

Testen van je code

Stel je voor: je werkt voor een bedrijf als backend developer. Het is zaterdagmiddag, je bent lekker aan het genieten van het zonnetje op de barbecue die je partner georganiseerd heeft voor je verjaardag. De hele tuin zit vol met familie en vrienden en je bent heerlijk aan het genieten van een koude versnapering, terwijl je gezellig zit te praten" kunnen vervangen door: "Maar nog voor je een hap kunt nemen van de aardappelsalade gaat plots je telefoon. Het is het bedrijf waar je voor werkt. De bedrijfswebsite is eruit geklapt en als je je baan wilt behouden, moet jij het probleem nu gaan fixen. Bij het nalopen van de code blijkt een collega voor het weekend een aanpassing te hebben gedaan waardoor jouw code niet meer helemaal klopt. Dit heeft ervoor gezorgd dat de applicatie crasht als er een specifiek endpoint wordt opgevraagd. Je realiseert je dat je dit had kunnen voorkomen als je je code had getest met automatisch testen. Dan had je collega het probleem al gezien voor het live was gezet... Je raadt het al: dit moet je nu rechtzetten!

Het testen van je code is dus van groot belang. Maar naast het rustig kunnen barbecueën zijn er nog meer belangrijke redenen waarom je gebruik wilt maken van automatisch testen:

- Het vinden van incorrecte code;
- Het vinden van bugs in een proces;
- Het zo vroeg mogelijk opsporen van problemen;
- De kosten om problemen op te lossen zo laag mogelijk te houden;
- Garantie dat je code nog werkt na aanpassingen of uitbreidingen;
- Niet na iedere aanpassing alles handmatig opnieuw te hoeven testen;
- Niet in het weekend of op een vrije dag opgeroepen worden om de applicatie te fixen.

Als je gebruik maakt van automatisch testen, worden deze iedere keer gedraaid wanneer je je applicatie opnieuw opstart. Dit houdt in dat als een van deze testen niet slaagt je applicatie niet draaibaar is. Voor de eindopdracht gaan we gebruik maken van unittesten en integratietesten.

!!!Belangrijk!!!

We gebruiken voor het testen nooit de werkelijke database. Dit zou namelijk kunnen leiden tot database vervuiling of het per ongeluk verwijderen van data.

Laten we eens een kijkje nemen naar de verschillende soorten testen.

Wat zijn unittesten?

Unittesten zijn testen die een klein stukje (een *unit*) van je code testen. Een unittest is, simpel gezegd, de vergelijking tussen de *verwachte* uitkomst en de *daadwerkelijke* uitkomst van een stukje code. Testen doen we geïsoleerd dus we gaan niet het hele proces volgen om tot dit stukje code te komen. We gaan kijken welk stukje code we los zouden kunnen koppelen en zien wat dit stukje doet. De uitkomst hiervan kunnen we vergelijken met wat we daar zouden verwachten.

Doordat we geïsoleerd testen kunnen we controleren wat we meegeven. Hierdoor is het ook voorspelbaar wat er terug uit de test zou moeten komen. Als er iets in de code verandert zal dit ook de uitkomst veranderen. En zal de test falen als we deze aanpassing niet ook aanpassen in de test. Op die manier houden we controle over wat een functie doet. Want als de functie verandert, slaagt de test niet meer en gaan de alarmbellen af.

Stel, we hebben een bibliotheek-applicatie en willen alle boeken van J.K. Rowling opvragen. Voor de complete code van de bibliotheek-applicatie kijk je [hier](#).

In de bookController staat de volgende code:

```
@GetMapping
public List<BookDto> getAllBooks(@RequestParam(value = "title", required =
false) Optional<String> title, @RequestParam(value = "initials", required =
false) Optional<String> initials , @RequestParam(value = "name", required =
false) Optional<String> name){
    if(title.isEmpty() && name.isPresent() && initials.isPresent()){
        return bookService.getAllBooksByAuthor(initials.get(), name.get());
    }else if (title.isPresent() && name.isEmpty() && initials.isEmpty()){
        return bookService.getAllBooksByTitle(title.get());
    }else {
        return bookService.getAllBooks();
    }
}

}

}

}
```

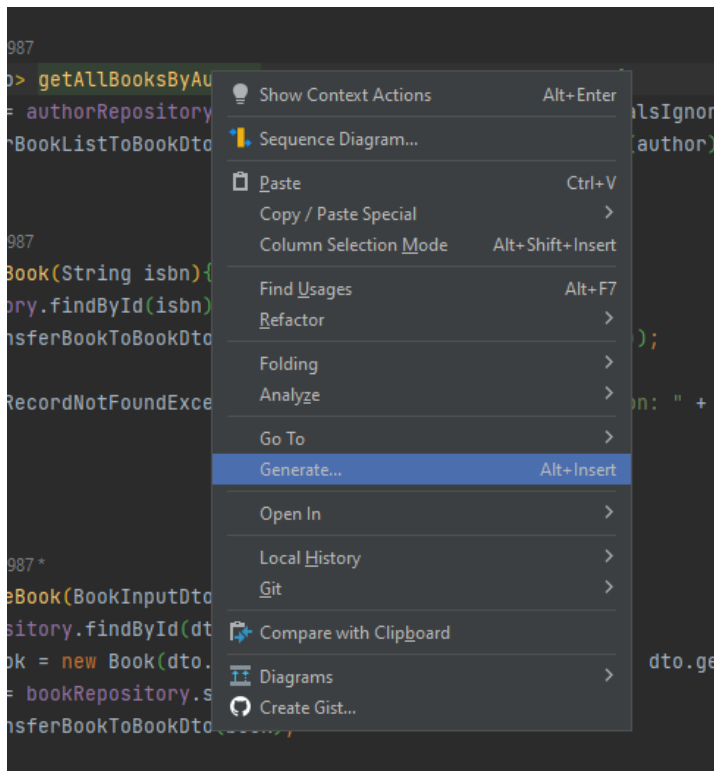
Voor het opzoeken van boeken van een specifieke auteur hebben we dus de bookService.getBooksByAuthor(name) methode nodig. Dit is de getAllBooksByAuthor methode in de bookService:

```
public List<BookDto> getAllBooksByAuthor(String initials, String name){
    Author author =
authorRepository.findAuthorByInitialsAndLastnameEqualsIgnoreCase(initials,
name);
    return
transferBookListToBookDtoList(bookRepository.findAllByAuthor(author));
}
```

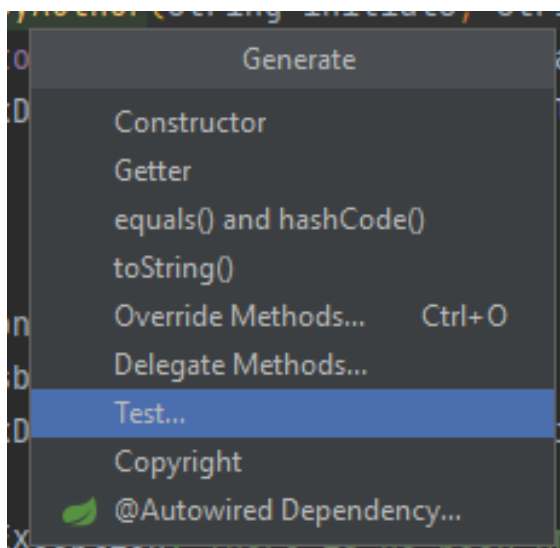
Dit is een uitstekend voorbeeld van een stukje code dat we kunnen testen met een unittest. We kunnen namelijk een test schrijven die deze methode aanroept met "J.K Rowling" als argument. We zouden dan kunnen testen of de boeken die we terugkrijgen, inderdaad geschreven zijn door J.K. Rowling. Laten we eens kijken hoe we deze test samenstellen.

