TQS: QA Manual

Maria-Aleksandra Korjenevskaya [118769], Ivan Horoshko [120603], Gonçalo Ferreira [120189]
v2025-01-30

Conteúdos

1 Gestão de projeto

1.1 Papéis atribuídos

Papel	Membro/os do grupo	Descrição
Team Coordinator (a.k.a. Team Leader)	Ivan Horoshko	Coordena a distribuição de tarefas e o cumprimento do plano, promovendo a colaboração e a resolução de problemas para garantir a entrega atempada do projeto.
Product owner	Maria-Aleksandra Korjenevskaya	Representa os interesses dos stakeholders e esclarece os requisitos do produto, validando a aceitação dos incrementos com base no seu conhecimento profundo do domínio.
QA Engineer	Gonçalo Ferreira	Responsável, em articulação com outros papéis, por promover as práticas de garantia de qualidade e colocar em prática instrumentos para medir a qualidade da entrega. Monitoriza se a equipa segue as práticas de QA acordadas.
DevOps master	Ivan Horoshko	Gere a infraestrutura e configurações de desenvolvimento e produção, assegurando o funcionamento da framework e liderando as operações de deployment.
Developer	Todos os membros do grupo	Participa nas tarefas de desenvolvimento, sendo as suas contribuições rastreadas através de pull requests e commits no repositório da equipa.

1.2 Refinamento do backlog e monitorização do progresso

Organização do trabalho:

- Usamos o Jira para gerir o *Project Backlog*, onde criamos *Epics*, *User Stories* e *Sprints*.

- O nosso trabalho é decomposto em *User Stories*. Cada *Story* recebe uma estimativa de esforço através de *Story Points* e tem critérios de aceitação, definindo a *Definição de Concluído*.
- Usamos o Confluence para a documentação.
- Usamos o Github Actions para CI Pipeline.

Monitorização do progresso através de:

- *Story Points,* utilizados para estimar o esforço das user stories e acompanhar a capacidade da equipa ao longo das iterações.
- *Burndown Charts (Board),* que permitem visualizar diariamente o trabalho restante na sprint.
- Ferramentas de relatório do projeto, como métricas de velocidade e evolução das tarefas disponíveis no Reporting Center do Jira.
- Avaliação da cobertura de requisitos, integrada através do sistema de gestão de testes Jira + Xray.

2 Gestão da qualidade do código

2.1 Política da equipa para o uso de IA generativa

Posição da equipa

A equipa incentiva o uso de assistentes de IA como uma ferramenta de apoio e aceleração, mas não de substituição. A responsabilidade final pela qualidade, segurança, adequação e testes do código gerado é sempre da nossa equipa.

- **O que fazer:**
  - Acelerar o desenvolvimento: Usar a IA para gerar código repetitivo, endpoints básicos, ou classes de teste (mocks).
  - Apoio a Testes: Gerar exemplos de testes unitários para casos limite ou gerar dados de teste.
- **O que não fazer:**
  - Ignorar o Pipeline SQA: Nunca submeter código gerado por IA que não passe nos Quality Gates (SonarQube) ou que não tenha cobertura de testes adequada.
  - Nunca inserir informação confidencial no assistente de IA relacionado com o projeto.
  - Não fazer copy-paste de código de produção complexo sem o compreender e sem verificar a lógica dele.

2.2 Diretrizes para contribuidores

Estilo de código

- Ser consistente com o estilo de código *Java/Spring Boot* já existente no repositório.
- Evitar repetições de código.
- Reduzir imports e packages desnecessários.

Revisão de código

- Quando fazer: A revisão é obrigatória via *Pull Requests* antes de qualquer *merge* para a *branch* principal.
- Foco: O membro da equipa foca-se na lógica, no cumprimento dos critérios de aceitação e na conformidade com o estilo de código.
- Ferramentas AI: A revisão pode ser suportada por ferramentas AI para identificação de alguns erros.
- Extensão de SonarQube no Visual Studio Code que analisa o código a procura de erros ou más práticas.

