JAVA Eğitimleri

Tasarım Örüntüleri ile Spring Eğitimi 2

harezmi bilişim çözümleri

XML Tabanlı Spring Container Konfigürasyonu

XML Tabanlı Konfigürasyon Nasıl Yapılır?

JAVA Eğitimleri

- XML konfigürasyon dosyaları içerisinde **bean tanımları** `<beans></beans>` root elemanı altında gerçekleştirilir
- Her bean için `<bean></bean>` elemanı ile **ayrı bir bean tanımı** yapılır
- Bean'in **ihtiyaç duyduğu bağımlılıklar** da `<property></property>` elemanları ile enjekte edilir

XML Tabanlı Konfigürasyon Nasıl Yapılır?

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

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

  <bean id="foo" class="x.y.Foo">
      <property name="myBar" ref="bar"/>
      <property name="age" value="15"/>
  </bean>
  ...

</beans>
```

```
Bar bar = new Bar();
```

```
Foo foo = new Foo();
foo.setMyBar(bar);
foo.setAge(15);
```

XML Tabanlı Konfigürasyon Nasıl Yapılır?

JAVA Eğitimleri

- Genellikle her **bean tanımı** uygulamadaki **bir Java nesnesi**ne karşılık gelir
- **Property tanımları** ile yapılan da, o nesnenin bağımlılıklarının **enjekte edilmesi**dir
- Bean tanımları **bir veya birkaç XML konfigürasyon dosyası**na yayılabilir

Bean İsimlendirme

- Her bean tanımı bir veya daha fazla **identifier**'a (bean ismi) sahiptir
- XML'de **id** veya **name** attribute'ları kullanılarak tanımlanırlar

```
<bean id="foo" class="x.y.z.Foo"/>

<bean name="bar,myBar" class="x.y.z.Bar"/>

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

Bean İsimlendirme

- Bean tanımlarına isim verilmesi **zorunlu değil**dir
- Eğer verilmez ise container **internal** bir isim verir
- Id attribute **tek bir bean ismi** tanımlamaya izin verir
- Bean tanımına **birden fazla isim** vermek için **name** attribute kullanılır
  - Birden fazla isim verilebilir
  - Virgül, boşluk veya noktalı virgül ile ayrılırlar

Bean İsimlendirme

- Aynı XML bean konfigürasyon dosyasında **aynı isimde birden fazla bean** tanımı olamaz
- Ancak **farklı XML dosyalarında** aynı isimde birden fazla bean tanımlanabilir
- Böyle bir durumda ApplicationContext yaratım esnasında sıralamada **sonra gelen** XML dosyasındaki bean tanımı **önce gelen tanımı ezer**
- Sonuç olarak **çalışma zamanında o isimde tek bir bean tanımı** olacaktır

Bean Tanımlarının Ezilmesi (Bean Definition Override)

- Aynı XML konfigürasyon dosyası içerisinde aynı isimde birden fazla bean tanımı olamaz

```
<beans ...>

  <bean id="userService" class="x.y.JdbcUserService">
  </bean>

  <bean id="userService" class="x.y.LdapUserService">
  </bean>

</beans>
```

Bean Tanımlarının Ezilmesi (Bean Definition Override)

- Farklı XML dosyalarında aynı isimli bean tanımı olabilir
- Bu durumda XML dosyalarının yüklenme sırası önem kazanır

beans-service.xml

```
<beans ...>
  <bean id="userService"
    class="x.y.JdbcUserService"/>
</beans>
```

beansOverride.xml

```
<beans ...>
  <bean id="userService"
    class="x.y.LdapUserService"/>
</beans>
```

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(
        "beans-dao.xml",
        "beans-service.xml",
        "beansOverride.xml");
```

XML Import

- XML konfigürasyon dosyalarını başka bir XML konfigürasyon dosya içerisinde **import ederek** de **yükletmek** mümkündür

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <import resource="beans-dao.xml"/>
  <import resource="classpath:/beans-config.xml"/>

  <bean id="foo" class="x.y.Foo">
   ...
  </bean>
</beans>
```

Dependency Injection Yöntemleri

JAVA Eğitimleri

- DI'nın **iki farklı şekli** vardır
  - **Constructor** injection
  - **Setter** injection
- Constructor injectionda bağımlılıklar nesne yaratılırken **constructor parametreleri** ile enjekte edilirler
- Setter injectionda ise bağımlılıklar nesne yaratıldıktan sonra **setter metotlar** aracılığı ile enjekte edilirler

Constructor Injection

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

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

<bean id="foo" class="x.y.Foo">
   <constructor-arg ref="baz"/>
   <constructor-arg ref="bar"/>
</bean>
```

```
Bar bar = new Bar();

Baz baz = new Baz();

Foo foo = new Foo(bar,baz);
```

```
public class Foo {
    public Foo(Bar bar,Baz baz) {

    }
}
```

Bean oluşturulurken verilen **constructor parametrelerinin** her biri bir bağımlılığı karşılar. **Constructor argümanların tiplerine** bakılarak parametrelerin uyumluluğu kontrol edilir

Constructor Injection

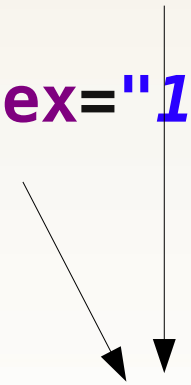
- `<constructor-arg>` elemanlarının **sırası önemli değil**dir
- Spring sınıf içerisinde **uygun constructor**'ı kendisi tespit edecektir
- Fakat bazı durumlarda **bu davranış yetersiz kalabilir**

```
public class Foo {
    public Foo(Baz baz,Bar bar) {
    }

    public Foo(Bar bar, Baz baz) {
    }
}
```

Böyle bir durumda Spring'e hangi constructor-arg hangi constructor argümanına karşılık geliyor belirtmek gerekir

Constructor Injection

