

Integration and Functional Tests

License



This work by Thalita Vergilio at Leeds Beckett University is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Contents

[License](#)

[Contents](#)

[Introduction](#)

[The student project](#)

[Creating a view for a student record](#)

[Implementing MVC functionality to find a student by its ID](#)

[Preparing to write integration tests](#)

[Writing integration tests for the student service](#)

[Writing functional tests for the API endpoints](#)

[Writing functional tests for the MVC application](#)

[Assignment](#)

[Conclusion](#)

Introduction

The aim of this week's lab is to write integration and functional tests for our student microservice. We will learn how to use two new Groovy-based test frameworks: Spock and Geb, designed to facilitate the creation of simple, concise, behaviour-driven tests. Their clear, intuitive syntax represents a powerful abstraction over established Java technologies such as JUnit, Mockito and Selenium.

The student project

We are going to use the student project we have been building over the course of this module. You should have a student service class with a method that gets a student by its ID. Your implementation may differ from the one below, but that's ok, as long as both are functionally equivalent.

```
StudentService.java X
1  package uk.ac.leedsbeckett.student.service;
2
3  import org.springframework.hateoas.EntityModel;
4  import org.springframework.stereotype.Component;
5  import uk.ac.leedsbeckett.student.controller.StudentController;
6  import uk.ac.leedsbeckett.student.model.Student;
7  import uk.ac.leedsbeckett.student.model.StudentRepository;
8
9  import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
10 import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;
11
12 @Component
13 public class StudentService {
14     private final StudentRepository studentRepository;
15
16     public StudentService(StudentRepository studentRepository) {
17         this.studentRepository = studentRepository;
18     }
19
20     public EntityModel<Student> getStudentByIdJson(Long id) {
21         Student student = studentRepository.findById(id)
22             .orElseThrow(() -> new RuntimeException("Course with id " + id + " not found."));
23         return EntityModel.of(student,
24             linkTo(methodOn(StudentController.class)
25                 .getStudentJson(student.getId())).withSelfRel());
26     }
27 }
28
```

Student service implementation

You should also have a student controller mapping the `/api/students/{id}` endpoint to your controller method. The controller method should call the service method to obtain a JSON representation of a student.

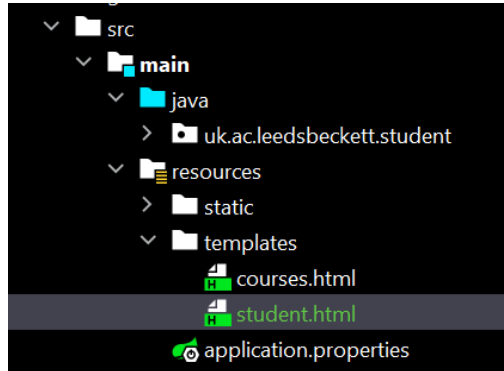
```
StudentController.java X
1  package uk.ac.leedsbeckett.student.controller;
2
3  import org.springframework.hateoas.EntityModel;
4  import org.springframework.stereotype.Controller;
5  import org.springframework.web.bind.annotation.GetMapping;
6  import org.springframework.web.bind.annotation.PathVariable;
7  import org.springframework.web.bind.annotation.ResponseBody;
8  import uk.ac.leedsbeckett.student.model.Student;
9  import uk.ac.leedsbeckett.student.service.StudentService;
10
11  @Controller
12  public class StudentController {
13
14      private final StudentService studentService;
15
16      public StudentController(StudentService studentService) {
17          this.studentService = studentService;
18      }
19
20      @GetMapping("/api/students/{id}")
21      @ResponseBody
22      public EntityModel<Student> getStudentJson(@PathVariable Long id) {
23          return studentService.getStudentByIdJson(id);
24      }
25
26  }
```

Student controller implementation

Creating a view for a student record

If you have already implemented something similar to this, feel free to skip this part. If not, we are going to create a simple front-end application view that displays the details of a student.

