

## Relatório TQS HW1: Desenvolvimento de testes

Testes e Qualidade Software

João Fidalgo, 62243

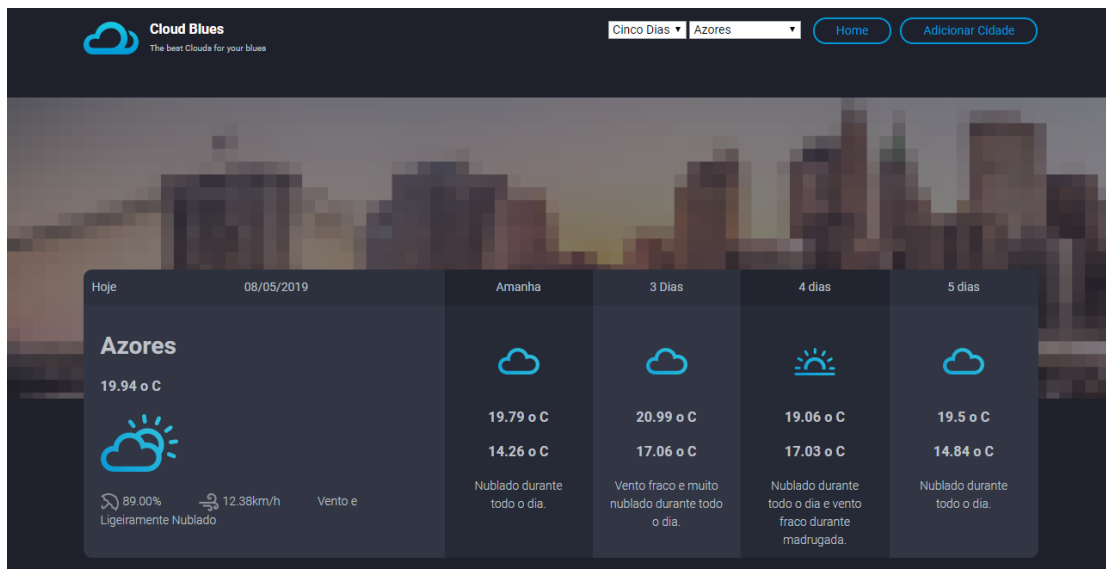
Ilídio Oliveira

Cláudio Teixeira

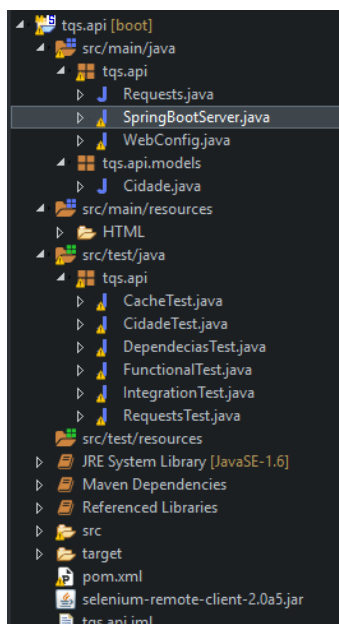
[Bitbucket](#)

[SonarCloud](#)

Front-End:



Package explorer



## Tecnologias adotadas

### Front-end

- Markup e estilos: HTML e CSS
- Manipulação e chamadas à API do localhost: Javascript, Ajax e Jquery
- [Template](#)
- Text editor: Visual Code

### Back-end

- Servidor Java: Spring-boot
- IDE: STS
- Datasource API: [Dark Sky](#)
- SPM: Maven

### Testes

- Unitários: JUnit
- Dependências: Mockito e JUnit
- API: RESTAssured
- Funcional: Selenium
- Métricas de qualidade: SonarCloud

## Estratégia Front-end

Para o front-end, foi tomada a decisão de usar um template simples e apenas manter as funcionalidades mais importantes devido ao escopo do trabalho. Assim, a página do front-end é composta por três áreas principais, os filtros, os menus e o conteúdo da página.

O menu permite adicionar uma cidade, desde que seja fornecido os campos requeridos (nome da cidade, latitude e longitude). Ao adicionar a cidade, é feita uma chamada ao servidor localhost por Ajax que invoca um endpoint para a adição da dita cidade na lista. Ao adicionar, a página é refrescada.

```
function adicionarCidade() {
    const latitude = $("#latitude").val();
    const longitude = $("#longitude").val();
    const cidadeNome = $("#cidadeInput").val();

    if (latitude.length == 0 || longitude.length == 0 || cidadeNome.length == 0) {
        alert("Por favor preencha os campos");
        return false;
    }

    if (!isNaN(parseFloat(latitude)) || !isNaN(parseFloat(longitude))) {
        alert("Por favor insira valores validos");
        return false;
    }

    $.ajax({
        url: "http://localhost:8080/adicionarCidade/" + cidadeNome + "/" + parseFloat(latitude) + "/" + parseFloat(longitude),
        method: "POST", //First change type to method here
        data: { nome: cidadeNome, latitude: parseFloat(latitude), longitude: parseFloat(longitude) },
        success: function (result) {
            alert("Cidade adicionada com sucesso, pode aceder pela dropdownlist");
            window.location.reload();
        },
        error: function (error) {
            console.log(error)
        }
    })
}
```

