GeekBrains

Тестирование Backend на Java

Автоматизированное тестирование REST API с использованием Retrofit или OkHttp3

На этом уроке

1. Разберём основы архитектуры фреймворка Retrofit.
2. Научимся встраивать интерцепторы.
3. Напишем свой маппер для логирования.
4. Узнаем об архитектуре.

Оглавление

Введение

- Знакомство с проектом

Retrofit 2. Архитектура фреймворка

- Использование Retrofit
- Написание тестов
  - Тесты на запросы с телом (POST, PUT)
- Логирование тестов
  - Интерцепторы
  - Интерцептор логирования запросов и ответов

Практическое задание

Дополнительные материалы

Используемые источники

Введение

В этом курсе мы рассматриваем два фреймворка для организации тестирования REST API. Retrofit — полноценный HTTP-клиент со своей архитектурой, наша задача сегодня — научиться им пользоваться и написать первые собственные тесты. Для этого вспомним следующие понятия:

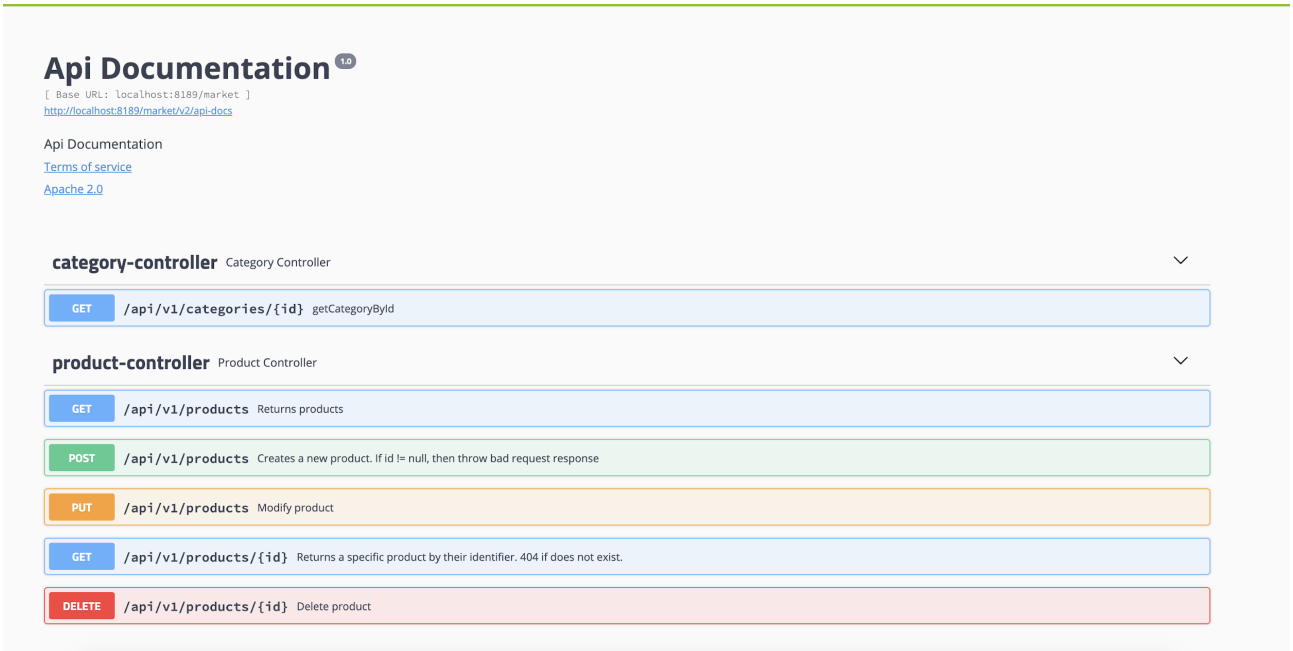
1. Интерфейсы.
2. Сериализация и десериализация.
3. Объектная модель.
4. OkHttpClient.

Познакомимся с такими терминами, как Swagger, маппер, научимся создавать ретрофит-клиент и сервисы для вызова контроллеров.

