Spring

© Legal Name City 13 April 2020

Internal

Agenda

- What is Spring?
- Spring Modules
- What is Spring Bean?
- Dependency Injection (DI)
- Spring Projects
- SpringBoot
- Spring MVC

What is Spring?

What is Spring?

- ❖ is a framework for building Java applications
- ❖ handles Dependency Injection (DI)
- ❖ wraps excellent Java libraries – for examples, Task Executors and Quartz Scheduler
- ❖ provides a lot of implementations like REST, MVC web framework, security, etc.

Spring Features - 1

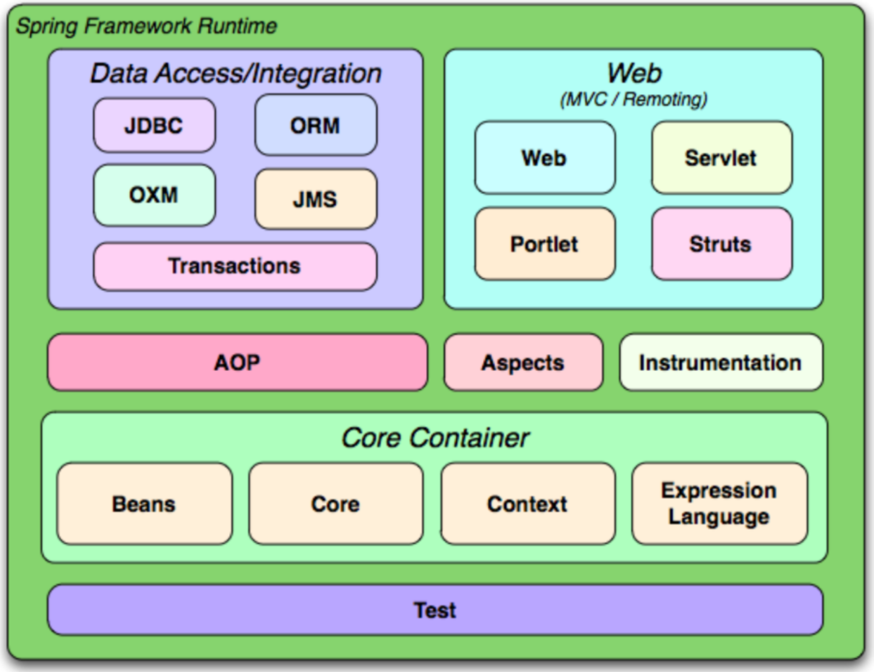
- Lightweight - the core of spring framework is around 1MB. And the processing overhead is also very negligible.
- Inversion of control (IOC) - loose coupling: the objects give their dependencies instead of creating or looking for dependent objects.
- Aspect oriented (AOP) - supports Aspect oriented programming
- Container - contains and manages the life cycle and configuration of application objects.

Spring Features - 2

- MVC Framework - comes with MVC framework, and accommodates view technologies like JSP, Velocity, POI, etc.
- Transaction Management - provides a generic abstraction layer for transaction management.
- JDBC Exception Handling - offers a meaningful exception hierarchy, which simplifies the error handling strategy.
- Integration with Hibernate, JDO, and iBATIS - provides integration services with Hibernate, JDO and iBATIS

Spring Modules

Spring Modules



Core Container

- The Core Container consists of the Core, Beans, Context, and Expression Language modules.
- The Core and Beans modules provide the IoC and Dependency Injection features.
- The Context module inherits its features from the Beans module and adds support for internationalization, event-propagation, resource-loading, and the transparent creation of contexts. It also supports Java EE features such as EJB, JMX ,and basic remoting.
- The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.
- The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service (JMS) module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction.

Web

- Spring's Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.
- The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications.
- The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0.
- The Web-Portlet module provides the MVC implementation to be used in a portlet environment.

What is Spring Bean?

What is Spring Bean?

