

universidade de aveiro



estga

escola superior de tecnologia
e gestão de águeda

Testes de Software: Projecto Final

Mestrado Informática Aplicada

Nome do autor:

Jorge Pinto

Docentes:

Joaquim Ferreira e André Campos

Águeda | 09 de Junho de 2024

Índice Geral

| | |
|--------------------------------------|----|
| 1 Introdução..... | 1 |
| 2 Desenvolvimento..... | 2 |
| 2.1 IDE e linguagem programação..... | 3 |
| 2.2 Ferramentas..... | 4 |
| 2.3 Estrutura projecto..... | 5 |
| 2.4 Testes..... | 8 |
| 3 Resultados..... | 10 |
| 4 Conclusões..... | 11 |

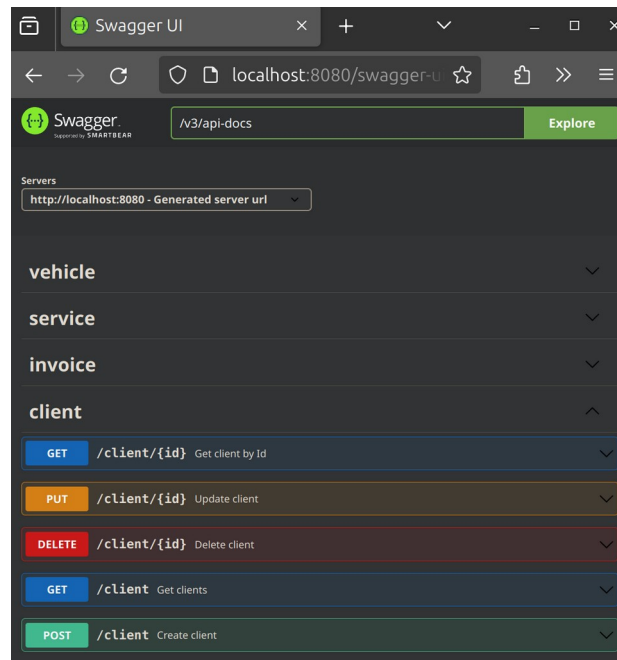
1 Introdução

Este trabalho incide na implementação de testes funcionais que têm como objectivo testar uma API que está em desenvolvimento. Esta API tem como propósito o processamento de informação de uma oficina de carros e contém dados de clientes, veículos, reparações e faturas bem, como as associações entre eles.

2 Desenvolvimento

Foi fornecida uma API implementada em Java, acessível em <http://localhost:8080/> e com a documentação disponível em <http://localhost:8080/swagger-ui/index.html>.

Esta API disponibiliza 4 endpoints, e estão visíveis na interface gráfica do Swagger, que permite executar testes manuais, executando chamadas com diferentes inputs e mostrando os diferentes outputs.



Tendo em conta que por limitações de tempo não seria possível testar todos os endpoints, decidiu-se testar o endpoint *client* por ter uma estrutura de dados bastante familiar.

O código desenvolvido encontra-se no Github: https://github.com/casainho/ProjetoFinal-TESTES_API

2.1 IDE e linguagem programação

A implementação dos testes à API foi feita em Java, recorrendo ao projecto exemplo disponibilizado com o enunciado.

Como IDE, inicialmente foi testado o VSCode mas posteriormente foi seleccionado o [IntelliJ IDEA](#) devido à sua facilidade de uso para desenvolvimento em Java.

2.2 Ferramentas

Foram usadas várias frameworks que aceleraram a implementação:

- [Retrofit](#): implementa chamadas HTTP à API.
- [Jackson](#): converte objectos JSON em objectos Java.
- [TestNG](#): funcionalidades mais fáceis de usar que o Junit.
- [Hamcrest](#): implementa expressões de teste tais como `assertThat()` and `is()`.
- [Lombok](#): gera automaticamente getters, setters e outros métodos comuns.

Recorreu-se frequentemente ao chatbot [Gemini](#) para esclarecimento de erros do código e identificação de possíveis soluções, entre outros.

2.3 Estrutura projecto

A estrutura base do projecto está de acordo com a estrutura do projecto exemplo disponibilizado e contém as seguintes partes:

- **api/calls**
 - A classe ClientCalls em ClientCalls.java contém uma interface com métodos que implementam chamadas HTTP REST à API. A implementação dos métodos é feita pela Retrofit.

```
public interface ClientCalls { 4 usages  ▲ casainho
    String CLIENT = "client"; 3 usages
    String CLIENT_ID = CLIENT +("/{id}"; 1 usage
    String ID = "id"; 1 usage

    @GET(CLIENT) 1 usage  ▲ casainho
    Call<List<ClientWithVehicles>> getClients();

    @POST(CLIENT) 1 usage  ▲ casainho
    Call<Integer> createClient(@Body Client client);

    @DELETE(CLIENT_ID) 1 usage  ▲ casainho
    Call<ResponseBody> deleteClient(@Path(ID) Integer id);
}
```

