

Spring Application Framework Overview 1



Spring Nedir?

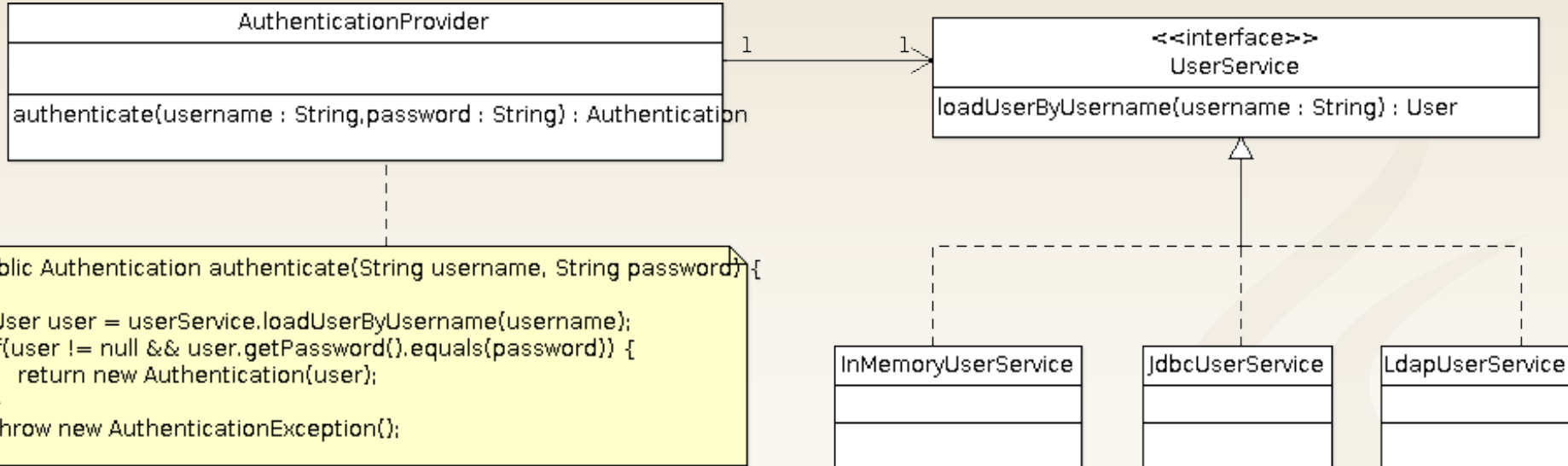
- Kurumsal Java uygulamalarını
 - kolay,
 - hızlı,
 - test edilebilir

biçimde geliştirmek ve Tomcat, Jetty, Resin gibi her tür ortamda çalıştırabilmek için ortaya çıkmış bir “**framework**”tür

Spring'i Öne Çıkaran Özellikler

- **POJO** tabanlı bir programlama modeli sunar
- “**Program to interface**” yaklaşımını temel ilke kabul etmiştir
- **Test edilebilirlik** her noktada **ön plandadır**
- **Modüler** bir **framework**tür, sadece ihtiyaç duyulan modüller istenilen kapsamda kullanılabilir

Program to Interface Yaklaşımı Nedir?



- Sınıflar arasındaki bağımlılıklar **arayüz ve soyut sınıflara** doğrudur
- Farklı ortamlardaki **farklı davranışlar** daha kolay biçimde ele alınabilir
- Bu sayede sınıfların **test edilebilirliği** de artmaktadır
- Testlerde **mock ve stub nesneler** daha rahat kullanılabilir

Spring IoC Container'a Doğru Yolculuk

Nesne bağımlılıklarını kendi içerisinde yaratır. Bu durumda farklı platformlar için farklı gerçekleştirmelerin yaratılması ve kullanılması problem olacaktır

```
public class AuthenticationProvider {  
    private UserService userService =  
        new JdbcUserService(new JdbcDataSource());  
    private RoleService roleService = new RoleServiceImpl();  
  
    public Authentication authenticate(  
        String username, String password)  
        throws AuthenticationException {  
        User user = userService.loadUserByUsername(username);  
        if(user != null && user.getPassword().equals(password)) {  
            List<Role> roles =  
                roleService.findRolesByUsername(username);  
            return new Authentication(user, roles);  
        }  
        throw new AuthenticationException("Authentication failed");  
    }  
}
```

