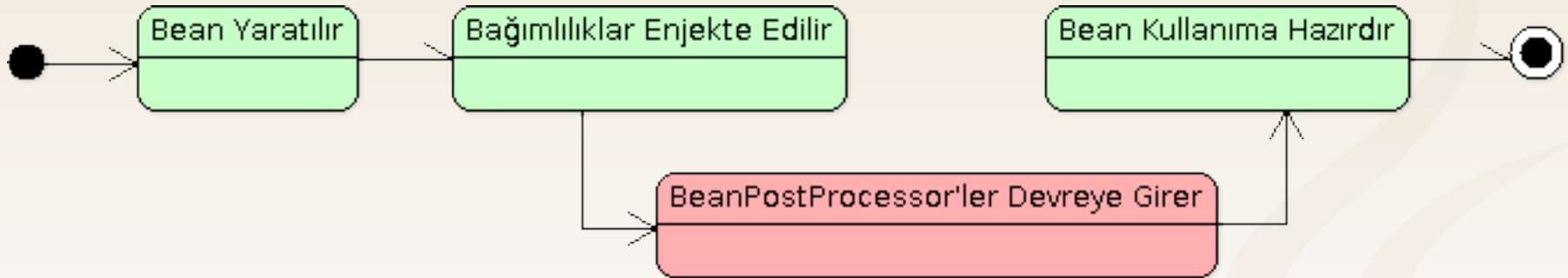
PropertySources
PlaceholderConfigurer

```
<bean id="dataSource"  
class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>  
    <property name="url" value="jdbc:hsqldb:hsqldb://production:9002"/>  
    <property name="username" value="sa"/>  
    <property name="password" value="secret"/>  
</bean>
```

BeanPostProcessor

- Normal **bean instance**'ların üzerinde değişiklik yapmayı sağlarlar
- Bunlarda özel bean'lardır
- Container tarafından otomatik olarak tanınırlar
- Diğer bean'lardan önce yaratılırlar

BeanPostProcessor Nasıl Çalışır?




Her bean instance'ı için post processor'ler ayrı ayrı devreye girer

Devreye girme ilk olarak bağımlılıklar enjekte edildikten sonra, fakat initialization metotları çağrılmadan önce, ikinci olarak da initialization metotları çağrıldıktan sonra olmak üzere iki defa gerçekleşir

Post processor'ler orijinal bean instance'ı üzerinde değişiklik yapabilirler, onu wrap edebilirler. Örneğin bir proxy yaratabilirler ve bu proxy'yi asıl bean olarak dönebilirler

Örnek: Caching Kabiliyeti

```
<beans...>  
  <cache:annotation-driven/>  
</beans>
```