Create a file called student.html in called StudentService in the location shown below.



Location of student.html

Enter some simple HTML to display a student record.

```
student.html x
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Student Profile</title>
6 </head>
7 <body>
8     <h2 th:inline="text">Student Profile</h2>
9     <!--/*@thymesVar id="student" type="uk.ac.leedsbeckett.student.Student"*/-->
10    <p th:inline="text">First Name: [[${student.forename}]]</p>
11    <p th:inline="text">Surname: [[${student.surname}]]</p>
12    <p th:inline="text">Student ID: [[${student.externalStudentId}]]</p>
13 </body>
14 </html>
```

Simple implementation for student.html

Implementing MVC functionality to find a student by its ID

Now that we have a simple view, we are going to create a service and a controller method to make use of it.

Create a method called `getStudentById()` in the student service class. Use the student repository to fetch the record. Follow the example below.

```

29     public ModelAndView getStudentById(Long id) {
30         ModelAndView modelAndView = new ModelAndView(viewName: "student");
31         modelAndView.addObject(studentRepository.findById(id).orElseThrow(RuntimeException::new));
32         return modelAndView;
33     }

```

Implementation of getStudentById() in StudentService

Finally, create a method called getStudent() in the student controller class. Map it to the /students/{id} endpoint. Follow the example below.

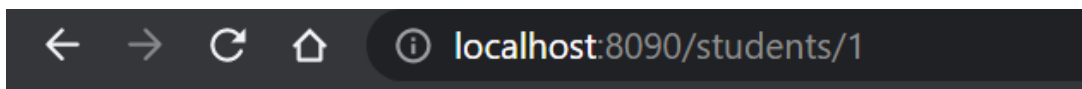
```

27     @GetMapping("/students/{id}")
28     public ModelAndView getStudent(@PathVariable Long id) {
29         return studentService.getStudentById(id);
30     }

```

Implementation of getStudent() in StudentController

If you run the application now, you should be able to access your new student page on the URL below.



A screenshot of a web browser's address bar. It shows navigation icons (back, forward, refresh, home) on the left, followed by an information icon and the URL "localhost:8090/students/1".

Student Profile

First Name: Thalita

Surname: Vergilio

Student ID: c9999999

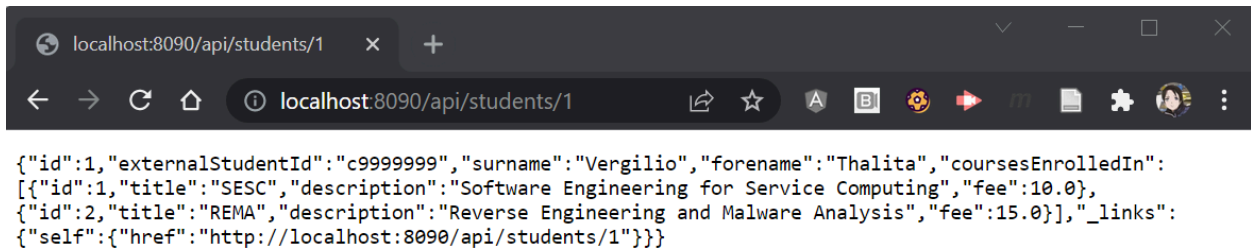
Student page in MVC front-end application

Now we are ready to start writing tests.

Preparing to write integration tests

We are going to write integration tests to verify that our service can communicate with the repository, and the repository with the database.