Spring IoC Container'a Doğru Yolculuk

Nesne içerisinde basit bir factory metot içerisinde farklı platformlar için farklı gerçekleştirmeleri oluşturma işlemi ele alınmaya çalışılır. Tabi bu bağımlılıkların da kendilerine ait bağımlılıkları vardır. Bunlarında ele alınması gerekecektir.

```
public class AuthenticationProvider {  
    private UserService userService = createUserService();  
    private RoleService roleService = new RoleServiceImpl();  
  
    private UserService createUserService() {  
        String targetPlatform =  
            System.getProperty("targetPlatform");  
        if("dev".equals(targetPlatform)) {  
            return new InMemoryUserService();  
        } else if("test".equals(targetPlatform) ||  
            "prod".equals(targetPlatform)) {  
            return new JdbcUserService(new JdbcDataSource());  
        } else {  
            return new LdapUserService(new LdapTemplate(  
                new LdapContextSource()));  
        }  
    }  
}
```

...

Spring IoC Container'a Doğru Yolculuk

Bağımlılıkları oluşturma ve platforma göre yönetme işi ayrı bir sınıfa çekilir. ServiceLocator isimli bu sınıf uygulama genelinde ihtiyaç duyulan bağımlılıklara erişim sağlar

```
public class ServiceLocator {
```

```
    private static final ServiceLocator  
        INSTANCE = new ServiceLocator();
```

```
    public static final ServiceLocator getInstance() {  
        return INSTANCE;  
    }
```

```
    private UserService userService;  
    private RoleService roleService;  
    private DataSource dataSource;
```

```
    ...
```

```
}
```

ServiceLocator genellikle singleton olur ve bootstrap esnasında target platforma göre (dev,test veya prod) servis nesnelerini genellikle reflection api ile yaratarak kullanıma hazır hale getirir

Spring IoC Container'a Doğru Yolculuk

targetPlatform'a göre property dosyalarından yüklenen sınıf bilgileri Reflection API kullanılarak ilgili servis nesnelerine dönüştürülür

```
public class ServiceLocator {  
    private ServiceLocator() {  
        try {  
            String targetPlatform = System.getProperty("targetPlatform");  
  
            InputStream is = this.getClass().getClassLoader()  
                .getResourceAsStream("service." + targetPlatform + ".properties");  
  
            Properties properties = new Properties();  
            properties.load(is);  
  
            String usClassName = properties.getProperty("userService");  
            String rsClassName = properties.getProperty("roleService");  
            String dsClassName = properties.getProperty("dataSource");  
  
            userService = (UserService) Class.forName(usClassName).newInstance();  
            roleService = (RoleService) Class.forName(rsClassName).newInstance();  
            dataSource = (DataSource) Class.forName(dsClassName).newInstance();  
        } catch (Exception ex) {  
            throw new ServiceLocatorException();  
        }  
    }  
    ...  
}
```


Spring IoC Container'a Doğru Yolculuk

Servis nesnelerini oluşturacak sınıfların FQN
service.dev.properties gibi classpath'deki dosyalarda
yönetilir

```
userService=com.javaegitimleri.example.JdbcUserService  
roleService=com.javaegitimleri.example.RoleServiceImpl  
dataSource=org.h2.jdbcx.JdbcDataSource
```

```
public class ServiceLocator {
```

```
...
```

```
public final UserService getUserService() {  
    return userService;  
}
```

```
public RoleService getRoleService() {  
    return roleService;  
}
```

```
public DataSource getDataSource() {  
    return dataSource;  
}
```

```
}
```

Servisler getter metotlar üzerinden erişilebilirler. bazı ServiceLocator varyasyonlarında **getService(String serviceName)** gibi genel bir metot üzerinden de servislere erişim sağlanabilmektedir

Spring IoC Container'a Doğru Yolculuk

```
public class AuthenticationProvider {
```

```
    private UserService userService =  
        ServiceLocator.getInstance().getUserService();
```

```
    private RoleService roleService =  
        ServiceLocator.getInstance().getRoleService();
```

```
    ...  
}
```



Nesneler ihtiyaç duydukları bağımlılıkları **ServiceLocator** üzerinden elde ederler

