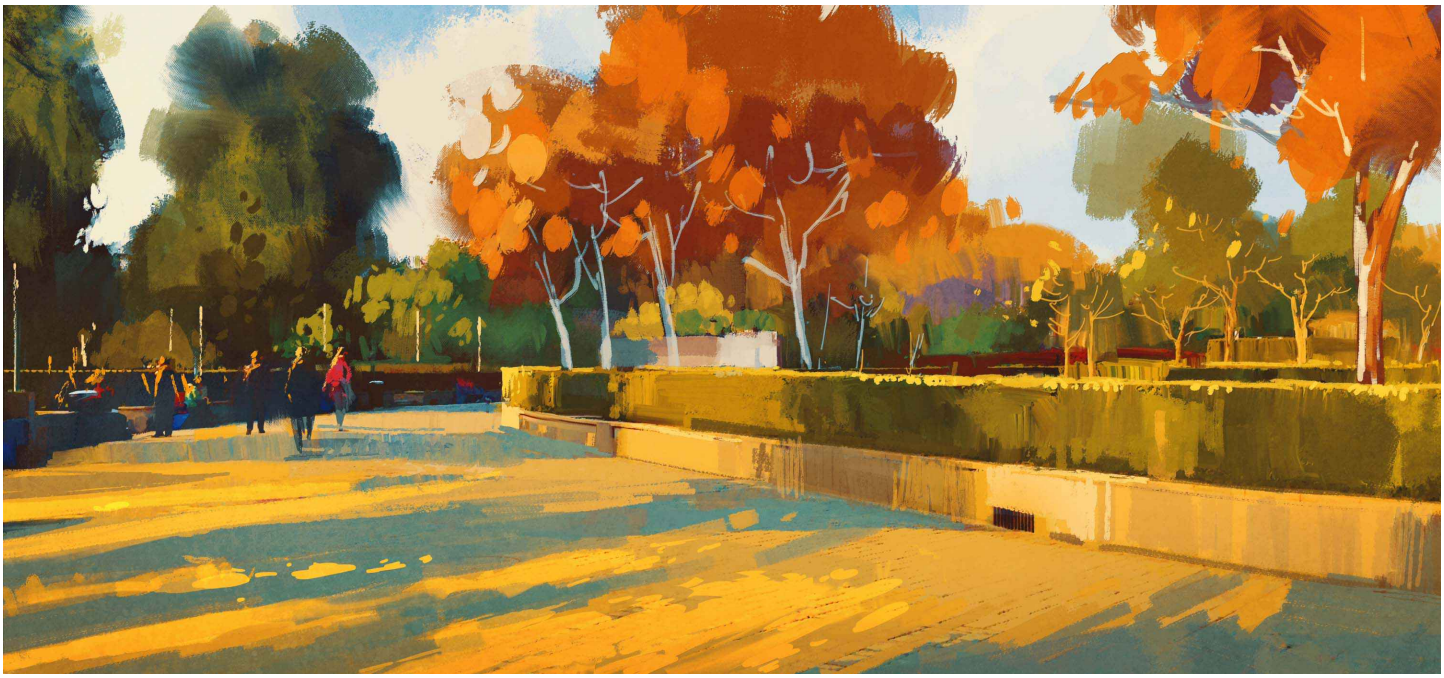


3-Spring Core Theory

Author: Vincent Lau

Learning Objectives

- Describe the pros and cons of using a framework.
- Describe the purpose of the Spring Framework.
- List the key Spring modules.
- Understanding the benefits of Spring.
- Understand the concept of Inversion of Control (IoC) and Dependency Injection (DI).
- Illustrate the purpose of the Spring Core Container.
- Describe the bean lifecycle and how to run custom init() and destroy() methods.
- List the bean scopes in Spring, and explain how the Singleton and Prototype scopes work.