2.3 Métricas de qualidade de código e dashboards

Práticas:

- Análise estática de código obrigatória com SonarQube em cada commit ou *Pull Request*.
- Cobertura de código medida com JaCoCo e integrada ao SonarQube.
- Avaliação de vulnerabilidades feita pelo SonarQube.

Quality Gates:

- Cobertura de código: mínimo de 80% para código novo.
- Vulnerabilidades: zero bugs e vulnerabilidades graves ou críticas.

Dashboards:

- O principal recurso para acompanhamento da qualidade é o dashboard do SonarQube.

3 Pipeline de entrega contínua (CI/CD)

3.1 Fluxo de trabalho de desenvolvimento

Fluxo de codificação

Seguindo o **GitHub Flow** para o desenvolvimento de *software*.

Cada vez que se pretende adicionar uma nova funcionalidade, isto é, desenvolver código para cumprir uma história de utilizador, ou se pretende corrigir algum erro, é criada uma *branch* com a seguinte convenção:

- **SCRUM-[Nº da chave JIRA]-[Nº história]-[Nome da história]**

Commits tem de seguir a seguinte convenção (retirado de https://gist.github.com/qoomon/5dfcdf8eec66a051ecd85625518cfd13) :

[Tipo] ([Âmbito]) : [Descrição]

Tipo	Descrição
feat	adiciona uma nova funcionalidade
fix	corrige um *bug*
refactor	reestruturar código sem alterar as funcionalidades
style	relativo ao estilo de código não em si na lógica do código
test	adição de testes ou correção dos mesmos
docs	relativo a documentação
build	relativo a ferramentas de *build*, depedências, ...
chore	outros (mudanças que não alteram a aplicação nem os testes)

Cada vez que é dando um *commit*, é feita a verificação dos testes unitários, anteriormente desenvolvidos, só se pode avançar com o *pull request* se os testes unitários passarem. Quando dada uma história de utilizador é completa, é feito um *pull request*, que ao cumprir a definição de feito (DoD), *branch* dado o *merge* com a *main*.

Definição de concluído:

- Testes unitários passam
- Testes de integração passam
- Cobertura de código ≥ 80%
- Critérios Aceitação são cumpridos
- Passa o Quality Gate do SonarCloud
- Aprovado pelo menos por 1 membro da equipa

O *pull request* tem de seguir este ***template:***

US[Nº da História]
[Nome da História]

[Descrição da História]

Critérios de Aceitação

- [ ] Critério 1
- [ ] Critério 2

Definição de Concluído

- [ ] Testes unitários passam
- [ ] Testes de integração passam
- [ ] Cobertura de código ≥ 80%
- [ ] Critérios Aceitação são cumpridos
- [ ] Passa o Quality Gate do SonarCloud
- [ ] Aprovado pelo menos por 1 membro da equipa

Em seguida a história de utilizador é considerada como **concluída**.

3.2 Pipeline e ferramentas de CI/CD

Ferramenta CI/CD: Github Actions.

CI Pipeline:

- Checkout: Obter o código da branch.
- Build: Compilação do projeto Spring Boot (Maven).
- Testes: Execução de Testes Unitários e Testes de Integração.
- SQA: Análise SonarQube.
- Análise de cobertura de código com JaCoCo.
- Quality Gate: Se falhar o Gate, o Pipeline bloqueia.

CD Pipeline:

- Checkout: Obter o código da branch.
- Docker-Compose: Constrói os contentores da aplicação Spring.
- Os contentores são arrancados em seguida.

3.3 Observabilidade do sistema

Para garantir a observabilidade do sistema, utilizamos o *Grafana* em conjunto com o *Pyrra*, que usamos para fazer a gestão de *SLOs*.