Spring IoC Container'a Doğru Yolculuk

```
public class JdbcUserService implements UserService {
```

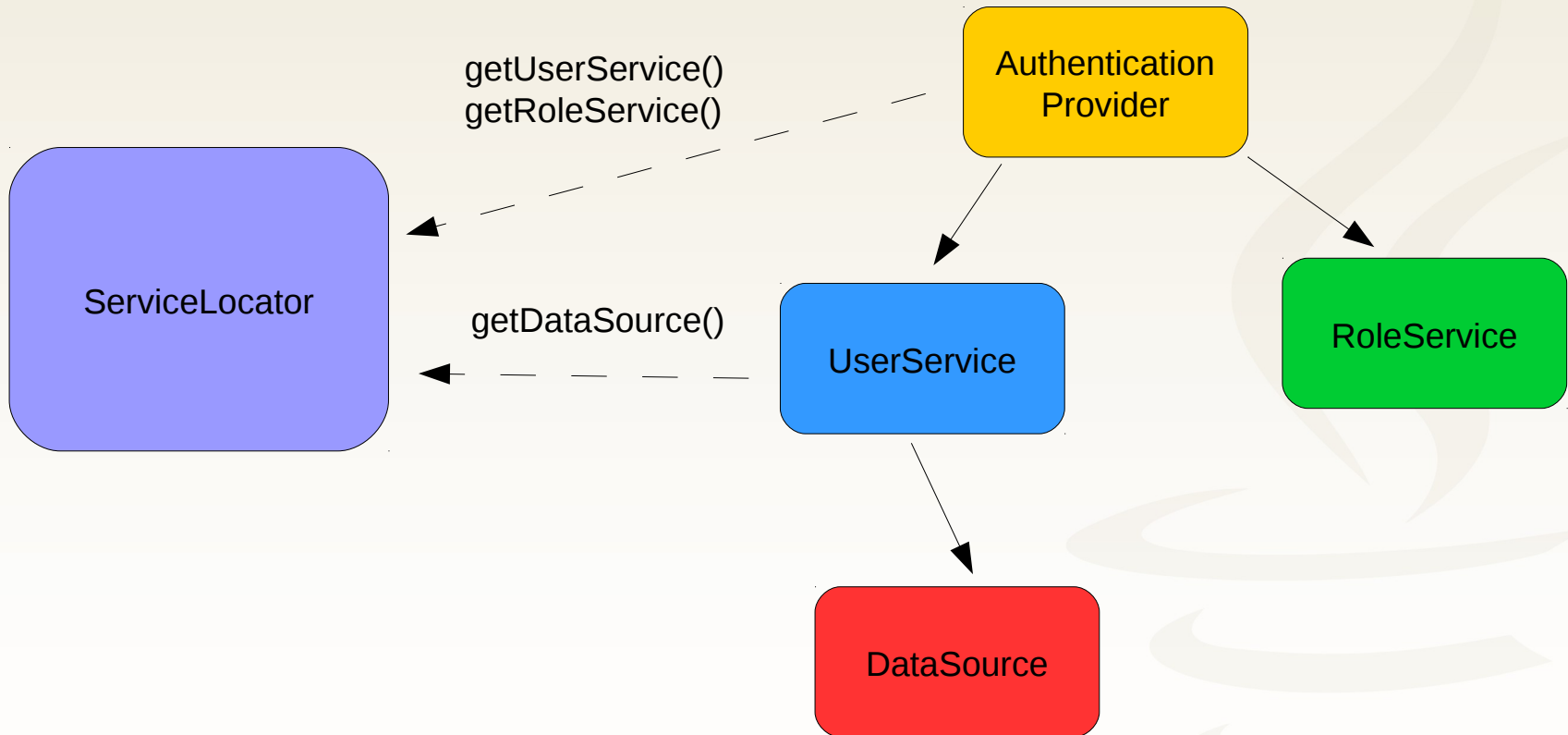
```
    private DataSource dataSource =  
        ServiceLocator.getInstance().getDataSource();
```

```
    ...  
}
```