Overview

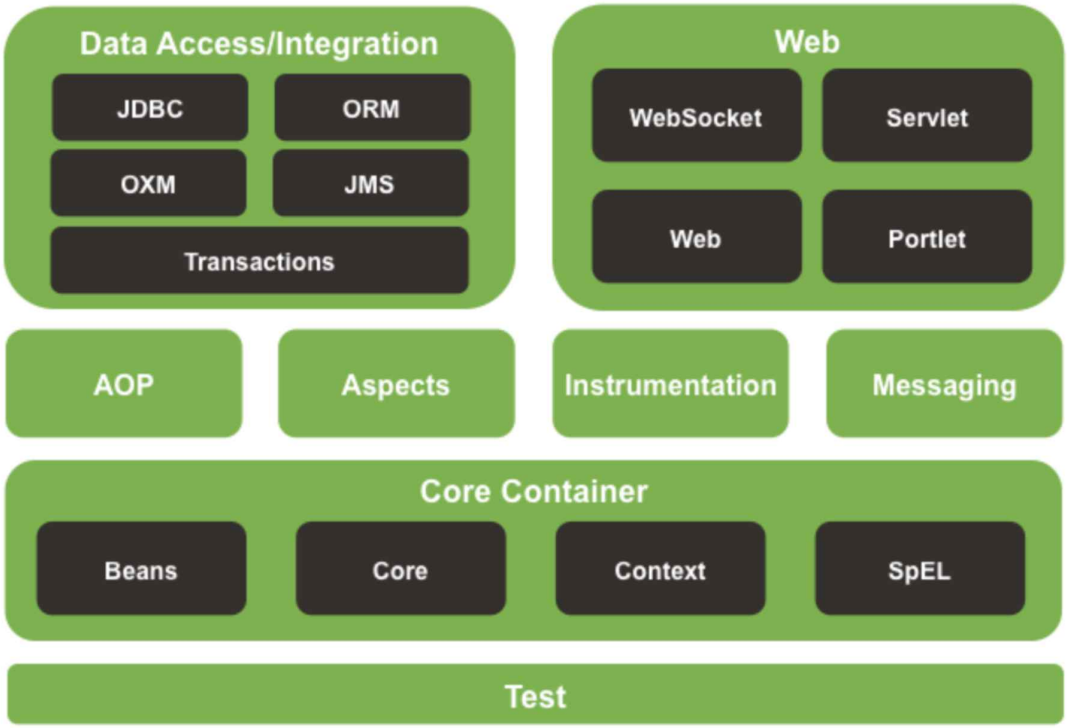
Why is Framework important for App Dev?

- It isn't absolutely necessary to use a framework to accomplish a task. But, it's often advisable to use one for several reasons:
  - Helps us **focus on the core task rather than the boilerplate** associated with it
  - Brings together years of wisdom in the form of design patterns
  - Helps us **adhere to the industry and regulatory standards**
  - Brings down the total cost of ownership for the application
- But it can't be all positives, so what's the catch:
  - Binds to a **specific version of language and libraries**

What is Spring?

- Spring was initially written by Rod Johnson and was first released in 2003 as a response to the complexity of the early J2EE specifications.
- The Spring Framework is **an open-source framework for building enterprise Java applications**.
- Spring aims to simplify the complex and cumbersome enterprise Java application development process by using techniques such as:
  - ***Inversion of Control (IoC)*** - IoC is a principle in software engineering that transfers the control of objects or portions of a program to a container or framework.
  - ***Dependency Injection (DI)*** - DI is implemented by passing dependencies as arguments to constructors or setter methods. A library that implements this approach is called an *IoC container*.
  - ***Aspect-Oriented Programming (AOP)*** - The functions that span multiple points of an application are called **cross-cutting concerns.** AOP allows you to distinguish cross-cutting concerns in applications, such as logging, transaction management, data validation, etc.

