

TQS: Quality Assurance manual

André Alves [89334], Miguel Mota [89331], Luís Fonseca [89066], João Silva [88813]

v01

1	Project management	1
1.1	Team and roles	1
1.2	Agile backlog management and work assignment	1
2	Code quality management	2
2.1	Guidelines for contributors (coding style)	2
2.2	Code quality metrics	2
3	Continuous delivery pipeline (CI/CD)	2
3.1	Development workflow	2
3.2	CI/CD pipeline and tools	2
3.3	Artifacts repository	2
4	Software testing	2
1.1	Overall strategy for testing	2
1.	Functional testing/acceptance	2
2.	Unit tests	3
3.	System and integration testing	3
4.	Performance testing [Optional]	3

1 Project management

1.1 Team and roles

- Team Manager - Miguel Mota
- Product Owner - André Baião
- DevOps Master - Luís Fonseca e João Silva
- Quality Engineer - Miguel Mota
- Developers - Todos

1.2 Agile backlog management and work assignment

Jira é a plataforma usada para backlog onde se encontram todas as tarefas a ser realizadas e o respetivo estado das mesmas - se estão em progresso ou finalizadas. Deste modo, realizaram-se vários *sprints* para realizar X tarefas em Y tempo (aproximadamente 1 semana entre sprints, dependendo do work-load).

As tarefas vão estar divididas maioritariamente em Frontend e Backend. Sendo que o André e o João estarão focados principalmente no Frontend enquanto o Miguel e Luís estarão a trabalhar mais no Backend.

Em termos de desenvolvimento Agile, seguiremos um processo próximo de scrum, não tendo reuniões todos os dias mas aproximando-nos deste método. Ao fim de cada sprint reunimo-nos para visualizar o que cada um desenvolveu e definir as próximas etapas.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Ao longo do desenvolvimento do Backend, seguiram-se as normas mais comuns de Java development. Alguns dos exemplos do coding-style passam por:

- CamelCase;
- Não ignorar exceções, devolvendo exceções para o cliente descritivas o suficiente para resolver o problema, mas sem dar informação sobre o funcionamento interno do sistema;
- Não capturar exceções genéricas, que vai ao acordo do ponto anterior, no qual este facilita o tratamento das exceções de forma mais personalizada;
- Evitar fazer imports genericos;
- Escrever os nomes dos métodos de forma a que estes sejam auto-explicativos, descrevendo de forma natural o funcionamento do código que este executa;
- Se um método receber demasiados atributos dividir de forma a que estes se leiam um a um na vertical (limite do comprimento das linhas);
- Usar o modo de chavetas mais comum;
- Deixar um issue juntamente com `//TODO` caso seja preciso alguma alteração conhecida.

2.2 Code quality metrics

Utilização Sonar Qube para análise do código quer localmente quer através do workflow ativo no GitHub. Ou seja, quando se finaliza uma feature, faz-se um refactor ao código através da ajuda do Sonar, com o intuito de melhorar a qualidade do código antes de ser feito o merge.

Em termos de quality gates a serem efetuadas, usaram-se as seguintes:

- Coverage superior ou igual a a 85%;

- Linhas duplicadas inferior a 5%;
- Maintainability igual a A;
- Reliability igual a A;
- Security Rating igual a A;

Optamos por estas quality gates pois são aquelas que mais se aproximam do que consideramos um bom projecto e que vão ao encontro do que temos vindo a aprender ao longo da cadeira de TQS.

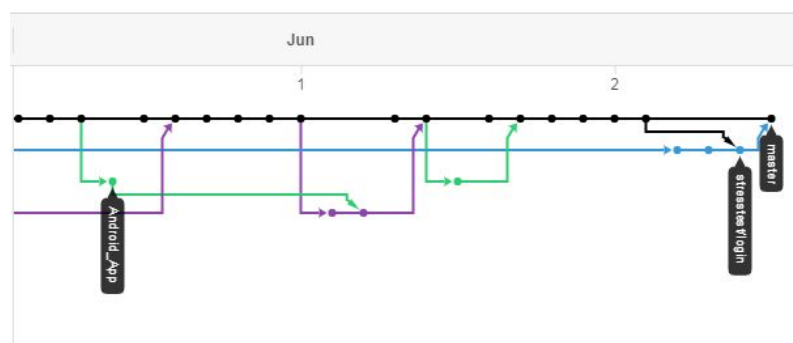
3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Para este projecto trabalhamos segundo o **github workflow**:

- Criamos branches para cada feature, inicialmente estas servem para criar o core do projecto, posteriormente cada feature vai corresponder a uma user story e/ou use case. Por norma as branches são criadas a partir da master, mas em alguns momentos havendo necessidade de código já testado e criado noutra branch que seria necessário nesta nova branch, fazia-se branch a partir desta. Da mesma maneira, priorizava-se o merge destas branches, associadas normalmente à mesma user story, antes de posteriormente o realizar na master;
- Após termos a branch criada desenvolve-se a feature associada; através de diversos commits, representativos do progresso contínuo de desenvolvimento;
- Assim que a feature estiver completa e testada (local e remotamente - graças ao CI) efetua-se um pull request;
- O Pull-Request, é alvo de Code Review, sendo avaliado e revisto por outro developer a trabalhar no mesmo contexto, validando ou não o que o parceiro fez. Caso detete alguma irregularidade quer seja com o código, falta de testes ou não cumprimento dos padrões do coding style definido deixa um comentário de forma a que as alterações necessárias sejam efetuadas;
- Se o workflow de validação dos testes falhar este é fechado;
- Caso o Pull-Request esteja válido, faz-se o merge das duas branches.

Em baixo podemos ver parte do workflow efetuado no projecto. Demonstrando o uso de branches, merges e pull requests.



Dá-se por concluída uma user-story quando os testes para a mesma estiverem todos feitos e a passar, para além de ter o Frontend atualizado com o Backend relativamente aos novos incrementos.

Os testes de Selenium nem sempre foram considerados para a definition of “done”, devido não só ao *crunch* temporal, mas também à elevada volatilidade do Frontend durante o desenvolvimento, sendo que por vezes para avançar noutras outras core features do sprint foram deixados para alturas posteriores.

Não obstante, os testes de Selenium associados ao vários use cases foram criados e postos a prova - e integrados obviamente na pipeline de CI.

































<input type="checkbox"/>	 Feature/ask rent 	#30 by andrebaiao was merged 13 days ago	 2
<input type="checkbox"/>	 user can now search property properly 	#28 by MotaMiguel was merged 15 days ago	 4
<input type="checkbox"/>	 User can now check reviews, get ratings and create reviews 	#27 by MotaMiguel was merged 15 days ago	 1
<input type="checkbox"/>	 Create new user 	#26 by andrebaiao was merged 15 days ago	 1
<input type="checkbox"/>	 Testing/create new user 	#25 by luis99fonseca was merged 15 days ago	 3
<input type="checkbox"/>	 Feature/search property 	#21 by andrebaiao was merged 16 days ago	 2

Imagem demonstrativa do processo de workflow para os pull-request remetendo para os uses cases. Caso o pull-request tivesse algo que não estivesse de acordo com os uses cases, coding-style ou quality-gates (como foi mencionado anteriormente) era efetuado um comentário a pedir para alterar o que se encontrava errado antes de se proceder para a aceitação do pull-request.

Quando detectados erros quer no que diz respeito a bugs ou funcionalidades mal implementadas ou até “incompatibilidades” - por exemplo, se ao nível do Frontend fosse preferível X enquanto que no Backend se desenvolveu de forma Y - estes problemas eram comunicados tanto no JIRA como em Github Issues. Os issues mais complexos eram constituídos por comentários explicitando em concreto o que estava contra os princípios e de que forma poderia ser alterado. Além disso os issues, por norma, ao estarem associados com a implementação feita por um elemento, eram atribuídos a esse mesmo elemento.

