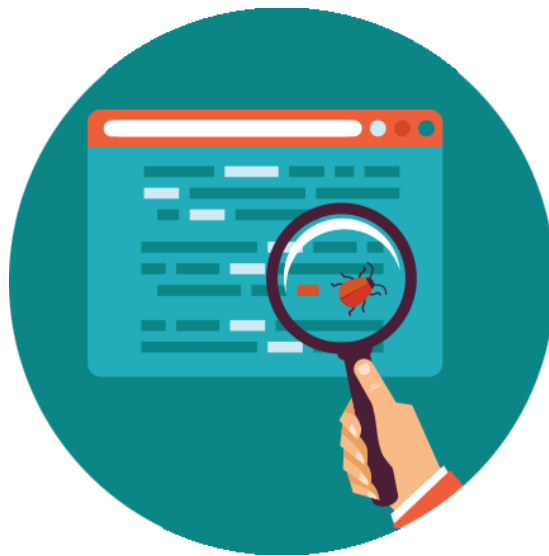


Relatório do Projeto de Testes de Software

Desenvolvimento de testes

2021-2022

Link GitHub do Projeto: <https://github.com/oliveirasmj/TestesSoftware>



Escola Superior de Tecnologias e Gestão de Águeda
Testes de Software
2º semestre | 1º ano
Miguel Oliveira

1. Índice

1. Índice	2
2. Descrição do objeto de teste.....	3
3. Objetivo dos testes	5
4. Estratégia e Casos de Teste	6
4.1. Cenário de testes.....	6
4.1.1. Cenário 1 – Procurar um cliente por um ID que existe (teste positivo)	6
4.1.2. Cenário 2 – Procurar um cliente por um ID que não existe (teste negativo)	6
4.1.3. Cenário 3 – Criar um cliente válido (teste positivo)	7
4.1.4. Cenário 4 – Criar um cliente inválido (teste negativo)	7
4.1.5. Cenário 5 – Remover um cliente válido (teste positivo)	8
4.1.6. Cenário 6 – Atualizar um cliente válido (teste positivo).....	9
4.1.7. Cenário 7 – Atualizar um cliente inválido (teste negativo).....	9
4.1.8. Cenário 8 – Procurar um veículo pelo ID (teste positivo).....	10
4.1.9. Cenário 9 – Procurar um veículo por um ID que não existe (teste negativo) ...	10
4.1.10. Cenário 10 – Criar um veículo (teste positivo)	11
4.1.11. Cenário 11 – Criar um veículo inválido (teste negativo).....	11
4.2. Ambiente de testes	12
5. Execução e resultados obtidos	13
5.1. Testes de integração	13
5.2. Testes unitários	14
6. Discussão/Avaliação dos resultados	15

2. Descrição do objeto de teste

Os objetos de teste escolhidos para este projeto são endpoints presentes na API de nome Garage. Esta API encontra-se em desenvolvimento e tem como propósito o processamento de informação de uma oficina. Todos estes objetos estão relacionados contém dados de clientes, veículos, reparações e faturas bem como as associações entre eles. Os objetos de teste representam algumas das operações mais comuns, desde a criação, atualização e listagem dos veículos e clientes, além de outras funcionalidades disponibilizadas e que é possível visualizar através do Swagger. Outros endpoint estão relacionados com faturas e serviços e que estão relacionados com os restantes mencionados anteriormente.

Os endpoint disponibilizados pela API estão indicados na figura seguinte:

The image shows a Swagger API documentation interface with four resource sections: vehicle, client, service, and invoice. Each section lists endpoints with their HTTP methods and descriptions.

Resource	Method	Endpoint	Description
vehicle	PUT	/vehicle/{vehicleId}/client/{clientId}	Add vehicle to client
	GET	/vehicle/{id}	Get vehicle by id
	PUT	/vehicle/{id}	Update vehicle
	DELETE	/vehicle/{id}	Delete vehicle
	GET	/vehicle	Get vehicles
	POST	/vehicle	Create vehicle
	DELETE	/vehicle/{vehicleId}/client/	Remove service from vehicle
client	GET	/client/{id}	Get client by id
	PUT	/client/{id}	Update client
	DELETE	/client/{id}	Delete client
	GET	/client	Get clients
	POST	/client	Create client
service	PUT	/service/{serviceId}/invoice/{invoiceId}	Add service to invoice
	GET	/service/{id}	Get service by id
	PUT	/service/{id}	Update service
	DELETE	/service/{id}	Delete service
	GET	/service	Get services
	POST	/service	Create service
	DELETE	/service/{serviceId}/invoice/	Remove service from invoice
invoice	GET	/invoice/{id}	Get invoice by id
	PUT	/invoice/{id}	Update invoice
	DELETE	/invoice/{id}	Delete invoice
	GET	/invoice	Get invoices
	POST	/invoice	Create invoice
	GET	/invoice/vehicle/{vehicleId}	Get invoices by vehicle

