SPRING FRAMEWORK OVERVIEW

Outline

- Introduction to Spring Framework
- Spring Module Stack
- Dependency Injection/Inversion of Control
- Bean Life Cycle Management
- Bean Wiring

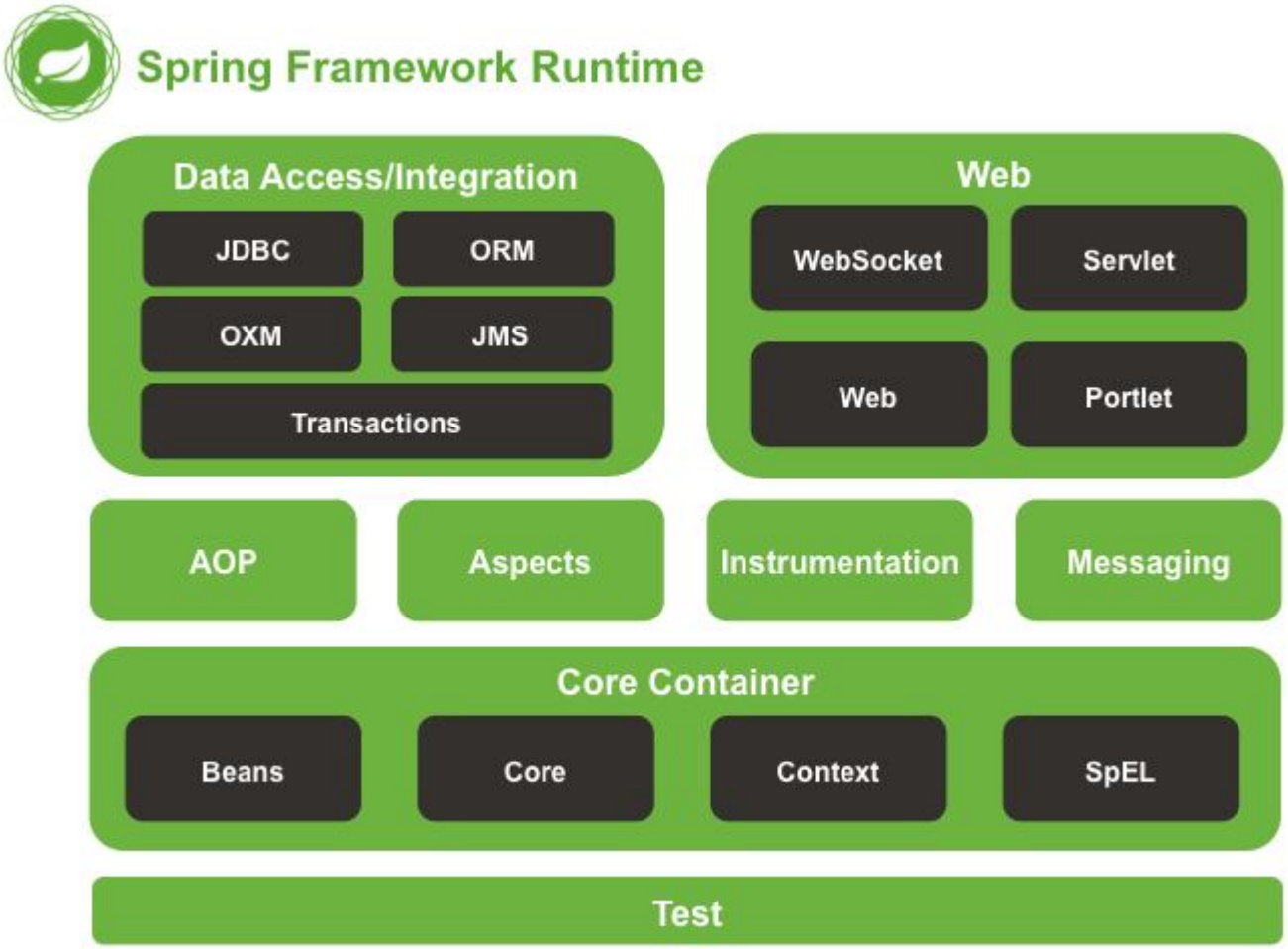
Why framework?

- Provides underlying platform readily to get start quickly
- Uniform and standardized way of application development
- Faster application development
- Enforce to use better architecture and design patterns
- Proven and tested
- Ease of maintenance

What is Spring Framework?