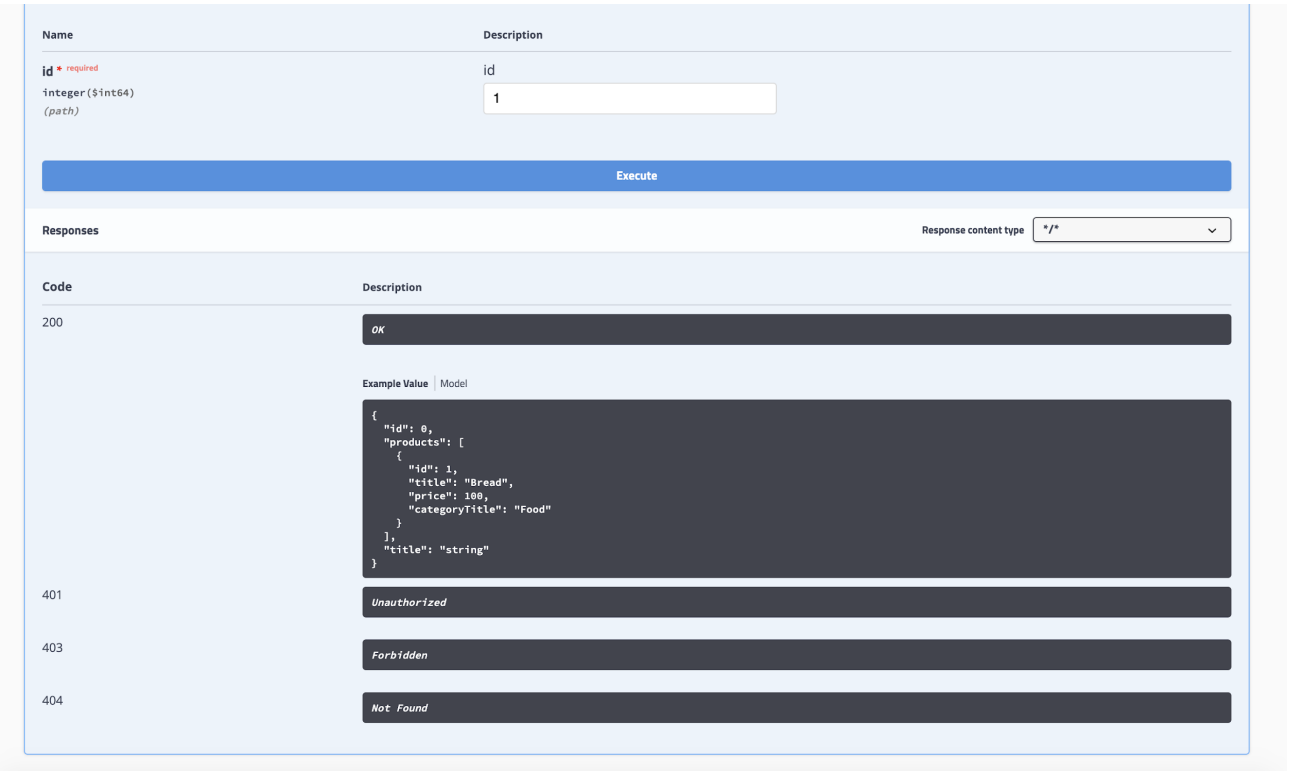
На этом уроке поработаем с другим проектом — mini-market. Он сделан простым, чтобы мы могли отточить свои навыки тестирования и сделать максимально полный тест-дизайн на некоторых эндпоинтах. Ссылка на проект — в задании к уроку в личном кабинете.

Знакомство с проектом

Документация нашего проекта — Swagger. Это широко используемый формат. Локально он располагается по этому адресу: http://localhost:8189/market/swagger-ui.html#/product-controller. Выглядит он следующим образом:



Каждый эндпоинт можно развернуть и посмотреть. Есть возможность указывать что-то в запросе и смотреть, как формируется ответ. В хорошем сваггере указываются все коды ответов сервера, возникающие для каждого запроса и требующие проверки:



В Swagger можно вызвать эндпоинт. Однако он реализуется через утилиты cURL — после нажатия кнопки Execute появится команда, которая на самом деле вызывается в cli.

