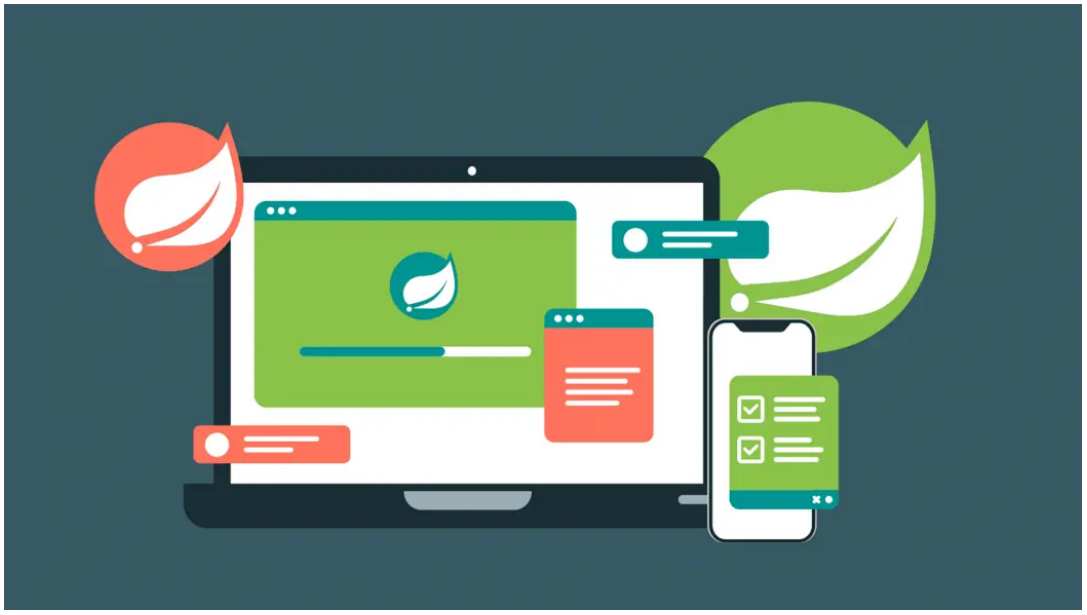
MAGAZIN FÜR PROFESSIONELLE
INFORMATIONSTECHNIK

Spring Boot: Aufbau und Funktion einer Anwendung am Beispiel eines REST-Service

05.04.2023 09:30 Uhr Julius Mischok

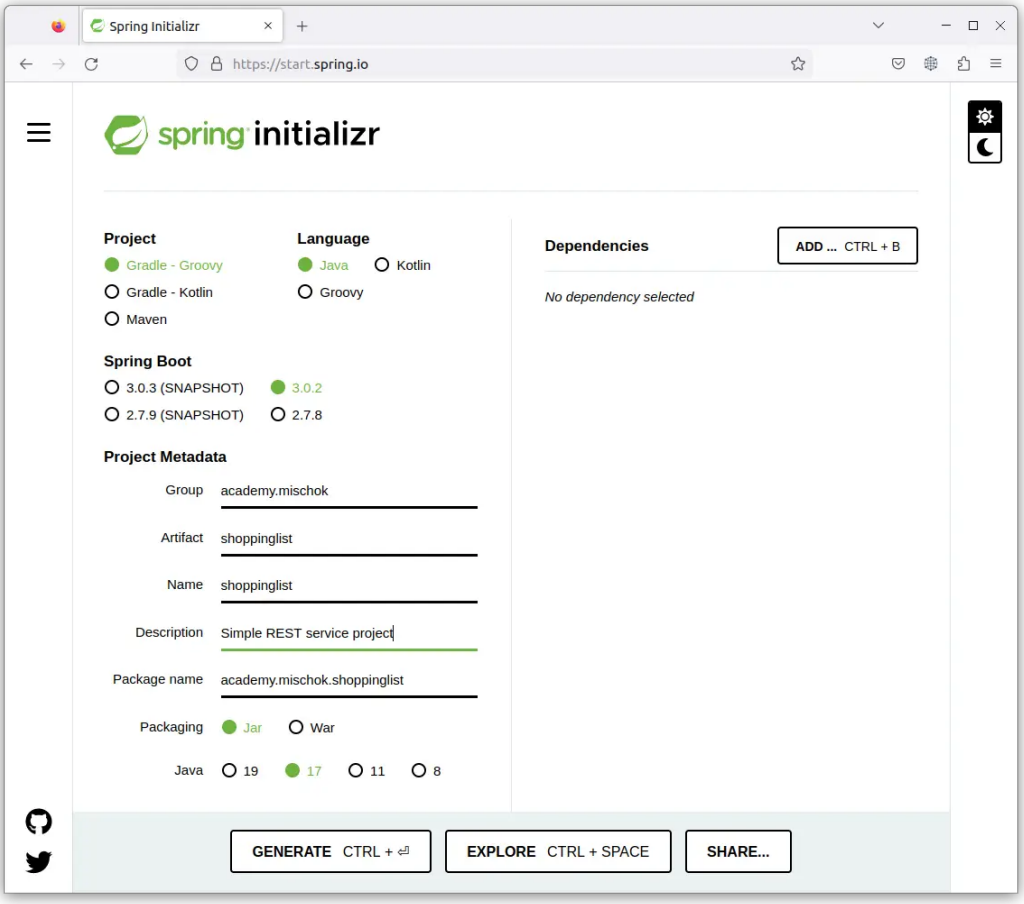


Mit Spring Boot lassen sich Spring-basierte Java-Anwendungen ohne viel Konfigurationsaufwand erstellen. Wir zeigen den Aufbau und die Funktion einer Anwendung.

MEHR ZU SPRING UND SPRING-BOOT

- Spring Boot: Aufbau und Funktion einer Anwendung am Beispiel eines REST-Service [1]
- E-Health: Digitale Klientenakte mit Java und Spring [2]
- Spring Framework 6 im Überblick: Major-Version setzt Java 17 voraus [3]
- GraphQL-APIs mit Spring bauen [4]
- Webentwicklung: HTTP-Antworten mit Kotlin und Spring programmieren [5]