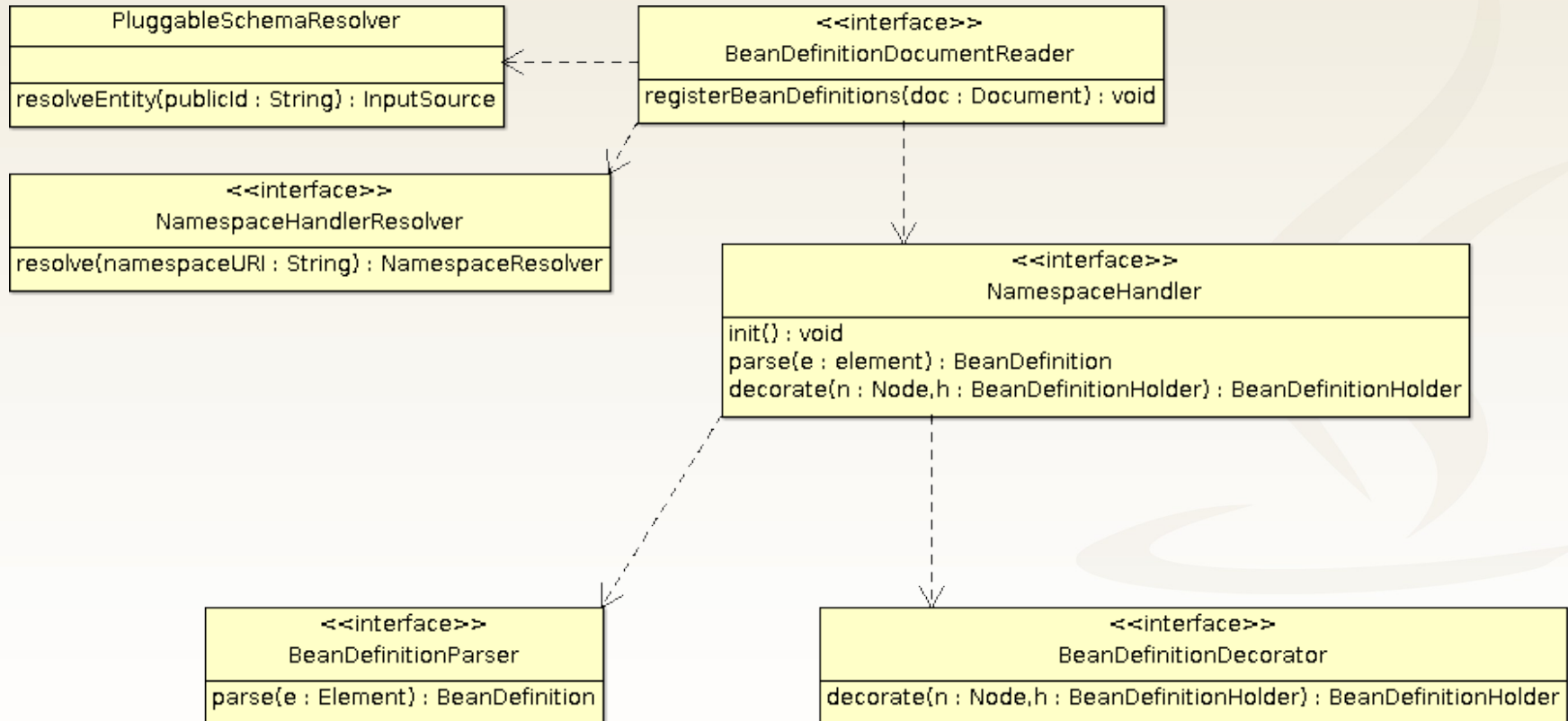


Bu namespace tanımı sayesinde ApplicationContext'e özel bir BeanPostProcessor tanımı eklenir.

Bu post-processor'de diğer bean tanımlarının sınıf veya metotları üzerinde @Cacheable anotasyonlarını tespit ederek bu bean instance'larında cache kabiliyetini devreye sokar.

NamespaceHandler Kabiliyeti Nasıl Çalışır?

/META-INF/spring.schemas: namespaceURI=XSD location path
/META-INF/spring.handlers: namespaceURI=NamespaceHandler FQN



```

<beans...>
  <jee:jndi-lookup id="dataSource"
    jndi-name= "java:comp/env/jdbc/DS"/>
</beans>
  
