Microservices with Spring Boot

Lab docs

Mark Paluch

Version 1.0

Table of Contents

1. Lab Setup

1.1. Prerequisites

You need an installed JDK, at least Java 7 if you write code your self and Java 8 if you want to run the provided code. If you use Git then you can checkout the Code using Git. You can also download the initial code or complete code directly.

An installed IDE is recommended but not necessary. Having `cURL` or any other sophisticated HTTP client installed is beneficial for issuing HTTP requests.

1.2. Register a GitHub account (Optional, recommended)

Go to https://github.com and signup for a free account. If you already have a GitHub account, you can skip this step.

1.3. Clone the Git repository

Clone the Microservices with Spring Boot project using Git.

Console

Open a console and type the following command to clone the project:

Example 1. Cloning the Code Repository

```
$ git clone https://github.com/mp911de/microservices-with-spring-boot.git
```

Eclipse or STS

1. From the project explorer, right click, `Import`, `Git`, `Projects from Git`.
2. Select `Clone URI` and click `Next`.
3. Enter the URI https://github.com/mp911de/microservices-with-spring-boot.git and click `Next`.

4. Select the master branch and click `Next` The projects will be imported.

NOTE: Note that depending on the version of Eclipse / STS, your version of Java, and other factors, you may encounter errors. These will need to be fixed on a case-by-case basis before proceeding.

IntelliJ IDEA

1. From the menu bar `VCS`, select `Checkout from Version Control`, `Github`.
2. Enter the URI https://github.com/mp911de/microservices-with-spring-boot.git and click `Clone`.
3. The repository will be cloned.

1.4. IDE Import

1.5. Eclipse or STS

1. From the project explorer, right click, `Import`, `Maven`, `Existing Maven Projects`.
2. Navigate to the folder where you cloned the repository and `Finish`.
3. The projects will be imported.

Eclipse users wanting to use Lombok need to install the Eclipse/Lombok plugin. Run `lombok.jar` as a java app (i.e. double-click it, usually) to install. Also, add `lombok.jar` to your project. Supported variants: SpringSource Tool Suite, JBoss Developer Studio

NOTE: Note that depending on the version of Eclipse / STS, your version of Java, and other factors, you may encounter errors. These will need to be fixed on a case-by-case basis before proceeding.

1.6. IntelliJ IDEA

1. From File menu select `Import Project` and navigate to wherever the course labs were installed during the previous step.
2. Click `Import project from external model` and import as Maven projects.
3. After clicking `Next`, select the `Search for projects recursively` checkbox.
4. Click `Next` again and several projects should be listed.
5. Click `Next` and select a suitable SDK.
6. Click `Next` one last time. You can leave the project name and location. Click Finish.

IntelliJ IDEA users wanting to use Lombok need to install the Eclipse/Lombok plugin. A plugin developed by Michael Plushnikov adds support for most features.

Congratulations! You have setup everything you will need for the rest of the course. The hard part is over!

2. Runnable Application

About this Lab

Your will learn about the properties of an initial Spring Boot application.

Prerequisites

To complete these steps, you will need:

1. The code checked out.
2. Optiona: The code imported into your IDE.

Steps

1. The Microservices with Spring Boot is a runnable application.
   - IDE: Navigate to the main class `CarSaleApplication`. Run this main class. The application will boot up and start a web server on port 8080.
   - Maven users: Open a console and navigate to the code directory. Run `./mvnw spring-boot:run`
   - Gradle users: Open a console and navigate to the code directory. Run `./gradlew bootRun`
2. Navigate in your browser to the following endpoints:
   - http://localhost:8080/actuator - Actuator Endpoint Overview
   - http://localhost:8080/env - Environment properties
   - http://localhost:8080/trace - Endpoint request tracing

When you are done, stop the application.

Congratulations, you have finished this exercise.

3. Create the domain model and repositories

About this Lab

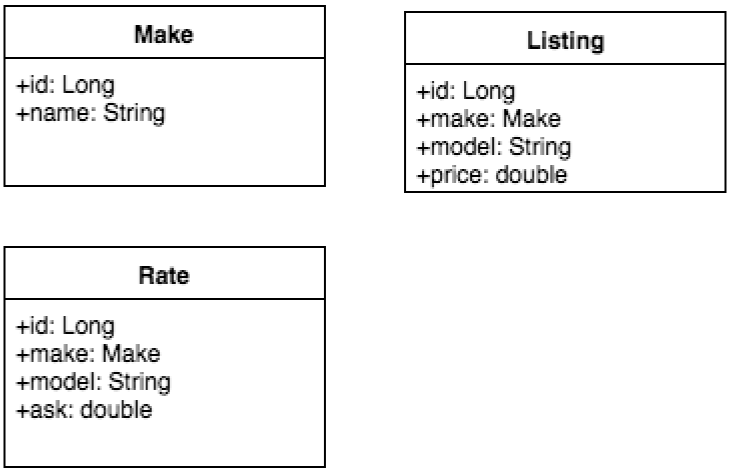
You will learn how to create a domain model using Spring Data. You don't require any external database services because we use the H2 in-memory database.