- Java application development framework
- Handles underlying infrastructure. Enables developer to focus on business logic
- Spring allows you to build applications from "plain old Java objects" (POJOs)
- And to apply enterprise services non-invasively to POJOs

Spring Module Stack



Dependency Injection / Inversion of Control

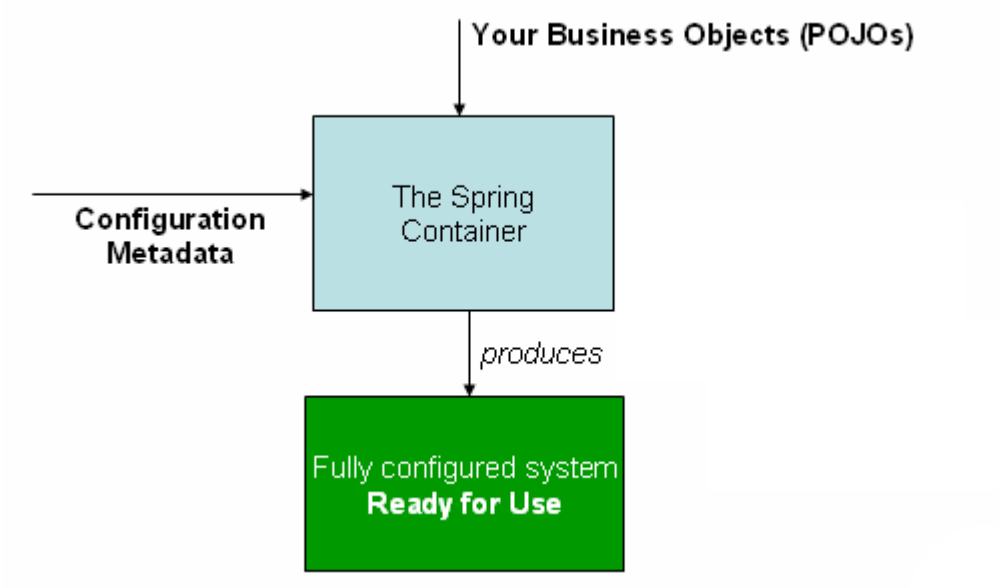
- Passing the service to the client, rather than allowing a client to build or find the service
- Allows a program design to follow the dependency inversion and single responsibility principles
- The container injects those dependencies when it creates the bean
- This is fundamentally the inverse of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes
- Achieved through three ways constructor injection, setter injection and interface injection

DI Advantages

- Code is cleaner with reduction of boilerplate code
- Decoupling is more effective when objects are provided with their dependencies
- Can be leveraged for system configuration externalization – no recompilation for system changes
- Promotes concurrent / independent development of inter-dependent classes / modules
- The object does not look up its dependencies, and does not know the location or class of the dependencies
- Classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests

Components

- Beans
- IoC Container
- Configuration Metadata



IoC Container

- Responsible for instantiating, configuring, and assembling the beans
- BeanFactory
  - XmlBeanFactory (for resource constrained apps)
- ApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext

Metadata

- Container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata.
- In the form of
  - XML
  - Java annotations
  - Java code

Sample XML based Configuration Metadata

```xml
<?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.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

Bean Instantiation

- Constructor
- Static Factory
- Instance Factory

Constructor

```
<beans>
    <bean id="exampleBean" class="examples.ExampleBean"/>
    <bean name="anotherExample" class="examples.ExampleBeanTwo"/>
</beans>
```

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

Static Factory Method

```
<bean id="clientService“ class="examples.ClientService“
        factory-method="createInstance"/>

public class ClientService {
        private static ClientService clientService = new ClientService();
        private ClientService() {}
        public static ClientService createInstance() {
                return clientService;
        }
}
```

Instance Factory

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator“/>

<bean id="clientService“ factory-bean="serviceLocator” factory-method="createClientServiceInstance"/>
<bean id=“accountService“ factory-bean="serviceLocator” factory-method=" createAccountServiceInstance "/>

public class DefaultServiceLocator {
        private static ClientService clientService = new ClientServiceImpl();
        private static AccountService accountService = new AccountServiceImpl();

        public DefaultServiceLocator() {};

        public ClientService createClientServiceInstance() {
                return clientService;
        }

        public AccountService createAccountServiceInstance() {
                return accountService;
        }
}
```

Constructor Injection

- Container invokes a constructor with a number of arguments, each representing a dependency.
- Calling a static factory method with specific arguments to construct the bean is equivalent

Constructor argument resolution

```
<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-arg type="int" value="7500000"/>
        <constructor-arg type="java.lang.String" value="42"/>
</bean>

<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-arg index="0" value="7500000"/>
        <constructor-arg index="1" value="42"/>
</bean>

<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-arg name="years" value="7500000"/>
        <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

Setter Injection

- Container invokes a setter method with a number of arguments, each representing a dependency.

Setter Injection example

```
<bean id="exampleBean" class="examples.ExampleBean">
        <property name="beanOne">
                <ref bean="anotherExampleBean"/>
        </property>
        <property name="beanTwo" ref="yetAnotherBean"/>
        <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean“ class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

Constructor or Setter Injection?

