Discentes:
Ana Beatriz Santos Silva
Hyan Vitor dos Santos Araujo
Jonatas Gomes Lima
Luan Cerqueira São Pedro
Naila Oliveira Barbosa Borges
Rebert da Silva Azevedo

Docente: Thiago Dotto Fiuza Neves

Implementação de DevOps e garantia de qualidade em Sistema de Gerenciamento de Pedidos

Salvador
2025

Implementação de DevOps e garantia de qualidade em sistema de gerenciamento de pedidos

Trabalho apresentado à unidade
curricular Gestão e Qualidade de Software,
como parte dos requisitos necessários para
obtenção de aprovação na disciplina.

Acesso ao repositório GitHub Projeto:
https://github.com/Rebert-Azevedo/A3-GQS-UNIFACS

Salvador
2025

1. Introdução

Este relatório documenta o ciclo de desenvolvimento do Sistema de Gerenciamento de Pedidos, com o foco na implementação de práticas de DevOps, Automação de Testes e Garantia de Qualidade. O objetivo principal do trabalho foi evoluir o processo de desenvolvimento de software, saindo de um modelo manual para um ambiente integrado e automatizado, garantindo a entrega contínua de valor com segurança e rastreabilidade. Foram realizados testes unitários para as classes principais do sistema, como Clientes, Produto, Pedido, Funcionario, e todas as demais, garantindo que cada uma delas funcionasse corretamente de forma isolada.

1.2 Análise e requisitos e Modelagem de Software:

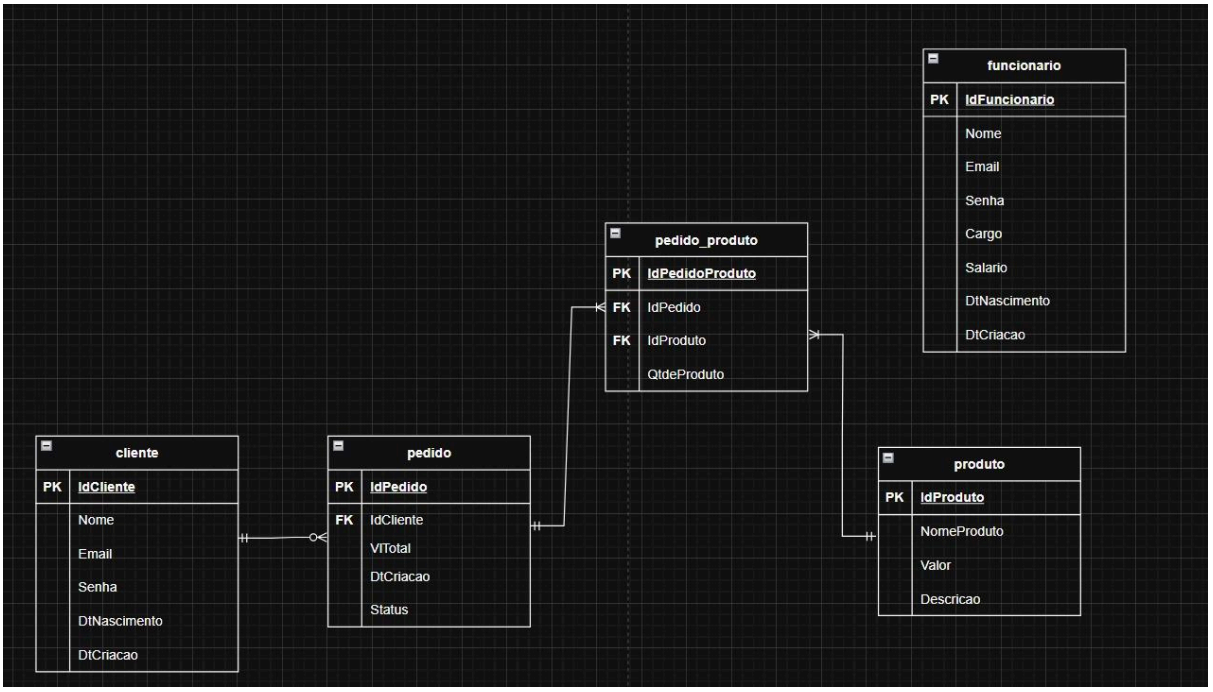
O projeto foi inicialmente concebido a partir de um alinhamento estratégico sobre sua finalidade principal, desenvolver um sistema de controle para restaurantes, abrangendo aspectos essenciais como gestão de estoque, identificação e rastreamento de pedidos, além do gerenciamento de funcionários. A partir dessa definição de escopo, avançou-se para a etapa de análise de requisitos, considerada fundamental para garantir que o sistema atenda de forma precisa às necessidades operacionais e de negócio.

Durante essa análise, foram levantados e documentados os **requisitos funcionais**, que descrevem as funcionalidades que o sistema deve obrigatoriamente oferecer. Entre eles, destacam-se o cadastro e atualização de produtos, o controle de entradas e saídas de estoque, a geração e acompanhamento de pedidos com identificação única, onde assegura que o projeto será capaz de executar as operações básicas e críticas para o funcionamento do restaurante.