Prerequisites

To complete these steps, you will need:

1. The code checked out.
2. Optional: The code imported into your IDE.

The domain model



Our model consists of three types:

1. `Make` – Represents a make.
2. `Listing` – Represents a listing with reference to the `Make`, also holding the model name and a price.
3. `Rate` – Represents the current rates for listings with reference to the `Make`. It's a read model based on `Listing`s for a particular `Make` and model.

Steps

1. Create a new package `com.example.data`. This package will contain the domain model and the repository declarations.
2. Create the domain model using JPA entity mapping.
3. The last step is to create repository declarations in the `com.example.data` package using Spring Data for each entity type (`MakeEntity`, `ListingEntity`, `RateEntity`). The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers. Repositories are declared as interfaces with the `Repository`, `CrudRepository` or store-specific `JpaRepository` super-interface.
4. Add a query method to retrieve a single `MakeEntity` by name in the `MakeEntity` repository.
5. Add a query method to retrieve a list of `ListingEntity` by Make name and model in the `ListingEntity` repository.
6. Add a query method to retrieve a single `RateEntity` by Make name and model in the `RateEntity` repository.

Your code could look like:

Example 2. MakeEntity.java

```java
@Data
@Entity
@Table(name = "make")
public class MakeEntity {

    @Id @GeneratedValue Long id;
    String name;
}
```

Example 3. ListingEntity.java

```java
@Data
@Entity
@Table(name = "listing")
public class ListingEntity {

    @Id @GeneratedValue Long id;
    @ManyToOne MakeEntity make;
    String model;
    double price;
}
```

Example 4. RateEntity.java

```java
@Data
@Entity
@Table(name = "rate")
public class RateEntity {

    @Id @GeneratedValue Long id;
    @ManyToOne MakeEntity make;
    String model;
    double ask;
}
```

NOTE This domain model is a uses basic JPA mapping. It also uses Lombok annotations to get rid of boilerplate code. Getter/Setter/Constructors and more are generated during runtime by the annotation processor. You're free to use the style which you're familiar with.

Example 5. MakeRepository.java

```
public interface MakeRepository extends CrudRepository<MakeEntity, Long> {

    Optional<MakeEntity> findByName(String name);
}
```

Example 6. ListingRepository.java

```
public interface ListingRepository extends JpaRepository<ListingEntity, Long> {

    List<ListingEntity> findListingByMakeNameAndModel(String makeName, String
model);
}
```

Example 7. RateRepository.java

```
public interface RateRepository extends CrudRepository<RateEntity, Long> {

    Optional<RateEntity> findRateByMakeAndModel(MakeEntity make, String model);
}
```

Congratulations, you have finished this exercise.

We will continue to the next step with creating tests for the repositories to verify they are correct.

4. Add tests for the `MakeRepository`

About this Lab

You will learn how to set up tests using JUnit and Spring Boot test to run tests in a JPA context.

Prerequisites

To complete these steps, you will need:

1. Have `MakeRepository` and `MakeEntity` set up.

Spring Boot Test support

Spring Boot 1.4 comes with several annotations to support testing.

You will create in a minute a test class using `@RunWith(SpringRunner.class)` and `@DataJpaTest` annotations. `@DataJpaTest` will set up the context and configuration for a JPA focused test hence not required components won’t be enabled by default.

Steps

1. Create a new test class to test the `MakeEntity` repository.
2. Annotate your test class with `@RunWith(SpringRunner.class)` and `@DataJpaTest`.
3. Declare an injection of the `MakeEntity` repository.
4. Write a test to verify that an absent record returns no `MakeEntity`.
5. Write a test to verify that a present record returns the `MakeEntity` you queried. You will need to insert a record before you can query it.
6. Run your test.