Existem dois filtros, um para selecionar o número de dias a mostrar a previsão e outro para escolher a cidade que pretende ver informações sobre. O primeiro filtro apenas manipula o DOM, pelo que, não é feito qualquer pedido novo, esta opção foi tomada porque não pareceu relevante para o trabalho fazer uma nova chamada quando a original continha a mesma informação. O segundo filtro é populado quando se entra na página e contém as cidades que estão numa lista no back-end, para tal, é feita uma chamada a API para devolver as cidades. Com esse retorno, a dropdownlist é populada com JQuery.

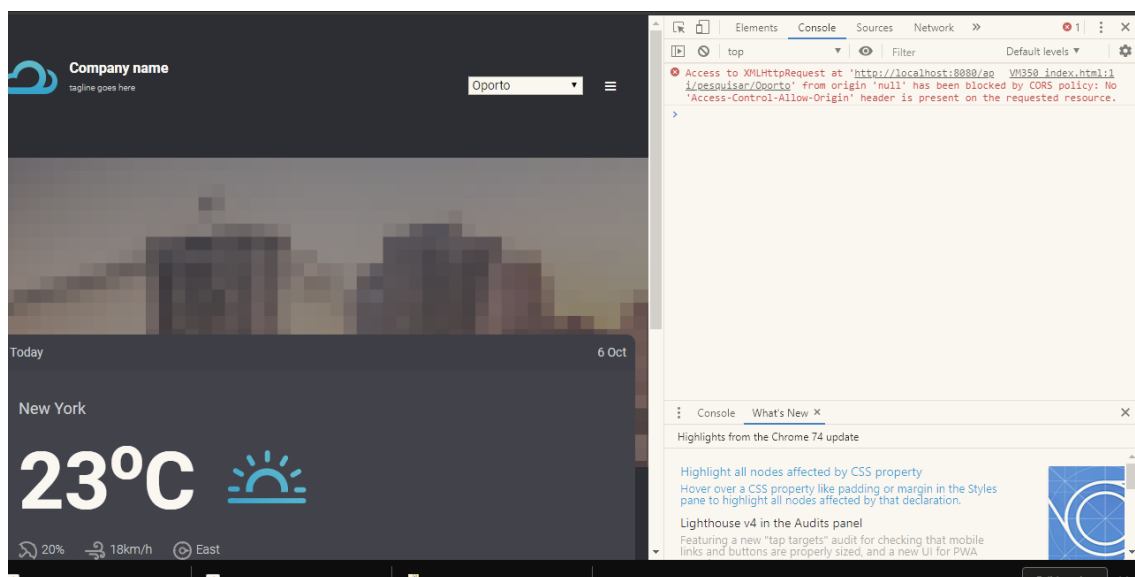
Ao selecionar o segundo filtro, é feita um pedido ao back-end que recolhe informação da API da DarkSky, desta forma, o conteúdo é dinâmico e baseia-se em alguns parâmetros que são passados no URL, como a latitude, longitude, a língua e as escalas. Com estes novos dados, o DOM é atualizado com previsão de três ou cinco dias (dependendo do primeiro filtro) e é mostrada informação sobre a temperatura atual, o estado do tempo (chuva, sol, nublado, etc.), a humidade, etc. e os dias seguintes contêm um sumário, as temperaturas máximas e mínimas e o estado.

## Estratégia back-end

### Servidor

A escolha do servidor passou por uma fase de experimentação, pelo que no início foi adotada uma solução para fazer a API baseada em Jax-RS, no entanto tal opção acabou por não ser a mais desejada devido a complicações relacionadas com CORS. [CORS](#) é um sistema que previne a chamada de API externa a partir da mesma origem, o que acontecia quando era feita a chamada Ajax para o endpoint que fazia outro pedido a API da DarkSky.

Foram feitas várias tentativas de usar esta tecnologia, no entanto, não foi possível chegar a uma solução que resolvesse o problema, pelo que, a decisão tomada foi a de migrar o trabalho feito para o Spring Boot (alguns tutoriais seguidos para tentar resolver o problema [aqui](#) e [aqui](#), no entanto não foi encontrada solução).



