deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

Quality Assurance manual

Autores: **85044: Luís Costa**
86182: Rui Coelho
89138: Pedro Iglésias
91153: João Laranjo

Data: **14/05/2020**

Índice

1. Gestão do projeto

1.1 Equipa e papéis

Papel	Responsabilidades	Membro
Team Manager	Assegurar que existe uma distribuição justa das tarefas e que os membros trabalham de acordo com o planeado. Promover ativamente a colaboração entre os membros da equipa bem como tomar a iniciativa para endereçar os eventuais problemas que possam surgir. Assegurar que os resultados esperados do projeto são entregues dentro dos prazos.	Rui Coelho
Product Owner	Representa os interesses dos *stakeholders*. Ter um vasto conhecimento do produto e o seu domínio de aplicação. Pessoa a quem a equipa deverá colocar as dúvidas que tem sobre as *features* esperadas do produto. Deve estar envolvido na aceitação dos incrementos à solução.	Luís Costa
Arquiteto	Conhecimento vasto da arquitetura proposta bem como o suporte das tecnologias. Todas as dúvidas que possam surgir relativas ao comportamento de cada componente bem como a interação entre módulos deve ser esclarecida com este.	Pedro Iglésias
DevOps Master	Responsável pelo desenvolvimento e produção de infraestruturas para as configurações requeridas. Assegurar que a ferramenta de desenvolvimento funciona de forma apropriada. Lidera a preparação do ambiente de *deployment*, repositório *git*, infraestruturas em *cloud*, operações da base de dados, etc.	João Laranjo
Desenvolvedores	Contribuir para o desenvolvimento das tarefas.	Todos

1.2 Gestão de backlog e distribuição do trabalho

Como ferramenta para gestão do *backlog* decidiu-se usar Pivotal Tracker. Esta ferramenta permitirá acompanhar a evolução e o desenvolvimento do projeto. Facilita também a visualização dos objetivos, permite gerir as entregas e as metas ao longo do trabalho e incentiva à divisão do projeto em partes mais simples.
É importante que haja uma apresentação transparente do estado atual e esperado do projeto. Devem ser tomadas as devidas medidas de forma a que os prazos sejam cumpridos e o projeto se encontre em continuo desenvolvimento.
As entregas devem ser revistas e analisadas antes de submetidas para que se tenha a certeza de que o objetivo foi concluído e que satisfaz os requisitos.

Para além disso é importante frisar a divisão ágil do trabalho. Este será dividido em módulos (*epics*) que por sua vez agrupam várias funcionalidades. Cada *feature* tem várias funções e estas possuem os seus requisitos funcionais. É também através do Pivotal Tracker que se pode gerir de melhor forma os módulos e *features* do trabalho.

2. Gestão da qualidade do código

2.1 Guidelines para os programadores (coding style)

A nível de *guidelines* para o estilo de código iremos adotar o *styleguide* para Java disponibilizado pela Google (consultar referência) [1] para todo o código desenvolvido em SpringBoot. Para o código desenvolvido em Flutter seguiremos as guidelines oficiais (consultar referência) [2]. O código desenvolvido em React irá todo ele seguir as indicações presentes no guia Pagarme (consultar referência) [3]. Todo o código escrito na criação da nossa plataforma de comércio online seguirá estes *coding styles,* sob pena de não ser aceite caso não se encontre em conformidade. Sempre que exista alguma dúvida sobre o estilo a adotar o *developer* deve referir-se ao *styleguide* de modo a obter esclarecimento. Caso seja algo que suscite várias abordagens a nível de estilo, a abordagem a tomar será decidida em conjunto pela equipa.

