GeekBrains

Тестирование Backend на Java

Расширенные возможности rest-assured

На этом уроке

1. Узнаем о возможностях библиотеки rest-assured.
2. Проведём рефакторинг кода.
3. Вспомним принципы DRY и KISS.
4. Поговорим о спецификациях проверок запросов и ответов.

Оглавление

Введение

На прошлом уроке мы создали проект и автоматизировали основные проверки REST API. Сегодня глубже ознакомимся с возможностями библиотеки rest-assured и проведём рефакторинг нашего кода. Вспомним принципы DRY и KISS, поговорим о паттернах проектирования (Builder), узнаем о спецификациях проверок запросов и ответов, которые нам предоставляет rest-assured. В результате выполнения практического задания мы получим готовый фреймворк, который будет не стыдно выложить в открытый доступ на GitHub.

Коллекция запросов для Postman с документацией лежит по этой ссылке.

Паттерны проектирования. Builder

Паттерны проектирования служат, чтобы избавиться от дублирования кода и сделать его понятнее и читаемее. Сегодня мы неоднократно воспользуемся различными билдерами (builder — строитель) для объектов. Их смысл состоит в использовании вместо конструкторов для сложных объектов больше трёх-четырёх полей. Подробнее о Builder можно прочитать здесь, а здесь — посмотреть пример кода на Java. Воспользуемся Builder, чтобы построить спецификации для запросов (RequestSpec) и ответов (ResponseSpec).

Спецификация ответа: ResponseSpecification

Создадим в классе с тестами новый объект:

```
ResponseSpecification responseSpecification = null;
```

В @BeforeEach определим основные проверки для респонса:

```
@BeforeEach
void beforeTest() {
   responseSpecification = new ResponseSpecBuilder()
           .expectStatusCode(200)
           .expectStatusLine("HTTP/1.1 200 OK")
           .expectContentType(ContentType.JSON)
           .expectResponseTime(Matchers.lessThan(5000L))
           .expectHeader("Access-Control-Allow-Credentials", "true")
           .build();
}
```

Для построения спецификации создаётся объект билдера: new ResponseSpecBuilder(), после чего через точку (fluent-интерфейс) перечисляются те ассерты, которые требуется выполнить.

Спецификация формируется для каждого конкретного запроса. Указываем в тесте, что её (спецификацию) надо использовать в методе spec():

```
    // Get recipe request
// авторизация в примере опущена, для повторения теста необходимо добавить параметр apiKey
    @Test
    void getRecipePositiveTest() {
        given()
                .when()
                .get("https://api.spoonacular.com/recipes/716429/information")
                .then()
                .spec(responseSpecification);
    }
```

После этого записываем дополнительные проверки или оставляем как есть — все ассерты, указанные в спецификации, сработают.

Если спецификация едина для всех ответов, указываем её в виде глобальной переменной:

```
RestAssured.responseSpecification = responseSpecification;
```

Даже уже созданная спецификация расширяется в любом месте использования, например:

```
RestAssured.responseSpecification =
       responseSpecification
                .expect()
                .body(containsString("84578389"));
```

Если спецификация задаётся глобально, то указывать её в тестах не требуется. Она применится по умолчанию:

```
    // Get recipe request
// авторизация в примере опущена, для прохождения теста необходимо добавить параметр apiKey
    @Test
    void getRecipePositiveTest() {
        given()
                .when()
                .get("https://api.spoonacular.com/recipes/716429/information");
}
```

Спецификация запроса: RequestSpecification

Построение происходит аналогично спецификации ответа, но используется строитель RequestSpecBuilder():

```
RequestSpecification requestSpecification = null;

@BeforeEach
void beforeTest() {
requestSpecification = new RequestSpecBuilder()
        .addQueryParam("apiKey", apiKey)
        .addQueryParam("includeNutrition", "false")
        .log(LogDetail.ALL)
        .build();
}
```

Удобно выносить туда основные заголовки: авторизацию, если она не меняется, типы отправляемых и получаемых данных.

