

Spring Test



1

ОСНОВНОЙ БЛОК

Введение

- Весеннее тестирование



Проблема

Когда мы писали unit-тесты, мы создавали объекты тестируемых классов и вызывали нужные методы, затем сравнивали результат. Такой подход не работает в *Spring-приложениях*, т.к. за создание их ранение объектов отвечает *Spring Context*.

Как получить нужный бин из Spring Context для тестирования?



Весеннее тестирование

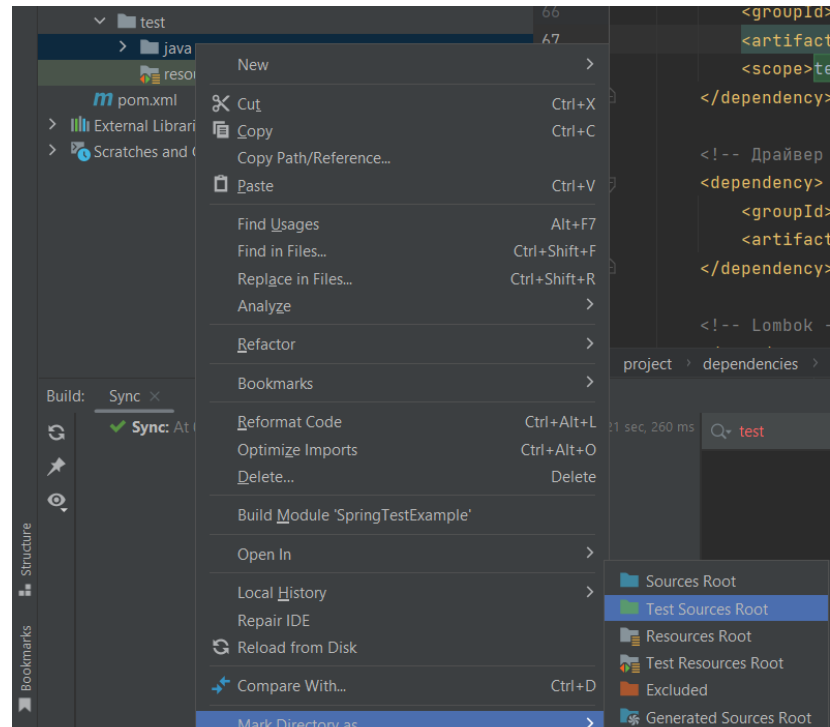
Добавление тестов

1 Добавьте зависимость *spring-boot-starter-test* в *pom.xml* Spring Boot приложения.

2 В каталоге *src* проекта, если он был создан по прототипу *maven*, обычно папки *main* и *test*. Если папка *test* отсутствует, её нужно создать.

2.1 В папке *test* создаём подпапку *java* для исходных кодов тестов, помечаем её как *Test Sources Root*.

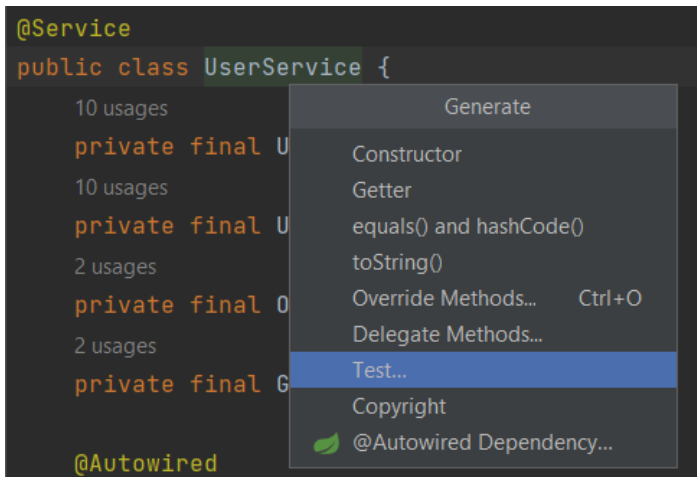
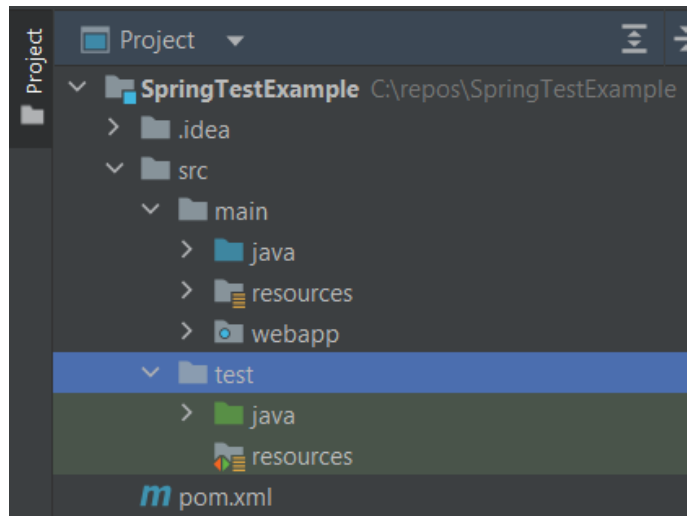
2.2 Аналогичным образом подготавливаем папку *resources* для тестовых ресурсов (файлов, настроек и т.д.) , помечаем её как *Test Resources Root*.



