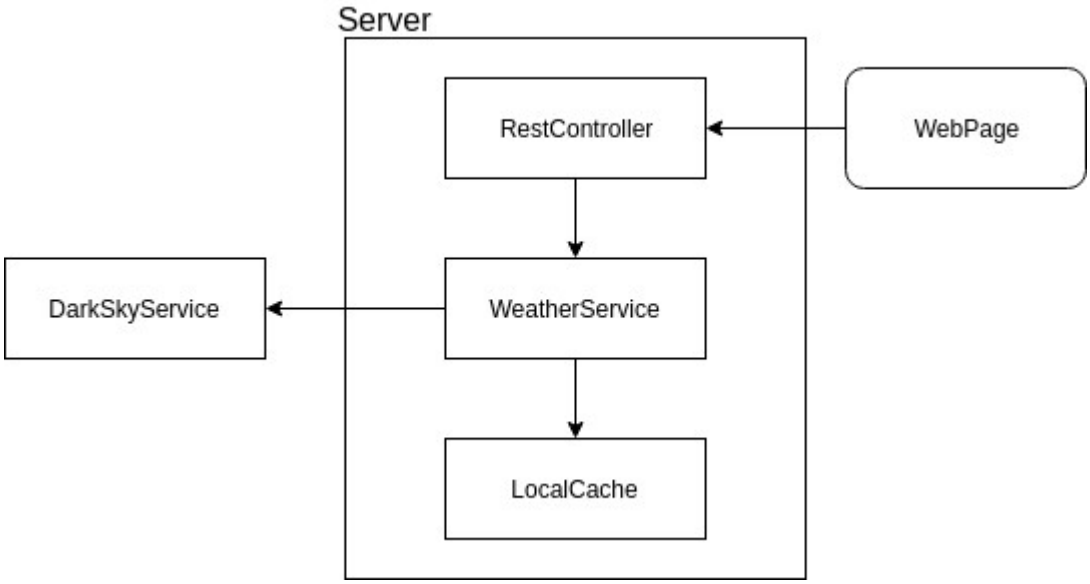
Homework 1:

Desenvolvimento de testes

Preparado por: João Pedro Simões Alegria
NMEC: 85048
Data: 3/5/2019
Git do Projeto: https://github.com/joao-alegria/TQS-HW1
SonarQube: https://sonarcloud.io/dashboard?id=joao-alegria_TQS-HW1

1- Estratégia adoptada

Arquitetura



O sistema implementado segue as diretivas definidas no enunciado do trabalho de casa e desse modo, tem como componente principal um servidor que internamente se decompõe em 3 sub-componentes, um controlador REST responsável por responder a todos os pedidos HTTP que sejam feitos ao servidor, um serviço com o nome de WeatherService que é composto por toda a lógica interna necessária para responder aos diferentes pedidos. Este componente é também responsável por comunicar com um terceiro componente, uma cache local usada para guardar a informação de retorno dos últimos pedidos à aplicação, durante um intervalo de tempo definido, que consequentemente permite diminuir os tempos de resposta do serviço. Este terceiro componente foi implementado de raiz, disponibilizando uma interface similar a objetos criados por terceiros para o mesmo efeito, tais como JCache.

Para fornecer a informação nos diferentes pontos REST e como pré-requisito do projeto, internamente é feita a comunicação com uma api externa, no caso foi escolhido o Dark Sky API como fonte de dados para esta aplicação.

Por fim, foi também implementada uma página web que age como cliente da api desenvolvida, muito simples e minimalista, baseada em Bootstrap, que fornece a opção de inserir uma localidade (em coordenadas), ou selecioná-la a partir de um mapa, sendo depois a previsão meteriológica do local selecionado apresentado consoante a extensão préviamente escolhida (atual, toda a semana, ou limitada por número de dias). As seguintes imagens ilustram as funcionalidades principais dessa página:

TQS Homework1 - Weather Prediction

Select the location to predict and the time duration

- (•) Current Weather
- ( ) Multiple Day Prediction

Insert in form | Select from map

Latitude:

40

Longitude:

-8

Submit

TQS Homework1 - Weather Prediction

Select the location to predict and the time duration

- ( ) Current Weather
- (•) Multiple Day Prediction

Number of days to predict(8 max.)