```

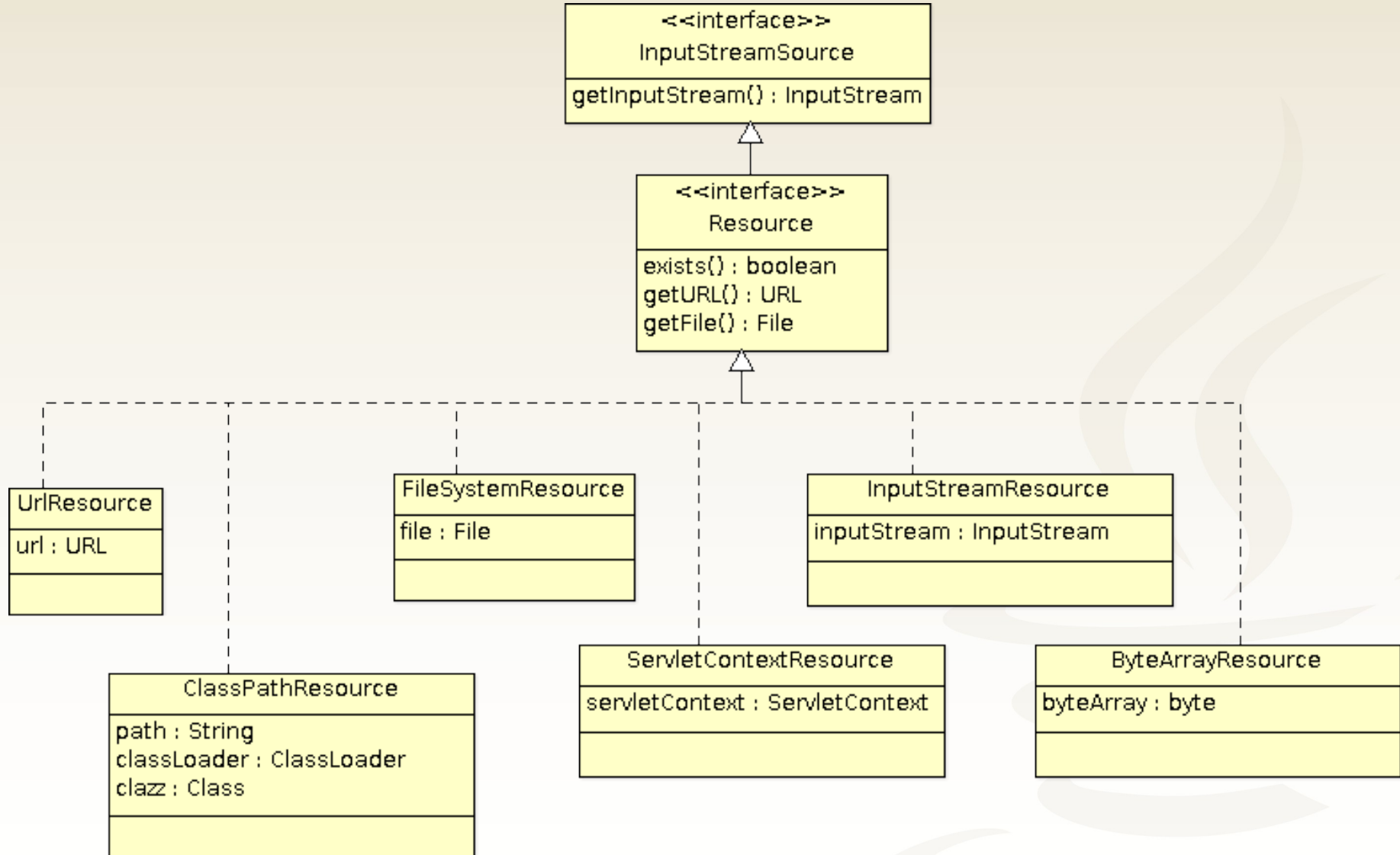
```

<beans...>
  <bean id="userPreferences"
    class="x.y.UserPreferencesImpl" scope="session">
    <aop:scoped-proxy/>
  </bean>
</beans>
  
```

Spring Resource API

- **Classpath, Filesystem veya ServletContext'e** relatif resource'lara erişimi kolaylaştırır
- Spring'in kendi içinde de sıkça kullan bir yöntemdir
- Temeli yine **java.net.URL**, **java.io.InputStream** gibi sınıflara dayanır
- Resource'ların nereden ve nasıl yükleneceği **başındaki prefix'e** bakılarak anlaşılır

Spring Resource Tipleri



ResourceLoader ve ResourceLoaderAware

- **ResourceLoader**, Resource nesnelerini yükler
- **ApplicationContext** aynı zamanda bir **ResourceLoader** nesnesidir
- **ResourceLoaderAware** arayüzüne sahip bean'lara ResourceLoader otomatik olarak enjekte edilir

