

TQS: Manual de Garantia da Qualidade

Carolina Araújo [93248], Hugo Almeida [93195], Mariana Santos [93257], Miguel Almeida [93372]

v2020-06-07 to 2020-06-22

Gestão de Projeto	1
Equipa e Papéis	1
Gestão do Backlog Agile e Atribuição de Trabalho	2
Gestão da Qualidade do Código	2
Guidelines para Contribuidores (Estilo de Código)	2
Métricas de Qualidade de Código	3
Pipeline de Continuous Integration e Delivery (CI/CD)	3
Workflow de Desenvolvimento	3
Pipeline de CI/CD e Ferramentas	5
Testagem de Software	6
Estratégia Geral de Testagem	6
Testes Funcionais/ de Acceptance	6
Unit tests	6
Testes de Integração e Sistema	6
Testes de Performance	7

1 Gestão de Projeto

1.1 Equipa e Papéis

- Product Owner - Mariana Santos
- QA Engineer - Carolina Araújo
- Team Leader - Miguel Almeida
- DevOps Master - Hugo Almeida

1.2 Gestão do Backlog Agile e Atribuição de Trabalho

Embora se tenha utilizado o *ZenHub* ao invés do *PivotalTracker* [1] para fazer o *backlog management*, o *flow* dos *issues* das *user stories* é o mesmo:

- Escrita das histórias, no início de cada iteração ou quando algum membro ficava sem tarefas, sendo que as *issues* criadas eram posteriormente repartidas por todos;
- Quando a tarefa se encontrava completa (*definition of done* satisfeita), os membros do grupo eram notificados pelo próprio para irem verificar o *pull request* associado à *issue* no *GitHub*: dar *review* comentando, aprovando ou desaprovando. Este processo repetia-se até que o código associado à *issue* estivesse com qualidade considerada suficiente pelos restantes, aceitando-se o *pull request* e dando-se merge da *branch* para a *branch dev*;
- Por fim, as *issues* passavam para o estado *closed*, as *branches* eram apagadas e recomeçava-se o procedimento.

2 Gestão da Qualidade do Código

2.1 Guidelines para Contribuidores (Estilo de Código)

Uma das *guidelines* que o grupo colocou logo desde início foi a criação de exceções concretas que identificassem o problema de forma mais declarativa. Isto significa que em vez de utilizar uma simples exceção, a tom de exemplo poderia-se utilizar:

Problema	Exceção lançada
Utilizador passa um argumento inválido	<code>InvalidValueException</code>
Indivíduo a tentar autenticar-se está a usar credenciais erradas	<code>InvalidLoginException</code>
Quando não se encontram os resultados pretendidos para a procura realizada pelo utilizador	<code>ResourceNotFoundException</code>
Quando um estafeta pede um novo pedido sem ter terminado o anterior	<code>ForbiddenRequestException</code>

Ainda falando sobre as exceções, serviços que recebem parâmetros provenientes do utilizador que possam não estar 100% corretos fazem sempre a devida verificação. Se necessário, é lançada uma exceção que é então encaminhada pelo *controller* até ao utilizador.

Outra metodologia adotada para o serviço *SpecificService*, HumberPeças, foi o padrão Data Transfer Object [2].

Cada método criado é focado num objetivo concreto, sendo que, muitas das vezes, se necessário, criaram-se métodos adicionais dedicados a outra funcionalidade que fosse necessária no primeiro.

Os testes têm um nome padrão que permite facilmente entender daquilo que tratam e o que vai ser testado, bem como o outcome esperado:

test<functionality>_when<case being tested>_then<expected outcome>

O código foi desenvolvido tendo em mente que seria lido pelos colegas, e, portanto, facilmente legível. Assim sendo, sempre que possível foi-se facilitando a leitura rápida do código, seja com comentários, com o encurtar de linhas muito extensas, remoção de repetições desnecessárias e confusas...