- In Spring, Bean is nothing special, any object in the Spring framework that we initialize through Spring container is called Spring Bean.
- Any normal Java POJO class can be a Spring Bean if it’s configured to be initialized via container by providing configuration metadata information.
- Configuration metadata:
  - How to create a bean
  - Bean's lifecycle details
  - Bean's dependencies

Bean Scopes

Scope	Description
singleton	a single instance per Spring IoC container (default)
prototype	a single bean definition to have any number of object instances
request	a bean definition to an HTTP request (only valid in the context of a web-aware Spring ApplicationContext)
session	a bean definition to an HTTP session (only valid in the context of a web-aware Spring ApplicationContext)
global-session	a bean definition to a global HTTP session (only valid in the context of a web-aware Spring ApplicationContext)

Custom init() and destroy() methods

Bean local definition - applicable to a single bean

```
<beans>
        <bean id="demoBean" class="com.manulife.DemoBean"
         init-method="customInit"
         destroy-method="customDestroy"></bean>
</beans>
```

Global definition - will be invoked for all bean definitions given under <beans> tag.

```
<beans default-init-method="customInit" default-destroy-method="customDestroy">
        <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bean>
</beans>
```

@PostConstruct and @PreDestroy annotations

❖Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.

- @PostConstruct - will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object.
- @PreDestroy - is called just before the bean is about be destroyed inside bean container.

Dependency Injection (DI)

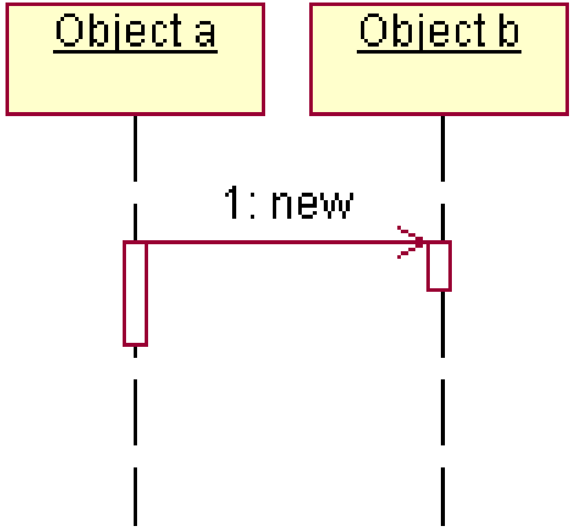
Inversion of Control (IoC)

- Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework.
- Advantages :
  - decoupling the execution of a task from its implementation
  - making it easier to switch between different implementations
  - greater modularity of a program
  - greater ease in testing

Dependency Injection (DI)

- Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object’s dependencies.
- The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler (i.e. Spring IoC container) rather than by the objects themselves.

Without DI – Traditional way



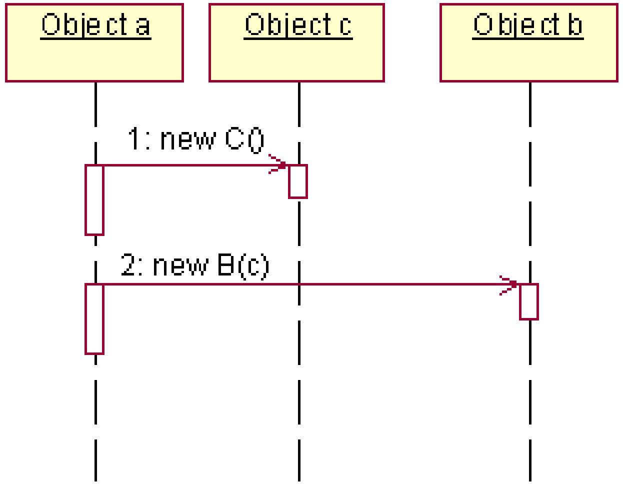
```
public class A {
    private B b;

    public A() {
        b=new B();
    }
}
```

design decisions:

- Class A needs a reference to Class B.
- Class B is a concrete class that has a default constructor.
- Class A owns the instance of class B.
- No other class can access the instance of Class B.

Without DI - Requirement Changes

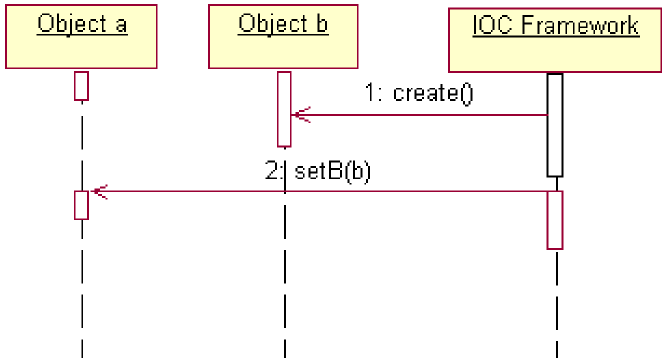


If any one of the above design decisions change, then the code must be modified. For example, if Class B changed to have a non-default constructor, which takes Class C, then the above listing would change to the following.

```
public class A {
    private B b;

    public A() {
        C c=new C();
        b=new B(c);
    }
}
```

DI Solution



The responsibility of creating Object "b" moves from Object "a" to the IoC framework, which creates and injects Object "b" into Object "a". This insulates Class A from the changes in Class B.

```
public class A {
    private B b;
    public A() { }

    public setB(B b) {
        this.b=b;
    }
}
```

DI - Constructor-Based

- The container will invoke a constructor with arguments each representing a dependency we want to set.

