2. Spring Core

3. Spring IoC Container

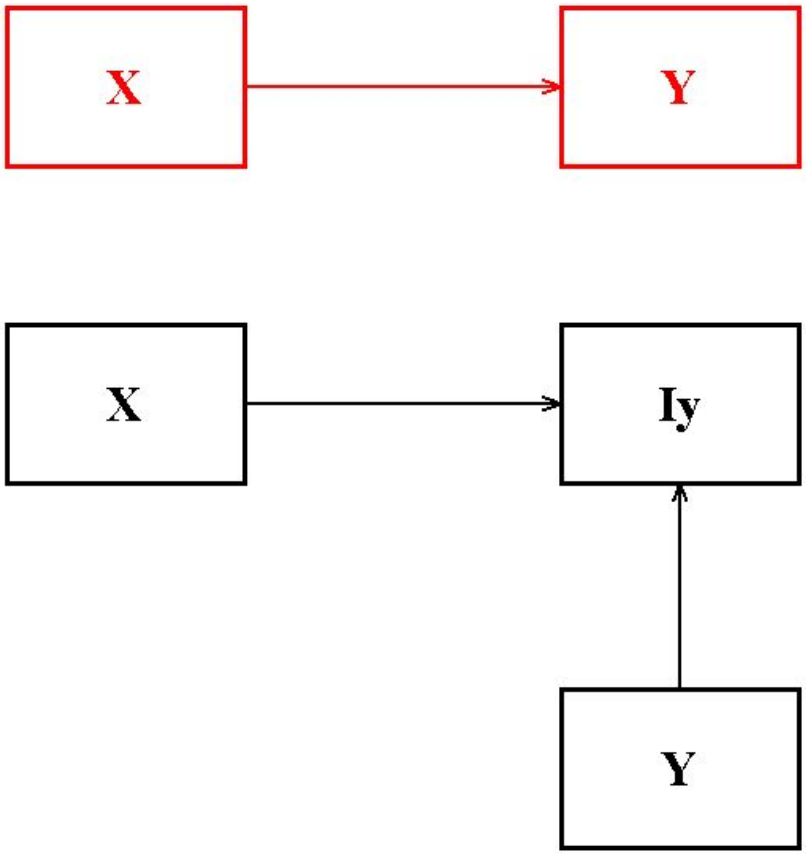
The Spring Container

- The container will:
  - create the objects
  - wire them together
  - configure them
  - manage their complete lifecycle from creation till destruction.
- These objects are called Spring Beans

Dependency Injection

- The Spring container uses dependency injection (DI) to manage the components
- In a complex Java application classes should be as independent as possible to increase the possibility to reuse these classes and to test them independently
- Dependency Injection helps in connecting these classes together and same time keeping them independent.

