УДК 004

Тестирование JUnit контроллера Spring Boot

Семченко Регина Викторовна
Приамурский государственный университет имени Шолом-Алейхема
Студент

Еровлев Павел Андреевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье рассмотрены возможности написания модульных тестов для REST API Spring Boot. Будут использованы библиотеки JUnit5 и Mockito.
Ключевые слова: Spring Boot, Java, JUnit

Testing JUnit Spring Boot Controller

Semchenko Regina Viktorovna
Sholom-Aleichem Priamursky State University
Student

Erovlev Pavel Andreevich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article explores the possibilities of writing unit tests for the Spring Boot REST API. JUnit5 and Mockito libraries will be used.
Keywords: Spring Boot, Java, JUnit

Модульные тесты используются для тестирования небольших модулей приложения. Модульные тесты проверяют, работает ли модуль кода должным образом.

Цель данной статьи написать модульные тесты JUnit для Spring Boot REST API.

В своей работе М.К.Ермаков, С.П.Вартанов рассмотрели вопросы проведения анализа программ интерпретируемых языков программирования [1]. С.В. Мельников провел обзор на работу с отладочным интерфейсом Java и методом модификации функциональности приложения, не изменяющий его бинарные файлы [2]. А.А.Шейн, Д.Г.Залевский, С.В. Автайкин, С.В.Карташев, С.А.Скороход разработали и описали действия программы предназначенной для автоматического создания набора классов для представления объектов модели Decode в виде нативных объектов языка Java

[3]. Н.Н. Глибовец проанализировала в своей работе особенности агентных технологий и перспективы их использования для разработки сложных многопользовательских программных систем [4]. Так же А.А. Птицын, Н.Л. Подколодный, Д.А.Григорович, С.В. Лаврюшев разработали молекулярно-биологический сервер и на его базе создали ряд информационно-вычислительных систем, для изучения регуляции экспрессии генов с использованием новейших технологий Java [5].

С начала создаем приложение Spring Boot с необходимой зависимостью. Необходима зависимость «spring-boot-starter-web» для поддержки создания REST API и зависимость «spring-boot-starter-test» для добавления библиотек тестовой среды в приложение (рис.1).

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Рисунок 1 – pom.xml

Далее создаем простые конечные точки CRUD для класса «Student DTO». Создаем класс «RestController», класс обслуживания и класс DTO с именем «Student.java» (рис.2).

```java
public class Student {

    private Integer id;
    private String name;

    public Integer getId() { return id; }

    public void setId(Integer id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }
}
```

Рисунок 2 – Student.java

Следующим делом создаем класс «RestController» с именем «StudentController.java». Этот класс будет иметь конечные точки CRUD REST для «Student DTO». Класс использует класс обслуживания «StudentService» для выполнения операции CRUD (рис.3).

```java
@RestController
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/getMapping")
    public ResponseEntity<List<Student>> getStudents() {
        List<Student> students = studentService.getStudents();
        return new ResponseEntity<>(students, HttpStatus.OK);
    }

    @PostMapping("/postMapping")
    public ResponseEntity<Student> saveStudent(@RequestBody Student student) {
        student = studentService.saveStudent(student);
        return new ResponseEntity<>(student, HttpStatus.CREATED);
    }

    @PutMapping("/putMapping")
    public ResponseEntity<Student> putExample(@RequestBody Student student) {
        student = studentService.updateStudent(student);
        return new ResponseEntity<>(student, HttpStatus.OK);
    }

    @DeleteMapping("/deleteMapping")
    public ResponseEntity<String> deleteExample(@RequestParam("student-id") String studentId) {
        String response = studentService.deleteStudent(studentId);
        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}
```

