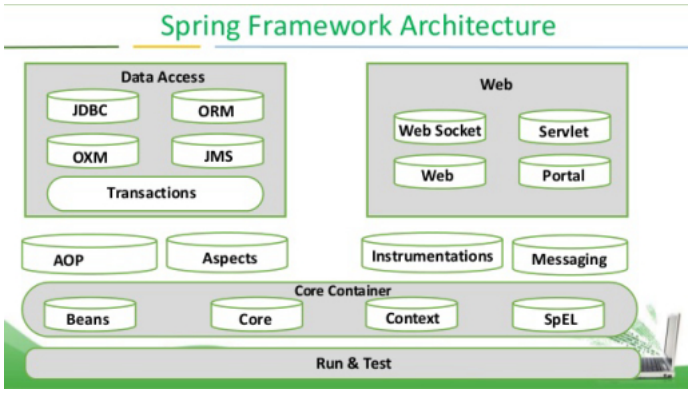
Spring Architecture



Dependency Injection

- **Definition**: DI is a design pattern used to implement Inversion of Control (IoC) by providing dependencies to a class from the outside rather than having the class create them.
- **Purpose**: To achieve loose coupling and improve code modularity, testability, and maintainability.

Simple Explanation

- **Analogy**: A car needs an engine to run. Instead of the car building its own engine, the engine is provided to it. Similarly, in software, a class receives its dependencies from an external source.

Types of Dependency Injection

1. **Constructor Injection**: Dependencies are provided through the class constructor.
   ```
   public Car(Engine engine) {    this.engine = engine;}
   ```
2. **Setter Injection**: Dependencies are provided through setter methods.
   ```
   public void setEngine(Engine engine) { this.engine = engine;}
   ```
3. **Annotation Injection/Field Injection**: Dependencies are injected directly into fields using annotations like `@Autowired`.

   @Autowired
   private Engine engine;

Spring is divided into modules. A module is a set of libraries and utilities that are related to a specific task or functionality. This modularity allows developers to pick and choose only the components they need for their application, making the framework flexible and efficient.

Key Modules in Spring

1. ## Core Module
   - **Purpose**: Provides essential features for building applications.
   - **Key Features**:
     - Dependency Injection (DI)
     - Internationalization
     - Validation
     - Aspect Oriented Programming (AOP)
2. ## Data Access Module
   - **Purpose**: Facilitates interaction with databases.
   - **Key Technologies**:
     - Java Transaction API (JTA)
     - Java Persistence API (JPA)

    - Java Database Connectivity (JDBC)
3. ### Web Module
    - **Purpose**: Supports web application development.
    - **Key Technologies**:
        - Spring MVC (Servlet API)
        - Spring WebFlux (Reactive API)
        - WebSockets
        - STOMP
        - WebClient
4. ### Integration Module
    - **Purpose**: Enables integration with other enterprise Java technologies.
    - **Key Technologies**:
        - Java Message Service (JMS)
        - Java Management Extension (JMX)
        - Remote Method Invocation (RMI)
5. ### Testing Module
    - **Purpose**: Provides tools for testing applications.
    - **Key Features**:
        - Mock Objects
        - Test Fixtures
        - Context Management
        - Caching

Inversion of Control (IoC) is a design principle in which the control of object creation and management is handed over to a container or framework rather than being managed by the application code itself. This helps in creating loosely coup led code.

IOC Container / Spring Context / Spring Container

The **Spring IOC (Inversion of Control) container** is at the core of the Spring Framework. It manages the lifecycle and dependencies of the beans (objects) in a Spring application.

Key Points:

- **IOC Container:** Manages the creation, configuration, and lifecycle of beans.
- **Types of IOC Containers:**
    - `BeanFactory`: The simplest container, provides basic DI features.
    - `ApplicationContext`: Extends `BeanFactory` to provide more enterprise-specific functionality (e.g., event propagation, declarative mechanisms).
- LifeCycle of IOC Container

Initialization

- Container is created and configured.
- Bean definitions are read and registered.
- Definitions are registered, but beans are not yet instantiated.