2.2 Métricas de Qualidade de Código

Utilizou-se o *SonarQube* para realizar a análise estática do código, uma vez que esta ferramenta disponibiliza um conjunto de avaliações pertinentes do código desenvolvido, reportando *bugs*, *code smells*, etc.

Ponderou-se a alteração das *quality gates default* providenciadas pela plataforma. No entanto, o grupo não sentiu necessidade de alterar o padrão de qualidade visto que estes já eram bastante altos. Vale a pena referir que alguns problemas reportados que não eram requisitos do projeto foram marcados como resolvidos não contando para as regras de *quality gate*. Um exemplo disto foram as configurações de *cors* que estão definidas para permitir qualquer endereço.

Conditions

Conditions on New Code

Metric	Operator	Value
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 Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Figura 1: *Default Quality Gates - SonarQube*

3 Pipeline de Continuous Integration e Delivery (CI/CD)

3.1 Workflow de Desenvolvimento

O *GitHub Kanban* permitiu que o grupo se possa sentir-se organizado no que toca aos *commits*, *pull requests*, saber quando um membro necessita de ajuda num branch, ver o progresso feito numa determinada iteração, *etc...* Por outro lado, o *ZenHub* definitivamente permite uma muito melhor absorção daquele que é o objetivo e quais os benefícios de utilizar *user story driven issues*.

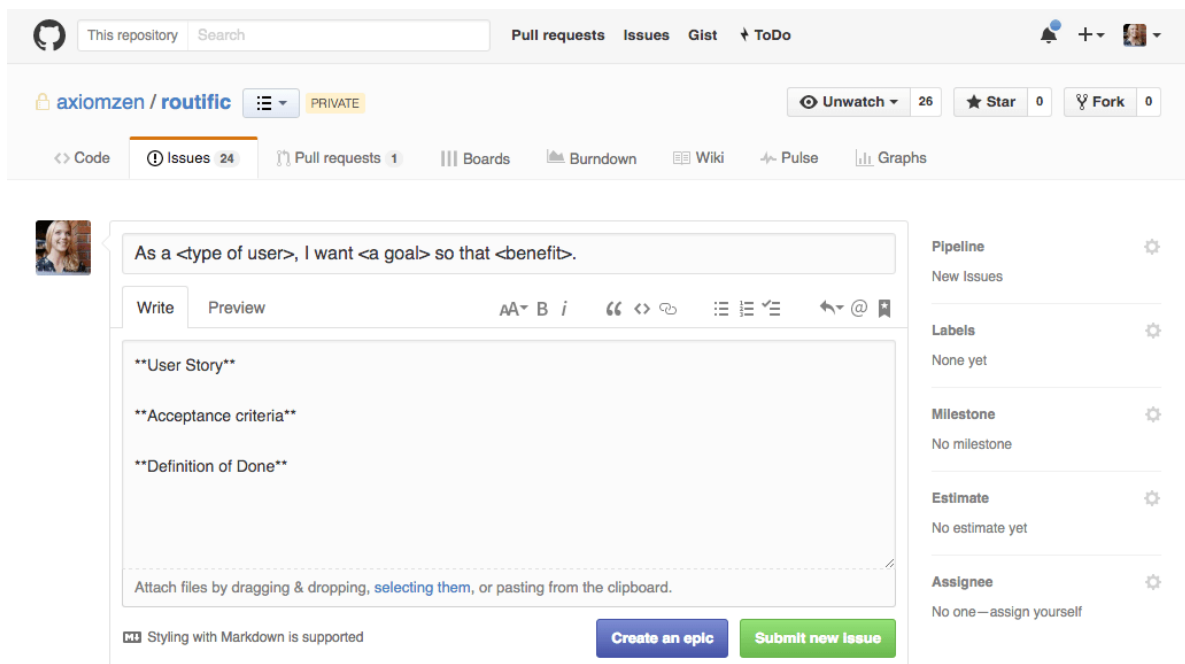
Assim, a equipa mantém-se organizada a partir das duas plataformas, recolhendo os benefícios de ambas. O *ZenHub* permite ver com facilidade quais as novas *issues*, quais as que têm de ser feitas, o que elas envolvem, entre outros. Para além disso, facilmente as *issues* podem ser editadas: adicionar *acceptance criteria* e *definition of done*. Finalmente, também é prático poder ver cada *Epic* em pormenor, o seu grau de completude, os *issues* que lhe pertencem, bem como facilmente adicionar novos, se necessário.