ResourceLoader ve ResourceLoaderAware

```
public class UserService implements ResourceLoaderAware {  
  
    private ResourceLoader resourceLoader;  
  
    @Override  
    public void setResourceLoader(ResourceLoader  
resourceLoader) {  
        this.resourceLoader = resourceLoader;  
    }  
  
    public InputStream getImage() throws IOException {  
        Resource resource =  
            resourceLoader.getResource("/some/path/image.png");  
        return resource.exists()?  
            resource.getInputStream():null;  
    }  
}
```

Resource Bağımlılıkları

```
public class Foo {
    private Resource image;

    public void setImage(Resource image) {
        this.image = image;
    }
}
```

Buradaki değişken Resource tipindedir

```
<bean id="foo" class="x.y.Foo">
    <property name="image"
value="some/resource/path/image.png" />
</bean>
```

```
<property name="image"
value="classpath:some/resource/path/image.png">
```

```
<property name="image"
value="file:/some/resource/path/image.png" />
```


Konfigürasyon Metadata ve Resource Path Kullanımı

- Resource path ile **ApplicationContext** dosyalarının yeri de belirtilebilir
- Resource path'inde **wildcard** kullanılabilir
- `file:/some/path/beans-*.xml`
- `classpath:/some/path/**/*.applicationContext.xml`

classpath*: Ön Eki Ne İşe Yarar?

- **classpath:/appcontext/beans-*.xml** şeklinde bir kullanımda Spring classpath'de **ilk bulduğu /appcontext/ dizininde** duracaktır
- Burası da web uygulamalarında muhtemelen **WEB-INF/classes** dizini olacaktır
- Spring'in taramaya devam etmesi ve **WEB-INF/lib** dizini altındaki **JAR dosyalarının** içerisini de taraması istenebilir

classpath*: Ön Eki Ne İşe Yarar?

- Böyle bir durumda **classpath*: ön eki** devreye girmektedir
- Resource'ları bulmak için eşleşen **bütün classpath lokasyonlarına** bakılır
 - `classpath*/appcontext/beans-*.xml`

classpath*: Ön Eki Ne İşe Yarar? (ikaz)

- JAR içindeki resource'lara **pattern ile erişilebilmesi için root path altında en azından bir dizinde olmaları gerekir**
 - classpath*/beans-*.xml, jar içerisindeki resourceları bulamaz
 - Sadece expanded dizindeki resourceları bulabilir

ResourcePatternResolver

- **ResourceLoader** arayüzünü extend eder
- Değişik **location-pattern**'ları resolve etmek için (/appcontext/beans-*.xml) kullanılır
- **Classpath***: prefix'inin davranışı da bu arayüz tarafından ortaya konmaktadır