```
<bean id="foo" class="x.y.Foo">
  <constructor-arg index="0" ref="baz"/>
  <constructor-arg index="1" ref="bar"/>
</bean>
```



index attribute ile constructor argümanının sırası belirtilebilir (index 0 tabanlıdır)

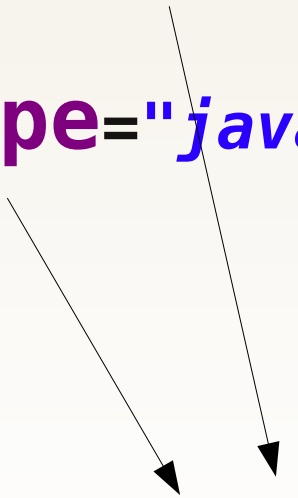
Constructor Injection

- Argüman tipleri primitive ise **hangi constructor-arg elemanının hangi parametreye karşılık geldiği** dışarıdan yardım almadan tespit edilemeyebilir

```
public class Foo {
    public Foo(String s, int i) {
    //...
    }
}
```

Constructor Injection

```
<bean id="foo" class="x.y.Foo">
  <constructor-arg type="int" value="750"/>

  <constructor-arg type="java.lang.String"
value="42"/>

</bean>
```



Type attribute'una verilen tip parametreleri bu belirsizliği çözmeye yardımcı olur

Setter Injection

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

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


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

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

```
Bar bar = new Bar();

Baz baz = new Baz();


Foo foo = new Foo();

foo.setBar(bar);
foo.setBaz(baz);
```

Bean'lerdeki **setter** metotları çağırılarak gerçekleştirilir
Öncelikle nesne **no-arg constructor** çağırılarak yaratılır. Ardından property elemanları ile Bağımlılıklar enjekte edilir

Constructor ve Setter Injection

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

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


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

```
Bar bar = new Bar();

Baz baz = new Baz();

Foo foo = new Foo(bar);

foo.setBaz(baz);
```

Bean tanımında hem constructor, hem de setter injection birlikte kullanılabilir

Tasarım Örüntüleri'ne Devam...

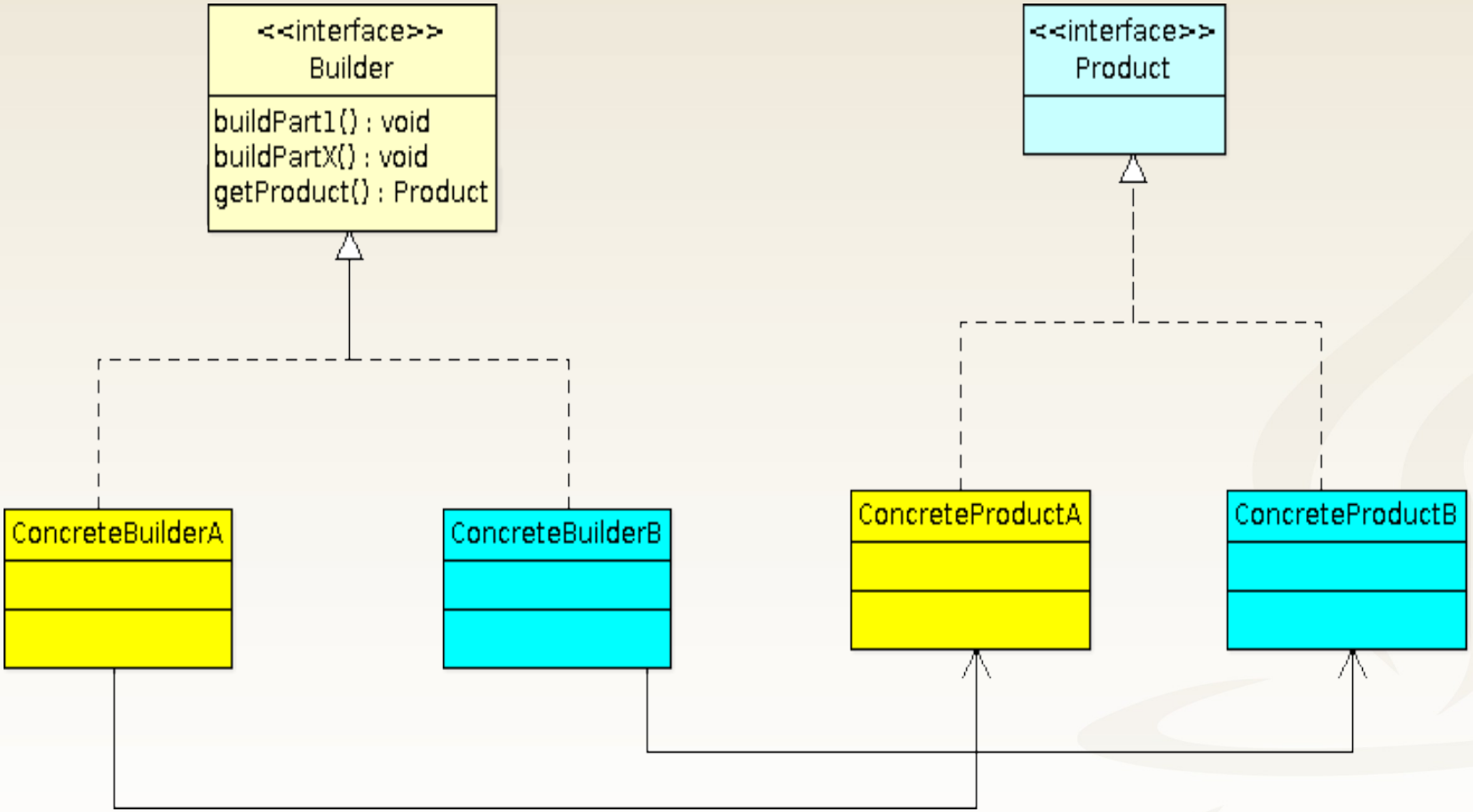
Builder

- Bazı durumlarda nesnelerin oluşturulması **birden fazla adım**dan oluşabilir ve birkaç parçanın bir araya getirilmesi söz konusu olabilir
- Eğer **constructor içerisinde pek çok input argüman** verilmesi gerekiyor ve bunların bir kısmı da opsiyonel ise bunları **nesne yaratıldıktan sonra adım adım metot çağrıları** ile vermek daha doğru ve anlaşılır olacaktır

Builder

- Bir fast-food restoranında menü siparişi düşünelim. Herhangi bir menünün hazırlanması **menü içeriğinden bağımsız** olarak aynıdır. Hamburger, patates kızartması ve içecek eklenir, çocuk menüsü ise hediye oyuncak da menüye dahil edilir vs.
- Hamburgerin et veya tavuk olması, içeceğin kola veya ayran olması, patates kızartmasının standart veya extra büyük olması **menü oluşturma süreci**ni farklılaştırmaz

Builder Sınıf Diagramı



Builder ile nesne **yaratım süreci ve adımları** encapsule edilerek modülerlik sağlanır

Java ve Builder

- JDK'da Builder örüntüsünün kullanıldığı yaygın bilinen iki örnek **StringBuffer** ve **StringBuilder** sınıflarıdır
- **String concatenation** yapmak için kullanılırlar
- **StringBuilder** Java 5 ile birlikte gelmiştir ve StringBuffer'a kıyasla metotları synchronized olmadığı için **daha performanslı**dır
- String contatenation işlemi genellikle metotlarda tek bir thread üzerinden gerçekleştiği için StringBuilder'ın kullanılması daha uygundur

Java ve Builder

String içeriği tutmak için belirtilen uzunlukta bir char dizisi oluşturulur. Eğer String içeriğin uzunluğu bu dizinin uzunluğunu aşarsa otomatik olarak yeni bir dizi yaratılır ve içerik oraya kopyalanır. Varsayılan uzunluk 16 char'dır