- Example

```
@Configuration
public class AppConfig {
        @Bean
        public Store store() {
        return new Store(item1());
        }
}
```

indicates that the class is a source of bean definitions

to define a bean

GENERALI

DI - Setter-Based

- The container will call setter methods of our class, after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

- Example

```
@Bean
public Store store() {
        Store store = new Store();
        store.setItem(item1());
        return store;
}
```

XML configuration:

```
<bean id="store" class=“com.manulife.Store">
        <property name="item" ref="item1" />
</bean>
```

DI - Field-Based (Autowired)

- In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:

```
public class Store {
        @Autowired
        private Item item;
}
```

- While constructing the Store object, if there’s no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.

Important Annotations

- Spring uses DI to configure and to bring your application together.
- It means that you just declare the components of your system and how they interact.
- Then Spring wires it all together at runtime.
- Here are the most important annotations you should know of: @Configuration, @Bean, @ComponentScan, @Component, @Service, @Repository, @Autowired, @Qualifier, etc.

Annotation Based Configuration

- Before we can use annotation-based wiring, we will need to enable it in our Spring 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"
              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:annotation-config/>
</beans>
```

Add this before using annotation in your code

Configuration, @Bean and @ComponentScan

@Configuration
to mark a class as a source of the bean definitions. Beans are the components of the system that you want to wire together.

@Bean
a bean producer. Spring will handle the life cycle of the beans for you, and it will use these methods to create the beans.

@ComponentScan
to make Spring scan the packages configured with it for the @Configuration classes.

@Autowired and @Qualifier

@Autowired
for automatic injection of beans. Spring @Qualifier annotation is used in conjunction with @Autowired to avoid confusion when we have two of more bean configured for same type.

@Qualifier
when you need more control of the DI process, this is used to avoid the confusion that occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Example to inject a specific bean