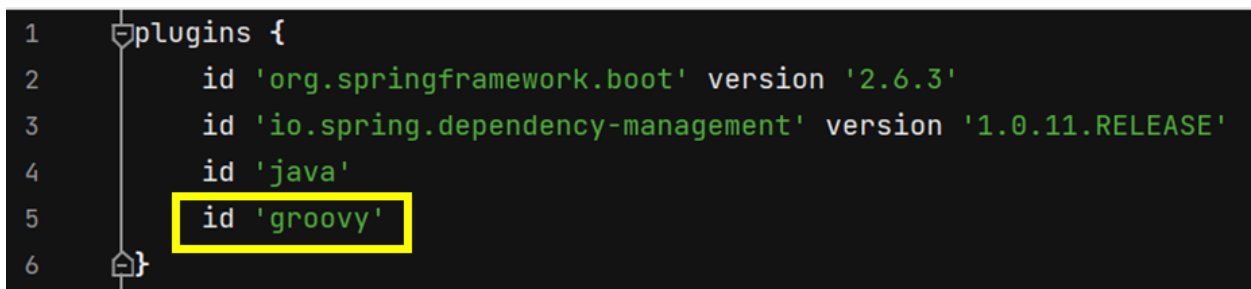
First of all, let's ensure that we have a working RESTful endpoint. **Run your application and visit `/api/students/1`.** You should get a response similar to the one below.



```
{
  "id": 1,
  "externalStudentId": "c99999999",
  "surname": "Vergilio",
  "forename": "Thalita",
  "coursesEnrolledIn": [
    {
      "id": 1,
      "title": "SESC",
      "description": "Software Engineering for Service Computing",
      "fee": 10.0
    },
    {
      "id": 2,
      "title": "REMA",
      "description": "Reverse Engineering and Malware Analysis",
      "fee": 15.0
    }
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8090/api/students/1"
    }
  }
}
```

Checking that the RESTful endpoint has been implemented

Now we need to import some dependencies. Since we are using Spock for integration tests, we first need to add Groovy support. **In your `build.gradle`, add the following line to the `plugins` closure:**



```
1  plugins {
2      id 'org.springframework.boot' version '2.6.3'
3      id 'io.spring.dependency-management' version '1.0.11.RELEASE'
4      id 'java'
5      id 'groovy'
6  }
```

Adding support for Groovy

Add the four dependencies below. The first is the core Spock dependency. The second adds integration with Spring. The third adds Groovy support. Finally, the fourth allows us to create and send HTTP requests programmatically.

```

22 dependencies {
23     implementation 'org.springframework.boot:spring-boot-starter-web'
24     implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
25     implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
26     implementation 'org.springframework.boot:spring-boot-starter-hateoas'
27     compileOnly 'org.projectlombok:lombok'
28     implementation 'com.h2database:h2'
29     annotationProcessor 'org.projectlombok:lombok'
30     testImplementation 'org.springframework.boot:spring-boot-starter-test'
31     testImplementation 'org.spockframework:spock-core:2.2-M1-groovy-3.0'
32     testImplementation 'org.spockframework:spock-spring:2.2-M1-groovy-3.0'
33     testImplementation 'org.codehaus.groovy:groovy-all:3.0.9'
34     testImplementation 'org.codehaus.groovy.modules.http-builder:http-builder:0.7.1'
35 }

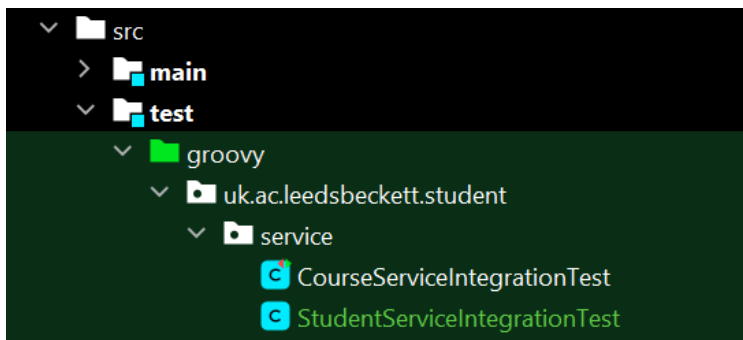
```

Adding dependencies for Spock

Refresh your Gradle project to ensure the dependencies are downloaded.

Writing integration tests for the student service

We can now start writing our first integration test. Create a Groovy class called `StudentServiceIntegrationTest` in `src/test/groovy`.