Наш проект имеет небольшой UI, где доступны некоторые операции, а именно, создание и удаление продукта. UI локально располагается по адресу: http://localhost:8189/market/. База данных h2 поднимается автоматически.

1. Доступ к консоли: http://localhost:8189/market/h2-console.
2. Пользователь — sa.
3. Пароля нет.

Retrofit 2. Архитектура фреймворка

Retrofit — это клиент REST для Java и Android. Он позволяет относительно легко получать и загружать JSON или другие структурированные данные через веб-клиент на основе REST. В Retrofit надо самим выбрать преобразователь для сериализации данных. Обычно для JSON используется Gson или Jackson, мы воспользуемся последним. Можно добавить собственные конвертеры для обработки XML или других протоколов. Retrofit использует библиотеку OkHttp для HTTP-запросов.

Импортируем библиотеки из maven-репозитория:

```
<dependencies>
   <dependency>
       <groupId>com.squareup.retrofit2</groupId>
       <artifactId>retrofit</artifactId>
       <version>2.9.0</version>
   </dependency>
   <dependency>
       <groupId>org.junit.jupiter</groupId>
       <artifactId>junit-jupiter</artifactId>
       <version>5.7.0</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>com.fasterxml.jackson.core</groupId>
       <artifactId>jackson-annotations</artifactId>
       <version>2.11.1</version>
   </dependency>
   <dependency>
       <groupId>com.fasterxml.jackson.core</groupId>
       <artifactId>jackson-core</artifactId>
       <version>2.11.1</version>
   </dependency>
   <dependency>
       <groupId>com.fasterxml.jackson.core</groupId>
       <artifactId>jackson-databind</artifactId>
       <version>2.11.2</version>
   </dependency>
   <dependency>
       <groupId>com.squareup.retrofit2</groupId>
       <artifactId>converter-jackson</artifactId>
       <version>2.6.1</version>
   </dependency>
   <dependency>
       <groupId>org.projectlombok</groupId>
       <artifactId>lombok</artifactId>
       <version>1.18.12</version>
   </dependency>
   <dependency>
       <groupId>org.hamcrest</groupId>
       <artifactId>hamcrest</artifactId>
       <version>2.2</version>
       <scope>test</scope>
   </dependency>
</dependencies>
```

Retrofit не содержит в себе библиотеки ассертов, поэтому hamcrest-библиотеку надо добавлять отдельно.

Появились две новые библиотеки — сам клиент Retrofit и связанная библиотека Converter-Jackson, которая позволяет добавлять функции Jackson к рест-клиенту Retrofit.

Использование Retrofit

Для работы с Retrofit потребуются три следующих класса:

1. Класс модели, который используется как модель JSON.
2. Интерфейсы, определяющие возможные HTTP-операции.
3. Класс Retrofit.Builder — экземпляр, который использует интерфейс и API Builder, чтобы определить конечную точку URL-адреса для операций HTTP.

Каждый метод интерфейса представляет собой один возможный вызов API. Он должен иметь аннотацию HTTP: GET, POST и т. д. Главное — указать тип запроса и относительный URL. Возвращаемое значение помещает ответ в объект Call с типом ожидаемого результата.

Создадим интерфейс CategoryService в src/main/java/<название пакета>/service:

```
package ru.geekbrains.lesson5.service;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Path;
import ru.geekbrains.lesson5.dto.GetCategoryResponse;

public interface CategoryService {

   @GET("categories/{id}")
   Call<GetCategoryResponse> getCategory(@Path("id") int id);
}
```

Обычно используются заменяющие блоки и параметры запроса для настройки URL-адреса. Замещающий блок добавляется к относительному URL путём применения {}. Через аннотацию @Path к параметру метода значение этого параметра привязывается к конкретному блоку замены.