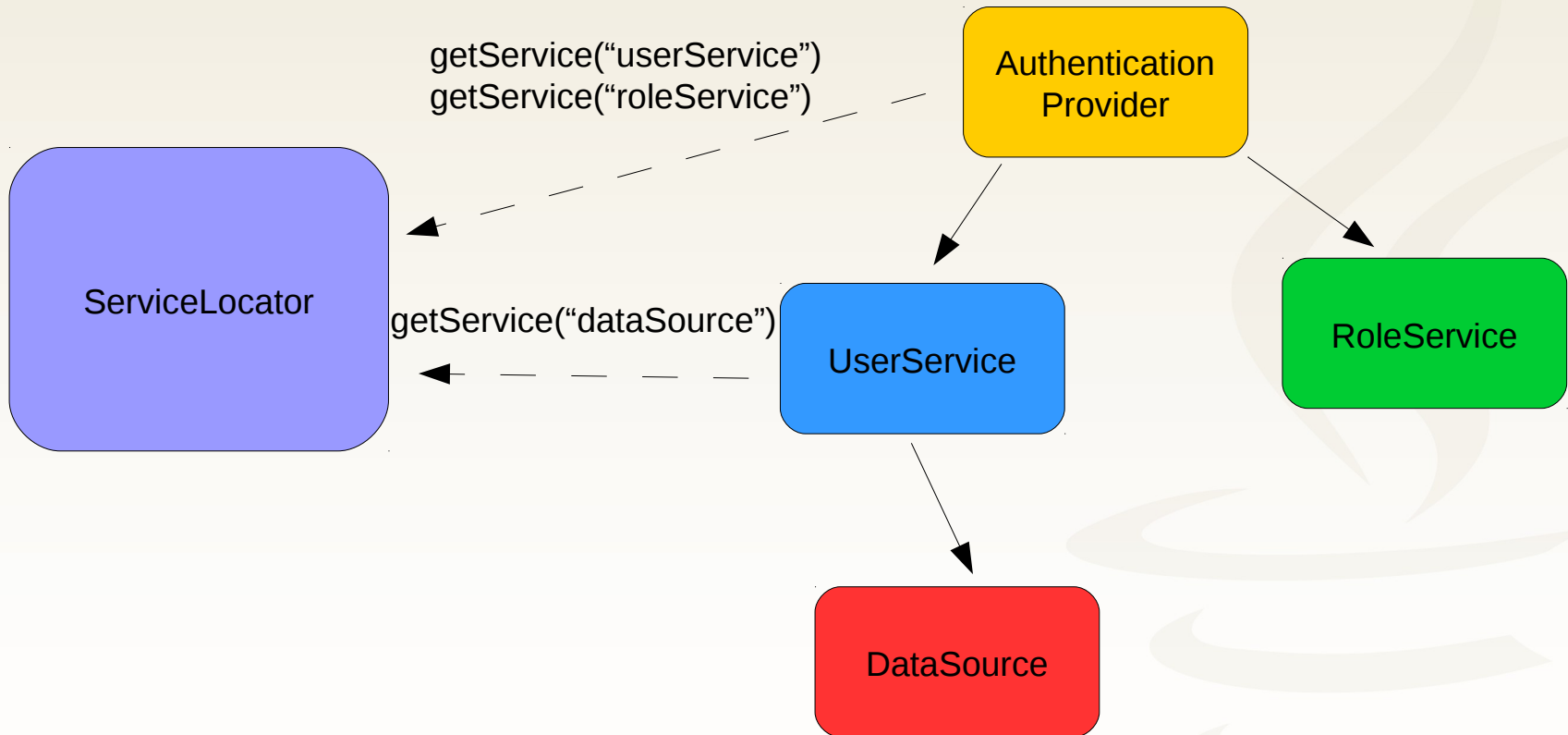


Bağımlılık hiyerarşisindeki bütün diğer nesneler de kendi bağımlılıklarını aynı şekilde ServiceLocator üzerinden çözümlerler