```
@Component
public class Doctor1 implements DoctorInterface {
    //
}

@Component
public class Doctor2 implements DoctorInterface {
    //
}

@Component
public class Clinic {
    @Autowired
    @Qualifier(“doctor2")
    private DoctorInterface dependency;
    ...
}
```

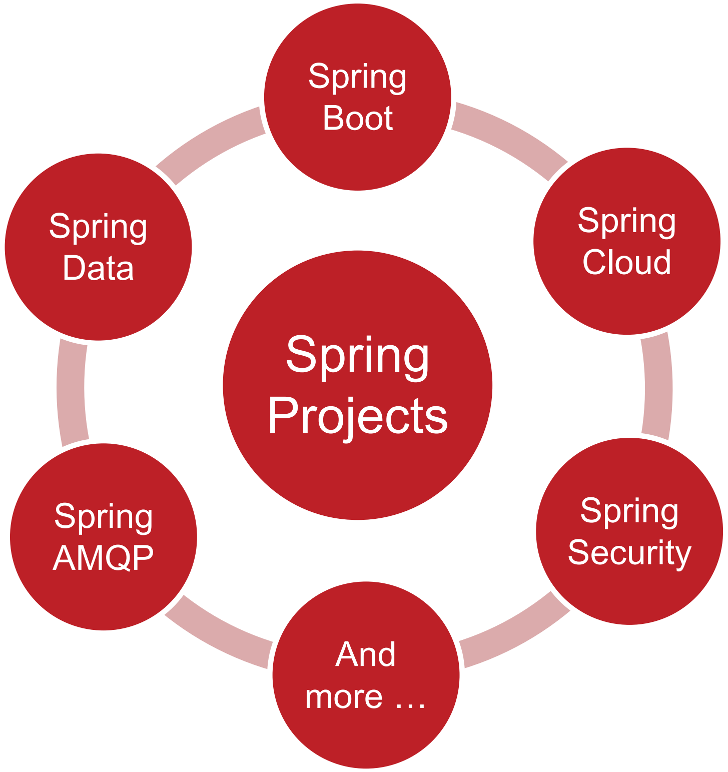
Two comoponents implement the same interface

Spring will now know which bean to autowire when @Qualifier is used

GENERALI

Spring Projects

Spring Projects



Spring Boot

Spring Boot provides a pre-configured set of frameworks to reduce boilerplate configuration so that you can have a Spring application up and running with the smallest amount of code.

Spring Boot in a minute

Spring Boot = Spring Framework + Embedded server (Tomcat, Jetty, …) − XML bean configuration

Spring Boot

@SpringBootApplication

=

Traditional Spring

@Configuration + @EnableAutoConfiguration + @ComponentScan

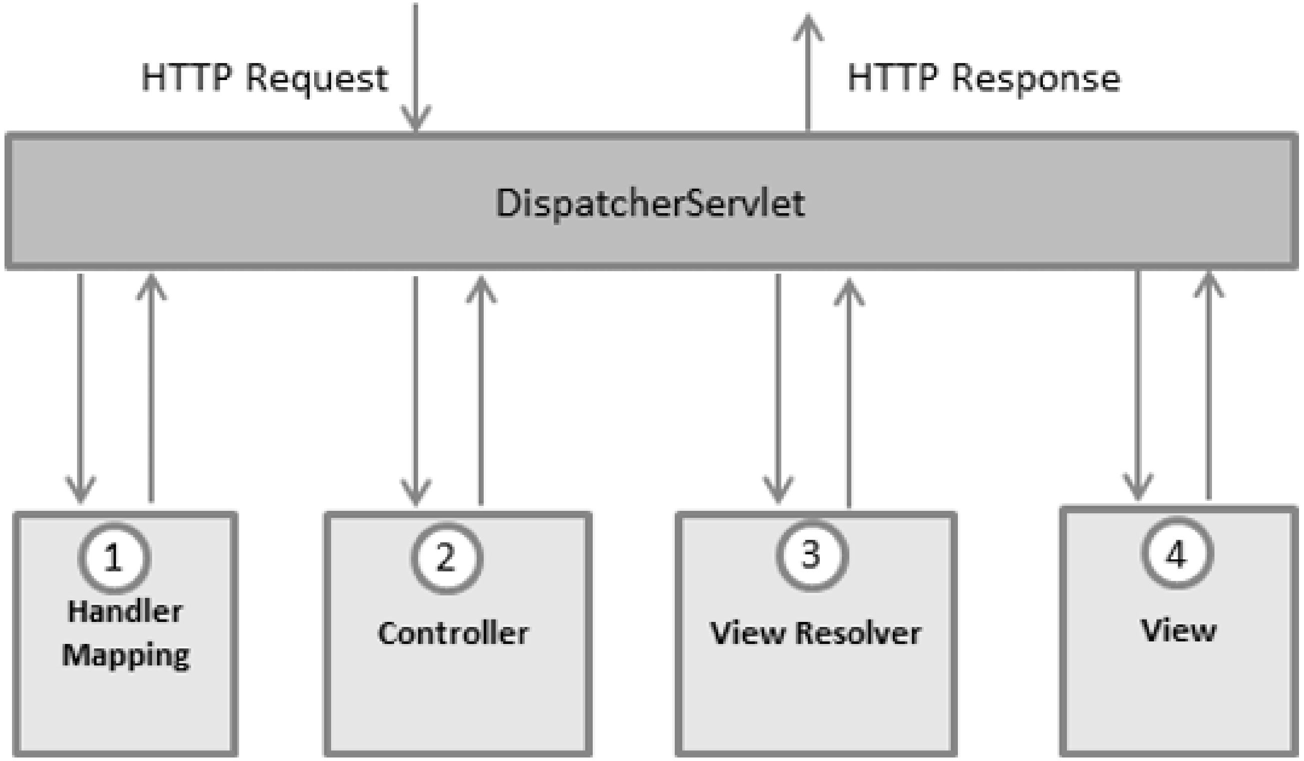
Features

- Auto-Configuration - No need to manually configure dispatcher servlet, static resource mappings, property source loader, message converters etc.
- Dependency Management - The different versions of commonly used libraries are pre-selected and grouped in different starter POMs that we can include in your project.
- Advanced Externalized Configuration - There is a large list of bean properties that can be configured through application.properties file without touching java or xml config.
- Production support - We get health checking, application and jvm metrics, jmx via http and a few more things for free.
- Runnable Jars - We can package your application as a runnable jar

Spring MVC framework

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files.

MVC Flow



https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

Required Configuration – web.xml

You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file.

