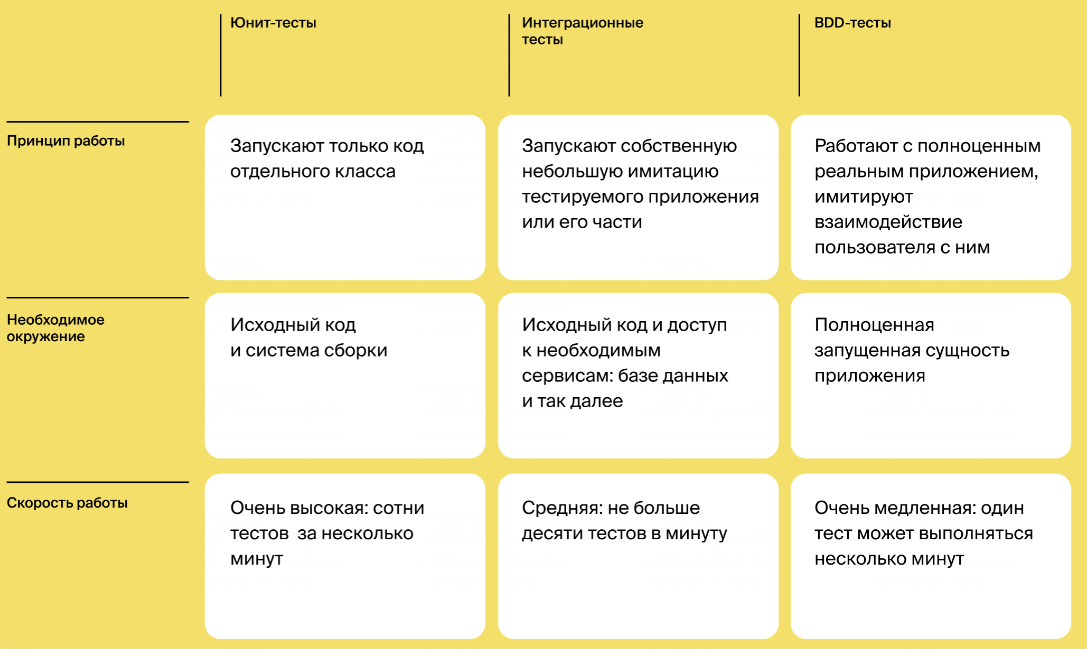
**Тестирование программы** – это исследование, при котором происходит проверка соответствия между ожидаемым и реальным поведением программы, а также выявление или подтверждение каких-либо её свойств. Требования к программе разделяются на:

* **Функциональные** – отвечают на вопрос, **что** должна делать программа.
* **Нефункциональные** – отвечают на вопрос, **как** программа должна выполняться.

Сценарии для тестирования называют **тест-кейсами**. Они состоят из трех частей:

1. **Подготовка** – определение входных параметров и условий.
2. **Исполнение** – определение процедуры тестирования.
3. **Проверка** – сравнение ожидаемого результата с полученным.



**Юнит тесты** используются для тестирования отдельных классов и методов. Они полностью автоматизированы. Используется локальная *JVM*, а связанные сущности мокируются. В зоне ответственности разработчика.

**Интеграционные тесты** используются для тестирования взаимодействия нескольких классов между собой или взаимодействия классов приложения с внешним окружением (БД). Например, проверка *API* с помощью *http*-запросов, способность программы подключения к БД и корректность выполнения запросов. Они могут быть полностью автоматизированы. Используется локальная *JVM* и только связанное окружение. В зоне ответственности разработчика, если это возможно (*Spring*). Помечаются суффиксом ***IT***.

***E2E* тестирование** (***end-to-end*** или ***BDD*-тесты**) используются для тестирования сценария работы пользователя. Например, через пользовательский интерфейс. Могут быть как автоматизированными так и ручными.

**Среда**, или **окружение** – это сервер или группа серверов, на котором находится копия приложения. **Стенд** – это конкретное окружение.

Как правило, для работы над продуктом используются разные окружения:

* **Среда разработки**, или ***dev*** – используется исключительно для разработки.
* **Тестовая среда**, или ***test*** – здесь продукт проходит сложное тестирование, например, интеграционное (когда объединяют несколько программ/сервисов вместе).
* **Продуктовая среда**, или ***prod*** – боевая среда для клиентов.

