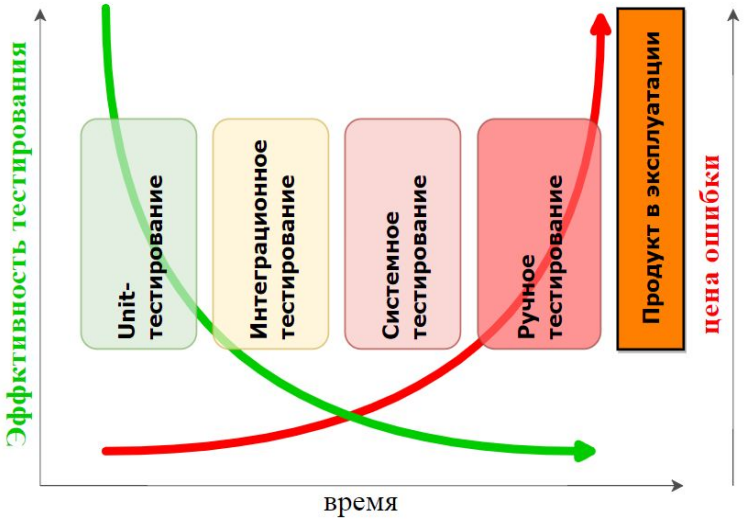
GeekBrains

Разработка интернет-магазина на Spring Framework

Тестирование в Spring

Тестирование.

Тестирование при разработке ПО



Цена ошибки растет со временем, а возможность ее устранения — наоборот, снижается. Чем раньше мы найдем и исправим ошибку в программном коде, тем дешевле это будет.

Ошибка, обнаруженная на этапе модульного тестирования, относительно легко «лечится» правкой тестируемого метода или класса. Она же, найденная на этапе интеграционного тестирования, может вызвать необходимость правки уже в нескольких модулях — возможно, мало связанных между собой. Это потребует внести больше изменений и снова все протестировать. Далее по возрастающей. Ошибка, найденная в эксплуатируемом продукте, может не исправиться вовсе. Бывает, что поверх ошибки написано уже так много кода, обходящего ее, что исправление приведет к неработоспособности объема кода, во много раз превосходящего размер ошибочного метода. Вот и выходит — «не баг, а фича» и «работает — не трогай!».

Виды тестирования

Модульное тестирование (Unit Testing) — процесс, позволяющий проверить на корректность отдельные модули исходного кода программы. Пишем тесты отдельно на каждый метод класса. При изменениях в коде программы Unit-тестирование позволяет быстро локализовать проблемный участок.

Типичные фреймворки — JUnit, TestNG.

Интеграционное тестирование (Integration Testing) — метод тестирования программного обеспечения, при котором отдельные программные модули объединяются и тестируется корректность их совместной работы. Обычно интеграционное тестирование проводится после модульного и предшествует системному.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определенные в плане тестирования, и представляет их в качестве выходных данных (и входных для последующего системного тестирования).

Системное тестирование (System Testing) — это тестирование ПО, выполняемое на полной, собранной системе, чтобы проверить соответствие системы исходным требованиям. Системное

тестирование относится к методам «черного ящика» и не требует знаний о внутреннем устройстве системы.

Инструменты автоматического тестирования

JUnit

Представим функцию **factorial**, реализацию которой необходимо протестировать:

```
public static int factorial(int n) {
   // внутреннее ее устройство нас не интересует
}
```

Для проведения Unit-тестирования необходимо:

1. Подключить тестовый фреймворк к проекту:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

2. Создать класс, содержащий методы, помеченные соответствующими аннотациями.

3. Выполнить тесты с использованием тестового фреймворка посредством IDE, выполнения Maven-цели **test** или другого средства автоматического тестирования:

```
public class MathTest {
    @Before
    public void setUp() throws Exception {
        // инициализация, выделение ресурсов
    }

    @After
    public void tearDown() throws Exception {
        // завершение, освобождение ресурсов
    }

    @Ignore
    @Test(expected = IllegalArgumentException.class)
    public void testNegative() {
        Math.factorial(-1);
    }

    @Test
    public void testNumbers() throws Exception {
        assertTrue(Math.factorial(0) == 1);
        assertTrue(Math.factorial(1) == 1);
        assertTrue(Math.factorial(5) == 120);
    }

    @Test(timeout = 100)
    public void testRandomNumbers() throws Exception {
        for (int i = 0; i < 100; i++) {
            int n = (int) (java.lang.Math.random() * 100);
            if (n > 0)
                assertTrue(Math.factorial(n - 1) * n == Math.factorial(n));
        }
    }
}
```

Методы, помеченные **@Before**, выполняются до тестов, а помеченные **@After** — после, что часто используется для инициализации окружения, выделения и освобождения ресурсов. Аннотация **@Test** помечает тестовый метод, как аргумент можно указать тип ожидаемого исключения и тайм-аут прохождения теста. Временно ненужные тесты можно отключить, используя **@Ignore**.

