Chapter 4. Adding Database Access to Your Spring Boot App

As discussed in the previous chapter, applications often expose stateless APIs for many very good reasons. Behind the scenes, however, very few useful applications are entirely ephemeral; state of some kind is usually stored for *something*. For example, every request to an online store's shopping cart may well include its state, but once the order is placed, that order's data is kept. There are many ways to do this, and many ways to share or route this data, but invariably there are one or more databases involved within nearly all systems of sufficient size.

In this chapter, I'll demonstrate how to add database access to the Spring Boot application created in the previous chapter. This chapter is meant to be a short primer on Spring Boot's data capabilities, and subsequent chapters will dive much deeper. But in many cases, the basics covered here still apply well and provide a fully sufficient solution. Let's dig in.

CODE CHECKOUT CHECKUP

Please check out branch *chapter4begin* from the code repository to begin.

Priming Autoconfig for Database Access

As demonstrated earlier, Spring Boot aims to simplify to the maximum extent possible the so-called 80–90% use case: the patterns of code and process that developers do over and over and over again. Once patterns are identified, Boot springs into action to initialize the required beans automatically, with sensible default configurations. Customizing a capability is as simple as providing one or more property values or creating a tailored version of one or more beans; once autoconfig detects the changes, it backs off and follows the developer's guidance. Database access is a perfect example.

What Do We Hope to Gain?

In our earlier example application, I used an `ArrayList` to store and maintain our list of coffees. This approach is straightforward enough for a single application, but it does have its drawbacks.

First, it isn't resilient at all. If your application or the platform running it fails, all changes made to the list while the app was running—whether for seconds or months—disappear.

Second, it doesn't scale. Starting another instance of the application results in that second (or subsequent) app instance having its own distinct list of coffees it maintains. Data isn't shared between the multiple instances, so changes to coffees made by one instance—new coffees, deletions, update—aren't visible to anyone accessing a different app instance.

Clearly this is no way to run a railroad.

I cover a few different ways to fully solve these very real problems in upcoming chapters. But for now, let's lay some groundwork that will serve as useful steps on the path there.

Adding a Database Dependency

In order to access a database from your Spring Boot application, you need a few things:

- A running database, whether initiated by/embedded within your application or simply accessible to your application
- Database drivers enabling programmatic access, usually provided by the database vendor
- A Spring Data module for accessing the target database

Certain Spring Data modules include the appropriate database drivers as a single selectable dependency from within the Spring Initializr. In other cases, such as when Spring uses the Java Persistence API (JPA) to access JPA-compliant datastores, it's necessary to choose the Spring Data JPA dependency *and* a dependency for the target database's specific driver, e.g., PostgreSQL.

To take the first step forward from memory constructs to persistent database, I'll begin by adding dependencies, and thus capabilities, to our project's build file.

H2 is a fast database written completely in Java that has some interesting and useful features. For one thing, it's JPA-compliant, so we can connect our application to it in the same manner we would to any other JPA database like Microsoft SQL, MySQL, Oracle, or PostgreSQL. It also has in-memory and disk-based modes. This allows us some useful options after we convert from our in-memory `ArrayList` to an in-memory database: we can either change H2 to disk-based persistence or—since we're now using a JPA database—change to a different JPA database. Either option becomes much simpler at that point.

To enable our application to interact with the H2 database, I'll add the following two dependencies to the `<dependencies>` section of our project's *pom.xml*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

NOTE

The H2 database driver dependency's scope of `runtime` indicates that it will be present in the runtime and test classpath but not in the compile classpath. This is a good practice to adopt for libraries that are not required for compilation.

Once you save your updated *pom.xml* and (if necessary) reimport/refresh your Maven dependencies, you have access to the functionality included within the added dependencies. Next, it's time to write a bit of code to use it.

Adding Code

Since we already have code in place to manage coffees in some fashion, we'll need to refactor a bit while we add our new database capabilities. I find the best place to begin is with the domain class(es), in this case, `Coffee`.

The @Entity

As mentioned earlier, H2 is a JPA-compliant database, so I'll add JPA annotations to connect the dots. To the `Coffee` class itself I add an `@Entity` annotation from `javax.persistence` that indicates `Coffee` is a persistable entity, and to the existing `id` member variable, I add the `@Id` annotation (also from `javax.persistence`) to mark it as the database table's ID field.

