Chapter 3. Creating Your First Spring Boot REST API

In this chapter, I explain and demonstrate how to develop a basic working application using Spring Boot. Since most applications involve exposing backend cloud resources to users, usually via a frontend UI, an Application Programming Interface (API) is an excellent starting point for both understanding and practicality. Let's get started.

The Hows and Whys of APIs

The age of the monolithic application that does everything is over.

This isn't to say that monoliths no longer exist or that they won't still be created for ages to come. Under various circumstances, a monolithic application that provides numerous capabilities in one package still makes sense, especially in the following settings:

- The domain and thus domain boundaries are largely unknown.
- Provided capabilities are tightly coupled, and absolute performance of module interactions takes precedence over flexibility.
- Scaling requirements for all related capabilities are known and consistent.
- Functionality isn't volatile; change is slow, limited in scope, or both.

For everything else, there are microservices.

This is a gross oversimplification, of course, but I believe it to be a useful summary. By splitting capabilities into smaller, cohesive "chunks," we can decouple them, resulting in the potential for more flexible and robust systems that can be more rapidly deployed and more easily maintained.

In any distributed system—and make no mistake, a system comprising microservices is exactly that—communication is key. No service is an island. And while there are numerous mechanisms for connecting applications/microservices, we often begin our journey by emulating the very fabric of our daily lives: the internet.

The internet was built for communication. In fact, the designers of its precursor, the Advanced Research Projects Agency Network (ARPANET), anticipated a need to maintain intersystem communication even in the event of “significant interruption.” It is reasonable to conclude that an HTTP-based approach similar to the one we use to conduct a great deal of our daily lives could also ably allow us to create, retrieve, update, and delete various resources “over the wire.”

As much as I love history, I won’t dive deeper into the history of REST APIs other than to say that Roy Fielding laid out their principles in his 2000 PhD dissertation, which built upon the *HTTP object model* from 1994.

What Is REST, and Why Does It Matter?

As mentioned earlier, an API is the specification/interface that we developers *write to* so our code can use other code: libraries, other applications, or services. But what does the *REST* in *REST API* represent?

REST is an acronym for *representational state transfer*, which is a somewhat cryptic way of saying that when one application communicates with another, Application A brings its current state with it; it doesn’t expect Application B to maintain state— current and cumulative, process-based information—between communication calls. Application A supplies a representation of its relevant state with each request to Application B. You can easily see why this increases survivability and resilience, because if there is a communication issue or Application B crashes and is restarted, it doesn’t lose the current state of its interactions with Application A; Application A can simply reissue the request and pick up where the two applications left off.

NOTE

This general concept is often referred to as *stateless* applications/services, because each service maintains its own current state, even within a sequence of interactions, and doesn’t expect others to do so on its behalf.

Your API, HTTP Verb Style

Now, about that REST API—sometimes called a RESTful API, which is a nice, relaxing take on things, isn’t it?

There are a number of standardized HTTP verbs defined within a handful of Internet Engineering Task Force (IETF) requests for comments (RFCs). Of these, a small number are typically used consistently for building APIs, with a couple more that find occasional use. REST APIs are primarily built upon the following HTTP verbs:

- `POST`
- `GET`
- `PUT`
- `PATCH`
- `DELETE`

These verbs correspond with typical operations we perform on resources: create (`POST`), read (`GET`), update (`PUT` and `PATCH`), and delete (`DELETE`).

NOTE

I'm admittedly blurring the lines somewhat by loosely equating `PUT` with updating a resource, and a bit less so by equating `POST` with creating a resource. I would ask the reader to bear with me as I step through the implementation and provide clarifications.

