KAPITEL 4

Datenbankzugriff für Ihre Spring-Boot-Anwendung

Wie im vorherigen Kapitel besprochen, setzen Anwendungen aus vielen guten Gründen oft zustandslose APIs ein. Hinter den Kulissen dagegen sind nur wenige sinnvolle Anwendungen wirklich ganz flüchtig; meist wird irgendeine Art von Zustand für *irgendetwas* gespeichert. So könnte z.B. jede Anfrage an den Warenkorb eines Onlineshops auch dessen Zustand enthalten. Doch sobald die Bestellung abgeschlossen ist, werden die Daten dieser Bestellung aufbewahrt. Es gibt viele Möglichkeiten, dies zu erreichen, und viele Wege, um diese Daten zu teilen oder weiterzuleiten. In fast allen Systemen einer gewissen Größe sind dabei fast ausnahmslos eine oder mehrere Datenbanken beteiligt.

In diesem Kapitel werde ich Ihnen zeigen, wie Sie der Spring-Boot-Anwendung, die Sie im vorangegangenen Kapitel erstellt haben, einen Datenbankzugang hinzufügen. Sie werden einen ersten Einblick in die Datenfähigkeiten von Spring Boot erhalten; in den nachfolgenden Kapiteln steigen wir dann tiefer in dieses Thema ein. In vielen Fällen sind die hier behandelten Grundlagen jedoch ausreichend und bieten eine völlig zufriedenstellende Lösung. Legen wir los.

Hinweis zum Code-Checkout
Checken Sie bitte zu Beginn den Zweig *chapter4begin* aus dem Code-Repository aus.

Die Autokonfiguration für den Datenbankzugriff vorbereiten

Wie bereits gezeigt, zielt Spring Boot darauf ab, den sogenannten 80-bis-90-Prozent-Anwendungsfall um das maximal mögliche Maß zu vereinfachen: die Muster aus Code und Verarbeitung, die Entwickler immer und immer wieder durchlaufen müssen. Nachdem Muster identifiziert wurden, tritt Boot in Aktion und initialisiert automatisch die erforderlichen Beans mit vernünftigen Standardkonfigurationen. Das Anpassen einer Funktionalität ist so einfach wie das Bereitstellen eines oder

mehrerer Eigenschaftswerte oder das Erstellen einer maßgeschneiderten Version einer oder mehrerer Beans; sobald die Autokonfiguration die Änderungen erkennt, zieht sie sich zurück und folgt der Führung des Entwicklers. Der Datenbankzugriff ist das perfekte Beispiel.

Was wollen wir erreichen?

In unserer früheren Beispielanwendung verwendete ich ein `ArrayList` zum Speichern und Pflegen unserer Kaffeeliste. Diese Vorgehensweise ist einfach genug für eine einzelne Anwendung, hat aber auch ihre Nachteile.

Erstens ist sie überhaupt nicht belastbar. Wenn Ihre Anwendung oder die Plattform, auf der sie läuft, abstürzt, verschwinden alle Änderungen, die vorgenommen wurden, während die Anwendung – ob nun für Sekunden oder für Monate – lief.

Zweitens skaliert sie nicht. Das Starten einer weiteren Instanz der Anwendung führt dazu, dass die zweite (oder jede nachfolgende) Instanz ihre eigene Liste von Kaffees hat, die sie pflegen muss. Die Daten werden zwischen den verschiedenen Instanzen nicht geteilt, was bedeutet, dass Änderungen an einer Instanz – neue Kaffees, gelöschte oder aktualisierte Einträge – für niemanden sichtbar sind, der auf eine andere Instanz der Anwendung zugreift.

Das ist ganz sicher nicht der richtige Weg, so etwas zu betreiben.

Ich werde in den folgenden Kapiteln verschiedene Möglichkeiten beschreiben, diese sehr realen Probleme zu lösen. Hier an dieser Stelle wollen wir das Fundament errichten, das uns als Grundlage dafür dienen soll.