Insert in form | Select from map

Predictions for the entire week for latitude=40.18 and longitude=-6.56

Day	Humidity	Ozone	Percip. Prob.	Percip. Type	Pressure	Visibility	Wind Speed
5/5/2019	0.57	348.07	0	undefined	1017.99	10	3.17
6/5/2019	0.73	347.66	0.23	rain	1017.83	8.84	4.43
7/5/2019	0.74	340.45	0.7	rain	1014.73	6.3	10.05
8/5/2019	0.76	331.26	0.48	rain	1016.64	8.86	4.19
9/5/2019	0.73	321.2	0.19	rain	1016.78	6.98	3.7
10/5/2019	0.57	323.92	0.01	rain	1019.69	10	1.49
11/5/2019	0.45	304.96	0	undefined	1019.24	10	3.37
12/5/2019	0.41	291.99	0	undefined	1018.81	10	3.33

API

A REST API desenvolvida é muito simples e permite a um utilizador saber qual a previsão meteriológica atual, dos próximos 8 dias ou de um número de dias menor que este.

A próxima tabela apresenta a api desenvolvida, os parâmetros necessários para cada pedido e os respetivos valores de retorno.

Method	Endpoint	Parameters	Output
GET	/api/weather/[lat],[long]	lat-latitude do local long-longitude do local	Retorna um JsonArray com as previsões de todos os dias dos próximos 7 dias.
GET	/api/weather/[lat],[long]/now	lat-latitude do local long-longitude do local	Retorna um JsonObject com a previsão atual meteriológica.
GET	/api/weather/[lat],[long]/[num]	lat-latitude do local long-longitude do local num-número de dias a limitar	Retorna um JsonArray com as previsões dos próximos número de dias, consoante o inserido.

Testes

Sendo um projeto da unidade curricular Teste e Qualidade de Software, os testes e mecanismos de controlo de qualidade são dos aspetos mais importantes deste trabalho, tendo por isso que ser dedicado algum tempo aos mesmos.

Em termos de testes foi usada uma estratégia similar a TDD (Test Driven Development), sendo desenvolvidos vários testes unitários, testes de integração relativos à api e testes funcionais sobre a página web implementada. Visto que a aplicação foi implementada com a ajuda da framework SpringBoot, os testes implementados usam o ambiente de testes já existente na framework, utilizando SpringBootTest para testes funcionais sobre a página web e WebMvcTest para testes de integração da api.

Para testar o cache local, o serviço interno e o serviço externo foram usados testes unitários com a ajuda de Junit5 e Mockito para fazer mocks nos casos de existirem dependências com outros componentes, fazendo com que os testes sejam o mais isolados possível, sendo mais fácil encontrar possíveis erros.

Os testes implementados foram os seguintes, divididos por objetos a testar:

- REST API(com mock do WeatherService):

* Testar o path *api/*weather/[lat],[long] e verificar se a informação passada é a desejada e se tem todos os valores necessário

* Testar o path *api/*weather/[lat],[long]/now e verificar se a informação passada é a desejada e se tem a previsão atual.

* Testar o path *api/*weather/[lat],[long]/[num] e verificar se a informação passada é a desejada e se o número de previsões é igual ao numero indicado no caminho.

* Testar o path *api/cacheStatus* e verificar se a informação passada é a desejada e se todas as métricas são enviadas.

- LocalCache:

* Testar um acesso à cache com uma chave não existente.

* Testar uma chamada ao método getLimited em que o valor não é iterável, sendo suposto retornar o valor como inserido.

* Testar uma chamada ao método getLimited em que o valor é iterável, sendo suposto retornar apenas o número de sub-elementos do valor indicado.

* Testar uma chamada ao método get, sendo suposto retornar o valor como inserido.

* Testar uma chamada ao método put, verificando se o par chave-valor foi realmente inserido.

* Testar uma chamada ao método clear, verificando se o par chave-valor foi realmente removido.

* Testar uma chamada ao método containsKey, verificando se o valor retornado coincidia com a existência da chave a testar.

* Testar uma chamada ao método getMetrics, verificando se todas as métricas eram enviadas.

* Testar se depois de inserido um valor e esperado o time to leave, o valor de facto já não se encontrava na cache.

* Testar com chamadas a chaves existentes ou não, se as métricas retornadas eram as corretas.

- WeatherService(mock do serviço externo):

* Teste ao método getWeather, verificando se o valor retornado fazia o processamento devido.

* Teste ao método getWeatherLimited, verificando se o valor retornado fazia o processamento devido, limitando o número de elementos da lista.

* Teste ao método getCurrentWeather, verificando se o valor retornado fazia o processamento devido e retornava apenas o objeto representativo da previsão atual.

* Teste ao método getCacheMetrics, verificando se todas as méricas da cache interna eram retornados.

* Teste a se as métricas fornecidas coincidem caso sejam feitos vários gets.

* Teste a se as métricas fornecidas coincidem caso sejam feitos vários gets e caso o time to leave de alguns métodos expire.

