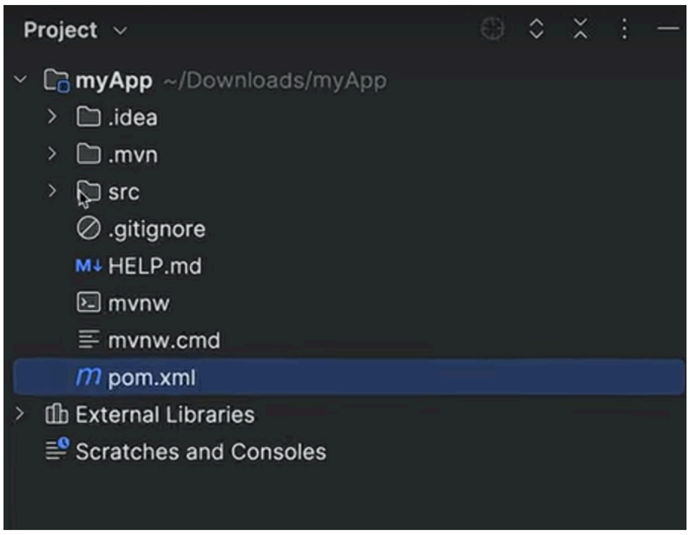
Dependency Injection using Spring Boot

Dependency Injection (DI) is a **design pattern** in Spring to facilitate the creation of dependent objects outside of a class and provide those objects to a class in various ways. This approach enhances the flexibility, testability, and maintainability of the code.

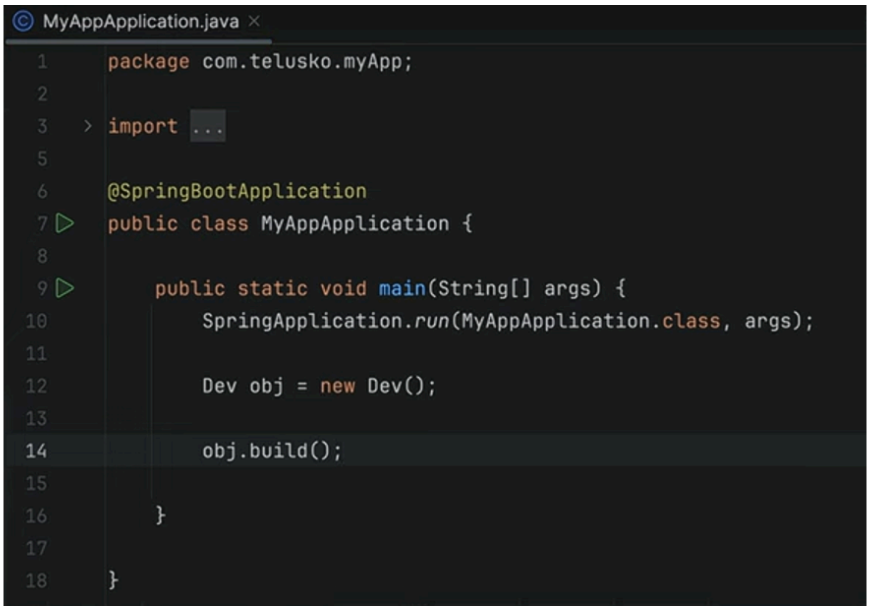


```
Project
myApp ~/Downloads/myApp
  .idea
  .mvn
  src
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
External Libraries
Scratches and Consoles
```

Setting Up the Project

Begin by creating a new Spring Boot project using Spring Initializer or your preferred IDE. Ensure you include necessary dependencies, such as spring-boot-starter-web. The main method of our application contains **`SpringApplication.run(MyApp.class,args )`**

Understanding IoC Container



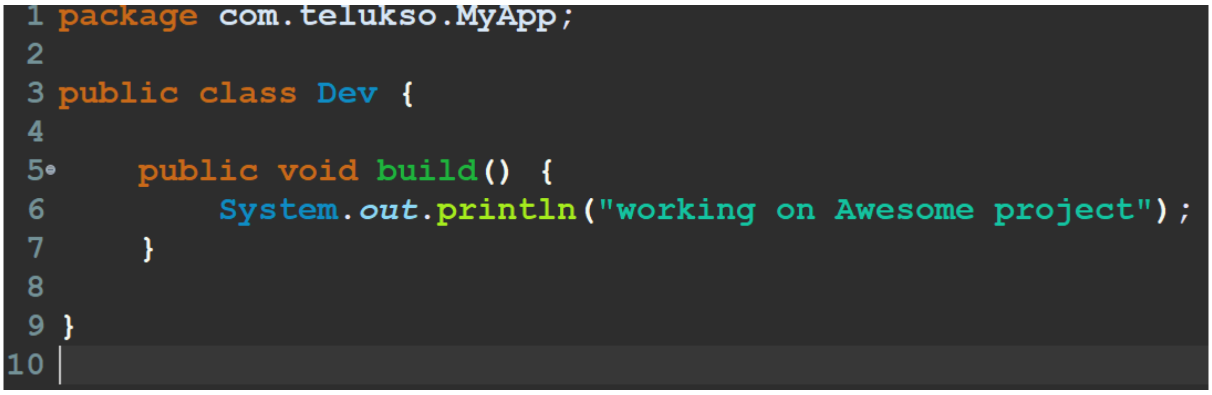
MyAppApplication.java

```java
package com.telusko.myApp;

import ...

@SpringBootApplication
public class MyAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyAppApplication.class, args);

        Dev obj = new Dev();

        obj.build();

    }

}
```

In a simple Java application, when an object of a class is created, it is instantiated inside the JVM. However, in the Spring framework, calling the run method creates an IoC (Inversion of Control) container inside the JVM. This container manages the creation of objects. In a project having 1000's of classes, even if only a few class objects are needed, the IoC container can manage those classes in the project, creating only the necessary objects as required. This process is efficiently handled by the IoC container.

Creating and Managing Beans

Create a class named Dev and a method named Build in it. Injecting dependencies into a target class is part of DI. Annotations make this easier to do in Spring Boot. In our project's main method, we need an instance of the Dev class to call the build method. It is possible to directly create this object and call the method, but this is not the best way to do things. Instead of object lifecycle management, developers should work on business logic. In this case, Spring's features become very useful.

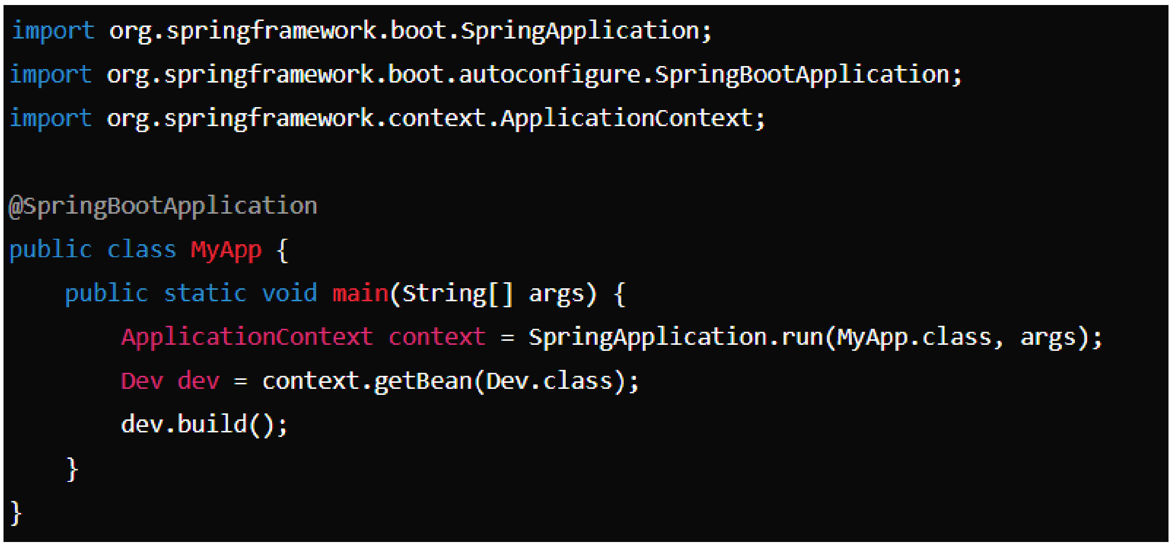


```java
package com.telukso.MyApp;

public class Dev {

    public void build() {
        System.out.println("working on Awesome project");
    }

}

```

In the Spring project, the main application class, **MyApp**, contains the main method, which calls the run method to initiate the application. To illustrate DI, the Dev class is created with a build method that prints "Awesome, you are building."

To access the build method from the Dev class within the main method, utilize the **ApplicationContext** object, which is part of the IoC container. By annotating the Dev class with **@Component**, Spring is instructed to manage this class as a bean, making it available for dependency injection.

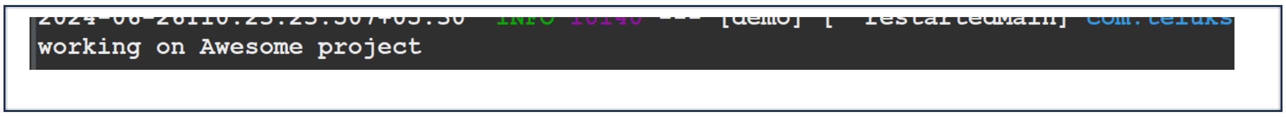
The run method in Spring Boot returns an ApplicationContext object, which is used to retrieve beans managed by the Spring container.



```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(MyApp.class, args);
        Dev dev = context.getBean(Dev.class);
        dev.build();
    }
}
```

.

And hence we get the output.



```
2024-08-26T10:23:23.307+05:30  INFO 10140 --- [demo] [  restartedMain] com.telusk
working on Awesome project
```

.

Types of dependency injection:

1. Constructor-based dependency injection
2. Setter-based dependency injection
3. Field-based dependency injection

1) Constructor-based dependency injection:

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

2) Setter-based dependency injection:

-Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

3) Field-based dependency injection:

This injection is least used; dependencies in field injections are directly added into the fields using **@Autowired** annotation, and it also does not support immutability, which is why it is less preferred over other injections.

Benefits of Dependency Injection

- **Decoupling**: It decouples the code, making it easier to manage and test.
- **Easier Testing**: Since dependencies are injected, it's easier to mock dependencies during unit testing.
- **Flexibility**: Allows for more flexible and reusable code.

By using Dependency Injection, Spring Boot makes it easier to manage your application's components and their dependencies, enhancing the maintainability and testability of your code.