Het opstellen van een unittest

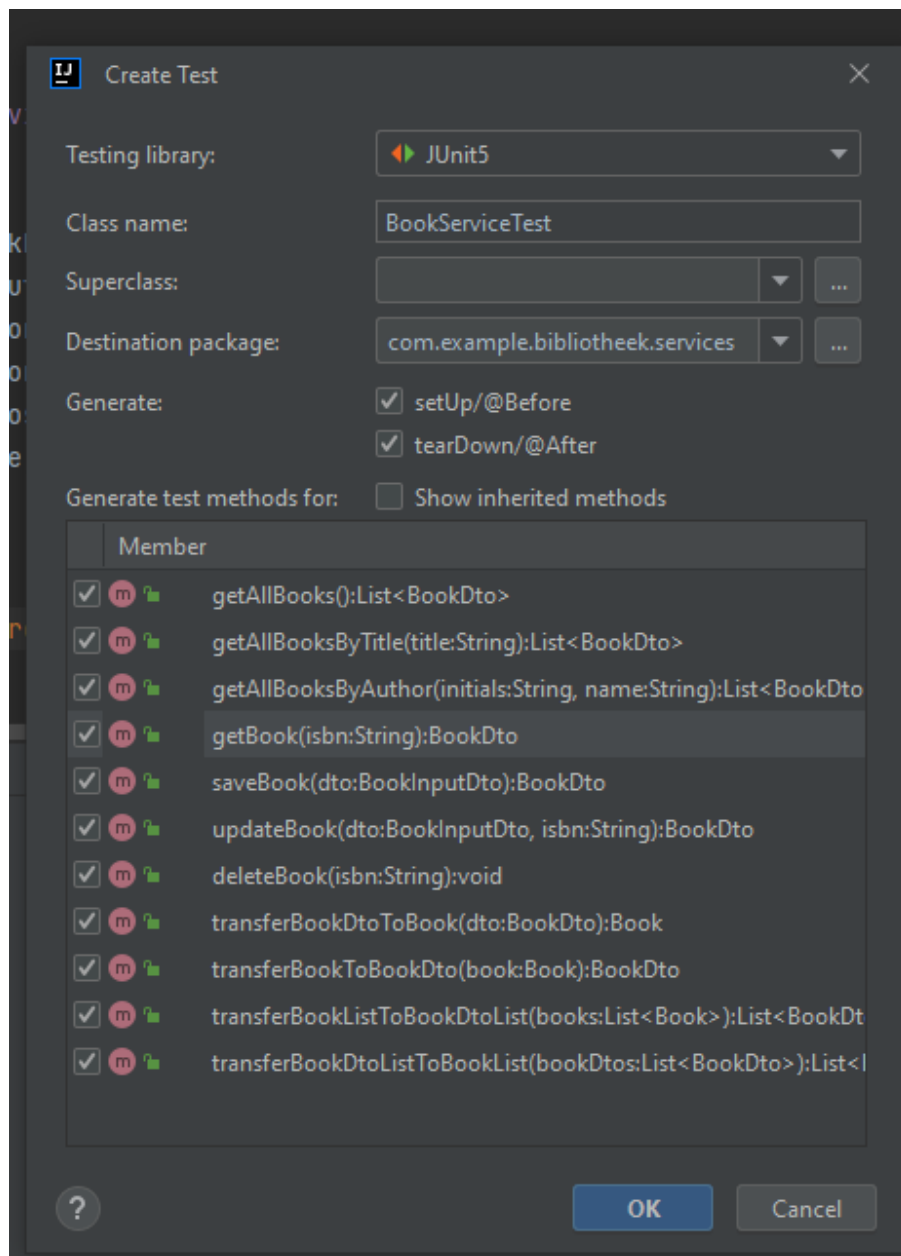
Klik met de rechtermuisknop op de naam van de getAllBooksByAuthor methode en kies voor Generate... > Test...



Kies hier voor Generate en krijg het volgende venster:



Kies hier voor Test. Dit opent het volgende venster:



In het Create Test-venster kan je kiezen welke library we gaan gebruiken, in dit geval is dit JUnit5. Dit is een libratie die het makkelijker maakt om frameworks testen te lanceren. Voor meer informatie over JUnit5 klik [hier](#).

JUnit 5 geeft ons onder meer de volgende mogelijkheden:

- `@Test` (Dit geeft ons de mogelijkheid om een methode te schrijven die door Spring Boot als test herkend wordt, zonder dat we allerlei dingen hiervoor in hoeven te stellen.)
- `@DisplayName` (Wat ons de mogelijkheid geeft om een uitgebreidere naam mee te geven aan de test, waardoor de echte naam kort en duidelijk kan zijn.)
- `@Disabled` (Waarmee we testen kunnen deactiveren, wat handig kan zijn als deze gegenereerd zijn, maar nog niet functioneel.)

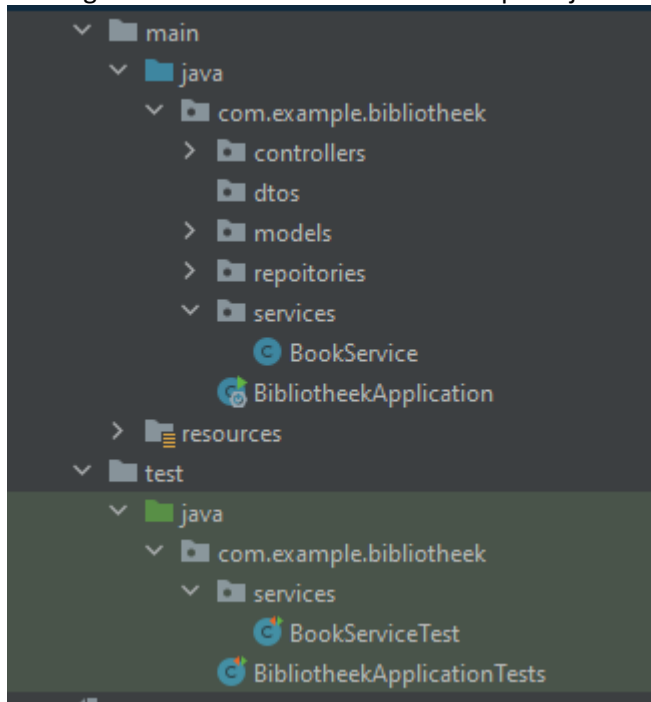
Bij generate zien we twee mogelijkheden staan die we kunnen aanvinken met de checkboxes: `setUp/@Before` en `tearDown/@After`

De eerste is een setUp methode die gebruikmaakt van de @BeforeEach annotatie. De tweede is een tearDown methode die gebruikmaakt van de @AfterEach annotatie. Deze opties bekijken we later wat beter.