Occasionally, the following two verbs are employed:

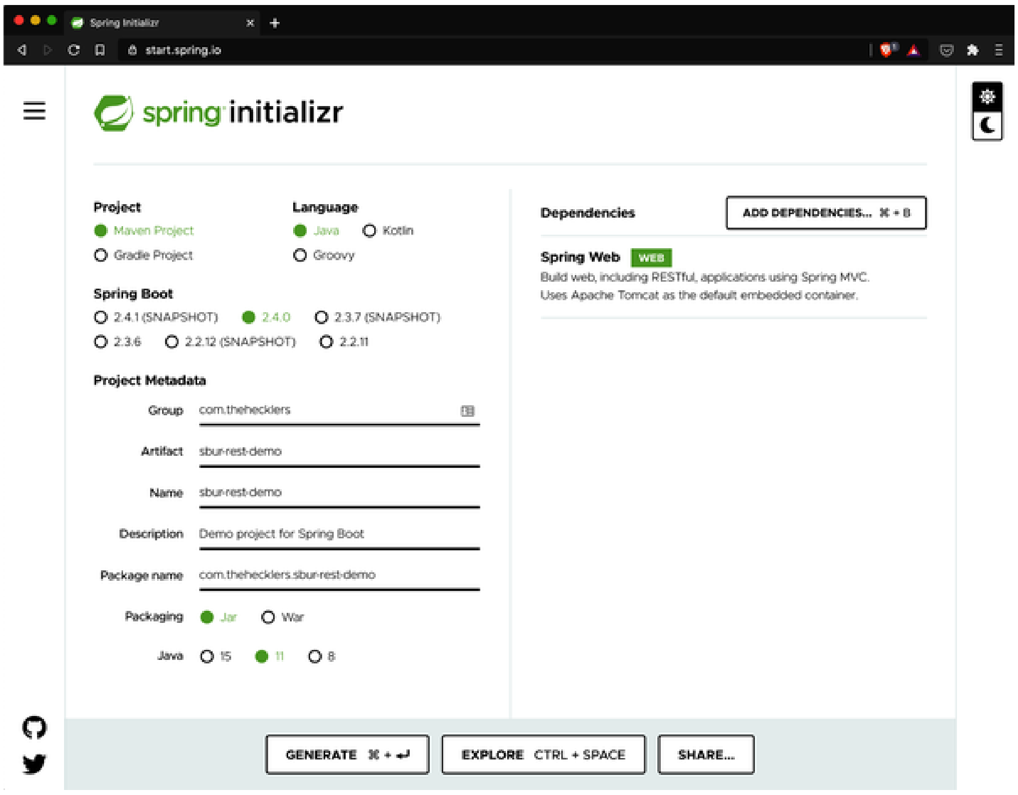
- `OPTIONS`
- `HEAD`

These can be used to retrieve the communication options available for request/response pairs (`OPTIONS`) and retrieve a response header minus its body (`HEAD`).

For this book, and indeed for most production use, I will focus on the first, heavily utilized group. To get (no pun intended) started, let's create a simple microservice that implements a very basic REST API.

Back to the Initializr

We begin as usual with the Spring Initializr, as shown in Figure 3-1. I've changed the Group and Artifact fields to reflect the details I use (please feel free to use your preferred nomenclature), selected Java 11 under Options (optional, any listed version will do nicely), and selected only the Spring Web dependency. As it indicates in the displayed description, this dependency brings with it several capabilities, including that of "[building] web, *including RESTful*, applications using Spring MVC" (emphasis added). This is exactly what we require for the task at hand.



Project: Maven Project [selected], Gradle Project
Language: Java [selected], Kotlin, Groovy
Spring Boot: 2.4.1 (SNAPSHOT), 2.4.0 [selected], 2.3.7 (SNAPSHOT), 2.3.6, 2.2.12 (SNAPSHOT), 2.2.11