```
StringBuilder builder = new StringBuilder(1024);

builder.append("aaa");
builder.append(1L);
builder.append(true);
builder.append('X');
builder.append(new Object());
builder.insert(1, "bbb");
builder.setCharAt(0, 'A');
builder.replace(3, 5, "YYY");

String content = builder.toString();

System.out.println(content);
```

String içerik append, insert, setCharAt, replace gibi metotlarla kademeli olarak oluşturulur. Append metodu ile farklı türde nesneler String içeriğe eklenebilir

Son aşamada toString() metodu çağrılarak String değer elde edilir

Builder Örüntüsünün Sonuçları

- Kompleks nesnenin oluşturulma süreci **nesnenin kendi dışına** taşınmıştır
- **Nesne oluşturma süreci** Builder içerisinde **encapsule** edilmiştir
- Süreç **istemci tarafından kontrol** edilebilir
- Aynı nesnenin **farklı representasyonları** oluşturulabilir

XML Tabanlı Spring Container Konfigürasyonuna Devam...

Bean Oluşturma Yöntemleri: Constructor Çağırmak

- Bean tanımından nesne oluşturmanın en temel yolu doğrudan **sınıf constructor**'ını çağırmaktır

```
<bean id="foo" class="x.y.z.Foo"/>
```

```
Foo foo = new Foo();
```

Bean Oluşturma Yöntemleri: Constructor Çağırmak

JAVA Eğitimleri

```
<bean id="foo" class="x.y.Foo">
  <constructor-arg ref="bar"/>
  <constructor-arg ref="baz"/>
</bean>
```

```
Foo foo = new Foo(bar,baz);
```

Bean Oluşturma Yöntemleri: FactoryBean

- Spring Container, **FactoryBean arayüzü**nü implement eden sınıflara farklı davranır
- FactoryBean arayüzünü implement eden sınıftan bir bean tanımı gördüğünde **FactoryBean.getObject()** metodundan dönen nesne asıl bean olur
- Spring'in kendi içerisinde de **yaygın biçimde kullanılan bir yapı**dır
- Nesnenin **yaratılması birden fazla adım**dan oluşuyor ise bu yöntem idealdir

Bean Oluşturma Yöntemleri: FactoryBean

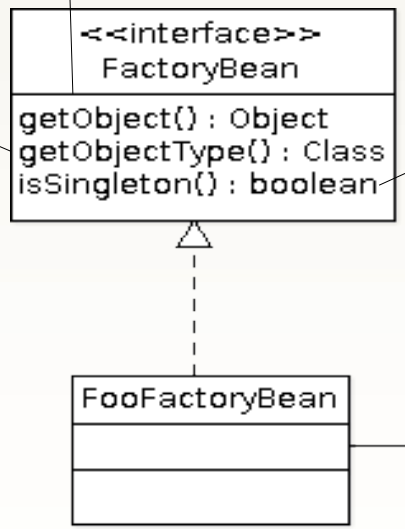
```
<bean id="foo" class="x.y.z.FooFactoryBean"/>
```

```
public Object getObject() {
    Foo f = new Foo();
    f.setName("my foo");
    return f;
}
```

Bean instance'ını döner

Oluşturulacak Bean instance'ın sınıfını döner

Bean scope'unun singleton olup olmadığını belirtir




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

```
public Object
        getObjectType() {
    return Foo.class
}
```

Depends-on Kabiliyeti

- ApplicationContext içerisindeki bean'lerin **yaratım sırası** birbirleri arasındaki **bağımlılık tanımlarına göre** belirlenir
- Eğer iki bean tanımı arasında **herhangi bir bağımlılık yoksa** bu bean'lerin **yaratım sırası container tarafından** belirlenir
- Ancak bazı durumlarda birbirleri arasında **explicit bağımlılık tanımı olmasa bile** bean'lerin **belirli bir sırada yaratılmaları** gerekebilir

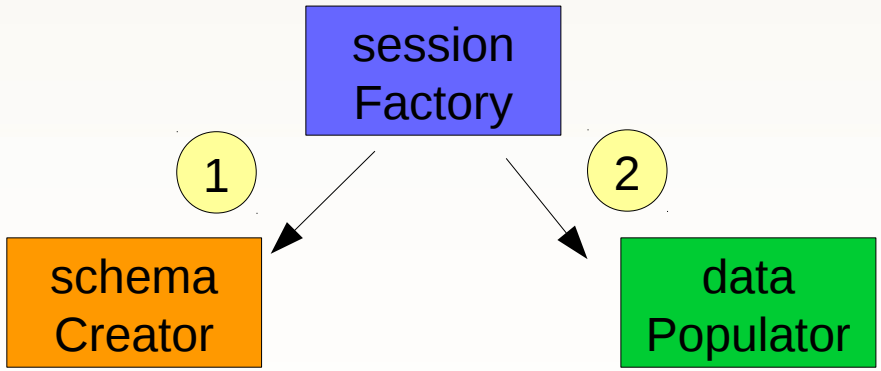
Depends-on Kabiliyeti

Bir bean'in başka bir bean'den **önce yaratılması**nı sağlar
Birden fazla bean belirtilebilir
Virgül, noktalı virgül veya boşluk ile ayrılırlar