```
<web-app>
	<display-name>Spring MVC Application</display-name>
	<servlet>
		<servlet-name>HelloWeb</servlet-name>
		<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>HelloWeb</servlet-name>
		<url-pattern>*.jsp</url-pattern>
	</servlet-mapping>
</web-app>
```

Required Configuration

- The web.xml file will be kept in the WebContent/WEB-INF directory of your web application.
- Upon initialization of HelloWeb DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INF directory. In this case, our file will be HelloWeb-servlet.xml.

Required Configuration – HelloWeb-servlet.xml

- Let us check the required configuration for HelloWeb-servlet.xml file, placed in your web application's WebContent/WEB-INF directory

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

    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name = "prefix" value = "/WEB-INF/jsp/" />
        <property name = "suffix" value = ".jsp" />
    </bean>
</beans>
```

Controller

- The @Controller annotation indicates that the class serves the role of a controller.
- The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
public class HelloController {
	@RequestMapping(value = "/hello", method = RequestMethod.GET)
	public String printHello(ModelMap model) {
		model.addAttribute("message", "Hello Spring MVC Framework!");
		return "hello";
	}
}
```

Exercise

Spring MVC

Spring – Exercise 1

Key steps (refer Generali-2b-Spring-exercises.docx Exercise 1 for details)

- Generate a Spring Boot project structure using Spring Initializr
- Understand the structure
- Understand the dependencies in pom.xml
- Create the controller and define the endpoints
- Create the views (JSPs)
- Startup and shut down the server
- Examine the pages
- Bonus
  - Inject bean in controller
  - Read value from property file

Spring Cloud

Spring Cloud builds on Spring Boot by providing a bunch of libraries that enhance the behaviour of an application.

You can take advantage of the basic default behaviour to get started really quickly, and then when you need to, you can configure or extend to create a custom solution.

Spring Cloud - Features

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

Spring Security

A framework that focuses on providing both authentication and authorization to Java applications.

Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements.

Spring Security - Features

- Proven technology, it’s better to use this than reinvent the wheel.
- Prevents some of the common attacks such as CSRF, session fixation attacks.
- Easy to integrate in any web application. We don’t need to modify web application configurations, spring automatically injects security filters to the web application.
- Provides support for authentication by different ways – in-memory, DAO, JDBC, LDAP and many more.
- Provides option to ignore specific URL patterns, good for serving static HTML, image files.
- Support for groups and roles.

Spring Data

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

Spring Data - Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Possibility to integrate custom repository code
- Easy Spring integration
- Advanced integration with Spring MVC controllers
- Experimental support for cross-store persistence

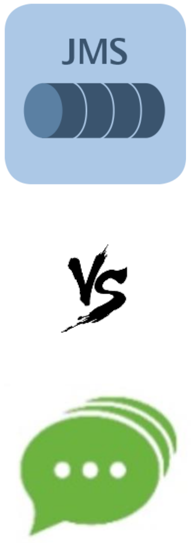
Spring AMQP (Advanced Message Queuing Protocol)

Messaging is a technique for inter-application communication.

Producer and consumer of messages are decoupled by an intermediate messaging layer known as the message broker.

Message broker provides features like persistent storage of messages, message filtering, message transformation, etc.

Why we need AMQP?



The main drawback or limitation of JMS API is interoperability that means we can develop messaging systems that will work only in Java based applications. It does not support other languages.

The major advantage of AMQP is that it supports interoperability between heterogeneous platforms and messaging brokers. We can develop our Messaging systems in any language (Java, C++, C#, Ruby etc.) and in any operating system

GENERALI

Exercise

Spring Data

Spring – Exercise 2

Key steps (refer Generali-2b-Spring-exercises.docx Exercise 1 for details)

- Base on Exercise 1, develop a service to access database
- Understand the dependencies in pom.xml
- Create the model for persistence
- Create the repository to access database
- Create the service for controller
- Modify the views to display the results from database
- Bonus
  - Connect to an external database server

Summary

What is covered

- What is Spring?
- Spring Modules
- What is Spring Bean?
- Dependency Injection (DI)
- Spring Projects

Thank You.

Contacts: