Java Spring Framework Dependency Injection Without Spring Boot

Dec 17, 2020

Today, Spring Framework is used as an umbrella term for Spring projects and Spring eco system, However, Spring Framework is basically a **Dependency Injection** Framework and we can use its Dependency Injection capabilities in *any* Java project, without using Spring Boot or any other Spring project.

Example Java Project

Let's say we have a Java project with an `EmployeeService` that has a dependency on `DepartmentService` and an `Application` class to run the application:

```
package com.thebackendguy.com.service;

public class DepartmentService {

    public String getDepartmentName(String employeeId) {
        System.out.println(this + ": getDepartmentName");
        return "Accounts";
    }
}
```

```
package com.thebackendguy.com.service;

public class EmployeeService {

    private final DepartmentService departmentService;

    public EmployeeService(DepartmentService departmentService) {
        this.departmentService = departmentService;
    }

    public String checkDepartment() {
        System.out.println(this + ": checkDepartment");
        return departmentService.getDepartmentName("EMP-0098");
    }
}
```

```
package com.thebackendguy.com;

import com.thebackendguy.com.service.DepartmentService;
import com.thebackendguy.com.service.EmployeeService;

public class Application {

    public static void main(String[] args) {
        DepartmentService departmentService = new DepartmentService();
        EmployeeService employeeService = new EmployeeService(departmentService);
        System.out.println(employeeService.checkDepartment());
    }
}
```

Here we are manually creating and injecting dependencies, we want Spring framework to do this, let's add Spring Framework to the project

Add Spring Framework Dependency

We only need `spring-context` dependency to be added in the project. It is will provide all dependency injection capabilities Spring Framework has to offer. At the time of writing this article, latest stable version is `5.2.12.RELEASE`.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.12.RELEASE</version>
</dependency>
```

Creating and Managing Dependencies (Beans)

Spring managed dependencies are called **Beans**. Since `EmployeeService` *depends* on `DepartmentService` we want Spring to manage that *dependency*. We need to have a configuration class where we can define dependencies and let Spring know about them.

Spring Beans Configuration class:

Create a new `ApplicationConfig` class that will hold dependency (Bean) configurations:

```
package com.thebackendguy.com.config;

import com.thebackendguy.com.service.DepartmentService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration // 1
public class ApplicationConfig {

    @Bean // 2
    public DepartmentService departmentService() {
        return new DepartmentService();
    }
}
```

1. **@Configuration** is a Spring annotation to mark a class as Bean configuration class, it will help Spring framework to find Beans.
2. **@Bean** is a Spring annotation to let Spring know about dependency. This annotation is used on a method that Spring will call to obtain that dependency. Here a new `DepartmentService` instance is returned for Spring to manage. That instance is now called a *Bean*.

Application Context:

Spring Framework use `ApplicationContext` to manage the beans. `ApplicationContext` represents a container, also called Spring Inversion of Control (IoC) container, that contains all of Spring managed instances or beans.

We need to create an Application Context for our application to access beans. Let's update our main `Application` class.

```
package com.thebackendguy.com;

import com.thebackendguy.com.config.ApplicationConfig;
import com.thebackendguy.com.service.DepartmentService;
import com.thebackendguy.com.service.EmployeeService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Application {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(Applicatio
        DepartmentService departmentService = context.getBean(DepartmentService.class)
        System.out.println(departmentService); // 3

        EmployeeService employeeService = new EmployeeService(departmentService);
        System.out.println(employeeService.checkDepartment());
    }
}
```

1. Create a new `ApplicationContext` using our `ApplicationConfig` class (this class contains our `DepartmentService` bean). We need to specify configuration class(s) while creating context so that Spring can find our beans. We can use XML based configuration too instead of Annotation based configurations, but for simplicity we will stick with Annotation based configurations.
2. We are getting `DepartmentService` bean from application context (or IoC container), instead of creating it here.
3. We are simply printing `departmentService` object.

We are passing `departmentService` to `EmployeeService` same as before.

Auto Injecting Dependencies (Autowiring Beans)

So far we have created `DepartmentService` Bean and an `ApplicationContext` to manage that bean. We can access that Bean using `context` and pass it to other service, but it would be much more cleaner if the Bean "gets injected" automatically, rather than we get it from application context and inject manually.

This auto injection of dependencies (or Beans) is called **Autowiring** and Spring provides it out of the box, given that both of the instances (the dependent and the dependency) are Spring Beans. In other words, `DepartmentService` gets injected in `EmployeeService` automatically if both of them are Beans. It won't work otherwise. So, let's make `EmployeeService` a bean too.

Update `ApplicationConfig class` to add `EmployeeService` bean:

```
package com.thebackendguy.com.config;

import com.thebackendguy.com.service.DepartmentService;
import com.thebackendguy.com.service.EmployeeService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

    @Bean
    public DepartmentService departmentService() {
        return new DepartmentService();
    }

    @Bean
    public EmployeeService employeeService() {
        return new EmployeeService(departmentService()); // 1
    }
}
```

1. `EmployeeService` needs an instance of `DepartmentService` at time of creation, so we are calling `departmentService()` to fullfil that dependency, but **this will not create a new DepartmentService instance** rather it will automatically Inject the previously existing `DepartmentService` bean. Both of the methods are actually Spring Bean configuration and default Bean scope is singleton (single shared instance per context) and Spring is smart enough to realize and inject existing beans where ever required. (For brevity of this article I am not going in to detail of bean scope)

Let's update our `Application` class to get `EmployeeService` instance rather than creating it.

```
public class Application {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(Applicatio

        EmployeeService employeeService = context.getBean(EmployeeService.class); // 1
        System.out.println(employeeService.checkDepartment());
    }
}
```

1. Now we don't need to manually create instance of `EmployeeService` and inject `DepartmentService` dependency. It is now getting handled by Spring framework.

These are the minimal steps required to use Spring Framework without Spring boot. Spring Framework Dependency Injection has a lot to offer in addition to these basic Dependency Injection capabilities. You will find more articles on similar topic on thebackendguy.com

Find code on GitHub: https://github.com/thebackendguy-code-examples/java-spring-demo-without-spring-boot