Hinzufügen einer Datenbankabhängigkeit

Um aus Ihrer Spring-Boot-Anwendung heraus Zugriff auf eine Datenbank zu erhalten, benötigen Sie einige Dinge:

- eine laufende Datenbank, ob initiiert durch/eingebettet in Ihre Anwendung oder einfach durch ihre Anwendung zugreifbar
- Datenbanktreiber, die einen programmatischen Zugriff erlauben, üblicherweise bereitgestellt durch den Hersteller der Datenbank
- ein Spring-Data-Modul zum Zugriff auf die Zieldatenbank

Bestimmte Spring-Data-Module enthalten passende Datenbanktreiber als einzeln wählbare Abhängigkeit aus dem Spring Initializr heraus. In anderen Fällen, etwa wenn Spring das Java Persistence API (JPA) zum Zugriff auf JPA-konforme Datenspeicher verwendet, ist es notwendig, die Spring-Data-JPA-Abhängigkeit *und* eine Abhängigkeit für den speziellen Treiber der Zieldatenbank, z. B. PostgreSQL, auszuwählen.

Um den ersten Schritt von Speicherkonstrukten zu einer persistenten Datenbank zu gehen, füge ich zunächst die Abhängigkeiten und damit die Funktionalitäten zur Build-Datei unseres Projekts hinzu.

H2 ist eine komplett in Java geschriebene schnelle Datenbank, die einige interessante und nützliche Eigenschaften besitzt. Zum einen ist sie JPA-konform, sodass wir die Verbindung zu unserer Anwendung auf die gleiche Weise herstellen können wie bei jeder anderen JPA-Datenbank wie Microsoft SQL, MySQL, Oracle oder PostgreSQL. Außerdem besitzt sie In-Memory- und Festplattenmodi. Dies bietet uns einige interessante Möglichkeiten, nachdem wir unsere im Speicher vorgehaltene (in-memory) `ArrayList` in eine im Speicher vorgehaltene Datenbank umgewandelt haben: Wir können entweder H2 persistent auf der Festplatte ablegen oder – da wir nun eine JPA-Datenbank verwenden – auf eine andere JPA-Datenbank wechseln. Beide Möglichkeiten sind an diesem Punkt viel einfacher umzusetzen.

Um es unserer Anwendung zu erlauben, mit der H2-Datenbank zu interagieren, füge ich die folgenden zwei Abhängigkeiten in den Abschnitt `<dependencies>` der *pom.xml* unseres Projekts ein:

```
<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>
```

Der Wirkungsbereich `runtime` der H2-Datenbanktreiber-Abhängigkeit bedeutet, dass sie im Laufzeit- und Test-Klassenpfad vorhanden ist, nicht jedoch im Compile-Klassenpfad. Diese Praxis ist empfehlenswert für Bibliotheken, die nicht für das Kompilieren erforderlich sind.

Wenn Sie Ihre aktualisierte *pom.xml* gespeichert und (falls notwendig) Ihre Maven-Abhängigkeiten neu importiert/aktualisiert haben, haben Sie Zugriff auf die Funktionalität, die in den hinzugefügten Abhängigkeiten enthalten ist. Jetzt schreiben wir erst einmal ein bisschen Code, um sie zu benutzen.

Code hinzufügen

Da wir bereits Code haben, um die Kaffees in gewisser Weise zu verwalten, müssen wir das Ganze ein bisschen umbauen, um unsere neuen Datenbankfähigkeiten in Stellung zu bringen. Ich finde, am besten ist es, mit den Domain-Klassen beziehungsweise in diesem Fall mit der Domain-Klasse zu beginnen, nämlich `Coffee`.

Die @Entity

Wie ich bereits erwähnte, ist H2 eine JPA-konforme Datenbank, sodass ich JPA-Annotationen hinzufüge. Zur Klasse `Coffee` selbst füge ich eine `@Entity`-Annotation aus `javax.persistence` hinzu, die signalisiert, dass `Coffee` eine dauerhaft speicherbare Entität ist. Zur existierenden `id`-Membervariablen füge ich die Annotation `@Id`

(auch aus `javax.persistence`) hinzu, um sie als ID-Feld der Datenbanktabelle zu kennzeichnen.

Falls der Klassenname – hier: `Coffee` – nicht dem gewünschten Datenbanknamen entspricht, akzeptiert die `@Entity`-Annotation einen `name`-Parameter, mit dem sich der Name der Datentabelle an die annotierte Entität anpassen lässt.