Duas das características que levou a escolha do Spring Boot é a [extensa documentação](#) e a facilidade de setup e configuração. Assim, seguindo [este](#) e [este](#) guia, foi possível resolver o problema de CORS que continuava a surgir aquando da migração. Para tal, foi adicionado o decorador `@CrossOrigin` no controlador REST e criada uma classe de configuração que permite chamadas ao localhost no porto desejado.

```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("http://localhost:8080/**");
}
```

```
@CrossOrigin
@RestController
public class Requests {
```

## Classe principal

A classe principal é chamada Cidade e contém três atributos: nome, latitude e longitude. Estes atributos são os necessários para o front-end (especialmente o nome) e fazer as chamadas à API externa (com a latitude e longitude). Esta classe tem os métodos getter e setter, para além de dois construtores, um vazio e outro com os parâmetros referidos.

## Criação da API

Como dito anteriormente, o Spring Boot tem uma forma bastante clara para o seu setup, existe uma classe main que contém a anotação `@SpringBootApplication` e que corre o servidor. De forma a que seja possível fazer chamadas à API, é usado o decorador `@RestController` na classe onde queremos ter os endpoints e, em cada função que contém o código do endpoint, adicionar o decorador `@RequestMapping` com o nome do caminho e o método.

```
@SpringBootApplication
public class SpringBootServer {
    public static void main(String[] args) {
        SpringApplicationBuilder builder = new SpringApplicationBuilder(SpringBootServer.class);
        builder.headless(false);
        ConfigurableApplicationContext context = builder.run(args);
    }
}
```

```
@RestController
public class Requests {
```

```
@RequestMapping(method = RequestMethod.POST, value="/adicionarCidade/{nome}/{latitude}/{longitude}")
public ArrayList<Cidade> adicionarCidade(@PathVariable("nome") String nome, @PathVariable("latitude")
{
```

## Chamadas API interna e externa

Existem cinco endpoints para pedidos internos:

- GET: “/start”
- GET “/devolverCidades”
- GET “/devolverCidade/{nome}”
- POST “/adicionarCidade/{nome}/...”
- GET “/pesquisarNome/{nome}”

O primeiro endpoint é uma chamada que providencia alguma QoL e foi construída com o intuito de expandir a fase de testes, aqui, é localizado o ficheiro index.html do front-end sendo que o mesmo é depois executado, assim, ao fazermos este pedido, é aberto uma nova página com o projeto.

O segundo endpoint retorna a lista de cidades preenchidas de forma hard-coded.

O terceiro endpoint retorna uma única cidade, este endpoint é usado na fase de testes.

O quarto endpoint permite acrescentar uma cidade à lista, esta funcionalidade é usada pelo front-end quando queremos adicionar uma nova cidade.

Até agora todas estas funcionalidades estão no escopo interno de API, ou seja, não buscam informação proveniente de datasources externos, no entanto, o último endpoint, para além de disponibilizar um caminho, faz um pedido externo a uma entidade, neste caso a Dark Sky. Assim, ao fazer o pedido, é passado um parâmetro de entrada (nome), que vai ser depois confrontado com a lista de cidades existente e é retirada a sua informação (latitude e longitude). Com esta informação, é feito um pedido à tal entidade que retorna um objecto JSON. Após processar a resposta, atribuindo-lhe o tipo de StringBuilder, é utilizada a biblioteca Gson de forma a transformá-la num objeto que possa depois ser transformado no front-end com JSON.parse. Existe uma limitação que não foi tratada, devido ao fato do nome não ser único (ou de não haver uma verificação para esse fim), é possível inserir nomes iguais, pelo que, a última cidade com o nome procurado sobrescreve a/s outras.

```
{
  "latitude": 40.64427,
  "longitude": -8.64554,
  "timezone": "Europe/Lisbon",
  "currently": {
    "time": 1557346165,
    "summary": "Ligeiramente Nublado",
    "icon": "partly-cloudy-night",
    "precipIntensity": 0.0152,
    "precipProbability": 0.04,
    "precipType": "rain",
    "temperature": 15.59,
    "apparentTemperature": 15.59,
    "dewPoint": 13.27,
    "humidity": 0.86,
    "pressure": 1015.65,
    "windSpeed": 1.75,
    "windGust": 4.4,
    "windBearing": 226,
    "cloudCover": 0.41,
    "uvIndex": 0,
    "visibility": 7.4,
    "ozone": 342.66
  },
  "hourly": {
    "summary": "Chuva começa esta manhã à tarde, continua até à manhã à noite.",
    "time": 1557346165
  }
}
```

```

@RequestMapping("/pesquisar/{nome}")
public Object pesquisar(@PathVariable String nome) throws IOException, JSONException
{
    Cidade cidadeAPesquisar = new Cidade();

    for (Cidade c : this.cidades)
    {
        if (c.getNome().equals(nome))
        {
            cidadeAPesquisar = c;
        }
    }

    URL url = new URL("https://api.darksky.net/forecast/6ecb9b96f185b1fb63ffdfd087f65f84/" +
        cidadeAPesquisar.getLatitude() + "," + cidadeAPesquisar.getLongitude() + "?lang=pt&units=si");

    HttpURLConnection con = (HttpURLConnection) url.openConnection();
    con.setRequestMethod("GET");
    con.setUseCaches(true);
    BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuilder response = new StringBuilder();
    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    String result = new Gson().toJson(response);
    return result;
}