```
<bean id="a" class="x.y.A" depends-on="b,c">
   ...
</bean>


<bean id="b" class="x.y.B" />

<bean id="c" class="x.y.C" />
```

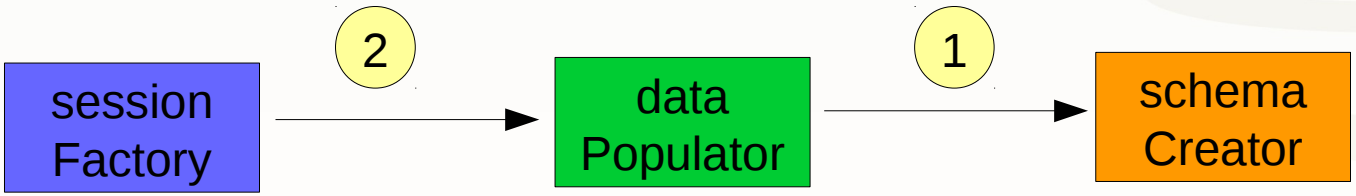
Depends-on Kabiliyeti

```
<bean id="sessionFactory"
class="x.y.LocalSessionFactoryBean"
      depends-on="schemaCreator,dataPopulator">
   ...
</bean>

<bean id="dataPopulator" class="x.y.DataPopulator" />

<bean id="schemaCreator" class="x.y.SchemaCreator" />
```



Depends-on Kabiliyeti

```
<bean id="sessionFactory"
class="x.y.LocalSessionFactoryBean"
      depends-on="schemaCreator,dataPopulator">
   ...
</bean>

<bean id="dataPopulator" class="x.y.DataPopulator"
      depends-on="schemaCreator"/>


<bean id="schemaCreator" class="x.y.SchemaCreator" />
```



Autowire Kabiliyeti

- Bağımlılıkların **Container tarafından otomatik olarak** enjekte edilmesidir
- Container, bean tanımlarının **constructor ve setter metotlarına bakarak** ihtiyaç duyulan bağımlılıkları otomatik biçimde enjekte eder
- Property ve constructor argümanların **explicit biçimde enjekte** edilmesine gerek kalmaz
- **Metadata bilgisi** geliştirme sürecinde sürekli **güncel** kalır

Autowire Kabiliyeti

```
public class Foo {
     private Bar bar;
     private Baz baz;

     public void setBar(Bar bar) {
          this.bar = bar;
     }

     public void setBaz(Baz baz) {
          this.baz = baz;
     }
}
```

```
<bean id="foo" class="x.y.Foo"
autowire="byType"/>

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

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

Autowire Kabiliyeti

```
public class Foo {
     private Bar bar;
     private Baz baz;

     public Foo(Bar bar, Baz baz) {
          this.bar = bar;
          this.baz = baz;
     }
}
```

```
<bean id="foo" class="x.y.Foo"
autowire="constructor"/>

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

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

Autowire Kabiliyeti

- İstenirse XML bean tanımında autowire özelliğinin yanında bağımlılıklar **explicit biçimde** enjekte edilebilir
- XML bean tanımında yapılan **explicit tanımlamalar** her zaman autowiring'i **override** eder

Autowire Kabiliyetinin Kısıtları

JAVA Eğitimleri

- **Dezavantajı,** uygulamanın yapısının tek bir noktadan görülemez hale gelmesidir
- **İlkel tip**teki property'ler **autowire edilemez**
- Birden fazla bean'ın autowire için uygun aday olma durumu **çok sık** ortaya çıkar
- Böyle bir durumda,
  - ya **bean ismi ile** hangi bean'in enjekte edileceği söylenir
  - ya da **aday bean'ler sınırlandırılarak** teke indirilir

Autowire Kabiliyeti

```
<bean id="foo" class="x.y.Foo"
autowire="byName"/>

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

<bean id="bar2" class="x.y.Bar"/>

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

Birden fazla bean'ın autowire için uygun aday olma durumunda kullanılır

Property ismi ile aynı isimde bean inject edilir

Autowire Adaylarını Sınırlandırmak

```
<bean id="foo" class="x.y.Foo"
autowire="byType"/>

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

<bean id="bar2" class="x.y.Bar"
autowire-candidate="false"/>

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

Bu bean autowire süreci dışında bırakılır

Lazy Bean Tanımları

- Default olarak beanlar **container başlatılırken** yaratılır
- Bu sayede konfigürasyon **hataları erkenden tespit** edilir
- Bu davranış uygun değil ise **lazy-init** attribute ile kapatılabilir
- Bu durumda bean, startup aşamasında değil, ilk **ihtiyaç duyulduğu anda** yaratılacaktır

Lazy Bean Tanımları

```
<bean id="foo" class="x.y.Foo" lazy-init
="true" />
```

Teker teker bean düzeyinde lazy özelliği yönetilebilir

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