Por outro lado, o *GitHub Kanban* permite verificar quais os *pull requests* que ainda precisam de ser revistos por membros, que precisam de ser *merged*, comentar os mesmos, definir *projects* (iterações) e organizar o repositório, no que toca às *branches*.

Uma vez que cada *issue* criada está diretamente associada a um *user story*, cada *branch*, estando associada a um *issue*, está também diretamente correlacionada com as *user stories*.

Uma regra geral implementada desde início do projeto, através de uma definição do repositório do *GitHub*, implica que todos os *pull requests* necessitem de pelo menos 2 *reviewers* (outros colegas a analisar o código e aceitar/rejeitar, e/ou comentar). Deste modo, garante-se que toda a equipa está envolvida no processo de contribuição com novas funcionalidades, bem como no progresso dos seus *peers*.

Uma *user story*, regra geral, tem a seguinte organização:



The screenshot shows the GitHub 'Issues' page for the repository 'axiomzen / routific'. The 'Issues' tab is selected, showing 24 issues. A form for creating a new issue is displayed, with a template for a User Story. The template includes fields for 'As a <type of user>', 'I want <a goal>', and 'so that <benefit>'. Below the template, there are sections for 'User Story', 'Acceptance criteria', and 'Definition of Done'. The right sidebar shows settings for the issue, including Pipeline, Labels, Milestone, Estimate, and Assignee.

Figura 2: Agile Concepts in GitHub [3]

O título, por si só, permitia que cada *dev-op* soubesse exatamente aquilo que teria de ser feito para poder completar a *issue*, a partir do *goal* que o user pretendia ver acessível. No entanto, a descrição pode ser importante para que terceiros ou colegas entendam o progresso da *issue*, ver se está aceitável e se pode ser finalmente fechada.

No entanto, grande parte das *user stories* tinham exatamente a mesma estrutura e pediam sempre os mesmos critérios.

A definição de feito, para cada *user story*, vai depender da quantidade de *acceptance criteria* que vão sendo verificadas à medida que o trabalho é completado para tornar aquela *user story* factível. No entanto, resumidamente, a definição de 'feito' é dada pelo objetivo geral do *issue/user story* em si. Por exemplo, o grupo considerava que um *issue* estava feito quando o código relativamente ao mesmo já estava completamente testado... isto deixou de parte a integração dos endpoints com o frontend nalguns casos de *definition of done*, uma vez que o objetivo inicial foi garantir que um número considerável de endpoints estavam prontos.

```
** User Story **
Rider manager

** Acceptance Criteria **

• endpoint ready for the rider manager to be able to GET the information
• authorization used, for the safety of the information
• rest controller mapped to this endpoint is created
• rest controller template IT tests
• rest controller mock mvc tests
• the needed services are created to communicate with the repositories
• service tests with mock repositories
• connect this to the frontend

** Definition of Done **

• all of the above, but they are in their importance order (sort of)
```

Figura 3: Exemplo de uma *issue* criada para o projeto

As *user stories* englobam a criação de um endpoint, um *RestController* que mapeie o mesmo, serviços que esse controller possa necessitar e, então, testes de ambos: ***REST Controller Mock MVC***, ***REST Controller Template IT Tests*** e ***Service Tests***. Por fim, poderia haver a ligação de estes componentes ao *frontend*, e testes relativos para ver o *flow* completo.