NOTE

If the class name—`Coffee` in this case—doesn't match the desired database table name, the `@Entity` annotation accepts a `name` parameter for specifying the data table name to match the annotated entity.

If your IDE is helpful enough, it may provide you feedback that something is still missing in the `Coffee` class. For example, IntelliJ underlines the class name in red and provides the helpful pop-up upon mouseover shown in Figure 4-1.

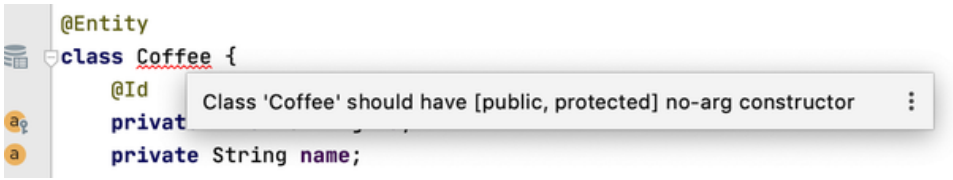

Figure 4-1. Missing constructor in the JPA `Coffee` class

Java Persistence API requires a no-argument constructor for use when creating objects from database table rows, so I'll add that next. This results in our next IDE warning, as displayed in Figure 4-2: in order to have a no-arg constructor, we must make all member variables mutable, i.e., nonfinal.

Figure 4-2. With a no-arg constructor, `id` cannot be final

Removing the `final` keyword from the declaration for the `id` member variable solves that. Making `id` mutable also requires our `Coffee` class to have a mutator method for `id` for JPA to be able to assign a value to that member, so I add the `setId()` method as well, as shown in Figure 4-3.

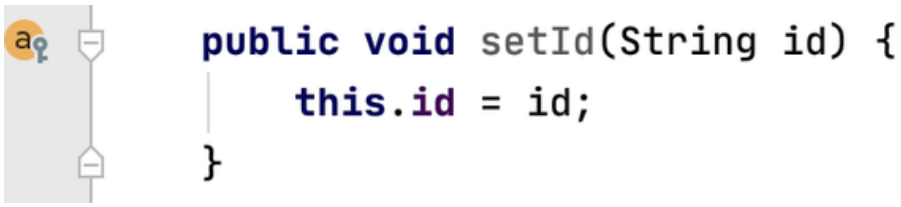

Figure 4-3. The new `setId()` method

The Repository

With `Coffee` now defined as a valid JPA entity able to be stored and retrieved, it's time to make the connection to the database

For such a simple concept, configuring and establishing a database connection in the Java ecosystem has long been a rather cumbersome affair. As mentioned in Chapter 1, using an application server to host a Java application required developers to perform several tedious steps just to get things ready. Once you started interacting with the database, or if you were accessing a datastore directly from a Java utility or client application, you would be expected to perform additional steps involving `PersistenceUnit`, `EntityManagerFactory`, and `EntityManager` APIs (and possibly `DataSource` objects), open and close the database,

and more. It's a lot of repetitive ceremony for something developers do so often.

Spring Data introduces the concept of repositories. A `Repository` is an interface defined in Spring Data as a useful abstraction above various databases. There are other mechanisms for accessing databases from Spring Data that will be explained in subsequent chapters, but the various flavors of `Repository` are arguably the most useful in the most cases.

`Repository` itself is a mere placeholder for the following types:

- The object stored in the database
- The object's unique ID/primary key field

There is a lot more to repositories, of course, and I cover a great deal of that in Chapter 6. For now, let's focus on two that are directly relevant to our current example: `CrudRepository` and `JpaRepository`.

Recall my earlier mention of the preferred practice of writing code to use the highest-level interface suited to purpose? While `JpaRepository` extends a handful of interfaces and thus incorporates broader functionality, `CrudRepository` covers all of the key CRUD capabilities and is sufficient for our (so far) simple application.

The first thing to do to enable repository support for our application is to define an interface specific to our application by extending a Spring Data `Repository` interface: .interfaceCoffeeRepo

```
  interface CoffeeRepository extends CrudRepository<Coffee, String> {}
```

NOTE

The two types defined are the object type to store and the type of its unique ID.