Весеннее тестирование

Добавление тестов

3 Структура папок проекта будет выглядеть так ->

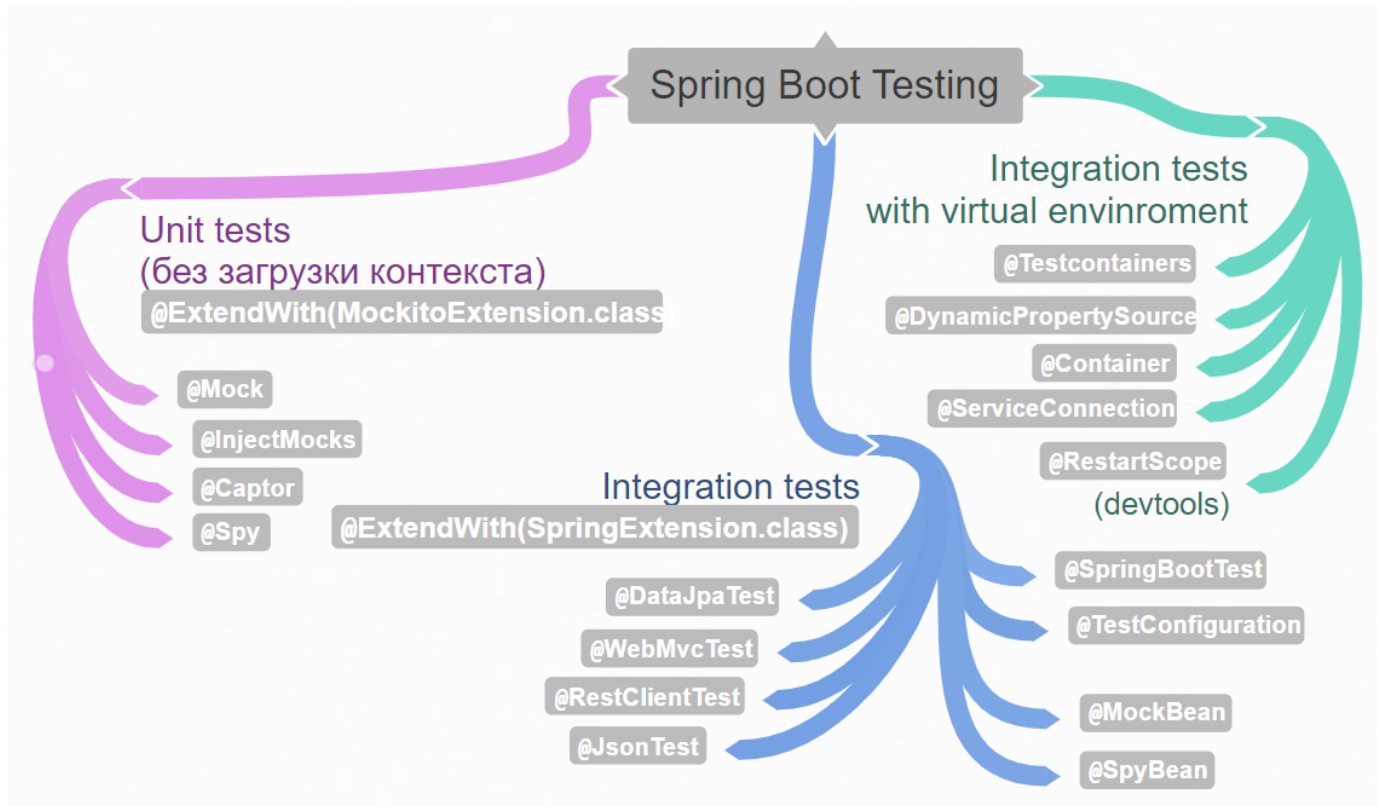


4 Для создания теста можно создать соответствующие *package* и классы. Либо в классе, для которого хотите написать тест, нажать ПКМ -> Generate -> Test... Затем выбрать тестируемые методы и прочие настройки.

Весеннее тестирование

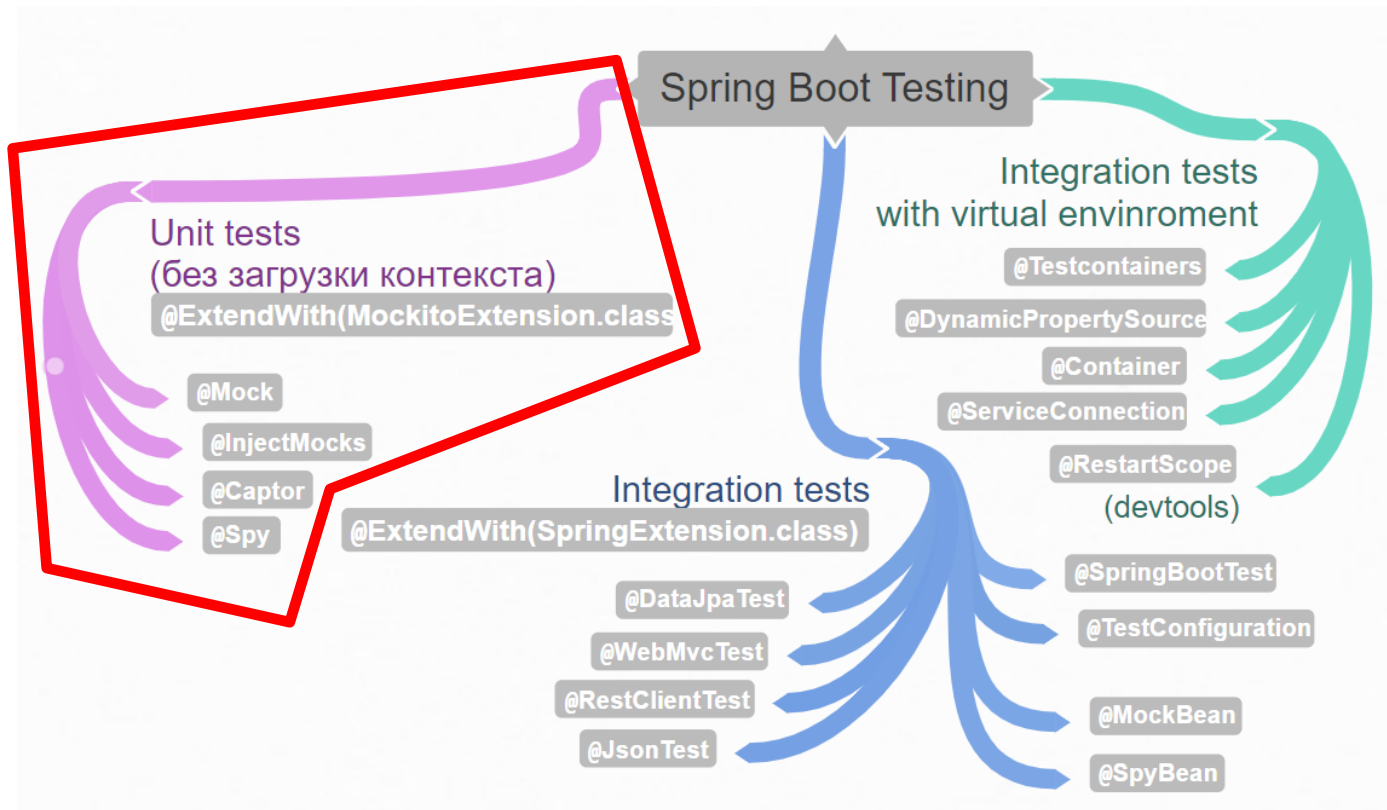
Подходы к тестированию

При тестировании *Spring Boot* приложения можно выделить три основных подхода



Весеннее тестирование

Unit-тесты



Весеннее тестирование

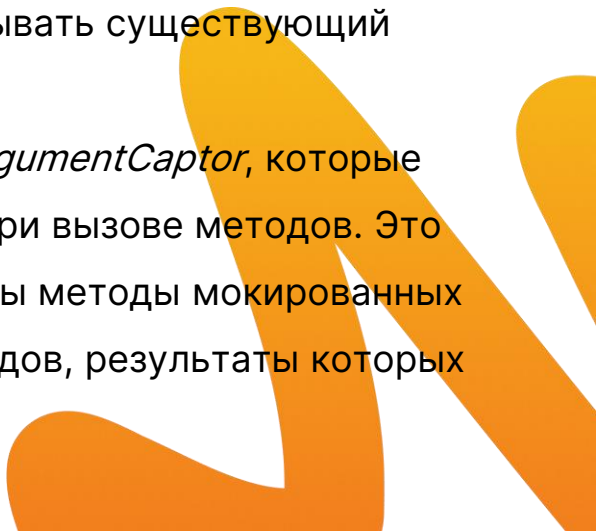
Unit-тесты

Unit-тесты не предполагают использования контекста Spring, поэтому выполняются быстрее других видов тестов. Тестовый класс должен быть помечен аннотацией **@ExtendWith** (*MockitoExtension.class*).