Foi utilizada a versão de Java 17. Ao executar a aplicação é criada uma pasta “tmp” onde são encontrados os ficheiros referentes à base de dados H2, utilizada pela API. Estes ficheiros são utilizados com memória do sistema de base de dados. A API fica acessível através do endereço <http://localhost:8080/> e a documentação da mesma está disponível em

<http://localhost:8080/swagger-ui/index.html>, conforme mencionado acima na referência do Swagger.

Além da tarefa relacionada com os testes de integração existe uma classe de Java, que contém métodos de suporte ao futuro módulo de descontos da API. Este módulo ainda não foi implementado na sua totalidade, pelo que não será possível efetuar testes de integração sobre o mesmo. Ao invés disso deverão ser implementados testes unitários necessários e adequados para o teste da classe.

Os objetos de teste são descritos nos tópicos seguinte:

Request	Nome	Path	Link	HTTP	Descrição
Client	getClientById	/cliente/{id}	http://localhost:8080/swagger-ui/index.html#/client/getClientById	GET	Get cliente by id
Client	createClient	/cliente	http://localhost:8080/swagger-ui/index.html#/client/createClient	POST	Create client
Client	removeClient_1	/cliente/{id}	http://localhost:8080/swagger-ui/index.html#/client/removeClient_1	DELETE	Delete client
Client	updateClient	/cliente/{id}	http://localhost:8080/swagger-ui/index.html#/client/updateClient	PUT	Update client
Vehicle	getVehicleRepositoryById	/vehicle/{id}	http://localhost:8080/swagger-ui/index.html#/vehicle/getVehicleRepositoryById	GET	Get vehicle by id
Vehicle	createVehicle	/vehicle	http://localhost:8080/swagger-ui/index.html#/vehicle/createVehicle	POST	Create vehicle

3. Objetivo dos testes

Cada cliente e cada veículo é identificado por um número que lhe é atribuído no momento da sua criação. Este número é incrementado cada vez que se cria um novo.

O objetivo da realização destes é garantir que é possível interagir com a API sem que haja falhas, ou detetar falhas antecipadamente antes das aplicações serem colocadas em produção. Existe o objetivo de explorar os objetos de teste e verificar que se comportam como esperado tanto para casos positivos quanto para casos negativos.

Além disso quando se faz um pedido com a informação correta ou incorreta ou fazendo um pedido sob determinadas condições é retornada uma resposta que pode variar de acordo com o tipo de erro ou de sucesso.

Os códigos de status das respostas HTTP indicam se uma requisição HTTP foi corretamente concluída. As respostas são agrupadas em cinco classes:

1. Respostas de informação (100-199),
2. Respostas de sucesso (200-299),
3. Redireccionamentos (300-399)
4. Erros do cliente (400-499)
5. Erros do servidor (500-599).

Existem diversos métodos HTTP relacionados com o protocolo responsável pela comunicação de sites na web. O método GET solicita a representação de um recurso específico. As requisições que utilizam o método GET devem retornar apenas dados. O método POST é utilizado para submeter uma entidade a um recurso específico. É utilizado para mudanças no estado do recurso. O método DELETE remove um recurso específico. O método PUT substitui todas as atuais representações do recurso de destino pela carga de dados da requisição. O método PATCH é utilizado para aplicar modificações parciais num recurso. A grande diferença entre o PUT e o PATCH é que o PATCH só é necessário indicar os que são para alterar e os restantes valores permanecem iguais.

A criação deste tipo de testes tem como objetivo ainda que seja feito o máximo de cobertura possível relativamente aos objetos de teste utilizados neste projeto. Ao fazer este tipo de testes é possível garantir que os objetos funcionam como o esperado durante todo o processo de desenvolvimento da aplicação. Dessa forma garante-se que caso os testes falhem que os responsáveis pelo desenvolvimento serão notificados através desses alertas para que algo deva ser modificado por estar perante um acontecimento não esperado. Isto permite que não sejam lançadas versões de software com erros que podem ser evitados ou detetados com antecedência para que não prejudique o ambiente de produção.