Project Metadata

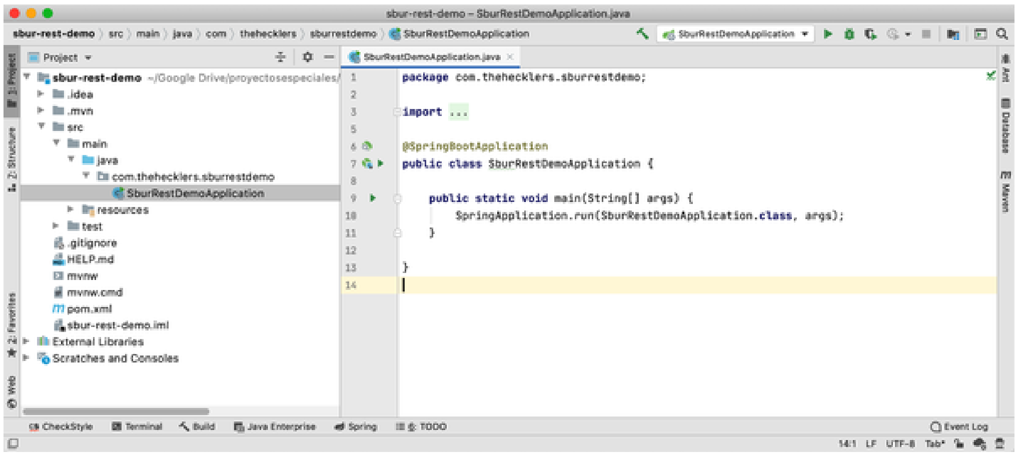
Group	com.thehecklers
Artifact	sbur-rest-demo
Name	sbur-rest-demo
Description	Demo project for Spring Boot
Package name	com.thehecklers.sbur-rest-demo
Packaging	Jar [selected], War
Java	15, 11 [selected], 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Figure 3-1. Creating a Spring Boot project to build a REST API

Once we've generated the project in the Initializr and saved the resultant *.zip* file locally, we'll extract the compressed project files—typically by double-clicking the *sbur-rest-demo.zip* file that was downloaded in your file browser or by using *unzip* from a shell/terminal window—and then open the now-extracted project in your chosen IDE or text editor for a view similar to Figure 3-2.



sbur-rest-demo – SburRestDemoApplication.java

```
sbur-rest-demo ~/Google Drive/proyectosespeciales/
  .idea
  .mvn
  src
    main
      java
        com.thehecklers.sburrestdemo
          SburRestDemoApplication
      resources
    test
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
  sbur-rest-demo.iml
External Libraries
Scratches and Consoles
```

```java
package com.thehecklers.sburrestdemo;

import ...

@SpringBootApplication
public class SburRestDemoApplication {

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

}
```

sbur-rest-demo › src › main › java › com › thehecklers › sburrestdemo › SburRestDemoApplication
Project
CheckStyle
Terminal
Build
Java Enterprise
Spring
TODO
Event Log
14:1 LF UTF-8 Tab*

Figure 3-2. Our new Spring Boot project, just waiting for us to begin

Creating a Simple Domain

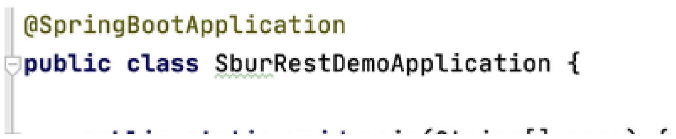
In order to *work with* resources, we'll need to write some code to *accommodate some* resources. Let's start by creating a very simple domain class that represents a resource we want to manage.

I’m somewhat of a coffee aficionado, as my good friends—this now includes you—know. With that in mind, I’ll be using a coffee domain, with a class representing a particular type of coffee, as the domain for this example.

Let’s begin by creating the `Coffee` class. This is essential to the example, as we need a resource of some kind to demonstrate how to manage resources via a REST API. But the domain simplicity or complexity is incidental to the example, so I’ll keep it simple to focus on the objective: the resultant REST API.

As shown in Figure 3-3, the `Coffee` class has two member variables:

- An `id` field used to uniquely identify a particular kind of coffee
- A `name` field that describes the coffee by name



```
@SpringBootApplication
public class SburRestDemoApplication {
```

Figure 3-3. The Coffee class: our domain class

I declare the `id` field as `final` so that it can be assigned only once and never modified; as such, this also requires that it is assigned when creating an instance of class `Coffee` and implies that it has no mutator method.