2.2 Métricas de qualidade de código

Todo o código escrito deve ser avaliado por ferramentas de *static code analysis.* No nosso caso, irão ser utilizados o Sonarqube e a Codacy. De modo a que um excerto de código seja aceite, ambas as ferramentas devem dar um feedback positivo. Caso o excerto falhe algum dos testes, será recusado e o *developer* irá ter de fazer *refactor* no seu código e/ou corrigir os bugs detetados.
Outra métrica de qualidade de código a ser implementada é a atribuição de um grau de severidade às falhas/bugs encontrados durante o desenvolvimento da nossa aplicação. Esta escala de severidade irá conter 4 níveis, sendo eles:

- 1 – Muito baixo
- 2 – Baixo
- 3 – Moderado
- 4 – Alto (prioritário).

Com base nesta escala, iremos priorizar a criação de testes de modo a aumentar a nossa *test coverage* para os problemas encontrados.

3. Pipeline de entrega contínua (CI/CD)

3.1 Workflow de desenvolvimento

Iremos adotar um *workflow* baseado no Github Flow [4]. Será criada uma *branch* (*featured branch*), isto é, cada *feature* terá a sua própria *branch*, são realizados alguns commits e, assim que o programador achar que o seu código está pronto cria um pull request no qual a restante equipa irá avaliar o código feito por este, discutir e chegar à conclusão de aceitação ou não. É de salientar que, nesta fase, serão realizados também todos os testes de software que serão levados a cabo pelo Github Actions, com isto teremos dois cenários:

1. O **código é aceite** e, portanto, pode ser dado o seguimento, isto é, pode ser colocado na branch master onde será realizado o consequente deployment. Considera-se aceite todo o código que tem parecer positivo da equipa e também das ferramentas de análise de código estático bem como dos testes de software realizados.
2. O **código não é aceite** e, portanto, o programador responsável por essa feature terá de rever o seu código e melhorá-lo de forma a estar em conformidade com as ferramentas de análise estática de código, a opinião dos colegas e os resultados positivos dos testes de software.

3.1.1 Definição de concluído

É considerado concluído todo o código que obedeça aos seguintes critérios:

- Feedback positivo do Sonarqube
- Feedback positivo da Codacy
- Todos os testes devem ter passado
- Aprovação do pull request de pelo menos dois membros da equipa

3.2 CI/CD pipeline e ferramentas

O pipeline de CI/CD será composto por dois serviços baseados em cloud. A componente de CI será levada a cabo pelo Github Actions e a parte de CD será levada a cabo pelo Heroku.
Existem três componentes base para este projeto, um API feita em Spring Boot, o frontend escrito em ReactJS e uma aplicação móvel feita em Flutter. O pipeline nestes três casos é ligeiramente diferente.
No caso do frontend e da API, será sempre realizado o deployment caso o código tenha condições para tal e, por isso, as duas ferramentas (Github Actions e Heroku) serão usadas.
No caso da aplicação móvel apenas poderemos realizar os testes e construir o seu ficheiro de instalação, assim, só será utilizado o Github Actions que tratará não só de realizar todos os testes necessários bem como criar o ficheiro de instalação para dispositivos android.
Na fase de CI, considera-se apto todo o código que passe corretamente todos os testes escritos pelos desenvolvedores e que tenham, também, passado na análise estática de forma positiva.

4. Testes de software

4.1 Estratégia geral de testagem

Para a realização de testes será utilizada uma abordagem de test-driven development ao longo de todo o projeto.
Numa primeira fase é construído o código de modo a encontrar uma solução que funcione para a feature que desejamos, após isso, devem ser construídos testes de software que iremos executar por forma a determinar a qualidade e robustez do mesmo.

4.1.1 Testes funcionais e de aceitação

4.1.1.1 Testes funcionais

As funcionalidades esperadas pelo software irão ser testadas com o auxílio de testes aos quais irá ser fornecido a *input data* esperada pelos métodos da classe e determinado output produzido por esse método. Após a execução de um teste, irá ser feita, de modo automático, a comparação entre o output produzido pelo teste, com o output esperado após a execução do método. Finalmente iremos verificar se a aplicação está a funcionar de acordo com as necessidades dos utilizadores.