Verder zien we dat we methodes die geschreven zijn, kunnen selecteren. Dit houdt in dat wanneer we deze checkboxes aanvinken, IntelliJ voor ons al de basis voor de test klaarzet.

Kies voor nu checkboxes van de setup, teardown en alle methodes en klik op de ok-knop.

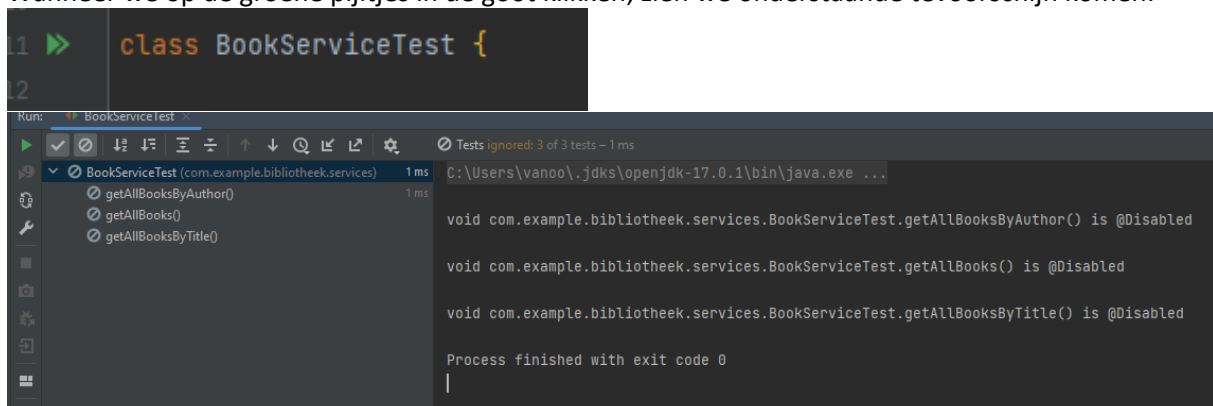
Vervolgens maakt IntelliJ een testklasse op het juiste niveau in je projectmap aan:



Het is belangrijk voor Spring Boot om te weten waar de testen te vinden zijn, anders ziet Spring Boot de testen niet en worden deze niet uitgevoerd bij het starten van de applicatie context.

Vervolgens opent een nieuw tabblad in het venster met de testklasse die zojuist is aangemaakt. Laten we beginnen met alle methodes waar de @Test annotatie boven staat, ook de @Disabled onder de @Test annotatie te zetten. Zo weet Spring Boot dat deze nog niet gedraaid hoeven te worden. Dit doen we omdat Spring Boot anders ook de lege testen gaat runnen. Alhoewel deze zullen slagen, neemt het extra tijd in beslag voor niks.

Wanneer we op de groene pijltjes in de goot klikken, zien we onderstaande tevoorschijn komen:



Hier zien we dat de testen slagen omdat de testen genegeerd worden. De applicatie is op dit moment dus gewoon draaibaar.

Zet boven je TestKlasse de volgende annotatie:

`@ExtendWith(MockitoExtension.class)` Dit zorgt voor allerlei technische settings waar we niet te veel van af hoeven te weten. Maar het is wel van belang dat we deze annotatie plaatsen, zodat Spring Boot weet dat we Mockito kunnen gebruiken bij het testen.

Setup methode met de `@BeforeEach` annotatie

We willen natuurlijk alle code zoveel mogelijk DRY (Don't Repeat Yourself) houden, dat geldt ook voor het testen. Als we bij elke test eerst een object van de klasse moeten aanmaken, zijn we onnodig veel code aan het herhalen. Met de setup methode kunnen we regelen dat Spring Boot dit voor ons doet vóór de test wordt uitgevoerd. Dit houdt onze codebase schoon en DRY. Deze methode wordt ook gebruikt in de bibliotheekapplicatie.

TearDown methode met de `@AfterEach` annotatie

Als we gebruikmaken van een setup methode die aangeroepen wordt voordat een test wordt uitgevoerd, kan dit natuurlijk problemen geven bij het runnen van de tweede test, want de data van de setup is al bekend. Door voor de tearDown methode te kiezen, wordt de data na het uitvoeren van iedere test direct weer verwijderd. Het is namelijk niet handig als we in de test die voor deze test komt de informatie aanpassen. Bij de tearDown methode maken we de aanpassingen in de data ongedaan.

Mocht je meer willen weten over deze twee methodes, neem dan een kijkje op:

[Bealdung Before](#) en [Bealdung Junit-5](#).

Mocking en testen

Zoals eerder gezegd, testen we met unittesten een stukje code in isolatie. Maar dat is wel wat makkelijker gezegd dan gedaan. Binnen Spring Boot werken we immers met verschillende lagen die met elkaar communiceren. Dit houdt in dat de lagen afhankelijk zijn van elkaar... Maar ook daar is een oplossing voor! We kunnen namelijk werken met Mockito. Mockito zorgt ervoor dat we een benodigde klasse kunnen “nadoen”, zonder de klasse echt te hoeven gebruiken. Daarnaast heeft Mockito nog veel meer verschillende mogelijkheden waar we gebruik van kunnen maken in onze testen. Meer informatie over Mockito vind je [hier](#).

Als we kijken naar de BookService zien we het volgende staan:

```
private final BookRepository bookRepository;
private final AuthorRepository authorRepository;

private final AuthorService authorService;

public BookService(BookRepository,
                  AuthorRepository authorRepository,
                  AuthorService authorService) {
    this.bookRepository = bookRepository;
    this.authorRepository = authorRepository;
    this.authorService = authorService;
}
```

Deze constructor (BookService) houdt in dat deze klasse afhankelijk is van de repositories BookRepository, de AuthorRepository en de AuthorService. Dat is niet zo handig, want deze repositories staan in verbinding met de database. Bij het schrijven van tests willen we nooit de echte database gebruiken.

In onze testklasse kunnen we deze twee repositories daarom *mocken*, door het volgende in de testklasse te zetten:

```
@Mock
BookRepository bookRepository;

@Mock
AuthorRepository authorRepository;

@Mock
AuthService authService;
```

Hierdoor worden niet de *echte* interfaces aangeroepen, maar de gemoekte versies. Daarnaast willen we natuurlijk de BookService testen, daarom gebruiken we hiervoor de volgende annotatie:

```
@InjectMocks
BookService bookService;
```

Dit mockt de klasse en injecteert de objecten die nodig zijn voor deze klasse. Deze zijn aangegeven zijn met een @Mock annotatie. Met andere woorden: de InjectMocks zorgt ervoor dat de service die we willen testen, wordt gemoekt en de repositories die zijn gemoekt met de @Mock annotatie worden vervolgens geïnjecteerd in deze service.

Als aanvulling kunnen we de @Captor gebruiken om een argumentCaptor te instantiëren. Dit doen we door het volgende te noteren in onze testklasse:

```
@Captor
ArgumentCaptor<Book> captor;
```

Dit geeft ons later de mogelijkheid om een save of update methode te testen. Hierbij willen we namelijk ook geen gebruik maken van de werkelijke database, maar we willen wel kunnen testen wat er dan gebeurt. Hierover leer je later meer over, maar voor nu is het genoeg om dit over te nemen.