This represents the simplest expression of repository creation within a Spring Boot app. It's possible, and very useful at times, to define queries for a repository; I'll dive into that in a future chapter as well. But here is the "magical" part: Spring Boot's autoconfiguration takes into account the database driver on the classpath (in this case, H2), the repository interface defined in our application, and the JPA entity `Coffee` class definition and creates a database proxy bean *on our behalf*. No need to write lines of nearly identical boilerplate for every application when the patterns are this clear and consistent, which frees the developer to work on new, requested functionality.

The utility, aka “Springing” into action

Now to put that repository to work. I’ll approach this step by step as in previous chapters, introducing functionality first and polishing afterward.

First, I’ll autowire/inject the repository bean into `RestApiDemoController` so the controller can access it when receiving requests via the external API, as shown in Figure 4-4.

First I declare the member variable with:

```
private final CoffeeRepository coffeeRepository;
```

Next, I add it as a parameter to the constructor with:

```
public RestApiDemoController(CoffeeRepository coffeeRepository){}
```

NOTE

Prior to Spring Framework 4.3, it was necessary in all cases to add the `@Autowired` annotation above the method to indicate when a parameter represented a Spring bean to be autowired/injected. From 4.3 onward, a class with a single constructor doesn’t require the annotation for autowired parameters, a useful time-saver.

Figure 4-4. Autowire repository into `RestApiDemoController`

With the repository in place, I delete the `List<Coffee>` member variable and change the initial population of that list in the constructor to instead save the same coffees to the repository, as in Figure 4-4.

Per Figure 4-5, removing the `coffees` variable immediately flags all references to it as unresolvable symbols, so the next task is replacing those with appropriate repository interactions.

Figure 4-5. Replacing the removed `coffees` member variable

As a simple retrieval of all coffees with no parameters, the `getCoffees()` method is a great place to begin. Using the `findAll()`

method built into `CrudRepository`, it isn't even necessary to change the return type of `getCoffees()` as it also returns an `Iterable` type; simply calling `coffeeRepository.findAll()` and returning its result does the job, as shown here:

```
@GetMapping
Iterable<Coffee> getCoffees() {
    return coffeeRepository.findAll();
}
```

Refactoring the `getCoffeeById()` method presents some insights into how much simpler your code can be, thanks to the functionality that repositories bring to the mix. We no longer have to manually search the list of coffees for a matching `id`; `CrudRepository`'s `findById()` method handles it for us, as demonstrated in the following code snippet. And since `findById()` returns an `Optional` type, no changes whatsoever are required for our method signature:

```
@GetMapping("/{id}")
Optional<Coffee> getCoffeeById(@PathVariable String id) {
    return coffeeRepository.findById(id);
}
```

Converting the `postCoffee()` method to use the repository is also a fairly straightforward endeavor, as shown here:

```
@PostMapping
Coffee postCoffee(@RequestBody Coffee coffee) {
    return coffeeRepository.save(coffee);
}
```

With the `putCoffee()` method, we again see some of the substantial time- and code-saving functionality of the `CrudRepository` on display. I use the built-in `existsById()` repository method to determine if this is a new or existing `Coffee` and return the appropriate HTTP status code along with the saved `Coffee`, as shown in this listing:

```
@PutMapping("/{id}")
ResponseEntity<Coffee> putCoffee(@PathVariable String id,
                                 @RequestBody Coffee coffee) {

    return (!coffeeRepository.existsById(id))
            ? new ResponseEntity<>(coffeeRepository.save(coffee),
                    HttpStatus.CREATED)
            : new ResponseEntity<>(coffeeRepository.save(coffee), HttpStatus.C
}
```

Finally, I update the `deleteCoffee()` method to use `CrudRepository`'s built-in `deleteById()` method, as shown here:

```
@DeleteMapping("/{id}")
void deleteCoffee(@PathVariable String id) {
    coffeeRepository.deleteById(id);
}
```