The Spring Framework Architecture



- Spring framework is divided into modules which makes it easy to pick and choose in parts to use in any application:
  - **Core**: Provides core features like ***DI (Dependency Injection)***, *Internationalization, Validation, and AOP (Aspect-Oriented Programming)*
  - **Data Access**: Supports data access through *JTA (Java Transaction API)*, ***JPA (Java Persistence API)***, *and JDBC (Java Database Connectivity)*
  - **Web**: Supports both *Servlet API (**Spring MVC**)* and of recently *Reactive API (**Spring WebFlux**)*, and additionally supports *WebSockets, STOMP, and WebClient*
  - **Integration**: Supports integration to Enterprise Java through *JMS (Java Message Service), JMX (Java Management Extension), and RMI (Remote Method Invocation)*
  - **Testing**: Wide support for unit and integration testing through *Mock Objects, Test Fixtures, Context Management, and Caching*

Why Spring?

- Spring is the **most popular** application development framework for enterprise Java.
- The Spring Framework is a **lightweight** solution and a potential one-stop shop for building your enterprise-ready applications, **especially Spring Boot Framework, actuator/ metrics/ starter.**
- It **minimizes boilerplate code** by taking care of most of the low-level implementations, such as *JDBCTemplate,* so that we can **focus on features and business logic**.
- Spring is **modular**, allowing you to **use only those parts that you need**, without having to bring in the rest.

    - For example, you can use the Core Container, with Web MVC on top, but you can also use only the *Hibernate integration code* or *JDBC abstraction layer.*
- Spring is designed to be **non-intrusive**, meaning that your domain logic code generally has no dependencies on the framework itself. It should be easy to isolate these dependencies from the rest of your codebase, especially **Spring boot project structure & design**.

IoC, DI, and IoC Container

Inversion of Control (IoC)

- `IoC` enables a framework to take control of the flow of a program and make calls to our custom code.
- The Spring framework uses abstractions with built-in implementations. **If we want to add our behavior, we need to extend the classes of the framework or plugin our own classes.**
- The advantages of this architecture are:
    - **[No object initialization during runtime]** Decoupling the execution of a task from its implementation
    - **[Easy Code change & lower risk]** Great modularity of a program, making it easier to switch between different implementations.
    - **[Good for Unit Test]** Greater ease in testing a program by isolating a component or mocking its dependencies, and **allowing components to communicate through contracts.**

Dependency Injection (DI)

- **Dependency injection** is a pattern that we use to implement IoC. The framework, to which the control is delegated or inverted, is responsible for injecting dependencies into objects.
- **Connecting objects with other objects**, or injecting objects into other objects, is done by the IoC Container rather than by the objects themselves.
- In traditional programming, we may instantiate an implementation of the *Dependency* interface within the *Service* class itself, hence tightly coupling the two:

```
1 // Traditional Programming
2 public class Service {
3     private Dependency dependency;
4 
5     public Service() {
```