As *SLOs* definidas do sistema são as seguintes:

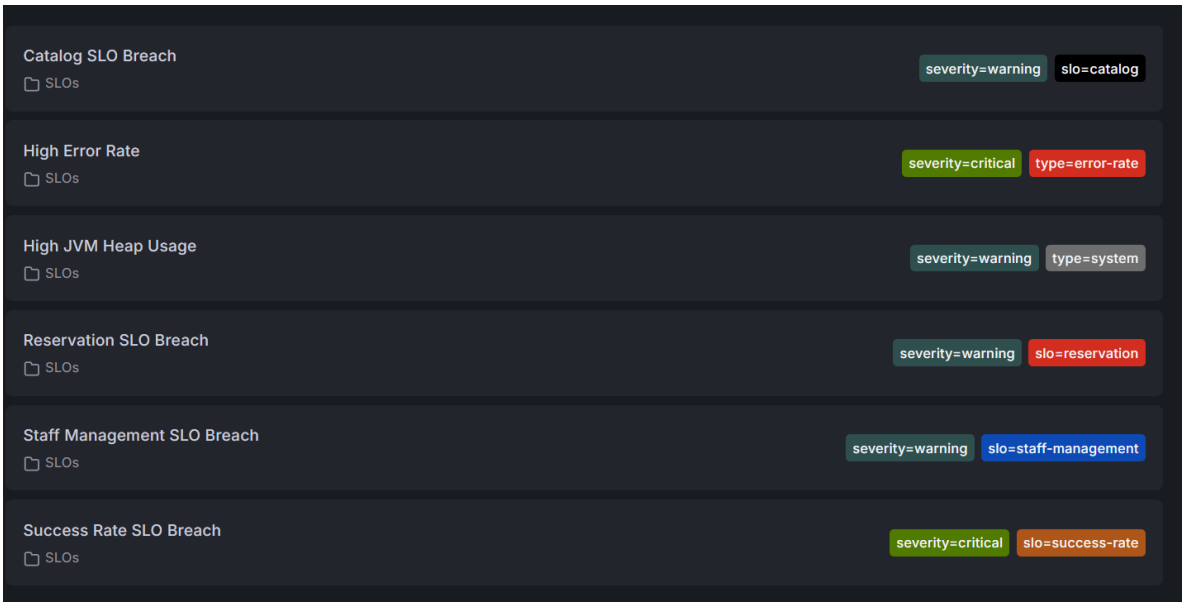
- As respostas para pedidos de reserva têm de demorar menos de 1 segundo para 98% dos pedidos, sob carga normal;

- As respostas para operações por parte do *Staff* têm de demorar menos de 2 segundos para 95% dos pedidos, sob carga normal;

- O catálogo dos itens tem de ser apresentado em menos de 800 *ms* para 95% dos pedidos, sob carga normal.

Usando o *Grafana*, visualizamos se essas *SLOs* são cumpridas através de diferentes *dashboards* adequados, como podemos ver na seguinte imagem:

Tendo também, no *Grafana*, definido alertas, caso essas mesmas *SLOs* passam a não ser cumpridas e para o caso, que o sistema apresente níveis altos de uso de memória ou número elevado de erros nos pedidos.



Além disso temos estatísticas do sistema, como o número de *threads* em uso, a carga da CPU, o uso de memória entre outros dados relevantes.

Todas as métricas obtidas são obtidas através do *Prometheus*.

4 Testes contínuos

4.1 Estratégia global de testes

Estratégia:

- Usamos a metodologia TDD que foca no desenvolvimento de testes antes da produção do código, implementando-o de forma que todos os testes passarem, abordamos a metodologia nos Testes Unitários e nos Testes de Integração, ambos são executados na CI para garantir qualidade do código.
- Para os Testes de Aceitação seguimos a abordagem BDD com Cucumber.

4.2 Testes de aceitação e ATDD

De forma a garantir que as funcionalidades atendem as expectativas do utilizador final, adotamos os testes de aceitação focando exclusivamente no comportamento observável pelo utilizador sem qualquer conhecimento da implementação do código.

Os testes devem ser claros, testáveis e escritos em linguagem acessível a todos membros da equipa.

Os testes são escritos antes do desenvolvimento do código, idealmente na parte do refinamento do *backlog* de forma que os desenvolvedores usam os critérios como guia para o desenvolvimento.

4.3 Testes orientados ao programador (unitários, integração)

Para garantir que os componentes funcionem corretamente de forma isolada. Os testes unitários, focam-se na perspetiva do desenvolvedor para verificar a lógica interna do código. Os testes são escritos para validar unidades pequenas e independentes de código, de forma a atingir uma cobertura no mínimo de 80.0%.