I create two constructors: one that takes both parameters and another that provides a unique identifier if none is provided upon creation of a `Coffee`.

Next, I create accessor and mutator methods—or getter and setter methods, if you prefer to call them that—for the `name` field, which is not declared `final` and is thus mutable. This is a debatable design decision, but it serves our upcoming needs for this example well.

With that, we now have a basic domain in place. Next it's time for a REST.

GET-ting

Perhaps the most-used of the most-used verbs is `GET`. So let's *get* (pun intended) started.

@RestController in a Nutshell

Without going too deep down the rabbit hole, Spring MVC (Model-View-Controller) was created to separate concerns between data, its delivery, and its presentation, assuming the views would be provided as a server-rendered web page. The `@Controller` annotation helps tie the various pieces together.

`@Controller` is a stereotype/alias for the `@Component` annotation, meaning that upon application startup, a Spring Bean—an object created and managed by the Spring inversion of control (IoC) container within the app—is created from that class. `@Controller`-annotated classes accommodate a `Model` object to provide model-based data to the presentation layer and work with a `ViewResolver` to direct the application to display a particular view, as rendered by a view technology.

NOTE

Spring supports several view technologies and templating engines, covered in a later chapter.

It's also possible to instruct a `Controller` class to return a formatted response as JavaScript Object Notation (JSON) or as another data-oriented format such as XML simply by adding an `@ResponseBody` annotation to the class or method (JSON by default). This results in the Object/Iterable return value of a method being the *entire body* of the response to a web request, instead of being returned as a part of the `Model`.

The `@RestController` annotation is a convenience notation that combines `@Controller` with `@ResponseBody` into a single descriptive annotation, simplifying your code and making intent more obvious. Once we've annotated a class as an `@RestController`, we can begin creating our REST API.

Let's `GET` Busy

REST APIs deal with objects, and objects can come alone or as a group of related objects. To leverage our coffee scenario, you may want to retrieve a particular coffee; you may instead wish to retrieve all coffees, or all that are considered dark roast, fall within a range of identifiers, or include "Colombian" in the description, for example. To accommodate the need to retrieve one instance or more than one instance of an object, it is a good practice to create multiple methods in our code.

I'll begin by creating a list of `Coffee` objects to support the method returning multiple `Coffee` objects, as shown in the rudimentary class definition that follows. I define the variable holding this group of coffees as a `List` of `Coffee` objects. I choose `List` as the higher-level interface for my member variable type, but I'll actually assign an empty `ArrayList` for use within the `RestApiDemoController` class:

```
@RestController
class RestApiDemoController {
        private List<Coffee> coffees = new ArrayList<>();
}
```

NOTE

It's a recommended practice to adopt the highest level of type (class, interface) that can cleanly satisfy internal and external APIs. These may not match in all cases, as they don't here. Internally, `List` provides the level of API that enables me to create the cleanest implementation based on my criteria; externally, we can define an even higher-level abstraction, as I'll demonstrate shortly.

It's always a good idea to have some data to retrieve in order to confirm that everything is working as expected. In the following code, I create a constructor for the `RestApiDemoController` class, adding code to populate the list of coffees upon object creation:

```
@RestController
class RestApiDemoController {
	private List<Coffee> coffees = new ArrayList<>();

	public RestApiDemoController() {
		coffees.addAll(List.of(
				new Coffee("Café Cereza"),
				new Coffee("Café Ganador"),
				new Coffee("Café Lareño"),
				new Coffee("Café Três Pontas")
		));
	}
}
```

As shown in the following code, I create a method in the `RestApiDemoController` class that returns an iterable group of coffees represented by our member variable `coffees`. I choose to use an `Iterable<Coffee>` because any iterable type will satisfactorily provide this API's desired functionality:

.Using `@RequestMapping` to `GET` the list of coffees

```
@RestController
class RestApiDemoController {
	private List<Coffee> coffees = new ArrayList<>();

	public RestApiDemoController() {
		coffees.addAll(List.of(
				new Coffee("Café Cereza"),
				new Coffee("Café Ganador"),
				new Coffee("Café Lareño"),
				new Coffee("Café Três Pontas")
		));
	}

	@RequestMapping(value = "/coffees", method = RequestMethod.GET)
	Iterable<Coffee> getCoffees() {
		return coffees;
	}
}
```

To the `@RequestMapping` annotation, I add a path specification of `/coffees` and a method type of `RequestMethod.GET`, indicating that the method will respond to requests with the path of `/coffees` and restrict requests to only `HTTP GET` requests. Retrieval of data is handled by this method, but updates of any kind are not. Spring Boot, via the Jackson dependencies included in Spring Web, performs the marshalling and unmarshalling of objects to JSON or other formats automatically.

We can streamline this even further using another convenience annotation. Using `@GetMapping` incorporates the direction to allow only `GET` requests, reducing boilerplate and requiring only the path to be specified, even omitting the `path =` since no parameter deconfliction is required. The following code clearly demonstrates the readability benefits of this annotation swap:

```
@GetMapping("/coffees")
Iterable<Coffee> getCoffees() {
    return coffees;
}
```

HELPFUL HINTS ABOUT @REQUESTMAPPING

`@RequestMapping` has several specialized convenience annotations:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@PatchMapping`
- `@DeleteMapping`

Any of these mapping annotations can be applied at the class or method level, and paths are additive. For example, if I were to annotate the `RestApiDemoController` and its `getCoffees()` method as shown in the code, the application would respond exactly the same as it does with the following code shown in the previous two code snippets:

```
@RestController
@RequestMapping("/")
class RestApiDemoController {
        private List<Coffee> coffees = new ArrayList<>();

        public RestApiDemoController() {
                coffees.addAll(List.of(
                                new Coffee("Café Cereza"),
                                new Coffee("Café Ganador"),
                                new Coffee("Café Lareño"),
                                new Coffee("Café Três Pontas")
                ));
        }

        @GetMapping("/coffees")
        Iterable<Coffee> getCoffees() {
                return coffees;
        }
}
```

Retrieving all coffees in our makeshift datastore is useful, but it isn't enough. What if we want to retrieve one particular coffee?

Retrieving a single item works similarly to retrieving several items. I'll add another method called `getCoffeeById` to manage this for us, as shown in the next code segment.

The `{id}` portion of the specified path is a URI (uniform resource identifier) variable, and its value is passed to the `getCoffeeById` method via the `id` method parameter by annotating it with `@PathVariable`.

Iterating over the list of coffees, the method returns a populated `Optional<Coffee>` if it finds a match, or an empty `Optional<Coffee>` if the `id` requested isn't present in our small group of coffees:

```
@GetMapping("/coffees/{id}")
Optional<Coffee> getCoffeeById(@PathVariable String id) {
    for (Coffee c: coffees) {
        if (c.getId().equals(id)) {
            return Optional.of(c);
        }
    }

    return Optional.empty();
}
```

POST-ing

To create resources, an `HTTP POST` method is the preferred option.

NOTE

A `POST` supplies details of a resource, typically in JSON format, and requests that the destination service creates that resource *under* the specified URI.

As shown in the next code fragment, a `POST` is a relatively simple affair: our service receives the specified coffee details as a `Coffee` object—thanks to Spring Boot's automatic marshalling—and adds it to our list of coffees. It then returns the `Coffee` object—automatically unmarshalled by Spring Boot to JSON by default—to the requesting application or service:

```
@PostMapping("/coffees")
Coffee postCoffee(@RequestBody Coffee coffee) {
    coffees.add(coffee);
    return coffee;
}
```

PUT-ting

Generally speaking, `PUT` requests are used to update existing resources with known URIs.

NOTE

Per the IETF's document titled Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, `PUT` requests should update the specified resource if present; if the resource doesn't already exist, it should be created.

