Интеграционные тесты

- Первый тест
- Взаимодействие с базой

Тестирование приложений на Spring Boot — неотъемлемая часть профессиональной жизни веб-разработчиков на Java. Сюда входит написание различных тестов:

- Юнит-тестов для отдельных модулей
- Интеграционных тестов, проверяющих работоспособность всего приложения

В этом уроке мы научимся создавать интеграционные тесты для наших веб-приложений на Spring Boot.

Интеграционное тестирование веб-приложений устроено сложнее, чем тестирование библиотечного кода, где мы вызываем какие-то методы и смотрим на результат.

Веб-приложения работают по сети, обрабатывая HTTP-запросы. Такое поведение придется повторять прямо в тестах или как-то имитировать. Spring Boot позволяет использовать оба подхода. Мы остановимся на подходе с подменой веб-сервера, чтобы ускорить запуск и выполнение тестов. В остальном эти тесты проверяют работу приложения от запроса до ответа, что дает очень высокую степень уверенности в том, что приложение работает.

Для работы с тестами нужно установить зависимости:

```
testImplementation(platform("org.junit:junit-bom:5.10.0"))
testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
testImplementation("org.springframework.boot:spring-boot-starter-test")
// Понадобится когда мы начнем работать с аутентификацией
testImplementation("org.springframework.security:spring-security-test")
```

Кроме классического Junit, здесь мы видим пакеты, специфичные для Spring Boot. Они дают все необходимые инструменты, чтобы мы могли писать тесты легко и эффективно.

Первый тест

Интеграционные тесты в Spring Boot связаны с маршрутами. Каждый тест — это запрс конкретный адрес для тестирования конкретного маршрута. Количество тестов для одного

маршрута может быть разным, но конкретный тест — это всегда запрос-ответ.

Начнем с примера. Предположим, что у нас есть маршрут */api/users*, который возвращает список пользователей. Тест на такой маршрут должен выполнить запрос на этот адрес. Вот как будет выглядеть структура файлов в этом случае:

```
tree src
src
├── main
│   ├── java
│   │   └── io
│   │       └── hexlet
│   │           └── spring
│   │               ├── Application.java
│   │               ├── controller
│   │               │   └── api
│   │               │       └── UsersController.java
│   │               ├── model
│   │               │   └── User.java
│   │               ├── repository
│   │               │   └── UserRepository.java
│   └── resources
│       ├── application.yml
└── test
    └── java
        └── io
            └── hexlet
                └── spring
                    └── controller
                        └── api
                            └── UsersControllerTest.java
```

Тесты Spring Boot расположены в директории *src/test/java/io/hexlet/spring*. Интеграционные тесты фактически повторяют структуру контроллеров, поэтому удобнее всего делать прямое соответствие между структурой контроллеров и тестами. В примере выше мы видим одни и те же директории. Название теста получается из названия контроллера с добавлением *Test* в название файла.

Сам тест выглядит так:

```java
package io.hexlet.spring.controller.api;

import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequest
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
public class UsersControllerTest {

    @Autowired
    private MockMvc mockMvc;

    // Технически имя тестового метода не важно
    // Лучше использовать шаблон testНазваниеМетодаКонтроллера для основного сценария
    @Test
    public void testIndex() throws Exception {
        mockMvc.perform(get("/api/users"))
                .andExpect(status().isOk());
    }
}
```

Файл тестов — это классический JUnit-класс, в котором тестовые методы помечены аннотациями `@Test`. Все остальное — это уже специфика Spring Boot. Сюда относятся аннотации `@SpringBootTest` и `@AutoConfigureMockMvc`. Во время старта тестов Spring Boot читает эти аннотации, стартует приложение и конфигурирует его в соответствие с аннотациями. Например, нам становится доступным объект `mockMvc`, через который можно выполнять HTTP-запросы к нашему приложению. Разберем по шагам:

- Метод `get("/api/users")` формирует объект запроса к указанной странице. Кроме запроса `get`, мы можем выполнить любой другой запрос
- Метод `mockMvc.perform()` выполняет сформированный запрос. На самом деле здесь не происходит HTTP-вызова — запрос передается в приложение напрямую, поэтому тесты работают быстрее, чем с реальным веб-сервером

- Метод `andExpect(status().isOk())` проверяем, что в ответ вернулся ответ *200*. По необходимости можно проверить любой другой статус

Проверка на код ответа считается одной из базовых проверок. Она показывает, что код в целом отработал ожидаемо. При этом мы не можем с уверенностью сказать, что все правильно.

Например, мы ожидаем, что в теле ответа будет JSON определенной структуры, но вдруг там ничего нет? Для контроля ответа нужно добавить проверку тела ответа. Сделать это можно множеством разных способов и библиотек, мы используем следующие:

```
testImplementation("net.javacrumbs.json-unit:json-unit-assertj:3.2.2")
```

Использование выглядит так:

```
package io.hexlet.spring.controller.api;

import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequest
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static net.javacrumbs.jsonunit.assertj.JsonAssertions.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
public class UsersControllerTest {

    @Autowired
    private MockMvc mockMvc;

    // Технически имя тестового метода не важно
    // Лучше использовать шаблон testНазваниеМетодаКонтроллера для основного сценария
    @Test
    public void testIndex() throws Exception {
        var result = mockMvc.perform(get("/api/users"))
```

```
                .andExpect(status().isOk())
                .andReturn();

        // Тело это строка, в этом случае JSON
        var body = result.getResponse().getContentAsString();
        assertThatJson(body).isArray();
        // Еще проверки
    }
}
```