В Call устанавливается специальный объект ResponseBody, если:

- тип результата — null;
- или результат неважен, то есть мы не собираемся его парсить в дальнейшем.

Пример:

```
package ru.geekbrains.lesson5.service;
```

```
import retrofit2.Call;
import okhttp3.ResponseBody;
import retrofit2.http.GET;
import retrofit2.http.Path;
import ru.geekbrains.lesson5.dto.GetCategoryResponse;

public interface CategoryService {

   @GET("categories/{id}")
   Call<ResponseBody> getCategory(@Path("id") int id);
}
```

Вернёмся к первоначальной реализации. Тип получаемого значения — GetCategoryResponse. Этот класс можно получить через сервис jsonschema2pojo, который мы использовали на предыдущих занятиях. Выглядит наш POJO следующим образом:

```
package ru.geekbrains.lesson5.dto;

import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;

import java.util.ArrayList;
import java.util.List;

@Data
public class GetCategoryResponse {
   @JsonProperty("id")
   private Integer id;
   @JsonProperty("title")
   private String title;
   @JsonProperty("products")
   private List<Product> products = new ArrayList<>();
}
```

Product может быть внутренним или отдельно созданным классом.

```
package ru.geekbrains.lesson5.dto;

import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;

@Data
public class Product {
   @JsonProperty("id")
   private Integer id;
   @JsonProperty("title")
   private String title;
```

```
    @JsonProperty("price")
    private Integer price;
    @JsonProperty("categoryTitle")
    private String categoryTitle;
}
```

Теперь определим Retrofit-клиент. Создадим класс RetrofitUtils в пакете util и там же сформируем метод getRetrofit():

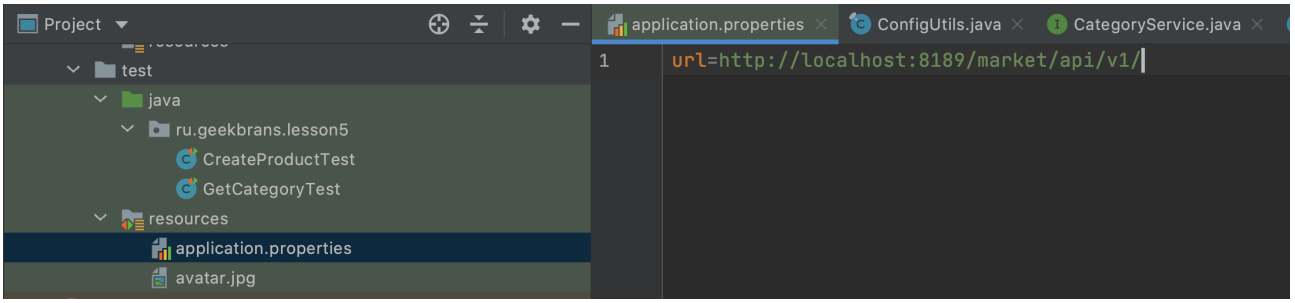
```
package ru.geekbrains.lesson5.util;

import lombok.experimental.UtilityClass;
import retrofit2.Retrofit;
import retrofit2.converter.jackson.JacksonConverterFactory;

@UtilityClass
public class RetrofitUtils {

    public Retrofit getRetrofit(){
        return new Retrofit.Builder()
                .baseUrl(ConfigUtils.getBaseUrl())
                .addConverterFactory(JacksonConverterFactory.create())
                .build();

    }
}
```

В baseUrl требуется передать хост, который вызывается во всех API-запросах. В нашем случае это http://localhost:8189/market/api/v1/. Далее происходит вызов статического метода getBaseUrl() класса ConfigUtils. Создадим этот класс в пакете util и пропишем в нём методы работы с properties-файлами.

```
package ru.geekbrains.lesson5.util;

import lombok.SneakyThrows;
import lombok.experimental.UtilityClass;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.Properties;

@UtilityClass
public class ConfigUtils {
    Properties prop = new Properties();
    private static InputStream configFile;

    static {
```

```
        try {
            configFile = new
FileInputStream("src/test/resources/application.properties");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    @SneakyThrows
    public String getBaseUrl() {
        prop.load(configFile);
        return prop.getProperty("url");
    }
}
```