Test samenstellen voor getAllBooksByAuthor

Als we unittesten opbouwen, doen we dit altijd aan de hand van de 3 A's. Waar stonden deze ook alweer voor?

Arrange, Act en Assert.

Deze worden ook wel eens gebruikt als Given, When en Then. De essentie blijft hetzelfde.

- Arrange:

Bij Arrange zetten we alle benodigdheden voor de test klaar.

- Act:

Bij Act gaan we daadwerkelijk iets uitvoeren.

- Assert:

Bij Assert gaan we checken of wat terugkomt, overeenkomt met wat we verwachten dat er uit de methode komt.

Voor de methode getAllBooksByAuthor hebben we het volgende nodig:

- String name;
- AuthorRepository authorRepository;
- BookRepository bookRepository;
- Book book1;
- Book book2;
- Book book3;
- Author author1;
- Author author2;
- BookService bookService.

We hebben de repositories en de service al gemoekt binnen de testklasse. Omdat we dit bovenaan in de klasse hebben gedaan, kunnen we deze door de hele klasse scope gebruiken.

De andere benodigdheden kunnen we in de testmethode instantiëren, onder het Arrange gedeelte.

```
Author author1 = new Author("J.K.", "Joanne Kathleen", "Rowling",  
    LocalDate.of(1965,7,31), Gender.FEMALE);  
Author author2 = new Author("R.", "Roald", "Dahl", LocalDate.of(1916,9,13),  
    Gender.MALE);  
Book book1 = new Book("9789076174105", "Harry Potter", "en de steen der  
wijzen", "fiction", "NL", "paperback", "uitgeverij de Harmonie", author1);  
Book book2 = new Book("9781781103470", "Harry Potter", "en de geheime  
kamer", "fiction", "NL", "paperback", "Pottermore publishing", author1);  
Book book3 = new Book("9789026139406", "Mathilda", "", "fiction", "NL",  
    "paperback", "de Fontein Jeugd", author2);
```

Omdat we straks in de Act iets willen uitvoeren met de repositories, moeten we ook vertellen wat we willen dat er gebeurt. De repositories zijn namelijk gemockt en kunnen niet hun werkelijke methodes uitvoeren. Dit doen we met de `when(...).thenReturn(...)` methode van Mockito *(hierbij staan op de puntjes normaal wel code, dit gaan we straks zien).

Dit doen we door:

```
when(authorRepository.findAuthorsByInitialsAndLastnameEquals("J.K.  
Rowling")).thenReturn(author1);  
when(bookRepository.findAllByAuthor(author1)).thenReturn(List.of(book1,book  
2));
```

Deze twee repositories komen vanuit de code van de servicemethode `getAllBooksByAuthor`, in deze service gebruiken we namelijk de `authorRepository.findAuthorByInitialsAndLastName` en `bookRepository.findAllByAuthor`.

Deze servicemethodes kunnen we nu ook aanroepen

```
Author = authorRepository.findAuthorByInitialsAndLastnameEquals("J.K.  
Rowling");  
List<Book> booksFound = bookRepository.findAllByAuthor(author);
```

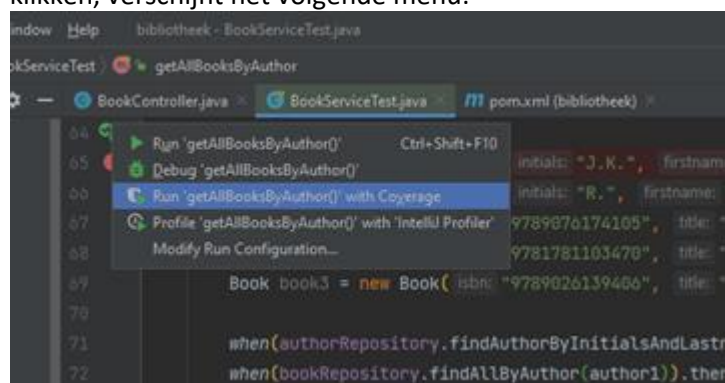
We roepen hier de methodes aan die we willen uitvoeren. Dat maakt dit het Act gedeelte van de test. Doordat we de uitkomst van de methodes opvangen in attributen, kunnen we deze hierna hergebruiken.

In het Assert gedeelte van de test gaan we de verkregen data vergelijken met de verwachte data met de volgende regels:

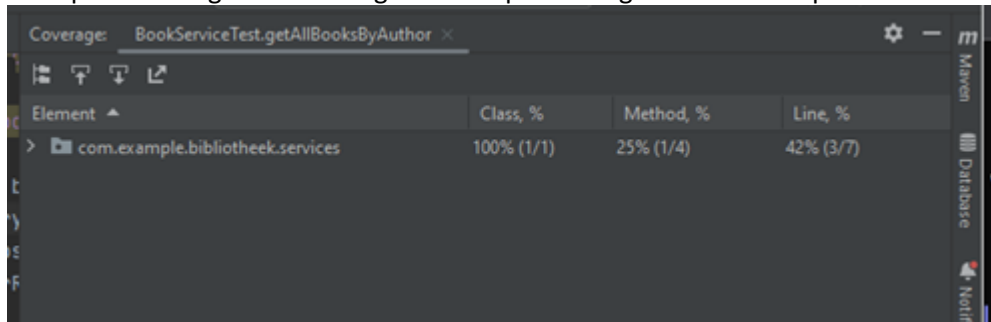
```
assertEquals("J.K.", author.getInitials());  
assertEquals("Rowling", author.getLastname());  
assertEquals(book1, booksFound.get(0));  
assertEquals(book2, booksFound.get(1));
```

Hier testen we wat we verwachten dat er uit de methode komt met de werkelijke uitkomst van de aanroep van de methodes.

Als we nu `@Disabled` verwijderen van de geschreven test, kunnen we de test runnen en zien dat deze slaagt. Als we runnen met coverage, door met de rechtermuisknop op de run-knop in de goot te klikken, verschijnt het volgende menu:



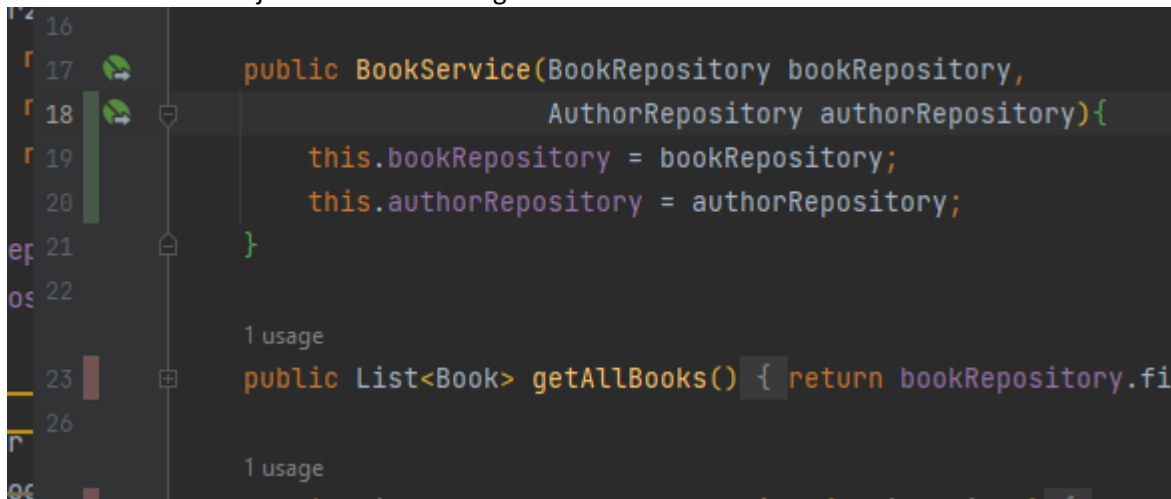
Klik op de blauw gearceerde regel. Dit roept het volgende venster op:



The screenshot shows a 'Coverage' window with a table of coverage data. The table has four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The data row shows 'com.example.bibliotheek.services' with 100% class coverage (1/1), 25% method coverage (1/4), and 42% line coverage (3/7). The window title is 'Coverage: BookServiceTest.getAllBooksByAuthor'.