@Mock – помечаем поле тестового класса, чтобы создать мок класса (mock), от которого зависит тестируемый класс.

@Spy – аналогично для создания шпиона. Шпион позволяет вызывать существующий функционал класса, от которого зависит тестируемый.

@Captor – используется для удобного создания экземпляров *ArgumentCaptor*, которые предназначены для захвата аргументов, передаваемых в моки при вызове методов. Это позволяет проверять, с какими именно значениями были вызваны методы мокированных объектов, что особенно полезно при написании тестов для методов, результаты которых сложно прямо проверить или которые возвращают *void*.

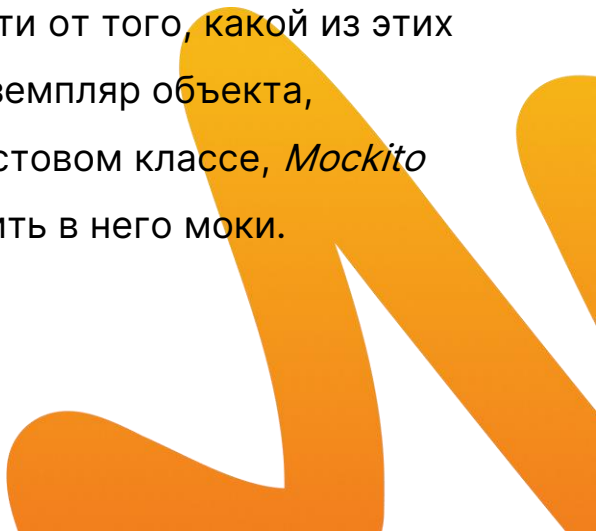


Весеннее тестирование

Unit-тесты

@InjectMocks - используется для автоматического внедрения моков или шпионов в тестируемый объект. Это очень удобный способ инициализации тестируемого объекта с автоматическим внедрением зависимостей (без контекста Spring!).

При использовании *@InjectMocks*, *Mockito* пытается внедрить моки, созданные с помощью аннотаций *@Mock* или *@Spy*, в тестируемый объект. Внедрение происходит через конструктор, сеттеры или непосредственно в поля, в зависимости от того, какой из этих способов доступен и подходит для конкретного случая. Если экземпляр объекта, помеченного аннотацией *@InjectMocks*, не был создан явно в тестовом классе, *Mockito* автоматически создает экземпляр этого класса, пытаясь внедрить в него моки.



Весеннее тестирование

Пример unit-теста

```
class SomeServiceTest {  
    @Mock  
    private Dependency dependency;  
    @InjectMocks  
    private SomeService someService;  
    @Captor  
    private ArgumentCaptor<SomeType> captor;  
    @Test  
    void testSomeMethod() {  
        someService.doSomething();  
  
        // Проверка, что метод dependency.someMethod() был вызван с правильным аргументом  
        verify(dependency).someMethod(captor.capture());  
        SomeType capturedArgument = captor.getValue();  
  
        // Теперь можно проверить свойства capturedArgument  
        assertEquals(expectedValue, capturedArgument.getSomeProperty());  
    }  
}
```

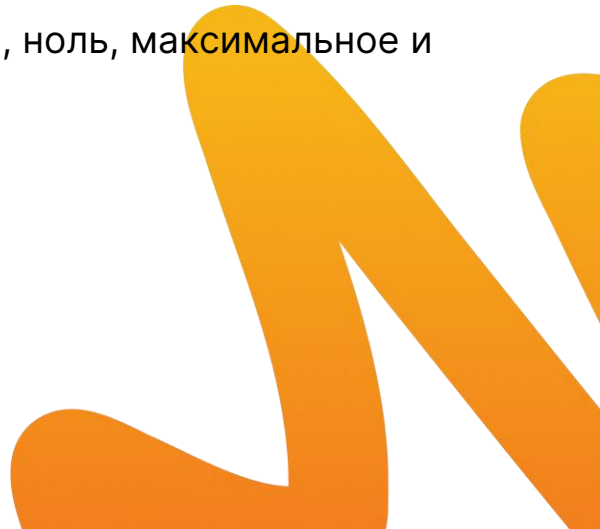
Весеннее тестирование

Тесты доменной модели и сервисов

1 Для тестирования классов доменной модели и сервисов используем unit-тесты как наиболее быстрые и изолированные. Тесты пишем до или сразу после написания классов.

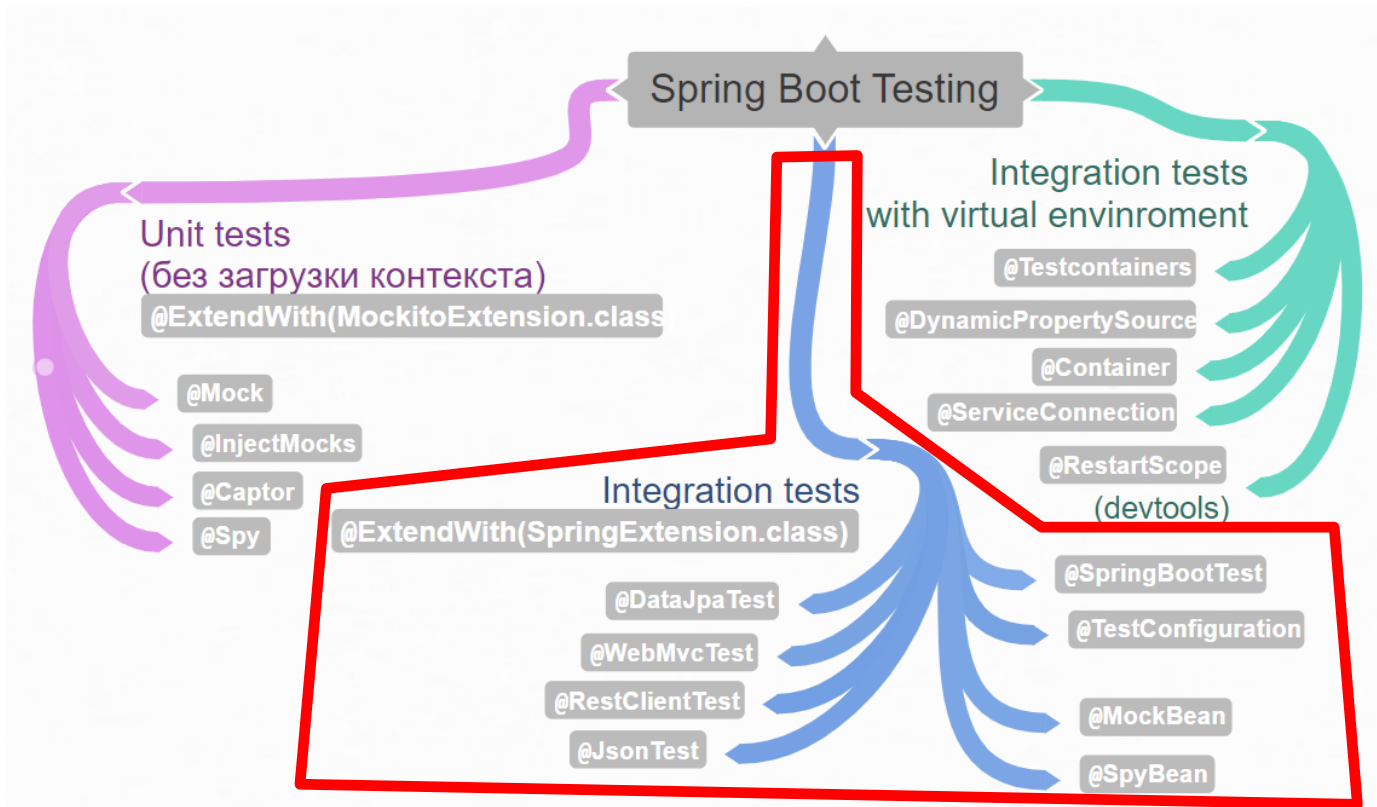
2 У классов должны быть протестированы:

- все публичные методы (включая разные ветки условных операторов в коде).
- пограничные условия (null вместо аргументов ссылочного типа, пустые значения, для числовых типов – положительные и отрицательные значения, ноль, максимальное и минимальное значения).
- все явно выбрасываемые в коде исключения (через *throw*).



Весеннее тестирование

Интеграционные тесты



Весеннее тестирование

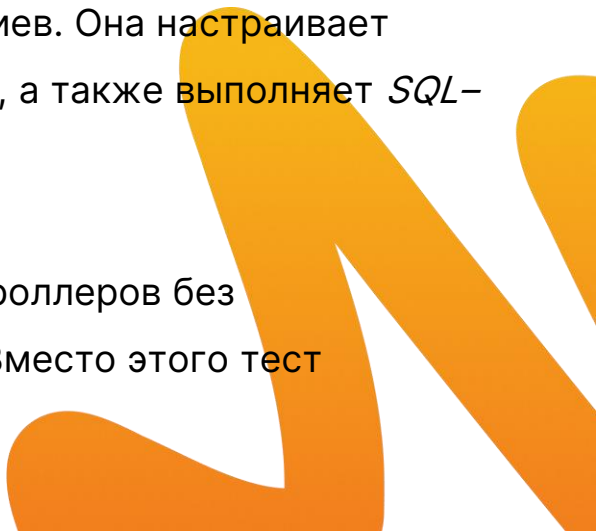
Интеграционные тесты

Интеграционные тесты предполагают вовлечение нескольких модулей приложения и проверку сквозного функционала.

@SpringBootTest – главная аннотация для тестирования в *Spring Boot*. Она загружает полный контекст приложения, имитируя поведение приложения в рабочей среде.

@DataJpaTest – используется для тестирования *JPA* репозиториев. Она настраивает встроенную БД, настраивает *Hibernate*, *Spring Data* и *DataSource*, а также выполняет *SQL*-скрипты по умолчанию.

@WebMvcTest – особенно полезна для тестирования *MVC* контроллеров без необходимости запускать полное HTTP-серверное окружение. Вместо этого тест сосредоточен только на слое *MVC*.



Весеннее тестирование

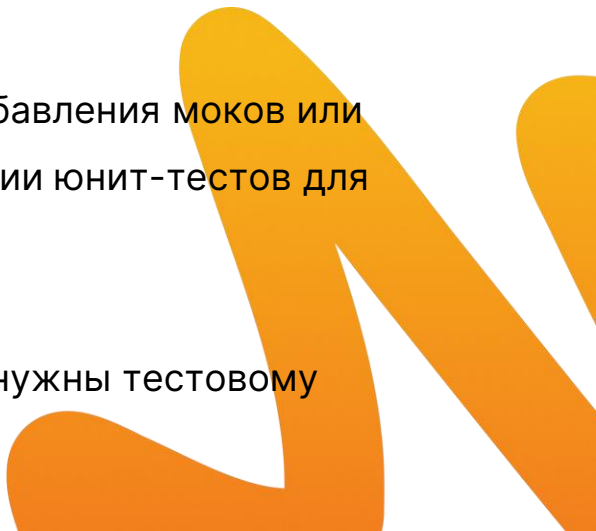
Интеграционные тесты

@RestClientTest – предназначена для тестирования *REST*-клиентов. Она настраивает *Jackson/Gson*, *@JsonComponent*, *Encoder/Decoder* и *@RestClient*.

@JsonTest – используется для тестирования *JSON* сериализации и десериализации. Автоматически настраивает *Jackson* и *Gson*.

@MockBean и **@SpyBean** – предоставляют возможности для добавления моков или шпионов в *Spring ApplicationContext*. Очень полезны при написании юнит-тестов для изолирования тестируемых компонентов.

@TestConfiguration – помечает классы конфигурации, которые нужны тестовому контексту.



Весеннее тестирование

Тесты репозитиев



1 Для тестирования репозитиев обычно используют In-memory БД, чтобы не привязываться к конкретной реализации СУБД. Для этого нужно добавить в *pom.xml* соответствующую зависимость: <https://mvnrepository.com/artifact/com.h2database/h2>

2 Над классом тестов репозитория нужно указать аннотацию **@DataJpaTest**. А также аннотацию для автоконфигурации подключения к БД:

@AutoConfigureTestDatabase (*connection = EmbeddedDatabaseConnection.H2*)

3 Создайте поле, в которое будет внедрён бин тестируемого репозитория с помощью *@Autowired*. Здесь допустимо делать внедрение на уровне поля.

4 Создать тестовые методы в классе и пометить их нужными аннотациями (*@Test*, *@ParametrizedTest* и др.).

5 Т.к. большая часть функционала Вашего репозитория заложена в интерфейсе *JpaRepository*, тестировать нужно только те методы, которые действительно используются в Вашем приложении, а также добавленные Вами в репозиторий.