4. Estratégia e Casos de Teste

4.1. Cenário de testes

Os casos de teste definidos para este projeto foram criados com base nos cenários descritos anteriormente, simulando as operações que um utilizador faria na interação com os objetos de teste. Estes casos de teste foram também definidos tendo em conta algumas situações que possam acontecer para além das que são esperadas e mais habituais.

4.1.1. Cenário 1 – Procurar um cliente por um ID que existe (teste positivo)

Neste cenário para que seja possível pesquisar um cliente é pedido como parâmetro ao utilizador um id do tipo integer através de um método GET. É esperado neste caso que seja devolvido uma operação de sucesso através de um código 200, onde é retornado um JSON com toda a informação sobre o cliente. Esse objeto tem toda a informação pessoal sobre ele tal como uma lista de veículos que o mesmo possui no caso de ter. No caso desse ID não existir é esperado que seja retornado um “404 – Client not found”.

Nome	CT01 – Procurar um cliente por um ID que existe (teste positivo)
Descrição	Validar que o cliente com um ID existe com a informação correta que é passada
Prioridade	Alta
Condições iniciais	Tem que existir um cliente com o ID a procurar e com valores
Procedimento	Realizar um GET
Dados de Entrada	Valor do id a procurar como parâmetro
Resultado Esperado	Que os valores esperados sejam iguais aos valores do GET e com o código 200
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.2. Cenário 2 – Procurar um cliente por um ID que não existe (teste negativo)

Um teste negativo consiste em verificar que perante uma falha a resposta é a adequada ao tipo de erro que é recebido relativamente ao pedido efetuado. Neste caso o objetivo será procurar um cliente que não existe e no qual se espera um erro “404 – Client not found”.

Nome	CT02 – Procurar um cliente por um ID que não existe (teste negativo)
Descrição	Validar que o cliente com um ID que não existe retorna a informação adequada
Prioridade	Alta

Condições iniciais	Não pode existir nenhum cliente com o ID a procurar
Procedimento	Realizar um GET
Dados de Entrada	Valor do id a procurar como parâmetro
Resultado Esperado	Esperar um 404 – Client not found
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.3. Cenário 3 – Criar um cliente válido (teste positivo)

Para este cenário, perante a criação de um cliente através de um POST é esperado que seja enviado um objeto através de um “Request Body”. Essa informação deve estar construída em formato de JSON e deve ter o “firstName”, “lastName”, “address”, “postalCode”, “city”, “country” em formato string. Os restantes são o “phoneNumber” e o “nif” em formato integer. Restam depois o “birthDate” e o “clientDate” que deverão ser string em formato de data, segundo o exemplo “AAAA-MM-DD”. É esperado neste caso que seja devolvido uma operação de sucesso através de um código 200, onde é retornado um ID. No caso de não conseguir ser criado é esperado um código “400 – Bad Request”.

Nome	CT03 – Criar um cliente válido (teste positivo)
Descrição	Validar que o cliente foi criado de forma correta na BD
Prioridade	Alta
Condições iniciais	Não necessita de pré-condições
Procedimento	Realizar um POST
Dados de Entrada	Request body com toda a informação do cliente que é required
Resultado Esperado	Que os valores do cliente criado sejam iguais aos valores do POST e com o código 200 e que seja retornado o ID do cliente
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.4. Cenário 4 – Criar um cliente inválido (teste negativo)

Um teste negativo consiste em verificar que perante uma falha a resposta é a adequada ao tipo de erro que é recebido relativamente ao pedido efetuado. Neste caso o objetivo será criar um cliente inválido e no qual se espera um erro “400 – Bad Request”. Para criar um cliente inválido pode ser usado um NIF errado, por exemplo.

Nome	CT04– Criar um cliente inválido (teste negativo)
Descrição	Validar que criar um cliente inválido retorna a informação adequada
Prioridade	Alta
Condições iniciais	Não necessita de pré-condições
Procedimento	Realizar um POST
Dados de Entrada	Objeto cliente com NIF inválido
Resultado Esperado	Esperar um 400 – Bad Request
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.5. Cenário 5 – Remover um cliente válido (teste positivo)

No cenário relacionado com a remoção de um cliente é esperado que seja enviado um ID do tipo integer através de um método do tipo DELETE. No caso do ID do cliente existir é esperado um código “204 – Successful operation” Caso o ID não exista é esperado um “404 – Client not found”. É ainda opcional que seja indicado se é desejável que sejam removidos os veículos ao remover o proprietário dos mesmos. Este parâmetro é indicado através de um boolean, sendo a opção possível um true ou um false.