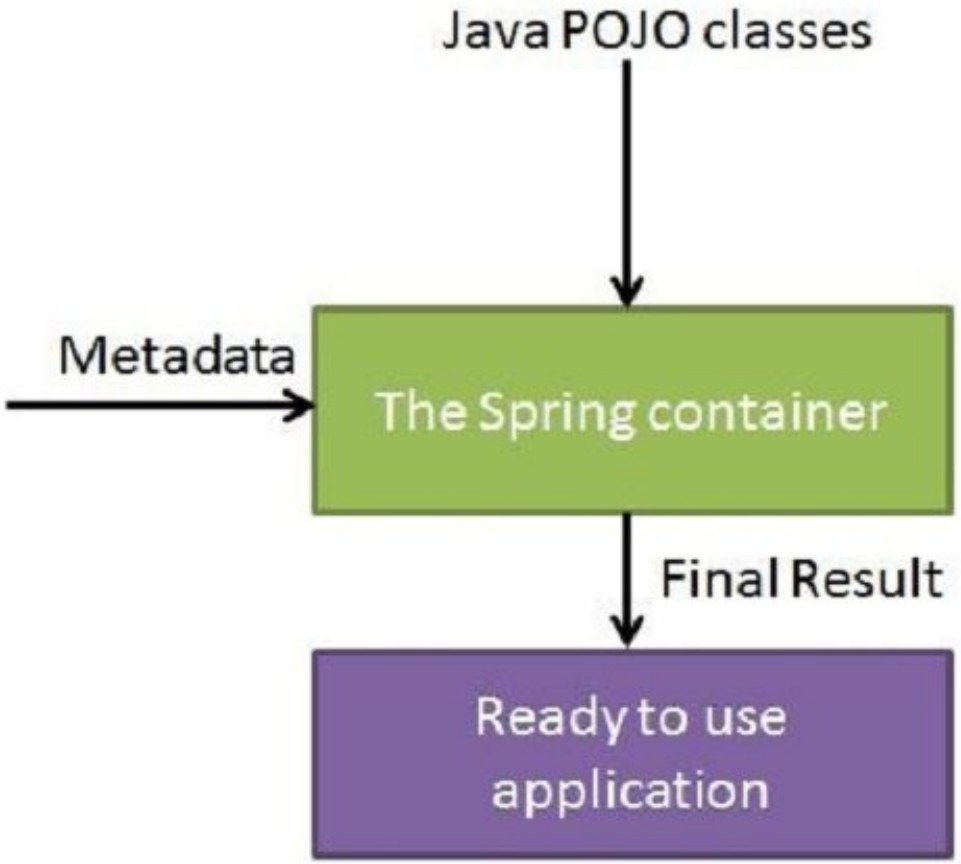
DI Implementation



Container's Metadata

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided:
  - by XML
  - Java annotations
  - Java code

How Spring Works



Spring Bean Definition

- The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans
- The bean definition contains the information called configuration metadata which is needed for the container to know:
  - How to create a bean
  - Bean's lifecycle details
  - Bean's dependencies

XML Metadata

```
<?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 declarations go here -->
</beans>
```

Spring Bean Definition

- Each bean definition can contain a set of the following properties:
  - Class
  - Name
  - Scope
  - constructor-arg
  - Properties
  - autowiring mode
  - lazy-initialization mode
  - initialization method
  - destruction method

Bean Definition Attributes

- Class attribute is mandatory and specify the bean class to be used to create the bean
- Name attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s)
- Properties is used to inject the dependencies

Spring Tool Suite Installation

1. Open Eclipse -> Help -> Eclipse Marketplace…
2. Find = STS -> click Find button -> select STS for Eclipse Luna -> Install button -> Confirm -> I accept -> Finish
3. Restart Eclipse

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

Example 1. Create Spring Project

- File -> New -> Other.. -> Spring -> Spring project -> Next
- Project name = P211BeanDefinition, Templates = Simple Spring Utility Project -> Next
- Package = com.bionic.edu -> Finish

Example 1. Create Spring Project

- Remove from com.bionic.edu package (src/main/java) template files Service.java and ExampleService.java
- Remove from com.bionic.edu package (src/test/java) template files ExampleConfigurationTests.java and ExampleServiceTests.java

Example 1. Logging Dependencies

Group Id	Artifact Id	Version
org.slf4j	slf4j-api	1.7.5
org.apache.logging.log4j	log4j-api	2.0.2
org.apache.logging.log4j	log4j-core	2.0.2
org.apache.logging.log4j.adapters	log4j-slf4j-impl	2.0-beta4
org.apache.logging.log4j	log4j-taglib	2.1
org.apache.logging.log4j	log4j-jcl	2.0.2

*

Add Dependencies to the pom.xml File

1. Open pom.xml -> Dependencies tab -> select junit: 3.8.1[test] -> Properties -> change version to 4.7 -> Ok
2. Press Add button -> GroupId = org.slf4j, ArifactId = slf4j-api, Vesion = 1.7.5 -> Ok
3. Press Add button -> GroupId = org.apache.logging.log4j, ArtifactId = log4j-api, Version = 2.0.2 -> Ok
4. And so on… Then -> Save