Refresh

- Container state is reset.
- All singleton beans are instantiated and configured.
- Beans are instantiated.
- Dependencies are injected.
- Initialization methods (e.g., `@PostConstruct`, `afterPropertiesSet()`) are called.
- `BeanPostProcessor` implementations are applied.

Start

- Container transitions to "running" state.
- Beans implementing `Lifecycle` have `start()` method invoked.

Stop

- Container transitions to "stopped" state.
- Beans implementing `Lifecycle` have `stop()` method invoked.

Close

- Container is closed.
- All resources are released.
- Beans undergo destruction methods (e.g., `@PreDestroy`, `destroy()`).
- Singleton beans are destroyed, and resources are released.

To use Lifecycle of IOC Container we use ConfigurableApplication Context instead of ApplicationContext

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MyApp {
    public static void main(String[] args) {
        // Create the application context
        ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        // Refresh the context to initialize beans
        context.refresh();
        // Start the context (Lifecycle beans will be started)
        context.start();
        // Retrieve and use the lifecycle bean
        MyLifecycleBean myLifecycleBean = context.getBean(MyLifecycleBean.class);
        System.out.println("Is MyLifecycleBean running? " + myLifecycleBean.isRunning());
        // Stop the context (Lifecycle beans will be stopped)
        context.stop();
        System.out.println("Is MyLifecycleBean running? " + myLifecycleBean.isRunning());
        // Close the context
        context.close();
    }
}
```

Bean - A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.

Sr.No.	Properties & Description
1	**class** This attribute is mandatory and specifies the bean class to be used to create the bean.
2	**name** This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
3	**scope** This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
4	**constructor-arg** This is used to inject the dependencies and will be discussed in subsequent chapters.
5	**properties** This is used to inject the dependencies and will be discussed in subsequent chapters.
6	**autowiring mode** This is used to inject the dependencies and will be discussed in subsequent chapters.
7	**lazy-initialization mode** A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.
8	**initialization method** A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter.
9	**destruction method** A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

There are three ways to define beans
1) using xml <Bean....
2) using @Component Annotation
3) using @Configuration with @Bean

BeanFactory

- **Definition:** Core interface for accessing Spring beans.
- **Initialization:** Lazy initialization, beans are instantiated when requested.
- **Usage:** Ideal for memory-constrained environments or when full container functionality isn't needed.
- **Example Code:**

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));MyBean bean = (MyBean) factory.getBean("myBean");
```

ApplicationContext

- **Definition:** Advanced interface extending BeanFactory with additional features.
- **Initialization:** Pre-instantiates all singleton beans at container startup.
- **Features:** Automatic BeanPostProcessor, event publication, resource loading, internationalization, etc.
- **Usage:** Preferred for most applications due to enhanced functionality.
- **Example Code:**

`ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");MyBean bean = (MyBean) context.getBean("myBean");`

Most Used Methods

BeanFactory Interface Methods

1. **`getBean(String name,Class<T> reqType)`** Retrieves an instance of the specified bean by its name.(with reqType - return type is T , without - return type is object) .
2. **`containsBean(String name)`**
   - Checks if the BeanFactory contains a bean with the given name.
3. **`isSingleton(String name)`**
   - Determines if the bean with the given name is a singleton (alsways same instance is returned)
4. **`isPrototype(String name)`**
   - Determines if the bean with the given name is a prototype(the bean returned by the container is always new)
5. **`getAliases(String name)`**
   - Retrieves alias names for the specified bean name.

ApplicationContext Interface Methods

1. **`getMessage(String code,Object[] args, String defaultMessage, Locale locale)`**
   - Retrieves a message from the message source for the given code, arguments, and locale.
2. **`getBeansOfType(Class<T> type)`**
   - Retrieves all beans of the specified type as a Map.
3. **`getBeanDefinitionCount()`**
   - Retrieves the number of bean definitions in the container.
4. **`getBeanDefinitionNames()`**
   - Retrieves the names of all beans defined in the container.
5. **`containsBeanDefinition(String name)`**
   - Checks if the container contains a bean definition with the given name.

