Spring Dependency Injection

A Quick Overview

Defining Some Terms...

- Sometimes, a class is dependant on other classes, this is referred to as a **dependency**.
  - Think about JDBC as an example: We have data sources, which are required by JDBC templates, which are required for rowsets.
- Coupling measures how difficult it is to make changes given these interconnected relationships, i.e. if I change something in one class or method, how many more changes will I have to do to other classes and methods?
  - **Tightly coupled code**: code that is very interconnected. Changing one thing necessitates a change elsewhere.
  - **Loosely coupled code**: code is designed so that classes are somewhat independent from one another (preferred)

Achieving Loose Coupling

- There are many techniques to achieve loose coupling, we have already seen how using Interfaces can help:

Tightly Coupled

```
// Assume that SalariedWorkers and HourlyWorkers are
// unrelated concrete classes:

List<SalariedWorkers> salaried = new ArrayList<SalariedWorkers>();
List<HourlyWorkers> hourly = new ArrayList<HourlyWorkers>();

sendOutChecks(salaried, hourly);
```

What if we had to add another category of workers?

Loosely Coupled

```
// Assume that HourlyWorkers and
// SalariedWorkers implement an interface called Worker.
// Assume also that sendOutChecks takes as a parameter a
// List of Workers.

List<Worker> salaried = new ArrayList<SalariedWorkers>();
sentOutChecks(salaried);

List<Worker> hourly = new ArrayList<HourlyWorkers>();
sentOutChecks(hourly);
```

As long as any type of new worker implements the Worker interface, the method does not require modifications!

Dependency Injection

- Dependency Injection is another technique in Spring (and most other Frameworks) to cut down on tight coupling.

- The key annotation we will examine is **@Autowired**.

- When we annotate a data member with @Autowired we are asking Spring the following: *go find me something in the project I can use to fulfill this dependency*.

Dependency Injection

Consider the code used to instantiate a DAO object that you used in module 2 vs module 3:

mod 2 code

```
private DepartmentDAO departmentDAO;

public ProjectsCLI() {
        this.menu = new Menu(System.in, System.out);
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl("jdbc:postgresql://localhost:5432/projects");
        dataSource.setUsername("postgres");
        dataSource.setPassword("elephant");

        departmentDAO = new JDBCDepartmentDAO(dataSource);
    }
```

mod 3 code

```
@Autowired
private ActorDao actorDao;
```

So how did we go from typing so much code to just one line?

Dependency Injection: Java Beans

It all starts with Beans… which are special objects managed by the Spring Application. Here we have a bean definition in the springmvc-servlet.xml file.

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url" value="jdbc:postgresql://localhost:5432/dvdstore" />
    <property name="username" value="postgres" />
    <property name="password" value="" />
</bean>
```

The bean is equivalent to the BasicDataSource object we manually instantiated.

DI: Java Beans

It all starts with Beans… which are special objects managed by the Spring Application. Here we have a bean definition in the springmvc-servlet.xml file.

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url" value="jdbc:postgresql://localhost:5432/dvdstore" />
    <property name="username" value="postgres" />
    <property name="password" value="" />
</bean>
```

The bean is equivalent to the BasicDataSource object we manually instantiated in module 2.

DI: Injecting the Bean into the DAO class:

Let’s examine the DAO class now:

```
@Component
public class JDBCActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;
   @Autowired
   public JDBCActorDao(DataSource datasource) {
       this.jdbcTemplate = new JdbcTemplate(datasource);
   }
```

- Here, we are using the @Autowired annotation to tell Spring: *Go fetch me a datasource I can use. Spring is able to use the bean we defined to fulfill this dependency.*
- Note that nowhere in this class are we doing a manual instantiation, i.e. DataSource ds = new BasicDataSource().

DI: Injecting the Bean into the DAO class:

Let’s examine the DAO class now:

```
@Component
public class JDBCActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;
   @Autowired
   public JDBCActorDao(DataSource datasource) {
       this.jdbcTemplate = new JdbcTemplate(datasource);
   }
```

- Note that there is an extra annotation: **@Component**.
- **@Component** will allows us to inject this JDBCActorDao class into some other class where it is needed as a dependency.

DI: Injecting the DAO class into the Controller

We have now arrived where we started:

```
@Controller
public class ActorSearchController {

    @Autowired
    private ActorDao actorDao;
```

- The **@Autowired** annotation here means: Hey Spring, find me a ActorDao I can use here. In the previous slide, we annotated JDBCActorDao with **@Component**. Since JDBCActorDAO implements ActorDAO, it is a suitable candidate, and thus it is injected here.

Summary

Note that each of the three steps are completely decoupled.

- The data source exists by itself as a bean in a configuration file.
- The DAO class never instantiates (calls a constructor) a data source, it merely asks Spring to find a suitable object to inject
  - ...in this case the bean.
- The Controller class never instantiates a DAO, it merely asks Spring to find a suitable object to inject
  - … in this case the DAO component.