сделать Тестирование сервиса поиска Продумайте и разработайте несколько сценариев для проверки SearchService и StorageService в этом тесте. Используйте при тестировании моки. Варианты сценариев Поиск в случае отсутствия объектов в StorageService . Поиск в случае, если объекты в StorageService есть, но нет подходящего. Поиск, когда есть подходящий объект в StorageService . В процессе работы внутри тестового класса вы можете писать любые вспомогательные методы, которые посчитаете нужными. Также приветствуются дополнительные сценарии для тестов.0

Хорошо, я должен написать тесты для SearchService и StorageService, используя моки. Давайте сначала разберусь, что нужно проверить.

Сначала посмотрю на требования: есть несколько сценариев тестирования.

Первый сценарий: поиск, когда в StorageService нет объектов. Нужно убедиться, что в этом случае возвращается пустой список.

Второй сценарий: объекты есть, но нет подходящих по поисковому запросу. Тоже должен вернуться пустой результат.

Третий сценарий: есть подходящие объекты. Нужно проверить, что они правильно находятся и возвращаются.

Также можно добавить дополнительные тесты, например, проверку регистра, частичное совпадение и т.д.

Теперь подумаю о структуре тестового класса. Поскольку используется Spring, возможно, я буду использовать Mockito для мокирования StorageService. Но в данном случае StorageService не имеет внешних зависимостей, так как он просто хранит данные в памяти. Но для изоляции тестов SearchService, я должен замокать StorageService, чтобы контролировать возвращаемые данные.

В каждом тесте я буду создавать экземпляр SearchService, передавая ему мок StorageService. Затем настрою мок так, чтобы он возвращал определенные данные при вызове getAllCollection(). После этого вызову метод search() и проверю результат.

Например, для первого сценария:

- Мокаем StorageService, чтобы getAllCollection() возвращал пустой список.

- Вызываем search() с любым запросом.

- Проверяем, что результат пуст.

Для второго сценария:

- Настраиваем мок StorageService, чтобы он возвращал список объектов (например, продукты и статьи), но ни один из них не содержит искомую строку.

- Проверяем, что результат пуст.

Третий сценарий:

- Создаем объекты, которые должны совпадать с запросом.

- Настраиваем мок на возврат этих объектов.

- Вызываем search() с нужным запросом и проверяем, что в результате есть ожидаемые объекты.

Также нужно учесть, что метод search() использует contains(searchName), то есть проверка на частичное совпадение без учета регистра? Или с учетом? В текущей реализации, как видно из кода, используется contains, который чувствителен к регистру. Значит, если ищем "сало", то "Сало" с большой буквы не совпадет. Возможно, стоит добавить тест на регистр, но это уже зависит от требований.

Теперь продумаю каждый тест подробнее.

1. Тест поиска при пустом хранилище:

- Мок StorageService возвращает пустую коллекцию.

- Вызов search("something") должен вернуть пустой список.

2. Тест, когда объекты есть, но не подходят:

- Создаем, например, продукт с именем "Шило" и статью с именем "articl1".

- Вызываем поиск по запросу "мыло".

- Ожидаем, что ничего не найдется.

3. Тест с найденными объектами:

- Создаем продукты и статьи, которые содержат искомую строку. Например, продукт "Сало" и продукт "Сало Сало Сало".

- Запрос "Сало" должен вернуть оба продукта.

Дополнительные тесты:

- Проверка чувствительности к регистру: запрос "сало" в нижнем регистре не найдет "Сало" с большой буквы.

- Частичное совпадение: например, поиск "articl" найдет все статьи с именами "articl1", "articl2" и т.д.

- Проверка, что возвращаются объекты как Product, так и Article, если они подходят.

Теперь нужно реализовать эти тесты с использованием JUnit и Mockito.

Сначала подключу зависимости. Предположу, что проект использует JUnit 5 и Mockito.

Структура тестового класса:

import static org.mockito.Mockito.\*;

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

class SearchServiceTest {

private StorageService storageService;

private SearchService searchService;

@BeforeEach

void setUp() {

storageService = mock(StorageService.class);

searchService = new SearchService(storageService);

}

// Тесты

}

Теперь напишу тесты.

Первый тест: поиск при пустом хранилище.

@Test

void search\_WhenStorageEmpty\_ReturnsEmptyList() {

when(storageService.getAllCollection()).thenReturn(Collections.emptyList());

Collection<SearchResult> results = searchService.search("test");

assertTrue(results.isEmpty());

}

Второй тест: объекты есть, но не подходят.

@Test

void search\_WhenNoMatchingObjects\_ReturnsEmptyList() {

List<Searchable> mockData = new ArrayList<>();

mockData.add(new FixPriceProduct(UUID.randomUUID(), "Шило"));

mockData.add(new Article(UUID.randomUUID(), "articl1", "Инструкция"));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("мыло");

assertTrue(results.isEmpty());

}

Здесь могут возникнуть проблемы, потому что конструкторы продуктов и статей требуют определенных параметров. Нужно правильно создать объекты.

Третий тест: есть совпадения.