Das Java-Framework Spring Boot eignet sich für das Erstellen von Webanwendungen und Microservices. Es beeindruckt durch ein Minimum an notwendiger Konfiguration, um eine Anwendung initial lauffähig zu machen. Ein sinnvoller Startpunkt ist der **Spring Initializr [6]**. Die übersichtliche Website ermöglicht, mit wenigen Klicks ein leeres, vorkonfiguriertes Projekt zu erstellen.



Der Spring Initializr erleichtert die Konfiguration neuer Projekte.

JULIUS MISCHOK

(Bild: COPYRIGHT@MARTINBECK.DE)

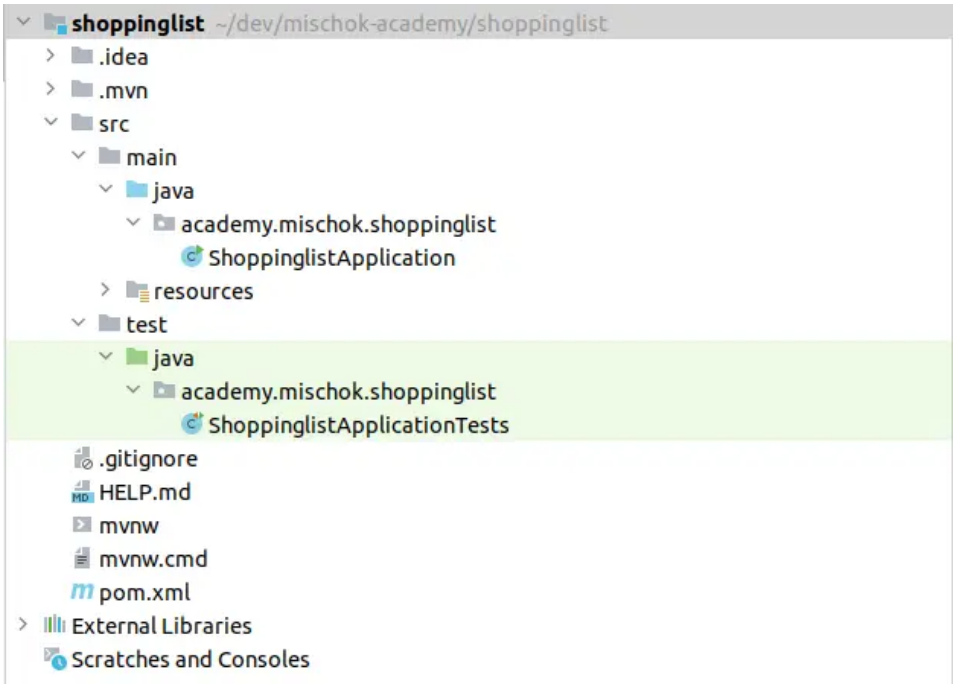
Julius Mischok ist Gründer der Mischok GmbH. Seine Erfahrung aus mehr als zwanzig Jahren Softwareentwicklung gibt er unter anderem als Dozent in der Mischok Academy weiter.

Nach der Auswahl des Build-Tools – im Fall des Beispielprojekts Maven – und der Eingabe der Metadaten erstellt ein Klick auf den Button GENERATE ein Archiv mit den Projektdaten **(Beispielprojekt) [7]**. Nach dem Herunterladen und Entpacken lässt sich das Projekt in einer beliebigen Entwicklungsumgebung weiter bearbeiten.

Spring Boot liefert das gewählte Build-Tool als Wrapper mit. Eine minimale Version des Tools liegt im versteckten Ordner .mvn, Skripte für Windows und Unix bieten Zugriff auf dessen Methoden. Was zunächst nach strukturellem Overhead aussieht, erweist sich in der Praxis als vorteilhaft: Bezieht man die entsprechenden Verzeichnisse und Dateien in die Versionsverwaltung ein, braucht es lediglich ein Java Development Kit (JDK), um die Anwendung zu kompilieren und zu starten. Zusätzliche Installationen und Konfigurationen auf lokalen Entwicklungsgeräten oder in CI/CD-Pipelines entfallen.

- Mit der Erweiterung Spring Boot des Spring-Frameworks kann man direkt Anwendungen entwickeln, ohne sich näher mit Spring beschäftigen zu müssen.
- REST-Services sind durch eine einfache Grundkonfiguration mit wenigen Codezeilen umsetzbar.
- Eine Beispielanwendung, die einen Einkaufszettel verwaltet und am Ende einen REST-Service bereitstellt, berücksichtigt verschiedene Testmethoden.

Neben dem Build-Tool legt der Spring Initializr die Maven-Projektstruktur mit den Hauptordnern src/main und src/test an. Unterhalb dieser beiden Hauptverzeichnisse befinden sich die eigentlichen Quellcodedateien und Ressourcen des Projekts. Die folgende Abbildung zeigt die initiale Projektstruktur: Im Hauptverzeichnis liegt die Datei pom.xml, die die wesentlichen Metainformationen des Projekts und die Konfiguration der Projektabhängigkeiten enthält.



Die Grundstruktur des Projekts legt der Spring Boot Initializr an.

Eine Spring-Boot-Anwendung starten

Um eine leere Spring-Boot-3-Anwendung zu starten, ist ein installiertes JDK der Version 17 oder höher erforderlich. **Die Vorgängerversion Spring Boot 2 berücksichtigt zwar Java 17, ist aber auch mit älteren Versionen lauffähig [8]** . Außerhalb einer Entwicklungsumgebung ist der Start über ein Terminal möglich. Der Befehl unterscheidet sich bei Unix- und Windows-Betriebssystemen:

```
./mvnw spring-boot:run
```

startet die Anwendung auf Linux und macOS; unter Windows ruft man den Maven-Wrapper folgendermaßen auf:

```
mvnw.cmd spring-boot:run
```

Der Befehl wird in dem Verzeichnis ausgeführt, in dem sich die Datei pom.xml befindet. Das Skript ruft ein spezielles Maven-Plug-in für Spring Boot auf, das zwei Schritte ausführt: Der erste kompiliert und packt das Projekt, der zweite Schritt startet die Anwendung. Sofern erfolgreich, begrüßt Spring Boot mit ASCII-Arts, fährt die Anwendung allerdings direkt nach dem Start wieder herunter.

Der relevante Quellcode der Anwendung befindet sich im Initialzustand in der Datei ShoppinglistApplication.java. Die Applikation ruft eine statische Hilfsklasse des Frameworks auf und reicht ihr die Kommandozeilenparameter durch.

Listing 1: Die Datei ShoppingListApplication.java startet die Anwendung

```
@SpringBootApplication
public class ShoppinglistApplication {

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


[9]

Die Anwendung um eine Web Dependency erweitern

Zur Zeit der Veröffentlichung der ersten Version von Spring Boot wurde der Softwarearchitekturansatz der Microservices zunehmend populär. Hatte man zuvor meist mehrere Anwendungen auf der gleichen Installation eines Web- oder Application-Servers betrieben, brachte Spring Boot den eigenen Webserver direkt im ausgelieferten Paket mit. Spring-Boot-Anwendungen setzen als Ausführungsumgebung bis heute lediglich ein Java-Runtime-Environment voraus. Die zusätzliche Installation und Konfiguration eines Webservers entfällt.

Die hier entwickelte Beispielanwendung zum Verwalten eines Einkaufszettels stellt am Ende einen einfachen REST-Service zur Verfügung. Um die dafür notwendigen Bibliotheken in das Projekt zu integrieren, erweitert man die Datei pom.xml um eine Dependency namens `spring-boot-starter-web`, die sich in der XML-Konfiguration innerhalb des Tags `<dependencies>` befindet.

Listing 2: Datei pom.xml mit Dependency

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    ...
</dependencies>
```

Alle beim Erstellen des Projekts mit dem Spring Initializr bekannten Abhängigkeiten lassen sich im Webformular komfortabel suchen und hinzufügen. Die generierte Datei pom.xml enthält dann die entsprechenden Dependencies.

Auffällig an der Konfiguration der hinzugefügten Abhängigkeit ist, dass keine explizite Versionsnummer angegeben ist. Wie erwähnt, trat Spring Boot mit dem Versprechen an, das Entwickeln von Spring-Anwendungen zu vereinfachen. Bei genauerer Analyse von pom.xml fällt hierbei das `<parent>`-Tag ins Auge.

Listing 3: Parent Dependency für Spring Boot einbinden

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Die auf den ersten Blick unscheinbare Konfiguration gewährleistet die nahtlose Integration der verschiedenen Spring Boot Starter. In den Starter Dependencies stellt das Framework kompatible Konfigurationen von Abhängigkeiten zusammen. Solange explizite Versionsnummern die Abhängigkeiten nicht überschreiben, sind alle Starter in sich und untereinander kompatibel.

Ist die Web Dependency hinzugefügt, zeigt sich beim Starten der Anwendung ein anderes Bild: Die Spring-Applikation terminiert nicht mehr sofort, sondern geht in einen Wartemodus. Das liegt daran, dass `spring-boot-starter-web` einen Embedded Tomcat mitbringt und startet. Wie im Konsolenoutput nachvollziehbar, lauscht der Webserver auf Port 8080 auf eingehende Anfragen.

Listing 4: Webserver erwartet auf Port 8080 Anfragen

```
2023-02-15T10:13:36.492+01:00  INFO 18426 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
```

Convention over Configuration als Grundlage

Das zugrunde liegende Prinzip nennt sich Convention over Configuration. Statt Entwicklerinnen und Entwickler zu zwingen, eine Standardkonfiguration in das Projekt zu kopieren, bringt der Starter diese bereits mit. Dazu zählt die Konfiguration des Ports für Tomcat.

Unklar ist jedoch, wodurch Tomcat als Webserver im Projekt aktiviert wurde. Spring Boot führt grundsätzlich alle auf dem Classpath liegenden Bibliotheken beim Start der Applikation aus. In der Beispielanwendung erkennt der Component Scan, dass sowohl die Bibliotheken für den Embedded Tomcat als auch eine vollständige Konfiguration verfügbar sind.

Abgesehen davon, dass sich weitere Komponenten auf diesem Weg mit minimalem Aufwand in das Projekt integrieren lassen, erscheint das Verhalten zeitweise wie Magie. Insbesondere bei den ersten Kontakten mit Spring Boot empfiehlt sich ein genauer Blick in das Logfile und in die Konsolenausgabe, um nachvollziehen zu können, welche Schritte das Framework im Hintergrund ausführt. Per Default ist der Loglevel von Spring Boot auf `INFO` eingestellt. Für die Analyse hilft es, den Loglevel für das Spring Framework in der Datei src/main/resources/application.properties auf `DEBUG` zu stellen:

```
logging.level.org.springframework = DEBUG
```

Konfigurationszeilen in den Schemata `logging.level.PACKAGE` und `logging.level.PACKAGE.CLASS` stellen den Loglevel für bestimmte Packages und Klassen explizit ein. Da Packages kaskadieren, betrifft die Einstellung

explizit auch alle Unterpackages, sofern keine spezifischere Konfiguration vorliegt.

Anwendung testgetrieben entwickeln

Einer der Gründe für die starke Verbreitung von Spring Boot ist die hervorragende Unterstützung für Unit-Tests. Dadurch lassen sich auch große Anwendungen streng testgetrieben entwickeln, um eine hohe Qualität sicherzustellen. Die Testklassen liegen unterhalb des Pfads src/test/java, die für Tests benötigte Ressourcen unter src/test/resources.

Ein bewährtes Vorgehen beim Entwickeln von REST- oder allgemeineren HTTP-Services ist, vor der Implementierung der Logik einen zunächst fehlschlagenden Test zu formulieren. Dieses Prinzip nennt sich Test-driven Development. Eine einfache Testklasse zeigt Listing 5.

Listing 5: Testmethode für Einkaufsliste

```
@SpringBootTest
@AutoConfigureMockMvc
public class EntriesControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testEntriesGetIsAvailable() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/shoppinglist/entries"))
               .andExpect(MockMvcResultMatchers.status().isOk());
    }
}
```

Entweder führt ein Plug-in direkt in der Entwicklungsumgebung den Test aus oder der Maven-Befehl `test`. Das Maven Goal `test` führt alle Unit-Tests des Projekts aus, der Befehl für Unix-Betriebssysteme lautet:

```
./mvnw test
```

Für Windows-Systeme lautet der Befehl

```
mvnw.cmd test
```

Unabhängig von der Art der Ausführung schlägt der Test fehl, was beim Test-driven Development die korrekte Ausgangslage ist. Erst danach startet die tatsächliche Implementierung der Funktion. Andernfalls besteht die Gefahr, dass der formulierte Test die zu entwickelnde Funktionalität gar nicht testet.

Wie bei Spring Boot üblich, lohnt sich trotz des Fehlschlags ein Blick in das Logfile. Es fällt auf, dass die Anwendung komplett gestartet wird, also die gleichen Meldungen wie beim manuellen Start von der Kommandozeile im Logfile erscheinen. Spring Boot fährt den gesamten Spring-Kontext hoch und startet Tomcat auf einem zufälligen Port. Durch die automatische Konfiguration von `MockMvc` ruft der Test diesen zufälligen Port dann korrekt auf.

Die Tatsache, dass Spring Boot komplett hochgefahren wird, macht den Testfall bei strenger Auslegung der Definition mehr zu einem Integrations- als zu einem Unit-Test. Es steht nicht eine Komponente isoliert unter Test, sondern das Zusammenspiel mehrerer Komponenten – im weiteren Verlauf des Tutorials auch mit externen Abhängigkeiten wie einer Datenbank.

Der Test aus Listing 5 ist noch rudimentär: Er prüft lediglich, ob der zu implementierende Endpunkt den korrekten HTTP-Status zurückgibt. Da noch kein Produktionscode besteht, findet der Test unter der Adresse

/shoppinglist/entries nichts und der Server antwortet folglich mit dem HTTP-Status "404 – Not found".

Test erweitern und über Controller verbessern

Listing 6 zeigt die minimale Umsetzung eines Controllers. Diese – noch nicht funktionsfähige – Implementierung sorgt aber dafür, dass der Test ohne Fehler durchläuft. Die Implementierung definiert das Mapping des vom Test angefragten URI und gibt eine vollständige, wenn auch leere HTTP-Antwort zurück.

Listing 6: Basisimplementierung des Controllers

```
@RestController
public class EntriesController {

    @GetMapping("/shoppinglist/entries")
    public ResponseEntity<?> getEntries() {
        return ResponseEntity.ok("");
    }
}
```

Spring Boot steuert viele Konfigurationen über Annotationen an Java-Klassen. Im Fall des `EntriesController` sind das `@RestController` und `@GetMapping`. `@RestController` bewirkt, dass die Klasse als Bean im Spring-Kontext registriert und behandelt wird. Beans sind Objekte, die der Spring-Kontext an anderer Stelle – vor allem in anderen Beans – injiziert. Im Fall des REST-Controllers scannt Spring die Bean weiter und wertet die Annotation `@GetMapping` aus, die eine Java-Methode mit einem URI verbindet und dann eingehende Anfragen an ihn delegiert.

Der Endpunkt liefert derzeit lediglich eine HTTP Response mit dem Status "200 – OK" und leerem Response-Body. Eine Erweiterung des bestehenden Tests stellt sicher, dass die Payload korrekt übermittelt wird. Um die Nachvollziehbarkeit im Beispielprojekt zu erhalten, enthält die Testklasse für den angepassten Code eine neue Testmethode.

Listing 7: Test für die JSON Payload

```
@Test
public void testEntriesGetIsAvailableAndReturnsData() throws Exception {
    mockMvc.perform(get("/shoppinglist/entries"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$").isArray())
            .andExpect(jsonPath("$", hasSize(2)))

            .andExpect(jsonPath("$[0].title", is("Butter")))
            .andExpect(jsonPath("$[0].amount", is(2)))
            .andExpect(jsonPath("$[0].category", is("dairy")))

            .andExpect(jsonPath("$[1].title", is("Potatoes")))
            .andExpect(jsonPath("$[1].amount", is(5)))
            .andExpect(jsonPath("$[1].category", is("vegetables")));
}
```

Um den Testcode lesbar zu halten, haben sich statische Importe in Java bewährt. Fügt man die Zeilen aus Listing 8 hinzu, muss man beispielsweise die Methode `MockMvcRequestBuilders.get` nicht mehr qualifizieren. Die gängigen Entwicklungsumgebungen helfen, die Imports zu verwalten und auf statische Imports zu migrieren.

Listing 8: Statische Imports

```
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.Matchers.hasSize;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
```

Tests mit der Bibliothek Hamcrest auswerten

Der neue Testfall vereinfacht nicht nur den Testcode, sondern prüft auch die fachliche Korrektheit der Implementierung. Das betrifft neben dem HTTP-Statuscode den Inhalt des Response-Bodys. Dabei kommt die Abfragesprache **JsonPath [10]** zum Einsatz, um Abfragen gegen JSON-Dokumente zu formulieren und auszuwerten. Spring Boot 3 wertet die Abfrageergebnisse mit Hamcrest Matchers aus und verifiziert die Testannahmen.

Im hier verwendeten Beispiel stellt der Test sicher, dass der Endpunkt ein JSON-Array mit genau zwei Elementen zurückgibt. Zusätzlich gleicht er die Eigenschaften beider Elemente ab. Diese relativ elementaren Operationen funktionieren intuitiv und bieten mit einem Minimum an Code eine hohe Sicherheit.

Test-driven Development sieht vor, Test- und Produktionscode im Wechsel zu bearbeiten. Der Produktionscode wird so gestaltet, dass alle bestehenden Tests grün sind, also fehlerfrei durchlaufen. Im Beispiel des Einkaufszettels genügt der Code aus Listing 9 im Controller, um den Test erfolgreich zu durchlaufen.