Положим файл с одним (пока) свойством url в src/test/resources:



Написание тестов

Создадим тестовый класс, а в нём сформируем и проинициализируем экземпляр сервиса:

```
package ru.geekbrans.lesson5;

import lombok.SneakyThrows;
import org.hamcrest.CoreMatchers;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import retrofit2.Response;
import ru.geekbrains.lesson5.dto.GetCategoryResponse;
import ru.geekbrains.lesson5.service.CategoryService;
import ru.geekbrains.lesson5.util.RetrofitUtils;

import static org.hamcrest.MatcherAssert.assertThat;

public class GetCategoryTest {
  static CategoryService categoryService;
    @BeforeAll
    static void beforeAll() {
        categoryService =
RetrofitUtils.getRetrofit().create(CategoryService.class);
```

```
    }

    @SneakyThrows
    @Test
    void getCategoryByIdPositiveTest() {
        Response<GetCategoryResponse> response =
categoryService.getCategory(1).execute();
        assertThat(response.isSuccessful(), CoreMatchers.is(true));
    }

}
```

Сервис categoryService требуется для отправки запроса на сервер. Разберём код выше подробнее.

```
  static CategoryService categoryService;
   @BeforeAll
   static void beforeAll() {
       categoryService =
RetrofitUtils.getRetrofit().create(CategoryService.class);
   }
```

Создаём статическое поле подходящего нам сервиса — в нашем случае это CategoryService. Затем инициализируем его через Retrofit-клиента. Инициализация сервиса происходит один раз перед всеми тестами, поэтому помещаем инициализацию в beforeAll. У класса Retrofit вызываем метод create, куда рефлексивно добавляем наименование интерфейса сервиса.

В тесте вызываем подходящий нам метод из интерфейса сервиса и передаём требуемые параметры Path и Query, а также тело запроса. Пример с POST-запросом — ниже.

```
   Response<GetCategoryResponse> response =
categoryService.getCategory(1).execute();
```

Значение кладётся в переменную типа Response<тип Callback в сервисе>. Затем в ассертах используется, например, такое значение:

```
@SneakyThrows
@Test
void getCategoryWithResponseAssertionsPositiveTest() {
   Response<GetCategoryResponse> response =
categoryService.getCategory(1).execute();
   assertThat(response.isSuccessful(), CoreMatchers.is(true));
   assertThat(response.body().getId(), equalTo(1));
   assertThat(response.body().getTitle(), equalTo("Food"));
   response.body().getProducts().forEach(product ->
```

```
            assertThat(product.getCategoryTitle(), equalTo("Food")));
}
```

У объекта типа Response есть много интересных методов. Например, isSuccessful() возвращает true, если код ответа 2xx и false — в противном случае. Проверяется и конкретный код ответа, если выбрать метод code(). Вызывая метод body(), мы получаем объект типа GetCategoryResponse, который можно будет парсить.

Тесты на запросы с телом (POST, PUT)

Создадим сервис, работающий с продуктами:

```
package ru.geekbrains.lesson5.service;

import okhttp3.ResponseBody;
import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.DELETE;
import retrofit2.http.POST;
import retrofit2.http.Path;
import ru.geekbrains.lesson5.dto.Product;

public interface ProductService {
    @POST("products")
    Call<Product> createProduct(@Body Product createProductRequest);

    @DELETE("products/{id}")
    Call<ResponseBody> deleteProduct(@Path("id") int id);
}
```

Напишем тест для создания продукта, а после теста удалим из БД созданный тестовый объект. Наш класс Product в точности повторяет тело запроса для POST/products. Изменим аннотации в этом классе:

- удалим аннотацию @Data;
- и добавим построение объекта через wither — способ, аналогичный билдеру.