```

## Cache

Cache guarda informação de um pedido de forma a que quando outro pedido seja feito nas mesmas condições, a informação guardada seja apresentada, sem que seja preciso fazer uma chamada externa.

No back-end, é implementado uma forma de cache providenciada pelo Spring Boot, seguindo [este](#) guia, foi possível implementar este sistema usando certas anotações nas chamadas API, no RESTController e no ficheiro de configuração da aplicação. Assim, no ficheiro de configuração, é usado um cache manager que guarda uma referência de uma string, para que, quando o pedido seja feito novamente, seja possível devolver o resultado mais rapidamente.

```

@EnableCaching
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(new ConcurrentMapCache("result")));
        return cacheManager;
    }
}

@Override

@CacheConfig(cacheManager="cacheManager")
@CrossOrigin
@RestController
public class Requests {

    @Cacheable("result")
    @RequestMapping("/pesquisar/{nome}")
    public Object pesquisar(@PathVariable String nome)
    {

        String result = new Gson().toJson(response);
        return result;
    }
}

```

## Estratégia de testes

Para realizar os testes, foi utilizada a versão 4 do JUnit devido a compatibilidade com as ferramentas utilizadas,

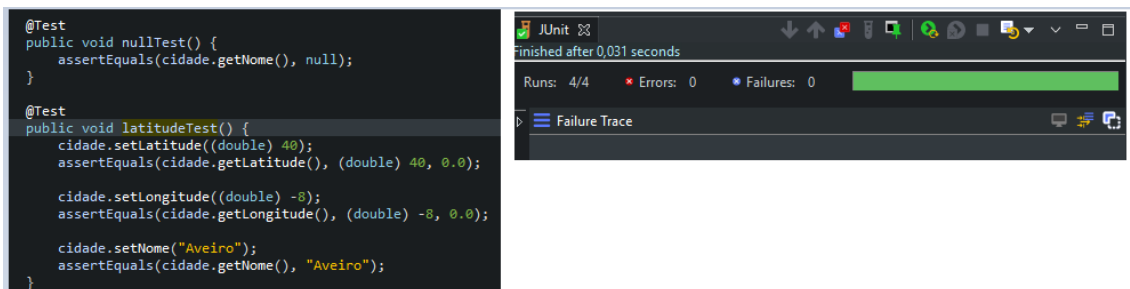
Nesta fase, foi possível testar se as funcionalidades implementadas anteriormente são válidas, pelo que, houve uma série de testes (treze) que operam sobre a aplicação de forma a testar a sua robustez.

### Testes unitários

Os testes unitários permitem separar e testar os módulos de cada função um por um, desta forma, é possível verificar se, de forma isolada, cada componente cumpre os seus objetivos.

Assim, foram realizados testes unitários à classe cidade, à lista da classe de requests e à implementação da cache na aplicação.

Em relação à cidade, foram testadas as operações de set e get, pelo que, ao fazer um set, o resultado obtido deverá ser igual ao enviado. Também foi feito um teste para verificar que quando se pretende retirar o nome da cidade, sem que esta seja setada, a operação retorna um valor nulo.



Em relação a requests, foi feito um teste de forma a validar o caminho do recurso `index.html` da parte front-end e testado se o ficheiro existe. Desta forma, podemos validar a sua existência, se é um ficheiro e se temos permissões para ler. Para além deste teste, foram ainda realizados outros onde se valida o tamanho da lista, o processo de adição de cidades e se o nome da cidade introduzida corresponde.



Em relação a cache, o processo foi retirado [deste](#) SO e passa por todos os momentos de cache implementados na API. Assim, inicialmente é criada uma interface que sirva como base para os testes e que guarde em cache um valor (no caso da aplicação seria o result). Depois é feita a configuração do cache manager que recebe o valor anterior. Então é feito um mock da

interface de forma a que seja possível testar sem definir valores, mas simulando o seu comportamento.

Após as configurações estarem concluídas, chega a altura de testar se a interface é, de fato, guardada em cache. Para tal, são criados dois objetos e, de forma a simular um resultado, é utilizado o Mockito de forma a prever os resultados pedidos, ou seja, no primeiro pedido é devolvido o primeiro objeto e, no segundo, o segundo objeto.