```
<beans default-lazy-init="true">
...
</beans>
```

Container düzeyinde de bu özellik yönetilebilir

Bean özelinde bu davranışı yine değiştirmek mümkündür

Bean Scope Tanımları

- Bean tanımına göre oluşturulacak **nesnelerin ömürleri**ni kontrol etmek mümkündür
- Bu işleme “**scoping**” adı verilir
- Spring container tarafından built-in **desteklenen scope'lar** şunlardır:
  - Singleton,
  - prototype,
  - request,
  - session ve globalSession

Singleton

- Container'ın ömrü boyunca **tek bir instance** yaratılır
- **Default** scope'dur
- Nesnenin **yaşam döngüsü** Spring Container tarafından yönetilir
- **GoF Singleton** örüntüsünden **farklı**dır
  - GOF'da JVM veya ClassLoader genelinde tek instance vardır

Prototype

- ApplicationContext.**getBean**(“name”) çağrısı her seferinde yeni bir instance yaratır
- Nesne yaratıldıktan sonra Spring Container'ın **kontrolünden çıkar**
- Bir açıdan Spring Container, Java'daki **new operatörü**nün görevini devralmıştır

Scope Tanımlarına Örnekler

```
<bean id="foo" class="x.y.Foo"/>
```

Default singleton'dur
Container genelinde bu bean tanımından tek bir nesne oluşturulur

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

Explicit belirtmeye gerek yoktur

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

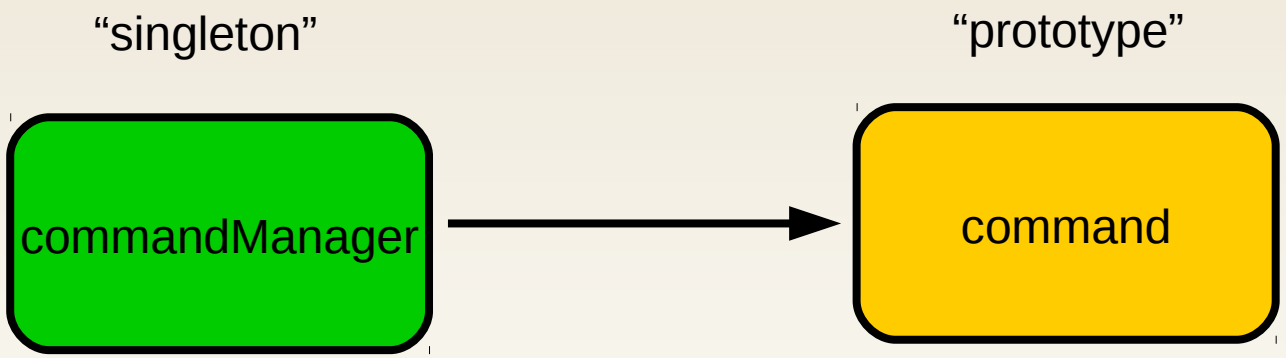
Her getBean("baz") çağrısı yeni bir instance yaratır

Singleton ve Prototype Nesneler Arasındaki Bağımlılıklar

JAVA Eğitimleri

- Singleton bean prototype bean'a bağımlı olabilir
- Bağımlılık **instantiation zamanı**nda çözülür
- Dolayısı ile singleton bean'ın ömrü boyunca **aynı prototype bean** kullanılır
- Başka bir ifade ile prototype bean'de singleton gibi davranmış olur

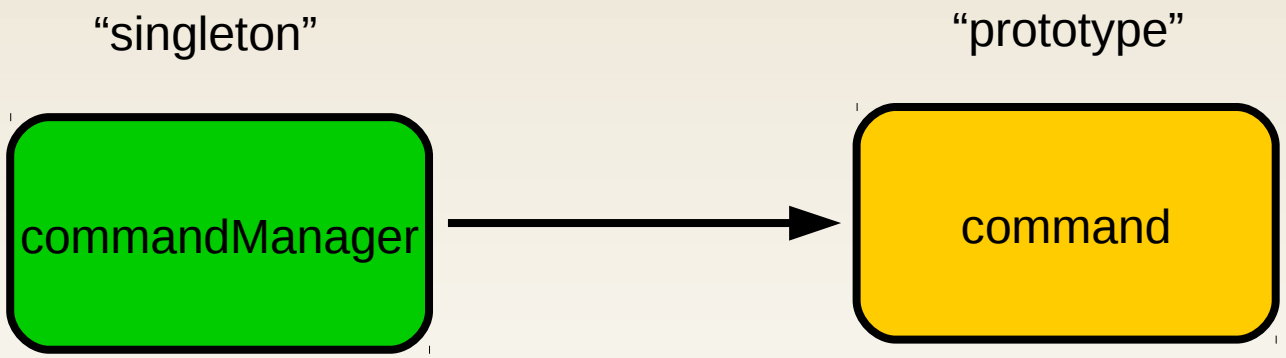
Singleton ve Prototype Nesneler Arasındaki Bağımlılıklar




```
executeCommand(Object state) {
        command.setState(state);
        command.execute();
}
```

Bazı durumlarda singleton bean içerisindeki her metot invokasyonunda prototype scope bean'den yeni bir instance'ın kullanılması gerekebilir

Singleton ve Prototype Nesneler Arasındaki Bağımlılıklar




```
executeCommand(Object state) {
        Command command = applicationContext
                                    .getBean(Command.class);
        command.setState(state);
        command.execute();
}
```

Böyle bir durumda prototype scope bean'i singleton scope bean'e doğrudan enjekte etmek yerine, çalışma zamanında her metot invokasyonunda prototype bean'e ApplicationContext ile lookup yaparak erişmek gerekir

Singleton ve Prototype Nesneler Arasındaki Bağımlılıklar

```
public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext appContext) {
        this.applicationContext = appContext;
     }

    public void executeCommand(Object state) {
        Command command = applicationContext.getBean(Command.class);
        command.setState(state);
        command.execute();
    }
}
```

CommandManager sınıfı **ApplicationContextAware** arayüzünü implement ettiği için bean yaratılırken **ApplicationContext** kendisini bean'a enjekte eder. Çalışma anında da **getBean()** metodu aracılığı ile farklı command instance'ları elde edebilir

Singleton ve Prototype Nesneler Arasındaki Bağımlılıklar