Eine besonders hilfsbereite IDE meldet Ihnen vielleicht sogar zurück, dass in der Klasse `Coffee` noch etwas fehlt. So unterstreicht z. B. IntelliJ den Klassennamen rot und liefert beim Darüberfahren mit der Maus das hilfreiche Pop-up, das in Abbildung 4-1 zu sehen ist.

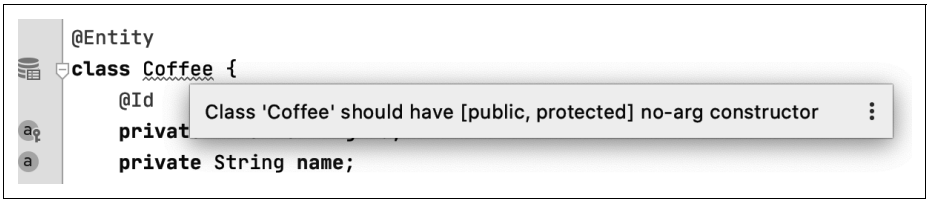

Abbildung 4-1: Fehlender Konstruktor in der JPA-Klasse »Coffee«

Java Persistence API verlangt einen Konstruktor ohne Argument, der verwendet wird, wenn Objekte aus Datenbanktabellenzeilen erzeugt werden. Diesen werde ich deshalb als Nächstes hinzufügen. Und damit bekommen wir auch schon unsere nächste IDE-Warnung, die in Abbildung 4-2 gezeigt wird: Um einen argumentlosen Konstruktor zu bekommen, müssen alle Membervariablen veränderlich sein, das heißt nicht final.

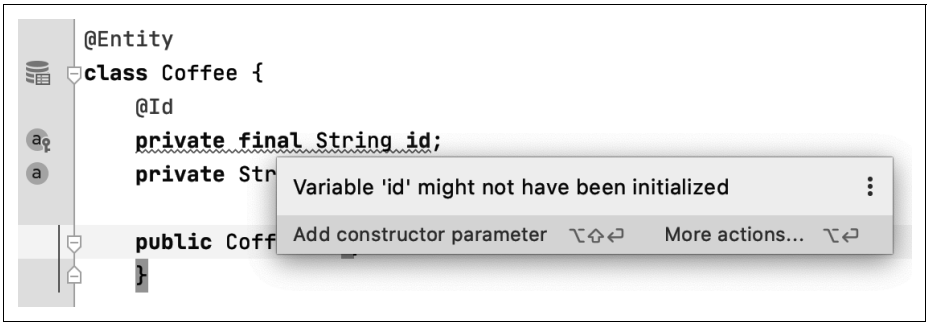

Abbildung 4-2: Bei einem Konstruktor ohne Argument darf »id« nicht final sein.

Wir lösen das, indem wir das Schlüsselwort `final` aus der Deklaration der Membervariablen `id` entfernen. Wenn wir `id` veränderbar machen, benötigt unsere Klasse `Coffee` außerdem eine Mutator-Methode für `id`, damit JPA einen Wert zu diesem Member zuweisen kann. Ich füge also noch die Methode `setId()` hinzu, wie Abbildung 4-3 zeigt.

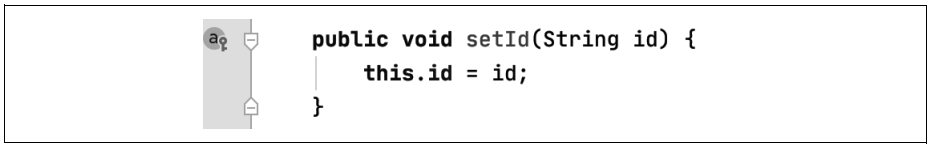

Abbildung 4-3: Die neue »setId()«-Methode

Das Repository

Nachdem `Coffee` nun als gültige JPA-Entität definiert ist, die gespeichert und abgerufen werden kann, wird es Zeit, die Verbindung zur Datenbank herzustellen.

Obwohl es sich eigentlich um ein ganz einfaches Konzept handelt, war das Konfigurieren und Herstellen einer Datenbankverbindung im Java-Ökosystem sehr lange eine ziemlich umständliche Angelegenheit. Wie in Kapitel 1 erwähnt, verlangte der Einsatz eines Application-Servers zum Ablegen einer Java-Anwendung von den Entwicklern die Durchführung mehrerer mühsamer Schritte, um überhaupt einmal auf den richtigen Weg zu kommen. Sobald Sie begonnen hatten, mit der Datenbank zu interagieren oder wenn Sie einen Datenspeicher direkt aus einem Java-Programm oder einer Client-Anwendung heraus angesprochen hatten, mussten Sie zusätzliche Schritte unternehmen, bei denen die `PersistenceUnit`-, `EntityManagerFactory`- und `EntityManager`-APIs (und möglicherweise `DataSource`-Objekte) zum Einsatz kamen, die Datenbank geöffnet und geschlossen werden musste und Weiteres. Für Routineaufgaben der Entwickler waren also zahlreiche sich wiederholende Abläufe erforderlich.

Spring Data führt das Konzept der Repositories ein. Ein `Repository` ist eine Schnittstelle, die in Spring Data als nützliche Abstraktion über den verschiedenen Datenbanken definiert ist. Es gibt noch andere Mechanismen zum Zugriff auf Datenbanken aus Spring Data heraus, die in nachfolgenden Kapiteln erklärt werden, doch in den meisten Fällen sind die verschiedenen Ausprägungen von `Repository` ganz zweifellos am nützlichsten.

`Repository` selbst ist nur ein Platzhalter für die folgenden Typen:

- das Objekt, das in der Datenbank gespeichert ist
- die eindeutige ID/das Primärschlüssel-Feld des Objekts

Natürlich gibt es über Repositories noch viel mehr zu sagen, und ich behandele einen Großteil davon in Kapitel 6. Im Augenblick wollen wir uns auf die zwei konzentrieren, die für unser aktuelles Beispiel direkt relevant sind: `CrudRepository` und `JpaRepository`.

Erinnern Sie sich, dass ich die bevorzugte Praxis erwähnt habe, Code so zu schreiben, dass die höchstmögliche geeignete Schnittstelle verwendet wird? Während `JpaRepository` eine Handvoll von Schnittstellen bereithält und damit eine breitere Funktionalität erlaubt, deckt `CrudRepository` alle entscheidenden CRUD-Fähigkeiten ab und ist ausreichend für unsere (bisher) einfache Anwendung.

Um Repository-Unterstützung für unsere Anwendung zu aktivieren, muss als Erstes eine Schnittstelle speziell für unsere Anwendung definiert werden, indem eine Spring-Data-`Repository`-Schnittstelle erweitert wird:

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

Die zwei definierten Typen sind der zu speichernde Objekttyp und der Typ seiner eindeutigen ID.

Dies stellt den einfachsten Ausdruck der Repository-Erstellung innerhalb einer Spring-Boot-Anwendung dar. Es ist möglich und manchmal auch ganz sinnvoll, Abfragen für ein Repository zu definieren; auch darauf werde ich in einem späteren Kapitel eingehen. Doch hier ist der »magische« Teil: Spring Boots Autokonfiguration berücksichtigt den Datenbanktreiber auf dem Klassenpfad (in diesem Fall H2), die in unserer Anwendung definierte Repository-Schnittstelle und die JPA-Klassendefinition `Coffee` und erzeugt *in Ihrem Auftrag* eine Datenbank-Proxy-Bean. Es ist nicht nötig, für jede Anwendung immer wieder die fast gleichen Zeilen zu schreiben, wenn die Muster so klar und konsistent sind. Die Entwicklerin kann sich stattdessen darum kümmern, an neuen, wichtigen Funktionen zu arbeiten.

Das Utility oder auch: in Aktion »springen«

Setzen wir nun das Repository ein. Ich werde hier wie in den vorangegangenen Kapiteln wieder Schritt für Schritt vorgehen und zunächst die Funktionalität einführen, bevor ich das Ganze dann verfeinere.

Zuerst injiziere ich die Repository-Bean in den `RestApiDemoController`, damit der Controller darauf zugreifen kann, wenn er Anfragen über das externe API empfängt, wie in Abbildung 4-4 gezeigt.

Zunächst deklariere ich eine Membervariable mit:

```
private final CoffeeRepository coffeeRepository;
```

Anschließend füge ich sie dem Konstruktor als Parameter hinzu:

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

Vor Spring Framework 4.3 war es in allen Fällen erforderlich, die Annotation `@Autowired` vor der Methode anzugeben, um anzuzeigen, wenn ein Parameter eine Spring-Bean darstellte, die injiziert/automatisch verdrahtet (autowired) wurde. Seit 4.3 verlangt eine Klasse mit einem einzigen Konstruktor keine Annotation für automatisch verdrahtete Parameter mehr, was eine schöne Zeitersparnis ist.

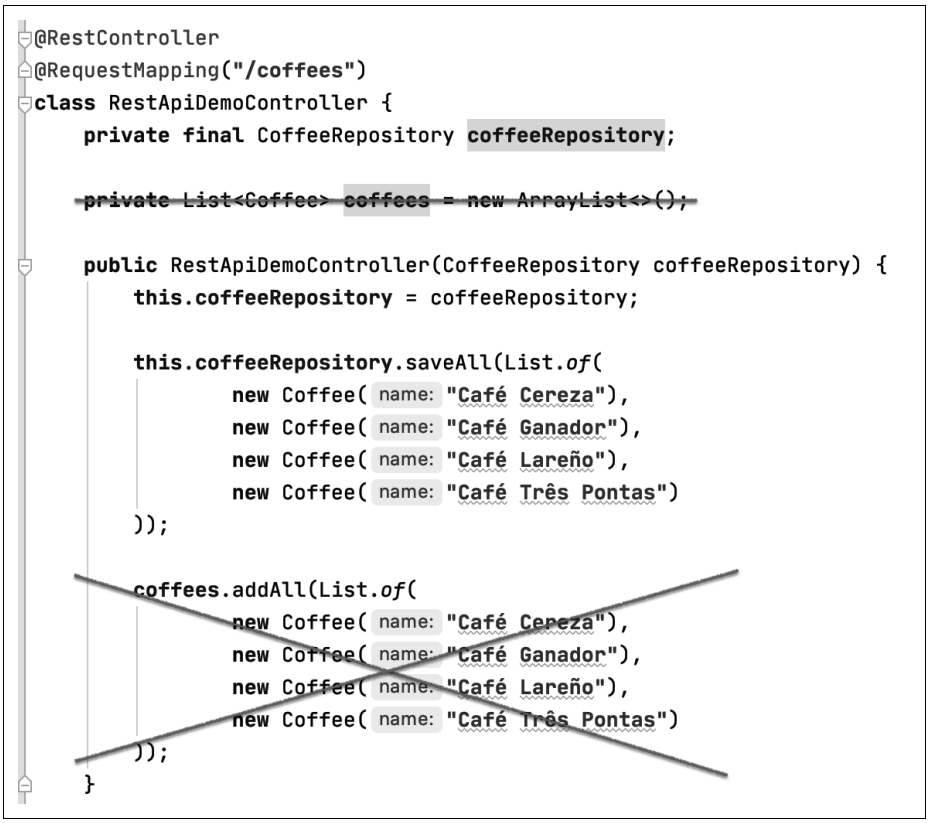

Abbildung 4-4: Einfügen des Repository in »RestApiDemoController«

Wenn das Repository an Ort und Stelle ist, lösche ich die Membervariable `List<Coffee>` und ändere die Anfangsbelegung dieser Liste im Konstruktor, um die Kaffees nun im Repository zu speichern, wie in Abbildung 4-4 gezeigt.

Wie Sie in Abbildung 4-5 sehen, werden beim Entfernen der `coffees`-Variablen sofort alle Referenzen darauf als unauflösbare Symbole angezeigt, sodass die nächste Aufgabe darin besteht, diese durch die entsprechenden Repository-Interaktionen zu ersetzen.