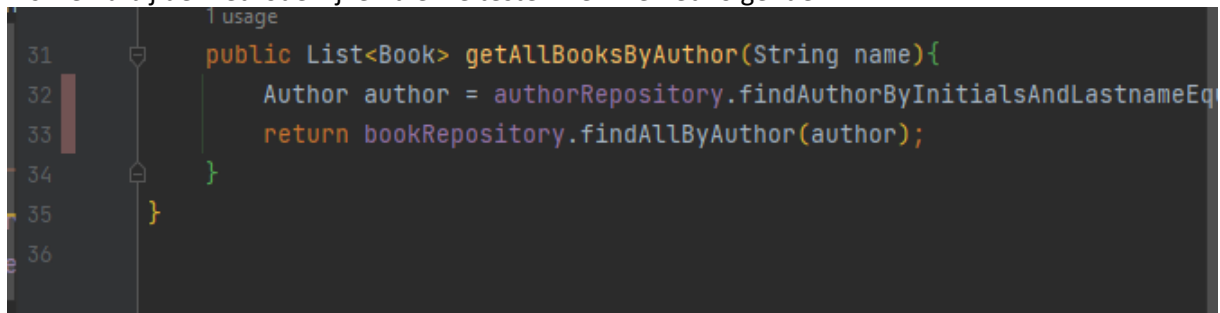
Element	Class, %	Method, %	Line, %
> com.example.bibliotheek.services	100% (1/1)	25% (1/4)	42% (3/7)

Hier zien we dat de line coverage van de applicatie in dit geval 42% is. Als we vervolgens in de BookService klasse kijken zien we het volgende:



De groene lijntjes in de goot geven aan welke regels nu getest zijn, de rode lijntjes geven aan wat nog niet getest is.

Als we nu bij de methode kijken die we testen zien we het volgende:



Dit houdt in dat deze methode nu niet getest wordt. Dit komt omdat we wel de repositories aanspreken, maar niet de daadwerkelijke methode. Laten we de test aanpassen naar:

```

@Test

void getAllBooksByAuthor() {
    Author author1 = new Author( initials: "J.K.", firstname: "Joanne Kathleen", lastname: "Rowling", LocalDate.of(1967, 7, 31));
    Author author2 = new Author( initials: "R.", firstname: "Roald", lastname: "Dahl", LocalDate.of( year 1916, month 1, day 13));
    Book book1 = new Book( isbn: "9789076174105", title: "Harry Potter", subtitle: "en de steen der wijzen", genre: "fiction", language: "Nederlands");
    Book book2 = new Book( isbn: "9781781103470", title: "Harry Potter", subtitle: "en de geheime kamer", genre: "fiction", language: "Nederlands");
    Book book3 = new Book( isbn: "9789026139466", title: "Mathilda", subtitle: "", genre: "fiction", language: "Nederlands");

    when(authorRepository.findAuthorByInitialsAndLastnameEquals("J.K. Rowling")).thenReturn(author1);
    when(bookRepository.findAllByAuthor(author1)).thenReturn(List.of(book1, book2));

    List<Book> booksFound = bookService.getAllBooksByAuthor( name: "J.K. Rowling");

    assertEquals( expected: "J.K.", booksFound.get(0).getAuthor().getInitials());
    assertEquals( expected: "Rowling", booksFound.get(0).getAuthor().getLastname());
    assertEquals(book1, booksFound.get(0));
    assertEquals(book2, booksFound.get(1));
}
}

```

Hier zien we dat we wel de bookservice.getAllBooksByAuthor aanspreken. Laten we eens kijken wat we zien als we opnieuw runnen met coverage.

```

31 public List<Book> getAllBooksByAuthor(String name){
32     Author author = authorRepository.findAuthorByInitialsAndLastnameEquals(name);
33     return bookRepository.findAllByAuthor(author);
34 }
35 }
36

```

En zo wordt de methode wel geheel getest.

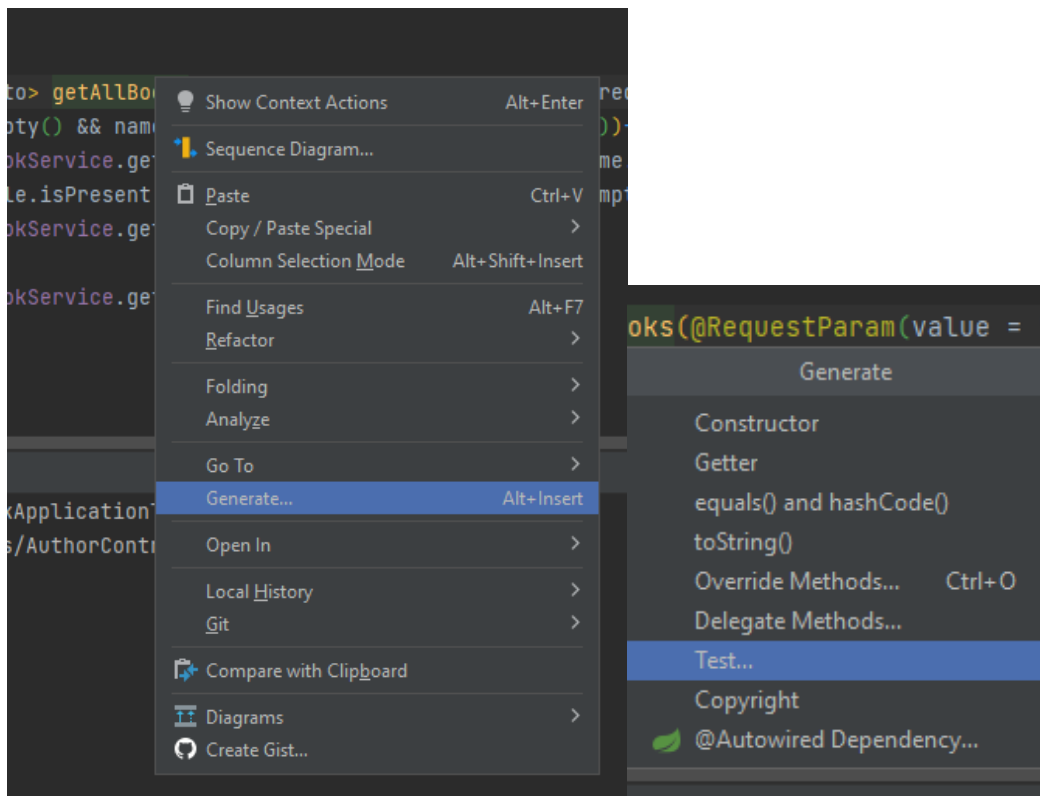
In de [bibliotheek applicatie](#) staan alle unittesten uitgeschreven Ook de testen die niet hier besproken zijn voor de CREATE, UPDATE en DELETE requests. Aan de hand van het voorbeeld dat we hebben besproken, kunnen jullie voor de andere requests oefenen met het schrijven van unittesten.