```
<beans...>
   <bean id="commandManager"
         class="x.y.CommandManager" scope="singleton"/>


   <bean id="command"
         class="x.y.Command" scope="prototype"/>
</beans>
```

Lifecycle Callback Metotları

- Spring Container bean'ler **yaratılıp bağımlılıkları enjekte edildikten sonra** bean'lerin bir takım metotlarını invoke edebilir
- Bu metotlara **initialization metotları** denir
- Benzer şekilde bean instance'larının **ömrü sonlandığı vakit** de başka bir takım metotlar invoke edilebilir
- Bunlara da **destruction metotları** denir

Lifecycle Callback Metotları: Bean Initialization

```
public class Foo {

    public void init() {
        // bean ile ilgili initialization islemi
yapilir...
    }
}
```

Herhangi bir arayüz implement etmeye Gerek yoktur

İnput argüman almayan herhangi bir metot initialization için kullanılabilir

```
<bean id="foo" class="x.y.Foo" init-method="init"/>
```

Bean instance yaratıldıktan ve bağımlılıkları enjekte edildikten sonra initialization için bu metot çağrılır

Lifecycle Callback Metotları: Bean Initialization

```
public class Bar implements InitializingBean {

    public void afterPropertiesSet() {
        // bean ile ilgili initialization islemi
yapilir...
    }
}
```

Initialization işlemi için diğer alternatif Spring'in InitializingBean arayüzünü implement etmektir

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

Lifecycle Callback Metotları: Bean Initialization

```
public class Foo {

    @PostConstruct
    public void init() {
        // init işlemleri...
    }

}
```

Lifecycle Callback Metotları: Bean Destroy

```
public class Foo {

    public void cleanup() {
        // connection release gibi destruction
    işlemleri yapılır...
    }
}
```

Herhangi bir arayüz implement etmeye Gerek yoktur

İnput argüman almayan herhangi bir metot destruction için kullanılabilir

```
<bean id="foo" class="x.y.Foo" destroy-method="cleanup"/>
```

Container kapatılırken veya bean instance Scope dışına çıktığında çağrılır
Prototype bean'lar için anlamlı değildir

Lifecycle Callback Metotları: Bean Destroy

```
public class Bar implements DisposableBean {

    public void destroy() {
        // connection release gibi destruction islemleri
yapilir...
    }
}
```

Destroy işlemi için diğer alternatif Spring'in DisposableBean Arayüzünü implement etmektir

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

Lifecycle Callback Metotları: Bean Destroy

```
public class Foo {

    @PreDestroy
    public void destroy() {
        // destruction işlemleri...
    }
}
```

@PostConstruct ve @PreDestroy'un Aktivasyonu

- **@PostConstruct** ve **@PreDestroy** JSR-250 annotasyonlarıdır
- **JSR-250**, Java SE için ortak annotationların belirlendiği bir spec'dir
- Spring Container içerisinde aktive olmaları için **`<context:annotation-config/>`** namespace elemanının tanımlı olması gerekir

Spring XML Şema Kabiliyeti

- Temel amaç **XML tabanlı** ApplicationContext konfigürasyonunu **kolaylaştırmak**tır
- Bean kavramının üstüne kuruludur
- Bean tanımlarını **daha kolay ve anlaşılır** biçimde yapmayı sağlar
- **Namespace konfigürasyonu** olarak da bilinir
- XML şema tanımları ile yapılabilen herşey **klasik bean tanımları** ile yine yapılabilir

Namespace Kabiliyeti Olmadan MVC Konfigürasyonuna Örnek

```
<bean name="handlerAdapter"
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHan
dlerAdapter">
<property name="webBindingInitializer" ref="webBindingInitializer" />
<property name="messageConverters" ref="messageConverters"/>
</bean>

<bean name="handlerMapping"
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHan
dlerMapping">
<property name="useSuffixPatternMatch" value="false" />
</bean>

<bean id="webBindingInitializer"
class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer"
>
<property name="conversionService" ref="conversionService"></property>
<property name="validator">
<bean
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean
" />
</property>
</bean>
```

Namespace Kabiliyeti Olmadan MVC Konfigürasyonuna Örnek

```
<bean id="messageConverters"
class="org.springframework.beans.factory.config.ListFactoryBean">
<property name="sourceList">
<list>
<bean
class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
<bean
class="org.springframework.http.converter.StringHttpMessageConverter" />
<bean
class="org.springframework.http.converter.ResourceHttpMessageConverter" />
<bean
class="org.springframework.http.converter.xml.SourceHttpMessageConverter" />
<bean
class="org.springframework.http.converter.xml.XmlAwareFormHttpMessageConverter"
/>
<bean
class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConvert
er" />
<bean
class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverte
r" />
</list>
</property>
</bean>
```

Namespace Kabiliyeti ile MVC Konfigürasyonu

Spring'in değişik modüllerine karşılık gelen çeşitli namespace elemanları vardır. Kullanılabilmeleri için konfigürasyon dosyasına XSD tanımının eklenmesi gerekir

```
<?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:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<mvc:annotation-driven/>

</beans>
```

JSR-303 validator, conversionService, httpMessageConverters gibi pek çok bean tanımını tek bir ifadeye indirger

Placeholder Kabiliyeti

- Bean tanımlarında ortama veya platforma göre **değişiklik gösteren değerleri** konfigürasyon **metadata'nın dışında bir yer**de yönetmeyi sağlar
- Örneğin JDBC bağlantı ayarlarını doğrudan bean tanımı içerisine koymak yerine bu değerler **bir properties dosyasından** yüklenebilir
- Ya da Properties dosyaları yanında ilaveten Java **system property**'leri de kullanılabilir

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>
```

${} ile belirtilmiş placeholder değerleri properties dosyalarından veya JVM sistem property'lerinden resolve edilir

