End to End Spring Boot Microservice

**1. Overview**

Spring Boot is an opinionated addition to the Spring platform, focused on convention over configuration — highly useful for getting started with minimum effort and creating standalone, production-grade applications.

**This the final project for SpringBoot Microservices**, in other words, a way to get started in a simple manner with a basic web application.

We'll go over some core configuration, a front-end, quick data manipulation, and exception handling.

**2. Setup**

First, let's use [Spring Initializr](https://start.spring.io/) to generate the base for our project.

The generated project relies on the Boot parent:

<**parent**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-parent</**artifactId**>

<**relativePath** />

</**parent**>

The initial dependencies are going to be quite simple:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-web</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-data-jpa</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>com.h2database</**groupId**>

<**artifactId**>h2</**artifactId**>

</**dependency**>

**3. Application Configuration**

Next, we'll configure a simple *main* class for our application:

@SpringBootApplication

**public** **class** **Application** {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Notice how **we're using *@SpringBootApplication*as our primary application configuration class.** Behind the scenes, that's equivalent to *@Configuration*, *@EnableAutoConfiguration,* and *@ComponentScan* together.

Finally, we'll define a simple *application.properties*file, which for now only has one property:

server.port=8081

*server.port* changes the server port from the default 8080 to 8081; there are of course many more [Spring Boot properties available](https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html).

**4. Simple MVC View**

Let's now add a simple front end using Thymeleaf.

First, we need to add the *spring-boot-starter-thymeleaf* dependency to our *pom.xml*:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

That enables Thymeleaf by default. No extra configuration is necessary.

We can now configure it in our *application.properties*:

spring.thymeleaf.cache=false

spring.thymeleaf.enabled=true

spring.thymeleaf.prefix=classpath:/templates/

spring.thymeleaf.suffix=.html

spring.application.name=Bootstrap Spring Boot

Next, we'll define a simple [controller](https://www.baeldung.com/spring-controllers) and a basic home page with a welcome message:

@Controller

**public** **class** **SimpleController** {

@Value("${spring.application.name}")

String appName;

@GetMapping("/")

**public** String **homePage**(Model model) {

model.addAttribute("appName", appName);

**return** "home";

}

}

Finally, here is our *home.html*:

<**html**>

<**head**><**title**>Home Page</**title**></**head**>

<**body**>

<**h1**>Hello !</**h1**>

<**p**>Welcome to <**span** th:text="${appName}">Our App</**span**></**p**>

</**body**>

</**html**>

Note how we used a property we defined in our properties and then injected that so we can show it on our home page.

**5. Security**

Next, let's add security to our application by first including the security starter:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-security</**artifactId**>

</**dependency**>

By now, we can notice a pattern: **Most Spring libraries are easily imported into our project with the use of simple Boot starters.**

Once the *spring-boot-starter-security* dependency is on the classpath of the application, all endpoints are secured by default, using either *httpBasic* or *formLogin* based on Spring Security's content negotiation strategy.

That's why, if we have the starter on the classpath, we should usually define our own custom Security configuration:

@Configuration

@EnableWebSecurity

**public** **class** **SecurityConfig** {

@Bean

**public** SecurityFilterChain **filterChain**(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.anyRequest()

.permitAll()

.and()

.csrf()

.disable();

**return** http.build();

}

}

In our example, we're allowing unrestricted access to all endpoints.

Of course, Spring Security is an extensive topic and not easily covered in a couple of lines of configuration. So, we definitely encourage [deeper reading into the topic](https://www.baeldung.com/security-spring).

**6. Simple Persistence**

Let's start by defining our data model, a simple *Book* entity:

@Entity

**public** **class** **Book** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** **long** id;

@Column(nullable = false, unique = true)

**private** String title;

@Column(nullable = false)

**private** String author;

}

and its repository, making good use of Spring Data here:

**public** **interface** **BookRepository** **extends** **CrudRepository**<Book, Long> {

List<Book> **findByTitle**(String title);

}

Finally, we need to of course configure our new persistence layer:

@EnableJpaRepositories("com.baeldung.persistence.repo")

@EntityScan("com.baeldung.persistence.model")

@SpringBootApplication

**public** **class** **Application** {

...