Весеннее тестирование

Тесты репозитиев



6 Для наполнения БД *H2* тестовыми данными можно использовать тестируемый репозиторий. Но если хочется полной чистоты эксперимента, можно воспользоваться бином **TestEntityManager**, который является аналогом *EntityManager* для тестов.

```
@DataJpaTest
@AutoConfigureTestDatabase(connection = EmbeddedDatabaseConnection.H2)
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here
}
```

Весеннее тестирование

Тесты MVC-контроллеров



1 Указать над классом теста аннотацию *@WebMvcTest* с тестируемым контроллером в качестве параметра. Это создаст только тот компонент Spring MVC, который необходим для тестирования данного контроллера. *@WebMvcTest* автоматически настраивает инфраструктуру *Spring MVC*, включая конфигурацию *DispatcherServlet*.

Для автоматической настройки нужных бинов над классом тестов добавляют *@AutoConfigureMockMvc(addFilters = false)*.

2 Внедрить бины, от которых зависит контроллер, в тестовый класс (поля с *@Autowired*), либо сделать шпионов (поля с *@SpyBean*) или моки (поля с *@MockBean*). Обычно контроллер зависит от одного или нескольких сервисов, поэтому достаточно сделать моки этих сервисов.

3 Для выполнения запросов к контроллеру и проверки ответов используется утилита *MockMvc*, которая позволяет тестировать контроллеры без запуска полноценного сервера. Внедряется в тестовый класс с помощью *@Autowired*.

Пример теста MVC-контроллера

```
@Controller
public class WelcomeController {
    @GetMapping("/welcome")
    public String welcome(Model model) {
        model.addAttribute("message", "Hello, World!");
        return "welcome"; // Имя представления
    }
}
```

```
@WebMvcTest(WelcomeController.class)
public class WelcomeControllerTest {
    @Autowired private MockMvc mockMvc;
    @Test
    public void shouldReturnWelcomeView() throws Exception {
        this.mockMvc.perform(MockMvcRequestBuilders.get("/welcome"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.view().name("welcome"))
            .andExpect(MockMvcResultMatchers.model().attributeExists("message"))
            .andExpect(MockMvcResultMatchers.model().attribute("message", "Hello, World!"));
    }
}
```

Весеннее тестирование

Тесты REST-контроллеров



Тестирование REST-контроллеров выполняется тем же способом. При проверке результата (JSON-объекта, который вернул контроллер) используется подобный код

```
response.andExpect(MockMvcResultMatchers.status().isCreated())
    .andExpect(MockMvcResultMatchers.jsonPath("$.name", CoreMatchers.is(myDto.getName())))
    .andExpect(MockMvcResultMatchers.jsonPath("$.type", CoreMatchers.is(myDto.getType())));
```

Хотя ничто не мешает собрать весь JSON в dto и проверять содержимое его полей с помощью обычных assert-методов.



Пример теста Rest-контроллера

```
@RestController
public class GreetingController {

    @GetMapping("/greeting")
    public String greeting() {
        return "Hello, World!";
    }
}
```

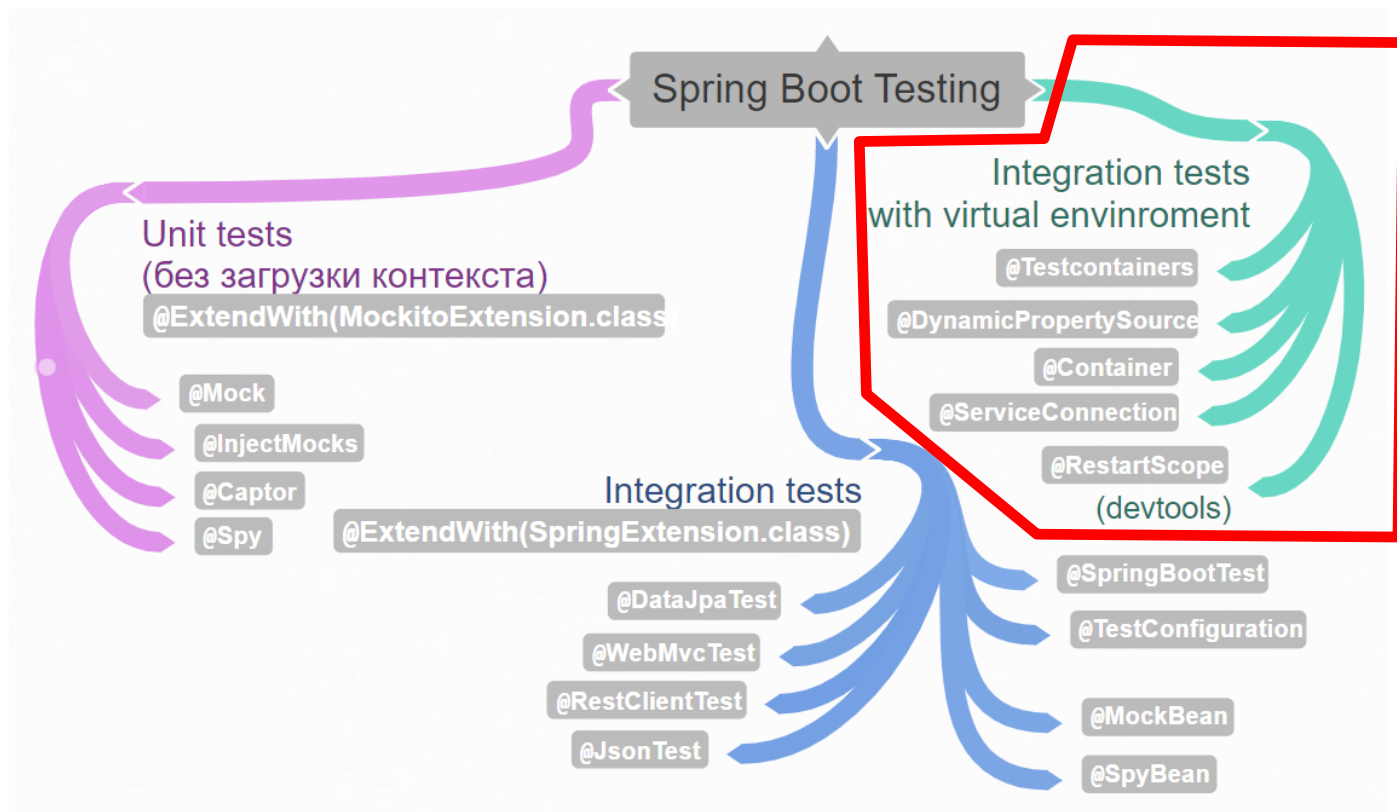
```
@WebMvcTest(GreetingController.class)
public class GreetingControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(MockMvcRequestBuilders.get("/greeting"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.content().string("Hello, World!"));
    }
}
```