Location of student service integration test

Your test class should extend `spock.lang.Specification`.

```

1 package uk.ac.leedsbeckett.student.service
2
3 import org.springframework.beans.factory.annotation.Autowired
4 import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
5 import org.springframework.boot.test.context.SpringBootTest
6 import org.springframework.test.context.ActiveProfiles
7 import spock.lang.Specification
8
9 @SpringBootTest
10 @AutoConfigureTestDatabase
11 @ActiveProfiles("test")
12 class StudentServiceIntegrationTest extends Specification {
13
14     @Autowired
15     private StudentService studentService
16 }
17

```

Student service integration test class

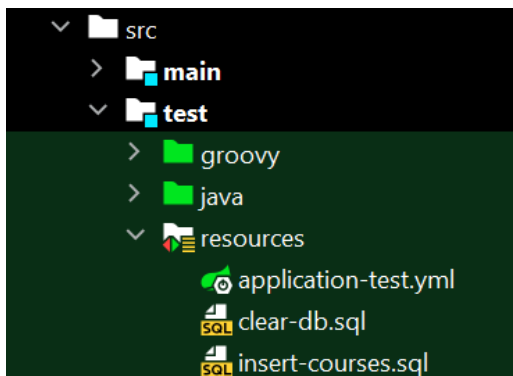
Add the annotations following the example above.

Add a private property of type `StudentService` and ensure it is annotated as `@Autowired`.

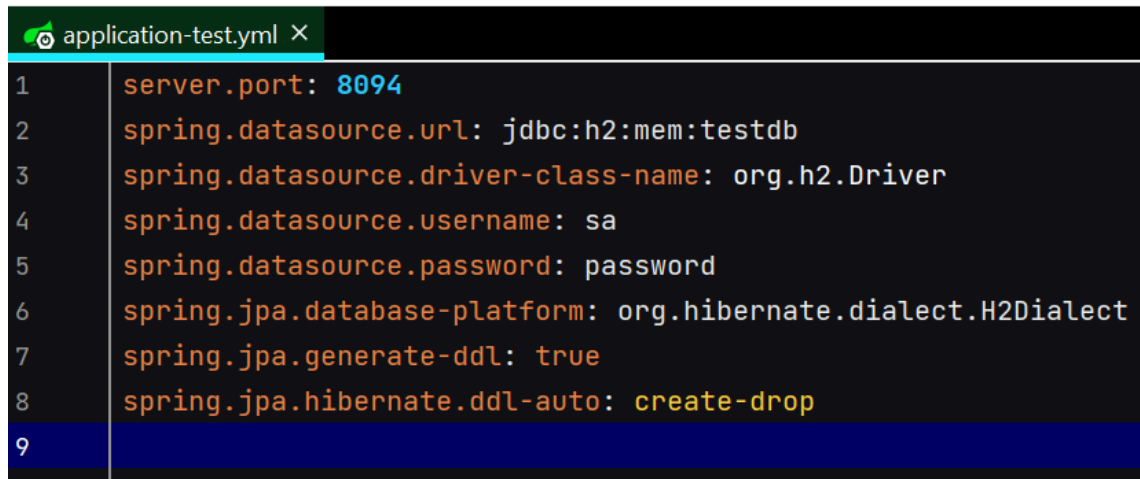
In your log book: explain the purpose of the four annotations in the example above.

Since we are testing how our classes communicate with the database, we need to have an actual database running. It is bad practice to run tests on a production database, so we are going to create some configuration to ensure our tests are run on a dedicated test database.

Create a file called `application-test.yml` in `src/test/resources`.