Your code could look like:

Example 8. MakeRepositoryTests.java

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class MakeRepositoryTests {

    @Autowired
    MakeRepository makeRepository;

    @Test
    public void shouldReturnEmptyWhenRecordIsAbsent() throws Exception {

        Optional<MakeEntity> absent = makeRepository.findByName("Toyota");
        assertThat(absent.isPresent()).isFalse();
    }

    @Test
    public void shouldReturnRecordIfPresent() throws Exception {

        MakeEntity make = new MakeEntity();
        make.setName("Audi");

        makeRepository.save(make);

        Optional<MakeEntity> present = makeRepository.findByName("Audi");
        assertThat(present.isPresent()).isTrue();
        assertThat(present.get()).isEqualTo(make);
    }
}
```

Congratulations, you have finished this exercise.

5. Initializing the database

About this Lab

You will learn how to initialize the database using Spring Boot.

Prerequisites

To complete these steps, you will need:

1. Have `MakeEntity` set up.

Spring Boot Database initialization

Spring Boot creates by default the database schema, if not specified otherwise. You can integrate with Flyway or Liquibase to control database changes in a more sophisticated fashion. Spring Boot also can initialize your database from `schema.sql` and `data.sql` which is perfectly fine for tests or showcases. Spring Boot uses by default Hibernate and `spring.jpa.hibernate.ddl-auto` default to `create-drop` when using an embedded database, `none` otherwise.

Steps

1. Create a new file `data.sql` in `src/main/resources`
2. Add some test data to populate the `make` table. You will need to set only the `name` column in your `INSERT`s.

Your code could look like:

Example 9. data.sql

```sql
insert into make (name) values('Audi');
insert into make (name) values('Honda');
insert into make (name) values('Toyota');
```

Congratulations, you have finished this exercise.

6. Creating a use-case

About this Lab

You will learn how to add business rules to your application.

Prerequisites

To complete these steps, you will need:

1. Have `MakeRepository` and `MakeEntity` set up.

2. Have `ListingRepository` and `ListingEntity` set up.

Creating business rules

There are plenty of styles, paradigms and patterns how to organize code inside an application This part is not specific to Spring Boot. By adding business rules in one place we will be able to use that code from a facade to perform an action.

Let’s create a piece of code that takes the make of a car, the model name and a price and the code creates a listing. The listing will be stored in a `ListingEntity` and the make is referenced from `MakeEntity`. We will use constructor injection to inject dependencies. Spring 4.3 will autowire components with a single constructor without `@Autowired`.

Steps

1. Create a new class in a package of your choice.
2. Annotate it with `@Service`.
3. Use constructor-injection to inject the `MakeEntity` and `ListingRepositories`.
4. Create some code to lookup the make by its name.
5. Create some code to create a new `ListingEntity` with the make, model name and price set.
6. Save the `ListingEntity` and return it as return value.

Your code could look like:

Example 10. CreateListing.java

```
@Service
@AllArgsConstructor
public class CreateListing {

    ListingRepository listingRepository;
    MakeRepository makeRepository;

    public ListingEntity createListing(String makeName, String model, double
price) {

        MakeEntity make = makeRepository.findByName(makeName)
                .orElseThrow(() -> new IllegalArgumentException("Cannot find make
" + makeName));

        ListingEntity listing = new ListingEntity();
        listing.setMake(make);
        listing.setModel(model);
        listing.setPrice(price);

        listingRepository.saveAndFlush(listing);

        return listing;
    }
}
```

Congratulations, you have finished this exercise.

7. Create an HTTP facade to expose the functionality

About this Lab

You will learn how to create an HTTP endpoint that accepts data and passes it to your business code.

Prerequisites

To complete these steps, you will need:

1. Have `MakeRepository` and `MakeEntity` set up.
2. Have `ListingRepository` and `ListingEntity` set up.
3. The business rules class set up.

Creating HTTP endpoints

`spring-web` provides the core functionality to implement `@Controller`s. A plain `@Controller` uses Model-View-Controller to process return values, usually within the context of server-generated HTML pages. We want to fully control our responses therefore we use `@RestController`.

Let’s create the final class to accept data using JSON and invoke the business rules code.

Steps

1. Create a new package: `com.example.web`
2. Create the representation class that encapsulates the `make` (String), `model` (String), and `price` (double). This class is used to deserialize the JSON request.
3. Create a `ListingController` class.
4. Annotate it with `@RestController`.
5. Use constructor-injection to inject the business rules class.
6. Create a controller method annotated with `@PostMapping(value = "/listings", consumes = MediaType.APPLICATION_JSON_VALUE)`. The controller method accepts the previously created representation class as only argument.
7. Create some code to decompose the representation payload and to invoke the business method.
8. **BONUS**: Validate the arguments to be non-empty
9. **BONUS**: Create an `ExceptionHandler` method and set different HTTP status codes on success/failure

Your code could look like:

Example 11. ListingController.java

```
@RestController
@RequiredArgsConstructor
public class ListingController {

    final CreateListing createListing;