* Teste a caso o número de limitação de previsões seja maior que 8.

* Teste a caso o número de limitação de previsões seja menor que 0.

- DarkSkyService:

* Teste a se a resposta obtida da fonte externa é a necessário e se os valores

retornados são os necessários e se o processamento é realmente feito:

- Página Web:
    * Verificação funcional caso se peça a previsão atual atravez do formulário.
    * Verificação funcional caso se peça a previsão dos próximos 3 dias atravez da seleção no mapa.
    * Verificação funcional caso se peça a previsão de toda a semana(não inserindo nenhum valor no campo de limitação), atravez da seleção no mapa.

Exemplo de teste unitário sobre a cache local:

```
@Test
   public void testGet() throws Exception {
      System.out.println("get");
      Object key = "key";
      Object data = "value";
                              MyCache instance =new MyCache.Builder<String, String>().ttl(0.5*60*1000).updateTime(0.5*60*1000).build();
      assertEquals(false, instance.containsKey(key));
      instance.put(key, data);
      assertEquals(true, instance.containsKey(key));
      assertEquals(data, instance.get(key));
   }
```

Exemplo de configuração de um mock para os testes do WeatherService, isolando-o do serviço externo:

```
@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.LENIENT)
public class WeatherServiceTest {


   @InjectMocks
   private WeatherService weatherService;

   @Mock
   private DarkSkyService darkService;

   @BeforeEach
   public void setUp() {
      expList =  new ArrayList();
      ...
      Mockito.when(darkService.getPredictions(0.0, 0.0)).thenReturn(expList);
   }
```

Exemplo de um teste à api criada, usando o ambiente de testes disponibilizado pelo framwork (usando também mock para isolar do serviço):

```
@ExtendWith(MockitoExtension.class)
@WebMvcTest(value=REST.class)
public class RESTTest {

   @Autowired
   private MockMvc mvc;

   @MockBean
   private WeatherService weatherService;

   @BeforeEach
   public void setUp() {
      ...
      Mockito.when(weatherService.getCacheMetrics()).thenReturn(obj);
   }


   @Test
   public void testWeather() throws Exception {
      System.out.println("weather");
      Double latitude = 0.0;
      Double longitude = 0.0;
      mvc.perform(get("/api/weather/"+latitude.toString()+","+longitude.toString())
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", hasSize(3)))
            .andExpect(jsonPath("$[0]", hasKey("key1")))
            .andExpect(jsonPath("$[0]", hasKey("key2")))
            .andExpect(jsonPath("$[0]", hasKey("key3")));
   }
```

Exemplo de um teste funcional feito sobre a página web com a ajuda do ambiente de testes fornecido pela plataforma:

```
@ExtendWith(SeleniumExtension.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
public class WebTestIT {

   @Test
   public void testT1() throws Exception {
      driver.get("http://localhost:8080/");
      driver.findElement(By.id("latitude")).click();
```

```
        driver.findElement(By.id("latitude")).clear();
        driver.findElement(By.id("latitude")).sendKeys("40");
        driver.findElement(By.id("longitude")).click();
        driver.findElement(By.id("longitude")).clear();
        driver.findElement(By.id("longitude")).sendKeys("-8");
             driver.findElement(By.xpath("(.//*[normalize-space(text()) and normalize-space(.)='Longitude:'])[1]/following::input[2]")).click();
            assertEquals("Current Weather for latitude=40.00 and longitude=-8.00", driver.findElement(By.id("infoTitle")).getText());
             assertTrue(driver.findElement(By.xpath("(.//*[normalize-space(text()) and normalize-space(.)='Wind Speed'])[1]/following::td[1]")).isDisplayed());
    }
```