Placeholder Kabiliyetinin Aktivasyonu

Aşağıdaki bean tanımının Spring Container içerisinde yapılması gerekir

```
<bean
class="org.springframework.beans.factory.config.Pro
pertySourcesPlaceholderConfigurer">
    <property name="locations"
value="classpath:application.properties"/>
</bean>
```

Placeholder değişkenleri locations ile belirtilen properties dosya(ları)dan çözümlenecektir

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=secret
```

application.properties

Namespace Kabiliyeti ile Placeholder Aktivasyonu

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

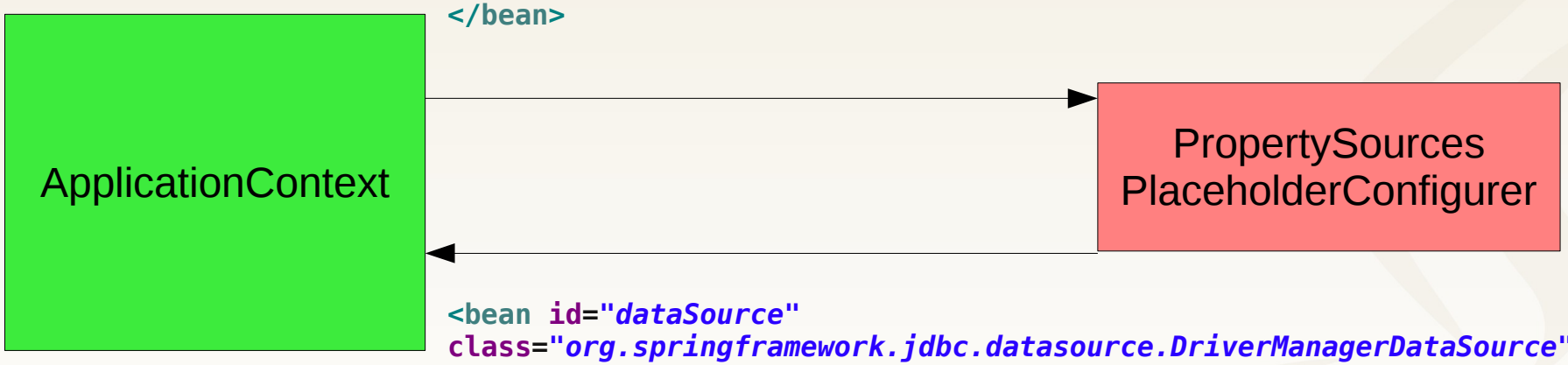
Bir önceki bean tanımına karşılık gelir. Context namespace'inin sağladığı bir kolaylıktır

Spring 3 ile birlikte önerilen ve doğru tanımlama yöntemi budur

Location attribute'una virgülle ayrılarak birden fazla properties dosyasının path'i yazılabilir

Placeholder Kabiliyeti Nasıl Çalışır?

```
<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>
```




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

Placeholder Kabiliyeti Ne Zaman Devreye Girer?

Konfigürasyon metadata bilgisi yüklendikten sonra bean'lar yaratılmadan evvel devreye girer

```
<bean id="foo" class="${foo.class}"/>
```

Dolayısı ile bean tanımlarındaki sınıf attribute'ları da properties'den yüklenebilir