Рекомендуется использовать assert-методы из класса **org.junit.Assert** как более информативные относительно java-выражения **assert**, возвращающего только **java.lang.AssertionError**.

Тестирование в Spring

Модульное тестирование в Spring

Классы, составляющие приложение (POJO), должны проходить модульные тесты (JUnit или TestNG). При этом тестируемые объекты просто создаются при помощи оператора **new,** без использования DI-контейнера. Для изолированного тестирования кода применяются mock-объекты.

Так как одни компоненты приложения используют другие, а в рамках Unit-тестирования мы проверяем их работу изолированно, будем применять mock-объекты («объекты-заглушки») с управляемым поведением — чтобы подменить поведение компонентов, которые не тестируем в данный момент. Для этого используем фреймворк Mockito.

Пример модульного тестирования REST-контроллера

Допустим, есть REST-контроллер:

```
@Controller
@RequestMapping("/user")
public class UserController {
    @Autowired
    private final UserService userService;

    @RequestMapping(method = RequestMethod.GET)
    public @ResponseBody User getUser(@RequestParam(name = "name") String name)
{
        return userService.getUser(name);
    }
}
```

Создадим для него модульный тест:

```
import static org.mockito.Mockito.*;
public class UserControllerTest {
    private UserController userController;
    private User user = new User("foo","bar");

    @Before
    public void setUp() throws Exception {
        // создаем объект-заглушку
        UserService mockService = mock(UserService.class);
        userController = new UserController(mockService);

        // задаем нужное поведение объекта-заглушки
        when(mockService.getUser(user.getName())).thenReturn(user);
    }
    @Test
    public void getUserTest() throws Exception {
        User userActual = userController.getUser(user.getName());
        assertTrue(userActual != null);
        assertTrue(userActual.getName().equalsIgnoreCase(user.getName()));
    }
}
```

Здесь мы тестируем контроллер, а не сервис, которым он пользуется, поэтому при инстанциации контроллера используем объект-заглушку, реализующую интерфейс сервиса, не тестируемого здесь, и задаем его поведение. В данном случае при поиске предопределенного пользователя по имени он будет возвращен сервисом-заглушкой.

Интеграционное тестирование в Spring Boot

Интеграционные тесты позволяют проверить требования к совместной работе объектов.

Так как жизненным циклом объектов управляет контейнер DI, необходим способ его «поднять» для выполнения интеграционных тестов. Достигается это с помощью JUnit-аннотации @RunWith(..) и указания соответствующего **Runner**. Spring Boot использует SpringRunner, унаследованный от **SpringJUnit4ClassRunner**.

SpringJUnit4ClassRunner — это Spring-расширение **BlockJUnit4ClassRunner** от JUnit, которое обеспечивает функциональность Spring TestContext Framework стандартным тестам JUnit с помощью **TestContextManager** и связанных классов поддержки и аннотаций.

Для инициализации DI-контейнера необходимо указать способ инициализации контекста: конфигурационный класс или файл XML-конфигурации. Для этого предназначена аннотация **@ContextConfiguration(...)** или Spring Boot вариант — @SpringBootTest. Последняя автоматически находит конфигурационный класс приложения, если явно не указан конкретный. Такое указание может быть полезным, если тестовая конфигурация отличается от «боевой».

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class IntegrationTest {
    @Autowired
    private ApplicationContext applicationContext;

    @Test
    public void test() {
        assertTrue(applicationContext != null);
    }
}
```

Запустим данный тест и в консоли увидим типичные сообщения об инициализации контекста:

```
INFO 6496 --- [main] com.example.IntegrationTest              : Starting
IntegrationTest on ... with PID 6496 (started by program in D:\spring\test)
INFO 6496 --- [main] com.example.IntegrationTest              : No active
profile set, falling back to default profiles: default
INFO 6496 --- [main] o.s.w.c.s.GenericWebApplicationContext   : Refreshing
org.springframework.web.context.support.GenericWebApplicationContext@67080771:
startup date [...]; root of context hierarchy
...
INFO 6496 --- [main] com.example.IntegrationTest              : Started
IntegrationTest in 14.256 seconds (JVM running for 16.54)
```

SpringRunner выполняет типичную для Spring Boot инициализацию: считывает настройки из файла **application.properties** и тому подобное.

Аннотация **@SpringBootTest** может использоваться в качестве альтернативы стандартной аннотации Spring-test **@ContextConfiguration**, когда вам нужны функции Spring Boot.

Аннотация **@SpringBootTest** принимает параметр **webEnvironment**, который позволяет уточнить, как именно будут работать тесты. Возможны следующие режимы:

- **MOCK** (по умолчанию) — загружает **WebApplicationContext** и обеспечивает среду с сервлетом-заглушкой (встроенные контейнеры сервлетов не запускаются). Может использоваться совместно с **@AutoConfigureMockMvc** для тестирования приложения на основе MockMvc.
- **RANDOM_PORT** — загружает **EmbeddedWebApplicationContext** и запускает встроенные контейнеры сервлетов, которые прослушивают HTTP-запросы на случайном порту.
- **DEFINED_PORT** — загружает **EmbeddedWebApplicationContext** и запускает встроенные контейнеры сервлетов, которые прослушивают HTTP-запросы на определенном порту (если в **application.properties** не указано иное, то 8080 по умолчанию).
- **NONE** — загружает **ApplicationContext** с использованием **SpringApplication**, без сервлетов.

Пример интеграционного тестирования REST-контроллера

Для тестирования веб-сервисов Spring Boot предоставляет класс **TestRestTemplate**. У него удобный построитель HTTP-запросов, и он поддерживает маршалинг доменных типов (модели). Результат выполнения запроса может быть представлен как объект домена (модель) или обернут в объект возврата (**ResponseEntity**).

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class IntegrationTest {
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void userControllerTest() {
        ResponseEntity<User> entity =
restTemplate.getForEntity("/user?name={user}", User.class, "user1");
        assertTrue(entity.getStatusCode() == HttpStatus.OK);
    }
}
```

Здесь мы создаем класс интеграционного теста, с реальным (не mock) контроллером. С помощью метода **getForEntity** делаем запрос, получаем типизированный моделью ответ и проверяем статус возврата.

RestTemplate поддерживает широкие возможности создания запросов — например, имеется полностью конфигурируемый метод **execute**:

```
        HttpStatus status = restTemplate.execute("/", HttpMethod.GET,
                // кастомное действие при запросе
                request -> log.info(request.getHeaders().toString()),
                // кастомное действие при получении ответа
                response -> {
                    log.info(response.getHeaders().toString());
                    return response.getStatusCode();
                });
```

Mocks // заглушки

Зачастую может оказаться чрезмерно сложным или невозможным использовать реальные компоненты системы, например из сервисного слоя. В таком случае можно создать объект-заглушку:

```
    @MockBean
    private UserService userService;

    private User predefinedUser1  = new User("user1", "user1@mail.ru");

    @Before
    public void setUp() {
        given(this.userService.getUser(predefinedUser1.getName()))
            .willReturn(predefinedUser1);
    }
```

Аннотация **@MockBean** используется для добавления бинов-заглушек в Spring **ApplicationContext**. Любой существующий компонент того же типа, определенный в контексте, будет заменен заглушкой. Действие этой аннотации перекрывает «боевой» бин, возможно, созданный ранее.

Далее, в методе **setUp()**, используя **BDD Mockito**, переопределяем поведение бина-заглушки.

Добавляем в тест проверку полей объекта модели:

```
    @Test
    public void userControllerTest() {
        ResponseEntity<User> entity = restTemplate
          .getForEntity("/user?name={user}", User.class,
                        predefinedUser1.getName());

        assertTrue(entity.getStatusCode() == HttpStatus.OK);

        // проверяем корректность ответа
        User actualUser = entity.getBody();
        assertTrue(actualUser.getName().
                    equalsIgnoreCase(predefinedUser1.getName()));
    }
```

Помимо создания бинов-заглушек, замещающих основную логику приложения, можно создавать обертки вокруг реального бина. У них будет программируемое поведение, и они не нарушат основную логику мокируемого бина.

Например:

```
    @SpyBean
    private UserService userService;

    private final User user1 = new User("user1", "user1@mail.ru");

    @Test
    public void spyBeanTest() {
        // входим в тест со стороны контроллера через HTTP,
        // выполняем HTTP-запрос PUT
        restTemplate.put("/user", user1);

        // проверяем выполнение метода, интересующего нас
        verify(userService).addUser(Mockito.any(User.class));

        // исследуем переданные параметры
        ArgumentCaptor<User> captor = ArgumentCaptor.forClass(User.class);
        verify(userService).addUser(captor.capture());

        System.out.println(captor.getValue());
    }
```

Здесь мы создаем **SpyBean** («шпионящий бин») типа **UserService**. Исходный бин **UserService** продолжает выполнять свою работу, а «шпионящий бин» может, например, убедиться в факте запуска определенного метода или собрать информацию об аргументах, с которыми он выполняется.

Тестирование по слоям

Полная загрузка контекста приложения имеет оборотную сторону: инстанцирует избыточное количество бинов — явно не все из них необходимы для проведения конкретного теста. Это может занимать чрезмерно много времени. Часто для проведения тестов необходимо сконфигурировать отдельный слой приложения, например сервисный или слой персистентности.