3.2 Pipeline de CI/CD e Ferramentas

As práticas de Continuous Integration são feitas a partir do *GitHub Actions*. A pipeline de CI corre dois tipos diferentes de testes:

- Testar a *build* das aplicações *web* em *Angular*
- Testar os serviços de *Spring Boot*.

Commits para *branches* com *pull requests* abertos no *GitHub* permitem aos membros, com uma pipeline de *CI* integrada, verificar se tudo está conforme os requisitos, *i.e* se todos os testes passaram. De notar que, para os testes dos serviços em *Spring Boot*, foram definidas as regras *default* de *Quality Gate* que necessitam de ser cumpridas para que os testes passem, estes últimos que, dada a arquitetura do sistema, necessitam da execução da aplicação *web* correspondente e do serviço com que se liga em produção para a execução dos testes com *Selenium* e de *Integração*, respetivamente.

Por outro lado, as práticas de Continuous Deployment também foram integradas no projeto através do *GitHub Actions*. Sempre que existe um *commit* para a *branch main* é executada uma *pipeline* que executa comandos *ssh* numa máquina virtual da plataforma *Google Cloud* onde é feita novamente a *build* dos *containers Docker* e a nova execução dos mesmos.

4 Testagem de Software

4.1 Estratégia Geral de Testagem

Ainda que alguns dos membros tenham sentido que foi um pouco difícil para adaptar e incorporar esta prática, tentou-se seguir a abordagem de *Test Driven Development (TDD)* [3], sempre que possível.

Foram utilizadas diferentes ferramentas e modos de teste durante o desenvolvimento do projeto, *REST-Assured* é um exemplo disso no serviço *SpecificService* e, outro exemplo fora do habitual é a utilização do *Page Object Pattern* durante o desenvolvimento dos testes de aceitação com *Selenium*.

4.2 Testes Funcionais/ de Acceptance

Os Testes Funcionais foram desenvolvidos com recurso às ferramentas *Selenium* e *TestContainers*. Isto acontece porque o grupo queria garantir a execução destes testes na pipeline *CI* logo, o *browser* utilizado pelo *Selenium* teria de ser automaticamente configurado independentemente do sistema onde está a ser desenvolvido, daí os *TestContainers*.

Nestes testes tentou-se garantir o bom funcionamento das aplicações *web*, bem como das *Rest APIs* uma vez que estas necessitam de estar a ser executadas para o funcionamento das aplicações. Assim, testou-se a aplicação tendo a base de dados em diversos estados: com dados, sem dados, parcialmente com dados, etc, de modo a tentar abranger o máximo de cenários possíveis.

De notar que, uma vez que a aplicação *web* tem algumas funcionalidades que utilizam modais, existem alguns testes que foram colocados como desativados, com a devida explicação no método, uma vez que trariam instabilidade aos testes da pipeline *CI*.

4.3 Unit tests

Os Testes Unitários têm como objetivo testar apenas uma parte isolada do sistema, ou uma *unidade*, para garantir que as partes mais básicas do sistema se comportam como esperado.

Para este propósito, desenvolvemos testes com objetivo de testar o comportamento dos repositórios, dos *Services* e dos *Controllers*.

Para testar a camada de persistência, ou repositórios, usamos a anotação `@DataJpaTest`, com a base de dados em `Testcontainers`. Fez-se testes para não só todos os métodos declarados explicitamente por nós na interface de cada repositório, como alguns outros métodos que eram usados frequentemente. Nestes testes, tentou-se sempre testar casos extremos, ou de erro, para além dos casos de utilização mais “normal”.

Quanto aos *Services*, utilizou-se a extensão `Mockito` para se conseguir testar os métodos independentemente da camada de persistência, ou seja, utilizou-se `mocks` em vez dos repositórios. Mais uma vez, tentou-se fazer vários testes representativos de vários cenários, dando ênfase a possíveis cenários de erro.