- **api/mappings**

A classe Client em ClientCalls.java contém objectos Java que representam os dados associados a cada cliente e seus carros. Aqui é usado o Lombok para implementar os métodos de get e set para cada objectos / variável, assim como também é usado o Jackson para converter os objectos Java para JSON e vice-versa – os dados trocados com a API sobre o Client, estão no formato JSON.

```
@Data 11 usages  ▲ casainho
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Client {

    @JsonProperty("id")
    private Integer id;
    @JsonProperty("firstName")
    private String firstName;
    @JsonProperty("lastName")
    private String lastName;
    @JsonProperty("address")
    private String address;
    @JsonProperty("postalCode")
    private String postalCode;
    @JsonProperty("city")
    private String city;
    @JsonProperty("country")
    private String country;
```

- **api/retrofit**

- A classe Clients em Clients.java contém métodos que permitem comunicar com a API. A implementação destes métodos depende das implementações anteriores api/calls e api/mappings.

```
public class Clients { 13 usages  ▲ casainho
    public ClientCalls clientCalls = new RetrofitBuilder().getRetrofit().create(ClientCalls.class); 3 usages

    // getAllClients
    @SneakyThrows 2 usages  ▲ casainho
    public Response<List<ClientWithVehicles>> getAllClientsWithVehicles() {
        return clientCalls.getClients().execute();
    }

    // postClient
    @SneakyThrows 2 usages  ▲ casainho
    public Response<Integer> createClient(Client client) {
        return clientCalls.createClient(client).execute();
    }

    // deleteClient
    @SneakyThrows 1 usage  ▲ casainho
    public Response<ResponseBody> deleteClient(int id) {
        return clientCalls.deleteClient(id).execute();
    }
}
```

- api/validators

- A classe ResponseValidator em ResponseValidator.java, contém métodos que implementam as expressões de teste assertThat() and is() - é usado o Hamcrest.

```
@NoArgsConstructor(access = AccessLevel.PRIVATE) 4 usages  ▲ casainho
public class ResponseValidator {

    public static void assertOk(Response response) { 4 usages  ▲ casainho
        assertThat(String.format("Expected response code to be [%s] but was [%s]", HTTP_OK, response.code()), response.code(), is(HTTP_OK));
    }

    public static void assertCreated(Response response) { 2 usages  ▲ casainho
        assertThat(String.format("Expected response code to be [%s] but was [%s]", HTTP_CREATED, response.code()), response.code(), is(HTTP_CREATED));
    }
}
```

- tests/GarageAPI

- Foram criadas várias classes com o objectivo de testarem as chamadas à API para receber a listagem de todos os clientes e, criar ou apagar um cliente.

```
@Test(description = "Get all clients with success", enabled = true)  △ casainho
public void getAllClientsWithVehiclesTest() {
    Response<List<ClientWithVehicles>> response = new Clients().getAllClientsWithVehicles();
    assertNotNull(response);

    assertThat(reason: "Body should not be null", response.body(), notNullValue());
    List<ClientWithVehicles> clientWithVehicles = response.body();

    // Client 1
    ClientWithVehicles client1 = clientWithVehicles.get(0);

    // ID
    int id = 1;
    assertThat(String.format("id should be %d", id), client1.getId(), is(id));

    // First name
    String name = "Afonso";
    assertThat(String.format("First name should be %s", name), client1.getFirstName(), is(name));

    // Phone Number
    assertThat(reason: "Phone number must be an Integer", client1.getPhoneNumber(), isA(Integer.class));
    assertThat(reason: "Phone number must be positive", client1.getPhoneNumber(), is(greaterThan(value: 0)));
}
```

2.4 Testes

Foram implementados testes ao endpoint Client, nomeadamente aos verbos GET, POST e DELETE, para listar, criar e apagar utilizadores, respectivamente.

Foram criados testes positivos e negativos para os verbos GET e POST, e por falta de tempo, não criados os negativos para o DELETE.

Para o GET, não é necessário enviar nenhum input para a API e é recebida uma lista de objectos do tipo Client.

O teste implementado `getAllClientsWithVehiclesTest()` valida que existe uma resposta e que têm conteúdo. Depois é validado que o conteúdo tem um objecto do tipo Client e são validados os dados esperados de alguns campos, como o Client ID, first name, número de telefone (validado que é um inteiro e não negativo).

Como verificado no Swagger, a resposta inclui uma lista de dois clientes e cada um deles com um ou dois carros – estas condições foram também validadas.

Para o POST, foi criado um objecto Client com dados hardcoded. O objecto foi enviado para o endpoint e foi posteriormente validado o body da resposta, que tem de ser não nulo no caso de sucesso e o seu valor é o user ID do cliente criado pela API.

Neste classe `PostClientPositiveTests()` previram-se vários métodos de testes em que teria sempre de ser criado objecto de um cliente, assim, foram criados os métodos com anotações `@BeforeClass` e `@AfterClass`, the criam o cliente que depois será utilizado por todos os métodos de teste e finalmente será apagado chamando o DELETE da API. Para chamar o DELETE é necessário dar como input à API o client ID e por essa razão é guardado na variável `clientId`.