Spring Boot Test предоставляет аннотации конфигурирования приложения для тестирования отдельного слоя. Это быстрее и требует меньше усилий, чтобы создать изолированную управляемую среду для тестируемого компонента.

Тестирование слоя персистентности

Чтобы проверить слой персистентности приложения (**Hibernate** и **Spring Data**), Spring Boot предоставляет аннотацию **@DataJpaTest**, в задачи которой входит:

- настроить in-memory базу данных для проведения тестов (драйвер БД, например H2 Database, требуется указать в зависимостях pom.xml);
- настроить Hibernate, Spring Data и DataSource;
- выполнить **@EntityScan** (отсканировать иерархию в поисках классов, отображаемых в БД);
- включить логирование SQL-запросов.

Пример:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class UserRepositoryTest {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void findByNameTest() {
        this.entityManager.persist(new User("user1", "user1@mail.ru"));
        User user = this.repository.findByName("user1");
        assertTrue(user.getName().equals("user1"));
        assertTrue(user.getMail().equals("user1@mail.ru"));
    }
}
```

Здесь бин TestEntityManager от Spring Boot, — альтернатива EntityManager для использования в тестах JPA. Он предоставляет подмножество методов EntityManager, которые применяются при написании тестов, а также вспомогательные методы: **persist**, **flush**, **find**.

Примечание. Для сторонников более низкоуровневого доступа в БД есть аналогичный модуль, подключаемый аннотацией @JdbcTest.

Тестирование слоя контроллеров

Для тестирования слоя контроллеров приложения используем аннотацию **@WebMvcTest**, которая:

- конфигурирует Spring MVC, Jackson, Gson, конвертеры;
- инстанциирует только бины контроллеров (**@Controller**, **@RestController**, **@JsonComponent**);
- конфигурирует бин **MockMvc**.

Пример:

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
public class UserControllerTest {
    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserService userService;

    @Test
    public void test() throws Exception {
        // определяем нужное поведение заглушки userService
        given(this.userService.getUser("user1"))
                .willReturn(new User("user1", "user1@mil.ru"));
        this.mvc.perform(get("/user?name={name}","user1")
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk());
    }
}
```

Тестирование слоя JSON-сериализации

Для тестирования слоя JSON используется аннотация **@JsonTest**, которая:

- конфигурирует и инжектирует **JacksonTester** (JSON-сериализатор/десериализатор + тестовые методы) в класс теста. Опционально доступен Gson-сериализатор;
- инстанциирует кастомные JSON-сериализаторы/десериализаторы, отмеченные **@JsonComponent**.

Пример:

```
@JsonTest
@RunWith(SpringRunner.class)
public class UserJsonTests {
    @Autowired
    private JacksonTester<User> json;

    @Test
    public void toJsonTest() throws IOException {
        User user = new User("user1", "user1@mail.ru");

        assertThat(json.write(user))
                .extractingJsonPathStringValue("@.name")
                .isEqualTo("user1");
    }
}
```

Здесь **JacksonTester<User>** — сериализатор/десериализатор и тестер. Используя его, мы сериализуем объект, а затем, используя **JsonPath**, анализируем соответствие полученного результата заявленным требованиям.

Дополнение

При подключении стартера **spring-boot-starter-test** в тестовой области проекта становятся доступны следующие библиотеки:

- **JUnit** — фактический стандарт модульного тестирования в мире Java.
- **Spring Test & Spring Boot Test** — утилиты интеграционного тестирования Spring-приложений.
- **AssertJ** — библиотека assert-утверждений.
- **Hamcrest** — библиотека assert-утверждений.
- **Mockito** — библиотека для создания и управления объектами-заглушками.
- **JSONassert** — библиотека assert-утверждений для JSON.

Выводы

Учитывая важность этого этапа разработки ПО, Spring предоставляет широкий спектр инструментов для модульного и интеграционного тестирования.

Мы можем использовать mock-объекты, чтобы изолировать отдельные бины или помещать все приложение или его часть в управляемую среду. Или использовать объекты-шпионы, чтобы автоматизированно контролировать, правильно ли взаимодействуют компоненты приложения.

Изолированное тестирование отдельных слоев приложения позволяет более узко очертить круг проблемных компонентов и выполнять тесты быстрее.

Практическое задание

1. Провести модульное тестирование компонентов курсового проекта.
2. Провести интеграционное тестирование компонентов курсового проекта.

Дополнительные материалы

1. Тестирование микросервисов.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Википедия — https://ru.wikipedia.org/wiki/Тестирование_программного_обеспечения.
2. Официальная документация — Testing. Spring Boot features.
3. Инженерный блог Spring.