Test configuration file in src/test/resources

A screenshot of a code editor showing the content of a file named 'application-test.yml'. The file contains eight lines of YAML configuration for a Spring application. The lines are numbered 1 through 9 on the left margin. The configuration includes settings for the server port, data source URL, driver class name, username, password, database platform, generate-ddl flag, and hibernate ddl-auto setting.

```
1  server.port: 8094
2  spring.datasource.url: jdbc:h2:mem:testdb
3  spring.datasource.driver-class-name: org.h2.Driver
4  spring.datasource.username: sa
5  spring.datasource.password: password
6  spring.jpa.database-platform: org.hibernate.dialect.H2Dialect
7  spring.jpa.generate-ddl: true
8  spring.jpa.hibernate.ddl-auto: create-drop
9
```

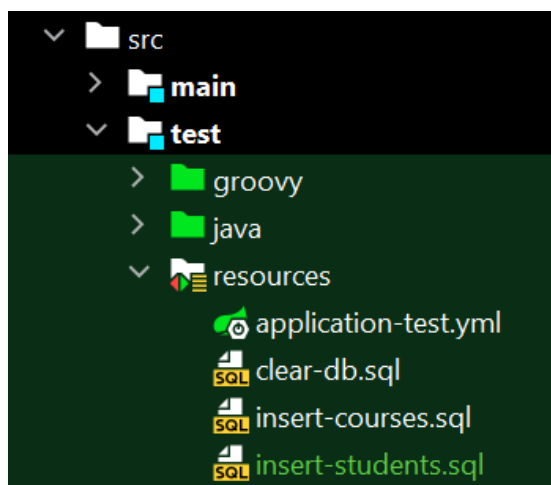
Test configuration in application-test.yml

In this example, we are using an in-memory H2 database for testing. Note how we have configured a different port for our tests to run on.

Next, we need to create some SQL scripts to clear all records from our database and insert the records needed for testing.

Answer in your log book: why have we chosen to do this via SQL, and not programmatically?

Create two files called clear-db.sql and insert-students.sql in src/test/resources. Follow the example below.



Test configuration file in src/test/resources

```
SQL clear-db.sql x
▶ | ● 🔧
1 DELETE FROM COURSE_STUDENT;
2 DELETE FROM STUDENT;
3 DELETE FROM COURSE;
```

Script to clear the database

```
SQL insert-students.sql x
▶ | ● 🔧
1 INSERT INTO STUDENT (ID, EXTERNAL_STUDENT_ID, FORENAME, SURNAME) VALUES (1, 'c7453423', 'Walter', 'White');
2 INSERT INTO STUDENT (ID, EXTERNAL_STUDENT_ID, FORENAME, SURNAME) VALUES (2, 'c3908978', 'Jesse', 'Pinkman');
3
4
```

Script to create student test data

Create a method in your `StudentServiceIntegrationTest` class to test the happy path. Follow the example below.

```
10 @SpringBootTest
11 @AutoConfigureTestDatabase
12 @ActiveProfiles("test")
13 class StudentServiceIntegrationTest extends Specification {
14
15     @Autowired
16     private StudentService studentService
17
18     @Sql(['clear-db.sql', 'insert-students.sql'])
19     def 'Testing GetStudentJson() reads a student from the database'() {
20
21         when: 'we read the student from the database'
22         def result: EntityModel<Student> = studentService.getStudentByIdJson(id: 1L)
23
24         then: 'all the attributes are fetched correctly'
25         result.content.id == 1L
26         result.content.forename == 'Walter'
27         result.content.surname == 'White'
28         result.content.externalStudentId == 'c7453423'
29     }
30 }
```

Testing the happy path when getting a student by its ID

Run the test using Gradle or the IDE.

In your log book: implement three more integration tests. Paste the code in your log book and briefly explain your approach.

Writing functional tests for the API endpoints

We are now going to practice writing functional or end-to-end tests for our RESTful API endpoints.

In your log book: reflect on the advantages and disadvantages of adding functional tests to your application's code base.

Since we are using Geb and Firefox Web Driver (also called Gecko Driver) for browser automation, we need to import some more dependencies.