The following code operates in accordance with the specification: search for the coffee with the specified identifier, and if found, update it. If no such coffee is contained within the list, create it:

```
@PutMapping("/coffees/{id}")
Coffee putCoffee(@PathVariable String id, @RequestBody Coffee coffee) {
    int coffeeIndex = -1;

    for (Coffee c: coffees) {
        if (c.getId().equals(id)) {
            coffeeIndex = coffees.indexOf(c);
            coffees.set(coffeeIndex, coffee);
        }
    }

    return (coffeeIndex == -1) ? postCoffee(coffee) : coffee;
}
```

DELETE-ing

To delete a resource, we use an `HTTP DELETE` request. As shown in the next code fragment, we create a method that accepts a coffee's identifier as an `@PathVariable` and removes the applicable coffee from our list using the `removeIf` `Collection` method. `removeIf` accepts a `Predicate`, meaning we can provide a lambda to evaluate that will return a boolean value of `true` for the desired coffee to remove. Nice and tidy:

```
@DeleteMapping("/coffees/{id}")
void deleteCoffee(@PathVariable String id) {
    coffees.removeIf(c -> c.getId().equals(id));
}
```

And More

While there are many more things that could be done to improve this scenario, I'm going to focus on two in particular: reducing repetition and returning HTTP status codes where required by the specification.

To reduce repetition in the code, I’ll elevate the portion of the URI mapping that is common to all methods within the `RestApiDemoController` class to the class-level `@RequestMapping` annotation, `"/coffees".` We can then remove that same portion of the URI from each method’s mapping URI specification, reducing the textual noise level a bit, as the following code shows:

```
@RestController
@RequestMapping("/coffees")
class RestApiDemoController {
        private List<Coffee> coffees = new ArrayList<>();

        public RestApiDemoController() {
                coffees.addAll(List.of(
                                new Coffee("Café Cereza"),
                                new Coffee("Café Ganador"),
                                new Coffee("Café Lareño"),
                                new Coffee("Café Três Pontas")
                ));
        }

        @GetMapping
        Iterable<Coffee> getCoffees() {
                return coffees;
        }

        @GetMapping("/{id}")
        Optional<Coffee> getCoffeeById(@PathVariable String id) {
                for (Coffee c: coffees) {
                        if (c.getId().equals(id)) {
                                return Optional.of(c);
                        }
                }

                return Optional.empty();
        }

        @PostMapping
        Coffee postCoffee(@RequestBody Coffee coffee) {
                coffees.add(coffee);
                return coffee;
        }

        @PutMapping("/{id}")
        Coffee putCoffee(@PathVariable String id, @RequestBody Coffee coffe
                int coffeeIndex = -1;

                for (Coffee c: coffees) {
                        if (c.getId().equals(id)) {
                                coffeeIndex = coffees.indexOf(c);
                coffees.set(coffeeIndex, coffee);
            }
        }

        return (coffeeIndex == -1) ? postCoffee(coffee) : coffee;
    }

    @DeleteMapping("/{id}")
    void deleteCoffee(@PathVariable String id) {
        coffees.removeIf(c -> c.getId().equals(id));
    }
}
```

Next, I consult the IETF document referenced earlier and note that while HTTP status codes are not specified for `GET` and suggested for `POST` and `DELETE` methods, they are required for `PUT` method responses. To accomplish this, I modify the `putCoffee` method as shown in the following code segment. Instead of returning only the updated or created `Coffee` object, the `putCoffee` method will now return a `ResponseEntity` containing said `Coffee` and the appropriate HTTP status code: 201 (Created) if the `PUT` coffee didn't already exist, and 200 (OK) if it existed and was updated. We could do more, of course, but the current application code fulfills requirements and represents straightforward and clean internal and external APIs:

```
@PutMapping("/{id}")
ResponseEntity<Coffee> putCoffee(@PathVariable String id,
    @RequestBody Coffee coffee) {
  int coffeeIndex = -1;

  for (Coffee c: coffees) {
    if (c.getId().equals(id)) {
      coffeeIndex = coffees.indexOf(c);
      coffees.set(coffeeIndex, coffee);
    }
  }

  return (coffeeIndex == -1) ?
      new ResponseEntity<>(postCoffee(coffee), HttpStatus.CREATED) :
      new ResponseEntity<>(coffee, HttpStatus.OK);
}
```