Leveraging a repository bean created using the fluent API of `CrudRepository` streamlines the code for the `RestApiDemoController` and makes it much clearer, in terms of both readability and understandability, as evidenced by the complete code listing:

```
@RestController
@RequestMapping("/coffees")
class RestApiDemoController {
    private final CoffeeRepository coffeeRepository;

    public RestApiDemoController(CoffeeRepository coffeeRepository) {
        this.coffeeRepository = coffeeRepository;

        this.coffeeRepository.saveAll(List.of(
                new Coffee("Café Cereza"),
                new Coffee("Café Ganador"),
                new Coffee("Café Lareño"),
                new Coffee("Café Três Pontas")
        ));
    }

    @GetMapping
    Iterable<Coffee> getCoffees() {
        return coffeeRepository.findAll();
    }

    @GetMapping("/{id}")
    Optional<Coffee> getCoffeeById(@PathVariable String id) {
        return coffeeRepository.findById(id);
    }

    @PostMapping
    Coffee postCoffee(@RequestBody Coffee coffee) {
        return coffeeRepository.save(coffee);
    }

    @PutMapping("/{id}")
    ResponseEntity<Coffee> putCoffee(@PathVariable String id,
                                     @RequestBody Coffee coffee) {

        return (!coffeeRepository.existsById(id))
                ? new ResponseEntity<>(coffeeRepository.save(coffee),
                HttpStatus.CREATED)
```

```
                : new ResponseEntity<>(coffeeRepository.save(coffee), HttpStat
    }

    @DeleteMapping("/{id}")
    void deleteCoffee(@PathVariable String id) {
        coffeeRepository.deleteById(id);
    }
}
```

Now all that remains is to verify that our application works as expected and external functionality remains the same.

NOTE

An alternative approach to testing functionality—and a recommended practice—is to create unit tests first, a la Test Driven Development (TDD). I strongly recommend this approach in real-world software development environments, but I've found that when the goal is to demonstrate and explain discrete software development concepts, less is better; showing as little as possible to clearly communicate key concepts increases signal and decreases noise, even if the noise is useful later. As such, I cover testing in a dedicated chapter later in this book.

Saving and Retrieving Data

Once more unto the breach, dear friends, once more: accessing the API from the command line using HTTPie. Querying the *coffees* endpoint results in the same four coffees being returned from our H2 database as before, as shown in Figure 4-6.

Copying the `id` field for one of the coffees just listed and pasting it into a coffee-specific `GET` request produces the output shown in Figure 4-7.

```
mheckler-a01 :: ~ » http :8080/coffees
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:08:48 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

[
    {
        "id": "ff3d96e0-236e-4157-8b45-9e9699276d6d",
        "name": "Café Cereza"
    },
    {
        "id": "d7a0f2a1-38f7-46ef-a884-8beb43e655cf",
        "name": "Café Ganador"
    },
    {
        "id": "d5458c8c-f480-47dc-9926-42fcb1f4051d",
        "name": "Café Lareño"
    },
    {
        "id": "1726fcdf-94f9-4f7b-9e60-e6e1b453f56f",
        "name": "Café Três Pontas"
    }
]
```

Figure 4-6. `GET`-ting all coffees

```
mheckler-a01 :: ~ » http :8080/coffees/ff3d96e0-236e-4157-8b45-9e9699276d6d
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:20:18 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "id": "ff3d96e0-236e-4157-8b45-9e9699276d6d",
    "name": "Café Cereza"
}
```

Figure 4-7. `GET`-ting a coffee

In Figure 4-8, I `POST` a new coffee to the application and its database.

```
mheckler-a01 :: ~/dev » http :8080/coffees < coffee.json
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:22:17 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "id": "99999",
    "name": "Kaldi's Coffee"
}
```

Figure 4-8. `POST`-ing a new coffee to the list

As discussed in the previous chapter, a `PUT` command should allow for updating an existing resource or adding a new one if the requested resource doesn't already exist. In Figure 4-9, I specify the `id` of the coffee just added and pass to the command a JSON object with a change to that coffee's name. After the update, the coffee with the `id` of "99999" now has a `name` of "Caribou Coffee" instead of "Kaldi's Coffee", and the return code is 200 (OK), as expected.

```
mheckler-a01 :: ~/dev » http PUT :8080/coffees/99999 < coffee2.json
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:24:04 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "id": "99999",
    "name": "Caribou Coffee"
}
```

Figure 4-9. `PUT`-ting an update to an existing coffee

Next I initiate a similar `PUT` request but specify an `id` in the URI that doesn't exist. The application adds a new coffee to the database in compliance with IETF-specified behavior and correctly returns an HTTP status of 201 (Created), as shown in Figure 4-10.