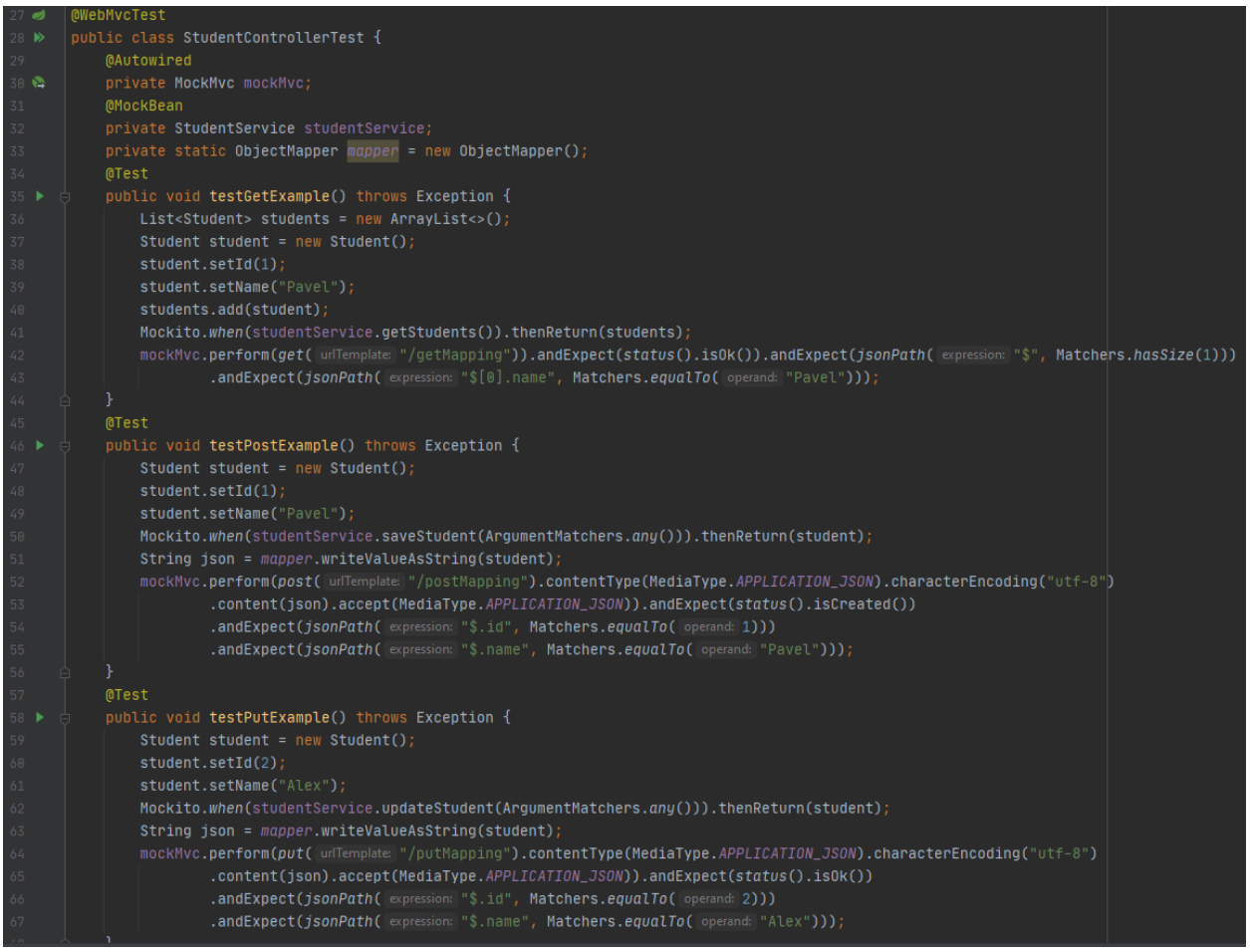
Рисунок 3 - StudentController.java

Осталось создать класс обслуживания «StudentService.java». Этот класс обеспечивает необходимые функции для уровня контроллера (рис.4).

```
@Service
public class StudentService {

    public List<Student> getStudents() {
        List<Student> students = new ArrayList<>();
        Student student = new Student();
        students.add(student);
        return students;
    }

    public Student saveStudent(Student student) {
        student.setId(1);
        student.setName("Pavel");
        return student;
    }

    public Student updateStudent(Student student) {
        student.setId(2);
        student.setName("Alex");
        return student;
    }

    public String deleteStudent(String studentId) { return "Судент удален"; }
}
```