    @PostMapping(value = "/listings", consumes = MediaType.APPLICATION_JSON_VALUE)
    @ResponseStatus(HttpStatus.CREATED)
    void createListing(@RequestBody ListingRepresentation listing) {

        Assert.notNull(listing, "Listing body must not be empty!");
        Assert.hasText(listing.getMake(), "Make must not be empty!");
        Assert.hasText(listing.getModel(), "Model must not be empty!");
        Assert.notNull(listing.getPrice(), "Price must not be empty!");

        createListing.createListing(listing.getMake(), listing.getModel(),
listing.getPrice());

    }

    @ExceptionHandler
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    private String handle(IllegalArgumentException e) {
        return e.getMessage();
    }
}
```

Congratulations, you have finished this exercise.

8. Launch and test the application

About this Lab

You will launch the application and execute some HTTP requests to verify your application is working.

Prerequisites

To complete these steps, you will need:

1. All steps from above.
2. A HTTP client such as `curl` or HTTP client plugins inside your browser.

Steps

1. Launch the `CarSaleApplication` main application class

2. Wait until the application is started up
3. Access http://localhost:8080. You will see a white page with some JSON on it.
4. Issue an HTTP request to the `ListingController` carrying some payload.
5. **BONUS**: Explore the http://localhost:8080/listingEntities endpoint to find the created listing.

Example 12. Example HTTP Request

```
POST /listings
Content-Type: application/json
Host: localhost:8080

{"make": "Audi", "model":"XYZ", "price": 105.1}
```

Example 13. Example cURL Request

```
curl -X POST -i --data-ascii '{"make": "Audi", "model":"XYZ", "price": 105.1}'
-H"Content-type: application/json" http://localhost:8080/listings
```

Example 14. Example HTTP Response

```
HTTP/1.1 202
```

Congratulations, you have finished this exercise.

9. Add caching to your application

About this Lab

You will learn how to enable caching with Spring Boot. You don't require external Caching services for this step because we use the simple in-memory caching provider.

Prerequisites

To complete these steps, you will need:

1. The code checked out.
2. Have `ListingController` set up.

Creating an expensive service

To get started with caching we require a service that either returns different responses on each invocation or takes a while to compute its result. We need to expose that newly created service so we can invoke it. Creating a new controller or reusing existing controllers will do the job. We also need to enable Spring Caching.

Steps

1. Include the Spring Boot Cache starter `spring-boot-starter-cache` to your build file.
2. Create a new class and give and name it `ExpensiveService`. Annotate that class with `@Service`.
3. Add a method to that class that takes some time and returns a different response on each invocation. Using `Thread.sleep(…)` and `Math.random()` are good candidates.
4. Annotate that method with `@Cacheable(…)` and specify a cache name with the `value` property of the annotation. You can choose any arbitrary name.
5. Create a new controller method. Use constructor-injection to inject `ExpensiveService` as dependency
6. Create a controller method annotated with `@GetMapping(value = "/listings")`. The controller returns the value produced in the expensive service method.
7. Spring Boot Caching requires an opt-in to enable caching. Annotate `CarSaleApplication` with `@EnableCaching`.
8. Start your application and access http://localhost:8080/listings multiple times.
9. **BONUS**: Fix `ListingControllerTests`. `ListingControllerTests` is broken if you've reused `ListingController`. This is, because the test does not initialize the `ExpensiveService` during tests. Add a `@MockBean` to fix that test.

Your code could look like:

Example 15. pom.xml

```xml
<project …>

    …

    <dependencies>
        …

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-cache</artifactId>
        </dependency>

        …
    </dependencies>


</project>
```

Some lines are left out for brevity.

Example 16. build.gradle

```
dependencies {

    ...

    compile('org.springframework.boot:spring-boot-starter-cache')
    ...

}
```

Some lines are left out for brevity.

Example 17. ExpensiveService.java

```
@Service
public class ExpensiveService {

    @Cacheable("my-cache")
    @SneakyThrows
    public String calculate() {

        Thread.sleep(10000);

        return "" + Math.random();
    }
}
```

Example 18. ListingController.java

```java
@RestController
@RequiredArgsConstructor
public class ListingController {

    final ExpensiveService expensiveService;

    ...

    @GetMapping(value = "/listings")
    String getListings() {
        return expensiveService.calculate();
    }

    ...

}
```

Some lines are left out for brevity.

Example 19. CarSaleApplication.java

```java
@SpringBootApplication
@EnableCaching
public class CarSaleApplication {

    public static void main(String[] args) {
        SpringApplication.run(CarSaleApplication.class, args);
    }
}
```

Congratulations, you have finished this exercise.