<input type="checkbox"/>	 0 Open  16 Closed	Author ▾	Label ▾	Projects ▾	Milestones ▾	Assignee ▾	Sort ▾
<input type="checkbox"/>	 Last user experiences	#48 by andrebaiao was closed 4 days ago					
<input type="checkbox"/>	 login with username instead of user id	#47 by MotaMiguel was closed 4 days ago					
<input type="checkbox"/>	 comodities	#44 by andrebaiao was closed 8 days ago					
<input type="checkbox"/>	 Owner shouldn't be able to rent his own house	#38 by MotaMiguel was closed 4 days ago					
<input type="checkbox"/>	 Rent a house	#34 by andrebaiao was closed 12 days ago					 1
<input type="checkbox"/>	 Profile API	#33 by andrebaiao was closed 8 days ago					
<input type="checkbox"/>	 Review should only display after it has passed the date	#32 by MotaMiguel was closed 4 days ago					
<input type="checkbox"/>	 Rents requests	#29 by andrebaiao was closed 8 days ago					
<input type="checkbox"/>	 Get specific house by id	#24 by andrebaiao was closed 16 days ago					

Rent a house #34

 Closed andrebaiao opened this issue 13 days ago · 1 comment



andrebaiao commented 13 days ago

Collaborator 🗨️ ⋮

If a user id=1 ask to rent a house by user id=2, when i gonna check the rent requests , id=2 didnt have no one request, however id=1 have their request :C




MotaMiguel commented 12 days ago

Owner 🗨️ ⋮

Fixed user that rented was being sent instead of owner



 MotaMiguel closed this 12 days ago

3.2 CI/CD pipeline and tools

Para este projecto trabalhamos com o **github** para produzir **continuous integration**. Através do **continuous integration** e do uso das pipelines do **github actions**, sempre que fosse efetuado um **push** ou **pull-request** na branch master eram efetuados todos os testes para validar que o código passava todos os testes existentes e as **quality gates** definidas no **Sonarcloud**. Caso estas condições passassem o **continuous delivery** e **deployment** era ativado, em caso de push era direto, em caso de **pull-request** o membro da equipa responsável por ver o **pull-request** tinha que validar que passavam, com ajuda das ferramentas fornecidas pelo github, antes de aceitar o **pull-request**, passando depois, caso tudo corra bem, para o deployment.

Para correr os testes deste projeto **Maven**, temos um workflow a correr no github que através de algumas configurações adicionais, como por exemplo uma **secret key** do Sonarcloud, permite correr os testes e verificar o coverage e quality gate estabelecidos e mencionados em cima e, caso estes não sejam cumpridos o teste falha e consequentemente a pipeline também.

```
on: [push,pull_request]
name: Tests w/ sonarcloud
jobs:
  sonarCloudTrigger:
    name: SonarCloud Trigger
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up JDK
        uses: actions/setup-java@v1
        with:
          java-version: 1.11
      - name: Maven Test and SonarCloud
        run:
          mvn -f JustLikeHomeStarter verify -Dtest=!WebPlatformTesting sonar:sonar
          -Dsonar.qualitygate.wait=true -Dsonar.projectKey=MotaMiguel_TQS_2 -Dsonar.organization=tqs-project
          -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=$SONAR_TOKEN
    env:
      SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
      GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

Antes de se iniciar o deployment correm-se os testes unitários, de integração e funcionais, mas desta vez sem uso do SonarCloud (deixando essa propriedade para o workflow mencionado anteriormente), se todos os testes passarem continuamos para o *continuous deployment*, caso haja algo erro o workflow não continua.

Desta forma, quando os testes passam e seguimos para deployment, tal é feito usando um **self-hosted github-runner**, a correr como um *service*, isto é, um programa a correr no servidor a “ouvir” certas mensagens por parte do github, e que quando as recebe faz pull do repositório correndo o **workflow** no nosso servidor em vez de nos servidores do github. Este método acaba por ser mais assíncrono comparativamente a métodos que periodicamente questionam o github sobre o estado do repositório.

Posteriormente, o **workflow**, elimina todos os containers existentes (Frontend e Backend - mas obviamente não afetando a camada de persistência) e cria novas imagens iniciando-as de forma a que os serviços estejam disponíveis para o exterior.