- Constructor injection helps implement application components as *immutable objects* and to ensure that required dependencies are not null.
- Constructor-injected components are always returned to client (calling) code in a fully initialized state
- Setter injection should primarily only be used for optional dependencies
- Benefit of setter injection is that setter methods make objects of that class amendable to reconfiguration or re-injection later.
- Management through JMX MBeans is a use case for setter injection

Primitives & Strings

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
        <property name="username" value="root"/>
        <property name="password" value="masterkaoli"/>
        <property name=“connectionTimeout” value=“60”/>
</bean>
```

References to other beans

```
<bean id="accountService" class="com.foo.SimpleAccountService“/>
<bean id=“factory“ class="org.spring.aop.framework.ProxyFactoryBean">
        <property name="target">
                <ref bean="accountService"/>
        </property>
</bean>
```

Inner Bean

```
<bean id="outer" class=“com.example.Group">
        <property name="target">
                <bean class="com.example.Person">
                        <property name="name" value="Fiona Apple"/>
                        <property name="age" value="25"/>
                </bean>
        </property>
</bean>
```

Collections

```
<bean id="moreComplexObject" class="example.ComplexObject">
        <property name="adminEmails">
                <props>
                        <prop key="administrator">administrator@example.org</prop>
                        <prop key="support">support@example.org</prop>
                        <prop key="development">development@example.org</prop>
                </props>
        </property>
        <property name="someList">
                <list>
                        <value>a list element followed by a reference</value>
                        <ref bean="myDataSource" />
                </list>
        </property>
        <property name="someMap">
                <map>
                        <entry key="an entry" value="just some string"/>
                        <entry key ="a ref" value-ref="myDataSource"/>
                </map>
        </property>
        <property name="someSet">
                <set>
                        <value>just some string</value>
                        <ref bean="myDataSource" />
                </set>
        </property>
</bean>
```

Depends On

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

```
<bean id="beanOne" class="EgBean" depends-on="manager,accountDao">
        <property name="manager" ref="manager" />
</bean>
<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

Bean Scopes

- **Singleton**: Single object instance per Spring IoC container
- **Prototype**: Each reference creates a new object instance
- **Request**: Each HTTP request has its own instance of a bean created
- **Session**: Scopes a single bean definition to the lifecycle of an HTTP Session
- **Application**: Scopes a single bean definition to the lifecycle of a ServletContext

Lazy Initialization

- Spring eagerly creates and configures all singleton beans as part of the initialization process.
- Pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately
- Lazy-initialized bean tells the IoC container to create a bean instance when it is first requested

Lazy Initialization - example

```
<bean id=“expensive" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name=“lightWeight" class="com.foo.AnotherBean"/>
```

Autowiring

- Spring can *autowire* relationships between beans.
- Autowiring has the following advantages:
  - Reduce the need to specify properties or constructor arguments
  - Can update a configuration as objects evolve. If a new dependency is added to a class, that dependency can be satisfied automatically without you needing to modify the configuration.
- Specify autowire mode for a bean definition with the autowire attribute of the <bean/> element

Autowiring Modes

- **no**: No autowiring
- **byName**: Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.
- **byType**: Allows a property to be autowired, if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown
- **constructor**: Similar to *byType*, but applies to constructor arguments

Disadvantages of autowiring

- Explicit dependencies in property and constructor-arg settings always override autowiring
- You cannot autowire *simple* properties such as primitives, Strings, and Classes
- Autowiring is less exact than explicit wiring
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. If no unique bean definition is available, an exception is thrown

Autowiring - Example

```
package com.examples.spring.core.autowire;
public class Customer {
        private Person person;
        public Customer(Person person) {
        this.person = person;
        }

        public void setPerson(Person person) {
        this.person = person;
        }
}

package com.examples.spring.core.autowire;
public class Person {
        //...
}
```

Autowiring – byName

```
<bean id="customer" class=“com.examples.g.spring.core.autowire.Customer"
        autowire="byName" />
<bean id="person" class="com.examples.spring.core.autowire.Person" />
```

Autowiring - byType

```
<bean id="customer" class="com.examples.spring.core.autowire.Customer"
        autowire="byType" />
<bean id="person" class="com. examples.spring.core.autowire.Person" />
```

Autowiring – constructor

```
<bean id="customer" class="com. examples.spring.core.autowire.Customer"
        autowire="constructor" />
<bean id="person" class="com. examples.spring.core.autowire.Person" />
```

Initialization & Destruction Callbacks

```
<bean id=“greetings" class="com.training.spring.core" init-method="init"
        destroy-method="cleanup"/>
```

Bean definition inheritance

```
<bean id="inheritedTestBean" abstract="true"
        class="org.springframework.beans.TestBean">
        <property name="name" value="parent"/>
        <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
        class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">
        <property name="name" value="override"/>
</bean>
```

Thank You!