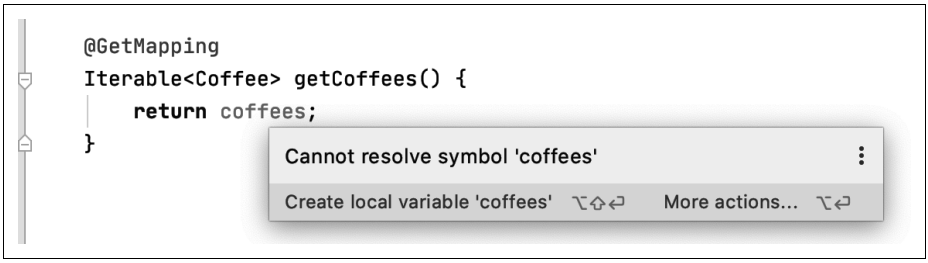

Abbildung 4-5: Ersetzen der gelöschten »coffees«-Membervariablen

Als einfache Abfrage aller Kaffees ohne Parameter ist die Methode `getCoffees()` ein guter Startpunkt. Wenn Sie die `findAll()`-Methode verwenden, die in `CrudRepository` eingebaut ist, müssen Sie nicht einmal den Rückgabetyp von `getCoffees()` ändern, da dies einen `Iterable`-Typ zurückliefert; ein einfaches Aufrufen von `coffeeRepository.findAll()` und Zurückgeben des Ergebnisses reicht völlig aus, wie hier zu sehen ist:

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

Das Refaktorieren der Methode `getCoffeeById()` liefert uns einige Einblicke, wie viel einfacher Ihr Code dank der Funktionalität sein kann, die das Repository mitbringt. Wir müssen die Liste der Kaffees nicht mehr manuell nach einer passenden `id` durchsuchen, die `findById()`-Methode von `CrudRepository` erledigt das für uns, wie der folgende Codeausschnitt beweist. Und da `findById()` einen `Optional`-Typ zurückliefert, sind für unsere Methodensignatur überhaupt keine Änderungen erforderlich:

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

Das Konvertieren der Methode `postCoffee()` für die Benutzung des Repository ist ebenfalls eine relativ einfache Angelegenheit:

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

Die Methode `putCoffee()` demonstriert uns erneut die zeit- und codesparende Funktionalität von `CrudRepository`. Ich verwende die eingebaute `existsById()`-Repository-Methode, um festzustellen, ob dies ein neuer oder ein bereits vorhandener `Coffee` ist, und gebe zusammen mit dem gespeicherten `Coffee` den passenden HTTP-Statuscode zurück, wie dieses Listing zeigt:

```
@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.OK);
}
```

Schließlich aktualisiere ich die Methode `deleteCoffee()` so, dass sie nun die in `CrudRepository` eingebaute `deleteById()`-Methode benutzt:

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

Das Einsetzen einer Repository-Bean, die mit dem »sprechenden« API von `CrudRepository` erzeugt wurde, glättet den Code für den `RestApiDemoController` und macht ihn viel lesbarer und verständlicher, wie das komplette Listing beweist:

```
@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), HttpStatus.OK);
    }

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

Nun müssen wir nur noch verifizieren, dass unsere Anwendung erwartungsgemäß arbeitet und die externe Funktionalität gleich geblieben ist.

Eine alternative Vorgehensweise zum Testen der Funktionalität – und eine empfohlene Praxis – ist es, zuerst Unit-Tests herzustellen, also im Sinne einer Test-driven Development (TDD; testgetriebene Entwicklung) zu arbeiten. Ich rate Ihnen unbedingt, bei realen Softwareentwicklungen so vorzugehen, habe aber festgestellt, dass beim Demonstrieren und Erklären einzelner Entwicklungskonzepte weniger durchaus mehr ist. Deshalb zeige ich nur das Nötigste, wenn ich wichtige Konzepte vermitteln möchte, und gehe auf das Testen in einem eigenen Kapitel weiter hinten in diesem Buch ein.

Speichern und Abrufen von Daten

Begeben wir uns noch einmal daran, liebe Freunde: Greifen wir uns das API von der Kommandozeile mittels HTTPie. Wird der *coffees*-Endpoint abgefragt, erhalten wir dieselben vier Kaffees aus unserer H2-Datenbank wie zuvor – siehe Abbildung 4-6.

Wenn wir das `id`-Feld für einen der gerade aufgelisteten Kaffees kopieren und in eine kaffeespezifische `GET`-Anforderung einfügen, erhalten wir die Ausgabe, die in Abbildung 4-7 zu sehen ist.

