

Test Project

Læringsmål

Læringsmålene har til formål at sikre et fagligt og organiseret projekt forløb. Vores læringsmål består af 2 typer læringsmål: Web Service (REST Testing) og Database (Mock Testing). Disse læringsmål er udarbejdet i samarbejde med Lars Mortensen og godkendt af begge parter.

REST Testing

- Learn how to use SoapUI for REST Testing, and achieve a better understanding of some of problems involved with REST Testing

Dette læringsmål havde til formål at videreudvikle vores evner inden for RESTful Web Service og lære at arbejde på tværs over forskellige værktøjer/platforme som SoapUI til at teste vores RESTful Web Service.

Vi valgte at gøre brug af SoapUI fordi det er branchens mest omfattende værktøj og hurtig at komme i gang med. Derudover hjælper SoapUI med at sikre, at vores API'er fungerer som forventet og opfylder vores funktionelle krav (API Testing Tool). Vi har kombineret SoapUI og Postman til RESTful Web Service testing og har benyttet begge værktøjer frem og tilbage for at

sikre at vores applikation fungerer som forventet. Vi har primært

gjort brug af SoapUI's interface til at oprette HTTP test metoder såsom GET og DELETE. Ud fra vores standard HTTP metoder har

vi lært hvordan man gør brug af "Assertions" i SoapUI til at teste HTTP Validation koder og Contains Assertions til at sikre at programmet returnere som forventet. SoapUI giver også mulighed for at lave Load test og security test, men Load test har vi ikke gjort brug af, fordi vi har brugt Gutenberg Databasen fra database projektet og dette er en meget stor database. Databasens størrelse har gjort at SoapUI ikke har været istand til at returnere alle elementer og fungere korrekt. Security test har vi hellere ikke gjort brug af, fordi vi ikke har implementeret sikkerhed i vores applikation. Men i en virkelig applikation ville disse funktioner være uundværlige, da man skal sikre at sikkerheden og ydeevnen er i orden. Vi har primært arbejdet med Funktionel API tests i SoapUI, og har lavet tests til alle vores RESTful Web Service ressourcer, herunder author (GET, GET by id, DELETE by id), book (GET, GET by id, DELETE by id), location (GET, GET by id, DELETE by id), getBookByCity (GET without parameter, GET by parameter), getBookByAuthor (GET without parameter, GET by parameter), getBookByGeolocation (GET without parameter, GET by

● Contains Richard Harding Davis – VALID

● Contains locations – VALID

● Valid HTTP Status Code 404 – VALID

● Invalid HTTP Status Code 200 – VALID

parameter) og getLocationByTitle (GET without parameter, GET by parameter)¹. Vi har lært hvornår man gør brug af TestCase, TestSteps og TestSuite. SoapUI strukturerer funktionel tests i tre niveauer, TestSuites, TestCases og TestSteps. TestSuites er en samling af TestCases der kan bruges til gruppering af funktionelle tests i en logisk enhed. En TestCase er en samling af TestSteps der er samlet for at teste nogle specifikke aspekter af vores services. Sidst men ikke mindst har vi TestSteps som er byggestenene af funktionelle tests i SoapUI og bruges til at kontrollere "flow of Execution" og validere funktionaliteten af vores services. Når vi kører vores TestSuite som består af vores TestSteps, kan vi se resultatet i Test Output, som indeholder en eksekveringslog i bunden, som har til formål at vise løbende information om udførte TestSteps og deres status.

Vi har lært hvorfor "The Testing Pyramid" fra undervisning er en vigtig princip. Formålet med pyramiden er, at man bør

have mange flere low-level unit tests end high-level end-to-end tests, der kører gennem en GUI.

Omkostningerne ved test går op, når man går op i pyramiden. Dækningen er højere ved bunden. I starten ville vi gerne gøre brug af Selenium testing til User interface testing, men efter samtalen med Lars, valgte vi at bruge SoapUI som IKKE er User interface testing, men derimod WebAPI og Web Service testing. Vi kan se hvorfor Lars anbefalede os at arbejde med REST testing fremfor Selenium, da det er langt vigtigere at have flere low-level unit tests end high-level end-to-end tests, der kører gennem en GUI (Selenium).

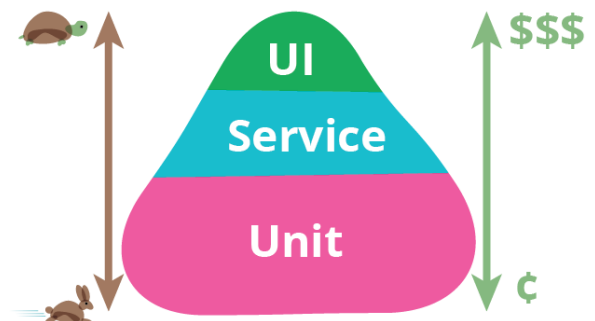
Vi har lært hvor vigtigt det er at teste API'er, bl.a. fordi det sikrer at vores API'er fungerer korrekt, men også fordi de kan afsløre problemer med din API, server og andre tjenester, netværk og meget mere, som man måske ikke opdager eller løser ved nemhed efter implementering. Vi har lært at man kan kombinere API testing med CI/CD og TDD. Man kan tilføje RESTful API testing i udviklingsprocessen på forskellige måder. Mange virksomheder inkludere API tests i deres Continuous Integration (CI) og Continuous Deployment (CD). Hvis API testene fejler under CI eller CD, stoppes processen og API problemet løses, før builden er færdig. Vi har lært at API tests er kernen i API overvågning, fordi under udviklingen af enhver software applikation har API tests mange fordele på tværs over hold og helt ned til hvordan kunden oplever produktet. I dette

```

REST author TestSuite
└─ http://localhost:8080 TestSuite
   └─ Test Steps (3)
      └─ GET all authors
      └─ GET author by id
      └─ DELETE author by id
      └─ Load Tests (0)
      └─ Security Tests (0)

```

Step 1 [GET all authors] OK: took 15 ms
 Step 2 [GET author by id] OK: took 9 ms



¹ SoapUI XMI filer og Skærmbilleder kan findes i "Soap XMI" og "Soap Screenshots" mappen

projekt forløb har vi været i stand til at sætte os ind i SoapUI og udviklet API tests til vores RESTful Web Service. Derudover har vi videreudviklet vores kompetencer inden for RESTful Web Service og lært hvor vigtigt det er at teste sine API'er.