Por fim, para os Testes Unitários sobre os métodos dos *Controllers* utilizou-se `MockMvc` (no serviço genérico) e `RestAssuredMockMvc` (no serviço específico), permitindo assim utilizar `mocks` no lugar dos *Services*, isolando o comportamento dos *Controllers*. Tal como nos outros casos, foram desenvolvidos testes que cobrem vários casos de uso, incluindo alguns que produzem resultados não desejados normalmente.

4.4 Testes de Integração e Sistema

Os Testes de Integração visam entender se todos os componentes envolvidos no processo de resposta a um pedido por parte de um cliente ao serviço não levantam qualquer problema. O *Spring Mvc* mapeia os pedidos *HTTP* para o respectivo *Rest Controller*, se existir um dedicado a esse *URI*.

É preciso confirmar se todos os pedidos que esse *Controller* venha a receber, estão devidamente controlados, *i.e* se as componentes se comportam como devido.

Então, para cada endpoint foi criado um teste de *TemplateIT* onde se verificaram que as diversas exceções e verificações que se pretendiam desenvolver realmente aconteciam. Desta forma, foi possível garantir a qualidade dos *endpoints* desenvolvidos e a sua garantia contra erros.

4.5 Testes de Performance

Para os testes de performance, foi utilizada a ferramenta open source `Apache JMeter`. Na sua interface gráfica foram criados 2 *test plans* diferentes, devido a diferentes desafios que foram surgindo. Estes testes apenas foram efetuados sobre o serviço genérico.

Ambos os planos têm algumas características em comum:

- Ambos têm 3 *thread groups* (*Clients/Stores Pool*, *Riders Pool* e *Managers Pool*) em que cada um corresponde ao tipo de utilizador que normalmente invoca o *endpoint*.
- O número de threads no thread group *Clients/Stores Pool* é de 70 com um ramp-up period de 120 segundos; enquanto que o número de threads no thread group *Managers Pool* é de 25 com um ramp-up period de 120 segundos.
- O *Clients/Stores Pool* apenas faz pedidos ao endpoint dedicado para receber compras.
- Quanto às threads da *Managers Pool*, estas vão fazer pedidos a todos os endpoints dedicados à gestão do sistema que existem no sistema, para além do que permite *login*.
- O *Riders Pool* simula condutores que fazem pedidos de *login*, pedem por todas as encomendas já realizadas por si e também pelas suas estatísticas.

- Para visualizar os dados, é utilizado um Simple Data Writer *listener*.

No primeiro *test plan*, o Riders Pool faz pedidos a um número maior de *endpoints*. Para além dos já referidos, o condutor simulado vai também pedir pela encomenda atual, pedir novas encomendas, e também atualizar o estado da encomenda que lhe foi atribuída. Para definir quais pedidos ele pode fazer ou não são utilizados *If Controller* que verificam o estado da sua encomenda atual, sendo que só irá pedir uma nova caso este seja “*DELIVERED*”.

Contudo, esta abordagem tem problemas de concorrência. Como apenas está a ser utilizado um utilizador nas várias threads, vai haver conflitos quanto ao pedido de novas encomendas e de atualizar os status das encomendas, sendo que o que aconteceria seria ele atualizar várias vezes seguidas o estado, mesmo depois da encomenda atual estar entregue. Como este comportamento gera um erro no *report* retornado, decidiu-se que esta *thread group* apenas teria uma thread, mas seria executada 70 vezes (com um ramp-up period de 2 segundos), simulando 70 pedidos para cada endpoint do mesmo utilizador. Apesar de se conseguir enviar pedidos a quase todos os *endpoints*, esta solução não é muito escalável.

Para resolver isto foi criado o segundo *test plan*, em que foi retirado os pedidos a estes endpoints que geram conflitos, conseguindo assim utilizar várias threads também para este *thread group*. Nomeadamente, são usadas 70 threads com um ramp-up period de 120 segundos. Os resultados presentes no repositório foram gerados tendo em base este *test plan*. Para além do ficheiros JTL, a ferramenta JMeter também gera um website a partir dos dados obtidos, que permite a análise destes.

