Components, Beans, & Autowiring

1. Introduction to components and beans
2. Autowiring

Demo Spring module: demo-components-beans-autowiring

1. Intro to Components & Beans

- Overview of components
- Defining a simple component
- Specifying a name for a component
- Specifying bean scope
- Lazy bean instantiation
- Accessing a bean
- Component scanning in Spring Boot

Demo package: `demo.beans`

Overview of Components

- In Spring, a component is:
  - A class that Spring will automatically instantiate
- To define a component in Spring, annotate a class with any of the following annotations:
  - `@Component`
  - `@Service`
  - `@Repository`
  - `@Controller/@RestController`

Defining a Simple Component

- Here's an example of how to define a component:

```
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    …
}
```

MyComponent.java

- Spring will automatically create an instance - a "bean"
  - The bean will have a name (by default it's the same as the class name, with first letter lowercased)

Specifying a Name for a Component

- You can specify a name for the component, like this:

```
import org.springframework.stereotype.Component;

@Component("myCoolBean")
public class SomeComponent {
    …
}
```

- When Spring creates a bean, the bean will now be named `myCoolBean`

Specifying Bean Scope

- By default, Spring creates a singleton bean instance
  - i.e. the default scope is `singleton`
- You can specify a different scope:
  - `prototype`, `request`, `session`, `application`
  - For example:

```
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;

@Component
@Scope("prototype")
public class SomeComponent {
    …
}
```

Lazy Bean Instantiation

- Spring always eagerly creates singleton beans
  - This can result in (very) slow start-up
- You can tell Spring to lazily-instantiate a bean
  - Avoids creating a lot of beans that you might never use
  - Speeds start-up time

```
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Lazy;

@Component
@Lazy
public class SomeComponent {
    …
}
```

Accessing a Bean

- In Spring Boot, you can access a bean in your program via the `ApplicationContext` object

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

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        MyComponent bean = ctx.getBean(MyComponent.class);
        System.out.println(bean);
    }
}
```

Application.java

Component Scanning in Spring Boot

- When a Spring Boot application starts, it scans for component classes
  - It looks in the application class package plus subpackages
- You can tell Spring Boot to look elsewhere instead

```
@SpringBootApplication(
  scanBasePackages={"mypackage1", "mypackage2"}
)
public class Application {
    …
}
```

2. Autowiring

- Injecting dependencies into fields
- Injecting dependencies into a constructor
- Fine-tuning autowiring
- Autowiring collections
- Injecting values into beans
- Specifying values in application properties
- Aside: Common application properties

Demo package: `demo.autowiring`

Injecting Dependencies into Fields

- If a bean has dependencies...
  - You can inject via `@Autowired`
- You can use `@Autowired` on a field
  - Spring injects a bean of the specified type into the field

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private BankRepository repository;
    …
}
```

BankServiceImpl.java

Injecting Dependencies into a Constructor

- You can also use `@Autowired` on a constructor
  - Spring will inject beans into all constructor parameters

```
@Service
public class BankServiceImpl implements BankService {

    private BankRepository repository;

    @Autowired
    public BankServiceImpl(BankRepository repository) {
        this.repository = repository;
    }
    …
}
```

BankServiceImpl.java

- If a component only has one constructor, you can omit `@Autowired` (Spring autowires params automatically)

Fine-Tuning Autowiring

- You can specify which bean instance to inject
  - Use `@Qualifier` to specify the bean name you want

```
@Autowired
@Qualifier("primaryRepository")
private BankRepository repository;
```

- You can mark an `@Autowired` member as optional
  - Set `required=false`

```
@Autowired(required=false)
private BankRepository repository;
```

Autowiring Collections (1 of 2)

- You can autowire a `Collection<T>`
  - Spring injects a collection of all the beans of type `T`
- Example
  - Autowire a collection of all beans that implement the `BankRepository` interface

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Collection<BankRepository> repositories;
    …
}
```

Autowiring Collections (2 of 2)

- You can also autowire a `Map<String,T>`
  - Spring injects a map indicating all beans of type `T`
  - Keys are bean names, values are bean instances
- Example
  - Autowire `BankRepository` names/beans

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Map<String, BankRepository> repositoriesMap;
    …
}
```

Injecting Values into Beans

- You can inject values into beans, via `@Value`
  - Use `$` to inject an application property value
  - Use `#` to inject a general Java value via SpEL

```
import org.springframework.beans.factory.annotation.Value;
…

@Component
public class MyBeanWithValues {

    @Value("${name}")           // Inject value of "name" application property.
    private String name;

    @Value("#{ 5 * 7.5 }")      // Inject general Java value via SpEL.
    private double workingWeek;
    …
}
```

MyBeanWithValues.java

Specifying Values in Application Properties

- You can define values in the application properties file

```
name=John Smith
```

application.properties

- Here's how to access the bean in the main code

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        …
        MyBeanWithValues beanWithValues = ctx.getBean(MyBeanWithValues.class);
        System.out.println(beanWithValues);
    }
}
```

Application.java

Aside: Common Application Properties

- Spring Boot defines lots of common application properties by default - you can see the full list here:
  - https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html
- You can override any of these properties in your code
  - In `application.properties` or `application.yml`

Summary

- Introduction to components and beans
- Autowiring