Wat zijn integratietesten?

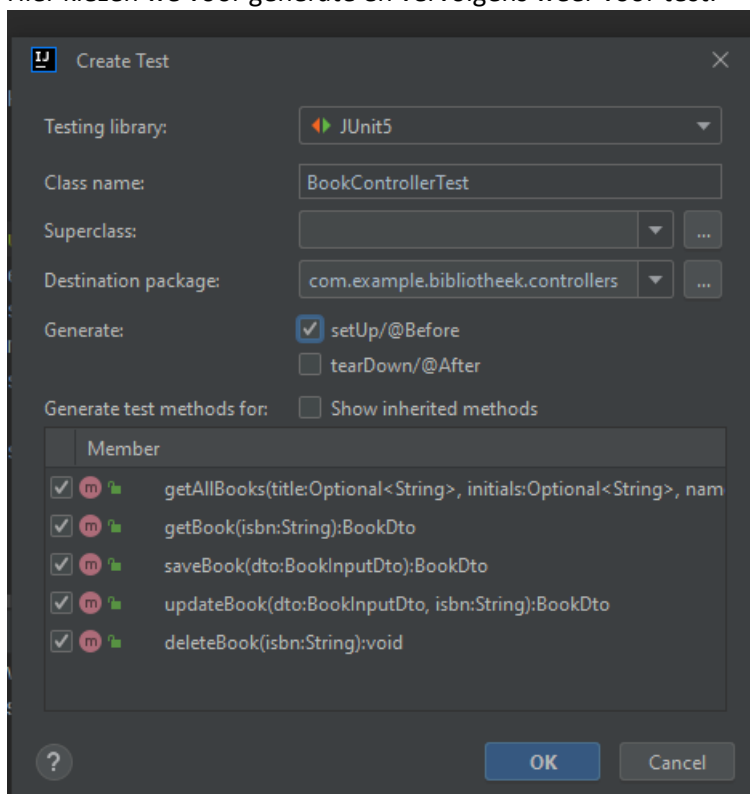
Bij de unittesten zagen we dat we de toegepaste logica van de methodes in isolatie testen, maar komen de resultaten wel ongewijzigd door alle lagen heen? Oftewel: zijn de lagen allemaal juist geïntegreerd? Dat is wat we testen bij de integratietesten.

De laag die we hier testen is de controller laag van de applicatie. De controller laag is zoals jullie weten bij voorkeur zo dom mogelijk. Met andere woorden: hier staat het liefst zo min mogelijk logica in. De controller moet de juiste methode aanroepen en de juiste data doorgeven en meer niet.

Om de testen van de controller te laten genereren, klikken we net als in de service laag op de methodenaam in de controller laag met de rechtermuisknop.



Hier kiezen we voor generate en vervolgens weer voor test.



Vervolgens selecteren we alle methodes uit de controller en de setup methode.

Boven de gegenereerde testklasse zetten we de annotatie:

```
@RunWith(SpringRunner.class)
```

Deze annotatie zorgt dat Spring Boot weet dat we SpringJUnit4ClassRunner willen gebruiken. Hierdoor worden achter de schermen bepaalde connecties gelegd, waar we niet heel diep op in hoeven te gaan. Wel is het van belang dat we weten dat we deze annotatie nodig hebben om onze test klasse te kunnen runnen als test klasse. Mocht je toch meer willen weten over deze annotatie kun je [hier](#) kijken.

Ook gaan we de volgende annotatie boven de klasse plaatsen:

```
@WebMvcTest(BookController.class)
```

Deze annotatie hebben we nodig omdat we Spring Boot willen laten weten dat het om een Spring Boot test klasse gaat met autoconfiguratie voor MockMvc. Dit zou ook kunnen met de annotaties:

```
@SpringBootTest  
@AutoConfigureMockMvc
```

Maar een enkele annotatie is sneller, schoner en netter en hier zijn we als developers natuurlijk heel blij mee. Wil je meer weten over de WebMvcTest annotatie? Kijk dan [hier](#).

Vervolgens willen we MockMvc gebruiken voor het schrijven van onze integratietesten, omdat deze library komt met een hoop methodes die we nodig hebben om te kijken of de integratie goed geïmplementeerd is. Hiervoor moeten we een lokale variabele aanmaken en deze verbinden met de klasse door middel van de autowire annotatie.

```
@Autowired  
private MockMvc mockMvc;
```

(De mockMvc kan de volgende foutmelding geven:

“Could not autowire. No beans of ‘MockMvc’ type found.”

Deze melding mag je negeren. Dit is een foutje in Spring Boot. De applicatie doet er het juiste mee.

Meer over MockMvc vind je [hier](#).)

Om de juiste klasse van de service laag te definiëren, moeten we ook vertellen tegen het programma welke service we gaan gebruiken bij het testen. Althans, welke mockBean van een service we gaan gebruiken. Voor meer over mockbeans kijk [hier](#).

```
@MockBean  
private BookService bookService;
```

Nu hebben we alles ingesteld wat we nodig hebben om aan het testen te beginnen. Voor de eerste test gaan we kijken naar het GET request van getAllBooks van de bibliotheek applicatie. Deze is in de applicatie te vinden in de Bookcontroller klasse.

```
@GetMapping  
public List<BookDto> getAllBooks(@RequestParam(value = "title", required =  
false) Optional<String> title, @RequestParam(value = "initials", required =  
false) Optional<String> initials , @RequestParam(value = "name", required =  
false) Optional<String> name) {  
    if(title.isEmpty() && name.isPresent() && initials.isPresent()){  
        return bookService.getAllBooksByAuthor(initials.get(), name.get());  
    }  
}
```

```

    }else if (title.isPresent() && name.isEmpty() && initials.isEmpty()){
        return bookService.getAllBooksByTitle(title.get());
    }else {
        return bookService.getAllBooks();
    }
}

```

Hier zien we dat deze methode is opgesplitst in verschillende mogelijkheden. Om deze klasse honderd procent getest te krijgen, zullen we al deze mogelijkheden ook moeten testen.

Dus om de getAll request mapping te testen zullen we drie verschillende tests moeten schrijven. Eén voor alle boeken, één voor alle boeken met een bepaalde titel en één voor alle boeken met een bepaalde auteur.

Laten we beginnen met de eerste: een test voor alle boeken. Om dit te testen hebben we een aantal boeken nodig. Om te voorkomen dat we straks weer bij elke test meerdere boeken moeten aanmaken, gaan we gelijk gebruikmaken van de beforeEach methode.

Voor elk boek dat we gaan aanmaken, zetten we een declaratie van de variabele boven in de klasse. Laten we drie boeken gebruiken. Omdat we gebruik willen maken van deze objecten door heel de klasse, declareren we deze bovenaan en vullen we ze pas in de beforeEach methode.