При подготовке тестового класса для нескольких тестов может возникнуть нежелательная зависимость между тестами, поэтому тестовое окружение лучше готовить под каждый тест отдельно. Для этого можно использовать следующие аннотации:

* ***@BeforeEach*** перед нестатическим методом означает, что метод будет запускаться всякий раз перед любым тестом внутри класса.
* ***@BeforeAll*** перед статическим методом означает, что метод будет запускаться перед всеми тестами.
* ***@AfterEach*** перед нестатическим методом означает, что метод будет запускаться всякий раз после каждого теста.
* ***@AfterAll*** перед статическим методом означает, что метод будет запускаться после окончания всех тестов.

Все четыре аннотации могут содержаться в одном тестовом классе.

Аннотация ***@Nested*** позволяет группировать тесты во вложенных классах. Вложенные классы не позволяют использовать аннотации *@BeforeAll* и *@AfterAll*. Подробнее можно почитать [тут](https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-nested-tests/).

При тестировании рекомендуется сократить количество тестов и оставить только самые необходимые. Для этого выделяют:

* **граничные значения** – это параметры, при переходе через которые поведение программы меняется.
* **классы эквивалентности** – это параметры, при вводе которых программа ведет себя одинаково.

Однако, покрыть тестами весь код масштабных проектов на 100% никогда не получится, поэтому используется комбинация метрик:

* **Метрика** **покрытия кода** показывает какой процент строк кода исполняется при запуске всех тестов. Оно еще называется **тестовым покрытием**. Распространенная практика – устанавливать порог тестового покрытия бизнес-логики в 80%.
* **Метрика** **покрытия требований** показывает процент требований, проверенных набором тестов.

Для просмотра покрытия в *IDEA* нужно запустить тесты ***with Coverage***.

***JUnit*** – один из самых популярных фреймворков (библиотек) для тестирования на *Java*. Для его использования нужно подключить к проекту саму библиотеку, создать метод и пометить его аннотацией ***@Test***. Аннотация ***@DisplayName*** позволяет изменить отображаемое имя теста.

*import org.junit.jupiter.api.Assertions;*

*import org.junit.jupiter.api.Test;*

*public class DiscountCalculatorTest {*

*DiscountCalculator discountCalculator = new DiscountCalculator();*

*@DisplayName(“test name”)*

*@Test*

*public void shouldGiveNoDiscountForValue999() {*

*// Подготовка*

*int buySum = 999;*

*int expectedSum = 999;*

*// Исполнение*

*int resultSum = discountCalculator.sumAfterDiscount(buySum);*

*// Проверка*

*Assertions.assertEquals(expectedSum, resultSum); } }*

*class DiscountCalculator {*

*public int sumAfterDiscount(int sum) {*

*if (sum < 1000) { return sum;*

*} else { return (int) (sum \* 0.98); } } }*

Еще один элемент *JUnit* – это множество статических методов класса ***Assertions***.

* Метод ***assertNull(Object)*** проверяет, что значение является *null*.
* Метод ***assertNotNull(Object)*** проверяет, что значение является не *null*.
* Метод ***assertTrue(boolean)*** проверяет, что значение является *true*.
* Метод ***assertFalse(boolean)*** проверяет, что значение является *false*.
* Метод ***assertEquals(Object expected, Object actual, [String message])*** сравнивает ожидаемый и фактический результат и выводит сообщение (опционально). Для корректного сравнения в классах объектов должен быть корректно переопределен метод *equals()*. Нельзя переопределить *equals()* для массивов.
* Метод ***assertArrayEquals (Array[] expected, Array[] actual, [String message])*** сравнивает ожидаемый и фактический результат и выводит сообщение (опционально) для массивов.
* Метод ***assertThrows(ErrorClass, Executable)*** проверяет выброс исключения.

*import org.junit.jupiter.api.function.Executable;*

*class ExceptionTest {*

*@Test*

*void shouldThrowException() {*

*// после исполнения блока ошибка попадёт в переменную exception*

*final ArithmeticException exception = assertThrows(*

*ArithmeticException.class, // класс ошибки*

*new Executable() { // создание и переопределение экземпляра класса Executable*

*@Override*

*public void execute() {*

*// здесь блок кода, который хотим проверить*

*int number = 10 / 0; } }); // при делении на 0 ожидаем ArithmeticException*

Интерфейс *Executable* можно записать с помощью лямбда-функции:

*() -> int number = 10 / 0; // при делении на 0 ожидаем ArithmeticException*

Для передачи в тест набора параметров можно использовать поток ***Stream<Arguments>***. Для этого нужно указать аннотации ***@MethodSource****(“имя\_метода\_возвращающего\_поток”)* и ***@ParametrizedTest***.

Пример использования билдера в параметризованных тестах:

*private static Stream<Arguments> invalidUsers() {*

*return Stream.of( Arguments.of(User.builder()*

*.name("Test name")*

*.login("ValidTestLogin")*

*.birthday(LocalDate.of(1964, 6, 11))*

*.build(),*

*"Электронная почта не может быть пустой и должна содержать символ @"),*

*Arguments.of(User.builder()*

*.email("")*

*.name("Test name")*

*.login("ValidTestLogin")*

*.birthday(LocalDate.of(1964, 6, 11))*

*.build(),*

*"Электронная почта не может быть пустой и должна содержать символ @")); }*

*@ParameterizedTest*

*@MethodSource("invalidUsers")*

*public void shouldNotAddFilmWithNotValidFields(User user, String message) {*

*ValidationException exception = assertThrows(ValidationException.class, () -> {controller.add(user);});*

*assertEquals(message, exception.getMessage());*

*assertEquals(controller.getAll().size(), 0); }*

**Мок** – простой объект-заглушка, к которому можно обращаться и либо получать предопределенные ответы, либо запланированный проброс исключения, либо вызывать реальные методы. Он позволяет не только изолировать тестируемый метод/класс, но и зафиксировать взаимодействия, которые были с ними осуществлены. Внедрить мок можно, например, наследовавшись от подменяемого объекта и переопределив нужные методы:

*class MockAuthorService extends AuthorService {*

*@Override*

*public String getAuthorDescription(int authorId) throws SQLException {*

*return "знаменитый русский писатель"; } }*

*@Test*

*void testCreateBookDescription() {*

*BookService bookService = new BookService();*

*AuthorService mockAuthorService = new MockAuthorService();*

*bookService.setAuthorService(mockAuthorService);*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, знаменитый русский писатель",*

*bookDescription);}*

***Mockito*** – один из самых популярных фреймворков (библиотек) для мок-тестирования на *Java*. Он уже включена в стартер ***spring-boot-starter-test***. Перепишем пример выше:

*@Test*

*void testCreateBookDescriptionWithMock() throws SQLException {*

*BookService bookService = new BookService();*

*AuthorService mockAuthorService = Mockito.mock(AuthorService.class); // класс для мокирования*

*bookService.setAuthorService(mockAuthorService);*

*Mockito // логика уже внутри самого теста*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenReturn("знаменитый русский писатель");*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, знаменитый русский писатель",*

*bookDescription); }*

Аннотация ***@Mock*** позволяет создать мок на весь тестовый класс. Аннотация ***@InjectMocks*** означает, что для аннотируемого объекта *Mockito* внедрит моки во все поля, где это возможно. Работают только в комплекте с аннотацией ***@ExtendWith(MockitoExtension.class)*** для корректной генерации мока.

*@ExtendWith(MockitoExtension.class)*

*public class BookServiceTestByAnnotations {*

*@Mock*

*AuthorService mockAuthorService; // мокируемый сервис*

*@InjectMocks*

*BookService bookService; // тестируемый класс*

*@Test*

*void testCreateBookDescriptionWithMockito() {*

*Mockito*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenReturn("великий русский писатель");*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, великий русский писатель",*

*bookDescription); } }*

Вспомогательный класс ***ReflectionTestUtils*** позволяет получать доступ к приватным полям и методам в тестовых целях. Он помогает обойти *dependency injection* через аннотации *@Autowired*.