Os testes têm de ser escritos antes do desenvolvimento do código de forma a seguir a abordagem TDD. As partes mais relevantes para testagem são a camada de serviço e a camada dos controladores.

4.4 Testes de atributos não funcionais e de arquitetura

Garantir que o sistema consiga manter níveis aceitáveis de desempenho, escalabilidade e estabilidade sob cargas esperadas e de pico, os testes de performance identificam gargalos, validam a capacidade do sistema e suportam decisões de arquitetura e infraestrutura.

Os testes focam-se em métricas como tempo de resposta, utilização de CPU/memória, concorrência e escalabilidade. Para tal, fizemos diferentes tipos de testes de performance de forma abranger vários lados da aplicação.

Smoke Test

Realizado imediatamente após cada *deploy*.

Objetivo: Validar a saúde básica da infraestrutura. Confirma se os serviços essenciais (Login, Páginas principais, Gateway de API) estão operacionais antes de avançar para testes mais profundos.

Configuração:

- **Carga:** 3 Utilizadores Virtuais (VUs).
- **Duração:** 1 minuto.

Testes de Integridade e Concorrência

Concorrência em Reservas

Cenário de alto risco para o negócio.

Objetivo: Garantir a integridade transacional e evitar *double-booking*. O sistema deve ser capaz de bloquear ou gerir filas quando múltiplos utilizadores tentam adquirir o mesmo item nas mesmas datas simultaneamente.

Configuração:

- **Carga:** 50 VUs focados num único item.
- **Critério de Sucesso:** 0 conflitos de reserva.

Fluxo de Cancelamento

Teste de fluxo crítico sob carga.

Objetivo: Assegurar que o processo de cancelamento liberta corretamente o inventário e atualiza o estado das encomendas, mesmo quando o sistema está ocupado.

Configuração:

- **Carga:** 30 VUs constantes.
- **Duração:** 3 minutos.

Testes de Performance de Componentes

Disponibilidade de API

Otimização de Latência.

Objetivo: Monitorizar a API mais solicitada do sistema. O foco é manter a latência baixa para garantir uma boa experiência de navegação (UX).

Configuração:

- **Throughput:** 50 requisições por segundo (RPS).
- **Duração:** 3 minutos.
- **KPI Alvo:** Tempo de resposta < 500ms.

Database Stress

Otimização de *Queries*.

Objetivo: Executar buscas complexas e relatórios pesados para verificar o comportamento dos índices da base de dados e identificar queries lentas.

Configuração:

- **Carga:** 10 VUs (operações pesadas).
- **Duração:** 2.5 minutos.

Pesquisa e Filtros

Validação de Funcionalidade.

Objetivo: Avaliar a rapidez com que o sistema devolve resultados ao utilizar múltiplos filtros de produtos simultaneamente.

Configuração:

- **Carga:** Ramping (Gradual) de 20 até 50 VUs.
- **Duração:** 3.5 minutos.

Simulação de Utilização Real

Objetivo: Simular o comportamento orgânico de um utilizador real percorrendo todo o funil: Registo ➔ Navegação ➔ Reserva ➔ Logout. Testa a integração entre todos os microserviços/componentes.

Configuração:

- **Carga:** 100 VUs.
- **Duração:** 5 minutos.

General Load

Identificação de *Bottlenecks*.

Objetivo: Analisar como o sistema gere a alocação e libertação de recursos com a variação natural de tráfego.

Configuração:

- **Padrão:** Ondas de 50 ➔ 150 ➔ 50 VUs.
- **Duração:** 6 minutos.

Testes de Resiliência

Spike Test

Simulação de Eventos de Marketing/Flash Sales.

Objetivo: Validar se a infraestrutura suporta aumentos súbitos e violentos de tráfego sem entrar em colapso, e se recupera a performance normal após o pico.

Configuração:

- **Padrão:** 2 Picos agressivos (200 VUs e 150 VUs).
- **Duração:** 3 minutos.