Further pom.xml Tuning

- Select log4j (1.2.14) dependency and click Remove button
- Go to pom.xml tag and change 1.5 Java version to 1.8 both in <source> and <target> tags of maven-compiler-plugin artefact
- Change Spring version to 4.1.1.RELEASE in <spring.framework.version> tag
- Save pom.xml file

Maven clean&install actions

1. Right click on the project's name -> Maven -> Update project -> Ok
2. Right click on the project's name -> Run As… -> Maven clean
3. Right click on the project's name -> Run As… -> Maven install

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

Example 1. log4j2.xml in resources

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="WARN">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
             %logger{36} - %msg%n"/>
        </Console>
    </appenders>
    <loggers>
        <root level="warn">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>
```

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

E1. GreetingService Interface

```
package com.bionic.edu;

public interface GreetingService {
    void sendGreeting();
}
```

E1. HelloWorldService Class

```
package com.bionic.edu;

public class HelloWorldService implements
        GreetingService {
    public void sendGreeting() {
     System.out.println("Hello, world!");
    }
}
```

E1. HelloKittyService Class

```
package com.bionic.edu;

public class HelloKittyService implements
        GreetingService {
    public void sendGreeting(){
      System.out.println("Hello, Kitty!");
    }
}
```

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

Bean Definition

- The <bean> element tells Spring to create an object for you.
- The id attribute gives the bean a name by which it'll be referred to in the Spring container.
- When the Spring container loads its beans, it'll instantiate the bean using the default constructor.

Example 1. Bean Definition

```
<bean id="helloWorldService"
class="com.bionic.edu.HelloWorldService" />
```

Example 1. Configuration File

- Right click on src/main/resources -> New -> File
- Fill File name with configuration file Id (beans.xml) -> Finish
- Create configuration file context (see next slide) -> Save

Example 1. Configuration File

```
<?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="helloWorldService"
class="com.bionic.edu.HelloWorldService" />

</beans>
```

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

Application Context

- You can load the Spring application context using the following code:

```
ApplicationContext ctx = new
    ClassPathXmlApplicationContext("beans.xml");
GreetingService service =
    (GreetingService)ctx.getBean("helloWorldService");
service.sendGreeting();
```

Example 1. Application Class

```
package com.bionic.edu;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        GreetingService service =
                (GreetingService)ctx.getBean("helloWorldService");
        service.sendGreeting();
    }
}
```

Example 1. Hello World in Spring

1. Create Spring project
2. Tune project for logging
3. Create project classes (POJOs)
4. Create Spring metadata
5. Create application context
6. Run application

Example 1.

- Run application. You will get

  Hello, world!

- See P211BeanDefinition project for the full text

Injecting through Constructors

- A class can be constructed in two different ways:
  - Using the default constructor
  - Using a constructor that takes an argument(s)
- If no <constructor-arg> are given in a bean definition, the default constructor is used.
- A <constructor-arg> with a value attribute leads to the other constructor will be used instead.

Example 2. HelloWorldService

```
package com.bionic.edu;
public class HelloWorldService implements GreetingService
{
    public String message;
    public HelloWorldService(){ message = ""; }
    public HelloWorldService(String message){
     this.message = message;
    }
    public void sendGreeting() {
     System.out.println("Hello, world! " + message);
    }
}
```

Example 2. Bean Definition

```
<bean id="helloWorldService"
class="com.bionic.edu.HelloWorldService">
    <constructor-arg value="I am Victor." />
 </bean>
```

Example 2.

- Run application. You will get

  Hello, world! I am Victor.

- See P212ConstructorInjection project for the full text

Injecting Object References

- You should use ref attribute in a <constructor-arg> for passing references to other beans

Example 3. Application Class

```
public class Application {
    GreetingService greeting = null;
    public Application(){}
    public Application(GreetingService greeting){ this.greeting = greeting; }

    public static void main(String[] args) {
     ApplicationContext ctx = new
                ClassPathXmlApplicationContext("beans.xml");
     Application application = (Application)ctx.getBean("application");
     application.start();
     }
     public void start(){
     if (greeting != null) greeting.sendGreeting();
    } }
```

Example 3. Bean Definition