Constructors for Bean creation

1.
   - Choose `XmlBeanFactory` for memory-constrained environments or when lazy loading(bean instance returned only when asked) is preferred.
   - Use `ClassPathXmlApplicationContext` for most applications for its combination of lazy loading and pre-instantiation.
   - Opt for `FileSystemXmlApplicationContext` when loading bean definitions from XML files located outside the classpath.

Internationalizaion (i18n)

`getMessage("key", params, "Default Message", Locale.ENGLISH):`
Create a messsage_<locale>.properties in resource folder ,the message file should contains messages in form of key = value
greeting=Bonjour, {0}! Bienvenue à {1}.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

        // Parameters to replace in the message
        Object[] params = new Object[]{"John", "Spring Framework"};

        // Retrieve message for English locale
        String messageEn = context.getMessage("greeting", params, "Default Message",
Locale.ENGLISH);
        System.out.println(messageEn);  // Output: Hello, John! Welcome to Spring Framework.

        // Retrieve message for French locale
        String messageFr = context.getMessage("greeting", params, "Default Message",
Locale.FRENCH);
        System.out.println(messageFr);  // Output: Bonjour, John! Bienvenue à Spring Framework.

        // Retrieve a farewell message for English locale
        String farewellEn = context.getMessage("farewell", params, "Default Message",
Locale.ENGLISH);
        System.out.println(farewellEn);  // Output: Goodbye, John! See you at Spring Framework.
    }
}
```

Bean Property

Suppose for a class that you are injecting the object as a bean , there may be some data fields that are null, or are not being set. So, you can also set them using the bean tag.

this property tag uses getters and setters
this constructor-org tag uses constructor for setting the value

```
<bean id="vehicle"  class ="com.navin.telusko.Bike/">
     <property name = "feild_name" value ="value></property>
    <constructor-org  value = "value" > </constructor-org>
</bean>
```

Configuration in Spring

XML-based Configuration:

```
<?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/springc
</beans>
```

Defining a bean: and Injecting beans :

1) Create the variable as private
2) Create Getter and Setter methods for that varaible as beans use getter setter
3) now config them in the xml file (bean config file) using either property value tag or p : schema

```
<bean id="myBean" class="com.example.MyBean"> <property name="propertyName" value="propertyValue"/></bean>
<bean id="myService" class="com.example.MyService"> <property name="myBean" ref="myBean"/></bean>
```

```
<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.xsd
           http://www.springframework.org/schema/p http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="myService" class="com.example.MyService" p:studentName="Rajat" p:studentUid="2871" />
```

Injecting collections

```
<property name="set">
   <set>
       <value>19</value>
       <value>20</value>
       <value>17</value>
   </set>
</property>
<property name="map">
    <map>
        <entry key="Rajat" value="19"></entry>
        <entry key="Travis" value="20"></entry>
        <entry key="Gys" value="17"></entry>
    </map>
</property>
```

Injecting Reference

just replace the value property with the name of the bean
value for variables Primitive , ref for objects / classes

Constructor Injection

<constructor-arg value = "" />
each constructor-arg represents a single value of a constructor
if there are multiple constructor , suppose that con(string,string) and con(string,int) , we need to use value="13" type="int" , to avoid ambiguity

LifeCycle Methods of Beans

1) **Define** the bean Class and XML File
When we do getBean()
2) Object Created
3) **Value Injected/Instantiated**
4) Called init()
5) Then we can read and use the bean // can call access methods and variables of bean
6) Destroy() called
7) Object Destroyed

These are just two methods define in a bean one is for initialisation and other is for clean up.And these init and dest. methods are called automatically by spring.

XML

<bean name / class init-method="init" destroy-method="destroy">
you can call these methods yourself
the destroy method is not automatically called , so we use AbstractApplicatoinContext instead of ApplicationContext and use its method registerShutdownHook
if you have many beans having the same init and destroy method then u can simply intiallise

default-init-method = "init" default-destroy-method = "destroy"

Interface Config

implement interfaces like **InitializingBean** and override the **afterPropertiesSet()** (init()) and for destroy implement interface **DisposableBean** and override its method **destroy()**

Annotation Based

@postConstruct and @preDestroy