В нашем случае — это 4 wither, то есть по одному на каждое поле. В отличие от setter, которые возвращают void, wither возвращают сам объект класса, например, Product. Это позволяет поддерживать fluent-интерфейсы.

```
public class Product {
    @JsonProperty("id")
    private Integer id;
```

```
    @JsonProperty("title")
    private String title;
    @JsonProperty("price")
    private Integer price;
    @JsonProperty("categoryTitle")
    private String categoryTitle;

public Product withId(Integer id) {
    this.id = id;
    return this;
}

public Product withTitle(String title) {
    this.title = title;
    return this;
}

public Product withPrice(Integer price) {
    this.price = price;
    return this;
}

public Product withCategoryTitle(String categoryTitle) {
    this.categoryTitle = categoryTitle;
    return this;
}
}
```

Конечно, Lombok предоставляет аннотации и для wither. Сейчас это аннотация @Wtih, а в прошлых версиях плагина — @Wither.

Важно! *Для создания объекта через Wither требуется иметь пустой конструктор для класса (без аргументов).*

В результате наш класс будет выглядеть так:

```
package ru.geekbrains.lesson5.dto;

import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.*;

@JsonInclude(JsonInclude.Include.NON_NULL)
@Getter
@AllArgsConstructor
@NoArgsConstructor
@With
public class Product {
    @JsonProperty("id")
```

```
    private Integer id;
    @JsonProperty("title")
    private String title;
    @JsonProperty("price")
    private Integer price;
    @JsonProperty("categoryTitle")
    private String categoryTitle;
}
```

Мы удалили аннотацию @Data и оставили только @Getter, так как у нас уже написаны ассерты в предыдущих тестах с геттерами этих полей. Аннотации @AllArgsConstructor и @NoArgsConstructor добавляют конструктор со всеми аргументами и пустой конструктор соответственно.

Подготовим данные для теста, используя wither и javafaker. Чтобы применить javaFaker, добавим соответствующую зависимость в pom.xml:

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.15</version>
</dependency>
```

Мы создаём продукт без id, попрактикуйтесь создавать с id и посмотрите, что выйдет.

```
public class CreateProductTest {
    static ProductService productService;
    Product product;
    Faker faker = new Faker();

    int id;

    @BeforeAll
    static void beforeAll() {
        productService = RetrofitUtils.getRetrofit()
                .create(ProductService.class);
    }

    @BeforeEach
    void setUp() {
        product = new Product()
                .withTitle(faker.food().ingredient())
                .withCategoryTitle(Category.FOOD.title)
                .withPrice((int) (Math.random() * 10000));
    }
```

Создадим тест и сохраним созданный id в переменную, чтобы использовать потом при удалении:

```
@Test
@SneakyThrows
void createProductInFoodCategoryTest() {
   Response<Product> response = productService.createProduct(product)
           .execute();
      id =  response.body().getId();
       assertThat(response.isSuccessful(), CoreMatchers.is(true));
}
```

Чтобы передать тело запроса, указываем объект соответствующего типа в методе createProduct.

Напишем в tearDown-методе удаление тестового объекта, передав сохранённый id как Path-параметр метода deleteProduct:

```
@SneakyThrows
@AfterEach
void tearDown() {
   Response<ResponseBody> response = productService.deleteProduct(id).execute();
   assertThat(response.isSuccessful(), CoreMatchers.is(true));
}
```

Логирование тестов

Интерцепторы

Для логирования в retrofit используются интерцепторы (перехватчики) OkHttp. Перехватчики — это мощный механизм, который отслеживает, перезаписывает и повторяет вызовы.

Это простой перехватчик, который регистрирует исходящий запрос и входящий ответ, но не используется в коде тестов:

```
class LoggingInterceptor implements Interceptor {
  @Override public Response intercept(Interceptor.Chain chain) throws
IOException {
    Request request = chain.request();

    long t1 = System.nanoTime();
    logger.info(String.format("Sending request %s on %s%n%s",
        request.url(), chain.connection(), request.headers()));

    Response response = chain.proceed(request);

    long t2 = System.nanoTime();
    logger.info(String.format("Received response for %s in %.1fms%n%s",
```