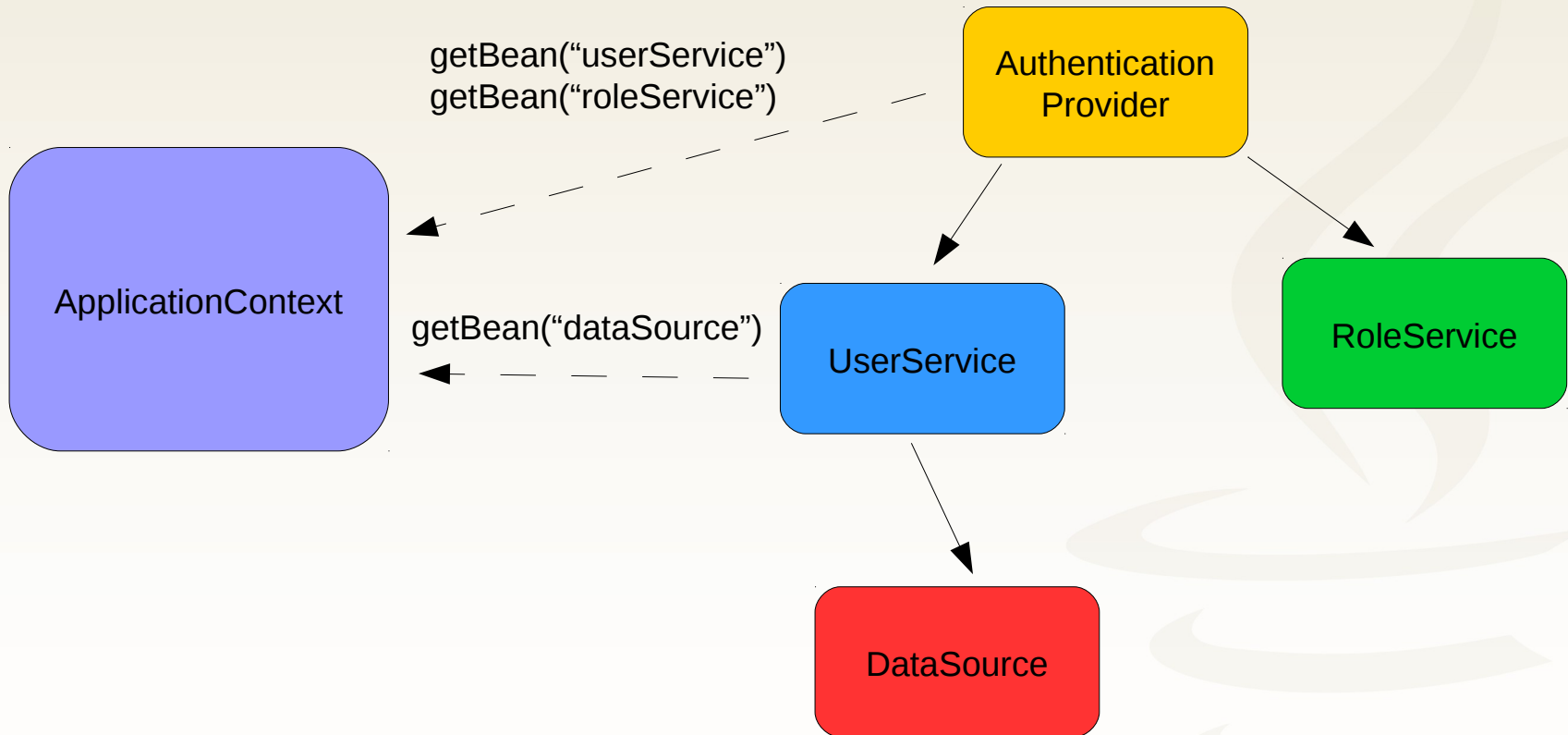
Spring IoC Container'a Doğru Yolculuk



Spring IoC Container'a Doğru Yolculuk



Spring IoC Container'a Doğru Yolculuk



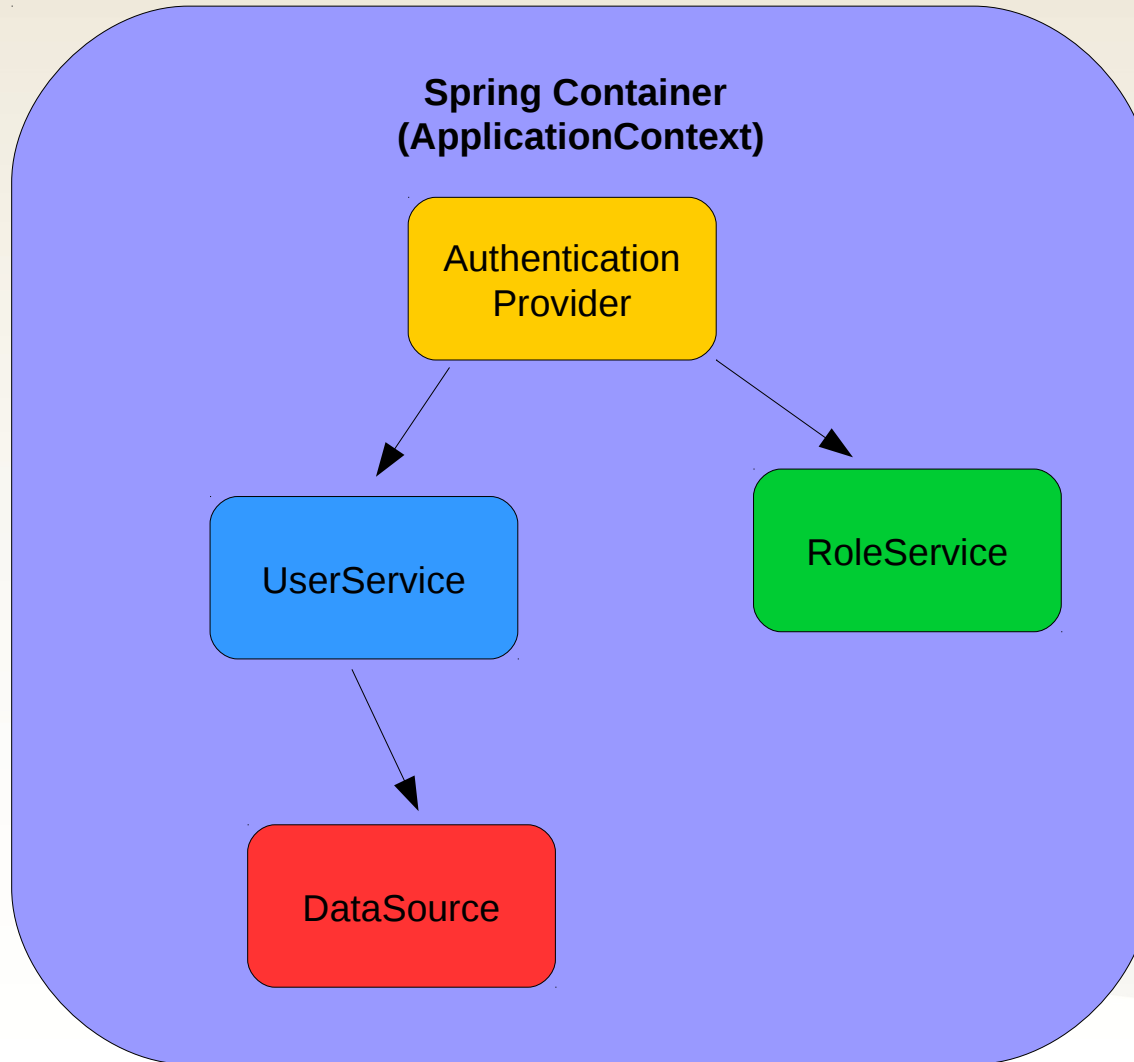
Spring IoC Container ve Dependency Injection

- Spring nesneleri oluşturma, bir araya getirme ve yönetme işine ServiceLocator'dan çok daha sistematik ve kapsamlı bir yol sunmaktadır
- Bağımlılıkları oluşturma ve yönetme işi nesnenin kendi içinden çıkıp, **Spring Container'a geçmektedir**
- Nesneler bağımlılıkların hangi **concrete sınıflarla** sağlandıklarını bilmezler

Spring IoC Container ve Dependency Injection

- Bağımlı olunan nesnelerin kim tarafından oluşturulduğu, nereden geldiği de bilinmez
- **Bağımlılıkların yönetimi nesnelerden container'a geçmiştir**
- Spring IoC container tarafından yönetilen nesnelere “**bean**” adı verilir
- Spring bean'lerinin **sıradan Java nesnelerinden** hiç bir farkı yoktur