O teste negativo `createClient()`, tenta criar um cliente com ID null, ao qual a API devolve o código de erro 400.

Para o DELETE, foi criado o teste positivo que envia para a API o user ID 0. Porque este user existe, a API apaga-o e devolve o código 204.

Para correr os testes, tem de ser apagada previamente a pasta `tmp` criada pela API `garage.jar`, que contém dados de clientes, veículos, etc. Esta função foi integrada no Run/Debug do IDE do seguinte modo:

Edit Tool

Name: Remove API database

Group: External Tools

Description:

Tool Settings

Program: /usr/bin/rm

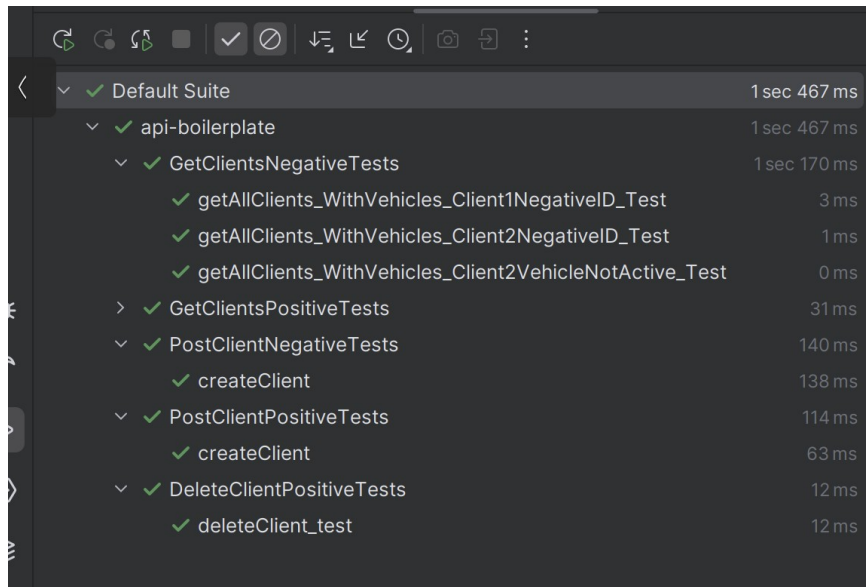
Arguments: -r ../ProjetoFinal-RUN_API/tmp

Working directory: D4_Mestrado_Informatica_Aplicada/03-TS/TESTES-Projecto_final/ProjetoFinal-TESTES_API

> Advanced Options

3 Resultados

Os resultados dos testes (assim como o seu tempo de execução) foram todos positivos, como se pode ver na imagem seguinte, que foi gerada pelo IDE, indicando assim que não foi encontrado nenhum erro na API.



The screenshot shows a test results window from an IDE. It displays a tree view of test results for a suite named 'Default Suite'. The suite and all its sub-items are marked with green checkmarks, indicating successful execution. The execution times for each item are listed on the right side of the tree.

| Test Suite / Test Case | Execution Time |
|---|----------------|
| Default Suite | 1 sec 467 ms |
| api-boilerplate | 1 sec 467 ms |
| GetClientsNegativeTests | 1 sec 170 ms |
| getAllClients_WithVehicles_Client1NegativeID_Test | 3 ms |
| getAllClients_WithVehicles_Client2NegativeID_Test | 1 ms |
| getAllClients_WithVehicles_Client2VehicleNotActive_Test | 0 ms |
| GetClientsPositiveTests | 31 ms |
| PostClientNegativeTests | 140 ms |
| createClient | 138 ms |
| PostClientPositiveTests | 114 ms |
| createClient | 63 ms |
| DeleteClientPositiveTests | 12 ms |
| deleteClient_test | 12 ms |

Durante o desenvolvimento foram identificados dois possíveis erros durante os testes. O primeiro é o facto de que a API não cria um novo utilizador se a data de nascimento de cliente não for igual a data de criação do cliente – testou-se com data de nascimento igual à data de criação do cliente menos 18 anos e obteve-se erro. O outro erro acontece quando o NIF do cliente começa pelo número 2 (pessoa singular) mas não acontece quando começa por 5 (pessoa colectiva).

Foram feitas algumas tentativas para configurar o fluxo a não parar quando quando existe um erro, mas não tendo sido possível, optou-se como colocar os testes todos positivos de modo a ter o relatório de testes completo.

4 Conclusões

Neste trabalho foi possível testar uma API recorrendo à linguagem de programação Java e várias ferramentas, como indicado nos respectivos capítulos.

A falta de experiência pelo autor em Java e no IDE selecionado, tornou o desenvolvimento lento e limitou assim o número de testes que foram implementados.

Para uma futura continuação deste trabalho, recomenda-se descrever cada caso de teste e identificar as condições iniciais. Também, cobrir com mais testes cada campo de dados como no caso do cliente e identificar os testes em que a API falha.