*@Service*

*public class OrderService {*

*@Autowired*

*CustomerService customerService;*

*@Autowired*

*BookServiceInternal bookService;*

*@Autowired*

*OrderDao orderDao; }*

*@Test*

*void testSaveOrder() {*

*OrderService orderService = new OrderService();*

*CustomerService customerService = Mockito.mock(CustomerService.class);*

*BookService bookService = Mockito.mock(BookService.class);*

*OrderDao orderDao = Mockito.mock(OrderDao.class);*

*ReflectionTestUtils.setField(orderService, "customerService", customerService);*

*ReflectionTestUtils.setField(orderService, "bookService", bookService);*

*ReflectionTestUtils.setField(orderService, "orderDao", orderDao);*

*// проверки }*

Метод ***Mockito.when()*** позволяет определить условие, на которое будет реагировать мок. Например, вызывается определенный метод, мокируемого класса, с заданным значением аргументов. Кроме конкретных значений аргументов, можно использовать статические функции *Mockito* из семейства ***anyType()*** – *anyInt()*, *anyString()* и т.д. Такие проверки на значения аргументов называются ***argument matcher*** (**сопоставители аргументов**). Одновременно использовать матчеры и конкретные значения нельзя.

Метод ***.thenReturn()*** позволяет определить, что именно должен делать мок.

Метод ***.thenAnswer()*** позволяет использовать поступившие данные для формирования ответа. Объект *invocationOnMock* содержит в себе аргументы, с которыми был вызван метод *getAuthorDescription*.

Метод ***invocationOnMock.getArgument()*** возвращает один конкретный аргумент вызова. Первым параметром указывается индекс аргумента, начиная с нуля, а вторым – его тип данных.

*Mockito*

*.when(mockAuthorService.getAuthorDescription(anyInt()))*

*.thenAnswer(invocationOnMock -> {*

*int authorId = invocationOnMock.getArgument(0, Integer.class);*

*if (authorId % 2 == 0) {*

*return "великий русский писатель"; // чётные id у русских авторов*

*} else {*

*return "великий английский писатель"; } }); // нечётные id у английских авторов*

Метод ***.thenThrow()*** позволяет выбросить определенное исключение. Само исключение передается параметром. Метод ***doThrow().*** позволяет выбросить определенное исключение, если мокируемый метод возвращает *void*.

*Mockito*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenThrow(new DataNotAvailableException("Ошибка при доступе к базе"));*

Метод ***Mockito.verify()*** позволяет проверить содержимое мока. Например, проверим, что у мока *orderDao* метод *saveOrder()* был вызван ровно один раз с указанными аргументами.

*Mockito.verify(orderDao, times(1))*

*.saveOrder(2, "адрес", 5, 1, deliveryDate);*

Метод ***Mockito.verifyNoInteractions()*** позволяет проверить, что во время работы теста не происходило никаких вызовов к мокам из списка.

Метод ***Mockito.verifyNoMoreInteractions()*** позволяет проверить, что во время работы теста не происходило никаких вызовов к мокам, не входящим в список.

*Mockito.verifyNoMoreInteractions(customerService, bookService, orderDao);*

Условие ***Mockito.times(int)*** задает ожидаемое количество вызовов. Также можно использовать методы ***atLeast()*** или ***atMost()***, задающие минимальное или максимальное допустимое количество вызовов.

Условие ***Mockito.never()*** проверяет, что вызовов с такими аргументами не было.

Метод ***inOder()*** принимает коллекцию моков и возвращает экземпляр класса ***InOrder***. Он позволяет проверять последовательность вызовов этих моков с помощью метода ***verify()***.

*InOrder inOrder = inOrder(paymentRepository, notificationService);*

*inOrder.verify(paymentRepository).save(payment);*

*inOrder.verify(notificationService).notifySupplier(from);*

*inOrder.verify(notificationService).notifyConsumer(to);*

Аннотация ***@Captor*** позволяет создать класс-ловушку, которая зафиксирует состояние переданного в нее объекта с помощью метода ***capture()***. Метод ***getValue()*** позволяет получить этот объект.

*@Captor*

*private ArgumentCaptor<User> userArgumentCaptor;*

*verify(userRepository).save(userArgumentCaptor.capture());*

*User savedUser= userArgumentCaptor.getValue();*

Аннотация ***@MockitoSettings(strictness = Strictness.LENIENT)*** позволяет подавить ошибки о лишних неиспользуемых моках в *when()*.