Dit ziet er als volgt uit:

```

Book book1;
Book book2;
Book book3;

@BeforeEach
public void setUp() {

    book1 = new Book("9789076174105", "Harry Potter", "en de steen der wijzen", "fiction", "NL", "paperback", "uitgeverij de Harmonie", author1);
    book2 = new Book("9781781103470", "Harry Potter", "en de geheime kamer", "fiction", "NL", "paperback", "Pottermore publishing", author1);
    book3 = new Book("9789026139406", "Mathilda", "", "fiction", "NL", "paperback", "de Fontein Jeugd", author2);
}

```

Zoals we zien hebben we voor een boek een auteur nodig. Deze moeten we dus aanmaken. Daarnaast zien we in de bookController dat we werken met dtos. We hebben dus ook een book en auteur dtos nodig. Omdat we in de book controller ook werken met een bookInputDto, moet ook deze worden aangemaakt. De informatie die je moet invullen, is niet altijd van tevoren te besluiten. Daarom kun je de informatie aanvullen op het moment dat je het nodig hebt. Het ziet er dan als volgt uit:

```

Book book1;
Book book2;
Book book3;
BookDto bookDto1;
BookDto bookDto2;
BookDto bookDto3;

BookDto bookDto4;

```

```

Author author1;
Author author2;
AuthorDto authorDto1;
AuthorDto authorDto2;

BookInputDto bookInputDto1;
BookInputDto bookInputDto2;

@BeforeEach
public void setUp() {
    author1 = new Author(UUID.fromString("aabe4998-522a-4bd7-97c4-c296b7fb0336"), "J.K.", "Joanne Kathleen", "Rowling",
        LocalDate.of(1965, 7, 31), Gender.FEMALE);
    author2 = new Author(UUID.fromString("70080f94-539c-466d-9976-b838dc037842"), "R.", "Roald", "Dahl", LocalDate.of(1916, 9, 13),
        Gender.MALE);

    book1 = new Book("9789076174105", "Harry Potter", "en de steen der wijzen", "fiction", "NL", "paperback", "uitgeverij de Harmonie", author1);
    book2 = new Book("9781781103470", "Harry Potter", "en de geheime kamer", "fiction", "NL", "paperback", "Pottermore publishing", author1);
    book3 = new Book("9789026139406", "Mathilda", "", "fiction", "NL", "paperback", "de Fontein Jeugd", author2);

    authorDto1 = new AuthorDto(UUID.fromString("aabe4998-522a-4bd7-97c4-c296b7fb0336"), "J.K.", "Joanne Kathleen", "Rowling",
        LocalDate.of(1965, 7, 31), Gender.FEMALE);
    authorDto2 = new AuthorDto(UUID.fromString("70080f94-539c-466d-9976-b838dc037842"), "R.", "Roald", "Dahl", LocalDate.of(1916, 9, 13),
        Gender.MALE);

    bookDto1 = new BookDto("9789076174105", "Harry Potter", "en de steen der wijzen", "fiction", "NL", "paperback", "uitgeverij de Harmonie", authorDto1);
    bookDto2 = new BookDto("9781781103470", "Harry Potter", "en de geheime kamer", "fiction", "NL", "paperback", "Pottermore publishing", authorDto1);
    bookDto3 = new BookDto("9789026139406", "Mathilda", "", "fiction", "NL", "paperback", "de Fontein Jeugd", authorDto2);
    bookDto4 = new BookDto("9789076174105", "Harry Potter", "en de geheime kamer", "science fiction", "EN", "ebook", "Pottermore publishing", authorDto2);

    bookInputDto1 = new BookInputDto("9789076174105", "Harry Potter", "en de steen der wijzen", "fiction", "NL", "paperback", "uitgeverij de Harmonie", UUID.fromString("aabe4998-522a-4bd7-97c4-c296b7fb0336"));
    bookInputDto2 = new BookInputDto("9789076174105", "Harry Potter", "en de geheime kamer", "science fiction", "EN", "ebook", "Pottermore publishing", UUID.fromString("70080f94-539c-466d-9976-b838dc037842"));
}

```

Dit zorgt ervoor dat we in de test methodes niet steeds hetzelfde hoeven te plaatsen, dus krijgen we schonere en overzichtelijkere code.

Laten we nu kijken naar de get allBooks test methode. Deze heeft de @Test annotatie zodat Spring Boot weet dat dit een test is.

```
@Test
void getAllBooks() {
}
```

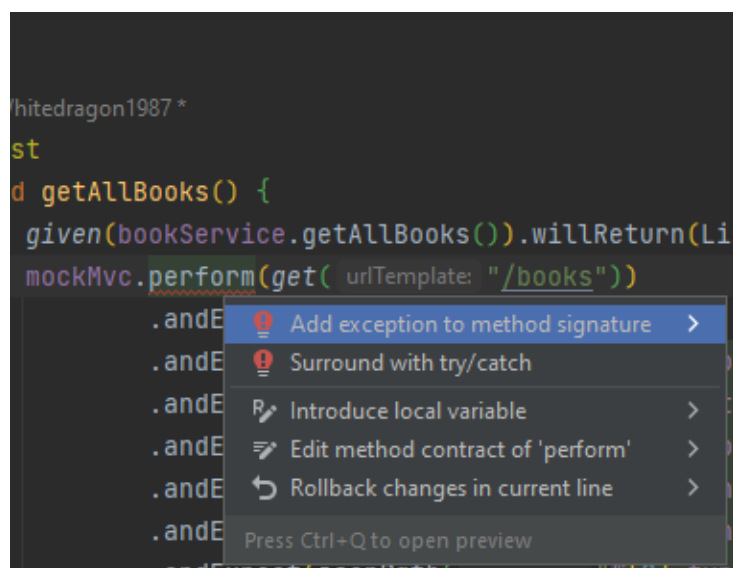
We willen testen, dat er iets gebeurt wanneer we het endpoint van deze methode aanroepen. Om controle te houden over wat er precies gebeurt, kunnen we gebruikmaken van de statische methode `given().willReturn()` van Mockito. In dit geval gaan we vertellen dat we graag een lijst van de drie `bookDtos` terug willen krijgen als de `bookService.getAllBooks()` wordt aangeroepen. Dit doen we als volgt:

```
given(bookService.getAllBooks()).willReturn(List.of(bookDto1, bookDto2,
bookDto3));
```

Daarna maken we gebruik van `MockMvc` door `mockMvc.perform()` te gebruiken.

```
mockMvc.perform(get("/books"))
    .andExpect(status().isOk())
```

We zeggen hier dat we een GET request willen uitvoeren op het endpoint “/books” en dat we een 200 statuscode verwachten met de waarde `isOk()`. Deze methodes komen met `MockMvc` mee, dus daar hoeven we niet allerlei extra code voor te schrijven. Op het moment dat we dit hebben geschreven, wordt het woord ‘perform’ rood onderstreept door Spring Boot. Dit komt omdat als er niks gevonden wordt in de `bookService`, bij de methode de applicatie een exception gooit. Dan verwacht de test dat we daar hier ook wat over melden. We kunnen dit oplossen door met de rechtermuisknop op het woord ‘perform’ te klikken. Vervolgens kunnen we kiezen voor ‘Show context actions’ en voor ‘Add exception to method signature’.