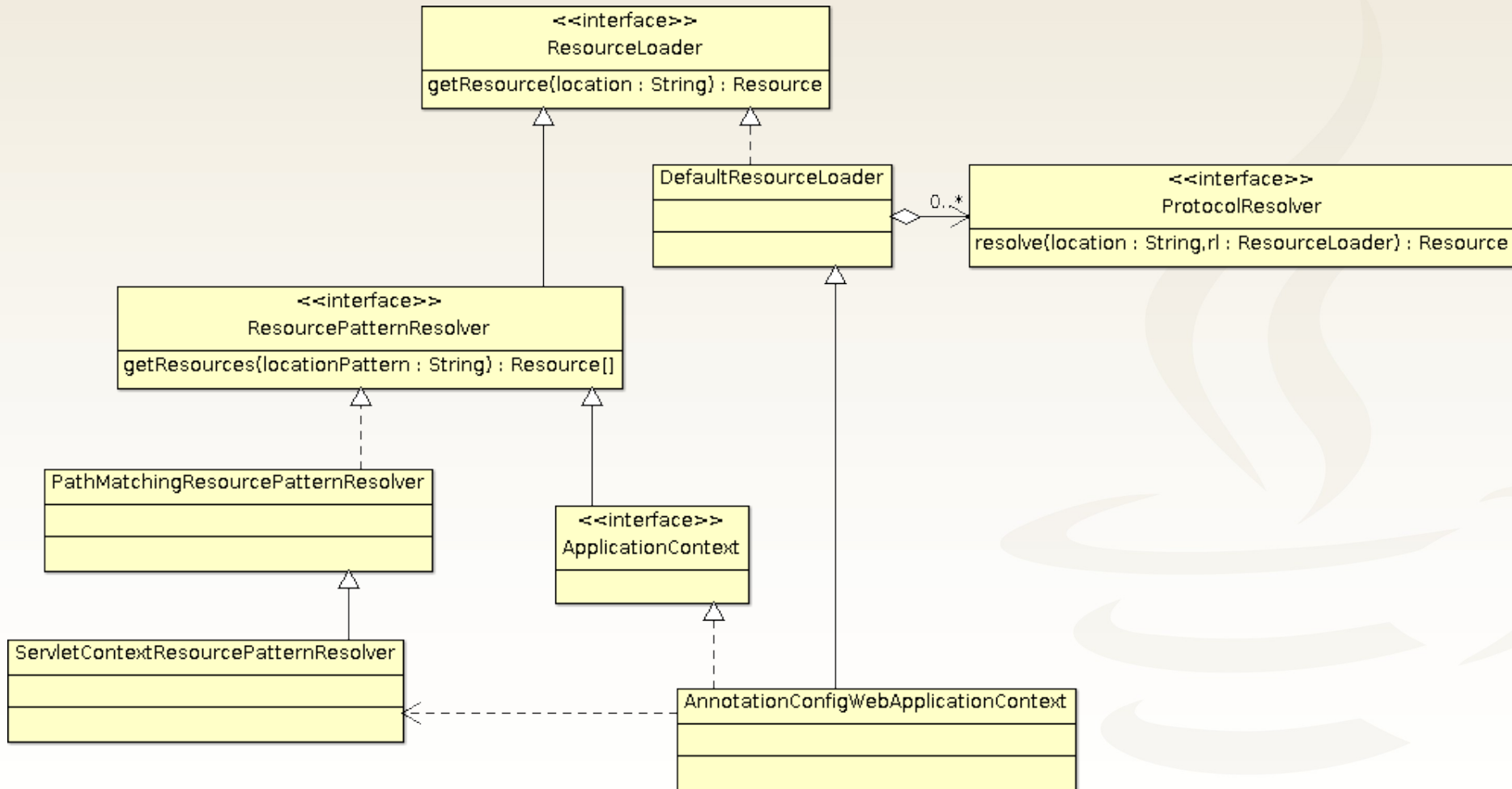
```
public interface ResourcePatternResolver extends ResourceLoader {
    String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
    Resource[] getResources(String locationPattern) throws IOException;
}
```

ProtocolResolver

- **DefaultResourceLoader** tarafından kullanılan bir SPI'dır
- **ResourceLoader override/implement etmeden** Resource resolution işlemini özelleştirmek için kullanılır
- Kullanmak için spesifik implemantasyonun **DefaultResourceLoader.addProtocolResolver()** metodu ile eklenmesi gerekir

```
public interface ProtocolResolver {
    Resource resolve(String location, ResourceLoader resourceLoader);
}
```