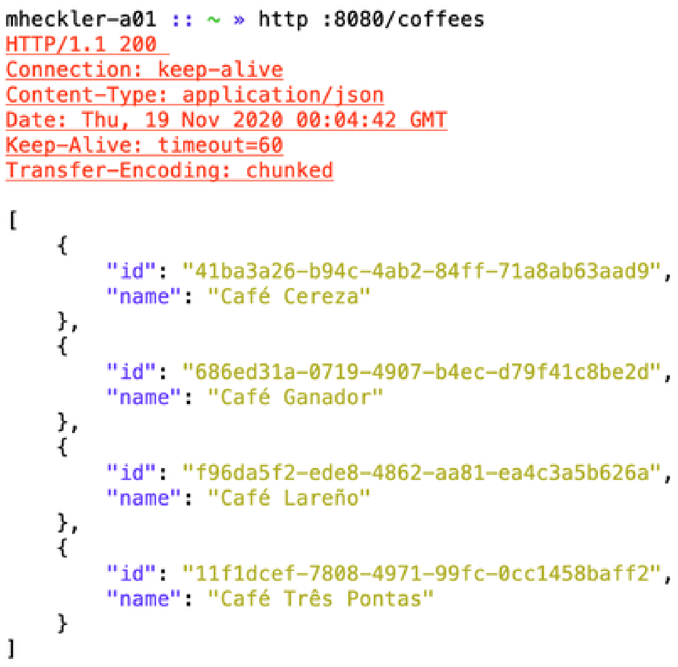
Trust, but Verify

With all of the code in place, let's put this API through its paces.

NOTE

I use the HTTPie command line HTTP client for nearly all of my HTTP-based chores. Occasionally I will also use curl or Postman, but I find HTTPie to be a versatile client with a streamlined command-line interface and superb utility.

As shown in Figure 3-4, I query the *coffees* endpoint for all coffees currently in our list. HTTPie defaults to a `GET` request and assumes *localhost* if no hostname is provided, reducing unnecessary typing. As expected, we see all four coffees with which we prepopulated our list.

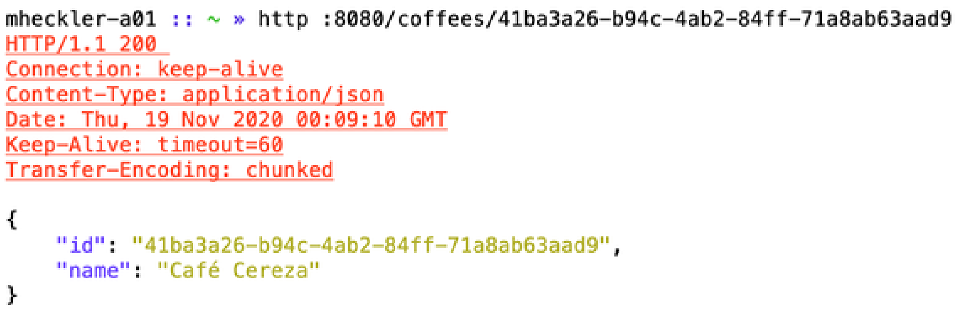


```
mheckler-a01 :: ~ » http :8080/coffees
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Thu, 19 Nov 2020 00:04:42 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

[
    {
        "id": "41ba3a26-b94c-4ab2-84ff-71a8ab63aad9",
        "name": "Café Cereza"
    },
    {
        "id": "686ed31a-0719-4907-b4ec-d79f41c8be2d",
        "name": "Café Ganador"
    },
    {
        "id": "f96da5f2-ede8-4862-aa81-ea4c3a5b626a",
        "name": "Café Lareño"
    },
    {
        "id": "11f1dcef-7808-4971-99fc-0cc1458baff2",
        "name": "Café Três Pontas"
    }
]
```

Figure 3-4. `GET`-ting all coffees

Next, I copy the `id` field for one of the coffees just listed and paste it into another `GET` request. Figure 3-5 displays the correct response.