Nome	CT05 – Remover um cliente válido (teste positivo)
Descrição	Validar que o cliente foi removido de forma correta na BD
Prioridade	Alta
Condições iniciais	É necessário que o cliente já exista na BD
Procedimento	Realizar um DELETE
Dados de Entrada	Valor do id a procurar como parâmetro
Resultado Esperado	204 – Successful operation
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.6. Cenário 6 – Atualizar um cliente válido (teste positivo)

No exemplo do cenário responsável por atualizar a informação de um cliente é esperado como parâmetro um ID do tipo integer correspondente ao ID do cliente do qual é desejável atualizar a sua informação. Como “Request Body” deve ser indicado um objeto com a propriedade completa acerca de um cliente. Deverão ser indicados todos os campos à exceção do ID no “Request Body” visto que se está perante um método PUT e não de um método do tipo PATCH. Os códigos esperados serão “200 - successful operation” caso o utilizador seja modificado com sucesso. O código “400 – Bad Request” é esperado caso ocorra um erro enquanto o “404 - Client not found” caso o ID passado não exista.

Nome	CT06 – Atualizar um cliente válido (teste positivo)
Descrição	Validar que a atualização do cliente foi resolvida com sucesso
Prioridade	Alta
Condições iniciais	É necessário que o cliente já exista na BD
Procedimento	Realizar um PUT
Dados de Entrada	Valor do id a procurar como parâmetro e objeto do cliente completo no Request Body
Resultado Esperado	200 - successful operation
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.7. Cenário 7 – Atualizar um cliente inválido (teste negativo)

Um teste negativo consiste em verificar que perante uma falha a resposta é a adequada ao tipo de erro que é recebido relativamente ao pedido efetuado. Neste caso o objetivo será atualizar um cliente inválido e no qual se espera um erro “400 – Bad Request”. Para atualizar um cliente inválido é necessário indicar objeto com informações que não passem na validação de dados, tais como o número de telefone com menos de 9 dígitos.

Nome	CT07 – Atualizar um cliente inválido (teste negativo)
Descrição	Validar que criar um atualizar um cliente inválido retorna a informação adequada
Prioridade	Alta
Condições iniciais	É necessário que o cliente já exista na BD
Procedimento	Realizar um PUT
Dados de Entrada	Valor do id a procurar como parâmetro e objeto do cliente completo no Request

	Body mas com número de telefone com menos de 9 digitos
Resultado Esperado	400 – Bad Request
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.8. Cenário 8 – Procurar um veículo pelo ID (teste positivo)

Neste cenário para que seja possível pesquisar um veículo é pedido como parâmetro ao utilizador um id do tipo integer através de um método GET. É esperado neste caso que seja devolvido uma operação de sucesso através de um código 200, onde é retornado um JSON com toda a informação sobre o veículo. Esse objeto tem toda a informação do carro sobre que se pretende pesquisar através do valor como parâmetro. No caso desse ID não existir é esperado que seja retornado um “404 – Vehicle not found”.

Nome	CT08 – Procurar um veículo por um ID que existe (teste positivo)
Descrição	Validar que o veículo com um ID existe com a informação correta que é passada
Prioridade	Alta
Condições iniciais	Tem que existir um veículo com o ID a procurar e com valores
Procedimento	Realizar um GET
Dados de Entrada	Valor do id a procurar como parâmetro
Resultado Esperado	Que os valores esperados sejam iguais aos valores do GET e com o código 200
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.9. Cenário 9 – Procurar um veículo por um ID que não existe (teste negativo)

Um teste negativo consiste em verificar que perante uma falha a resposta é a adequada ao tipo de erro que é recebido relativamente ao pedido efetuado. Neste caso o objetivo será procurar um veículo que não existe e no qual se espera um erro “404 – Vehicle not found”.