Add the three dependencies below to your build.gradle

```
22 dependencies {
23     implementation 'org.springframework.boot:spring-boot-starter-web'
24     implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
25     implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
26     implementation 'org.springframework.boot:spring-boot-starter-hateoas'
27     compileOnly 'org.projectlombok:lombok'
28     implementation 'com.h2database:h2'
29     annotationProcessor 'org.projectlombok:lombok'
30     testImplementation 'org.springframework.boot:spring-boot-starter-test'
31     testImplementation 'org.spockframework:spock-core:2.2-M1-groovy-3.0'
32     testImplementation 'org.spockframework:spock-spring:2.2-M1-groovy-3.0'
33     testImplementation 'org.codehaus.groovy:groovy-all:3.0.9'
34     testImplementation 'org.codehaus.groovy.modules.http-builder:http-builder:0.7.1'
35     testImplementation 'org.gebish:geb-spock:5.1'
36     testImplementation 'org.seleniumhq.selenium:selenium-firefox-driver:3.141.59'
37     testImplementation 'org.seleniumhq.selenium:selenium-support:3.141.59'
38 }
```

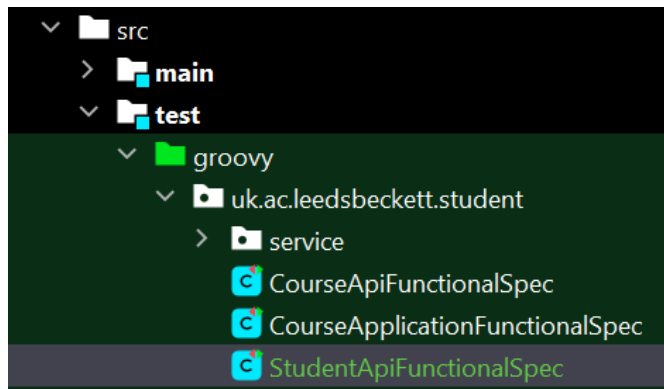
Dependencies to enable functional testing with Geb

We also need to download the Gecko Driver (version 0.30.0) and extract it to a directory on our PC.

You can get it from the following URL: <https://github.com/mozilla/geckodriver/releases>

Now we are ready to start.

Create a Groovy class called `StudentApiFunctionalSpec` in `src/test/groovy`. Follow the example below.



Location of `StudentApiFunctionalSpec`

```
StudentApiFunctionalSpec.groovy x
1  package uk.ac.leedsbeckett.student
2
3  import geb.spock.GebSpec
4  import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
5  import org.springframework.boot.test.context.SpringBootTest
6  import org.springframework.test.context.ActiveProfiles
7
8  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
9  @AutoConfigureTestDatabase
10 @ActiveProfiles('test')
11 class StudentApiFunctionalSpec extends GebSpec {
12
13 }
14
```

Annotations for `StudentApiFunctionalSpec`

Note how we are using the first annotation to tell Spring to use our test environment (not our production environment).

Declare the path and client properties following the example below. Implement a `setup()` method to initialise the client.

```
StudentApiFunctionalSpec.groovy X
1 package uk.ac.leedsbeckett.student
2
3 import geb.spock.GebSpec
4 import groovyx.net.http.RESTClient
5 import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
6 import org.springframework.boot.test.context.SpringBootTest
7 import org.springframework.http.MediaType
8 import org.springframework.test.context.ActiveProfiles
9
10 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
11 @AutoConfigureTestDatabase
12 @ActiveProfiles('test')
13 class StudentApiFunctionalSpec extends GebSpec {
14     def path = 'http://localhost:8094/api/'
15     def client
16
17     def setup() {
18         client = new RESTClient(path, MediaType.APPLICATION_JSON)
19     }
20
21 }
22
```

Setting up the REST client

Answer in your log book: what is the purpose of the setup() method and how often is it called?

We are now ready to write our first functional test.