```
<dependency>
<groupId>javax.annotation</groupId>
<artifactId>javax.annotation-api</artifactId>
<version>1.3.2</version>
</dependency>
```

IMP for using Annotation Config

Also to use annotation in this , we also need to add <context:annotation-config/> in your configure.xml file
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

Why do alll the beans get Injected and Initiallised without calling them ?

Eager Initialisation -> By default, spring initilizes all singleton Beans at startup,It instantiate all singleton beans , inject all dependencies, call all initialization methods (afterPropertiesSet or custom Initi-Method) , There is also a concept of Lazy Initialization where beans are created and intiialized only when they are actually needed. (To Enable Lazy ini, lazy-init=true;

Spring Java Code Based Configuration Vs XML Based Configuration

(Java Concept Of The Day)

Java Code Based Configuration

```
package com.javaconceptoftheday;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig
{
        @Bean
        public ClassABC classABC()
        {
                return new ClassABC();
        }


        @Bean
        public ClassXYZ classXYZ()
        {
                return new ClassXYZ();
        }
}
```

XML Based Configuration

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

<beans .....
        .....
        .....>
    .
    .
    .
    .
    .
    .
    <bean id = "classABC" class =
                        "com.javaconceptoftheday.ClassABC" />
    .
    .
    .
    .
    <bean id = "classXYZ" class =
                        "com.javaconceptoftheday.ClassXYZ" />
    .
    .
</beans>
```