Nome	CT09 – Procurar um veículo por um ID que não existe (teste negativo)
Descrição	Validar que o veículo com um ID que não existe retorna a informação adequada

Prioridade	Alta
Condições iniciais	Não pode existir nenhum veículo com o ID a procurar
Procedimento	Realizar um GET
Dados de Entrada	Valor do id a procurar como parâmetro
Resultado Esperado	Esperar um 404 – Vehicle not found
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.10. Cenário 10 – Criar um veículo (teste positivo)

Para este cenário, perante a criação de um veículo através de um POST é esperado que seja enviado um objeto através de um “Request Body”. Essa informação deve estar construída em formato de JSON e deve ter o “brand”, “model”, “type”, “plate” em formato string. Os restantes são o “year” em formato integer. Resta depois o “active” que é do tipo boolean. É esperado neste caso que seja devolvido uma operação de sucesso através de um código 200, onde é retornado um ID. No caso de não conseguir ser criado é esperado um código “400 – Bad Request”.

Nome	CT10 – Criar um veículo válido (teste positivo)
Descrição	Validar que o veículo foi criado de forma correta na BD
Prioridade	Alta
Condições iniciais	Não necessita de pré-condições
Procedimento	Realizar um POST
Dados de Entrada	Request body com toda a informação do veículo que é required
Resultado Esperado	Que os valores do veículo criado sejam iguais aos valores do POST e com o código 200 e que seja retornado o ID do veículo
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.1.11. Cenário 11 – Criar um veículo inválido (teste negativo)

Um teste negativo consiste em verificar que perante uma falha a resposta é a adequada ao tipo de erro que é recebido relativamente ao pedido efetuado. Neste caso o objetivo será criar um veículo inválido e no qual se espera um erro “400 – Bad Request”. Para criar um veículo inválido pode ser usada uma matrícula errada, por exemplo.

Nome	CT11 – Criar um cliente inválido (teste negativo)
Descrição	Validar que criar um cliente inválido retorna a informação adequada
Prioridade	Alta
Condições iniciais	Não necessita de pré-condições
Procedimento	Realizar um POST
Dados de Entrada	Objeto cliente com matrícula inválida
Resultado Esperado	Esperar um 400 – Bad Request
Sucesso	Passou
Resultado Atual	Como esperado
Ambiente	Teste local
Técnica	Automático

4.2. Ambiente de testes

Neste projeto foram utilizados tipo de testes automatizados através de um ambiente de testes baseado num projeto com o requisito de utilização do Java 17. Foram também utilizadas diversas ferramentas para a conceção do projeto:

- TestNG → framework de testes inspirada no JUnit e com mais funcionalidades
- Retrofit → usado estruturar e configurar o cliente REST que interage com a API a ser alvo dos testes via pedidos HTTP
- Hamcrest → framework de matchers(regras) utilizadas nas validações dos testes
- Lombok → biblioteca do Java que facilita a escrita de código nesta linguagem
- IntelliJ → IDE utilizada para o desenvolvimento do projeto
- GitHub → plataforma de hospedagem de código e arquivos com controlo de versão que faz uso do Git

5. Execução e resultados obtidos

Foram criados testes automatizados para cada um dos casos de teste definidos no ponto anterior. Estes testes automatizados estão localizados no package “api.integration”. Dentro desse package existem mais dois designados de “client” e “vehicle” sendo que cada um possui testes negativos e testes positivos.

Foram criados dois métodos que são executados antes e depois de cada teste: “setupXXX” e “cleanupXXX”. O objetivo destes métodos é criar o objeto inicial para ser usado na BD e posteriormente eliminar após a execução do teste. Deste modo não ficará a ocupar memória na BD visto que o objetivo era apenas testar.



Além disso também é garantido que todos os testes são independentes uns dos outros. Isto garante que é possível correr cada método individualmente sem que estejam dependentes dos resultados de outros testes.

5.1. Testes de integração

Os resultados obtidos foram positivos, sendo que 14 dos 14 casos de teste passaram com os resultados esperados, incluindo os positivos e os negativos.

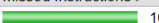
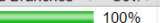

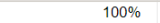


api-MiguelOliveira_83816 > Discounts

Discounts

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Discounts		100%		100%	0	11	0	22	0	3	0	1
Total	0 of 80	100%	0 of 14	100%	0	11	0	22	0	3	0	1

api-MiguelOliveira_83816 > Discounts > Discounts

Discounts

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
discountByAge(Integer)		100%		100%	0	5	0	9	0	1
calculateTotalDiscount(Integer,String,LocalDate)		100%		100%	0	2	0	8	0	1
discountByBrand(String)		100%		100%	0	4	0	5	0	1
Total	0 of 80	100%	0 of 14	100%	0	11	0	22	0	3