Spring ResourceLoader Hiyerarşisi



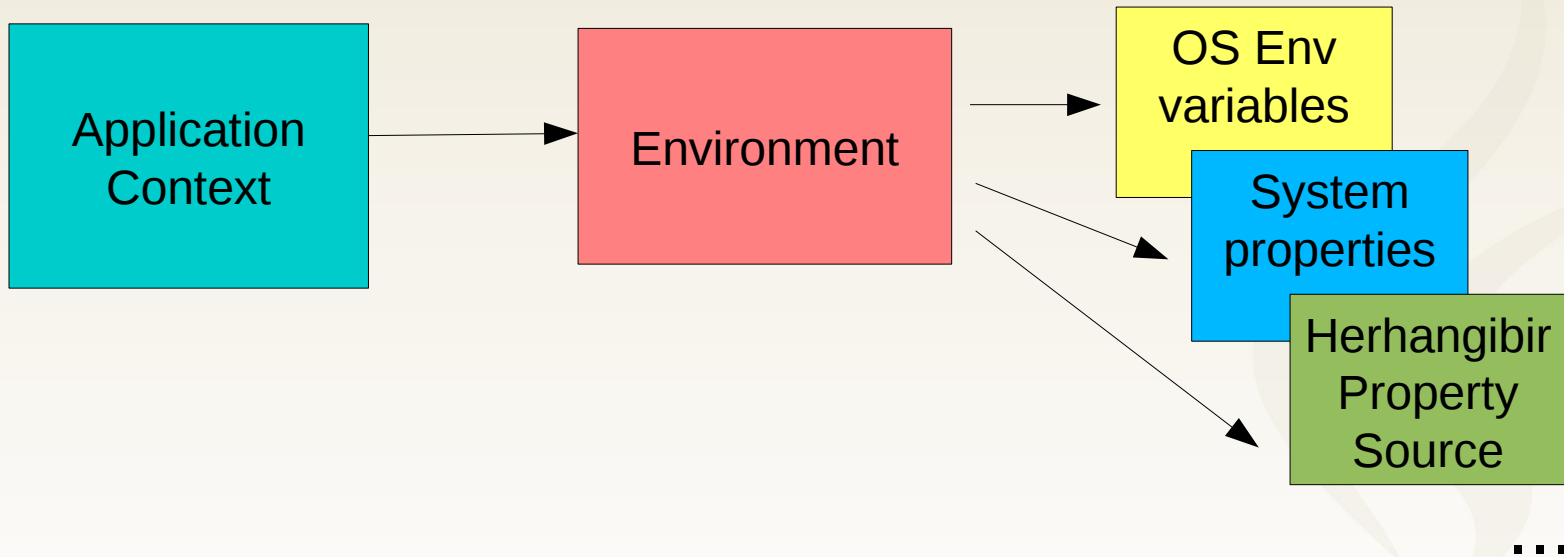
Spring 3 ve Environment Kabiliyeti

- Spring 3 ile birlikte ApplicationContext'e **Environment** isimli bir soyutlama eklendi
- Environment **uygulamanın çalıştığı ortamı** ifade etmektedir
- Uygulamanın ortamsal iki temel özelliğini yönetir
 - Profiller
 - Property değerleri

PropertySource

- Spring 3 ile birlikte Environment üzerinden **property değerlerinin çözümlemesi** çok daha esnek bir hale geldi
- Artık sadece işletim sisteminin ortam değişkenleri, JVM sistem değişkenleri ve properties dosyalarındaki tanımlar dışında **farklı kaynaklardan da property değerleri** yüklenebilir
- Bu property kaynaklarının **öncelik sıralaması** değiştirilebilir

PropertySource



```
ApplicationContext context = new
    ClassPathXmlApplicationContext("/appcontext/beans.xml");
```

```
Environment env = context.getEnvironment();
String foo = env.getProperty("foo");
```

PropertySource Konfigürasyonu: Web

ApplicationContextInitializer arayüzü implement edilerek custom PropertySource nesnesi environment'a eklenebilir

```
public class MyInitializer implements  
ApplicationContextInitializer<ConfigurableWebApplicationContext> {  
  
    public void initialize(ConfigurableWebApplicationContext ctx) {  
        PropertySource ps = new MyPropertySource();  
        ctx.getEnvironment().getPropertySources().addFirst(ps);  
    }  
}
```