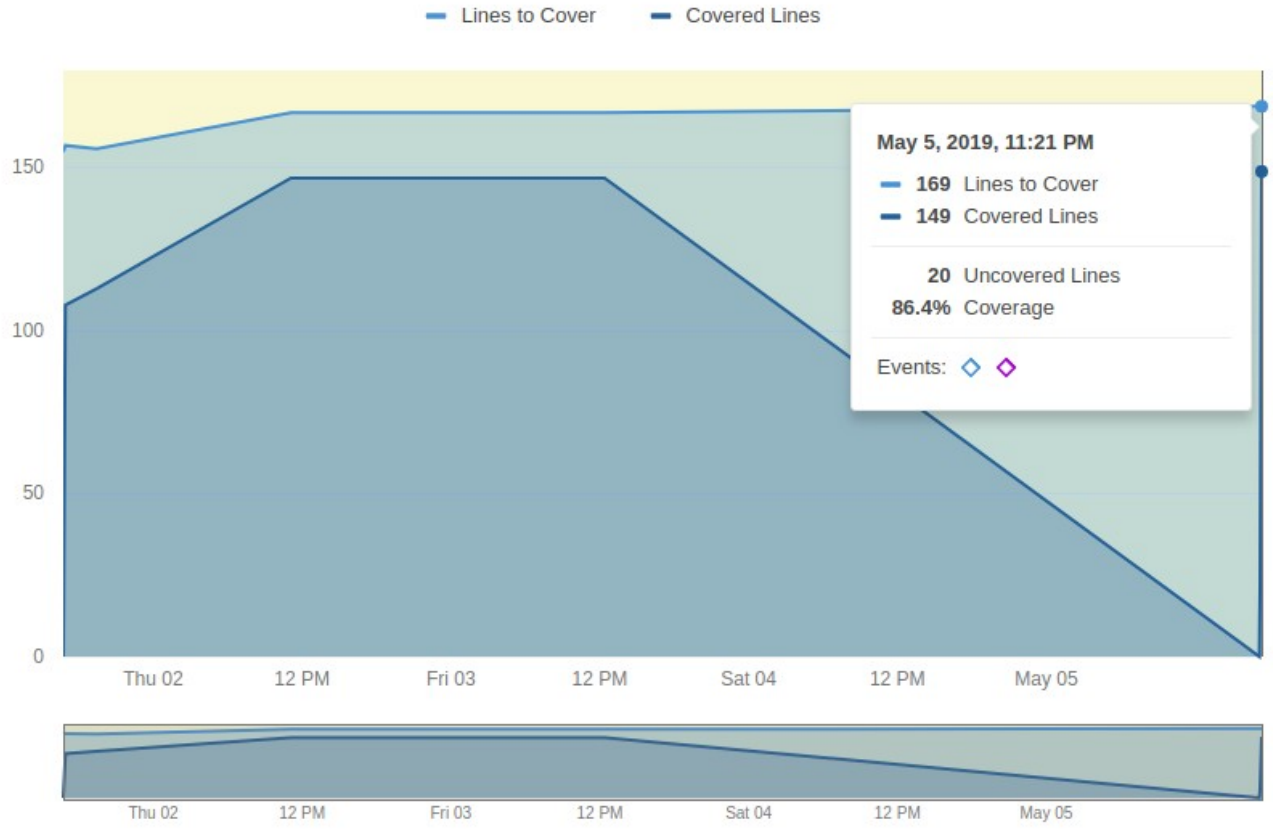
2- SonarQube

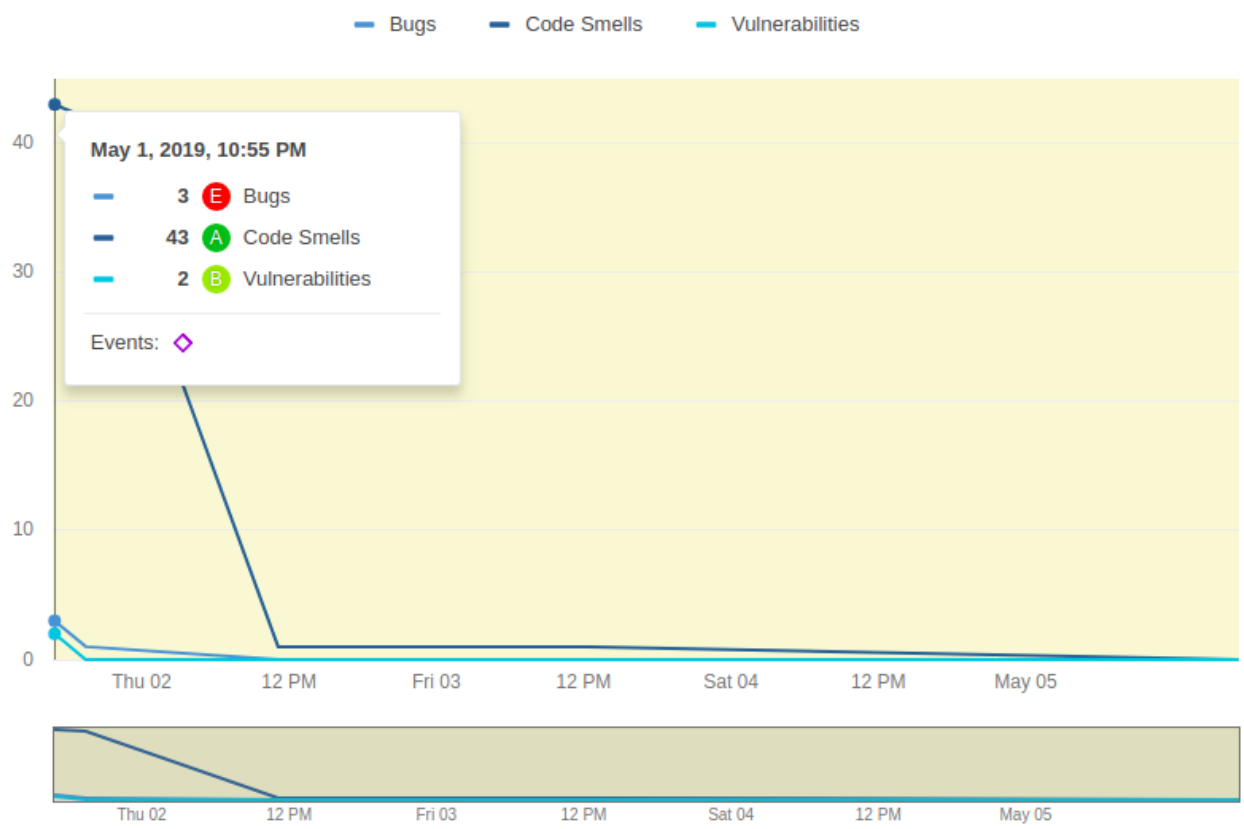
SonarQube, como já foi dito, foi a ferramenta usada para realizar a análise estática de código, servindo como mecanismo de garantia de qualidade. Visto que o projeto é público foi usado o SonarCloud pois é mais conveniente e rápido sem ter nenhuma desvantagem em relação ao SonarQube.