Spring IoC Container ve Dependency Injection

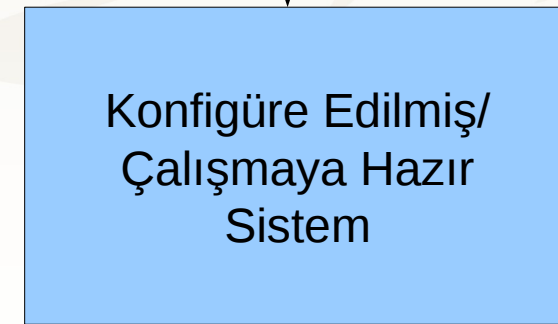
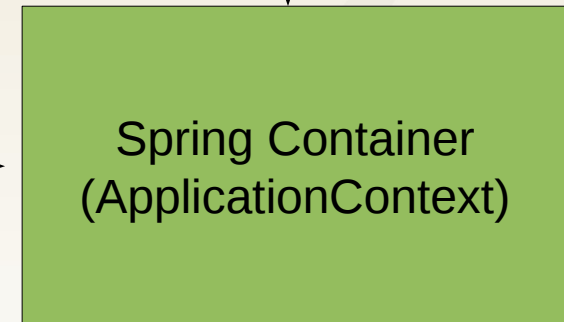


Spring IoC Container ve Konfigürasyon Metadata

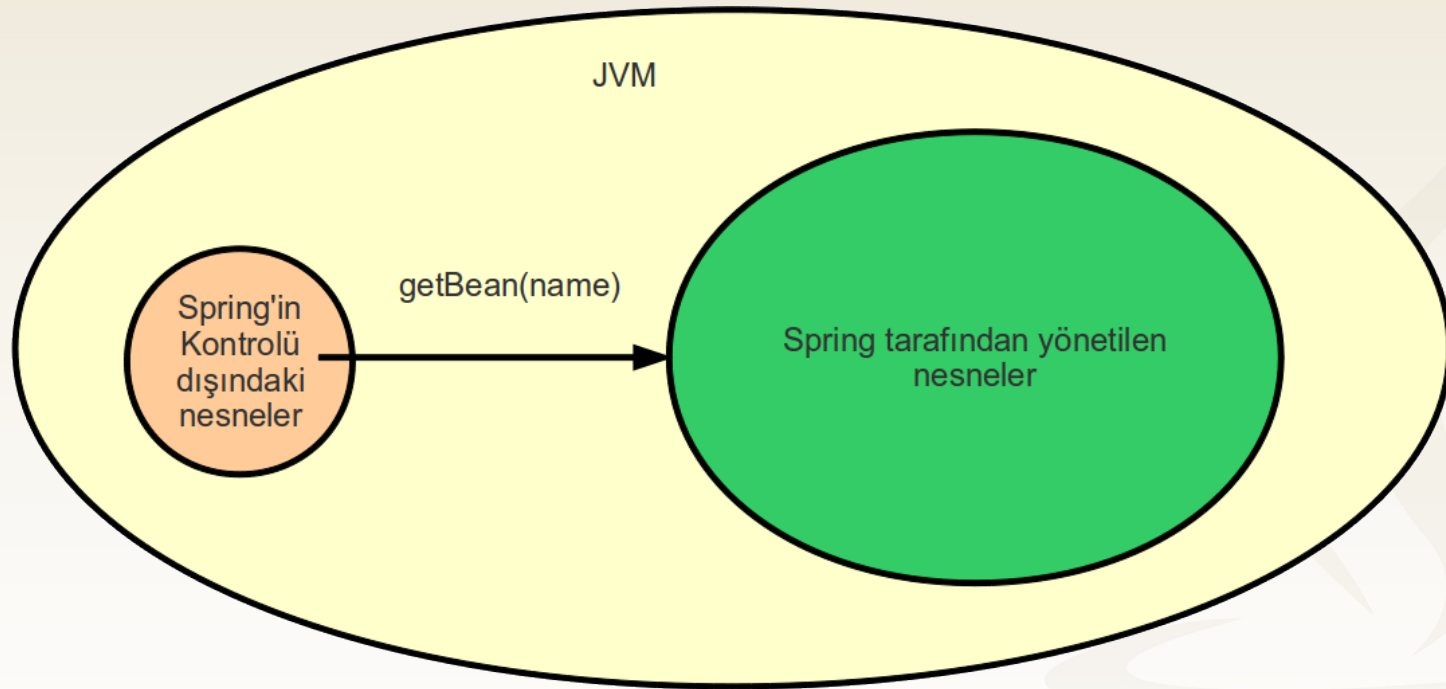
- Bean'lerin yaratılması, bağımlılıkların enjekte edilmesi için Spring Container **bir bilgiye** ihtiyaç duyar
- Bean ve bağımlılık tanımları **konfigürasyon metadata'sını** oluşturur

