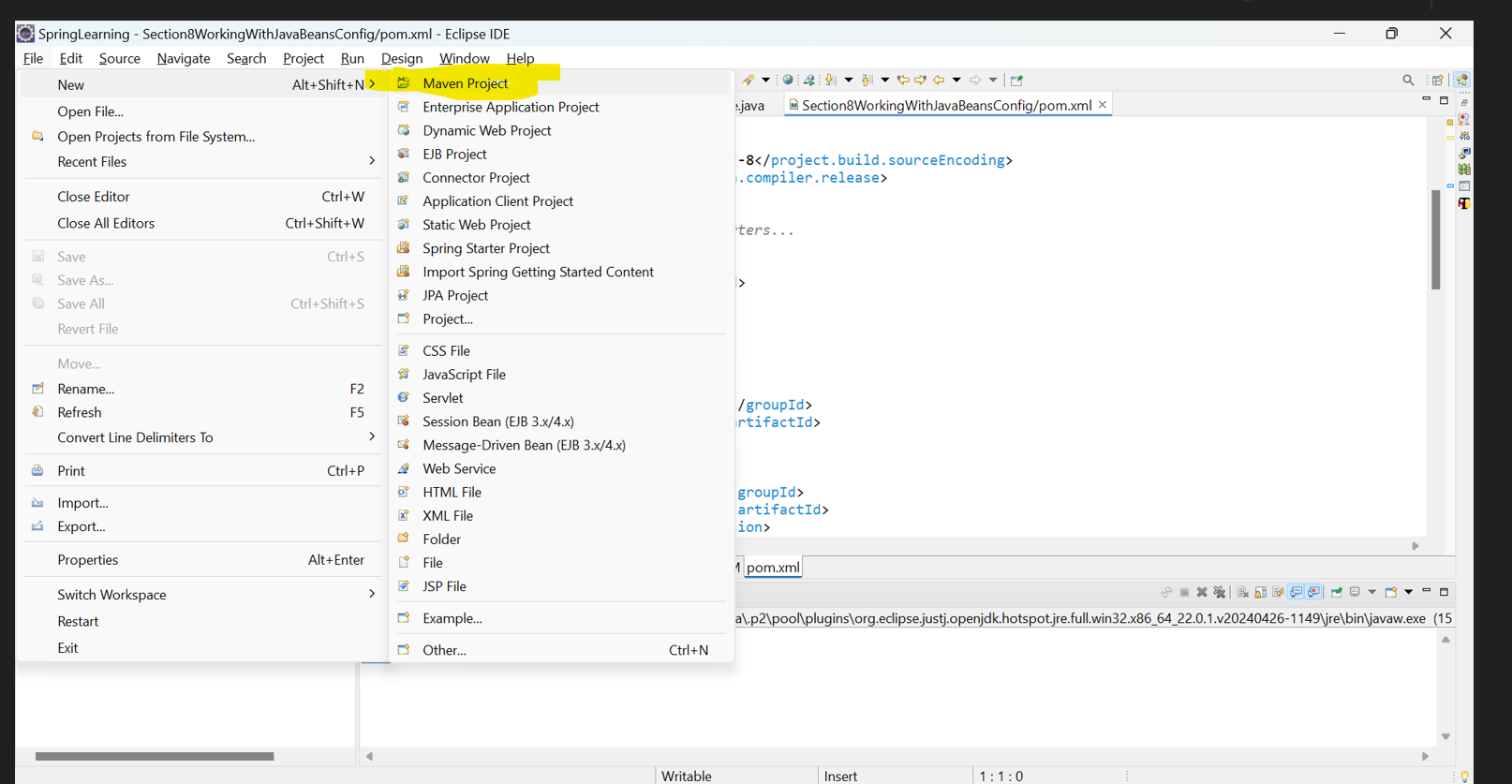
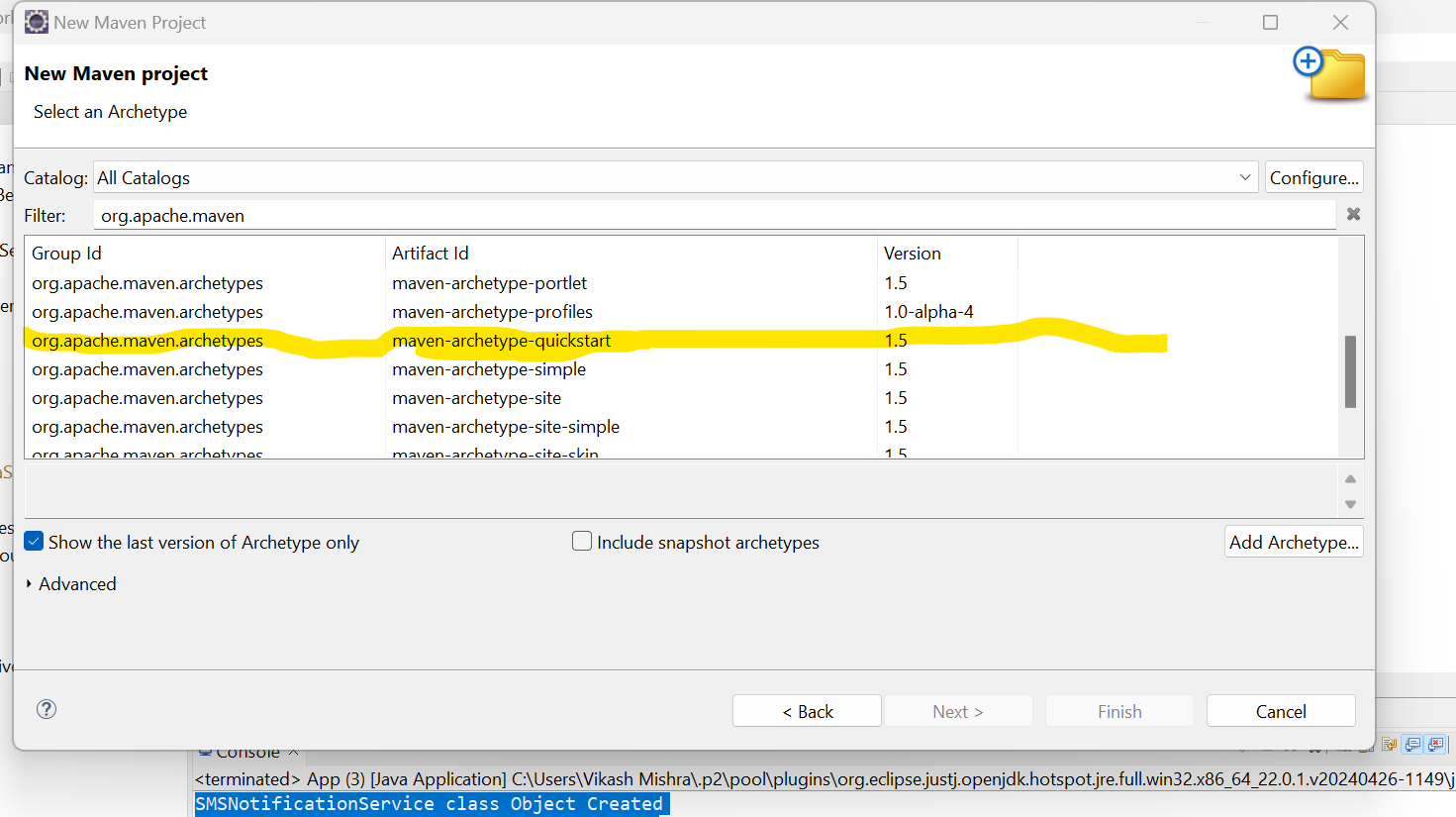
**Section 8: Working with Java Based Configuration**

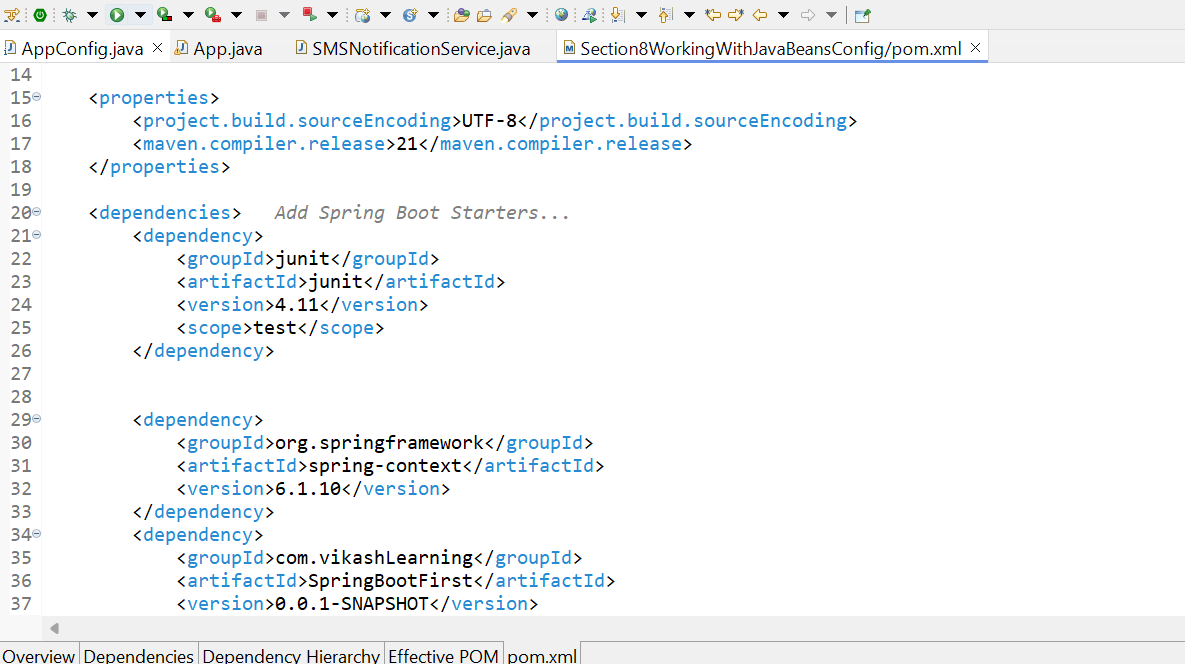
**Prerequisite**

**Project Creation**

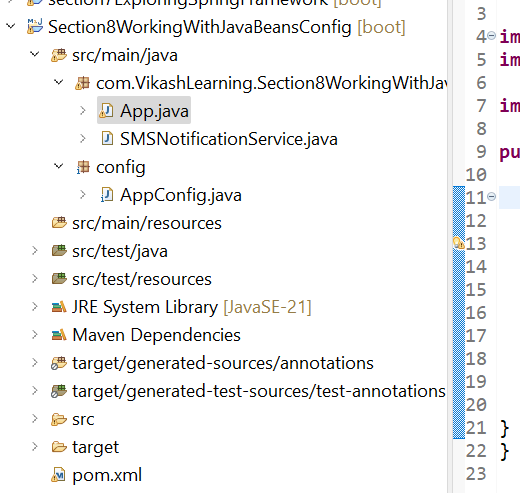
****

****

**POM.XML File**

****

**Project Structure**

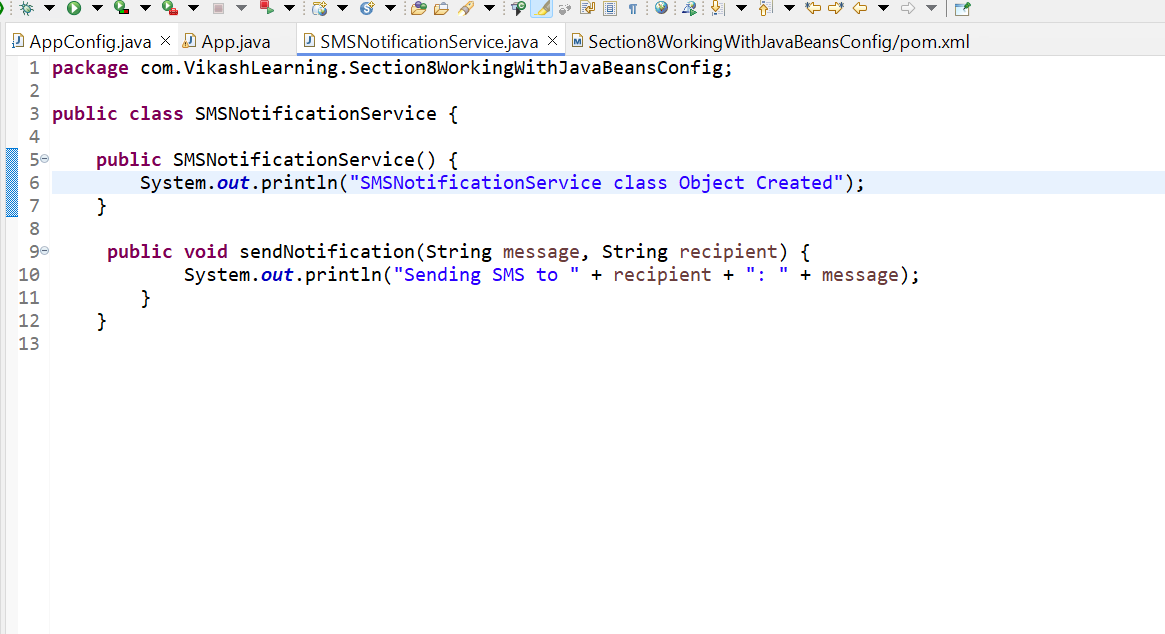


1. **Java Based Config**

we created a simple Spring application using Java-based configuration without interfaces or constructors for dependency injection. Let me walk you through the key concepts and steps involved:

1.1 **SMSNotificationService Class**:

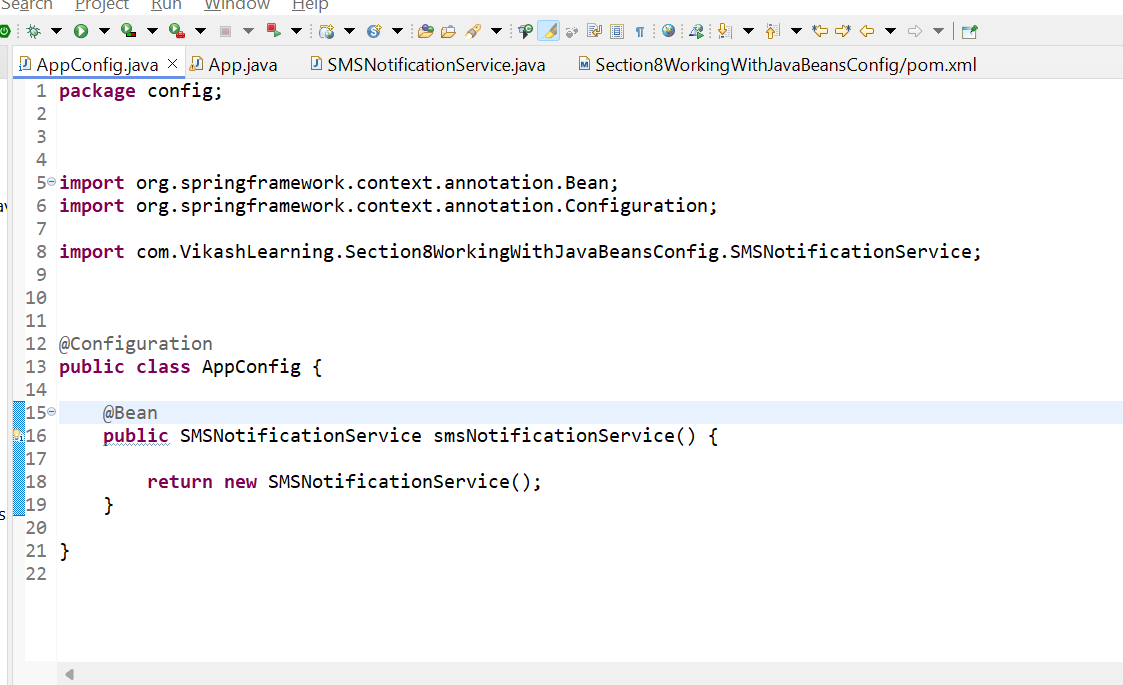
This is the main class responsible for sending SMS notifications. It has a method sendNotification() that takes two parameters: a message and a recipient.

****

* **Purpose**: This class contains the business logic to "send" an SMS message by printing it to the console.

**1.2 AppConfig Class**

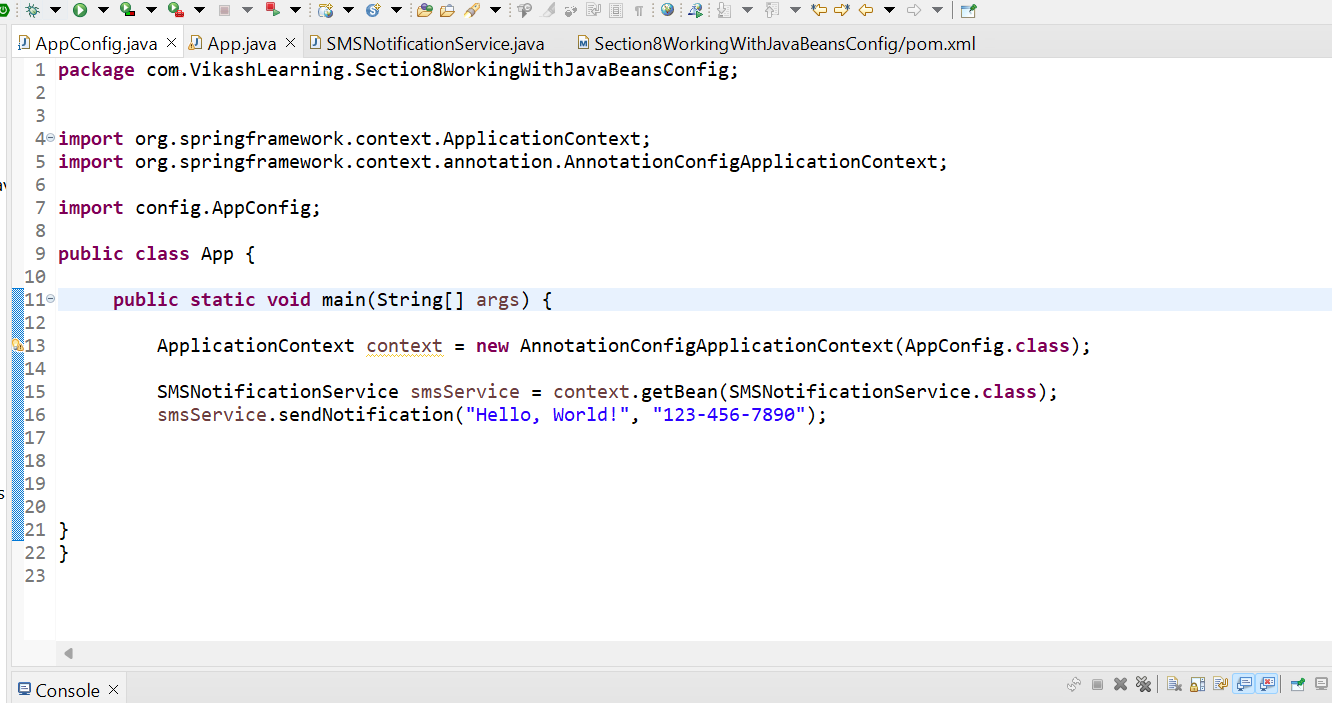
This class is annotated with @Configuration, telling Spring that it contains bean definitions. The smsNotificationService() method is annotated with @Bean, which tells Spring to create and manage an instance of SMSNotificationService**.**

****

* **Purpose**: This class defines how Spring should instantiate the SMSNotificationService object.
* **@Bean Annotation**: This annotation tells Spring to manage this particular object as a Spring bean. So, when Spring needs an instance of SMSNotificationService, it will call this method and provide the object.
* **No Interface or Constructor**: We directly return the object of SMSNotificationService, making it simpler and more direct.
  1. **App class**

This is the entry point of our application, where we load the Spring context and retrieve the SMSNotificationService bean. After that, we use this bean to send an SMS notification.

* **Purpose**: This class sets up the Spring context, which manages the beans and their lifecycle.
* **AnnotationConfigApplicationContext**: This class is used to load the Spring context from the AppConfig class (Java-based configuration). Spring scans AppConfig to identify the beans it needs to manage.
* **getBean() Method**: This method is used to retrieve the SMSNotificationService bean that Spring created for us.

****

**1.4 Flow of Execution:**

1. **AppConfig** class is loaded by Spring.

* Spring sees the @Bean annotated method smsNotificationService() and creates an instance of SMSNotificationService**.**

1. **App class** initializes the Spring context using annotationConfigApplicationContext(AppConfig.class), which loads the configuration and initializes the beans.
2. **getBean()** is used to fetch the SMSNotificationService bean from the Spring context.
3. The sendNotification() method is called on the SMSNotificationService bean, which prints the message to the console.

**1.5 Expected Output:**

Sending SMS to 123-456-7890: Hello, World!

1. **Bean Name**

The bean name will be the method name from the @Bean definition in the AppConfig class. In this case, the method name is smsNotificationService(), so the bean name will be "smsNotificationService".

If you want to pass the **bean name** explicitly when retrieving the bean, you can use the overloaded version of the getBean() method that accepts a String (bean name) and the class type.

Here’s how you can do that:

**2.1 Current AppConfig Class:**

The method name smsNotificationService() defines the bean name as "smsNotificationService".

@Configuration

public class AppConfig {

@Bean

public SMSNotificationService smsNotificationService() {

return new SMSNotificationService();

}

}

* 1. **Fetching the Bean by Name in App Class:**

In the App class, instead of using the class type to fetch the bean, you can use the bean name "smsNotificationService" explicitly.

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {

public static void main(String[] args) {

// Load Spring context using Java-based configuration

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Retrieve the bean by its name

**SMSNotificationService smsService = (SMSNotificationService) context.getBean("smsNotificationService");**

// Send a notification

smsService.sendNotification("Hello, World!", "123-456-7890");

}

}

**2.2.1 Explanation:**

* **context.getBean("smsNotificationService")**: Here, the getBean() method takes the bean name "smsNotificationService" and returns the corresponding bean. We then cast it to SMSNotificationService to use it.
* The bean name in this case comes from the method name in the AppConfig class: smsNotificationService().

**2.2.2 Expected Output**:

Sending SMS to 123-456-7890: Hello, World!

When using the getBean method with both the bean name and the class type, the method signature looks like this:

<T> T getBean(String name, Class<T> requiredType) throws BeansException;

Here's how it works in this case:

**2.2.3 Code Example**

Assuming you have the following configuration:

@Configuration

public class AppConfig {

@Bean

public SMSNotificationService smsNotificationService() {

return new SMSNotificationService();

}

}

And you want to retrieve the bean using both its name and class type:

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {

public static void main(String[] args) {

// Load Spring context using Java-based configuration

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Retrieve the bean by name and type

SMSNotificationService smsService = context.getBean("smsNotificationService", SMSNotificationService.class);

// Use the retrieved bean

smsService.sendNotification("Hello, World!", "123-456-7890");

}

}

**2.2.4 Explanation**

1. **Bean Definition**:
   * In the AppConfig class, the method smsNotificationService() defines a bean with the name "smsNotificationService" and the type SMSNotificationService.
2. **Retrieving the Bean**:
   * context.getBean("smsNotificationService", SMSNotificationService.class) retrieves the bean using the specified name and ensures it is of the required type. The method performs a lookup for the bean by name and then checks that the type matches the SMSNotificationService class. If it does not match, it throws a ClassCastException.

**2.2.5 Why Use This Method?**

1. **Type Safety**: It provides a way to get a bean by its name while also ensuring type safety, reducing the risk of ClassCastException.
2. **Explicit Bean Retrieval**: Useful in scenarios where you have multiple beans of the same type and need to specify the exact bean to retrieve by name.

**2.2.6 Expected Output**

Sending SMS to 123-456-7890: Hello, World!

The sendNotification method of the SMSNotificationService will be invoked, and the output will confirm that the message was sent to the specified phone number.

**2.3 Changing the Bean Name**

you can change the bean name in the Spring configuration. The bean name is the identifier used to reference the bean in the Spring container. Here’s how you can change the bean name in various scenarios:

**2.3.1 Java-Based Configuration**

If you are using Java-based configuration (with @Bean annotation), you can specify the bean name by providing a name to the @Bean annotation:

@Configuration

public class AppConfig {

@Bean(name = "customSmsNotificationService")

public SMSNotificationService smsNotificationService() {

return new SMSNotificationService();

}

}

In this case, the bean name is set to "customSmsNotificationService". To retrieve this bean, you would use:

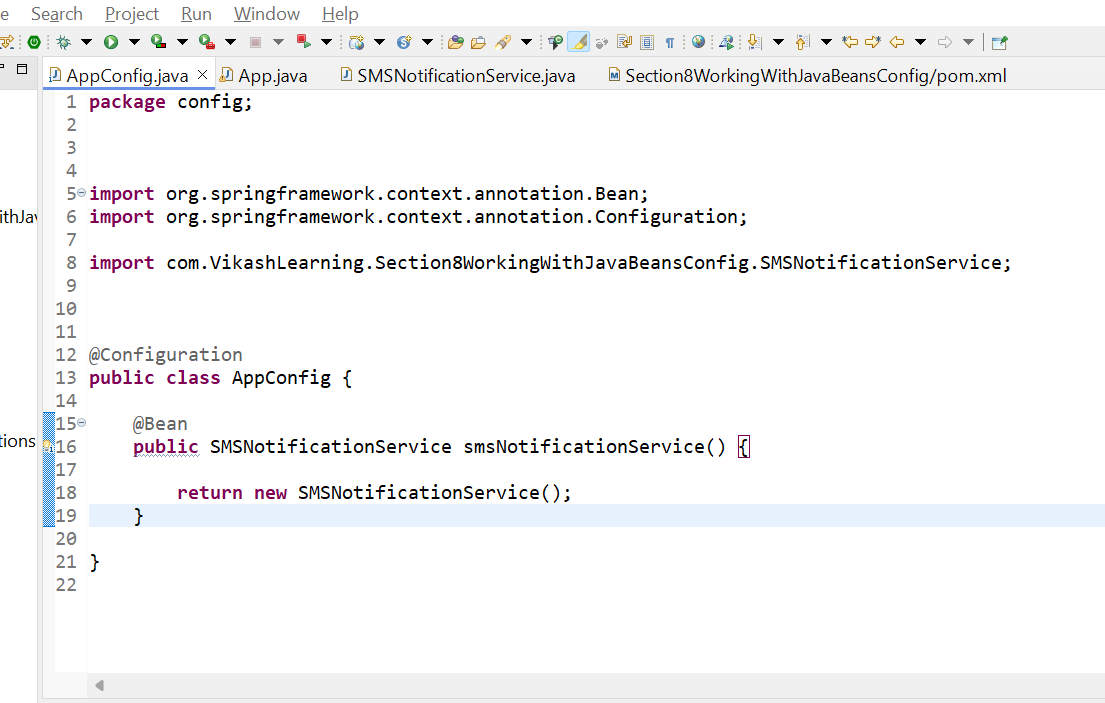
SMSNotificationService smsService = context.getBean("customSmsNotificationService", SMSNotificationService.class);

**3.Scope Annotation**

By Default, the Scope of annotation is Singleton that’s why whatever object we will create only one object will be created.

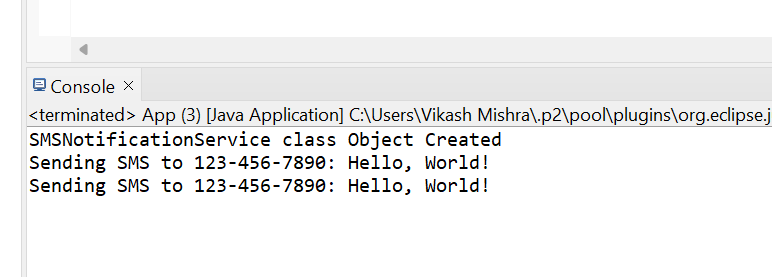
For Example:

Observe the below both classes and Output. We have not defined the scope of bean in AppConfig class and we have created two object of SMSNotificationService class but only one object gets created.



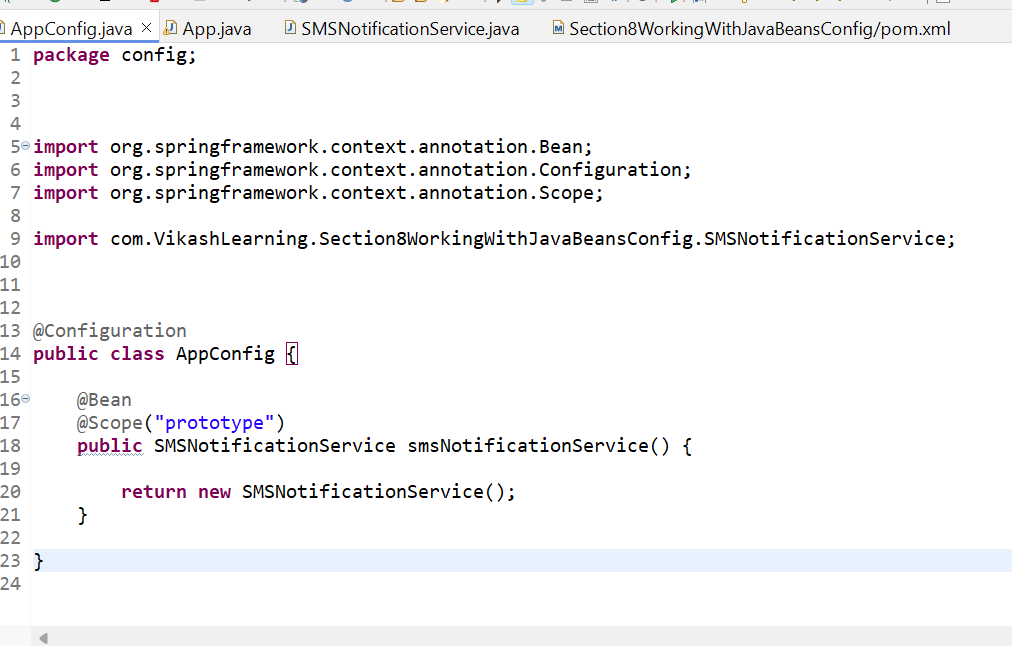


Output :

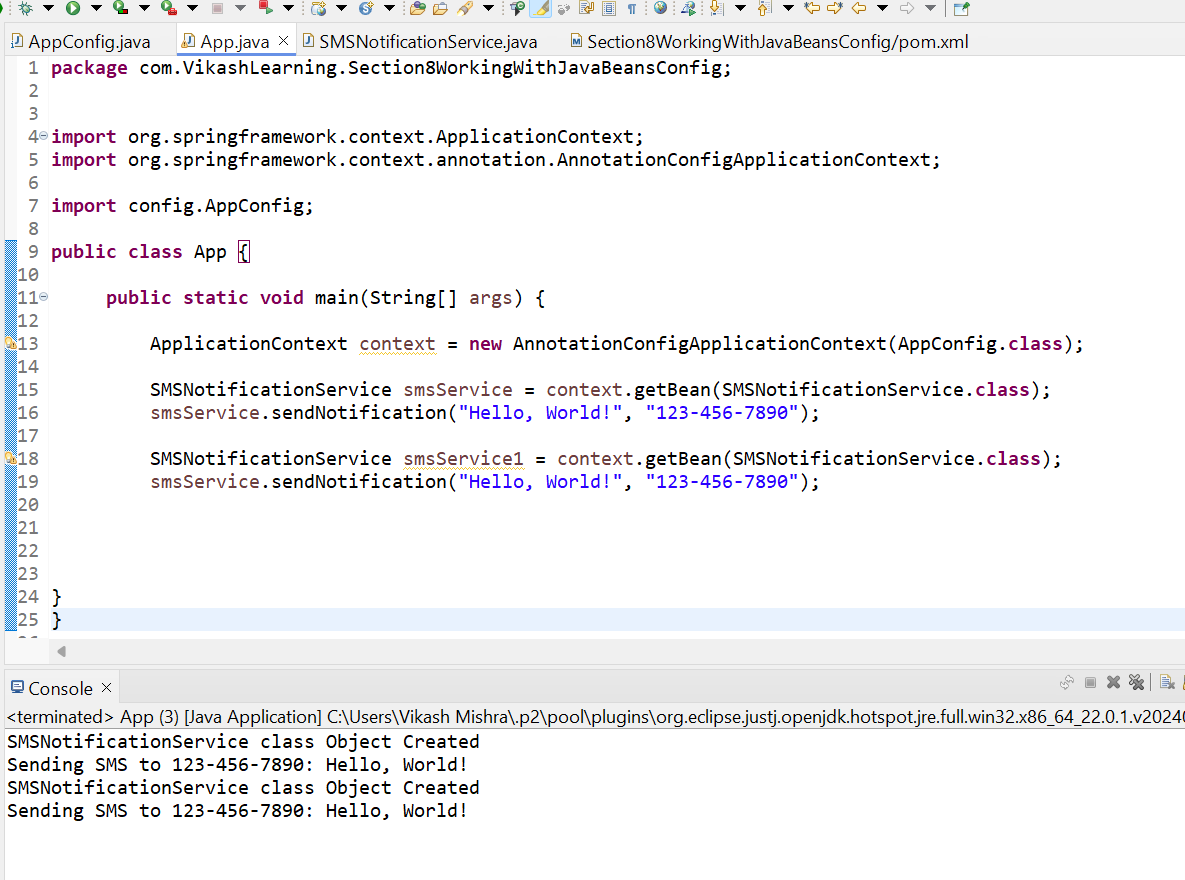


Now we will provide the scope of bean to prototype and we will see the object is created two times.

Lets see below Snipped of updated code:



Output



1. **Aurowire And @Qualifiers**

Autowiring in Spring is a feature that allows Spring to automatically resolve and inject collaborating beans into your bean, avoiding the need to explicitly define dependencies in the configuration. It helps to reduce the amount of configuration required and simplifies dependency management.

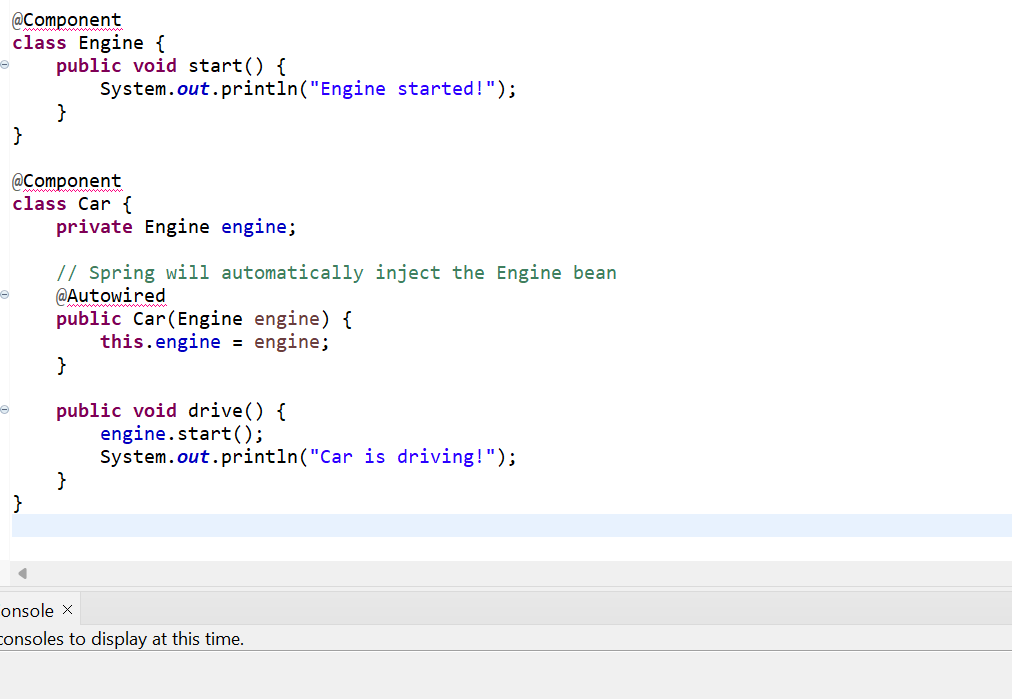
Here’s an overview of how autowiring works, including its different modes and how it is applied:

**3.1 Autowiring Modes**

1. **@Autowired Annotation**:
   * **By Type**: Spring tries to match the bean by its type. If there is exactly one bean of the required type, it is injected. If there are multiple beans, Spring throws an exception unless you specify which one to use.
   * **By Name**: Spring looks for a bean with a name that matches the property name in the class where it is being injected.
   * **By Qualifier**: Used in combination with @Autowired to specify the exact bean to be injected when there are multiple candidates.
   * **Constructor Injection**: Spring injects dependencies through the constructor of the class.

**Example** :

This annotation allows Spring to automatically inject (or "wire") dependencies into a class. Instead of manually creating objects, Spring will find the right bean (an instance of a class) and inject it where needed.

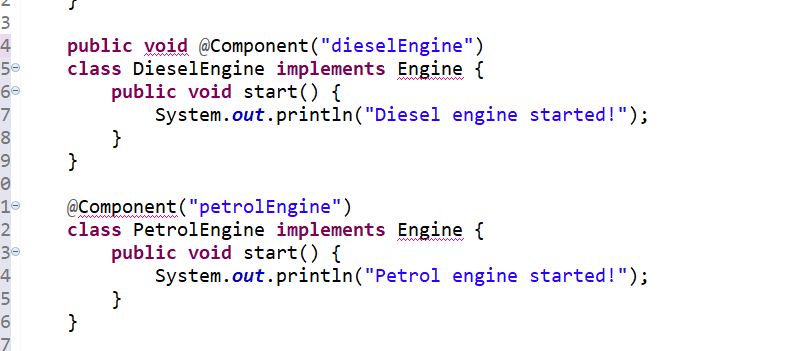
****

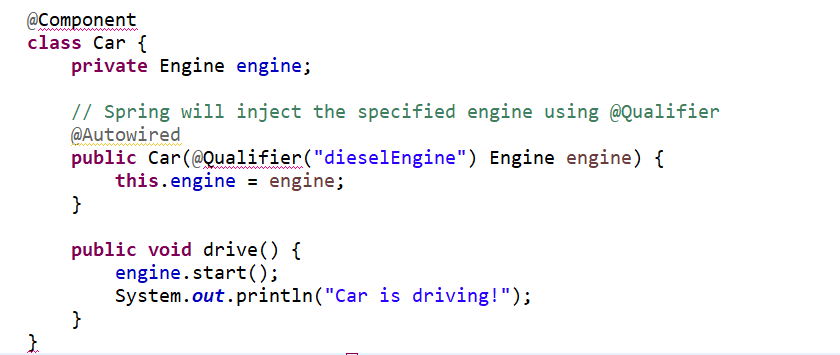
In this example, the Car class depends on Engine. The @Autowired tells Spring to inject an instance of Engine when the Car is created. You don't need to manually create the Engine object—Spring does it for you.

**@Qualifier:**

Sometimes, you may have more than one bean of the same type, and Spring won't know which one to inject. This is where @Qualifier comes in, as it allows you to specify exactly which bean to use.

**Example:**

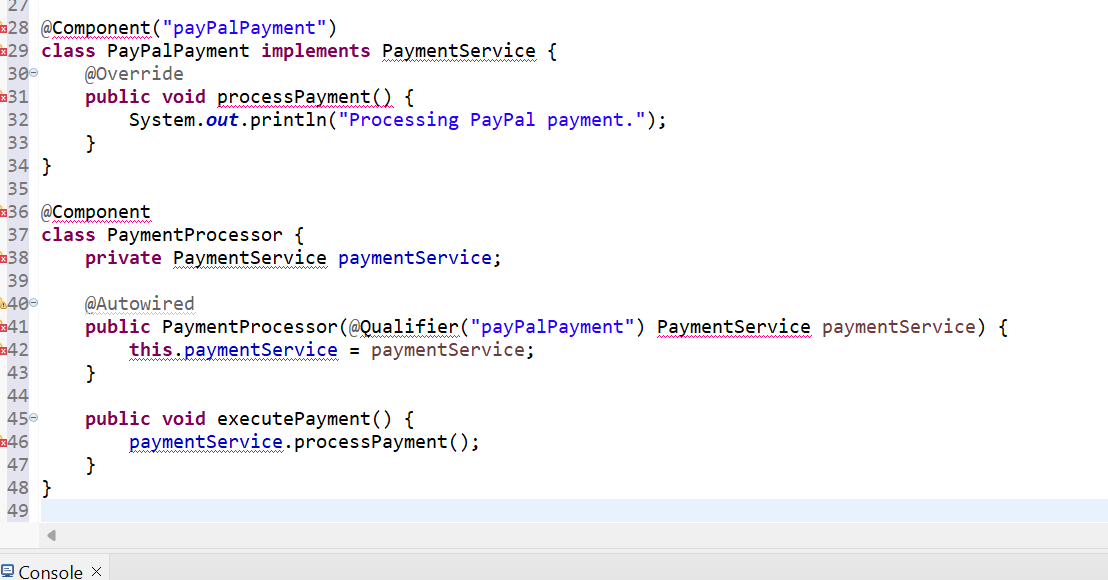
****

****

In this case, we have two types of engines: DieselEngine and PetrolEngine. To specify which engine to use, we use @Qualifier("dieselEngine"), and Spring will inject the diesel engine into the car.

**Real-Time Example:**

Imagine an e-commerce system where different payment methods are available. For instance, you may have classes to handle different payment types like CreditCardPayment and PayPalPayment. Both classes might implement the same PaymentService interface.



Here, the PaymentProcessor class can process payments, and by using @Qualifier("payPalPayment"), Spring knows to inject the PayPalPayment implementation. You could easily switch to CreditCardPayment by changing the @Qualifier.

In summary:

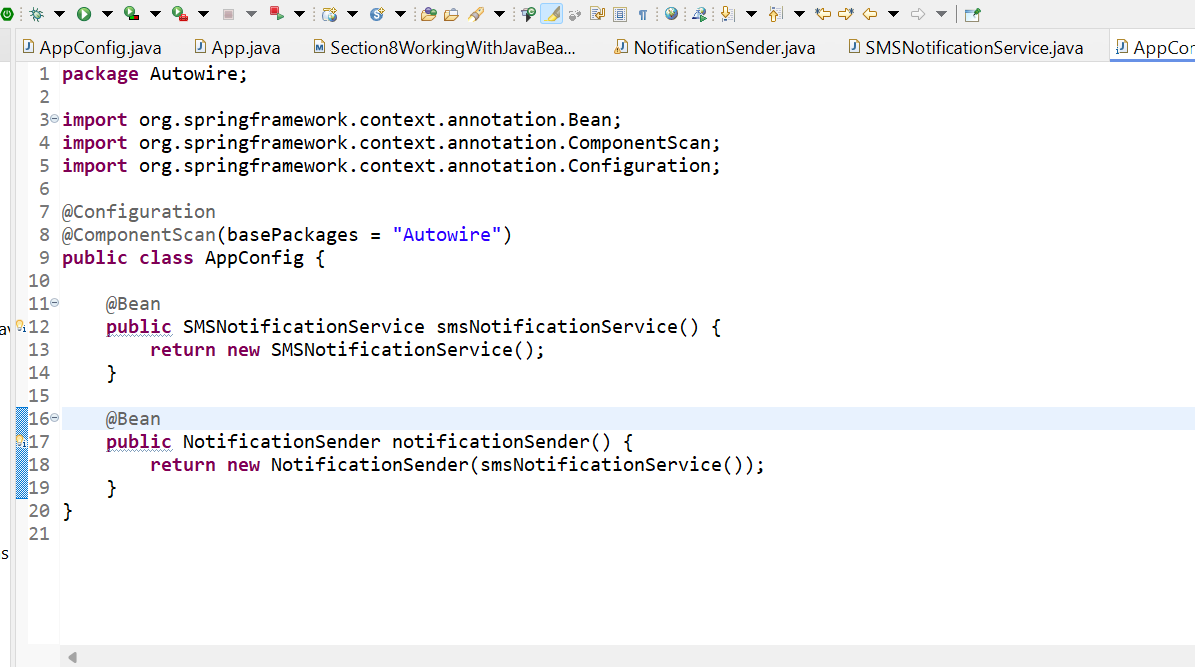
* @Autowired automatically injects a bean into your class.
* @Qualifier is used to specify which bean to inject when multiple options are available.

**AppWise Example:**

App.java class



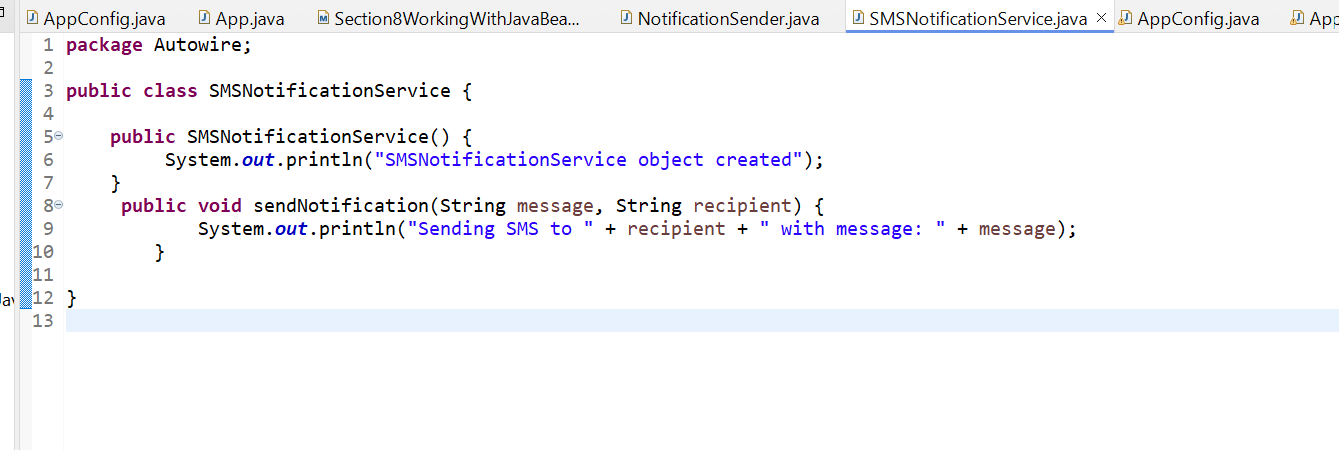
AppConfig.java Class



MNotificationSender.java class



SMSNotificationService.java class



**3.2 Usage**

In both configurations, when you get the NotificationSender bean from the Spring context and call its send method, it will use the injected SMSNotificationService to send the notification.

* 1. **Key Points**
  + **Autowiring** helps to automatically resolve and inject de pendencies, reducing configuration effort.
  + **Types of Autowiring** include byType, byName, and constructor, each serving different use cases.
  + **Annotations** (@Autowired, @Qualifier) and XML settings (autowire) are used to specify how dependencies should be injected.

Autowiring improves the maintainability and flexibility of your code by reducing the need for manual configuration and promoting loose coupling between components.

1. **@Primary**

**4.1 What is @Primary in Spring?**

In Spring, @Primary is an annotation used to indicate the default bean that should be injected when there are multiple beans of the same type. If you don't specify a bean explicitly using @Qualifier and there are multiple beans of the same type, Spring will choose the bean marked as @Primary.

It solves the issue of bean ambiguity, which arises when Spring finds more than one bean of the same type, and it doesn't know which one to inject.

* 1. **Why Use @Primary?**

When you have multiple beans of the same type in the Spring context, and you don't want to specify a particular bean each time, marking one of them with @Primary makes it the default choice.

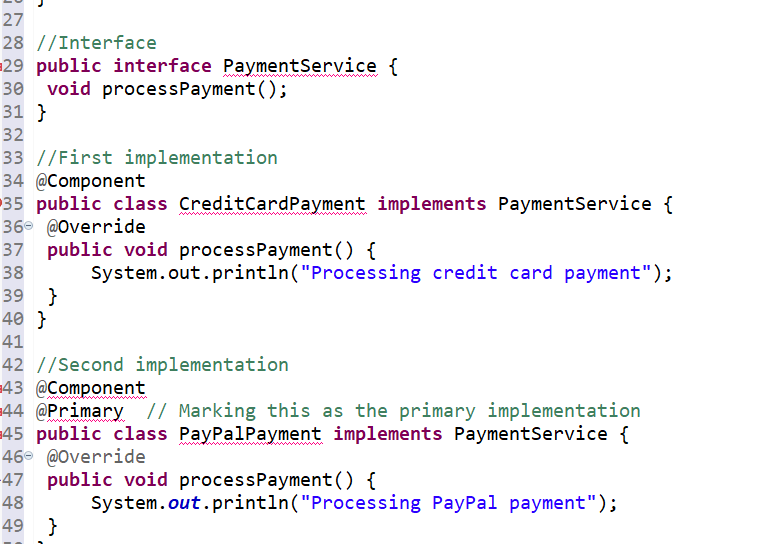
The @Primary annotation in Spring is used to indicate which bean should be given priority when multiple beans of the same type are available in the Spring container. It helps resolve ambiguity when Spring is trying to autowire a dependency, but more than one candidate bean exists for that type.

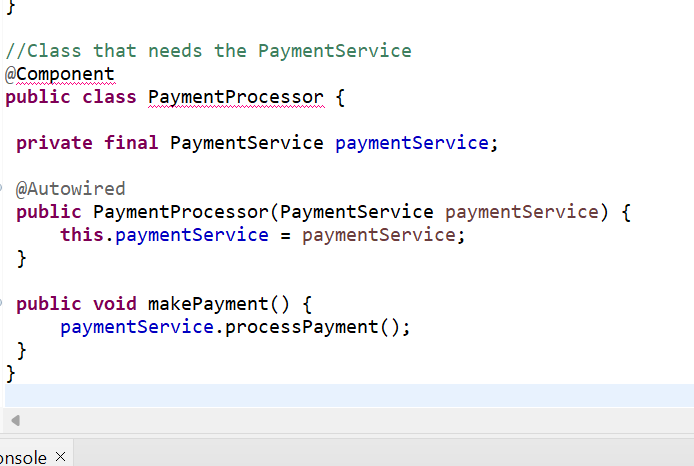
**4.3 How @Primary Works:**

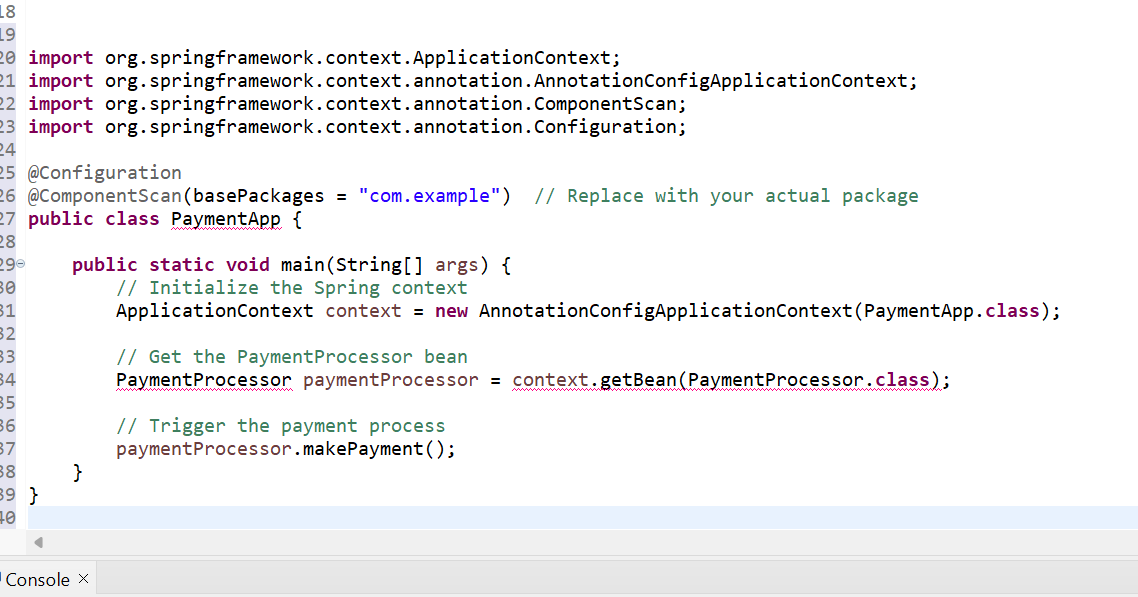
When you have multiple beans of the same type, and Spring tries to inject one of them into a class, it will face a conflict if no specific bean is mentioned (using @Qualifier or any other mechanism). The @Primary annotation tells Spring which bean should be chosen by default if no specific qualifier is provided.

**4.4 Example of @Primary:**

Let’s consider a scenario with two different implementations of a PaymentService: CreditCardPayment and PayPalPayment. In this case, we can use @Primary to tell Spring which one to pick by default when there's ambiguity.

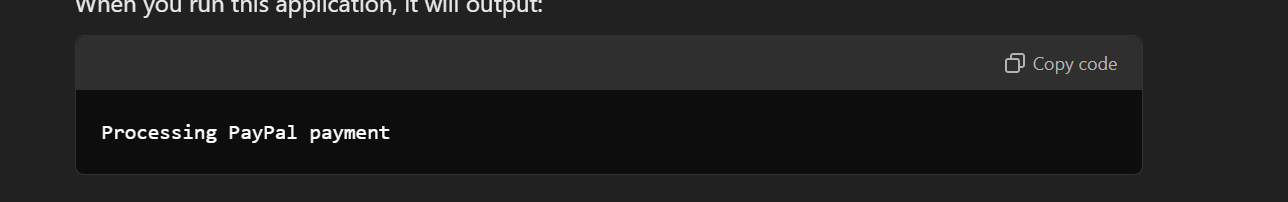






**4.5 Explanation:**

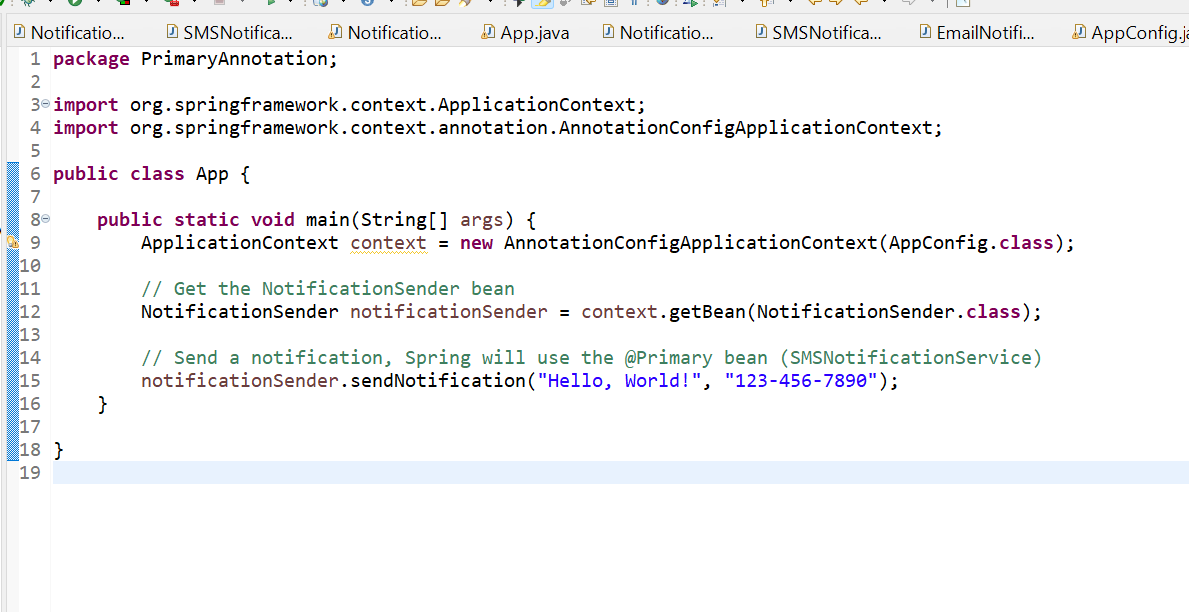
1. **@Configuration:** Marks the class as a Spring configuration class.
2. **@ComponentScan:** Tells Spring to scan the specified package for components (like @Component, @Service, etc.). Replace "com.example" with the actual package where your beans are located.
3. **ApplicationContext:** This is the Spring container that holds the beans and their lifecycle.
4. **AnnotationConfigApplicationContext:** This is used to initialize the Spring context based on Java configuration (annotations in this case).
5. **PaymentProcessor:** It fetches the PaymentProcessor bean from the Spring container and calls the makePayment() method, which will trigger the default payment process (in this case, PayPal since it’s marked with @Primary).



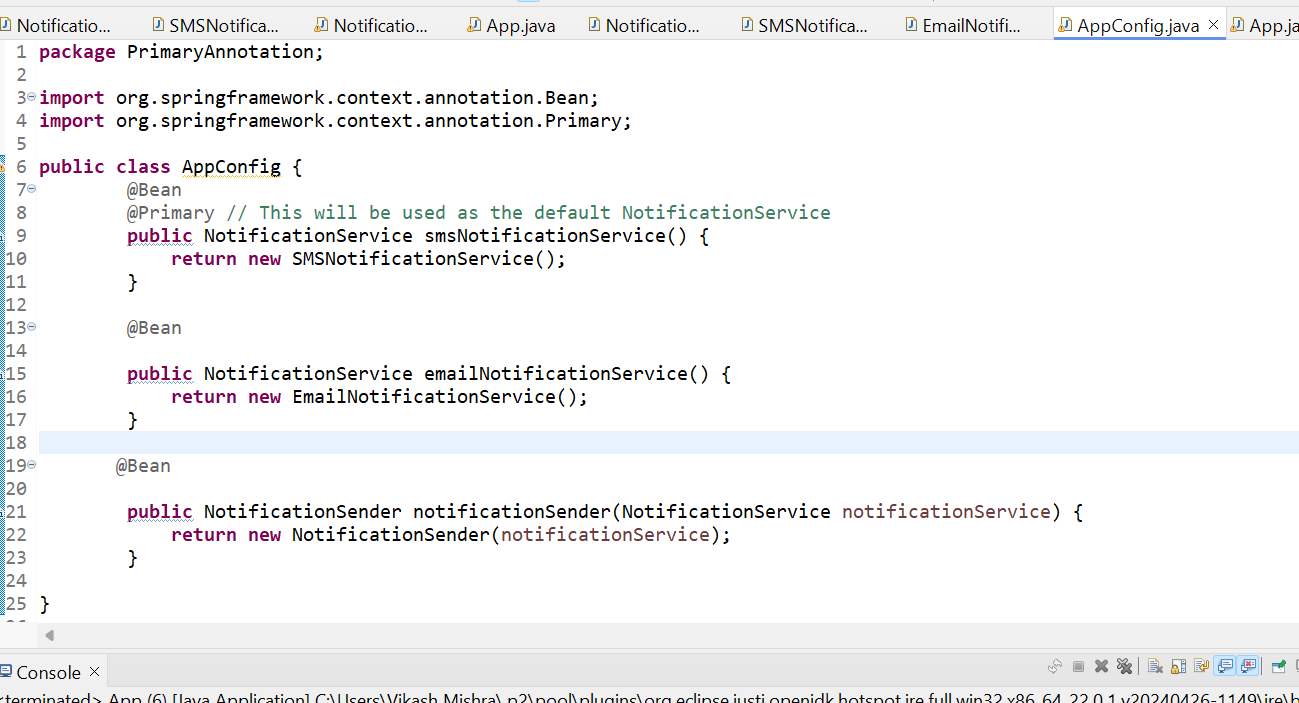
**4.6 Example Scenario**:

* + Let’s say you have two beans of type NotificationService: one for SMS and one for Email. Without @Primary, Spring would throw an error because it doesn’t know which bean to use when injecting a NotificationService. By marking one bean as @Primary, you make it the default.