Para este projeto foi definida uma quality gate adequada e com métricas consideradas fundamentais, quality gate ilustarado na imagem seguinte:

Metric	Operator	Error
Bugs	is greater than	3
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A
Unit Test Success (%)	is less than	100%

O SonarCloud foi usado ao longo do desenvolvimento e todos os erros indicados pela plataforma foram corrigidos, focando principalmente nos bugs e vulnerabilidades. A imagem seguinte mostra a evolução das várias análises feitas ao código, tanto relativamente a erros, tal como cobertura dos testes sobre o código.





A existência de erros é natural, apesar de que em pouca quantidade devido ao tamanho do projeto. Assim, foram encontrados alguns erros, sendo que no início existiam 3 bugs, o que o SonarCloud considera perigoso, sendo estes erros corrigidos de imediato. Existiam também 2 vulnerabilidades e 43 code smells, o que o SonarCloud considerava dentro do normal como se pode ver pela avaliação de "A" (a melhor na plataforma). Mesmo assim, estes erros foram corrigidos para que o código fosse o melhor possível. Depois de todos os erros corrigido e todas as duplicações de código eliminados, numa tentativa de diminuir a dívida técnica inicial de quase 6 horas, o projeto encontra-se a passar o quality gate definido, com uma boa aprovação do SonarCloud e a passar todos os testes implementados como ilustra as seguintes imagens.

```
Results:

Tests run: 21, Failures: 0, Errors: 0, Skipped: 0


--- maven-jar-plugin:3.1.1:jar (default-jar) @ Hw1Application ---
Building jar: /home/joaoalegria/Desktop/TQS/Pratica/TQS-HW1/HW1/target/Hw1Application-0.0.1-SNAPSHOT.jar

--- spring-boot-maven-plugin:2.1.4.RELEASE:repackage (repackage) @ Hw1Application ---
Replacing main artifact with repackaged archive

--- maven-install-plugin:2.5.2:install (default-install) @ Hw1Application ---
Installing /home/joaoalegria/Desktop/TQS/Pratica/TQS-HW1/HW1/target/Hw1Application-0.0.1-SNAPSHOT.jar to /home/joaoalegria/.
Installing /home/joaoalegria/Desktop/TQS/Pratica/TQS-HW1/HW1/pom.xml to /home/joaoalegria/.m2/repository/tqs/hw1/Hw1Applicat
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 43.626 s
Finished at: 2019-05-06T00:42:16+01:00
Final Memory: 43M/322M
------------------------------------------------------------------------
```

Quality Gate Passed

Reliability Measures

New code: last 30 days

0 A	0 A
Bugs	New Bugs

started 4 days ago

Security Measures

0 A	6	0 A	0
Vulnerabilities	Security Hotspots	New Vulnerabilities	New Security Hotspots

Maintainability Measures

0 A	0	0 A	0
Debt	Code Smells	New Debt	New Code Smells

Coverage Measures

86.4%	21	80.6%
Coverage	Unit Tests	Coverage on 82 New Lines to Cover