Подробнее о методах *Mockito* можно почитать [тут](https://site.mockito.org/).

Аннотация ***@SpringJUnitConfig( { PersistenceConfig.class, MockitoExtension.class} )*** уже включает в себя аннотацию ***@ExtendWith*** и создает *Spring*-контекст. Обычно используется в чистом *Spring Framework*.

Автоматическое внедрение зависимостей по умолчанию не работает в тестах. Аннотация ***@RequiredArgsConstructor(onConstructor\_ = @Autowired)*** указывает, что конструктор, созданный через *Lombok*, сможет получать зависимости через механизм *Autowired*.

Аннотация ***@TestPropertySource*** позволяет задать настройки (или файл с настройками) для подключения к тестовой БД. Имеет более высокий приоритет, чем *application.properties*.

*@Transactional*

*@RequiredArgsConstructor(onConstructor\_ = @Autowired)*

*@TestPropertySource(properties = { "db.name=test"}) // переопределение только имени БД*

*@SpringJUnitConfig( { PersistenceConfig.class, UserServiceImpl.class})*

*class UserServiceImplTest { … }*

Задать настройки тестовой БД можно также в *application.properties* с помощью профилей жизненного цикла *Maven*. Например,

spring.config.activate.on-profile=ci,test  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.url=jdbc:h2:mem:shareit  
spring.datasource.username=test  
spring.datasource.password=test

Для тестирования, кроме *JUnit*, можно использовать библиотеку ***Harmcrest***. Подробнее [тут](https://hamcrest.org/JavaHamcrest/).

Аннотация ***@AutoConfigureTestDatabase*** создает тестовую БД вместе основной.

Аннотация ***@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER\_EACH\_TEST\_METHOD)*** очищает бины и БД после каждого теста.

Аннотация ***@SpringBootTest*** уже включает в себя аннотации ***@SpringJUnitConfig*** и ***@TestPropertySource***. Она позволяет запустить целиком все приложение при тестировании.

*@Transactional*

*@SpringBootTest(*

*properties = "db.name=test",*

*webEnvironment = SpringBootTest.WebEnvironment.NONE)*

*@RequiredArgsConstructor(onConstructor\_ = @Autowired)*

*class UserServiceImplTest { … }*

Аннотация ***@Transactional*** перед классом означает, что каждый тест будет запускаться в отдельной транзакции и откатываться в его конце. Таким образом в БД не будут попадать тестовые данные.

Аннотация ***@Rollback(false)*** отключает откат транзакции по окончанию теста. В этом случае все изменения сохраняются. Может использоваться как на уровне класса, так и на уровне теста.

Рассмотрим **интеграционное тестирование** *Spring MVC* (**контроллеров**).

1. В чистом *Spring Framework*. Экземпляр класса ***ObjectMapper*** используется для сериализации и десериализации *JSON*. Экземпляр класса ***MockMvc*** имитирует выполнение *http*-запроса по определенному *URL* и проверяет корректность ответа. Для его создания можно использовать один из двух статических методов класса ***MockMvcBuilders***:

* ***standaloneSetup*** – объект создается сам по себе, без *Spring*-контекста.

*@ExtendWith(MockitoExtension.class)*

*class UserControllerIT {*

*@Mock*

*private UserService userService;*

*@InjectMocks*

*private UserController controller;*

*private final ObjectMapper mapper = new ObjectMapper();*

*private MockMvc mvc;*

*private UserDto userDto;*

*@BeforeEach*

*void setUp() {*

*mvc = MockMvcBuilders*

*.standaloneSetup(controller)*

*.setControllerAdvice(ExceptionHandler.class)*

*.build();*

*userDto = new UserDto( 1L,*

*"john.doe@mail.com",*

*"John",*

*"Doe",*

*"2022.07.03 19:55:00",*

*UserState.ACTIVE); }*

*@Test*

*void saveNewUser() throws Exception {*

*when(userService.saveUser(any()))*

*.thenReturn(userDto);*

*mvc.perform(post("/users")*

*.content(mapper.writeValueAsString(userDto))*

*.characterEncoding(StandardCharsets.UTF\_8)*

*.contentType(MediaType.APPLICATION\_JSON)*

*.accept(MediaType.APPLICATION\_JSON))*

*.andExpect(status().isOk())*

*.andExpect(jsonPath("$.id", is(userDto.getId()), Long.class))*

*.andExpect(jsonPath("$.firstName", is(userDto.getFirstName())))*

*.andExpect(jsonPath("$.lastName", is(userDto.getLastName())))*

*.andExpect(jsonPath("$.email", is(userDto.getEmail()))); } }*

* ***webAppContextSetup*** – создает объект на основе контекста. Например, по умолчанию *MockMvc* ничего не знает об обработчике ошибок. Аннотация *@SpingJUnitWebConfig* создаст контекст, основанный на указанных в ней бинах.