```
<bean id="helloWorldService"
        class="com.bionic.edu.HelloWorldService">
    <constructor-arg value="I am Victor." />
</bean>
<bean name="application" class="com.bionic.edu.Application">
    <constructor-arg ref="helloWorldService" />
</bean>
```

Example 3 Output

- Hello, world! I am Victor.

Example 3. Bean Definition

```
<bean id="helloWorldService"
        class="com.bionic.edu.HelloWorldService">
    <constructor-arg value="I am Victor." />
</bean>
<bean id="helloKittyService"
        class="com.bionic.edu.HelloKittyService" />
<bean name="application" class="com.bionic.edu.Application">
    <constructor-arg ref="helloKittyService" />
</bean>
```

Example 3 Output

- Hello, Kitty!
- See P213ConstructorInjection project for the full text

Property Tags

- You can use <property> tag to pass the values of different variables used at the time of object creation
- <property> is similar to <constructor-arg> in many ways, except that instead of injecting values through a constructor argument, <property>injects by calling a property's setter method.

Example 4. Simple Value Injection

1. Add accessor to HelloWorldService class
2. Change beans.xml as follows

```
<bean id="helloWorldService" class="com.bionic.edu.HelloWorldService">
   <property name="message" value="I am Victor." />
</bean>
<bean id="helloKittyService" class="com.bionic.edu.HelloKittyService" />
<bean name="application" class="com.bionic.edu.Application">
        <constructor-arg ref="helloWorldService" />
</bean>
```

Example 4. Output

- Hello, world! I am Victor.

- See P214PropertySimple project for the full text

Simple Value Injection

- <property> isn’t limited to injecting String values
- The value attribute can also specify numeric (int, float, java.lang.Double, and so on) values as well as boolean values

Example 5. Numeric Injection

```
public class HelloWorldService implements GreetingService {
    public String message;
    public int repeat;

   // constructors, getters&setters

    public void sendGreeting() {
        for (int i = 0; i < repeat; i++){
        System.out.println("Hello, world! " + message);
        }
    }}
```

Example 5. beans.xml

```
<bean id="helloWorldService" class="com.bionic.edu.HelloWorldService">
    <property name="message" value="I am Victor." />
    <property name="repeat" value="3" />
</bean>
<bean id="helloKittyService" class="com.bionic.edu.HelloKittyService" />
<bean name="application" class="com.bionic.edu.Application">
     <constructor-arg ref="helloWorldService" />
</bean>
```

Example 5. Output

- Hello, world! I am Victor.
- Hello, world! I am Victor.
- Hello, world! I am Victor.

- See P215NumericInjection project for the full text

Object Injection

- The real value of DI is found in wiring an application's collaborating objects together so that they don't have to wire themselves together
- Use ref attribute of <property> tag for this purpose

Example 5. Reference Injection

1. Add getter&setter for greeting field of Application class
2. Change beans.xml as follows

```
<bean id="helloWorldService" class="com.bionic.edu.HelloWorldService">
    <property name="message" value="I am Victor." />
</bean>
<bean id="helloKittyService" class="com.bionic.edu.HelloKittyService" />
<bean name="application" class="com.bionic.edu.Application">
    <property name="greeting" ref="helloKittyService" />
</bean>
```

Example 5. Output

- Hello, Kitty!

Example 5. Bean.xml Changes