```
        response.request().url(), (t2 - t1) / 1e6d, response.headers()));

    return response;
  }
}
```

Вызов chain.proceed (request) — важная часть реализации каждого перехватчика. В этом простом на вид методе выполняется вся работа HTTP, и создаётся ответ, подходящий запросу. Если chain.proceed (запрос) вызывается более одного раза, предыдущие тела ответа должны быть закрыты. Подробнее — в официальной документации.

Интерцептор логирования запросов и ответов

Воспользуемся классом HttpLoggingInterceptor(). Для его использования добавим зависимость:

```
<dependency>
   <groupId>com.squareup.okhttp3</groupId>
   <artifactId>logging-interceptor</artifactId>
   <version>3.9.0</version>
</dependency>
```

Видоизменим наш класс RetrofitUtils и включим две новые строчки:

```
@UtilityClass
public class RetrofitUtils {
   HttpLoggingInterceptor logging = new HttpLoggingInterceptor();

   OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
```

Первая строка создаёт экземпляр объекта-перехватчика, вторая — клиента, в который этот объект поместим. Изменим метод getRetrofit() в этом классе:

```
public Retrofit getRetrofit(){
//ставим уровень логирования, чтобы логировались запросы и ответы
   logging.setLevel(BODY);
//добавляем перехватчик в okhttp-клиент
   httpClient.addInterceptor(logging);
   return new Retrofit.Builder()
           .baseUrl(ConfigUtils.getBaseUrl())
           .addConverterFactory(JacksonConverterFactory.create())
//добавляем okhttp-клиент к retrofit-клиенту
           .client(httpClient.build())
           .build();
}
```

Теперь при запуске тестов мы будем видеть логирование запросов и ответов. Можно также расширить имеющиеся интерцепторы и даже написать свои:

```
HttpLoggingInterceptor logging = new HttpLoggingInterceptor(new PrettyLogger());
```

PrettyLogger — это класс, реализующий интерфейс HttpLoggingInterceptor.Logger. Надо переопределить всего один метод — log():

```
public class PrettyLogger implements HttpLoggingInterceptor.Logger {
    @Override
    public void log(String message) {
    }
}
```

Реализация может быть различной, но лучше работать по такому алгоритму:

1. Инициализируем ObjectMapper().
2. Обрезаем пробелы с обеих сторон (делаем trim()).
3. Находим сообщение, похожее на JSON, оно будет в {} или в [].
4. Используя маппер, кладём в новый объект наше JSON-сообщение.
5. Через prettyPrint-метод самого маппера делаем наш JSON «красивым».
6. Выводим его в log.info().
7. Если мы ошиблись, и это оказался не JSON — выводим логгером как Log.warn().

Код реализованного метода представлен ниже:

```
public class PrettyLogger implements HttpLoggingInterceptor.Logger {
    ObjectMapper mapper = new ObjectMapper();

    @Override
    public void log(String message) {
        String trimmedMessage = message.trim();
        if ((trimmedMessage.startsWith("{") && trimmedMessage.endsWith("}"))
                || (trimmedMessage.startsWith("[") &&
trimmedMessage.endsWith("]"))) {
            try {
                Object value = mapper.readValue(message, Object.class);
                String prettyJson =
mapper.writerWithDefaultPrettyPrinter().writeValueAsString(value);
```

```
                Platform.get().log(Platform.INFO, prettyJson, null);
            } catch (JsonProcessingException e) {
                Platform.get().log(Platform.WARN, message, e);
            }
        } else {
            Platform.get().log(Platform.INFO, message, null);
        }
    }
}
```

Практическое задание

1. Напишите автотесты для CRUD-запросов, относящихся к сервису продуктов с использованием retrofit.
2. В README приложите чек-лист или майнд-карту того, что проверяют ваши тесты.
3. Сдайте ссылку на репозиторий, указав ветку с кодом.

Дополнительные материалы

1. Официальная документация.
2. Доклад об архитектуре тестовых фреймворков.

Используемые источники

1. Официальная документация 1.
2. Официальная документация 2.
3. Статья Using Retrofit 2.x as REST client — Tutorial — Vogella.