Figure 4-10. `PUT`-ting a new coffee

Finally, I test deletion of a specified coffee by issuing a `DELETE` request, which returns only an HTTP status code of 200 (OK), indicating the resource was successfully deleted and nothing else, since the resource no longer exists, per Figure 4-11. To check our end state, we once again query the full list of coffees (Figure 4-12).

Figure 4-11. `DELETE`-ing a coffee

Figure 4-12. `GET`-ting all coffees now in the list

As before, we now have one additional coffee that wasn't initially in our repository: Mötor Oil Coffee.

A Bit of Polishing

As always, there are many areas that could benefit from additional attention, but I'll confine the focus to two: extracting the initial population of sample data to a separate component and a bit of condition reordering for clarity.

Last chapter I populated the list of coffees with some initial values in the `RestApiDemoController` class, so I maintained that same structure—until now—in this chapter after converting to a database with repository access. A better practice is to extract that functionality to a separate component that can be enabled or disabled quickly and easily.

There are many ways to execute code automatically upon application startup, including using a `CommandLineRunner` or `ApplicationRunner` and specifying a lambda to accomplish the desired goal: in this case, creating and saving sample data. But I prefer using an `@Component` class and an `@PostConstruct` method to accomplish the same thing for the following reasons:

- When `CommandLineRunner` and `ApplicationRunner` bean-producing methods autowire a repository bean, unit tests that mock the repository bean within the test (as is typically the case) break.
- If you mock the repository bean within the test or wish to run the application without creating sample data, it's quick and easy to disable the actual data-populating bean simply by commenting out its `@Component` annotation.

I recommend creating a `DataLoader` class similar to the one shown in the following code block. Extracting the logic to create sample data to the `DataLoader` class's `loadData()` method and annotating it with `@PostContruct` restores `RestApiDemoController` to its intended single purpose of providing an external API and makes the `DataLoader` responsible for *its* intended (and obvious) purpose:

```
@Component
class DataLoader {
    private final CoffeeRepository coffeeRepository;

    public DataLoader(CoffeeRepository coffeeRepository) {
        this.coffeeRepository = coffeeRepository;
    }

    @PostConstruct
    private void loadData() {
        coffeeRepository.saveAll(List.of(
                new Coffee("Café Cereza"),
                new Coffee("Café Ganador"),
                new Coffee("Café Lareño"),
                new Coffee("Café Três Pontas")
        ));
    }
}
```

The other dab of polishing is an admittedly small adjustment to the boolean condition of the ternary operator within the `putCoffee()` method.

After refactoring the method to use a repository, no compelling justification remains for evaluating the negative condition first. Removing the not (!) operator from the condition slightly improves clarity; swapping the true and false values of the ternary operator are of course required to maintain the original outcomes, as reflected in the following code:

```
@PutMapping("/{id}")
ResponseEntity<Coffee> putCoffee(@PathVariable String id,
                                 @RequestBody Coffee coffee) {

    return (coffeeRepository.existsById(id))
            ? new ResponseEntity<>(coffeeRepository.save(coffee),
                HttpStatus.OK)
            : new ResponseEntity<>(coffeeRepository.save(coffee),
                HttpStatus.CREATED);
}
```

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter4end* from the code repository.

Summary

This chapter demonstrated how to add database access to the Spring Boot application created in the last chapter. While it was meant to be a concise introduction to Spring Boot's data capabilities, I provided an overview of the following:

- Java database access
- The Java Persistence API (JPA)
- The H2 database
- Spring Data JPA
- Spring Data repositories
- Mechanisms to create sample data via repositories

Subsequent chapters will dive much deeper into Spring Boot database access, but the basics covered in this chapter supply a solid foundation upon which to build and, in many cases, are sufficient by themselves.

In the next chapter, I'll discuss and demonstrate useful tools Spring Boot provides to gain insights into your applications when things aren't functioning as expected or when you need to verify that they are.