Dit verandert de methode zoals hieronder afgebeeld:

```
@Test
void getAllBooks() throws Exception {
    given(bookService.getAllBooks()).willReturn(List.of(bookDto1, bookDto2, bookDto3));
    mockMvc.perform(get( urlTemplate: "/books"))
        .andExpect(status().isOk())
```

Vervolgens moeten we kijken of alle waarden overeenkomen met wat we verwachten. Dit doen we als volgt:

```

@Test
void getAllBooks() throws Exception {
    given(bookService.getAllBooks()).willReturn(List.of(bookDto1, bookDto2,
bookDto3));
    mockMvc.perform(get("/books"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.isbn").value("9789076174105"))
        .andExpect(jsonPath("$.title").value("Harry Potter"))
        .andExpect(jsonPath("$.subtitle").value("en de steen der
wijzen"))
        .andExpect(jsonPath("$.genre").value("fiction"))
        .andExpect(jsonPath("$.language").value("NL"))
        .andExpect(jsonPath("$.type").value("paperback"))
        .andExpect(jsonPath("$.publisher").value("uitgeverij de
Harmonie"))
        .andExpect(jsonPath("$.authorDto.uuid").value("aabe4998-
522a-4bd7-97c4-c296b7fb0336"))
        .andExpect(jsonPath("$.authorDto.initials").value("J.K.))
        .andExpect(jsonPath("$.authorDto.firstname").value("Joanne
Kathleen"))
        .andExpect(jsonPath("$.authorDto.lastname").value("
Rowling"))
        .andExpect(jsonPath("$.authorDto.dateOfBirth").value("1965-
07-31"))
        .andExpect(jsonPath("$.authorDto.gender").value("FEMALE"))
        .andExpect(jsonPath("$.isbn").value("9781781103470"))
        .andExpect(jsonPath("$.title").value("Harry Potter"))
        .andExpect(jsonPath("$.subtitle").value("en de geheime
kamer"))
        .andExpect(jsonPath("$.genre").value("fiction"))
        .andExpect(jsonPath("$.language").value("NL"))
        .andExpect(jsonPath("$.type").value("paperback"))
        .andExpect(jsonPath("$.publisher").value("Pottermore
publishing"))
        .andExpect(jsonPath("$.authorDto.uuid").value("aabe4998-
522a-4bd7-97c4-c296b7fb0336"))
        .andExpect(jsonPath("$.authorDto.initials").value("J.K.))
        .andExpect(jsonPath("$.authorDto.firstname").value("Joanne
Kathleen"))
        .andExpect(jsonPath("$.authorDto.lastname").value("
Rowling"))
        .andExpect(jsonPath("$.authorDto.dateOfBirth").value("1965-
07-31"))
        .andExpect(jsonPath("$.authorDto.gender").value("FEMALE"))
        .andExpect(jsonPath("$.isbn").value("9789026139406"))
        .andExpect(jsonPath("$.title").value("Mathilda"))
        .andExpect(jsonPath("$.subtitle").value(""))
        .andExpect(jsonPath("$.genre").value("fiction"))
        .andExpect(jsonPath("$.language").value("NL"))
        .andExpect(jsonPath("$.type").value("paperback"))
        .andExpect(jsonPath("$.publisher").value("de Fontein
Jeugd"))
        .andExpect(jsonPath("$.authorDto.uuid").value("70080f94-
539c-466d-9976-b838dc037842"))
        .andExpect(jsonPath("$.authorDto.initials").value("R.))
        .andExpect(jsonPath("$.authorDto.firstname").value("
Roald"))
        .andExpect(jsonPath("$.authorDto.lastname").value("Dahl"))
        .andExpect(jsonPath("$.authorDto.dateOfBirth").value("1916-
09-13"))

```



```
.andExpect(jsonPath("$.authorDto.gender").value("MALE"));
}
```

AndExpect is een ingebouwde methode van MockMvc, waarmee we de uitkomst kunnen vergelijken met de verwachte waarde van een attribuut bij terugkomst. We werken met Json format dus kunnen we gebruikmaken van de waarde JsonPath. Met het dollarteken (\$) geven we aan dat we het hebben over de lijst die terugkomt. Tussen de squared brackets geven we de positie van de lijst die we willen checken, gevolgd door de naam van het attribuut dat we willen vergelijken. Met de .value() kunnen we vergelijken welke waarde we verwachten dat het attribuut heeft, die we natuurlijk plaatsten tussen de haakjes van de .value(de waarde die we verwachten).

In deze applicatie is bewust gebruik gemaakt van een genest object in het book object, namelijk de auteur. Door \$[0].authorDto.uuid gaan we dus naar de eerste positie van de betreffende lijst en vervolgens naar het attribuut authorDto (wat het geneste object is) en daarna naar .uid om de uid waarde van de auteur te vergelijken.

Op deze manier testen we alle waardes die we terugkrijgen om te zorgen dat we een honderd procent dekking van de huidige methode krijgen. Zo schrijven we voor elke methode in de controller en eventuele splitsingen van de methodes integratietesten. Voor de gehele code en voorbeelden van de gebruikte testen kan je [hier](#) kijken.

Hier staan ook testen voor de GET, PUT, POST en DELETE requests.

Wat te doen met de lagen die nu nog niet getest zijn?

De model en dto lagen testen we niet expliciet. Impliciet worden deze wel mee getest door de unittesten van de service laag. Het kan zijn dat je toekomstige werkgever dat wel graag wilt, maar dit valt buiten het bereik van onze opleiding. Wel heb je nu kennis van de twee van de meest gebruikte testen in de development wereld: de unit- en integratietesten.

De repositories die we gebruiken extenden in de meeste gevallen puur de JpaRepository. Deze repository is geheel getest in zijn vorm. Daarom hoeven we deze laag meestal niet te testen. Hooguit in het geval dat we zelf een query schrijven. Ook zijn deze testen niet noodzakelijk voor je eindopdracht.

!!! Tips !!!

Bij het testen van je applicatie zal je vaak problemen tegenkomen met je autorisatie en authenticatie. Schrijf daarom eerst je test uit zonder gebruik te maken van filters. Dit kan je doen door boven aan je test klasse de @AutoConfigureMockMvc(addFilters = false) te zetten. Let wel op dat je nog wel alle afhankelijkheden uit de service laag moet mocken. Als je test vervolgens slaagt kan je proberen om de security weer aan te zetten door @AutoConfigureMockMvc(addFilters = false) weer te verwijderen en daarvoor in de plaats @WithMockUser te gebruiken met de juiste autorisatie.

Maak gebruik van een aparte applications.property voor je testen. Dit wordt ook in de bibliotheek applicatie gedaan. Dit is belangrijk, omdat je soms voor je testen andere configuraties nodig hebt dan in je werkelijke applicatie. Op deze manier maak je niks kapot in je applicatie.

Dit zijn natuurlijk enkele voorbeelden van hoe je je applicatie kan testen. Dat wil niet zeggen dat het niet op andere manieren kan. Je kan bijvoorbeeld ook een h2 database aanmaken en gebruiken voor je testen. Maar daar gaan we hier verder niet op in.