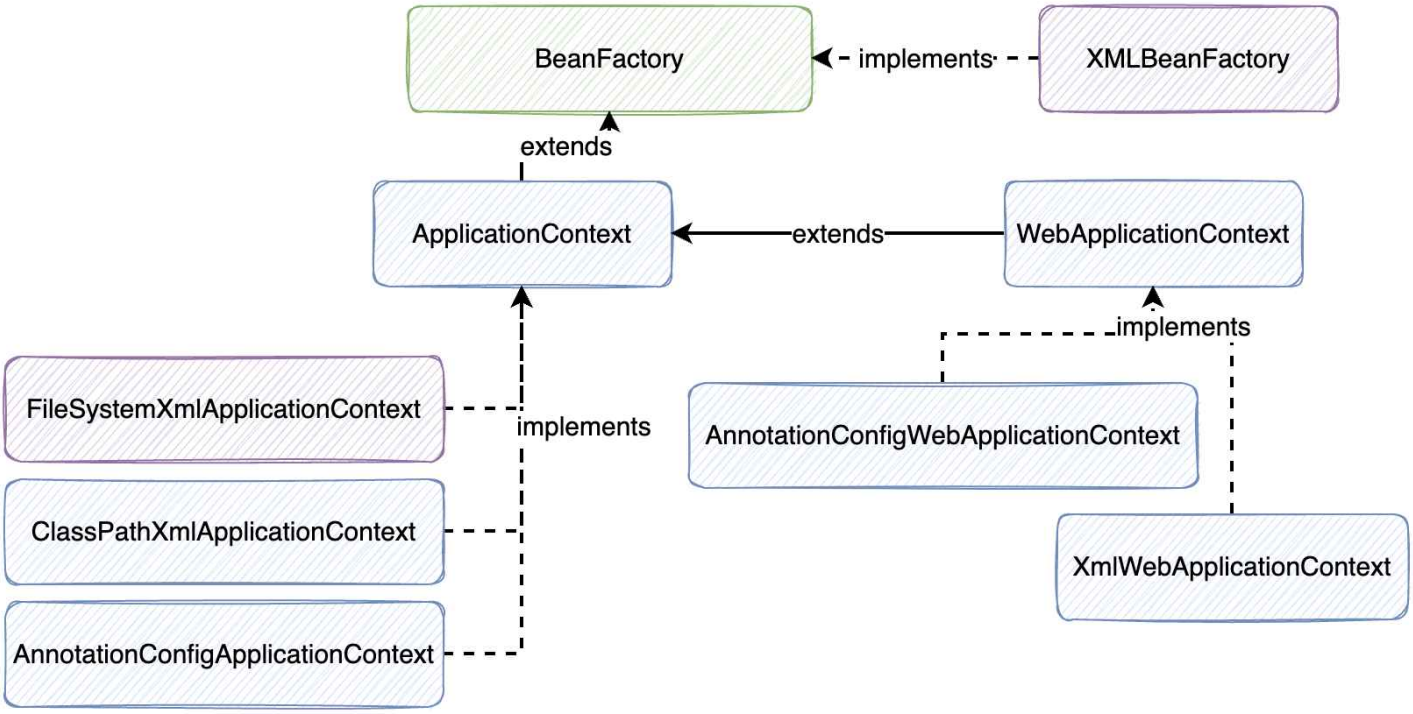
```
        this.dependency = new DependencyImpl();
    }
}
```

- By using DI, we can **decouple *Dependency* from the *Service* class** by doing this:
- The *Service* class can interact with the *Dependency* interface without any knowledge about its concrete implementation, thus achieving loose coupling.

```
// Traditional Programming, with the idea of DI
public class Service {
    private Dependency dependency;

    public Service(Dependency dependency) { // Say Dependency is an interface
        this.dependency = dependency;
    }
}
```

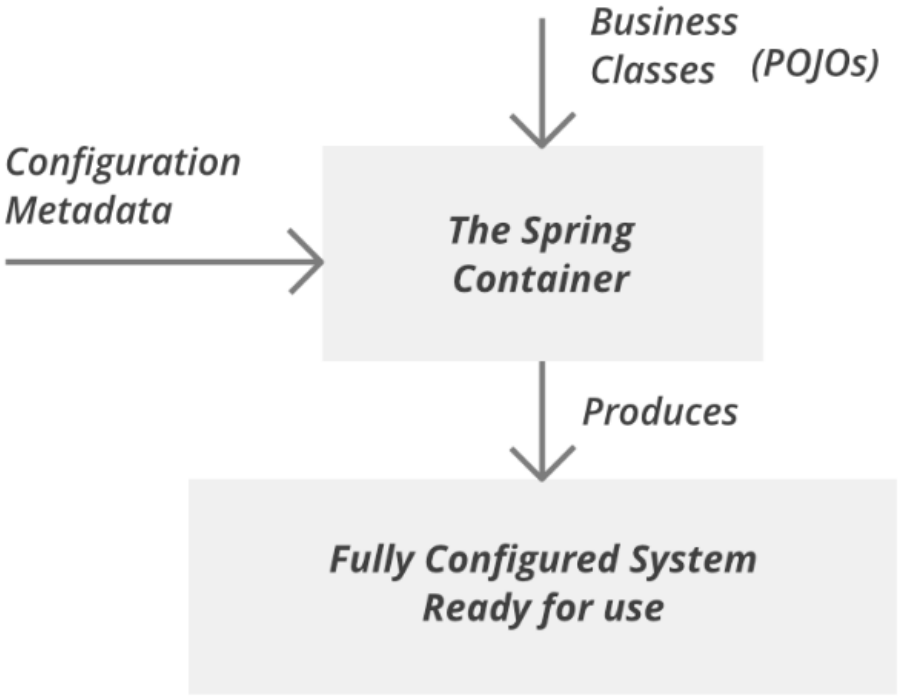
```
// Spring Programming, with the idea of DI
public class Service {
    @Autowired // Constructor is no longer needed
    private Dependency dependency;

}
```