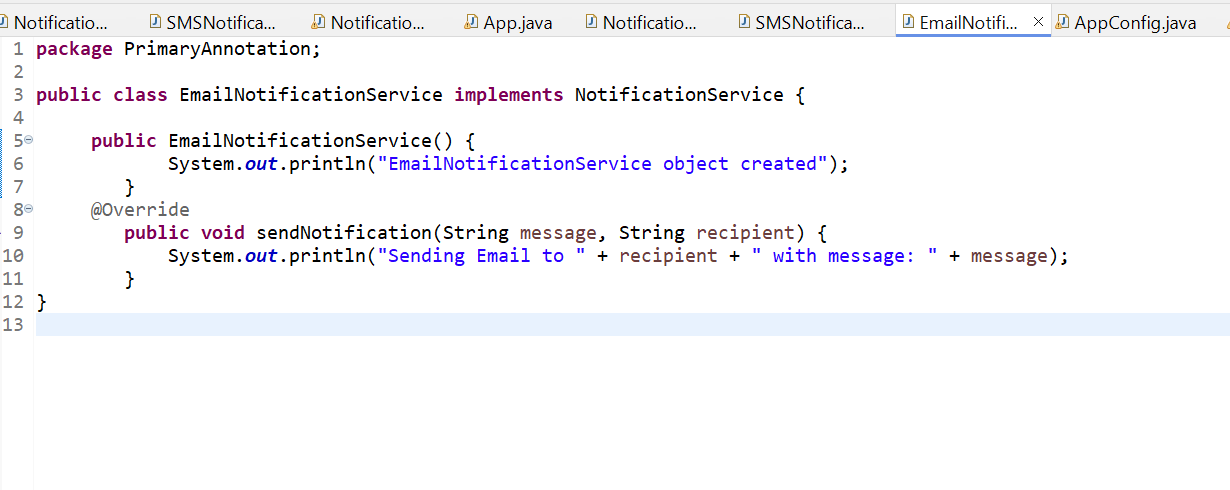
App.java

****

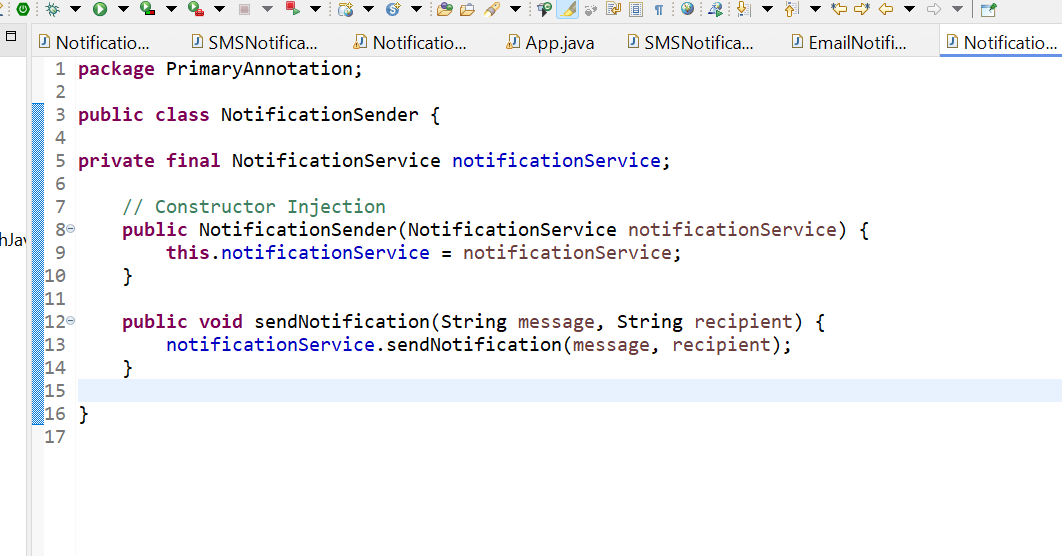
AppConfig.java

****

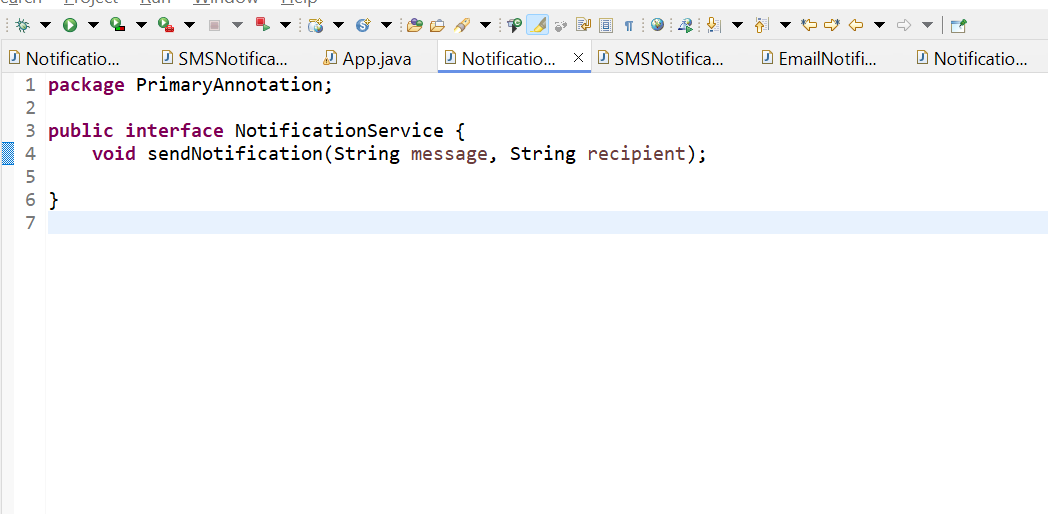
EmailNotificationService.java

****

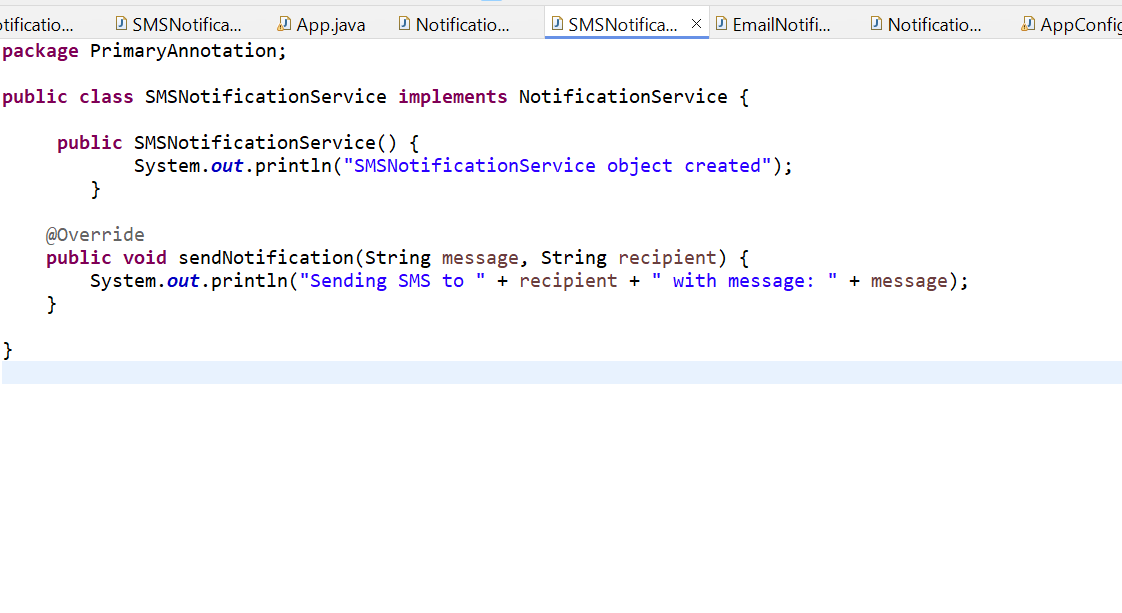
NotificationService.java



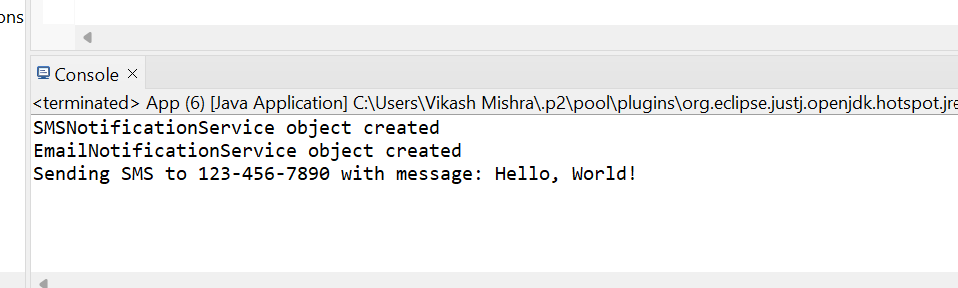
NotificationService



SMSNotification.jaava



O/p



**4.6 Step-by-Step Execution:**

1. **Spring Context Initialization**:
   * Spring loads the AppConfig class and identifies three beans:
     + smsNotificationService (which is @Primary)
     + emailNotificationService
     + notificationSender
2. **Bean Creation**:
   * Spring creates an instance of SMSNotificationService and EmailNotificationService.
   * It then creates an instance of NotificationSender, which needs a NotificationService.
3. **Dependency Injection**:
   * For NotificationSender, Spring injects the NotificationService bean. Since there are two beans of type NotificationService, Spring uses the @Primary annotation to choose SMSNotificationService.
4. **Calling the Method**:
   * When the sendNotification() method is called on the NotificationSender, it internally calls the sendNotification() method of the SMSNotificationService (since it's the primary bean).

## **5.Stereotype Annotations with Example using @Component and @ComponentScan**

Stereotype annotations in Spring Framework, such as @Component, @Service, @Repository, and @Controller, are used to automatically detect and register classes as beans in the Spring application context. These annotations reduce the need for explicit bean configuration in XML or Java-based configuration by allowing Spring to scan packages and manage the classes automatically.

Here's a detailed explanation with the example using the @Component and @ComponentScan annotations.