Mock Testing

- Further develop our competencies within the Mockito Framework
- Learn to mock away external dependencies (database).

Mocking bruges primært i unit testing. Et objekt under test kan have afhængigheder af andre (komplekse) objekter. For at isolere opførelsen af objektet som man vil teste, kan man så erstatte disse objekter med mocks. Disse mocks simulere opførelsen af de rigtige objekter. Dette kan være meget nyttigt hvis man f.eks. arbejder med objekter fra database, ved at bruge mock kan vi så undgå at dykke ned i databasen, og gøre testing mindre kompleks.

```
@Test
public void testGetBook() {
    System.out.println("getBook");

    DBfacade mockedFacade = mock(DBfacade.class);
    when(mockedFacade.getBook("1")).thenReturn(new book("1", "The Book", "Big bad book"));

    Controller instance = new Controller(mockedFacade);
    book expResult = new book("1", "The Book", "Big bad book");
    book actualResult = instance.getBook("1");
    assertEquals(expResult.toString(), actualResult.toString());
    // TODO review the generated test code and remove the default call to fail.
}
```

På ovenstående figur, kan vi se vores test af Get Book, hvor vi har benyttet os af Mock. I dette tilfælde har vi "mocked" vores database facade, som gør at vi ikke har rodet i databasen, og undgået diverse fejl der kunne opstå. Vi har benyttet os af mock, så vi ikke behøver at oprette dummy data i databasen, og på denne måde slipper for at indsætte og slette dummy data.

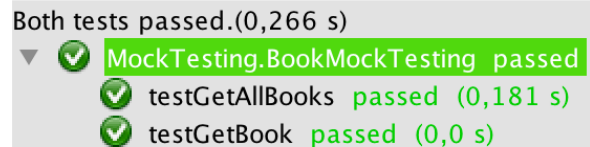
Vi kan se linjen:

```
when(mockedFacade.getBook("1")).thenReturn(new book("1", "The Book", "Big bad book"));
```

Det vi gør her er, at vi udarbejder et falskt dyk ned i databasen og lader som om at vi har hivet en bog ud, som har følgende værdier:

- ID: 1
- Titel: The Book
- Text: Big bad book

Derefter sætter vi vores forventede resultat til at være værdierne vi har hivet ud, og derefter benytter vi os af vores mocked controller, som kalder metoden `getBook("1")`; som har ID 1. Vi sammenligner så expected og actual results.



```
Both tests passed.(0,266 s)
MockTesting.BookMockTesting passed
  testGetAllBooks passed (0,181 s)
  testGetBook passed (0,0 s)
```

I vores test har vi også inkluderet Parameters. Hvis vi kigger på figuren ovenover, kan vi se, at er blevet oprettet et Array, med 5 forskellige by navne. I vores Mock tests har vi benyttet os af disse parametre og brugt dem til at mocke 5 forskellige byer. Dette gør at vi har undgået at lave flere asserts, men i stedet har udarbejdet en løsning via parameters der gør, at selvsamme test bliver kørt 5 gange med forskellige by navne.

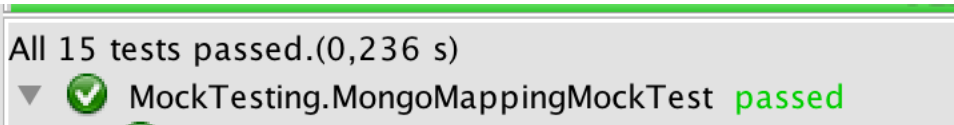
```
@Parameters
public static Collection<String> data() {
    return Arrays.asList(new String[]{"London", "Copenhagen", "Lissabon", "Berlin", "Stockholm"});
}

@Test
public void testGetBooksByCity() {
    System.out.println("getBooksByCity with city: " + _data);
    List<DTOAuthorBook> listBooks = new ArrayList<>();

    for (int i = 0; i < 10; i++) {
        listBooks.add(new DTOAuthorBook("title" + i, _data));
    }

    MongoMapping mocked = mock(MongoMapping.class);
    when(mocked.getBooksByCity(_data)).thenReturn(listBooks);

    for (int i = 0; i < 10; i++) {
        assertEquals(listBooks.iterator().next().toString(), mocked.getBooksByCity(_data).iterator().next().toString());
    }
}
```



```
All 15 tests passed.(0,236 s)
```

```
MockTesting.MongoMappingMockTest passed
```

I den ene testklasse hvor vi har 3 tests, ser resultatet sådan ud (se ovenstående figur). I test klassen som er med i eksemplet her, er der 3 tests. Dog er der 15 tests der er blevet kørt, grunden til dette er at hver test har 5 forskellige parametre, og bliver kørt 5 gange hver.

I dette projekt forløb har vi videreudviklet vores kompetencer inden for Mockito Frameworket, samt lært hvorfor hvor godt et redskab/værktøj det egentlig er. Alt i alt kan vi konkludere at vi har opnået vores læringsmål inden for database (mock) og Web Service (REST Testing).