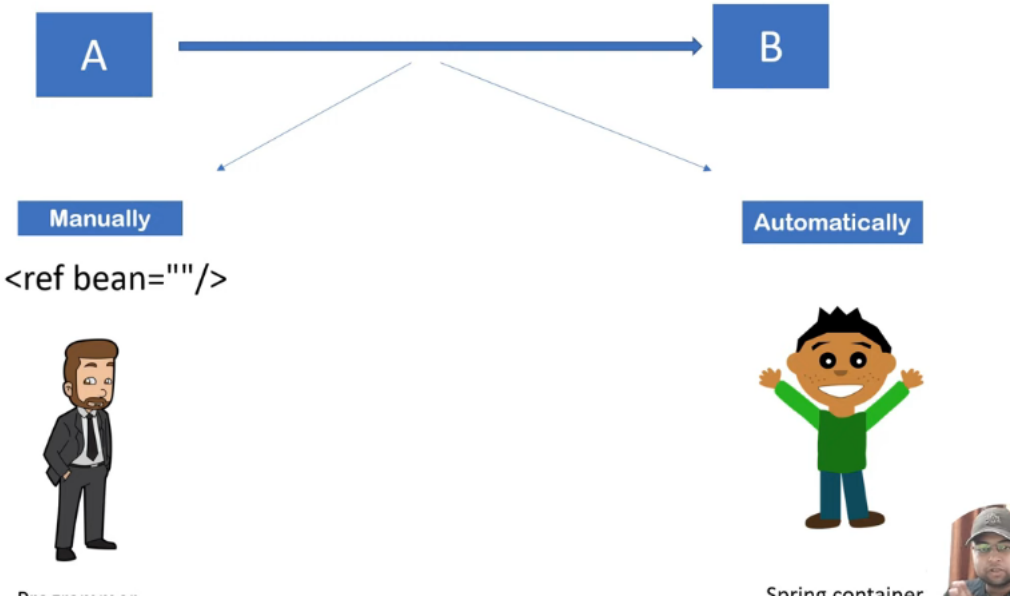
Scope Of Beans

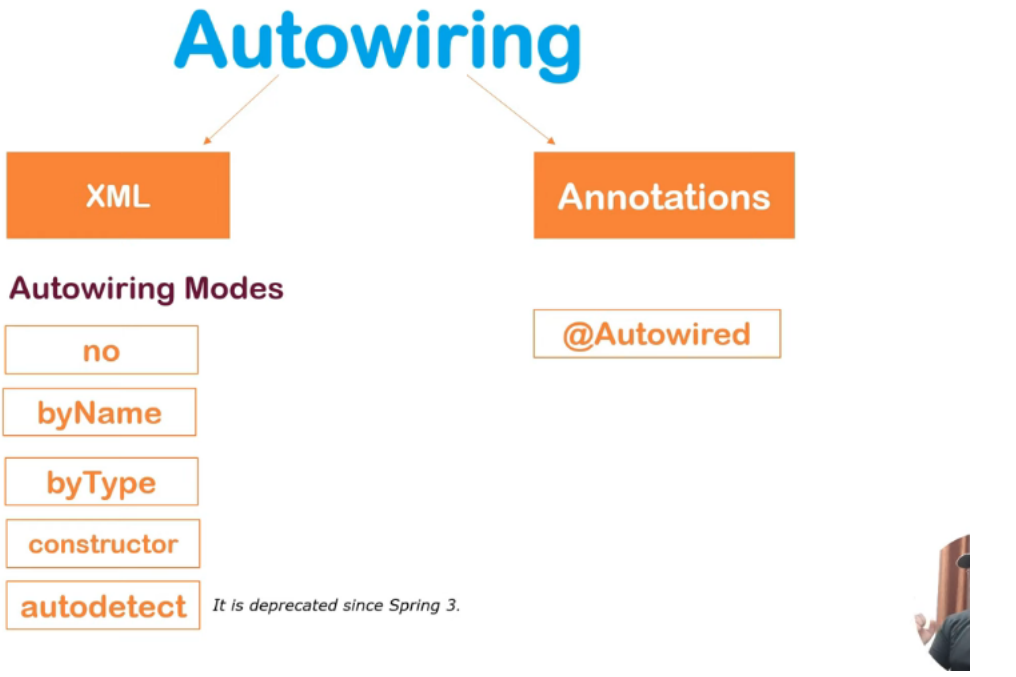
- **Singleton**: One instance per Spring IoC container (default).
- **Prototype**: A new instance each time the bean is requested.
- **Request**: A new instance per HTTP request (requires a web application context).
- **Session**: A new instance per HTTP session (requires a web application context).
- **Global Session**: A new instance per global HTTP session (specific to portlet-based web applications).

For `request` and `session` scopes to work, ensure you're running within a web application context with the necessary configurations and dependencies.

AutoWiring In DI

Automatically injects Dependecies into the Beans , only works for Reference Types





AW using XML
It only works for reference Types , there are some ways in ways u can inject Reference type dependencies automatically ,
<bean autowire="byName/byType"/>

byName
- byType, there should not be more than one bean of same type

you can also use autowire="constructor" but the name of variable and the name of bean provided should be same
AW using annotations @autowired

@autowired using three ways

@autowired works by checking the TYPE of the object not the name suppose you have multiple beans of the same type of object , then you can use the @Qualifier("nameOfThe Bean") annotation to explicitly define the bean to be injected

1) directly above property
2) above setter function
3) above constructor

StandAlone Collections in Spring

In Spring, standalone collections are collections (like lists, sets, and maps) that are defined and managed separately within the Spring configuration. These collections can be injected into beans, allowing for centralized and flexible management of grouped data.
can be created using **XML** or **Java Config @Bean tags**
using xml there are two ways :