Assim, é feita uma série de asserts que validam a teoria de cache, ou seja, quando é feito o primeiro pedido, devolve o primeiro objeto. Ao fazer mais pedidos, o retorno é o primeiro objeto e é feita uma verificação de quantas vezes o método foi invocado, que se espera que seja uma única vez, ou seja, depois do primeiro pedido, todos os outros foram manipulados pela cache.

Ao fazer um pedido com um parâmetro diferente, devolve então o segundo objeto.

```
@Test
public void methodInvocationShouldBeCached() {

    Object first = new Object();
    Object second = new Object();

    // Set up the mock to return *different* objects for the first and second call
    Mockito.when(repo.findByEmail(Mockito.any(String.class))).thenReturn(first, second);

    // First invocation returns object returned by the method
    Object result = repo.findByEmail("foo");
    assertThat(result, is(first));

    // Second invocation should return cached value, *not* second (as set up above)
    result = repo.findByEmail("foo");
    assertThat(result, is(first));

    // Verify repository method was invoked once
    Mockito.verify(repo, Mockito.times(1)).findByEmail("foo");
    assertThat(manager.getCache("sample").get("foo"), is(notNullValue()));

    // Third invocation with different key is triggers the second invocation of the repo method
    result = repo.findByEmail("bar");
    assertThat(result, is(second));
}
```

## Testes de dependência

Os testes de dependência permitem verificar se uma classe simulada tem o mesmo comportamento que uma real, assim, é possível verificar as suas funcionalidades mesmo sem atribuir-lhe os parâmetros.

Para estes testes, foi usado o Mockito e JUnit. O Mockito é uma biblioteca que permite replicar o comportamento de um objeto e definir o que as suas funções devem retornar, assim, ao definirmos um output, mesmo que o parâmetro de entrada não corresponda a esse output, o resultado obtido será aquele definido anteriormente.



Assim, foram feitos mocks das classes cidade e requests, de forma a simular o seu comportamento. Desta forma verifica-se que ao adicionar um mock cidade à lista real, o seu tamanho aumenta.

```
@RunWith(MockitoJUnitRunner.class)
public class DependenciasTest {

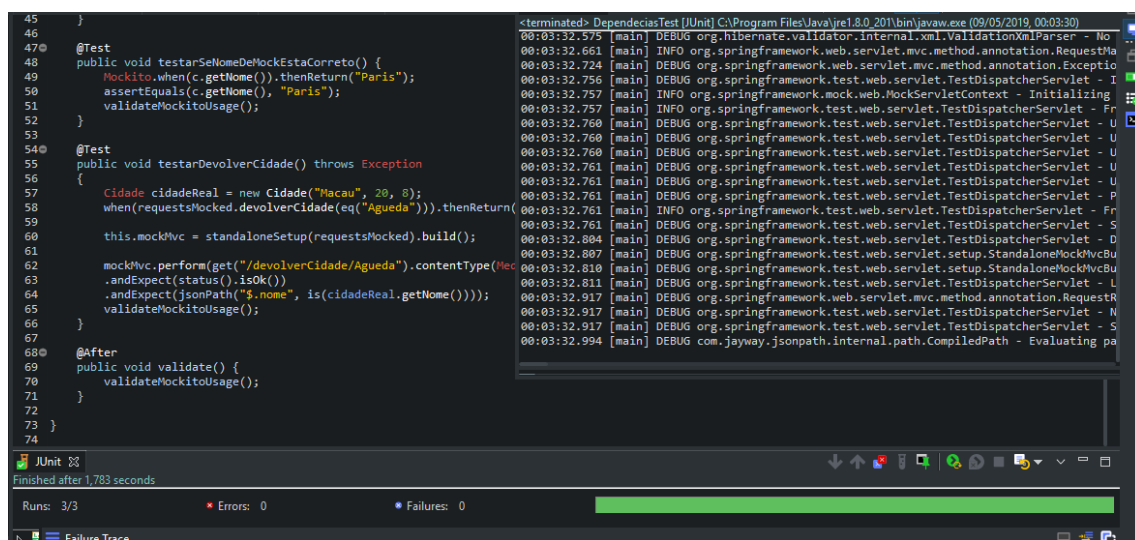
    Requests requests = new Requests();
    @Mock Requests requestsMocked = new Requests();
    @Autowired Requests requestsWired = new Requests();
    @Mock Cidade c;

    MockMvc mockMvc;

    @Test
    public void testarSeMockAdiciona() {
        int tamanho = this.requests.cidades.size();
        this.requests.cidades.add(c);
        assertTrue(this.requests.cidades.size() > tamanho);
        validateMockitoUsage();
    }
}
```

De forma a testar o comportamento do endpoint “/devolverCidade”, foi criado um objeto com o nome Macau, sendo que, ao fazer um pedido por o nome Águeda, devia retornar Águeda. Com o Mockito, é possível alterar o retorno esperado dessa função, assim, quando fazemos o pedido por Águeda, o nome devolvido é Macau.