Create a test that sends a GET request to the /api/students/{id} endpoint. Follow the example below.

```

22 @Sql(['/clear-db.sql', '/insert-students.sql'])
23 def 'Test GET a student by ID returns the correct student'() {
24
25     when: 'a GET request is sent to get a student by id'
26     def response = client.get(path: 'students/1')
27
28     then: 'the correct response is returned'
29     with(response) {
30         status == 200
31         data.id == 1
32         data.forename == 'Walter'
33         data.surname == 'White'
34         data.externalStudentId == 'c7453423'
35         data._links.containsKey 'self'
36     }
37 }

```

Test sending a GET request to `/api/student/{id}`

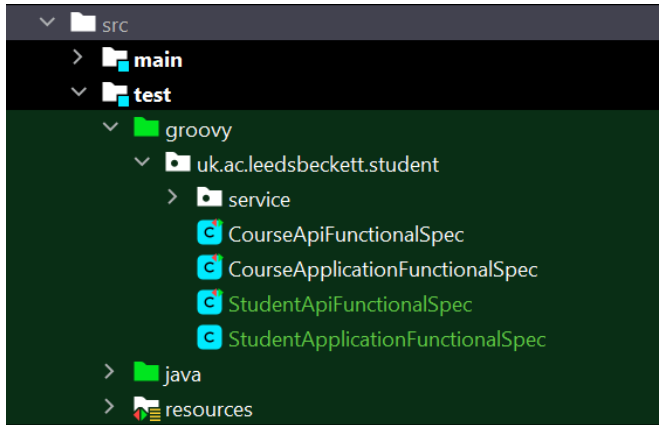
Note how the syntax is similar to our integration test. Remember however that we were testing the integration between the service, the repository and the database before. Here, we are testing the whole application, including the controller and controller mappings.

In your log book: implement three more functional tests for API endpoints. Paste the code in your log book and briefly explain your approach.

Writing functional tests for the MVC application

In this final section of the lab sheet, we are going to use the Gecko Driver we downloaded previously to simulate a user launching Firefox and navigating through the application.

Create a Groovy class called `StudentApplicationFunctionalSpec` in `src/test/groovy`. Follow the example below.



Location of StudentApplicationFunctionalSpec

```
StudentApplicationFunctionalSpec.groovy x
1  package uk.ac.leedsbeckett.student
2
3  import geb.spock.GebSpec
4  import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
5  import org.springframework.boot.test.context.SpringBootTest
6  import org.springframework.test.context.ActiveProfiles
7
8  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
9  @AutoConfigureTestDatabase
10 @ActiveProfiles('test')
11 class StudentApplicationFunctionalSpec extends GebSpec {
12
13     def setupSpec() {
14         System.setProperty('webdriver.gecko.driver', 'C:\\Program Files\\WebDrivers\\geckodriver.exe')
15     }
16
17 }
18
```

Annotations and setup for StudentApplicationFunctionalSpec

Remember to replace the second argument to `setProperty()` with the actual location where you downloaded the Gecko Driver.

Note how we are using the first annotation to tell Spring to use our test environment (not our production environment).

Follow the example below to verify that the student profile page is displayed correctly.