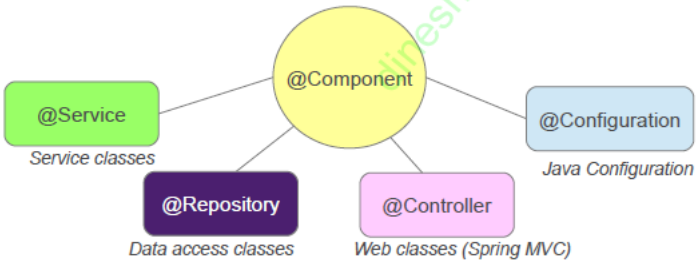
First one using util:

```
<util:list list-class="java.util.ArrayList" id="friends">
    <value > values </value>
  </util:list>
```

Second One using simple bean

```
<bean id="map" class="java.util.HashMap">
    <constructor-arg>
        <map>
            <entry key="First" value="1" value-type="int"/>
        </map>
    </constructor-arg>
</bean>
```

Stereotype Annotations/ Config Beans using Java



Stereotype Annotations in Spring

1. **@Component**
   - **Explanation**: General-purpose annotation indicating that a class is a Spring-managed component.
   - **Usage**: Used for any Spring-managed class not covered by more specific annotations.
2. **@Service**
   - **Explanation**: Indicates that a class provides business logic and services.
   - **Usage**: Used for service layer classes.
3. **@Repository**
   - **Explanation**: Indicates that a class is responsible for data access and interaction with the database.
   - **Usage**: Used for DAO (Data Access Object) or repository classes, providing automatic exception translation.
4. **@Controller**
   - **Explanation**: Indicates that a class is a web controller handling HTTP requests.
   - **Usage**: Used in Spring MVC applications to define controller classes.
5. **@RestController**
   - **Explanation**: Combines `@Controller` and `@ResponseBody`, indicating a RESTful web controller.
   - **Usage**: Used to create RESTful web services that return JSON or XML responses directly.

@Component // (same as <bean class="" .....) means that the objects of that class are being managed my spring
@Configuration is same as defining beans inside the configustion.xml file annotation which indicates that the class has **@Bean** definition methods i.e. it is the same as configure.xml file

> While accessing using java configuration , we will always get bean using **camelCasing** like if the name of class is Student then the bean name would be student (automatically created by spring)

```
@Component("BeanName")
public class Student {
// bean
}
config.xml
<context:component-scan base-package="com.SterotypeAnnotations"/>
```

BeanPostPorcessor

Object Created ---> add any SetterInjection ---> postProcessorBeforeInitiallization ---> init() ---> destroy() ---> postProcessorAfterInitiallization
BeanPostProcessors are interfaces that provide hooks into the Spring bean lifecycle, allowing for custom **processing before and after bean initialization.**

- `postProcessBeforeInitialization(Object bean, String beanName)`: Invoked before the bean is fully initi
- `postProcessAfterInitialization(Object bean, String beanName)`: Invoked after the bean has been fully i

SpEL (Spring Expression Language)

Its a way to write expressions that spring can understand and use to make decisions , get or set values and call methods.
Mostly the expression is passed in the **@Value** annotation

```
@Value ( "#{ expression }" )

SpelExpressionParser temp = new SpelExpressionParser();
temp.parseExpression("#{expression}")
```

Supports ***Classes , Variables , Methods , Constructors and Objects***
The expression should have a return value

> For method calling use this
> T(class).method(param)
> T(class).variable

How to create Object using SpEL ?

```
@Value( "#{ new Class() }")
@Value( "#{ new java.lang.String('ubuseyo') }" )
```

Java Based Configuration

@Configuration @ConfigurationScan @Bean
@Scope
use IOC Container

```
ApplicationContext context = new AnnotationConfigurationApplicationContext(Class.class);
```

@Configuration is the replacement of using config.xml file , it totally replaces xml file with a java class that defines all the components

```
@Configuration

@ComponentScan("package")

public class Demo{
        // Beans
}
```