Весеннее тестирование

Интеграционные тесты в виртуальной среде



Весеннее тестирование

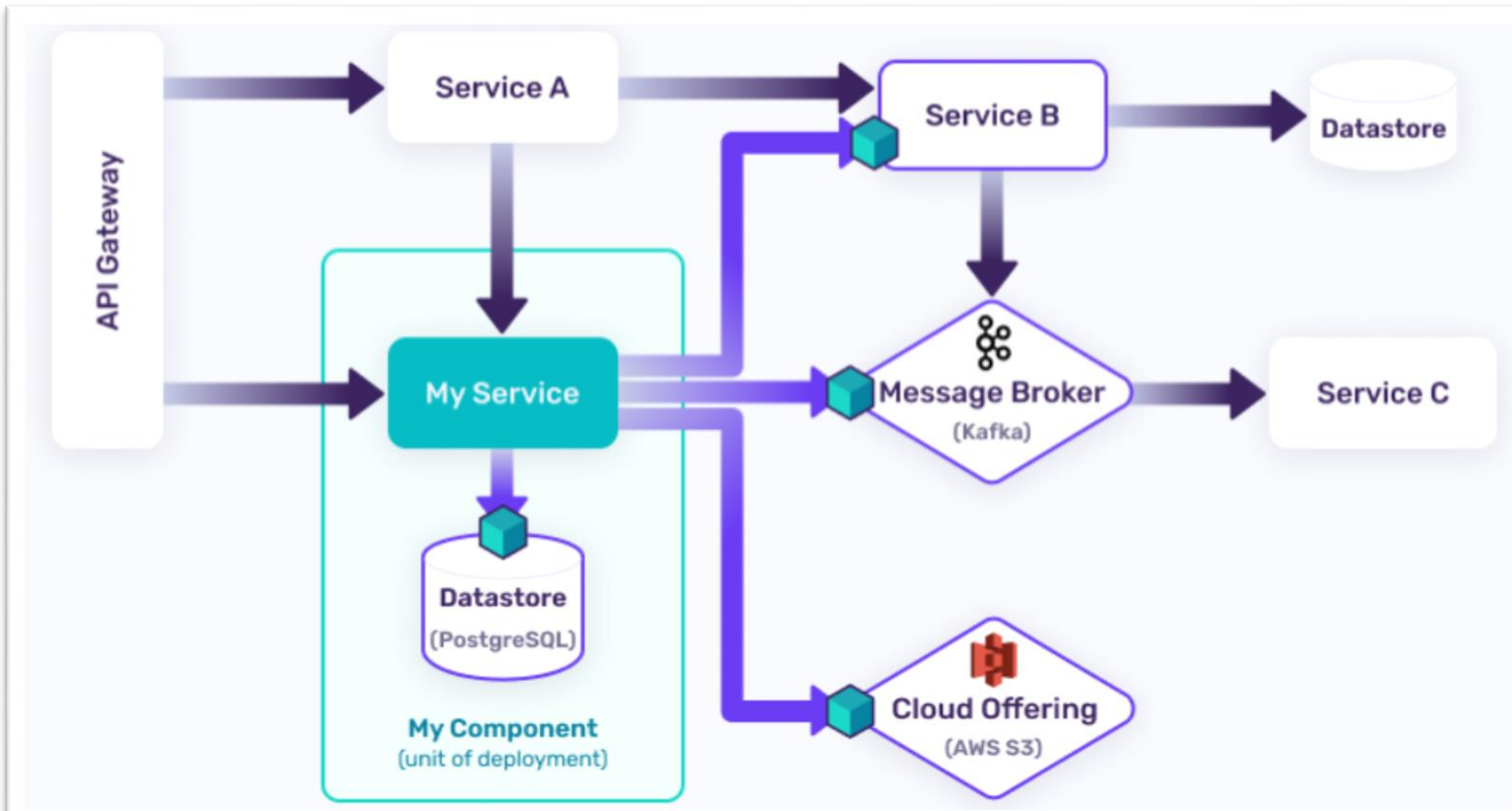
Интеграционные тесты в виртуальной среде

Такие тесты нужны для тестирования сложного взаимодействия частей приложения. Например, нескольких модулей монолитного приложения, специфики взаимодействия с СУБД (сервис -> процедуры БД) или нескольких микросервисов в связке (сервис1 -> брокер сообщений -> сервис2).



Весеннее тестирование

Интеграционные тесты в виртуальной среде



Весеннее тестирование

Интеграционные тесты в виртуальной среде

Для развёртывания виртуальной среды используется технология **Test Containers**

<https://testcontainers.com/guides/testing-spring-boot-rest-api-using-testcontainers/>

1 Скачайте и установите (под администратором) *Docker Desktop*

<https://www.docker.com/products/docker-desktop/>

Будет работать только с ОС и процессорами, поддерживающими виртуализацию.

2 Добавьте в *pom.xml* зависимости

<https://mvnrepository.com/artifact/org.testcontainers/junit-jupiter>

<https://mvnrepository.com/artifact/org.testcontainers/postgresql>

3 Создайте скрипт *Flyway* или *Liquibase* для заполнения БД. Менее правильный, но рабочий вариант – создать файл *schema.sql* в каталоге *src/main/resources* и добавить в *application.properties* настройку *spring.sql.init.mode=always*.

Весеннее тестирование

Интеграционные тесты в виртуальной среде

4 Создайте тестовый класс и укажите над ним аннотацию

@SpringBootTest(classes = Application.class), где *Application.class* – имя главного класса

Вашего приложения. При тестировании контроллеров можно дополнительно задать порт сервера:

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)

Тест будет выполняться путем запуска всего приложения на случайном доступном порту (можно указать фиксированный порт). Для получения значения порта создайте поле в тестовом классе:

```
@LocalServerPort  
private Integer port;
```

5 Также над классом укажите аннотацию *@Testcontainers*.

Весеннее тестирование

Интеграционные тесты в виртуальной среде

6 В статическом поле создайте экземпляр тестового [контейнера](#). Например, *PostgreSQLContainer*. Такое поле должно быть помечено аннотацией *@Container*.

```
@Container
```

```
@ServiceConnection
```

```
static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:latest");
```