```
foo.class=x.y.FooImpl
```

application.properties

Değişkenlerin Ortama Göre Yüklenmesi

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

targetPlatform bizim tanımladığımız bir değişkendir, değerleri de bizim tarafımızdan tanımlanmıştır

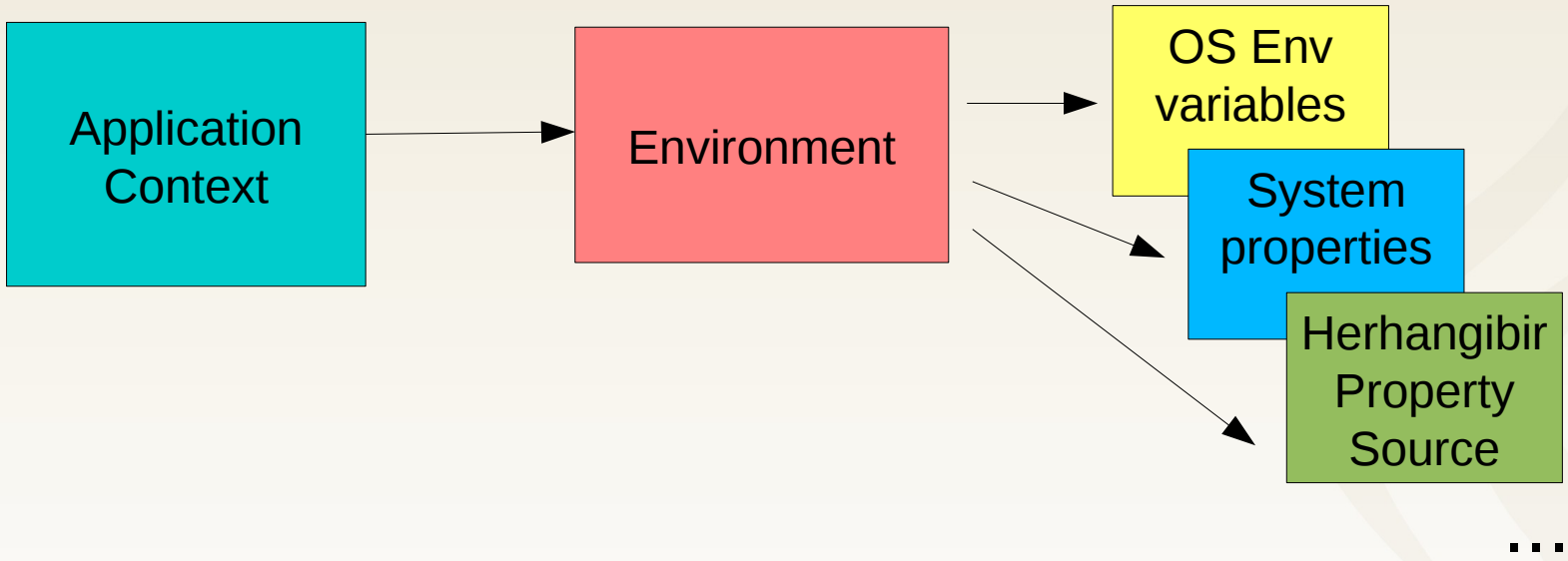
-DtargetPlatform=dev|test|prod gibi JVM system property ile uygulamaya verilebilir, yada **OS env değişkeni** olarak tanımlanabilir

Spring runtime'da **application-dev.properties**, **application-test.properties** veya **application-prod.properties** dosyalarından uygun olanını yükler

Spring ve Environment Kabiliyeti

- Spring 3 ile birlikte ApplicationContext'e **Environment** isimli bir soyutlama eklenmiştir
- Environment **uygulamanın çalıştığı ortamı** ifade etmektedir
- Environment üzerinden **property değerlerinin çözümlemesi** çok daha esnek bir hal almıştır

Spring ve Environment Kabiliyeti




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

...

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

Spring ve Bean Profile Kabiliyeti

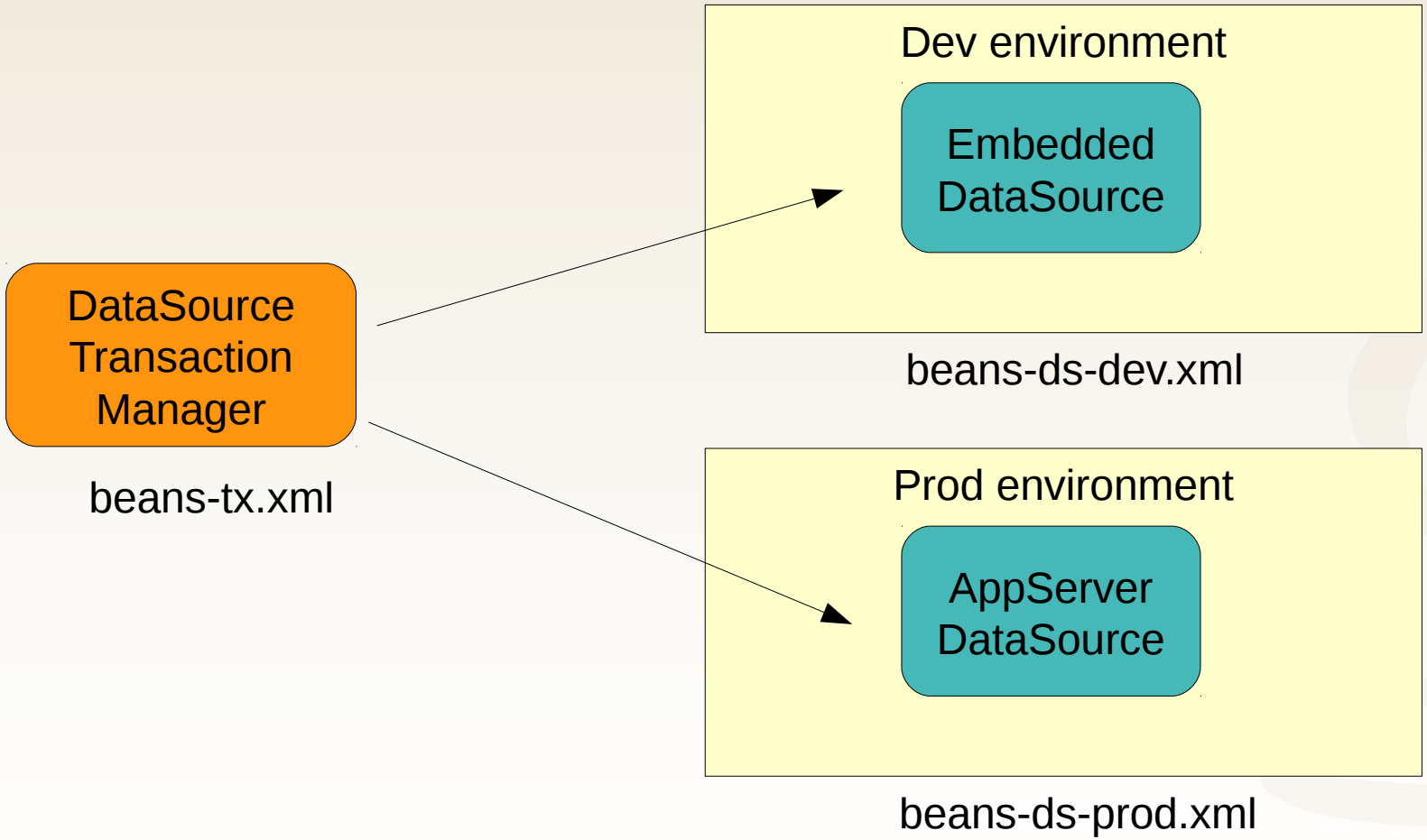
- Bazı durumlarda **farklı platformlara veya ortamlara farklı bean tanımları** yapmak gerekebilir
- Örneğin **dev ortamında** geliştirme sürecini kolaylaştıracak test amaçlı **embedded bir dataSource** bean'i tanımlanabilir
- **Prod ortamında** ise uygulama sunucusunda tanımlamış **connection pool kabiliyetine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

Spring 3 Öncesinde Platforma Özgü Bean Tanımları

JAVA Eğitimleri

- **Spring 3 öncesinde** böyle bir ihtiyaç, bean tanımlarını farklı konfigürasyon dosyalarında yapıp, **platforma uygun konfigürasyon dosyalarını** placeholder yardımı ile **import ederek** çözülmekteydi

Spring 3 Öncesinde Platforma Özgü Bean Tanımları



Spring 3 Öncesinde Platforma Özgü Bean Tanımları

```
<beans...>

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

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

</beans>
```

Spring 3 ve Bean Profile Kabiliyeti

- 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

JAVA Eğitimleri

```
<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>
```

Spring 3 ve Bean Profile Kabiliyeti

- 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

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

İletişim

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