Библиотека JsonUnit обладает широкими возможностями по проверке того, как устроен JSON. Подробнее с этими возможностями можно ознакомиться в официальной документации. Изучим несколько примеров:

```
import static net.javacrumbs.jsonunit.assertj.JsonAssertions.assertThatJson;
import static net.javacrumbs.jsonunit.assertj.JsonAssertions.json;

// Compares two JSON documents (note lenient parsing of expected value)
assertThatJson("{\"a\":1, \"b\":2}").isEqualTo("{b:2, a:1}");

// Objects are automatically serialized before comparison
assertThatJson(jsonObject).isEqualTo("{\n\"test\": 1\n}");

// AssertJ map assertions (numbers are converted to BigDecimals)
assertThatJson("{\"a\":1}").isObject().containsEntry("a", BigDecimal.valueOf(1));
```

И последний шаг — запуск тестов:

```
./gradlew test
# В этом выводе может быть много дополнительных строчек с логами
UsersControllerTest > testIndex() PASSED

BUILD SUCCESSFUL in 4s
```

Взаимодействие с базой

Пример теста списка пользователей не включает в себя одну важную деталь — наполнение базы данных. По умолчанию тесты используют ту базу данных, которая указана в

конфигурации. За ее наполнение отвечает программист, а не Spring Boot. Кроме наполнения базы, нам нужна еще и ее очистка.

Представьте, что мы написали тест, который создает пользователя. Если после теста мы не удалим этого пользователя, то следующий тест может завершиться с ошибкой — он не рассчитывает, что в базе уже есть такие данные. По этой причине в большинстве фреймворков каждый тест выполняется в отдельной транзакции, которая откатывается в конце теста. Таким образом достигается полная изоляция тестов друг от друга.

Можно наполнить базу данных, написав пачку SQL-запросов, но это неудобно и сложно в поддержке, особенно на больших объемах. Было бы удобнее, если бы могли автоматически создавать объекты на базе сущностей и сохранять их в базу. В Java есть специальная библиотека — Instancio.

Посмотрим на работу такого теста на примере запроса, обновляющего пользователя. Для этой операции используем маршрут */api/users/{id}*. Для выполнения запроса нам понадобится идентификатор пользователя, которого мы создадим с помощью библиотеки *Instancio*.

Для начала установим необходимые зависимости:

```
implementation("net.datafaker:datafaker:2.0.1")
implementation("org.instancio:instancio-junit:3.3.0")
```

Теперь посмотрим готовый тест, а затем разберем его:

```
package io.hexlet.spring.controller.api;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.HashMap;

import org.instancio.Instancio;
import org.instancio.Select;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
```

```java
import org.springframework.test.web.servlet.MockMvc;

import com.fasterxml.jackson.databind.ObjectMapper;

import io.hexlet.blog.model.User;
import io.hexlet.blog.repository.UserRepository;
import net.datafaker.Faker;

@SpringBootTest
@AutoConfigureMockMvc
public class UsersControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private Faker faker;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private ObjectMapper om;

    @Test
    public void testUpdate() throws Exception {
        var user = Instancio.of(User.class)
                .ignore(Select.field(User::getId))
                .supply(Select.field(User::getEmail), () -> faker.internet().emailAddress(
                .create();
        userRepository.save(user);

        var data = new HashMap<>();
        data.put("firstName", "Mike");

        var request = put("/api/users/" + user.getId())
                .contentType(MediaType.APPLICATION_JSON)
                // ObjectMapper конвертирует Map в JSON
                .content(om.writeValueAsString(data));

        mockMvc.perform(request)
                .andExpect(status().isOk());
```

```
        user = userRepository.findById(user.getId()).get();
        assertThat(user.getFirstName()).isEqualTo(("Mike"));
    }
}
```

Шаг 1. Сначала мы создаем пользователя. Instancio делает это автоматически, базируясь на полях переданной модели. По умолчанию данные создаются для всех полей, но это не всегда удобно. Во-первых, не нужно заполнять значение для идентификатора, во-вторых, `email` должен быть настоящим, поэтому здесь мы используем кастомизацию и добавляем адрес с помощью Faker:

```
var user = Instancio.of(User.class)
        .ignore(Select.field(User::getId))
        .supply(Select.field(User::getEmail), () -> faker.internet().emailAddress())
        .create();
userRepository.save(user);
```

Шаг 2. Затем мы подготавливаем запрос. Сначала формируем объект с данными, затем преобразуем их в JSON и устанавливаем соответствующий заголовок. В самом запросе формируем правильный адрес, подставляя идентификатор созданного пользователя:

```
var data = new HashMap<>();
data.put("firstName", "Mike");

var request = put("/api/users/" + user.getId())
        .contentType(MediaType.APPLICATION_JSON)
        // ObjectMapper конвертирует Map в JSON
        .content(om.writeValueAsString(data));
```

Шаг 3. Выполняем запрос и проверяем, что он действительно изменил пользователя в базе данных:

```
mockMvc.perform(request)
            .andExpect(status().isOk());

user = userRepository.findById(user.getId()).get();
assertThat(user.getFirstName()).isEqualTo(("Mike"));
```

Кроме изменения данных в базе, имеет смысл протестировать ответ, который возвращается после запроса.

Обратите внимание на важную деталь, связанную с интеграционными тестами. На протяжении урока мы писали тесты и убеждались, что приложение работает, даже не посмотрев на реализацию самого приложения. В этом и заключается суть интеграционных тестов. Нам не важно, как написано приложение внутри — мы убеждаемся только в том, что оно работает правильно. Из-за этого интеграционные тесты очень устойчивы к изменениям в коде, они меняются в основном из-за изменений API.

Дополнительные материалы

1. Документация Instancio

Далее →