Listing 9: Rückgabe von Daten im Response-Body

```
@GetMapping("/shoppinglist/entries")
public ResponseEntity<List<ShoppinglistEntryDto>> getEntries() {

    List<ShoppinglistEntryDto> result = List.of(
        new ShoppinglistEntryDto("Butter", 2, "dairy"),
        new ShoppinglistEntryDto("Potatoes", 5, "vegetables")
    );

    return ResponseEntity.ok(result);
}
```

Im Vergleich zur Initialversion in Listing 6 fallen zwei wesentliche Veränderungen ins Auge: Zum einen ist die `ResponseEntity` als Rückgabewert nun mit `List<ShoppinglistEntryDto>` typisiert, zum anderen werden zwei Instanzen dieser Klasse per Konstruktor erzeugt und in einer Liste zurückgegeben. Streng genommen handelt es sich bei `ShoppinglistEntryDto` nicht um eine Klasse, sondern um das seit Java 14 bestehende Sprachkonstrukt Records. Ein Record verhält sich wie eine finale, also nicht vererbbare Klasse. Sie besitzt ausschließlich finale Eigenschaften. Der folgende Code verdeutlicht, wie kurz die Syntax ist:

```
public record ShoppinglistEntryDto(String title,  int amount, String category) {
}
```

Java 17 ist die erste LTS-Version (Long-Term-Support), die Records berücksichtigt, und gleichzeitig die niedrigste von Spring Boot 3 unterstützte Java-Version. Die teilweise zu Recht als Boilerplate-lastig kritisierte Sprache Java macht damit einen großen Schritt in Richtung einer kompakteren und moderneren Syntax. Das Suffix `Dto` im Klassennamen steht für Data Transfer Object und ist eine gängige Abkürzung für Hilfsklassen, die Datenstrukturen für Schnittstellen in Services definieren.

Einkaufsliste um weitere Punkte erweitern

Das Beispielprojekt soll nicht nur statische Werte zurückliefern, sondern auch den Einkaufszettel um weitere Punkte ergänzen können. Zur Abdeckung dieses Anwendungsfalls erhält das Projekt einen zusätzlichen Testfall.

Der Code in Listing 10 legt mittels HTTP POST einen neuen Eintrag in der Einkaufsliste an und verifiziert, dass dieser Eintrag danach auch in der Liste erscheint.

Listing 10: Test für das Anlegen neuer Einträge

```
@Test
public void testAddEntry() throws Exception {
    mockMvc.perform(get("/shoppinglist/entries"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", hasSize(2)));

    String payload = """
        {
            "title": "Tomatoes",
            "amount": 4,
            "category": "vegetables"
        }
         """;

    mockMvc.perform(post("/shoppinglist/entries")
            .content(payload)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isCreated());

    mockMvc.perform(get("/shoppinglist/entries"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", hasSize(3)))

            .andExpect(jsonPath("$[2].title", is("Tomatoes")))
            .andExpect(jsonPath("$[2].amount", is(4)))
            .andExpect(jsonPath("$[2].category", is("vegetables")));
}
```

Interessant ist die auf den ersten Blick ungewohnte Definition eines Strings für die Variable `payload`. Die drei Anführungszeichen leiten einen Text Block ein – ein mit Java 15 initial veröffentlichtes Feature. Wie bei den Records ist Java 17 die erste LTS-Version mit Text Blocks. Sie ersparen das Escapen doppelter Anführungszeichen in Strings und erleichtern das Formatieren längerer Texte erheblich.

Um die Daten zwischen HTTP Requests zu persistieren, empfiehlt sich eine Datenbank. Die Einbindung erfolgt in zwei Schritten: Datenbank in Spring Boot konfigurieren und mit der Datenbank via Jakarta Persistence API (JPA) kommunizieren.