```
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"
    }
]
```

Abbildung 4-6: Alle Kaffees mit »GET« abrufen

```
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"
}
```

Abbildung 4-7: Einen Kaffee mit »GET« abrufen

In Abbildung 4-8 schicke ich mit `POST` einen neuen Kaffee an die Anwendung und ihre Datenbank.

```
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"
}
```

Abbildung 4-8: Mit »POST« einen neuen Kaffee an die Liste schicken

Wie wir bereits im vorangegangenen Kapitel diskutiert haben, sollte ein `PUT`-Befehl das Aktualisieren einer vorhandenen Ressource erlauben. Sollte die angeforderte Ressource noch nicht existieren, wird eine neue hinzugefügt. In Abbildung 4-9 gebe ich die `id` des gerade hinzugefügten Kaffees an und übergebe an den Befehl ein JSON-Objekt mit einer Änderung des Namens dieses Kaffees. Nach der Aktualisierung trägt der Kaffee mit der `id` »99999« nun den `name` »Caribou Coffee« statt »Kaldi's Coffee«. Der Rückgabecode ist erwartungsgemäß der Status 200 (OK).

Als Nächstes initiiere ich eine vergleichbare `PUT`-Anforderung, gebe aber in dem URI eine `id` an, die nicht existiert. Die Anwendung fügt nun entsprechend dem IETF-konformen Verhalten einen neuen Kaffee in die Datenbank ein und gibt ganz korrekt den HTTP-Status 201 (Created) zurück, wie Abbildung 4-10 zeigt.

```
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"
}
```

Abbildung 4-9: Mit »PUT« einen vorhandenen Kaffee aktualisieren

```
mheckler-a01 :: ~/dev » http PUT :8080/coffees/88888 < coffee3.json
HTTP/1.1 201
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:25:28 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "id": "88888",
    "name": "Mötor Oil Coffee"
}
```

Abbildung 4-10: Mit »PUT« einen neuen Kaffee anlegen

Schließlich teste ich das Löschen eines bestimmten Kaffees mit einer `DELETE`-Anforderung. Diese gibt nur den HTTP-Statuscode 200 (OK) zurück, der signalisiert, dass die Ressource erfolgreich gelöscht wurde, und sonst nichts – die Ressource gibt es nun ja nicht mehr; siehe Abbildung 4-11. Um unseren Endzustand zu prüfen, rufen wir noch einmal die komplette Liste der Kaffees ab (Abbildung 4-12).

```
mheckler-a01 :: ~/dev » http DELETE :8080/coffees/99999
HTTP/1.1 200
Connection: keep-alive
Content-Length: 0
Date: Wed, 25 Nov 2020 21:26:55 GMT
Keep-Alive: timeout=60
```

Abbildung 4-11: DELETE löscht einen Kaffee.