```
on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master
name: Continuous Deployment
jobs:
  Tests:
    name: Running unit,integration and web tests
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup java
        uses: actions/setup-java@v1
        with:
          java-version: 1.11
      - name: Setup node
        uses: actions/setup-node@v1
        with:
          node-version: '12.6'
      - name: Unit and Integration
        run: mvn -f ./JustLikeHomeStarter verify -Dtest=!WebPlatformTesting;
      - name: Start frontend
        run: |
          npm ci;
          npm start &
        working-directory: ./Frontend/JustLikeHome/example
      - name: Functional Testing
        run: |
          mvn -f ./JustLikeHomeStarter verify -Dtest=WebPlatformTesting;
  CD:
    needs: [Tests]
    name: Continuous Deployment for React and Java Project
    runs-on: [self-hosted]
    steps:
      - name: Docker clean
        run: |
```

```

docker ps -aq;
docker kill $(docker ps -aq) || true;
docker rm $(docker ps -aq) || true;
docker volume ls -qf dangling=true;
docker volume rm $(docker volume ls -qf dangling=true) || true;
docker rmi $(docker image ls) || true;
- name: Server pull repository
  run: |
    git -C ~/TQS_2/ pull;
- name: CD for Spring-Boot Backend
  run: |
    docker build --tag "backend" ~/TQS_2/JustLikeHomeStarter;
    docker run -p 8080:8080 -d backend;
- name: CD for React Frontend
  run: |
    docker build --tag "frontend" ~/TQS_2/Frontend;
    docker run -p 80:3000 -d frontend;

```

3.3 Artifacts repository

Em termos de *artifacts* apenas mantemos, no servidor, o artifact funcional mais recente, este é gerado na pipeline durante o continuous delivery sendo posteriormente colocado a correr dentro de um container docker, desta forma os artifacts antigos são repostos pelo mais recente.

4 Software testing

1.1 Overall strategy for testing

Tendo-se optado pelo método de *TDD*, foi portanto feita a criação de testes *a priori* da lógica das classes envolvidas. Estes testes iriam inicialmente falhar, mas constituíam a métrica necessária para validar se a lógica posteriormente desenvolvida estaria correta.

Iniciou-se obviamente por produzir testes unitários, pela sua simplicidade de escrita bem como rapidez de execução. Posteriormente, quando as classes necessárias estivessem desenvolvidas passou-se a criar teste de integração. Os testes funcionais começaram a ser efetuados depois dos testes de integração de cada funcionalidade estivesse concluído.

Os testes estiveram desde o início do desenvolvimento integrados, sendo que se após o desenvolvimento de uma nova feature caso um teste estivesse a falhar, quer localmente quer na pipeline, este seria assinalado e seria normalmente o criador que teria de eventualmente o analisar e corrigir. O correto funcionamento de todas as features implementadas até ao momento era um requisito essencial para concluir o ciclo de desenvolvimento. Adicionalmente e como referido anteriormente, tornou-se imperativo que os métodos de testing possuísem nomes auto-explicativos, e tal pode-se verificar nos exemplos apresentados mais à frente.

Como ferramenta primária nos testes, fez-se uso de JUnit, Hamcrest, Selenium, Mockito e *tools* oferecidas pelo *SpringBoot*, pela facilidade e familiaridade destas.

1. Functional testing/acceptance

Em termos de recursos para testes funcionais usamos as seguinte ferramentas:

- Selenium
- Selenium Wait Driver
- Chrome Driver
- Katalon Recorder
- Junit5

Considerando os restantes tipos de testing feitos - Unit e Integration, que veremos mais à frente - nesta fase optou-se por criar testes num paradigma de *black box testing*. Como justificação para tal, basta considerar que os 2 tipos de testes referidos anteriormente garantem o correto funcionamento das camadas de serviço e persistência do produto, ficando apenas a *presentation layer* por verificar. Nesse sentido, considerou-se que a melhor aproximação neste contexto seria realizar testes colocando-nos na perspetiva do utilizador, uma vez que a lógica interna da página Web não se apresenta como um aspecto crítico no que diz respeito à experiência do utilizador e ao bom funcionamento do produto como um todo. Deste modo, os testes foram realizados de forma a simular todas as possíveis interações (isto é o *behaviour*) de um utilizador, não excluindo as eventuais escolhas errôneas do mesmo, garantindo assim uma Web Page bastante robusta aos mais diversos tipos de users.

Uma das ferramentas mais importantes aquando desta fase de *testing* de nível mais elevado foi o Katalon Recorder, que facilitou consideravelmente a criação destes testes uma vez que permite que de forma natural, e bastante similar à de um utilizador real, se fizessem as mais variadas averiguações. Não obstante, é importante salientar que esta ferramenta não constitui todo o processo de desenvolvimento de testes nesta fase. Isto deve-se ao facto de estes, quando já traduzidos para Java, precisarem de sofrer *tunning* de forma a tornarem-se o mais robustos possíveis, pois há diversos fatores que podem fazer com que pontualmente uma execução falhe - sendo o mais notório o tempo de espera no carregamento da página, no qual se fez uso de Selenium Wait Driver, que torna de forma bastante natural este processo de espera (principalmente porque não exige a definição de tempos de espera arbitrários, isto é espera no máximo x segundos até estar disponível mas caso esteja mais cedo não corre durante o resto do tempo de espera, enquanto um `Thread.sleep()` teria que esperar obrigatoriamente o tempo inteiro).

Outro fator importante é o facto de neste contexto se usar uma base de dados diferente da de produção. Assim, não só é usada uma base de dados em memória - H2 - como esta é especificamente carregada com dados *dummy* de forma a não só simular um cenário real mas também garantir que este tipos de testes são executados sobre as mesmas condições em qualquer altura. Visando a tal, tirou-se o máximo partido das funcionalidades de *SpringBoot* para o efeito.