Relativamente à cobertura de código é possível analisar que é garantida de forma total conforme indicado na figura seguinte:

```
api-MiguelOliveira_83816 > Discounts > Discounts.java

Discounts.java

1. package Discounts;
2.
3. import java.time.DayOfWeek;
4. import java.time.LocalDate;
5.
6. public class Discounts {
7.
8.     private Discounts() {
9.     }
10.
11.     public static int discountByAge(Integer carYear) {
12.         int age = LocalDate.now().getYear() - carYear;
13.         int discount = 0;
14.
15.         if (age >= 5) {
16.             discount = 2;
17.         }
18.         if (age >= 6 && age < 10) {
19.             discount = discount + age - 3;
20.         } else if (age > 10) {
21.             discount = (discount + age) / 2;
22.         }
23.         return discount;
24.     }
25.
26.     public static int discountByBrand(String brand) {
27.         return switch (brand) {
28.             case "Vw" -> 3;
29.             case "Tesla" -> 5;
30.             case "BMW" -> 7;
31.             default -> 0;
32.         };
33.     }
34.
35.     public static int calculateTotalDiscount(Integer carYear, String brand, LocalDate today) {
36.         int MAX_TOTAL_DISCOUNT = 25;
37.         int PROMOTION_DAY_DISCOUNT = 5;
38.         boolean isPromotionDay = today.getDayOfWeek().equals(DayOfWeek.FRIDAY);
39.         int totalDiscount = discountByAge(carYear) + discountByBrand(brand);
40.
41.         if (isPromotionDay) {
42.             totalDiscount = totalDiscount + PROMOTION_DAY_DISCOUNT;
43.         }
44.
45.         totalDiscount = Math.min(totalDiscount, MAX_TOTAL_DISCOUNT);
46.
47.         return totalDiscount;
48.     }
49. }
50. }
```

5.2. Testes unitários

Os resultados obtidos foram positivos, sendo que 17 dos 17 casos de teste passaram com os resultados esperados.

Run: DiscountsTest x

Tests passed: 17 of 17 tests – 220 ms

C:\Users\olive\.jdk\azul-15.0.7\bin\java.exe ...

Test Results 220 ms

- DiscountsTest 220 ms
 - discountByAge(int, int) 167 ms
 - [1] -5, 0 141 ms
 - [2] 0, 0
 - [3] 4, 0
 - [4] 5, 2 9 ms
 - [5] 6, 5
 - [6] 8, 7
 - [7] 9, 8 8 ms
 - [8] 10, 2 5 ms
 - [9] 11, 6 4 ms
 - discountByBrand(String, int) 4 ms
 - [1] VW, 3 4 ms
 - [2] Tesla, 5
 - [3] BMW, 7
 - [4] Mercedes, 0
 - calculateTotalDiscount(int, String, 49 ms
 - [1] 5, VW, 2022-06-16, 5 47 ms
 - [2] 50, VW, 2022-06-16, 25
 - [3] 50, VW, 2022-06-17, 25 2 ms
 - [4] 5, VW, 2022-06-17, 10

Process finished with exit code 0

6. Discussão/Avaliação dos resultados

Todos os casos de teste passaram e os resultados obtidos foram os esperados, demonstrando assim que os objetos de teste funcionam como esperado e permitem realizar a grande parte das operações.

Por esse motivo é desde já possível garantir que a execução dos casos de teste em ambiente automatizado foi bem-sucedida.

Contudo, como a implementação e realização destes testes foi totalmente eficaz no que toca a sucesso, foi possível verificar os casos de teste poderiam ter sido mais exaustivos ou específicos para que fossem detetadas mais falhas. Seria eventualmente bom, numa altura mais indicada perceber onde a API tem mais lacunas ou até mesmo na classe de descontos que foi disponibilizada. De qualquer das formas é garantido que a API e a classe de descontos têm as condições mínimas para passar numa primeira fase de testes sendo que o ideal seria colocar a aplicação num ambiente de testes antes de entrar em produção de modo a evitar erros mais graves.

É por isso importantíssimo indicar que o projeto não disponibiliza todos os casos de teste possíveis para os objetos em questão. É possível e seria ótimo criar mais casos de teste para todos os diferentes objetos possíveis, tal como determinados cenários mais específicos em que esses objetos possam ser utilizados.

O projeto enviado e concebido contém alguns casos de teste, que no caso são considerados os mais importantes do ponto de vista do utilizador.