ApplicationContextInitializer web uygulamalarında web.xml'de context param vasıtası ile devreye sokulabilir

```
<context-param>  
    <param-name>contextInitializerClasses</param-name>  
    <param-value>x.y.z.MyInitializer</param-value>  
</context-param>
```

PropertySource

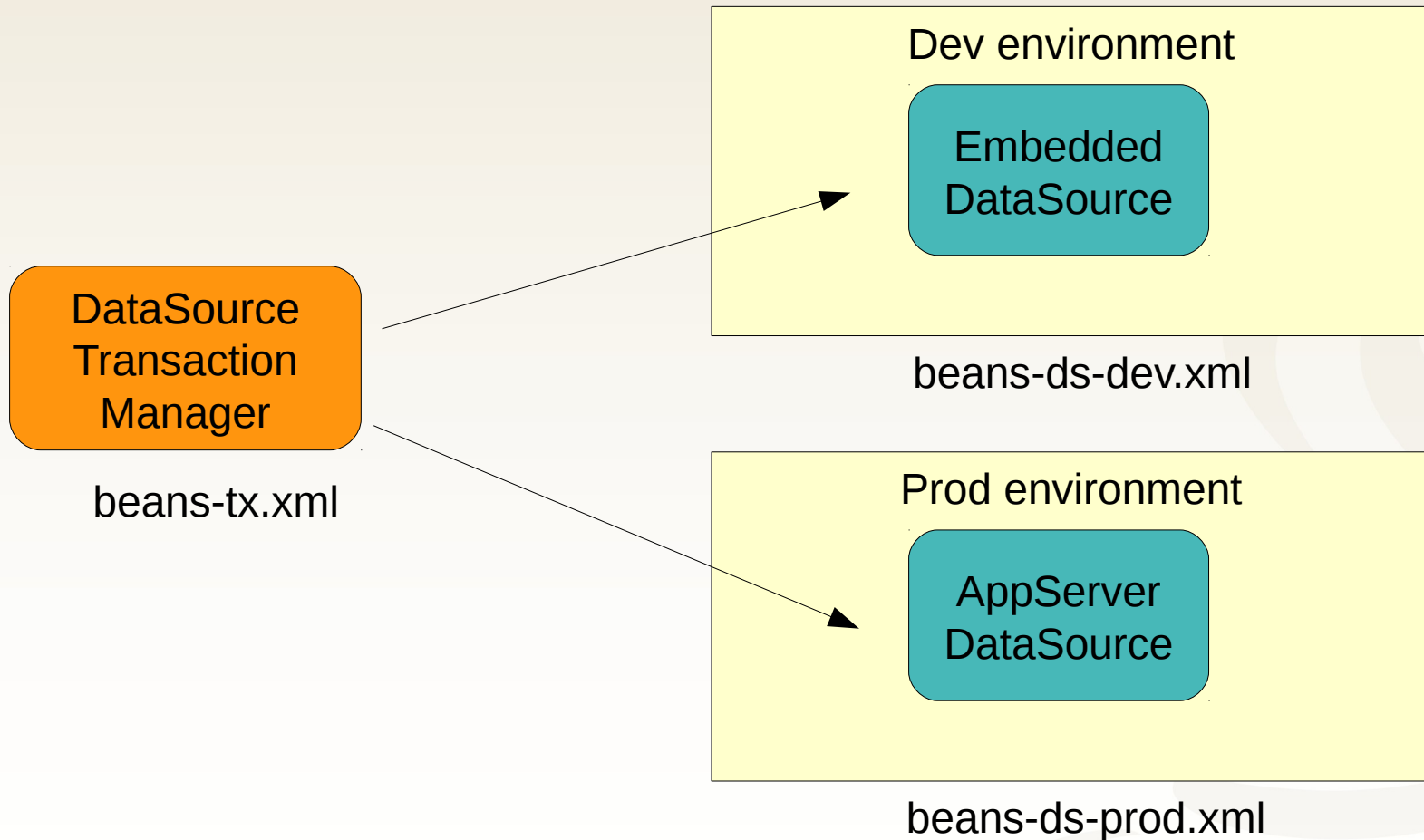
Konfigürasyonu: Standalone

Standalone ortamda ise custom PropertySource
GenericXmlApplicationContext üzerinden eklenebilir

```
GenericXmlApplicationContext context =  
    new GenericXmlApplicationContext();  
  
context.load(new ClassPathResource("/appcontext/beans.xml"));  
  
context.getEnvironment()  
    .getPropertySources().addFirst(new MyPropertySource());  
  
context.refresh();
```

