COURSEWARE

Dependency Injection

Contents

Overview

Dependency injection is a design pattern used to reduce coupling between software components.
Dependency injection is often used to facilitate Inversion of Control containers such as Spring Boot.

Benefits

Consider the following code:

```
@Service
public class ExampleService {

    private DbConnection db = new MySQLDBConnection();

    public ExampleService() {
        super();
    }

    //
}
```

This example service depends on an instance of `DBConnection`, which is an interface. With this set-up our example service **only** works with an example of `MySQLDBConnection`.

But what if we want it to work with many *different* db connections?

```
@Service
public class ExampleService {

    private DbConnection db;

    public ExampleService(DbConnection db) {
        super();
        this.db = db;
    }

}
```

Notice how the dependency is no longer instantiated inside the class but is now **injected** during object creation via the constructor.
By managing our dependencies like this our `ExampleService` will work with *any* kind of `DBConnection` and *not* just `MySQLDBConnection`.

Autowiring

In Spring Boot it is possible to define the exact *instance* that our class depends on through a process known as *wiring* but for the most part we let Spring do this for us, thus *autowiring*.

This is how the previous example would look when using *autowiring*.

Constructor Injection

```
@Service
public class ExampleService {

    private DbConnection db;

    @Autowired
    public ExampleService(DbConnection db) {
        super();
        this.db = db;
    }

}
```

This method injects the dependency via the *constructor*, this is the preferred method for reasons that will become clear soon.

Setter Injection

```
@Service
public class ExampleService {

    private DbConnection db;

    @Autowired
    public void setDB(DbConnection db) {
        this.db = db;
    }

}
```

When using setter injection it is important to be careful of two things:

1. The dependency is *not* mandatory, it is entirely possible for the injection to fail but for the class to still be created.
   This can obviously cause issues down the road if your class *requires* this dependency to function.
2. Setter injection allows for *cyclical dependencies*; `A` depends on `B`, which depends on `C` which depends on `A` and so forth. Cyclical dependencies *will* cause your application to fall over and are obviously to be avoided.

Field Injection

```
@Service
public class ExampleService {

    @Autowired
    private DbConnection db;

}
```

Field injection doesn't require a constructor or setter; simply annotate the desired field with `@Autowired`, however, Java still requires a setter/constructor in order to access a private field so at runtime Spring *reflects* in a setter so it can

inject the dependency.
This causes unnecessary overhead when running the application and is why constructor injection is generally seen as preferable.

Tutorial

There is no tutorial for this module.

Exercises

There are no exercises for this module.