De forma a simular uma chamada à API, foi usado o mockMvc e feito um pedido por Águeda. Devido à condição implementada anteriormente, a função retorna, mais uma vez, Macau.



The screenshot shows an IDE with a Java test class on the left and a console log on the right. The test class, `DependenciasTest`, includes a test method `testarSeNomeDeMockEstaCorreto()` and a test method `testarDevolverCidade() throws Exception`. The `testarDevolverCidade()` method creates a `Cidade` object named "Macau" and uses `MockMvc` to perform a GET request to `/devolverCidade/Agueda`. The console log shows the execution of the test, including the creation of the `Cidade` object and the successful execution of the `MockMvc` request, resulting in a 200 status code.

```
45 }
46
47 @Test
48 public void testarSeNomeDeMockEstaCorreto() {
49     Mockito.when(c.getNome()).thenReturn("Paris");
50     assertEquals(c.getNome(), "Paris");
51     validateMockitoUsage();
52 }
53
54 @Test
55 public void testarDevolverCidade() throws Exception
56 {
57     Cidade cidadeReal = new Cidade("Macau", 20, 8);
58     when(requestsMocked.devolverCidade(eq("Agueda"))).thenReturn(
59         cidadeReal);
60     this.mockMvc = standaloneSetup(requestsMocked).build();
61
62     mockMvc.perform(get("/devolverCidade/Agueda").contentType(MediaType.APPLICATION_JSON))
63         .andExpect(status().isOk())
64         .andExpect(jsonPath("$.nome", is(cidadeReal.getNome())));
65     validateMockitoUsage();
66 }
67
68 @After
69 public void validate() {
70     validateMockitoUsage();
71 }
72
73 }
74
```

Console Log:

```
<terminated> DependenciasTest [JUnit] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (09/05/2019, 00:03:30)
00:03:32.575 [main] DEBUG org.hibernate.validator.internal.xml.ValidationXmlParser - No
00:03:32.661 [main] INFO org.springframework.web.servlet.mvc.method.annotation.RequestMappingMethodProcessor - Mapping
00:03:32.724 [main] DEBUG org.springframework.web.servlet.mvc.method.annotation.RequestMappingMethodProcessor - Mapping
00:03:32.756 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - I
00:03:32.757 [main] INFO org.springframework.mock.web.MockServletContext - Initializing
00:03:32.757 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet - Fr
00:03:32.760 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - U
00:03:32.760 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - U
00:03:32.761 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - U
00:03:32.761 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - U
00:03:32.761 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - P
00:03:32.761 [main] INFO org.springframework.test.web.servlet.TestDispatcherServlet - Fr
00:03:32.761 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - S
00:03:32.804 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - D
00:03:32.807 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - D
00:03:32.810 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - D
00:03:32.811 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - L
00:03:32.917 [main] DEBUG org.springframework.web.servlet.mvc.method.annotation.RequestMappingMethodProcessor - Mapping
00:03:32.917 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - N
00:03:32.917 [main] DEBUG org.springframework.test.web.servlet.TestDispatcherServlet - S
00:03:32.994 [main] DEBUG com.jayway.jsonpath.internal.path.CompiledPath - Evaluating pa
```

JUnit 5  
Finished after 1,783 seconds  
Runs: 3/3 Errors: 0 Failures: 0

## Testes de integração

Os testes de integração permitem testar os endpoints da REST API e os seus resultados, assim, foi utilizado o REST-Assured devido à sua flexibilidade e o JUnit para fazer os asserts.

Os dois testes são feitos sobre o endpoint `"/devolverCidade/{nome}"` e testam o resultado dessa chamada.

No primeiro teste, é dado um certo registo de parâmetros, como o porto e o tipo de dados, e quando é feito o pedido GET, é extraída a resposta e o seu resultado é validado contra uma condição nula.