### 5.1 Key Stereotype Annotations in Spring:

### @Component:

### Generic annotation to mark a class as a Spring-managed component.

### Automatically registers the class as a Spring bean.

### @Service:

### Specialized version of @Component to indicate that the class contains business logic or service layer functionality.

### @Repository:

### Used to indicate that the class interacts with the persistence layer, such as database operations.

### Spring provides exception translation for this annotation (e.g., converts database exceptions to Spring's DataAccessException).

### @Controller:

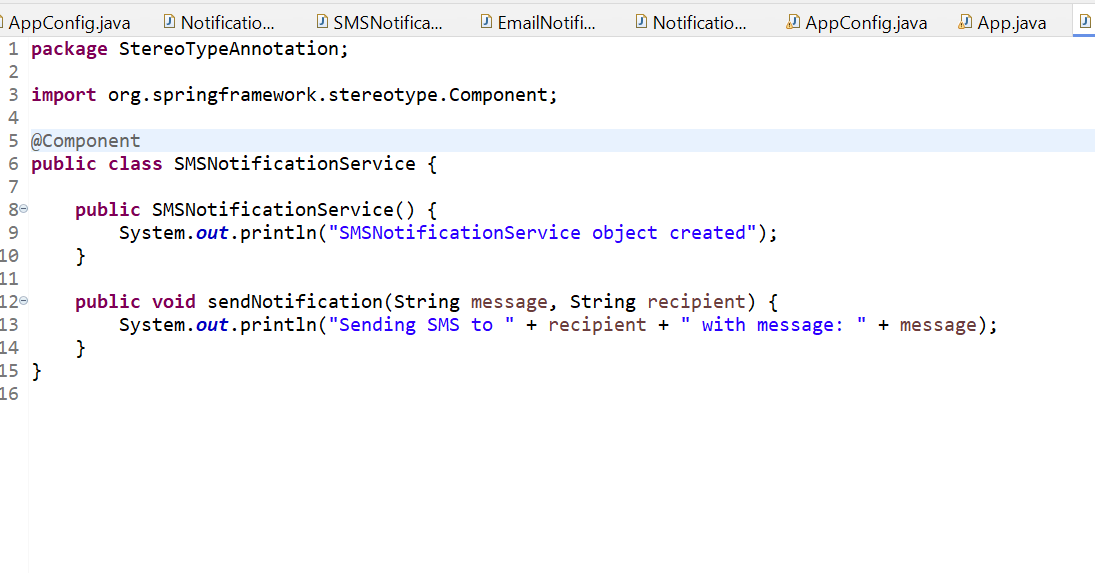
### Used to mark a class as a Spring MVC controller that handles HTTP requests and sends responses in web applications.

### 5.2 Example Using @Component and @ComponentScan

In the previous example, we used manual bean registration using @Bean. Now, let’s refactor the same example by utilizing stereotype annotations (@Component) to automatically detect and register beans.

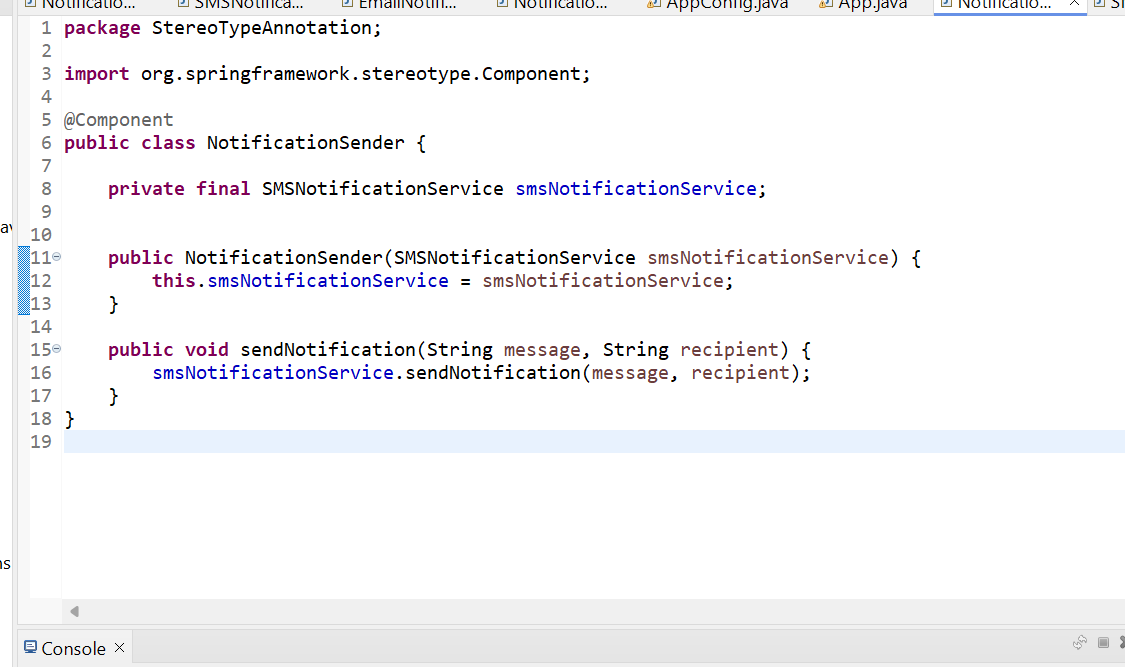
SMSNotificationService.java

By annotating SMSNotificationService with @Component, we let Spring automatically register this class as a bean.



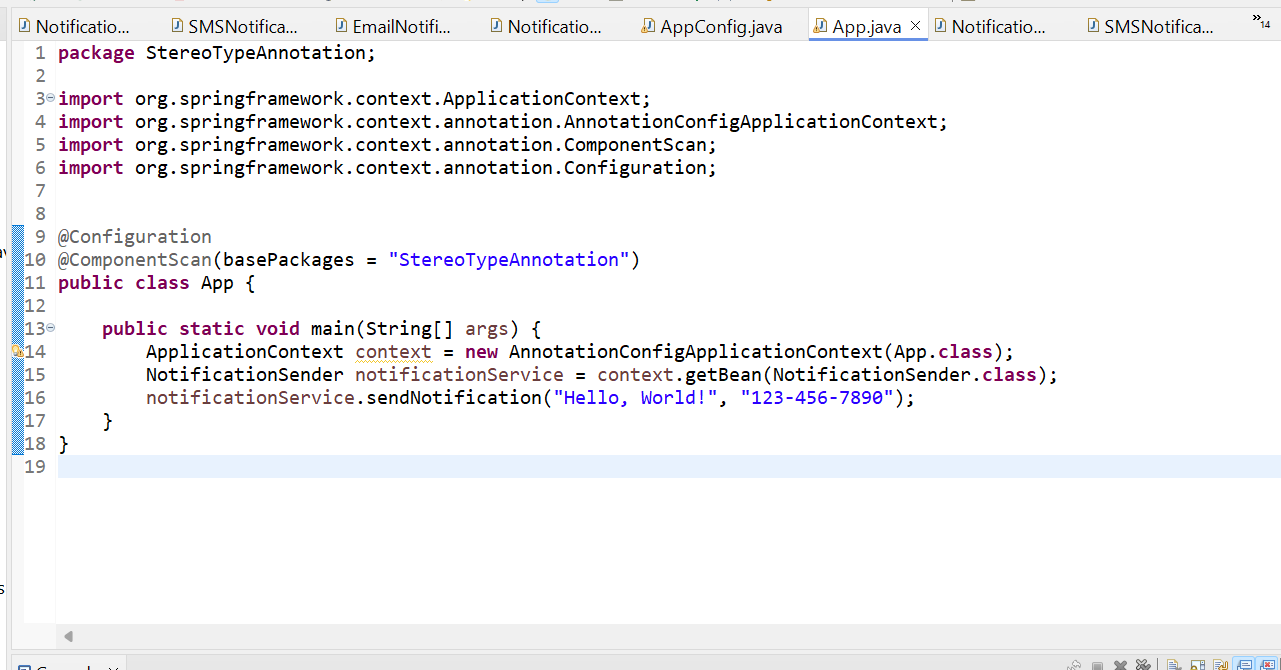
NotificationSender.java

NotificationSender is marked with @Component to indicate that it should be managed by Spring.

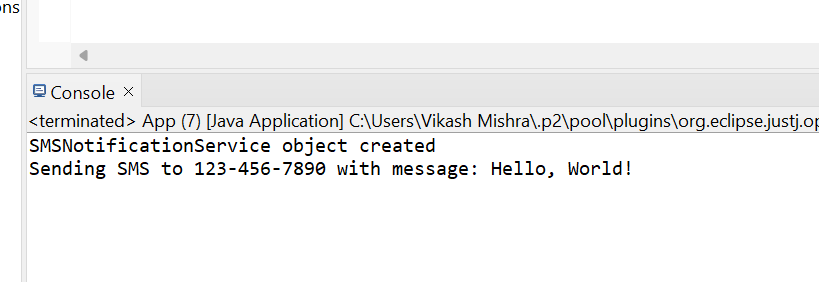


App.java

* + The @ComponentScan annotation tells Spring to scan the specified package for classes annotated with @Component, @Service, etc.
  + In this case, it will detect both SMSNotificationService and NotificationSender.



O/p



**Key Points:**

1. **@Configuration**:
   * Marks the App class itself as the configuration class.
   * This annotation tells Spring that the App class will contain bean definitions or other configuration setups.
2. **@ComponentScan**:
   * Tells Spring to scan the specified package (Autowire) for classes annotated with @Component, @Service, @Repository, etc.
   * This enables Spring to automatically discover and register beans.
3. **No Need for AppConfig**:
   * The AppConfig class is no longer necessary, as the configuration and component scanning are handled by the App class itself.

**Execution Flow:**

1. **Spring Bootstrapping**:
   * When new AnnotationConfigApplicationContext(App.class) is executed, Spring will treat the App class as both the entry point and the configuration class.
2. **Component Scanning**:
   * Spring will scan the Autowire package for beans, just like before.
3. **Dependency Injection**:
   * Spring will inject the SMSNotificationService bean into the NotificationSender class as part of the auto-wiring process.
4. **Sending Notification**:
   * The sendNotification() method in NotificationSender is called, and it uses the SMSNotificationService to send the notification.