Аналогично спецификации ответа спецификация запроса глобально задаётся через свойство класса RestAssured или для каждого запроса в отдельности:

Глобально:

```
RestAssured.requestSpecification = requestSpecification;
```

Для конкретного запроса:

```
@Test
void getRecipePositiveTest() {
   given()
           .spec(requestSpecification)
           .when()
           .get("https://api.spoonacular.com/recipes/716429/information")
           .then()
           .spec(responseSpecification);
}
```

Сериализация и десериализация

Сериализация объекта — процесс перевода какой-либо структуры данных в последовательность битов. В нашем случае, из java-объекта в JSON-файл или строку. Обратная к операции сериализации — операция десериализации, т. е. восстановление начального состояния структуры данных из

битовой последовательности, из JSON в java-объект. В курсе Java для тестировщиков мы уже сталкивались с Jackson, сегодня остановимся на практических аспектах.

Для работы с JSON есть несколько популярных библиотек, например, jackson и gson. Последний, кстати, разрабатывается компанией Google. В этом курсе воспользуемся первой библиотекой по двум причинам:

- она более популярна;
- есть аналогичные библиотеки jackson для других форматов, например, YAML и XML, в то время как gson работает только с JSON.

Конфигурация jackson настраивается через xml-файл или аннотации. Чаще всего используется второй вариант, поэтому нам понадобится импортировать три библиотеки:

В теги dependencies вставляем следующие зависимости:

```
<dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>${jackson.version}</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>${jackson.version}</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>${jackson.version}</version>
        </dependency>
```

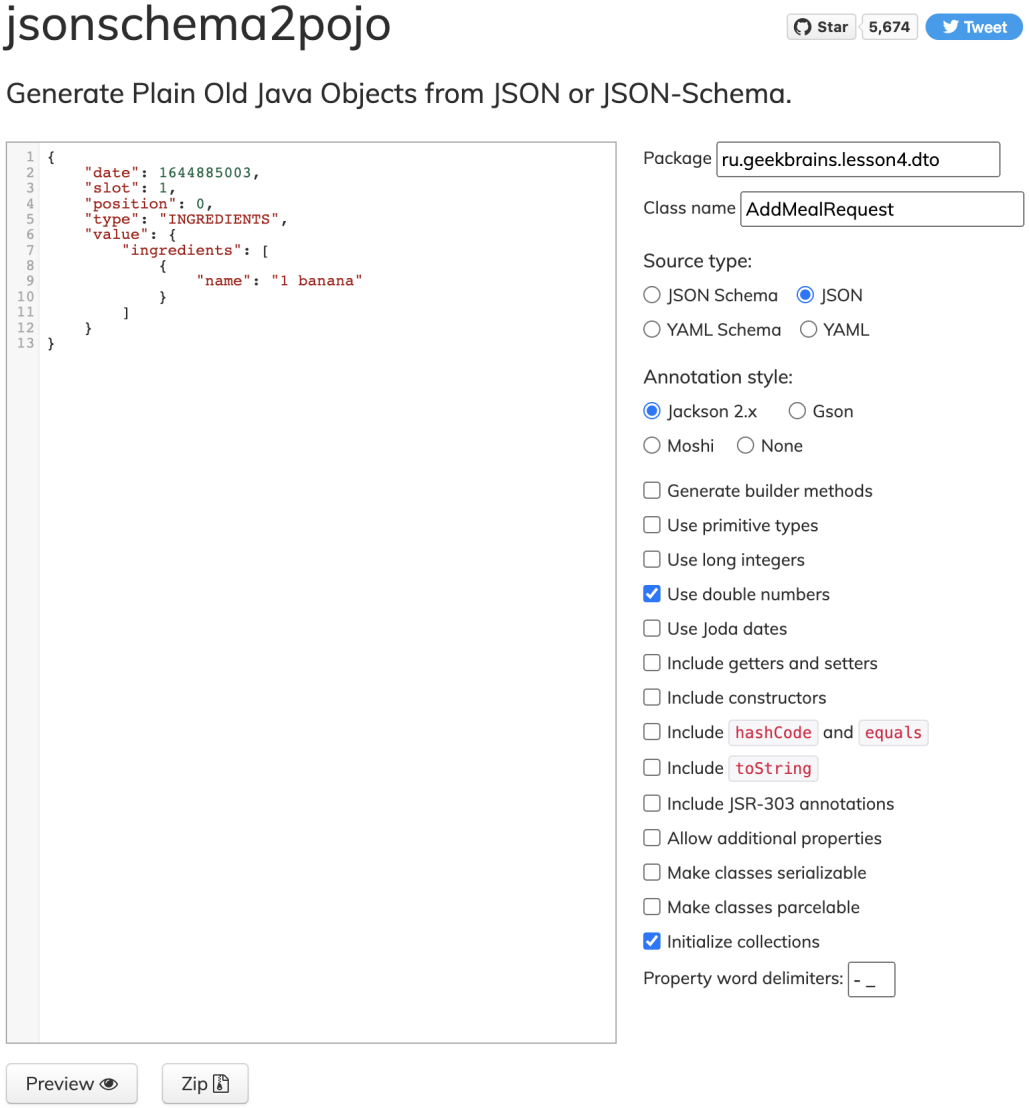
Версию jackson-библиотек можно уточнить на их странице в Maven-репозитории.