O segundo teste, é criado uma cidade e definido uma resposta base aos pedidos. Depois são feitos uma serie de testes que visam perceber se os valores não são nulos ou se o nome corresponde.

```
42 Requests requestsService;
43
44 @Test
45 public void testarDevolverCidadeResponse()
46 {
47     Response response = given().port(port).contentType("application/json").accept("application/json")
48     .when().get("/devolverCidade/Aveiro").then().statusCode(200).extract().response();
49     assertNotNull(response);
50 }
51
52 @Test
53 public void testarDevolverCidade() {
54     Cidade testCidade = new Cidade("California", 40, -112);
55     when(requestsService.devolverCidade("California")).thenReturn(testCidade);
56     get(uri + "/devolverCidade/" + testCidade.getNome()).then()
57     .assertThat()
58     .statusCode(HttpStatus.OK.value())
59     .body("nome", equalTo(testCidade.getNome()))
60     .body("latitude", notNullValue());
61
62     Cidade result = get(uri + "/devolverCidade/" + testCidade.getNome()).then()
63     .assertThat()
64     .statusCode(HttpStatus.OK.value())
65     .extract()
66     .as(Cidade.class);
67     assertEquals(result.getNome(), testCidade.getNome());
68
69     String responseString = get(uri + "/devolverCidade/" + testCidade.getNome()).then()
70     .assertThat()
71     .statusCode(HttpStatus.OK.value())
72     .extract()
73     .asString();
74 }
```

<terminated> IntegrationTest [JUnit] C:\Program Files\Java\jre1.8.0\_201\bin\java.exe (09/05/2019 00:53:07.599 INFO 11684 --- [main] ationConfi  
2019-05-09 00:53:10.246 INFO 11684 --- [main] s.b.c.e.t.  
2019-05-09 00:53:10.268 INFO 11684 --- [main] o.apache.c  
2019-05-09 00:53:10.271 INFO 11684 --- [main] org.apache  
2019-05-09 00:53:10.477 INFO 11684 --- [ost-startStop-1] o.a.c.c.C.  
2019-05-09 00:53:10.478 INFO 11684 --- [ost-startStop-1] o.s.web.co  
2019-05-09 00:53:10.662 INFO 11684 --- [ost-startStop-1] o.s.b.w.se  
2019-05-09 00:53:10.668 INFO 11684 --- [ost-startStop-1] o.s.b.w.se  
2019-05-09 00:53:11.842 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.844 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.849 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.851 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.852 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.853 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.856 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:11.978 INFO 11684 --- [main] s.w.s.m.m.  
2019-05-09 00:53:12.603 INFO 11684 --- [main] s.b.c.e.t.  
2019-05-09 00:53:12.611 INFO 11684 --- [main] tq.s.api.In  
2019-05-09 00:53:13.814 INFO 11684 --- [o-auto-1-exec-1] o.a.c.c.C.  
2019-05-09 00:53:13.815 INFO 11684 --- [o-auto-1-exec-1] o.s.web.se  
2019-05-09 00:53:13.848 INFO 11684 --- [o-auto-1-exec-1] o.s.web.se


JUnit  
finished after 8.52 seconds  
Runs: 2/2  
Errors: 0  
Failures: 0  
Failure Trace

## Testes funcionais

O teste funcional permite verificar o workflow do site.

Não foi possível correr este teste, embora o seu conteúdo pareça estar correto, visto que foi usado o Katalon Recorder para gravar os passos, existe uma incompatibilidade que não foi possível resolver.

```
17
18 @Test
19 public void testUntitledTestCase() throws Exception {
20     WebDriver driver = new ChromeDriver();
21
22     driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
23     String url = "file:///C:/Users/Asus/Desktop/tqs/tqs.api/target/classes/HTML/index.html";
24     driver.get(url);
25     assertEquals("Aveiro", findElement(driver, By.id("locationToday"), 1000));
26     new Select (driver.findElement(By.id("list"))).selectByVisibleText("Lisbon");
27     assertEquals("Lisbon", findElement(driver, By.id("locationToday"), 1000));
28     driver.findElement(By.id("modal")).click();
29     driver.findElement(By.id("cidadeInput")).sendKeys("electricmoon");
30     driver.findElement(By.id("latitude")).sendKeys("40");
31     driver.findElement(By.id("longitude")).sendKeys("-112");
32     driver.findElement(By.id("adicionarCidade")).click();
33     Thread.sleep(1000);
34     new Select (driver.findElement(By.id("list"))).selectByVisibleText("California");
35     assertEquals("California", findElement(driver, By.id("locationToday"), 1000));
36     driver.quit();
37 }
38
39 public static WebElement findElement(WebDriver driver, By selector, long timeOutInSeconds) {
40     WebDriverWait wait = new WebDriverWait(driver, timeOutInSeconds);
41     wait.until(ExpectedConditions.visibilityOfElementLocated(selector));
42 }
```