```
<bean id="helloWorldService" class="com.bionic.edu.HelloWorldService">
    <property name="message" value="I am Victor." />
</bean>
<bean id="helloKittyService" class="com.bionic.edu.HelloKittyService" />
<bean name="application" class="com.bionic.edu.Application">
    <property name="greeting" ref="helloWorldService" />
</bean>
```

Example 5. Output

- Hello, world! I am Victor.
- See P216PropertyRef project for the full text

Auto Wiring

- In large applications, the number of beans will increase and the corresponding XML written to configure the numerous beans will become very large
- Spring provides a feature called 'Auto-Wiring' that minimizes the XML to be written provided that certain assumptions are made about the nomenclature of beans and properties
- Spring provides auto-wiring based on both XML and Annotations

Auto-Wiring based on Annotations

- Use the <context:component-scan> tag in spring-context.xml
- Use the *@Inject* annotation to qualify either the member or a corresponding method (usually the setter method) which takes the injected type as argument

The @*Inject* annotation

- The @*Inject* annotation can be used to qualify:
  - a member
  - any method (including setter method) which takes the injected type as argument

Autodiscovery

- By default, <context:component-scan>looks for classes that are annotated as:
  - @Component - indicates that the class is a Spring component
  - @Controller - indicates that the class defines a Spring MVC controller
  - @Repository - the class defines a data repository
  - @Service - the class defines a service
  - Any custom annotation that is itself annotated with @Component

@Component vs @Named

- @Named and @Component annotations are used enabling a class to be auto detected as the bean definition for spring's application context
  - @Named is part of the Java specification JSR-330. It is more recommended since this annotation is not tied to Spring APIs.
  - @Component is part of the Spring's annotations library.

Example 6. Annotations

1. Create Spring project with name P221FirstInject
2. Tune pom.xml file
3. Tune beans.xnm file
4. Create application classes
5. Run application

Example 6. pom.xml

- Add the following dependency to the project's pom.xml file:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Example 6. beans.xml

- Create the following beans.xml file:

```
<?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-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="com.bionic.edu" />
</beans>
```

Example 6. HelloWorldService

```
package com.bionic.edu;
import javax.inject.Named;
@Named
public class HelloWorldService implements GreetingService {
    public String message;
    public int repeat;
    . . . . . .
    public void sendGreeting() {
        for (int i = 0; i < repeat; i++){
      System.out.println("Hello, world! " + message);
        }
    }
}
```

Example 6. Application Class

```
@Named
public class Application {
    @Inject
    GreetingService greeting = null;
    . . . . . .
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("beans.xml");
        Application application = (Application)ctx.getBean("application");
        application.start();
    }
    public void start(){
        if (greeting != null) greeting.sendGreeting();
    }}
```

Example 6. Output

- Hello, world!
- See P221FirstInject project for the full text

Example 7. Autowiring by Name

- Annotate HelloKittyService class with @Named:

```
package com.bionic.edu;
import javax.inject.Named;
@Named
public class HelloKittyService implements GreetingService
    {
    public void sendGreeting(){
     System.out.println("Hello, Kitty!");
    }
}
```

Example 7. Output

- Running the application leads to an exception

Exception in thread "main" org.springframework.beans.factory.BeanCreationException:

Error creating bean with name 'application': Injection of autowired dependencies failed;

. . . . . .

No unique bean of type [com.bionic.edu.GreetingService] is defined:

expected single matching bean but found 2: [helloKittyService, helloWorldService]

The @*Named* Annotation

- If multiple bean types are available for injection, then Spring will be unable to make a decision on which bean to inject and will throw an Exception
- In such cases, we can use the *@Named(name="..")* annotation and give the name of the bean that we want Spring to inject.

Example 7. @Named Annotation

```
@Named
public class Application {
    @Inject
    @Named("helloWorldService")
    GreetingService greeting = null;
    . . . . . .
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("beans.xml");
        Application application = (Application)ctx.getBean("application");
        application.start();
    }
 . . . . . .
```

Example 7. Output

- Hello, world!

Example 7. @Named Annotation

```
@Named
public class Application {
    @Inject
    @Named("helloKittyService")
    GreetingService greeting = null;
    . . . . . .
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("beans.xml");
        Application application = (Application)ctx.getBean("application");
        application.start();
    }
 . . . . . .
```

Example 7. Output

- Hello, Kitty!
- See P222InjectByName project for the full text