*@SpringJUnitWebConfig({ UserController.class, TestConfig.class, WebConfig.class})*

*class UserControllerITWithContext {*

*private final ObjectMapper mapper = new ObjectMapper();*

*private final UserService userService;*

*private MockMvc mvc;*

*private UserDto userDto;*

*@Autowired*

*UserControllerTestWithContext(UserService userService) {*

*this.userService = userService; }*

*@BeforeEach*

*void setUp(WebApplicationContext wac) {*

*mvc = MockMvcBuilders*

*.webAppContextSetup(wac)*

*.build();*

*userDto = new UserDto( 1L,*

*"john.doe@mail.com",*

*"John",*

*"Doe",*

*"2022.07.03 19:55:00",*

*UserState.ACTIVE); }*

*@Test*

*void saveNewUser() throws Exception {*

*when(userService.saveUser(any()))*

*.thenReturn(userDto);*

*mvc.perform(post("/users")*

*.content(mapper.writeValueAsString(userDto))*

*.characterEncoding(StandardCharsets.UTF\_8)*

*.contentType(MediaType.APPLICATION\_JSON)*

*.accept(MediaType.APPLICATION\_JSON))*

*.andExpect(status().isOk())*

*.andExpect(jsonPath("$.id", is(userDto.getId()), Long.class))*

*.andExpect(jsonPath("$.firstName", is(userDto.getFirstName())))*

*.andExpect(jsonPath("$.lastName", is(userDto.getLastName())))*

*.andExpect(jsonPath("$.email", is(userDto.getEmail()))); } }*

1. В *Spring Boot*. Аннотация ***@WebMvcTest*** инициирует для теста контекст, состоящий только из тех бинов, которые требуются для корректной работы тестируемых контроллеров (в том числе *MockMvc* и *ObjectMapper*). Опционально список контроллеров может передаваться в качестве параметров.

Аннотация ***@MockBean*** автоматически создаст и внедрит мок-объект аннотируемого класса.

Перепишем пример выше с использованием преимуществ *Spring Boot*:

*@WebMvcTest(controllers = UserController.class)*

*class UserControllerITWithContext {*

*@Autowired*

*private ObjectMapper mapper;*

*@MockBean*

*private UserService userService;*

*@Autowired*

*private MockMvc mvc;*

*private UserDto userDto = new UserDto( 1L,*

*"john.doe@mail.com",*

*"John",*

*"Doe",*

*"2022.07.03 19:55:00",*

*UserState.ACTIVE);*

*@Test*

*void saveNewUser() throws Exception {*

*when(userService.saveUser(any()))*

*.thenReturn(userDto);*

*mvc.perform(post("/users")*

*.content(mapper.writeValueAsString(userDto)) //преобразовываем в JSON и вкладываем в запрос*

*.characterEncoding(StandardCharsets.UTF\_8)*

*.contentType(MediaType.APPLICATION\_JSON)*

*.accept(MediaType.APPLICATION\_JSON))*

*.andExpect(status().isOk())*

*.andExpect(jsonPath("$.id", is(userDto.getId()), Long.class))*

*.andExpect(jsonPath("$.firstName", is(userDto.getFirstName())))*

*.andExpect(jsonPath("$.lastName", is(userDto.getLastName())))*

*.andExpect(jsonPath("$.email", is(userDto.getEmail()))); } }*

Аннотация ***@SneakyThrows*** позволяет игнорировать возможный выброс исключений и не писать *try-catch*.