```

@Test
void eliminateHouseFromBookmarker() throws Exception {
    driver.findElement(By.linkText("Imóveis")).click();
    driver.findElement(By.name("city")).click();
    driver.findElement(By.name("city")).sendKeys("viseu");
    driver.findElement(By.name("guests")).click();
    driver.findElement(By.name("guests")).sendKeys("1");
    driver.findElement(By.cssSelector(".btn-secondary")).click();
    wait.until(ExpectedConditions.elementToBeClickable(
        By.cssSelector(".rounded")
    ));
    driver.findElement(By.cssSelector(".rounded")).click();
    wait.until(ExpectedConditions.elementToBeClickable(
        By.cssSelector(".btn-warning")
    ));
    driver.findElement(By.cssSelector(".btn-warning")).click();
    assertEquals("Adicionado aos favoritos", driver.findElement(By.cssSelector(".col-lg-4 >
span:nth-child(5)")).getText());
    driver.findElement(By.linkText("Profile")).click();
    assertEquals("house by the beach", driver.findElement(By.cssSelector(".card:nth-child(2)
td:nth-child(2)")).getText());
    driver.findElement(By.linkText("Profile")).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(
        By.cssSelector(".card:nth-child(2) td:nth-child(2)")
    ));
    assertEquals("house by the beach", driver.findElement(By.cssSelector(".card:nth-child(2)
td:nth-child(2)")).getText());
    driver.findElement(By.cssSelector("td:nth-child(5) .fa")).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(
        By.cssSelector("h1")
    ));
    assertEquals("house by the beach", driver.findElement(By.cssSelector("h1")).getText());
    driver.findElement(By.linkText("Profile")).click();
    driver.findElement(By.cssSelector(".fe-trash")).click();
    {
        wait.until(new ExpectedCondition<Object>() {
            public Boolean apply(WebDriver driver){
                int elementCount =
driver.findElements(By.xpath("//*[@id=\"root\"]/div/div/div[3]/div/div/div[2]/div[2]/table/tbody/tr")).size();
                return elementCount==2;
            }
        });