A partir do website gerado pelo JMeter conseguimos obter alguns gráficos interessantes, tal como o da Figura 4, que mostra o tempo médio que demorou a receber uma resposta para cada pedido. O gráfico da Figura 5, também obtido no mesmo website, mostra que todos os pedidos foram realizados sem a ocorrência de uma falha, provando que o nosso serviço conseguiria aguentar uma quantidade de tráfego considerável, estando assim adequado para produção quanto a este parâmetro.

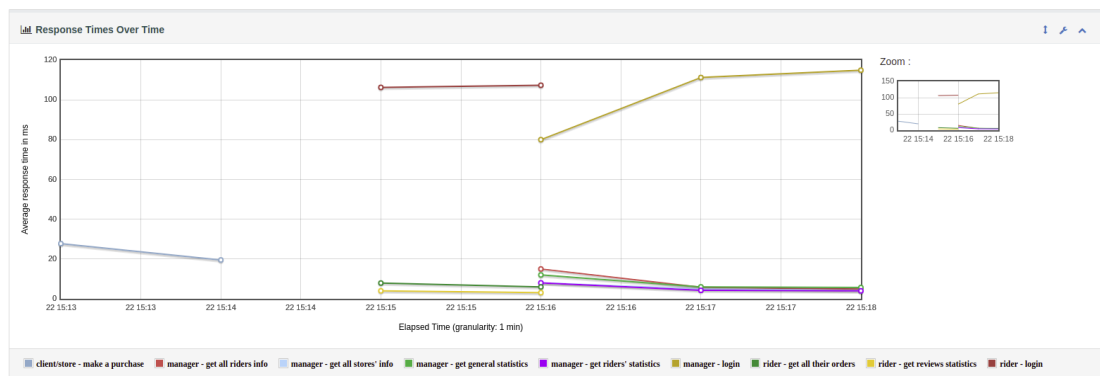


Figura 4: Gráfico gerado pelo JMeter com o tempo médio de respostas

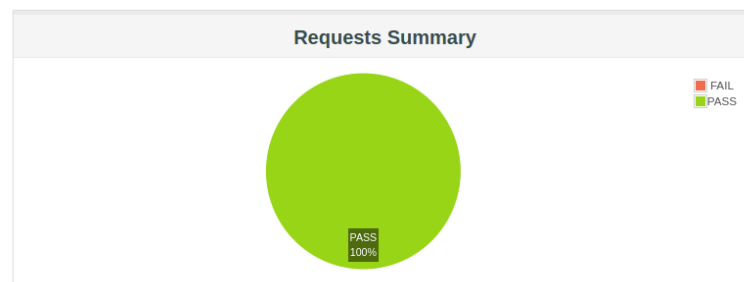


Figura 5: Gráfico gerado pelo JMeter com quantidade relativa de respostas com e sem sucesso

References

- [1] Workflow overview. (n.d.). Retrieved from https://www.pivotaltracker.com/help/articles/workflow_overview/
- [2] Seniuk, V. (n.d.). Data Transfer Object Pattern in Java - Implementation and Mapping. Retrieved from <https://stackabuse.com/data-transfer-object-pattern-in-java-implementation-and-mapping>
- [3] Agile Concepts in GitHub and ZenHub. (n.d.). Retrieved from <https://help.zenhub.com/support/solutions/articles/43000010338-agile-concepts-in-github-and-zenhub>
- [4] What is Test Driven Development (TDD)? (2021, March 04). Retrieved from [https://www.agilealliance.org/glossary/tdd/#q=~\(infinite~false~filters~\(postType~\(~'page~'post~'aa_book~'a_a_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video\)\)~tags~\(~'tdd\)\)~searchTerm~'~sort~false~sortDirection~'asc~page~1\)](https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'a_a_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video))~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1))
- [5] Team, T. G. (2019, September 16). GitHub Actions: Continuous Integration. Retrieved from <https://lab.github.com/githubtraining/github-actions:-continuous-integration>