}

Note that we're using the following:

* *@EnableJpaRepositories* to scan the specified package for repositories
* *@EntityScan*to pick up our JPA entities

To keep things simple, we're using an H2 in-memory database here. This is so that we don't have any external dependencies when we run the project.

Once we include H2 dependency, **Spring Boot auto-detects it and sets up our persistence** with no need for extra configuration, other than the data source properties:

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:bootapp;DB\_CLOSE\_DELAY=-1

spring.datasource.username=sa

spring.datasource.password=

Of course, like security, persistence is a broader topic than this basic set here and one to [certainly explore further](https://www.baeldung.com/persistence-with-spring-series).

**7. Web and the Controller**

Next, let's have a look at a web tier. And we'll start by setting up a simple controller, the *BookController*.

We'll implement basic CRUD operations exposing *Book* resources with some simple validation:

@RestController

@RequestMapping("/api/books")

**public** **class** **BookController** {

@Autowired

**private** BookRepository bookRepository;

@GetMapping

**public** Iterable **findAll**() {

**return** bookRepository.findAll();

}

@GetMapping("/title/{bookTitle}")

**public** List **findByTitle**(@PathVariable String bookTitle) {

**return** bookRepository.findByTitle(bookTitle);

}

@GetMapping("/{id}")

**public** Book **findOne**(@PathVariable Long id) {

**return** bookRepository.findById(id)

.orElseThrow(BookNotFoundException::**new**);

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

**public** Book **create**(@RequestBody Book book) {

**return** bookRepository.save(book);

}

@DeleteMapping("/{id}")

**public** **void** **delete**(@PathVariable Long id) {

bookRepository.findById(id)

.orElseThrow(BookNotFoundException::**new**);

bookRepository.deleteById(id);

}

@PutMapping("/{id}")

**public** Book **updateBook**(@RequestBody Book book, @PathVariable Long id) {

**if** (book.getId() != id) {

**throw** **new** **BookIdMismatchException**();

}

bookRepository.findById(id)

.orElseThrow(BookNotFoundException::**new**);

**return** bookRepository.save(book);

}

}

Given this aspect of the application is an API, we made use of the @*RestController* annotation here — which is equivalent to a *@Controller* along with *@ResponseBody* — so that each method marshals the returned resource right to the HTTP response.

Note that we're exposing our *Book* entity as our external resource here. That's fine for this simple application, but in a real-world application, we'll probably want to [separate these two concepts](https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application).

**8. Error Handling**

Now that the core application is ready to go, let's focus on **a simple centralized error handling mechanism** using *@ControllerAdvice*:

@ControllerAdvice

**public** **class** **RestExceptionHandler** **extends** **ResponseEntityExceptionHandler** {

@ExceptionHandler({ BookNotFoundException.class })

**protected** ResponseEntity<Object> **handleNotFound**(

Exception ex, WebRequest request) {

**return** handleExceptionInternal(ex, "Book not found",

**new** **HttpHeaders**(), HttpStatus.NOT\_FOUND, request);

}

@ExceptionHandler({ BookIdMismatchException.class,

ConstraintViolationException.class,

DataIntegrityViolationException.class })

**public** ResponseEntity<Object> **handleBadRequest**(

Exception ex, WebRequest request) {

**return** handleExceptionInternal(ex, ex.getLocalizedMessage(),

**new** **HttpHeaders**(), HttpStatus.BAD\_REQUEST, request);

}

}

Beyond the standard exceptions we're handling here, we're also using a custom exception, *BookNotFoundException*:

**public** **class** **BookNotFoundException** **extends** **RuntimeException** {

**public** **BookNotFoundException**(String message, Throwable cause) {

super(message, cause);

}

// ...

}

This gives us an idea of what's possible with this global exception handling mechanism. To see a full implementation, have a look at [the in-depth tutorial](https://www.baeldung.com/exception-handling-for-rest-with-spring).

Note that Spring Boot also provides an */error* mapping by default. We can customize its view by creating a simple *error.html*:

<**html** lang="en">

<**head**><**title**>Error Occurred</**title**></**head**>

<**body**>

<**h1**>Error Occurred!</**h1**>

<**b**>[<**span** th:text="${status}">status</**span**>]

<**span** th:text="${error}">error</**span**>

</**b**>

<**p** th:text="${message}">message</**p**>

</**body**>

</**html**>

Like most other aspects in Boot, we can control that with a simple property:

server.error.path=/error2

**9. Testing**

Finally, let's test our new Books API.