Рассмотрим методы класса ***MockMvc***:

* Метод ***perform()*** имитирует выполнение запроса к эндпоинту. Обязательно указывается его тип и путь.
* Метод ***content*** передает данные в тело запроса.
* Методы ***characterEncoding()***, ***contentType()***, ***accept()*** работают соответственно названию.
* Методы ***andExpect()*** в *MockMvc* работают аналогично методам семейства *assert* в обычных тестах.

Проверить возвращаемые ответы *JSON* можно с помощью матчеров [JsonPathResultMatchers](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/result/JsonPathResultMatchers.html). Подробнее о том, как составлять выражения для извлечения данных из JSON, можно в документации библиотеки [JsonPath](https://github.com/json-path/JsonPath).

Подробнее о работе *MockMvc* можно почитать [тут](https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework).

Рассмотрим **интеграционное тестирование** *Spring Data JPA* (**репозиториев**).

Аннотация ***@DataJpaTest*** позволяет инициировать для теста контекст, состоящий только из тех бинов, которые требуются для корректной работы тестируемых *JPA* (в том числе бины, помеченные *@Entity*, и вспомогательные бины). Также она сразу проверяет соответствие *SQL*-запросов и *Entity*-модели сущностей.

Методы проверки здесь являются транзакционными, поэтому нужно заранее подготовить БД.

*@DataJpaTest*

*class UserRepositoryIT {*

*@Autowired*

*private UserRepository userRepository;*

*@BeforeEach*

*private void addUsers() {*

*userRepository.save(User.builder()*

*.firstName(“firstname”)*

*.lastName(“lastname”)*

*.age(25)*

*.build());*

*userRepository.save(User.builder()*

*.firstName(“firstname2”)*

*.lastName(“lastname2”)*

*.age(35)*

*.build()); }*

*@AfterEach*

*private void deleteUsers() {*

*userRepository.deleteAll(); }*

*@Test*

*void findByFirstNameAndLastName() {*

*Optional<User> actualUser = userRepository.findByFirstNameAndLastName(“firstname”, “lastname”);*

*assertTrue(actualUser.isPresent()); }*

*@Test*

*void findAllUsersUnderThirty() {*

*List<User> actualUsers = userRepository.findAllUsersUnderThirty();*

*assertEquals(1, actualUsers.size());*

*assertEquals(“firstname”, actualUsers.get(0).getFirstName()); } }*

Аннотация ***@JsonTest*** отвечает за тестирование функциональности, связанной с сериализацией и десериализацией *JSON* в приложении. Он важен при конвертации *DTO*-объектов с изменением формата.

*@JsonTest*

*public class UserDtoJsonTest {*

*@Autowired*

*private JacksonTester<UserDto> json;*

*@Test*

*void testUserDto() throws Exception {*

*UserDto userDto = new UserDto( 1L,*

*"john.doe@mail.com",*

*"John",*

*"Doe",*

*"2022.07.03 19:55:00",*

*UserState.ACTIVE);*

*JsonContent<UserDto> result = json.write(userDto);*

*assertThat(result).extractingJsonPathNumberValue("$.id").isEqualTo(1);*

*assertThat(result).extractingJsonPathStringValue("$.firstName").isEqualTo("John");*

*assertThat(result).extractingJsonPathStringValue("$.lastName").isEqualTo("Doe");*

*assertThat(result).extractingJsonPathStringValue("$.email").isEqualTo("john.doe@mail.com"); } }*

Подробнее о тестовых аннотациях (слайсах) можно посмотреть [тут](https://docs.spring.io/spring-boot/docs/current/reference/html/test-auto-configuration.html#appendix.test-auto-configuration.slices).

Для тестирования *API* можно также использовать внешние программы, например, ***Insomnia*** или ***Postman***. Для этого нужно создать новую коллекцию, наполнить ее запросами и написать для каждого тесты. Запросы будут выполняться сверху вниз, поэтому порядок их расположения важен.

В *Postman* тесты пишутся на языке ***JavaScript*** в комплекте с библиотекой ***pm*** от *Postman*.

Важно проверить следующие утверждения:

* Запрос успешно отработан и сервер вернул ожидаемый код обработки.
* Если мы ожидаем в ответ какие-либо данные, то нужно проверить, что ответ содержит тело с форматом данных, соответствующих ожидаемому.

**Первый способ** проверки утверждений – цепочка вызовов ***to.be***. Получаем ответ сервера через метод ***pm.response*** и формируем для него утверждение.