```
mheckler-a01 :: ~/dev » http :8080/coffees
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Wed, 25 Nov 2020 21:28:20 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"
    },
    {
        "id": "88888",
        "name": "Mötor Oil Coffee"
    }
]
```

Abbildung 4-12: Mit »GET« alle Kaffees abrufen, die nun auf der Liste stehen

Wie zuvor haben wir jetzt einen zusätzlichen Kaffee, der anfangs nicht in unserem Repository stand: Mötor Oil Coffee.

Ein bisschen Nachpolieren

Wie überall gibt es auch hier Bereiche, die noch etwas zusätzliche Aufmerksamkeit vertragen können. Ich werde mich allerdings auf zwei beschränken: das Extrahieren der Anfangsbelegung der Beispieldaten in eine separate Komponente und ein Umsortieren der Bedingungsoperatoren aus Gründen der Klarheit.

Im vorigen Kapitel habe ich die Liste der Kaffees mit einigen Anfangswerten in der Klasse `RestApiDemoController` gefüllt. Diese Struktur habe ich auch in diesem Kapitel – bis jetzt – beibehalten, nachdem ich das Ganze in eine Datenbank mit Repository-Zugriff umgewandelt hatte. Besser ist es jedoch, diese Funktionalität in eine separate Komponente auszulagern, die schnell und einfach aktiviert oder deaktiviert werden kann.

Es gibt viele Möglichkeiten, Code automatisch beim Start einer Anwendung auszuführen. Sie könnten etwa einen `CommandLineRunner` oder `ApplicationRunner` einsetzen und ein Lambda festlegen, um das gewünschte Ziel zu erreichen: in diesem Fall das Erzeugen und Speichern von Beispieldaten. Ich jedoch benutze dafür lieber eine `@Component`-Klasse und eine `@PostConstruct`-Methode. Das hat folgende Gründe:

- Wenn die Bean-erzeugenden Methoden `CommandLineRunner` und `ApplicationRunner` eine Repository-Bean injizieren (automatisch verdrahten), scheitern Unit-Tests, die die Repository-Bean innerhalb des Tests »mocken« (was typischerweise der Fall ist).
- Falls Sie die Repository-Bean in dem Test mocken oder die Anwendung ausführen wollen, ohne Beispieldaten zu erzeugen, lässt sich die eigentliche datenerzeugende Bean schnell und einfach deaktivieren, indem Sie deren `@Component`-Annotation auskommentieren.

Ich empfehle das Anlegen einer `DataLoader`-Klasse ähnlich derjenigen, die Sie im folgenden Codeblock sehen. Wenn Sie die Logik zum Erzeugen der Beispieldaten in die `loadData()`-Methode der `DataLoader`-Klasse auslagern und sie mit `@PostContruct` annotieren, führen Sie `RestApiDemoController` wieder seiner eigentlichen Aufgabe zu, nämlich dem Bereitstellen eines externen API. Der `DataLoader` wiederum wird für *seinen* gedachten (und offensichtlichen) Zweck verantwortlich gemacht:

```
@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")
        ));
    }
}
```

Die andere Verfeinerung ist eine zugegebenermaßen kleine Änderung der booleschen Bedingung des Dreifachoperators in der Methode `putCoffee()`. Nachdem die Methode so umgebaut wurde, dass sie ein Repository benutzt, gibt es keinen zwingenden Grund mehr, zuerst die negative Bedingung auszuwerten. Durch das Entfernen des Not-Operators (`!`) aus der Bedingung wird das Ganze etwas klarer; es ist nun natürlich noch erforderlich, die True- und False-Werte des Dreifachoperators zu vertauschen, um wieder auf die ursprünglichen Ergebnisse zu kommen. Der folgende Code spiegelt das wider:

```
@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);
}
```

Hinweis zum Code-Checkout

Wenn Sie den kompletten Code für dieses Kapitel haben wollen, checken Sie den Zweig *chapter4end* aus dem Code-Repository aus.

Zusammenfassung

Dieses Kapitel demonstrierte, wie Sie die Spring-Boot-Anwendung, die wir im letzten Kapitel erzeugt haben, um Datenbankzugriff erweitern. Sie sollten kurz und bündig in die Datenfähigkeiten von Spring Boot eingeführt werden, und dazu habe ich Ihnen einen Überblick über folgende Dinge geboten:

- Java-Datenbankzugriff
- Java Persistence API (JPA)
- H2-Datenbank
- Spring Data JPA
- Spring-Data-Repositorys
- Mechanismen zum Erzeugen von Beispieldaten über Repositorys

Nachfolgende Kapitel werden noch tiefer in den Datenbankzugriff mit Spring Boot eintauchen. Die Grundlagen, die Sie in diesem Kapitel kennengelernt haben, bilden jedoch bereits eine solide Basis, auf der Sie aufbauen können, und sind in vielen Fällen selbst schon ausreichend.

Im nächsten Kapitel diskutiere und demonstriere ich nützliche Spring-Boot-Werkzeuge, mit denen Sie unter die Motorhaube Ihrer Anwendung schauen können, wenn die Dinge einmal nicht so laufen, wie Sie es erwarten, oder Sie nachweisen müssen, dass sie es tun.