@Test

void search\_WhenMatchingObjectsExist\_ReturnsResults() {

UUID productId1 = UUID.randomUUID();

UUID productId2 = UUID.randomUUID();

List<Searchable> mockData = new ArrayList<>();

mockData.add(new SimpleProduct(productId1, "Сало", 300));

mockData.add(new SimpleProduct(productId2, "Сало Сало Сало", 300));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("Сало");

assertEquals(2, results.size());

// Проверить, что в результатах есть оба продукта

assertTrue(results.stream().anyMatch(r -> r.getId().equals(productId1)));

assertTrue(results.stream().anyMatch(r -> r.getId().equals(productId2)));

}

Теперь проверка регистра:

@Test

void search\_IsCaseSensitive() {

List<Searchable> mockData = new ArrayList<>();

mockData.add(new SimpleProduct(UUID.randomUUID(), "Сало", 300));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("сало");

assertTrue(results.isEmpty());

}

Этот тест должен провалиться, если поиск не чувствителен к регистру, но согласно текущей реализации, поиск чувствителен.

Также можно проверить частичное совпадение:

@Test

void search\_PartialMatch() {

List<Searchable> mockData = new ArrayList<>();

mockData.add(new Article(UUID.randomUUID(), "articl1", "text"));

mockData.add(new Article(UUID.randomUUID(), "article2", "text"));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("articl");

assertEquals(2, results.size());

}

Здесь оба артикла содержат "articl" в имени.

Еще можно проверить смешанные типы: продукты и статьи.

@Test

void search\_ReturnsBothArticlesAndProducts() {

List<Searchable> mockData = new ArrayList<>();

mockData.add(new Article(UUID.randomUUID(), "searchTerm", "text"));

mockData.add(new SimpleProduct(UUID.randomUUID(), "searchTerm", 100));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("searchTerm");

assertEquals(2, results.size());

assertTrue(results.stream().anyMatch(r -> r.getType().equals("ARTICLE")));

assertTrue(results.stream().anyMatch(r -> r.getType().equals("PRODUCT")));

}

Но для этого нужно, чтобы SearchResult.fromSearchable правильно определял тип. Посмотрю, как реализован SearchResult:

public class SearchResult {

private final UUID id;

private final String name;

private final String type;

private SearchResult(UUID id, String name, String type) {

this.id = id;

this.name = name;

this.type = type;

}

public static SearchResult fromSearchable(Searchable searchable) {

return new SearchResult(

searchable.getId(),

searchable.searchTerm(),

searchable.searchTipContent()

);

}

// геттеры

}

Значит, тип берется из searchTipContent(), который для Article возвращает "ARTICLE", а для Product — "PRODUCT". Поэтому в тесте можно проверять тип через getType().

Теперь нужно убедиться, что все тесты используют моки правильно, и не зависят от реальной реализации StorageService.

Возможные проблемы:

- Создание объектов Searchable. Например, при создании Product, его подклассы (SimpleProduct и т.д.) могут требовать определенные параметры. Нужно правильно их инициализировать в тестах.

Например, при создании SimpleProduct требуется UUID, имя и цена. В тестах можно использовать случайные UUID и любую цену, так как в данном контексте они не важны.

Также нужно обратить внимание на то, что в текущей реализации метод searchTerm() для Product возвращает nameProduct, а для Article — nameArticle. Поэтому при создании моковых объектов нужно правильно задавать эти имена.

В целом, структура тестов должна быть такой:

1. Настройка мока StorageService для возврата определенного набора данных.

2. Вызов метода search() у SearchService.

3. Проверка результатов (количество элементов, наличие ожидаемых ID, типов и т.д.).

Дополнительно, можно проверить порядок элементов, но, так как используется stream(), порядок зависит от порядка в коллекции, возвращаемой StorageService. В тестах порядок может не быть важным, если не требуется сортировка.

Еще один тест: проверка обработки null или пустой строки в поисковом запросе.

@Test

void search\_WithEmptyQuery\_ReturnsAll() {

// Здесь зависит от логики: если searchName пустая строка, то contains("") вернет true для всех строк.

// Поэтому в этом случае должны вернуться все объекты.

List<Searchable> mockData = new ArrayList<>();

mockData.add(new Article(UUID.randomUUID(), "article1", "text"));

mockData.add(new SimpleProduct(UUID.randomUUID(), "product1", 100));

when(storageService.getAllCollection()).thenReturn(mockData);

Collection<SearchResult> results = searchService.search("");

assertEquals(2, results.size());

}

Этот тест проверяет, что при пустом запросе возвращаются все объекты, так как любая строка содержит пустую подстроку.

Но возможно, это нежелательное поведение, и нужно, чтобы пустой запрос не возвращал ничего. Тогда тест должен ожидать пустой результат, но в текущей реализации это не так. Нужно уточнить требования. В данном случае, согласно коду, поиск с пустой строкой вернет все объекты.

Возможно, это стоит учесть и добавить такой тест.

Еще один момент: метод searchTerm() возвращает имя, по которому идет поиск. Например, для Article это nameArticle, для Product — nameProduct. Поэтому при создании моковых объектов в тестах нужно задавать эти имена правильно.

В целом, все основные сценарии покрыты. Теперь можно приступать к написанию кода тестов.