Рисунок 4 - StudentService.java

Теперь создаем тестовый JUnit класс с именем «StudentControllerTest.java (рис.5).



```java
@WebMvcTest
public class StudentControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private StudentService studentService;
    private static ObjectMapper mapper = new ObjectMapper();
    @Test
    public void testGetExample() throws Exception {
        List<Student> students = new ArrayList<>();
        Student student = new Student();
        student.setId(1);
        student.setName("Pavel");
        students.add(student);
        Mockito.when(studentService.getStudents()).thenReturn(students);
        mockMvc.perform(get( urlTemplate: "/getMapping")).andExpect(status().isOk()).andExpect(jsonPath( expression: "$", Matchers.hasSize(1)))
                .andExpect(jsonPath( expression: "$[0].name", Matchers.equalTo( operand: "Pavel")));
    }
    @Test
    public void testPostExample() throws Exception {
        Student student = new Student();
        student.setId(1);
        student.setName("Pavel");
        Mockito.when(studentService.saveStudent(ArgumentMatchers.any())).thenReturn(student);
        String json = mapper.writeValueAsString(student);
        mockMvc.perform(post( urlTemplate: "/postMapping").contentType(MediaType.APPLICATION_JSON).characterEncoding("utf-8")
                .content(json).accept(MediaType.APPLICATION_JSON)).andExpect(status().isCreated())
                .andExpect(jsonPath( expression: "$.id", Matchers.equalTo( operand: 1)))
                .andExpect(jsonPath( expression: "$.name", Matchers.equalTo( operand: "Pavel")));
    }
    @Test
    public void testPutExample() throws Exception {
        Student student = new Student();
        student.setId(2);
        student.setName("Alex");
        Mockito.when(studentService.updateStudent(ArgumentMatchers.any())).thenReturn(student);
        String json = mapper.writeValueAsString(student);
        mockMvc.perform(put( urlTemplate: "/putMapping").contentType(MediaType.APPLICATION_JSON).characterEncoding("utf-8")
                .content(json).accept(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$.id", Matchers.equalTo( operand: 2)))
                .andExpect(jsonPath( expression: "$.name", Matchers.equalTo( operand: "Alex")));
    }
```

Рисунок 5 - StudentControllerTest.java

Основные моменты:

- @WebMvcTest - эта аннотация инициализирует конфигурации, связанные с веб-MVC, необходимые для написания тестового примера JUnit для классов контроллеров.
- MockMvc - этот класс предоставляет необходимые методы для тестирования уровня Spring MVC с помощью метода «perform()» можно тестировать различные конечные точки HTTP (GET, POST, PUT, DELETE и т. д.)
- @MockBean - эта аннотация создает фиктивные bean-компоненты в контексте приложения Spring .
- @Test - Указатель, что метод является тестовым.
- Mockito - этот класс фреймворка Mockito создает имитацию объекта.
- Jsonpath - Spring boot предоставляет встроенную поддержку JsonPath, которая полезна для проверки ответа JSON.
- MvcResult - MockMvc возвращает объект результата при вызове «andReturn()», который содержит детали ответа конкретной операции MVC.

Теперь можно запустить тестовые примеры JUnit. Если каждый тестовый пример выполняется успешно, то в консоли выведется вывод JUnit (рис.6).

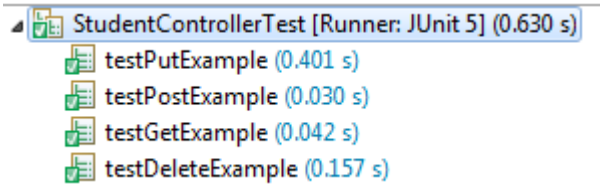


Рисунок 6 – Пример теста

В этой статье было рассмотрено как писать модульные тесты для REST API Spring Boot с помощью JUnit 5 и Mockito.

Библиографический список

1. Ермаков М.К., Вартанов С.П. Подход к проведению динамического анализа java-программ методом модификации виртуальной машины java // Научные труды Винницкого национального технологического университета. 2018. №6. С. 10-17.
2. Шейн А.А., Залевский Д.Г., Автайкин С.В., Карташев С.В.,. Скороход С.А Генератор исходного кода на языке java по описанию бортовых компонентов decode (decode java generator 0.2) // Вестник Волжского университета им. В.Н. Татищева. 2019. №3. С. 26-32.
3. Мельников С.В. Обзор и применение отладочного интерфейса java (jdi) для обратимой модификации программных продуктов // Современные проблемы науки и образования. 2018. №8. С. 8-19.
4. Глибовец Н.Н. Использование jade (java agent development environment) для разработки компьютерных систем поддержки дистанционного обучения агентного типа // Заметки по информатике и математике. 2019. №10. С. 15-20.
5. Птицын А.А., Подколодный Н.Л., Григорович Д.А., Лаврюшев С.В. Создание молекулярно-биологического сервера www с использованием новейших технологий java // Заметки по информатике и математике. 2020. №1. С. 11-20.