Konfigürasyon
Metadata
(Bean Tanımları)
XML
Annotation
Java Class
→

↓ Sınıflar



JVM ve Spring Container İlişkisi



JVM içerisinde hedefimiz mümkün olduğunca fazla sayıda nesnenin Spring Container tarafından yönetilmesidir, böylece bu nesneler Spring'in sunduğu kabiliyetlerden yararlanabilirler. Ancak JVM içerisinde Spring Container tarafından yönetilemeyecek nesneler de olacaktır. Spring tarafından yönetilmeyen nesnelerde ApplicationContext lookup yaparak Spring Container'daki nesnelere erişilebilir

ApplicationContext'in Yaratılması ve Kullanımına Örnek

1

ApplicationContext yaratılır , Container bu aşamadan itibaren kullanıma hazırdır

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

2

```
PetClinicService service = context.getBean(  
    "petClinicService", PetClinicService.class);
```

Bean lookup ile ilgili bean'a erişilir

3

```
List vets = service.getVets();
```

Artık bean'ler uygulama içerisinde istenildiği gibi kullanılabilir

XML Tabanlı Konfigürasyon Nasıl Yapılır?

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

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

Bar bar = new Bar();

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

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

```
...
```

```
</beans>
```

Constructor Injection

```
<bean id="bar" class="x.y.Bar"/> Bar bar = new Bar();
```

```
<bean id="baz" class="x.y.Baz"/> Baz baz = new Baz();
```

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

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

    }
}
```

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

Setter Injection

`<bean id="bar" class="x.y.Bar"/>` —→ `Bar bar = new Bar();`

`<bean id="baz" class="x.y.Baz"/>` —→ `Baz baz = new Baz();`

`<bean id="foo" class="x.y.Foo">` —→ `Foo foo = new Foo();`

`<property name="bar" ref="bar"/>` —→ `foo.setBar(bar);`

`foo.setBaz(baz);`

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

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

Bean Oluşturma Yöntemleri: FactoryBean



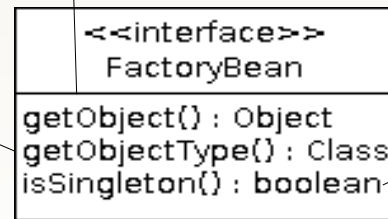
```
<bean id="foo" class="x.y.z.FooFactoryBean"/>
```

Bean instance'ını döner

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

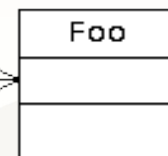
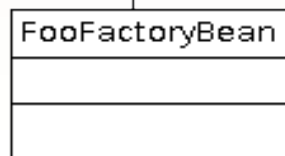
Oluşturulacak
Bean instance'ın
sınıfını döner

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



Bean scope'unun singleton
olup olmadığını belirtir

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



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

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

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

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

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

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

Property ismi ile aynı isimde bean inject edilir

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

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

Lifecycle Callback

Metotları: Bean Initialization

`public class Foo` { → Herhangi bir arayüz implement etmeye Gerek yoktur

```
    public void init() {  
        // bean ile ilgili initialization işlemi  
        yapılır...  
    }  
}
```

→ Input 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 işlemi  
        yapılır...  
    }  
}
```

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 { → Herhangi bir arayüz implement etmeye
Gerek yoktur

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

→
Input 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  
        yapılır...  
    }  
}
```

↓

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

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

Lifecycle Callback Metotları: Bean Destroy

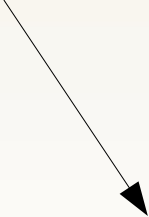
```
public class Foo {  
  
    @PreDestroy  
    public void destroy() {  
        // destruction işlemleri...  
    }  
}
```

@PostConstruct ve @PreDestroy'un Aktivasyonu

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

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

```
</beans>
```



@PostConstruct, @PreDestroy, @Required, @Autowired gibi
annotasyonları devreye sokar

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

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

Placeholder Kabiliyetinin Aktivasyonu

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

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



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

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

- 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