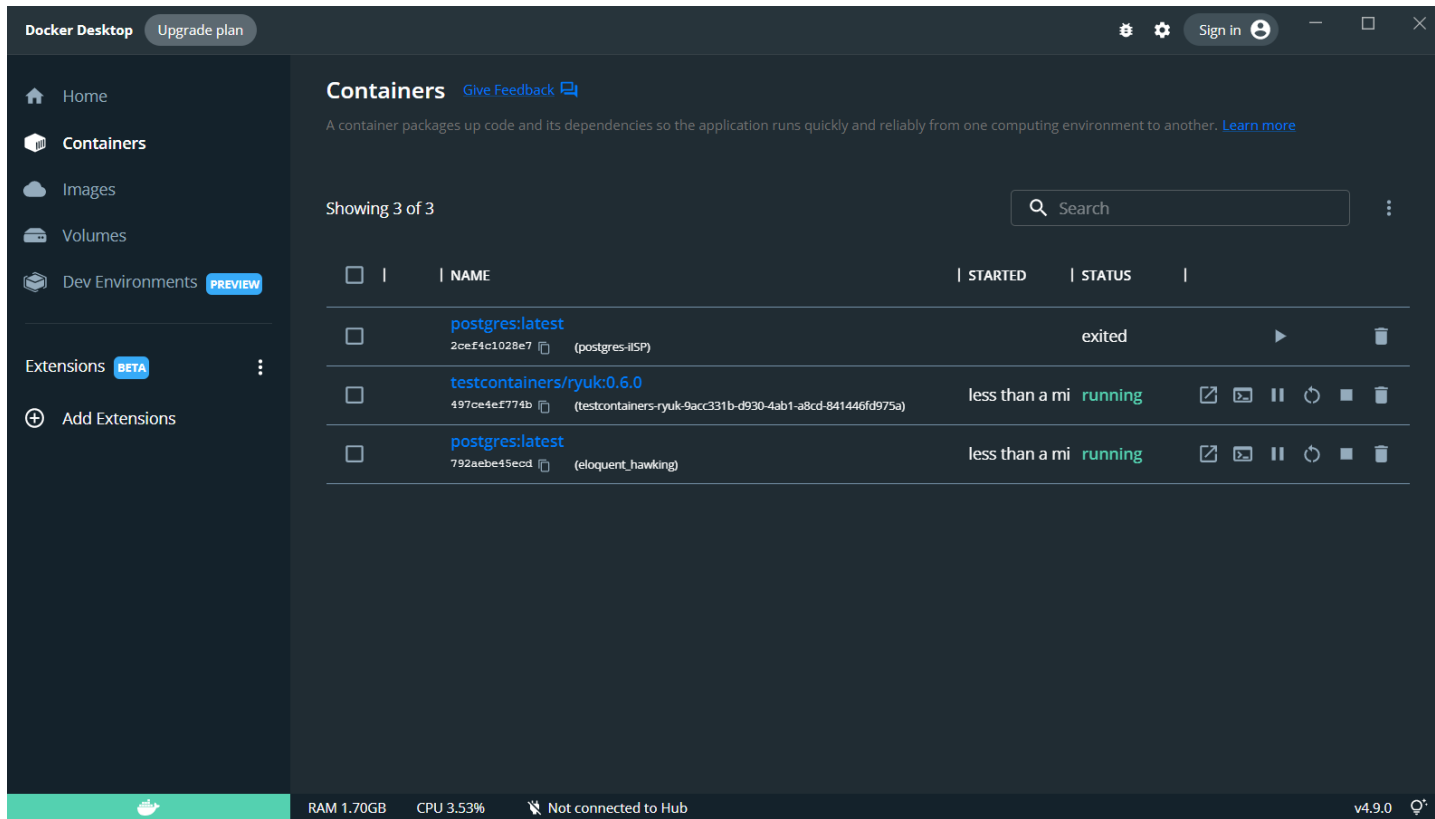
7 Контейнер Postgres запускайте в методе, помеченном *@BeforeAll*, а завершение – в методе с *@AfterAll*. База данных Postgres запускается через порт 5432 внутри контейнера и сопоставляется со случайным доступным портом на хосте.

8 Напишите необходимые тесты.

Весеннее тестирование

Интеграционные тесты в виртуальной среде

Для запуска
потребуется
иметь
установленный и
запущенный
Docker. Во время
выполнения теста
в *Docker* будет
отображать
используемый
контейнер.




Весеннее тестирование

Интеграционные тесты в виртуальной среде

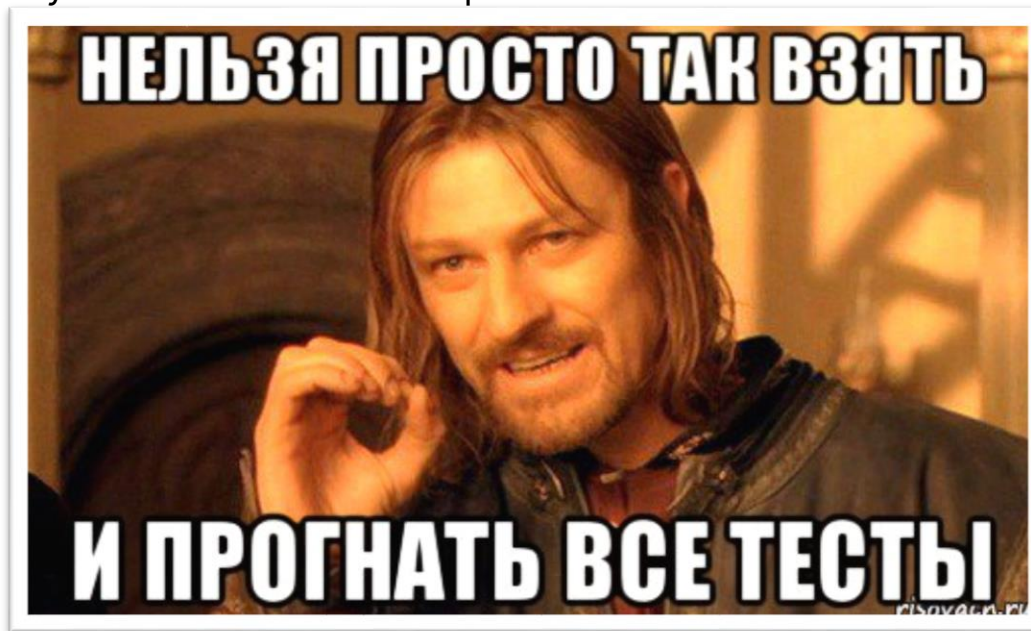
Для проведения локального дебага приложения можно запустить всё окружение в *Docker* с помощью *Test Containers*. Для этого всего лишь в папке `test/java` создать дополнительную точку входа в приложение

```
public class TestMyApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.from(Application::main)  
            .with(MyContainersConfiguration.class)  
            .run(args);  
    }  
}
```



Проблема

Сложившиеся деловые практики показывают, что хороший показатель покрытия кода – 80% и более. К этому показателю стоит стремиться.



Но как оценить, сколько кода было покрыто тестами?