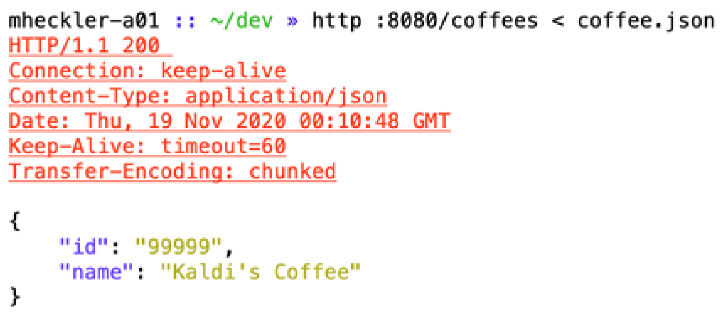


```
mheckler-a01 :: ~ » http :8080/coffees/41ba3a26-b94c-4ab2-84ff-71a8ab63aad9
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Thu, 19 Nov 2020 00:09:10 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "id": "41ba3a26-b94c-4ab2-84ff-71a8ab63aad9",
    "name": "Café Cereza"
}
```

Figure 3-5. `GET`-ting a coffee

To execute a `POST` request with HTTPie is simple: just pipe a plaintext file containing a JSON representation of a `Coffee` object with `id` and `name` fields, and HTTPie assumes that a `POST` operation is in order. Figure 3-6 shows the command and its successful outcome.



```
mheckler-a01 :: ~/dev » http :8080/coffees < coffee.json
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Thu, 19 Nov 2020 00:10:48 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

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

Figure 3-6. `POST` -ing a new coffee to the list

As mentioned earlier, 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 3-7, I specify the `id` of the coffee I just added and pass another JSON object with a different name to the command. The result is that the coffee with `id` of "99999" now has a `name` of "Caribou Coffee" instead of "Kaldi's Coffee" as before. The return code is 200 (OK), as expected as well.

Figure 3-7. `PUT` -ting an update to an existing coffee

In Figure 3-8 I initiate a `PUT` request in the same manner but reference an `id` in the URI that doesn't exist. The application dutifully adds it in compliance with IETF-specified behavior and correctly returns an HTTP status of 201 (Created).

Figure 3-8. `PUT` -ting a new coffee

Creating a `DELETE` request with HTTPie is very similar to creating a `PUT` request: the HTTP verb must be specified, and the resource's URI must be complete. Figure 3-9 shows the result: an HTTP status code of 200 (OK), indicating the resource was successfully deleted and with no shown value, since the resource no longer exists.

Figure 3-9. `DELETE` -ing a coffee

Finally, we re-query our full list of coffees to confirm the expected final state. As Figure 3-10 demonstrates, we now have one additional coffee that wasn't in our list before, as expected: Mötor Oil Coffee. API validation is a success.

Figure 3-10. `GET` -ting all coffees now in the list

Summary

This chapter demonstrated how to develop a basic working application using Spring Boot. Since most applications involve exposing backend cloud resources to users, usually via a frontend user interface, I showed how to create and evolve a useful REST API that can be consumed in numerous, consistent ways to provide needed functionality for creating, reading, updating, and deleting resources central to nearly every critical system.

I examined and explained the `@RequestMapping` annotation and its various convenience annotation specializations that align with defined HTTP verbs:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@PatchMapping`
- `@DeleteMapping`

After creating methods that addressed many of these annotations and their associated actions, I then refactored the code a bit to streamline it and provide HTTP response codes where required. Validating the API confirmed its correct operation.

In the next chapter, I discuss and demonstrate how to add database access to our Spring Boot application to make it increasingly useful and ready for production.