JUnit  finished after 0,188 seconds

Runs: 1/1    ✖ Errors: 1    • Failures: 0

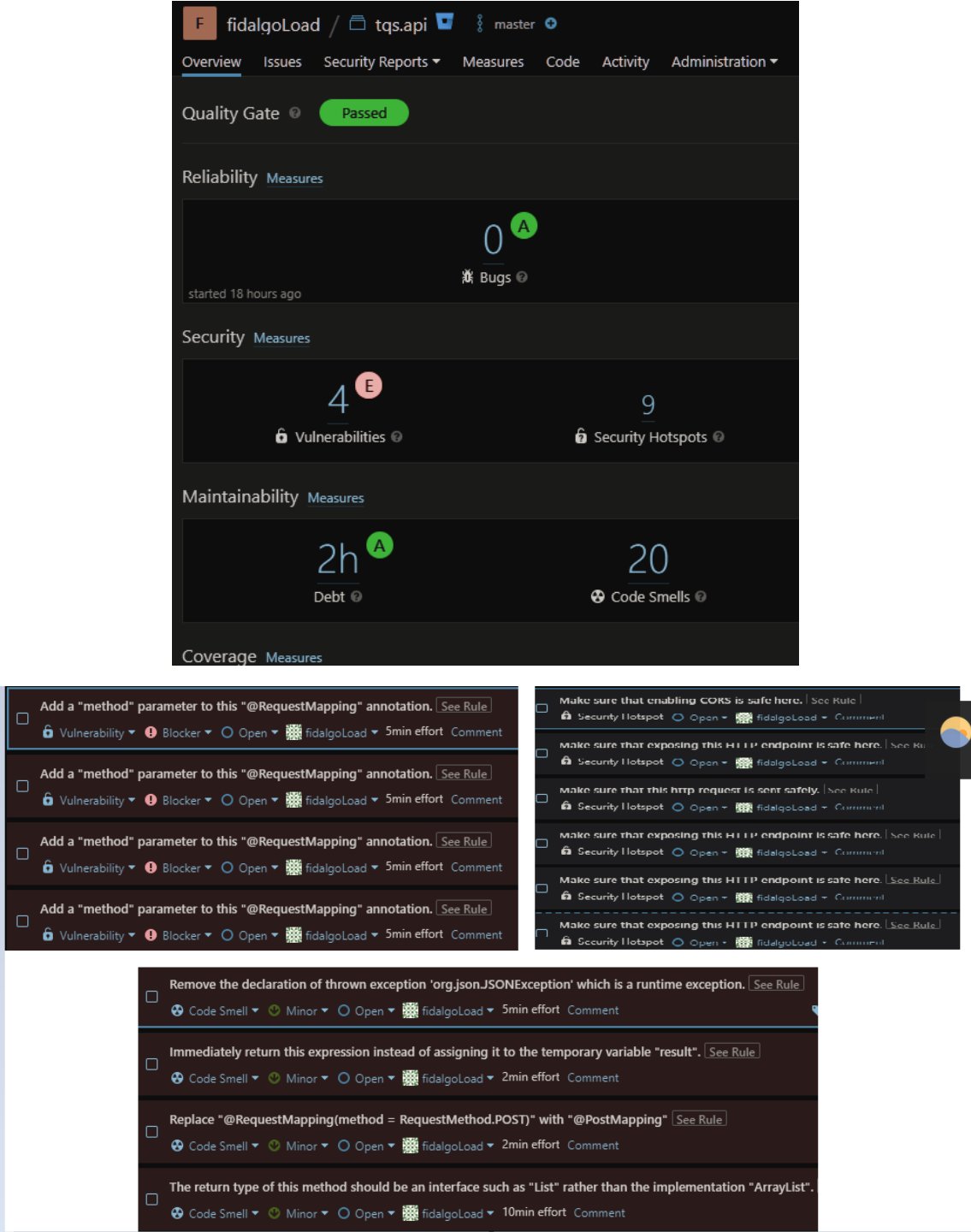
**Failure Trace**

- java.lang.NoClassDefFoundError: org/openqa/selenium/MutableCapabilities
- at java.lang.ClassLoader.defineClass1(Native Method)
- at java.lang.ClassLoader.defineClass(Unknown Source)
- at java.security.SecureClassLoader.defineClass(Unknown Source)

# Resultados do SonarQube

O SonarQube é uma ferramenta que permite rever o código produzido.

Após a análise, obtém-se o seguinte resultado.



Em relação às vulnerabilidades, são detetadas quatro ocorrências, estas acontecem devido ao facto de os pedidos da API não estarem tipificados com o método GET, pelo que, podia ser resolvido ao inserir esse atributo.

Em relação ao security hotspot, são uma espécie de avisos que não são vistos pela aplicação como vulnerabilidades, mas que o podem ser caso o desenvolvedor não tenha sido cuidadoso. Por exemplo, na aplicação do CORS.

Os code smells fazem uma análise do código e encontram incongruências, por exemplo, em vez de usar `@RequestMapping(method=RequestMethod.POST)`, é recomendado a utilização de `@PostMapping`.