    }

    assertEquals(2, driver.findElements(By.xpath("//*[@id=\"root\"]/div/div/div[3]/div/div/div[2]/div[2]/table/tbody/tr")).size());
}

```

2. Unit tests

Em termo de recursos para testes funcionais usamos as seguinte ferramentas:

- Mockito
- Hamcrest
- Junit5
- SonarQube

Representando o nível mais simples no que diz respeito a testes, e ao contrário dos mencionados anteriormente, aqui foi feita adotada uma perspectiva de desenvolvedor - ou de white box - onde a implementação futura da lógica era tida em conta aquando da criação dos testes. Desta forma ,se a lógica fosse retornar uma lista por norma , mas numa situação especifica retorna-se um HTTP 404 os testes previam isso mesmo antes da implementação ser feita. Sentiu-se como a melhor política a adotar, porque tratando-se do nível “mais baixo” de *testing* seria importante garantir que a lógica estava bem implementada, de forma a conferir confiança assim que se passasse para testes mais complexos.

Importante referir que estes testes foram feitos em todas as camadas do backend, isto é, no que diz respeito aos *controllers*, *services* e *repositories*.

O uso das ferramentas referidas acima foi fulcral para esta fase, uma vez que permitia a criação de mocks para métodos / classes ainda não implementadas - relembrando que é algo especialmente útil quando se trabalha sobre a metodologia TDD - bem como a possibilidade de expressão sobre forma de linguagem mais próxima da natural, tornando a leitura e perceção dos testes bastante mais intuitiva.

Durante o desenvolvimento da lógica por vezes apareciam situações em que o teste não estaria a contar com uma certa situação específica apenas detectada durante a implementação , nestas situações criava-se o teste e só depois era implementada a nova circunstância.

Procurou-se ter um code coverage grande uma vez que a lógica por detrás do sistema não era muito complexa, conseguimos corrigir alguns “leaks” de coverage com ajuda do SonarQube , uma vez que este reconhece as zonas que os testes não estão a cobrir.

3. System and integration testing

Nesta vertente, há que referir que o *SpringBoot* oferece bastantes *utils*, de forma a que este tipo de testes seja feito com maior facilidade: através do uso de simples tags como por exemplo `@MockMvc` ou `@SpringBootTest`, é possível muito rapidamente preparar um ambiente de *testing* mais complexo, característico dos IT.

Posto isto, e à semelhança do caso anterior, estes testes foram também feitos numa perspetiva de developer, embora talvez num grau não tão significativo quanto o anterior, já que nestes já não se tinha de ter tanto em atenção o funcionamento intra classe, mas sim verificar como os vários elementos comunicavam entre si num nível mais acima, o que de certa forma “fechava um pouco a caixa”. Contudo, não se deixou, de procurar que todas as respostas possíveis fossem testadas.

Através destes testes , se todos passarem, conseguimos garantir que o serviço está a funcionar como esperado e portanto pode ser deployed.

Ficheiros deste tipo eram eram identificados por `*IT.java`.

4. Performance testing

Com o intuito de verificar a robustez e confiabilidade do sistema a ser desenvolvido, optou-se também por criar testes a nível de performance.