The Spring IoC Container



- An `IoC container` is a common characteristic of **frameworks that implement IoC**.
- In the Spring framework, the interface ***ApplicationContext*** represents the IoC container.
- The Spring container is responsible for **instantiating, configuring, and assembling objects known as *beans*, as well as managing their life cycles.**
- The Spring framework provides several implementations of the *ApplicationContext* interface, such as:
  - `ClassPathXmlApplicationContext` for loading configuration from class path
  - `FileSystemXmlApplicationContext` for loading configuration from file system
  - `AnnotationConfigApplicationContext` for configuration via annotations
  - `AnnotationConfigWebApplicationContext` and `XmlWebApplicationContext` for web applications
- **The Spring IoC container gets initialized when the Spring application starts. When a bean is requested, the dependencies are injected automatically**.

Spring Beans



- The objects that form the backbone of your Spring application and that are managed by the Spring IoC container are called ***beans***.
- A **bean** is an object that is `instantiated`, `assembled`, and otherwise managed by a Spring IoC container.
- These beans are created with Java POJOs plus the configuration metadata that you supply to the container.
- The ***configuration metadata*** can be in the form of *XML configuration* or ***Annotations***. It is needed for the container to know the following:
  - **How to create** a bean (constructor/ setter/ field Injection)
  - Bean's **lifecycle** details
  - Bean's **dependencies**
- Here's one way to manually instantiate a container:

```
ApplicationContext context
            = new ClassPathXmlApplicationContext("applicationContext.xml");
```

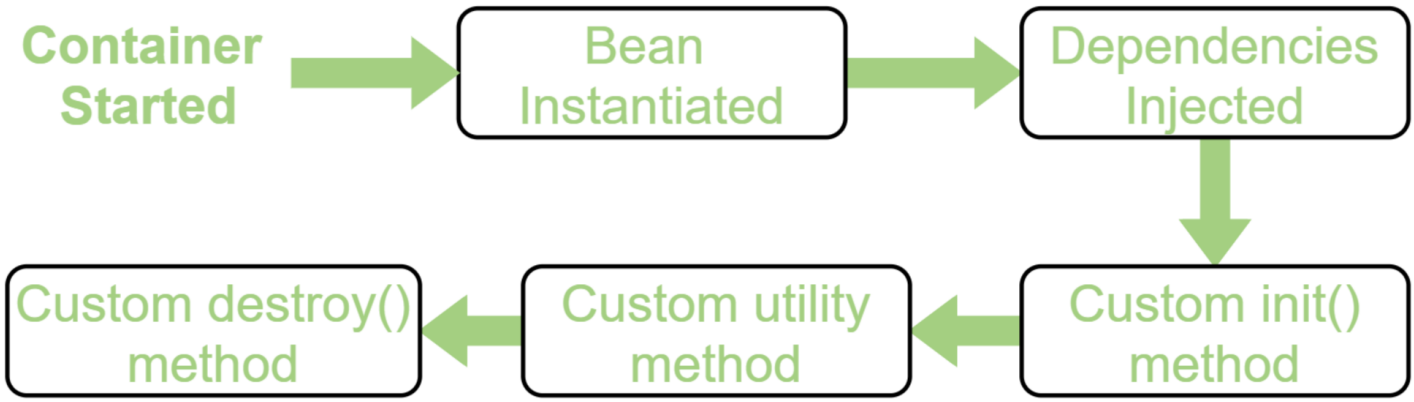
- The *applicationContext.xml* file may look something like this:

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

    <bean id="dependencyBean" class="com.bootcamp.springdemo.Dependency">
        <!-- collaborators and configuration for this bean go here -->
    </bean>
        <bean id="serviceBean" class="com.bootcamp.springdemo.Service">
        <constructor-arg name="dependency" ref="dependencyBean" />
    </bean>

</beans>
```

- To set the *dependency* attribute in the example above, we can use metadata. Then the container will read this metadata and use it to assemble beans at runtime.
- **Dependency Injection in Spring can be done through constructors, setters, or fields.**

Spring Bean Lifecycle



- On a high level, *Spring Beans* go through a lifecycle of various stages including:
  - ***Creation / Instantiation***
  - ***Injection***
  - ***Validation***
  - ***Initialization***
  - ***Destruction***

Init and Destroy Hooks

- We can use the `@PostConstruct` and `@PreDestroy` annotations to invoke custom `init()` and `destory()` methods.

- Note that this time we're using *Java Annotations* to instantiate the bean:

```
@Service
public class SomeService {
    public SomeService() {
        System.out.println("Creation of SomeService");
    }

    @PostConstruct
    public void init() {
        System.out.println("Custom initialization of SomeService");
    }

    public void doSomething() {
        System.out.println("Do something in SomeService");
    }

    @PreDestroy
    public void destroy() {
         System.out.println("Custom destruction of SomeService");
    }
}

@Configuration
public class BeanConfig {
    @Bean(name = "someServiceBean")
    public SomeService createSomeService() {
        return new SomeService(); // SomeService() & init()
    }
}

public class TestBeanLifeCycle {
    public static void main(String[] args) {
        ConfigurableApplicationContext ctx =
                    new AnnotationConfigApplicationContext(BeanConfig.class);
        SomeService someServiceBean = (SomeService)
ctx.getBean("someServiceBean");
        someServiceBean.doSomething(); //doSomething()
        ctx.close(); // must be called to invoke beans' destoroy() method
    }
}

/*
Creation of SomeService
Custom initialization of SomeService
Do something in SomeService
```

```
44 Custom destruction of SomeService
45 */
```

Spring Bean Scope

Sr.No.	Scope & Description
1	**singleton** This scopes the bean definition to a single instance per Spring IoC container (default).
2	**prototype** This scopes a single bean definition to have any number of object instances.
3	**request** This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
4	**session** This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
5	**global-session** This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

- The **scope of a bean** defines the **life cycle** and **visibility** of that bean in the context we use it.
- In this chapter, we will discuss the first two scopes, **singleton & prototype only**, as the remaining three are less frequently used and out of scope for our purpose. You can read more about them here, as well as in the official documentation.

Singleton Scope

- This scope is the ***default value*** if no other scope is specified.
- When we define a bean with the ***singleton*** scope, the **container creates a single instance of that bean**.
- All requests for that bean name will return the **same object, which is cached**.
- Any modifications to the object will be reflected in all references to the bean.
- To define a *singleton* scope for a bean, we can use either *annotation or XML*. With annotation, it will look like this:

```
1 @Bean
```

```
@Scope("singleton") // singleton is default value, not required to specify
public Service createSomeService() {
    return new Service();
}
```

- With XML, it will look something like this in the bean configuration file:

```
<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

Prototype Scope

- If the scope is set to **prototype**, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made.
- As a rule, use the prototype scope for all ***stateful*** beans and the singleton scope for ***stateless*** beans.
- To define a prototype scope, you can either use annotation:

```
@Bean
@Scope("prototype") //  stateful beans, like model class
public Service createSomeService() {
    return new Service();
}
```

- Or XML, just like defining the singleton scope:

```
<!-- A bean definition with prototype scope -->
<bean id = "..." class = "..." scope = "prototype">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

Questions

- What are the pros and cons of using a framework.

- What is the Spring framework, and why do we use it?
- What are some of the key Spring modules?
- What are the benefits of Spring?
- What is the difference between IoC and DI? How are they related to the Spring IoC Container?
- Describe the bean lifecycle.
- How do we invoke custom init() and destroy() methods?
- What is the default bean scope in Spring? How do we define it?