*pm.response.to.be.ok; // проверяем утверждение, что код ответа равен 200*

*pm.response.to.be.badRequest; // проверяем утверждение, что код ответа равен 400*

*pm.response.to.be.accepted; // проверяем утверждение, что код ответа равен 202*

*pm.response.to.be.notFound; // проверяем утверждение, что код ответа равен 404*

*pm.response.to.be.success; // проверяем утверждение, что код ответа равен 2xx*

*pm.response.to.be.error; // проверяем утверждение, что код ответа равен 4xx или 5xx*

*pm.response.to.be.withBody; // проверяем утверждение, что в ответе есть тело*

*pm.response.to.be.json; // проверяем утверждение, что тело в формате JSON*

**Второй способ** проверки утверждения – метод ***except***. В качестве аргумента мы передаем проверяемое значение, а затем формируем утверждение для этого значения.

*pm.expect(pm.response.code).to.equal(200); // проверяем утверждение, что код ответа равен 200*

*const body = pm.response.json(); // получаем тело ответа в формате JSON*

*pm.expect(body).is.an('array'); // проверяем, что тело ответа является массивом*

*pm.expect(body.length).to.be.gte(1); // проверяем, что массив не пустой*

Набор связанных утверждений удобно группировать в один тест-кейс. Для этого используется метод ***pm.test***. Он принимает на вход строку с описанием теста и функцию с набором проверяемых утверждений.

*pm.test("Ответ должен содержать код статуса 200 и данные в формате json", function() {*

*pm.response.to.be.ok; // код ответа должен быть равен 200 OK*

*pm.response.to.be.withBody; // ответ должен содержать тело*

*pm.response.to.be.json; }); // и тело ответа должно быть в формате JSON*

Для оптимизации работы с запросами, общие тесты можно вынести на уровень коллекции.

Для тестирования слоя DAO (слой обращения с БД) можно использовать ***Testcontainers***. Она позволяет развернуть любой вид БД с помощью образов докера. Рассмотрим на примере *PostgreSQL*.

Добавим в проект зависимости.

*<dependency>*

*<groupId>org.testcontainers</groupId>*

*<artifactId>testcontainers</artifactId>*

*<version>1.17.6</version>*

*<scope>test</scope>*

*</dependency>*

*<dependency>*

*<groupId>org.testcontainers</groupId>*

*<artifactId>junit-jupiter</artifactId>*

*<version>1.17.6</version>*

*<scope>test</scope>*

*</dependency>*

*<dependency>*

*<groupId>org.testcontainers</groupId>*

*<artifactId>postgresql</artifactId>*

*<version>1.17.6</version>*

*<scope>test</scope>*

*</dependency>*

Создадим абстрактный класс, который будет поднимать тестовую БД.

*@Testcontainers*

*public abstract class AbstractTestcontainers {*

*@Container*

*private static final PostgreSQLContainer<?> postgreSQLContainer =*

*new PostgreSQLContainer<>("postgres:latest")*

*.withDatabaseName("straigt-dao-unit-test")*

*.withUsername("straigt")*

*.withPassword("password");*

*@DynamicPropertySource // записывает данные в переменные среды*

*private static void registerDataSourceProperties(DynamicPropertyRegistry registry) {*

*registry.add("spring.datasource.url", postgreSQLContainer::getJdbcUrl);*

*registry.add("spring.datasource.username", postgreSQLContainer::getUsername);*

*registry.add("spring.datasource.password", postgreSQLContainer::getPassword); }*

*@BeforeAll // запускаем миграцию БД перед тестами*

*static void beforeAll() {*

*Flyway flyway = Flyway.configure()*

*.dataSource(postgreSQLContainer.getJdbcUrl(), postgreSQLContainer.getUsername(),*

*postgreSQLContainer.getPassword())*

*.load();*

*flyway.migrate(); }*

Создадим класс, который будет наследоваться от абстрактного и запускать тесты.

*public class TestcontainersTest extends AbstractTestcontainers {*

*@Test*

*void canStartPostgresDB() {*

*assertThat(postgreSQLContainer.isRunning()).isTrue();*

*assertThat(postgreSQLContainer.isCreated()).isTrue(); } }*