4.1.1.2 Testes de aceitação

Os testes de aceitação irão ser executados recorrendo ao uso de *data* previamente determinada. O resultado destes testes irá ser guardado e irá ser feita uma comparação com os resultados esperados. Por fim, é determinado se os resultados coincidem com os esperados.

4.1.2 Testes unitários

Ao nível da API, deverão ser criados testes para todas as classes (excetuando as classes de configuração como, por exemplo, a classe de configuração do Swagger, a classe *Main* e todo o código auto-gerado). Quando houver necessidade de realizar testes que envolvam a integração de recursos externos, o teste deve ser realizado recorrendo à utilização de *mock objects*.
Os testes devem basear-se nas funcionalidades esperadas pela classe a ser testada, isto é, devem ser testados tanto os casos em que uma determinada classe recebe um *input* correto, sendo que neste caso o que está a ser testado é se o *output* produzido pela classe é o desejado, como casos em que a classe recebe um *input* incorreto, em que se testa se o tratamento de erros se encontra aplicado de forma correta.

4.1.3 Testes de Sistema e integração

4.1.3.1 Teste de sistema

Os testes de sistema irão ser conduzidos pelos *stakeholders,* com base na informação presente no manual de qualidade do software e no relatório de especificações do produto. Estes testes irão envolver a execução de cada componente de software desenvolvido, e do sistema no geral, de modo a avaliar as diversas funcionalidades implementadas. Estes testes irão indicar se cada componente é capaz de satisfazer os requisitos definidos nas especificações do produto. Caso o componente cumpra os requisitos definidos, irá ser verificado se o faz num tempo aceitável.

4.1.3.2 Teste de integração

De modo a testar a interação entre os componentes do produto, os testes irão a replicar a interação dos módulos uns com os outros, começando por testar, por exemplo, a interação entre dois módulos e indo escalando até se testar a aplicação como um só.
Estes testes irão ser realizado recorrendo a uma *bottom-up strategy*. Com base nesta estratégia, iremos começar por testar os componentes de nível mais baixo e/ou que dependem pouco de outros componentes, indo aumentado a *test coverage* até que todos os componentes se encontrem testados.

4.1.4 Testes de performance

Teste de carregamento de dados vai ser usado para verificar os problemas que a nossa aplicação tem quando os utilizadores carregam os dados antecipadamente.

Teste de stress vai ser usado para verificar o limite da nossa aplicação e quando ela deixa de funcionar decentemente simulando uma grande quantidade de utilizadores e uma grande quantidade de operações serem realizadas a utilizar a aplicação ao mesmo tempo.

Teste de resistência vai ser usado para verificar se o software consegue aguentar ativo durante longos períodos.

Teste de espontaneidade vai ser usado para verificar a reação do software quando este recebe uma grande quantidade de informação ao mesmo tempo.

Teste de volume vai ser usado para verificar o desempenho/velocidade do software com diversos volumes da base de dados da aplicação.
Teste de escalabilidade vai ser usado para verificar se a aplicação é escalável ou não para suportar uma maior carga de utilizadores caso seja necessário.

5. Referências

[1] “Google Java Style Guide,” Google, [Online]. Available: https://google.github.io/styleguide/javaguide.html. [Acedido em 14 Maio 2020].

[2] Greg Spencer, “Style guide for Flutter repo,” Flutter, [Online]. Available: https://github.com/flutter/flutter/wiki/Style-guide-for-Flutter-repo. [Acedido em 14 Maio 2020].

[3] Pagarme, “React Code Style Guide,” Github, [Online]. Available: https://github.com/pagarme/react-style-guide. [Acedido em 14 Maio 2020].

[4] “Understanding the GitHub flow,” Github, [Online]. Available: https://guides.github.com/introduction/flow/. [Acedido em 14 Maio 2020].