Statt einer lokalen Datenbank verwendet man H2. ➕ **Das Datenbanksystem ist vollständig in Java geschrieben und zur Integration genügt es, eine Dependency hinzuzufügen [11]**. Insbesondere berücksichtigt H2 weitestgehend den SQL-Standard. Daher lässt sich in einer etwaigen Produktivumgebung problemlos ein anderes, leistungsfähigeres Datenbanksystem einsetzen. Zudem ermöglicht H2 den Betrieb im Speicher. Das ist vor allem für Unit-Tests attraktiv, da die Datenbank bei jedem Start des Projekts einen definierten Initialzustand aufweist.

Die benötigten Abhängigkeiten konfiguriert man in der Datei pom.xml. Die erste Abhängigkeit bringt die Bibliotheken für JPA und die Integration in Spring Boot mit. JPA ist die unabhängige Spezifikation für ein ORM-Framework (Object-Relational Mapping). Spring Data JPA integriert diese Spezifikation mit dem Framework Hibernate als Implementierung in das Spring-Boot-Projekt. Ziel von ORM ist die Abstraktion der Datenbankkommunikation: Der ORM übernimmt das Parsing zwischen Datensätzen und Objekten. Eine objektorientierte Programmiersprache wie Java arbeitet dann ausschließlich mit den Objektinstanzen.

Listing 11: Abhängigkeiten für Datenbankanbindung

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

Ist nur die Abhängigkeit für Spring Data JPA konfiguriert, schlägt der nächste Projektstart fehl: Dem Framework fehlt die Konfiguration der URL, unter der die zu verwendende Datenbank erreichbar ist. Die zweite Abhängigkeit integriert das Datenbanksystem H2 in das Projekt. Beim erneuten Projektstart signalisiert Spring Boot sofort, dass es eine Datenbankkonfiguration gefunden hat.

Listing 12: Spring Boot meldet eine Datenbankkonfiguration

```
2023-02-25T08:58:34.384+01:00  INFO 12319 --- [           main]
com.zaxxer.hikari.HikariDataSource     : Hikari-Pool-1 - Starting...
2023-02-25T08:58:34.464+01:00  INFO 12319 --- [           main]
com.zaxxer.hikari.pool.HikariPool        : Hikari-Pool-1 - Added connection conn0:
url=jdbc:h2:mem:ea4056f5-5e89-4811-b23e-f4763cdde44e user=SA
2023-02-25T08:58:34.464+01:00  INFO 12319 --- [           main]
com.zaxxer.hikari.HikariDataSource     : Hikari-Pool-1 - Start completed.
```

Relevant ist vor allem die zweite Meldung: Durch die Defaultkonfiguration im Spring Data JPA Starter startet H2 im In-Memory-Modus, zu erkennen am Einschub `:mem:` in der Datenbank-URL. Die Datenbank erhält eine zufällige UUID, die sich bei jedem Start ändert. Da die Datenbank in derselben Java Virtual Machine wie die Applikation läuft, muss man keine expliziten Credentials konfigurieren.

(nb [12])

URL dieses Artikels:
https://www.heise.de/-8515704

Links in diesem Artikel:
[1] https://www.heise.de/ratgeber/Spring-Boot-Aufbau-und-Funktion-einer-Anwendung-am-Beispiel-eines-REST-Service-8515704.html
[2] https://www.heise.de/hintergrund/E-Health-Digitale-Klientenakte-mit-Java-und-Spring-7394851.html
[3] https://www.heise.de/tests/Spring-Framework-6-im-Ueberblick-Major-Version-setzt-Java-17-voraus-7351343.html
[4] https://www.heise.de/ratgeber/GraphQL-APIs-mit-Spring-bauen-7329990.html
[5] https://www.heise.de/ratgeber/Webentwicklung-HTTP-Antworten-mit-Kotlin-und-Spring-programmieren-7060970.html
[6] https://start.spring.io/
[7] https://gitlab.mischok-it.de/open/shoppinglist
[8] https://www.heise.de/tests/Spring-Framework-6-im-Ueberblick-Major-Version-setzt-Java-17-voraus-7351343.html
[9] https://www.heise.de/ix/
[10] https://github.com/json-path/JsonPath
[11] https://www.heise.de/ratgeber/Spring-Boot-Teststrategien-mit-der-In-Memory-DB-H2-4641910.html
[12] mailto:nb@heise.de