Estes foram concebidos numa fase mais avançada do desenvolvimento, já após se ter uma aplicação web bem estruturada e obviamente uma pipeline de CD bem definida. Embora inicialmente se tenha considerado integrar este tipo de testes também na pipeline, esta foi uma opção que acabou por ser descartada, uma vez que se considerou que muitos fatores alheios ao programa em si poderia interferir com os resultados e, de certa maneira anunciar “falsos negativos”.

Este tipo de testes servem para aferir a robustez do software testando limites para lá dos normais da aplicação, sendo particularmente importantes quando se trata de software crítico. Não obstante, tendo em conta a natureza da aplicação em questão, considerou-se que limites “extremos” não seriam relevantes ou úteis, pelo que cenários um pouco acima dos expectáveis seriam os mais indicados averiguar.

Em termos de ferramentas usadas temos:

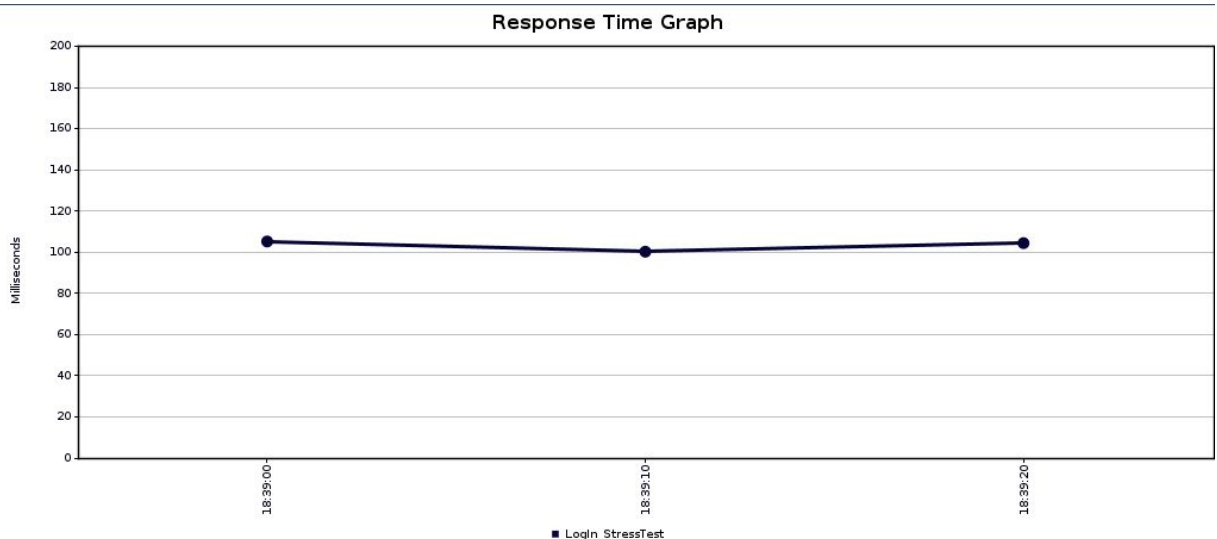
- Apache JMeter
- BlazeMeter

Neste sentido, foram tomadas 2 aproximações:

Na primeira fase, decidiu-se começar por um cenário mais simples, pelo que se verificou a capacidade de resposta da própria API face a inúmeros pedidos de Login num curto período de tempo. Aqui usou-se unicamente o JMeter para a definição do teste, graças à rica interface gráfica.

Após a criação do teste, optou-se por um uso mais recorrente à CLI por simplesmente ser mais eficiente aquando da iniciação da execução dos testes ao longo do tempo. Aqui era definido um ficheiro que iria conter o relatório do stress test efetuado, para posterior revisão.

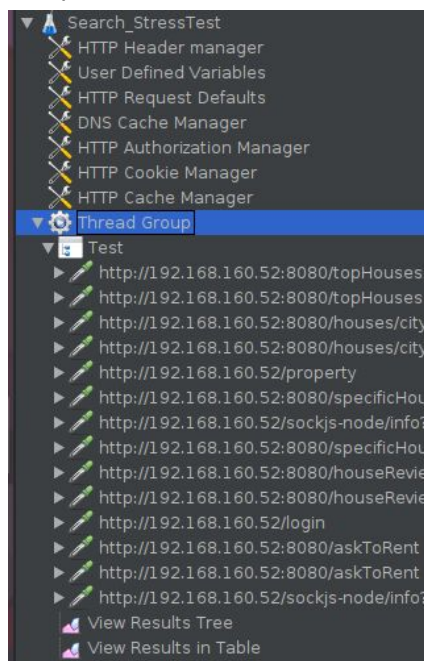
Assim, para cerca de 250 clientes a tentarem iniciar sessão num intervalo de 25 segundos, obtemos os seguintes gráficos:





Como se pode observar, os tempos de resposta rondam os 115 milissegundos, o que neste caso nos parece um ótimo resultado.

Numa segunda fase, fez-se uso da ferramenta BrazeMeter para a criação de um cenário mais realista: um utilizador sem sessão iniciada a procurar uma casa para arrendar, analisando-a e acabando mesmo por fazer o pedido para a arrendar.



Como se pode ver pela imagem acima, este teste apresenta uma complexidade considerável, uma vez que está a simular um use-case também mais complexo. Da mesma maneira, detetou-se que neste cenário o comportamento do servidor não era tão agradável, havendo pedidos a demorar bastante tempo, criando inclusive um impacto negativo na experiência do utilizador.

Como se pode ver na figura abaixo, à medida que o tempo passa e mais utilizadores acedem ao servidor, exigindo um maior número de operações, o tempo de resposta vai aumentando significativamente, chegando mesmo a atingir vários segundos, o que é algo bastante contrastante com o experimento anterior. Note-se que no gráfico exemplificado, reduziu-se em 100 o número de utilizadores a aceder simultaneamente, uma vez que em análises preliminares se concluiu que a capacidade era já bastante acima da tolerável.

Perante análises feitas localmente, o aumento do tempo de resposta aparenta ser um bottleneck existente no servidor e não na qualidade ou eficiência do código em si, uma vez que as análises locais apresentavam

valores mais próximos dos esperados (milissegundos em vez de segundos). Porém, os testes de performance tem que ser feitos em relação ao host daí os gráficos representados demonstrarem testes de performance inferior.