Воспользуемся только одной функцией этой библиотеки: десериализацией в описанный java-объект.

Подготовка POJO

POJO — это (уже) почти автоматически генерируемый объект из схем или объектов JSON или YAML, содержит описание полей. Для его генерации воспользуемся бесплатным сервисом jsonschema2pojo.

Возьмём JSON из реквеста “Add to Meal Plan” POST /mealplanner/:username/items, вставим его в предложенное поле и отметим подходящие галочки. Можно автоматически сгенерировать геттеры, сеттеры, билдеры, предусмотреть получение дополнительных полей и прочее.



Нажимаем на кнопку Preview и видим, что сгенерировалось несколько классов. На основе этого принимаем действующую иерархию или продумываем свою. Например, сохраняем все объекты в одном классе и делаем внутренние JSON-объекты такими же внутренними классами:

```
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonPropertyOrder;
import java.util.ArrayList;
import java.util.List;
import lombok.Data;

@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder( {
       "date",
       "slot",
       "position",
       "type",
       "value"
```

```
})
@Data
public class AddMealRequest {

   @JsonProperty("date")
   private Integer date;
   @JsonProperty("slot")
   private Integer slot;
   @JsonProperty("position")
   private Integer position;
   @JsonProperty("type")
   private String type;
   @JsonProperty("value")
   private Value value;

   @JsonInclude(JsonInclude.Include.NON_NULL)
   @JsonPropertyOrder( {
           "ingredients"
   })
   @Data
   private static class Value {
       @JsonProperty("ingredients")
       private List<Ingredient> ingredients = new ArrayList<Ingredient>();
   }

   @JsonInclude(JsonInclude.Include.NON_NULL)
   @JsonPropertyOrder( {"name"})
   @Data
   private static class Ingredient {
       @JsonProperty("name")
       private String name;
   }
}
```

Лучше сделать все поля приватными, по умолчанию они создаются публичными, что влияет на безопасность объекта.

Сделаем те же самые манипуляции для JSON-а ответа и сохраним поля респонса в класс AddMealResponse:

```
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonPropertyOrder;
import javax.annotation.processing.Generated;

@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder({
       "status",
       "id"
```