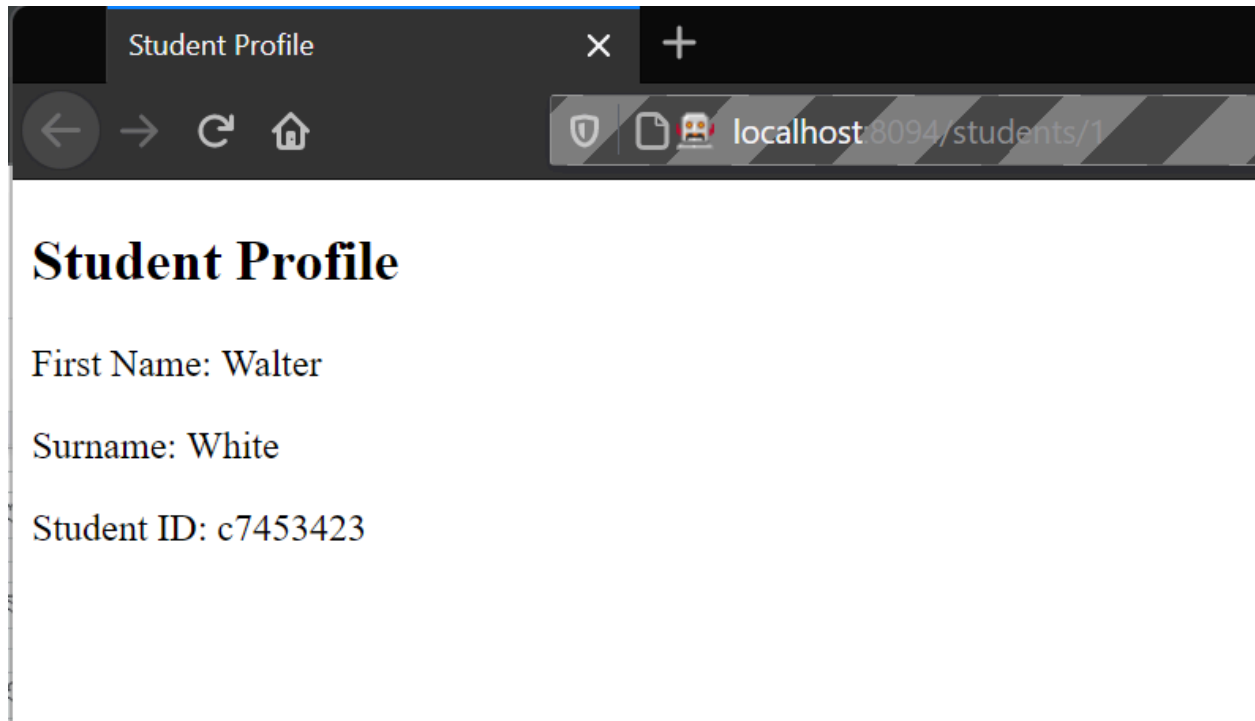
```

18 @Sql(['/clear-db.sql', '/insert-students.sql'])
19 def 'Getting the student profile'() {
20
21     when: 'I go to the student profile page'
22     browser.drive {
23         go url: 'http://localhost:8094/students/1'
24     }
25
26     then: 'I land on the right page'
27     title == 'Student Profile'
28
29     and: 'It has the correct heading'
30     $(selector: 'h2').text() == 'Student Profile'
31
32     and: 'The student details are displayed correctly'
33     $(selector: 'p').size() == 3
34     $(selector: 'p').first().text() == 'First Name: Walter'
35     $(selector: 'p')[1].text() == 'Surname: White'
36     $(selector: 'p').last().text() == 'Student ID: c7453423'
37 }

```

Testing that the student profile is displayed correctly

Run the test and note how a browser instance is opened automatically using the Gecko Web Driver.



Browser instance opened by Gecko Web Driver

Add three more functional tests to your student profile page. If necessary, implement three more features.

Document your implementation decisions and tests with screenshots **in your log book**.

Assignment

Use the skills we practiced today to add automated tests to your product. Remember to always be prepared to justify your decisions.

Conclusion

At this point you have:

- gained theoretical and practical knowledge of automated tests;
- practiced using Spock, Geb and the Gecko Web Driver to write unit, integration and functional tests;
- worked independently to design and implement your own test strategy;
- progressed with your assignment by adding a suite of tests to your code.

Well done!

Professionally developed products should always be delivered with a suite of unit, integration and functional tests. Automated tests are cheap to run and provide developers with reassurance that future changes or refactoring are not going to break the product. They are a must in large-scale engineering, as they guarantee that each specification implemented will not be overwritten by future code changes.

This concludes the taught content for SESC. We hope you have enjoyed the module and wish you every success in your future careers. Please remember to complete the module evaluation on MyBeckett to let us know how you found this module. Your comments, suggestions or even general impressions are very valuable to us and will help us make this module even better in the future.