WHAT IS: JAVA
By Matthew Tyson, Java Developer, JavaWorld
OCT 10, 2019 10:42 AM PDT

THE SPRING SERIES

What is Spring? Component-based development for Java

Tutorial introduction to inversion of control and dependency injection, with Spring Web examples

Example #3: Spring with JDBC

Now let's do something more interesting with our request handler: let's return some data from a database. For the purpose of this example, we'll use the H2 database. Thankfully, Spring Boot supports the in-memory H2 DB out of the box.

You can add the H2 DB to your app by including it in your `pom.xml`, as shown in Listing 9. We'll also add a dependency to `spring-boot-starter-jdbc`. This brings in what we need to control JDBC with Spring.

Listing 9. Adding a Maven dependency to the H2 DB

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.194</version>
</dependency>
```

Next, you'll want to configure the database. This is done with a `spring.database.properties` file, which is located in the `/resources` directory. Listing 10 shows how we can use H2 with the in-memory mode activated.

Listing 10. H2 in-memory config

```
driverClassName=org.hsqldb.jdbc.JDBCDriver
url=jdbc:hsqldb:mem:myDb
username=sa
password=sa
```

Service component classes

Now, we can start using the database. It's that easy. However, basic software design tells us never to access the data layer via the view layer. In this case, we don't want to access the JDBC support via the view controller. We need a service component. In Spring Web, we use the`@Service` annotation to create a service class. Like the `@Controller` annotation, using the `@Service` annotation designates a class as a kind of `@Component`. That means Spring will add it to the DI context, and you can autowire it into your controller.

Annotating components

Spring offers a few ways to annotate components. The most basic way to indicate that a *class* is available for auto-wiring is via the `@Component` annotation. The `@Service` annotation does the same thing, but indicates a specific type of class. You could use the `@Bean` annotation to designate a *method* that would serve the purpose of creating a bean to be autowired.

Listing 11 shows a simple `Service` component.

Listing 11. Service component

```
package hello.service;

import org.springframework.stereotype.Service;

@Service("myService")
public class MyService {
  public String getGreeting(){
    return "Hey There";
  }
        public boolean addSong(String name) {
                if (name.length() > 15){
                  return false;
                }
                return true;
        }
        public List<String> getSongs() {
                return new ArrayList();
        }
}
```

Now we can access the service class from the controller. In Listing 12, we'll inject it.

Listing 12. Injecting MyService into the controller

```
@Controller
public class GreetingController {
  @Inject
  private MyService myService;
    @RequestMapping(value = "/hi", method = RequestMethod.GET)
    public String hi(@RequestParam(name="name", required=false, defaultValue="JavaWorld") Str
        return myService.getGreeting() + name;

    }

}
```

Now the `Controller` is making use of the `Service` class. Notice how Spring is allowing us to define a layered architecture using the same DI system. We can do the same in defining a data layer that the service class can use, and leverage Spring's support for a variety of datastores and datastore access approaches at the same time.

We can annotate our data layer class with `@Repository`, as seen in Listing 13, and then inject it into the service class. In the same way `@Service` allowed us to define the service layer, we are now defining the data layer in a decoupled way.

The JdbcTemplate class

The data layer will require more than the service layer, because it will be talking to the database. Spring eases this primarily by providing the `JdbcTemplate` class.

Listing 13. Repository data class

```
import org.springframework.jdbc.core.JdbcTemplate;

@Repository
public class MyDataObject {
  public void addName(String name){
    jdbcTemplate.execute("DROP TABLE names IF EXISTS");
    jdbcTemplate.execute("CREATE TABLE names("id SERIAL, name VARCHAR(255))");
    jdbcTemplate.update("INSERT INTO names (name) VALUES (?)", name);
  }
}
```

Spring will automatically use the in-memory H2 DB we've configured. Notice how `jdbcTemplate` has eliminated all the boilerplate and error-handling code from this class. While this is a simplified example of accessing the database, it gives you an idea of how Spring works both to connect your application layers, and facilitates the use of other required services.

Conclusion

Spring is one of the most advanced and complete application development frameworks for Java, bar none. It makes setting up an application easier, allows you to easily bring in the dependencies you need as the application grows, and is fully capable of ramping up to high-volume, production-grade use.

It's tough to argue against using Spring in a new Java application. The Spring platform is maintained and advanced with vigor, and virtually any task you might need to undertake is doable with Spring. Using this platform will spare you considerable heavy lifting, and will help ensure your application design is robust and flexible. If you can use Spring to ease your development path, then do it.

Learn more about Spring with these in-depth tutorials:

- Mastering Spring framework 5: Spring MVC: Building Java web applications using Spring MVC with Spring Boot.
- Mastering Spring framework 5: Spring WebFlux: Building reactive web applications using Spring WebFlux annotations and functional programming techniques.