Refresh metodu çağrılana kadar ApplicationContext konfigürasyonu yapılabilir. Refresh metodu ile bean'ler yaratılacaktır.

Spring 3 Öncesinde Platforma Özgü Bean Tanımları



Spring 3 Öncesinde Platforma Özgü Bean Tanımları

beans-tx.xml



```
<beans...>

  <context:property-placeholder
location="classpath:/application.properties,
classpath:/application.${targetPlatform}.properties" />

  <import resource="classpath:/appcontext/beans-ds-${
targetPlatform}.xml" />

</beans>
```

targetPlatform değerine göre ya beans-ds-dev.xml yada beans-ds-prod.xml konfigürasyon dosyalarından sadece birisi yüklenecektir

Bean Profile Kabiliyeti

- Bazı durumlarda **farklı platformlara veya ortamlara farklı bean tanımları** yapmak gerekebilir
- Örneğin **dev ortamında** geliştirme sürecini kolaylaştıracak test amaçlı **embedded bir dataSource** bean'i tanımlanabilir
- **Prod ortamında** ise uygulama sonucusunda tanımlamış **connection pool kabiliyetine sahip dataSource** instance'ının kullanılması istenebilir

- **Spring 3 öncesinde** böyle bir ihtiyaç bean tanımlarını farklı konfigürasyon dosyalarında yapıp, **platforma uygun konfigürasyon dosyalarını** placeholder yardımı ile **import ederek** çözülmekteydi
- Spring 3 ile birlikte **bean profile kabiliyeti** sayesinde aynı bean konfigürasyonu içerisinde **bean tanımlarını platforma veya ortama göre gruplayarak yapmak** mümkün hale gelmiştir

Spring 3 ve Bean Profile Kabiliyeti

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

```
    <bean id="transactionManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
    </bean>
```

```
<beans profile="dev">  
    <jdbc:embedded-database type="H2" id="dataSource">  
        <jdbc:script location="classpath:/schema.sql"/>  
        <jdbc:script location="classpath:/data.sql"/>  
    </jdbc:embedded-database>  
</beans>
```

```
<beans profile="prod">  
    <jee:jndi-lookup jndi-name="java:comp/env/jdbc/DS" id="dataSource"/>  
</beans>
```

```
</beans>
```

Bean Profile Kabiliyeti

- Child `<beans profile="...">` tanımları **diğer bean tanımlarından sonra** yapılmalıdır
- Kendilerinden sonra **başka tanım olmamalıdır**
- Profil tanımları deklaratif veya programatik olarak aktive edilebilir
- Programatik olarak **ApplicationContextInitializer** arayüzü implement edilerek **Environment** üzerinde set edilebilir

- Dekleratif olarak JVM sistem parametresi olarak belirtilebilir
 - -Dspring.profiles.active=dev,oracle
- Ya da web.xml'de **spring.profiles.active** context paramteresi ile set edilebilir

```
<context-param>  
  <param-name>spring.profiles.active</param-name>  
  <param-value>dev,oracle</param-value>  
</context-param>
```

Bean Profile Kabiliyeti ve Testler

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/appcontext/beans.xml")
@ActiveProfiles({"prod"})
public class FooTests {
```

```
@Test
public void testFoo() {
    ...
}
```

```
}
```

`@ActiveProfiles` anotasyonu ile testlerin çalışması sırasında aktif olması istenen profiller listelenir

İletişim

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