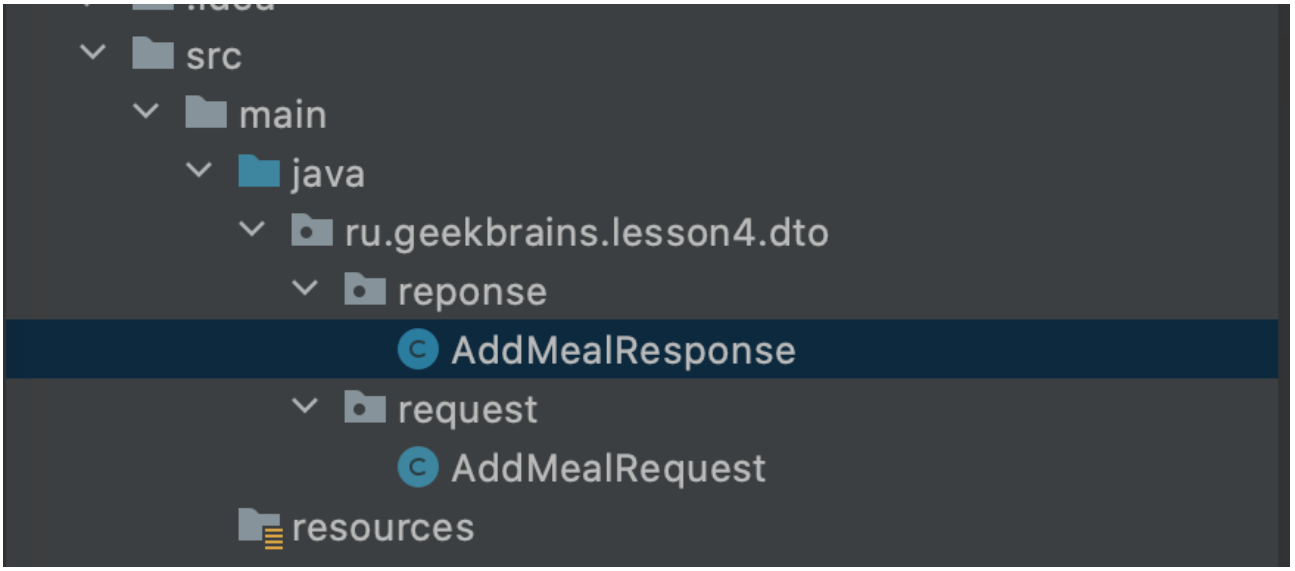
```
})
@Generated("jsonschema2pojo")
public class AddMealResponse {

    @JsonProperty("status")
    public String status;
    @JsonProperty("id")
    public Integer id;

}
```

Аннотации Jackson

В этом POJO-классе используется несколько аннотаций Jackson, разберём их подробнее:

1. **@JsonProperty** ставится над полем, которое надо сериализовать или десериализовать. В скобках указывается строковое значение, которое представлено в итоговом или исходном JSON.
2. **@JsonInclude(JsonInclude.Include.NON_NULL)** используется во время сериализации. Настройка Include.NON_NULL показывает, что null-поля не будут упомянуты в конечном JSON.
3. **@JsonPropertyOrder** применяется, если важен порядок свойств. Эта аннотация указывается в ({}) — порядок появления полей.

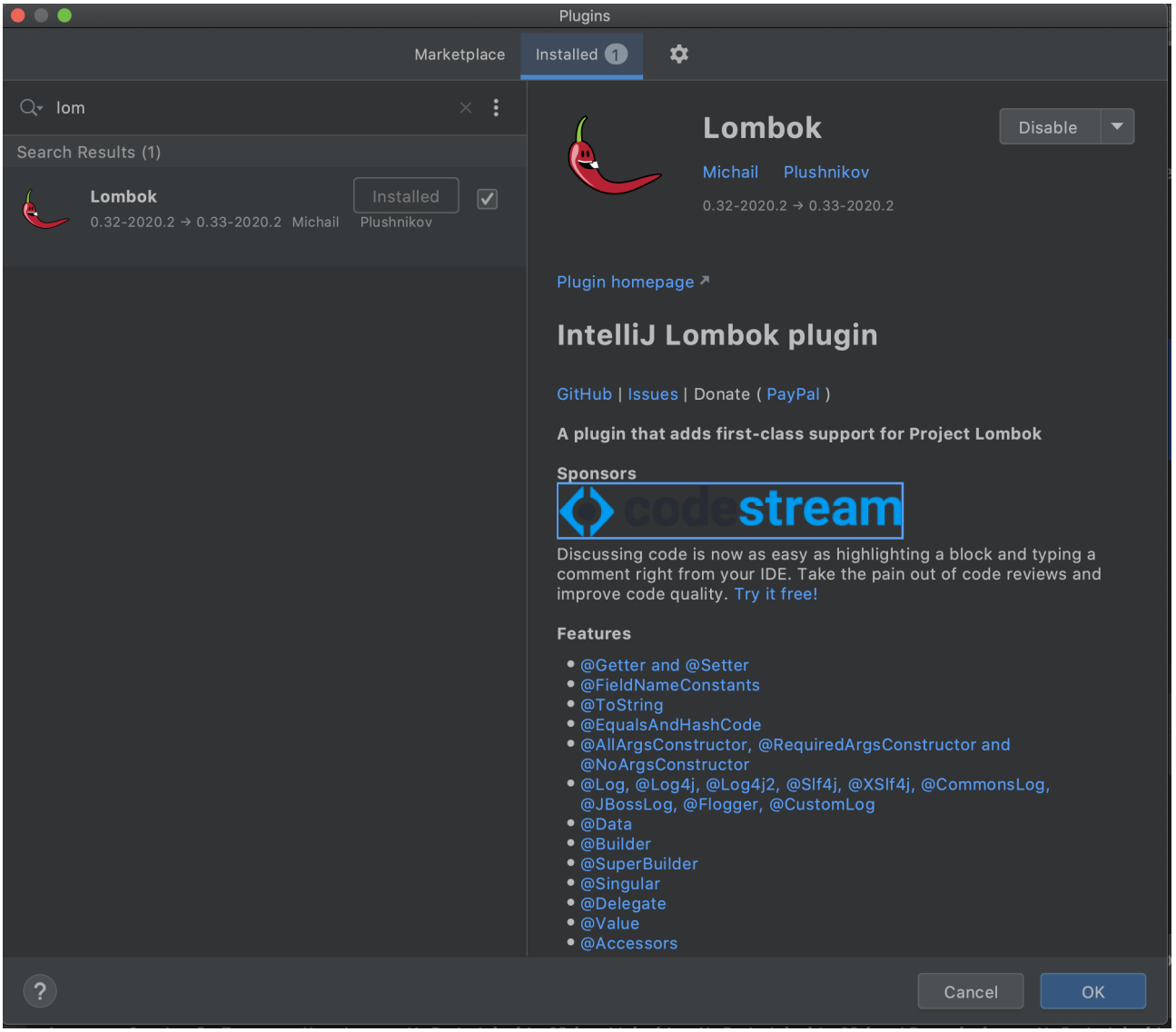
Такой POJO-класс ещё называется DTO (Data Transfer Object) и относится к слою данных. Обычно они хранятся в src/main/java в отдельном пакете, который обычно называется data, dto или иначе.