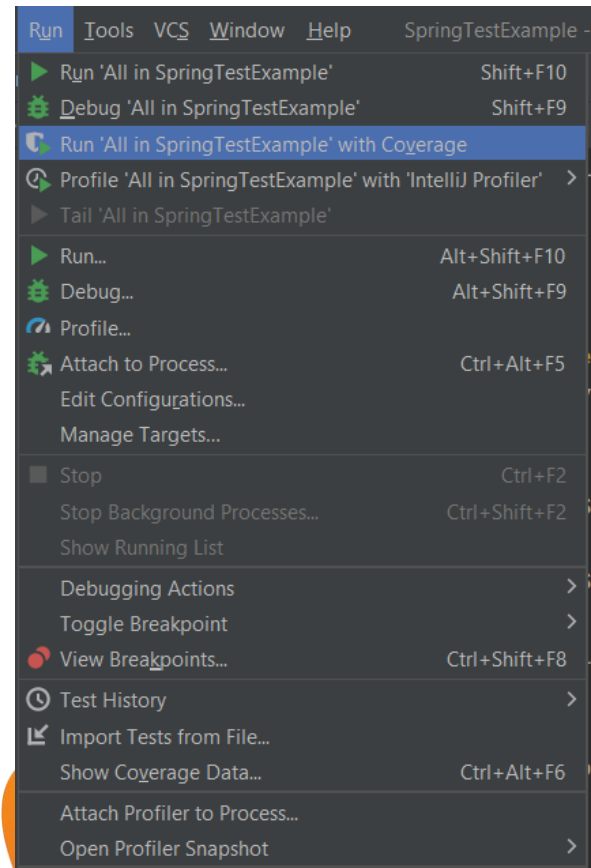
Весеннее тестирование

Покрытие тестами

Наиболее простой способ получить полный отчет по оценке тестового покрытия Java проекта – это использовать **coverage runner**, встроенный в *IntelliJ IDEA*.

1 Запустите все тесты. Например, в структуре проекта ПКМ -> Run All Tests.

2 Когда все тесты выполнены, в верхнем меню IDE выберите Run -> Run ... with coverage



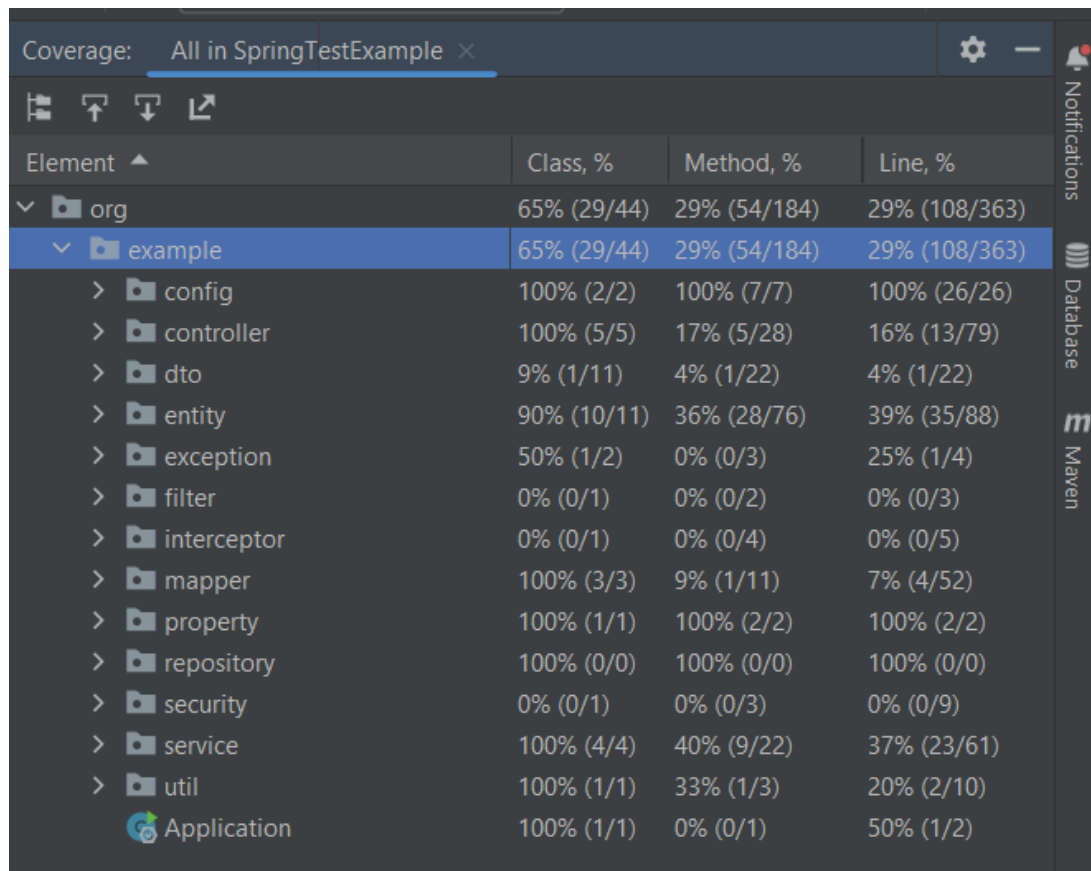
Весеннее тестирование

Покрытие тестами

После выполнения тестов IDE отобразит окно с результатами оценки покрытия.

Можно открыть каждый из пакетов, перейти в классы и увидеть, какие строки не были покрыты тестами.

При необходимости можно сделать выгрузку результата в файл.



The screenshot shows a code coverage report window titled 'Coverage: All in SpringTestExample'. The report is organized into a table with columns for 'Element', 'Class, %', 'Method, %', and 'Line, %'. The 'org' package is expanded, showing the 'example' package selected. Below it, a list of sub-packages is shown with their respective coverage statistics.

Element	Class, %	Method, %	Line, %
org	65% (29/44)	29% (54/184)	29% (108/363)
example	65% (29/44)	29% (54/184)	29% (108/363)
config	100% (2/2)	100% (7/7)	100% (26/26)
controller	100% (5/5)	17% (5/28)	16% (13/79)
dto	9% (1/11)	4% (1/22)	4% (1/22)
entity	90% (10/11)	36% (28/76)	39% (35/88)
exception	50% (1/2)	0% (0/3)	25% (1/4)
filter	0% (0/1)	0% (0/2)	0% (0/3)
interceptor	0% (0/1)	0% (0/4)	0% (0/5)
mapper	100% (3/3)	9% (1/11)	7% (4/52)
property	100% (1/1)	100% (2/2)	100% (2/2)
repository	100% (0/0)	100% (0/0)	100% (0/0)
security	0% (0/1)	0% (0/3)	0% (0/9)
service	100% (4/4)	40% (9/22)	37% (23/61)
util	100% (1/1)	33% (1/3)	20% (2/10)
Application	100% (1/1)	0% (0/1)	50% (1/2)

Весеннее тестирование

Покрытие тестами

Существуют и другие инструменты для оценки покрытия тестами. Сравнительная таблица [здесь](#).

Если будете прибегать к использованию сторонних библиотек в проекте, рекомендуется использовать **JaCoCo**.

Небольшой tutorial по подключению и использованию

<https://www.youtube.com/watch?v=LzSp9eKrWMw>



ЗАКЛЮЧЕНИЕ