We can make use of [*@SpringBootTest*](https://www.baeldung.com/spring-boot-testing) to load the application context and verify that there are no errors when running the app:

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **SpringContextTest** {

@Test

**public** **void** **contextLoads**() {

}

}

Next, let's add a JUnit test that verifies the calls to the API we've written, using [REST Assured](https://www.baeldung.com/rest-assured-tutorial).

First, we'll add the [*rest-assured*](https://mvnrepository.com/artifact/io.rest-assured/rest-assured) dependency:

<**dependency**>

<**groupId**>io.rest-assured</**groupId**>

<**artifactId**>rest-assured</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

And now we can add the test:

**public** **class** **SpringBootBootstrapLiveTest** {

**private** **static** **final** **String** API\_ROOT

= "http://localhost:8081/api/books";

**private** Book **createRandomBook**() {

**Book** book = **new** **Book**();

book.setTitle(randomAlphabetic(10));

book.setAuthor(randomAlphabetic(15));

**return** book;

}

**private** String **createBookAsUri**(Book book) {

**Response** response = RestAssured.given()

.contentType(MediaType.APPLICATION\_JSON\_VALUE)

.body(book)

.post(API\_ROOT);

**return** API\_ROOT + "/" + response.jsonPath().get("id");

}

}

First, we can try to find books using variant methods:

@Test

**public** **void** **whenGetAllBooks\_thenOK**() {

**Response** response = RestAssured.get(API\_ROOT);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

}

@Test

**public** **void** **whenGetBooksByTitle\_thenOK**() {

**Book** book = createRandomBook();

createBookAsUri(book);

**Response** response = RestAssured.get(

API\_ROOT + "/title/" + book.getTitle());

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

assertTrue(response.as(List.class)

.size() > 0);

}

@Test

**public** **void** **whenGetCreatedBookById\_thenOK**() {

**Book** book = createRandomBook();

**String** location = createBookAsUri(book);

**Response** response = RestAssured.get(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

assertEquals(book.getTitle(), response.jsonPath()

.get("title"));

}

@Test

**public** **void** **whenGetNotExistBookById\_thenNotFound**() {

**Response** response = RestAssured.get(API\_ROOT + "/" + randomNumeric(4));

assertEquals(HttpStatus.NOT\_FOUND.value(), response.getStatusCode());

}

Next, we'll test creating a new book:

@Test

**public** **void** **whenCreateNewBook\_thenCreated**() {

**Book** book = createRandomBook();

**Response** response = RestAssured.given()

.contentType(MediaType.APPLICATION\_JSON\_VALUE)

.body(book)

.post(API\_ROOT);

assertEquals(HttpStatus.CREATED.value(), response.getStatusCode());

}

@Test

**public** **void** **whenInvalidBook\_thenError**() {

**Book** book = createRandomBook();

book.setAuthor(null);

**Response** response = RestAssured.given()

.contentType(MediaType.APPLICATION\_JSON\_VALUE)

.body(book)

.post(API\_ROOT);

assertEquals(HttpStatus.BAD\_REQUEST.value(), response.getStatusCode());

}

Then we'll update an existing book:

@Test

**public** **void** **whenUpdateCreatedBook\_thenUpdated**() {

**Book** book = createRandomBook();

**String** location = createBookAsUri(book);

book.setId(Long.parseLong(location.split("api/books/")[1]));

book.setAuthor("newAuthor");

**Response** response = RestAssured.given()

.contentType(MediaType.APPLICATION\_JSON\_VALUE)

.body(book)

.put(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

response = RestAssured.get(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

assertEquals("newAuthor", response.jsonPath()

.get("author"));

}

And we can delete a book:

@Test

**public** **void** **whenDeleteCreatedBook\_thenOk**() {

**Book** book = createRandomBook();

**String** location = createBookAsUri(book);

**Response** response = RestAssured.delete(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

response = RestAssured.get(location);

assertEquals(HttpStatus.NOT\_FOUND.value(), response.getStatusCode());

}

**10. Conclusion**

This was a quick but comprehensive introduction to Spring Boot.