Плагин Lombok

Чтобы «спрятать» билдеры, геттеры, сеттеры или конструктор, в примере используется аннотация Data lombok-плагина. Для использования lombok требуется прописать его в зависимости pom.xml и установить плагин для IntelliJ IDEA:

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <version>${lombok.version}</version>
</dependency>
```



Вернёмся к нашему тесту и попытаемся «вытащить» значение из полученного ответа, чтобы его проверить:

```
@Test
void getAccountInfoWithExternalEndpointTest() {
   AccountInfoResponse response =
```

```
            given()
                    .pathParam("username", username)
                    .when()
                    .get("/account/" + username)
                    .then()
                    .extract()
                    .body()
                    .as(AccountInfoResponse.class);
    assertThat(response.getStatus(), equalTo(200));
    assertThat(response.getData().getBio(), containsString("system Electronics
Handcrafted migration Dakota"));
    assertThat(response.getData().getUrl(), containsString(username));
}
```

Для десериализации используется конструкция extract().body().as(название класса). Таким же образом можно «вытащить» любые свойства ответа — код ответа, заголовки, куки. Но лучше по возможности использовать встроенные в rest-assured методы — это повысит читаемость кода.

В ассертах используются стандартные геттеры для получения значений полей объекта response, что ничем не отличается от операций с обычными java-объектами.

Практическое задание

Работа со Spoonacular API

1. Отрефакторьте код проверок и добавьте дополнительные тесты для запросов из цепочки Shopping List, используя rest-assured.
2. Воспользуйтесь кейсами из ПЗ № 2 и 3, перенеся всю логику из постман-коллекции в код.
3. Сдайте ссылку на репозиторий, указав ветку с кодом.

Главные критерии для проверки — отсутствие хардкода в коде тестов и наличие тестов на запросы Add to Shopping list (POST /mealplanner/:username/shopping-list/items), Get Shopping List (GET /mealplanner/:username/shopping-list) и Delete from Shopping list (DELETE /mealplanner/:username/shopping-list/items/:id).

Дополнительные материалы

1. Официальная документация.
2. Статья «REST-assured: полезные советы».

Используемые источники

1. Alan Richardson. Automating and Testing a REST API. A Case Study in API testing using: Java, REST Assured, Postman, Tracks, cURL and HTTP Proxies.
2. Официальная документация.
3. Статья REST Assured Tutorial: How to test API with Example.