Paralelamente, foram definidos os **requisitos não funcionais**, igualmente indispensáveis para a qualidade da solução. Esses requisitos tratam de aspectos como desempenho, segurança da informação, escalabilidade, usabilidade e confiabilidade. Eles garantem que o sistema não apenas funcione corretamente, mas também ofereça uma experiência consistente e segura para os usuários, suportando o crescimento do negócio e evitando falhas que possam comprometer a

operação.

Seguido da Modelagem de Software (UML), que foi realizada logo após a definição do objetivo do projeto e sinalização dos requisitos funcionais e não-funcionais:



2. Configuração do Ambiente DevOps e Integração Contínua

Para atender aos requisitos de automação e integração contínua (CI/CD), configuramos pipelines robustos utilizando o GitHub Actions. A estratégia de DevOps foi segmentada em três fluxos de trabalho distintos para cobrir diferentes momentos do ciclo de vida do código:

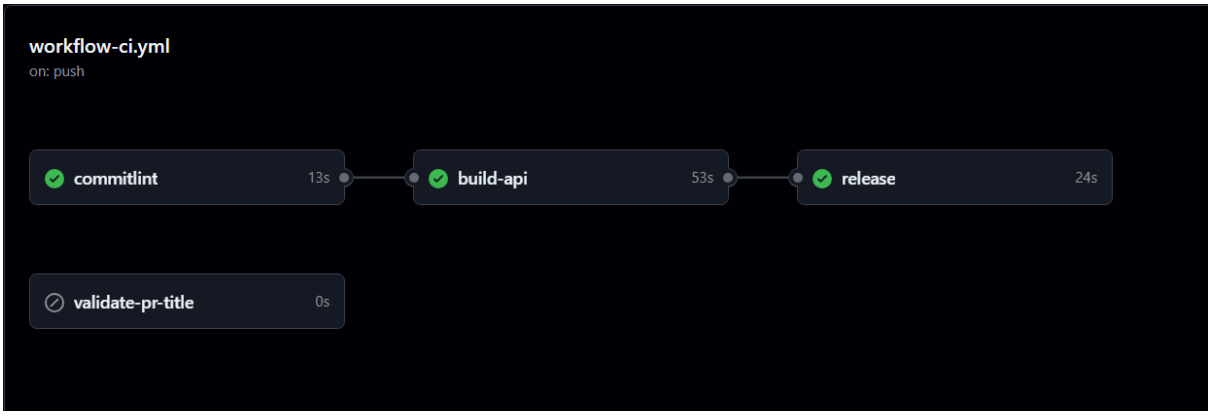
2.1. Estrutura dos Workflows

O ambiente foi configurado para reagir a diferentes eventos no repositório:

1. **Validação Rápida:** Ao realizar um *push* em branches de *feature* ou *fix*, o GitHub Actions executa o job “commitlint”. Isso garante que, desde o início, a mensagem do commit esteja padronizada, impedindo que descrições fora do padrão poluam o histórico.
2. **Pull Requests:** Quando um Pull Request é aberto para as branches main ou develop, dois processos críticos são disparados:

- **Validate PR Title:** Verifica se o título do PR segue o padrão de Conventional Commits.
- **Build-API:** Executa o build completo da aplicação e roda obrigatoriamente todos os testes unitários e de integração. Isso impede que códigos quebrados sejam mesclados nas branches principais.

3. **Lançamento e Deploy:** Após a aprovação e merge na main, o pipeline executa o job de release. Este passo foi configurado para suportar o lançamento semântico do software.



2.2. Padronização de Commits

Adotamos a especificação Conventional Commits. Embora o versionamento final das tags tenha sido realizado de forma manual para garantir controle total sobre as versões entregues, a padronização das mensagens foi automatizada e validada pelo linter.

Exemplos reais de commits do projeto:

Essa padronização facilitou a leitura do histórico e a identificação rápida do propósito de cada mudança no código.

3. Estratégia de Testes e Garantia de Qualidade

A garantia da qualidade do software foi uma prioridade, evoluindo de verificações manuais para uma suíte de testes automatizada.

3.1. Evolução da Estratégia de Testes

- **Fase Inicial:** No início do desenvolvimento, a validação das regras de negócio (como criar um pedido ou fechar uma conta) era feita manualmente pelos desenvolvedores.
- **Fase Final:** Implementamos testes unitários e de integração utilizando JUnit. O uso da biblioteca Mockito permitiu isolar camadas do sistema, simulando comportamentos de banco de dados e autenticação, garantindo que a lógica de negócio fosse testada de forma isolada e eficiente.

3.2. Cobertura de Código

Para mensurar a eficácia dos testes, integramos a ferramenta “JaCoCo” ao pipeline de build. Isso nos permitiu visualizar quais linhas de código foram exercitadas pelos testes. O relatório do JaCoCo serviu como métrica de qualidade, indicando onde novos testes eram necessários.

Essa parte do código foi a seção de configuração em um arquivo Maven geralmente pom.xml, que adiciona o JaCoCo Maven Plugin ao projeto, onde ajudou pois foi fundamental para a qualidade do software, pois ajuda a identificar partes do código

que não estão sendo testadas e que, portanto, podem ter bugs.

```
<groupId>org.jacoco</groupId>
<artifactId>jacoco-maven-plugin</artifactId>
<version>0.8.12</version>
```

Estamos configurando um goal (meta) de check no JaCoCo Maven Plugin para impor que a cobertura mínima de instrução (ou de branch, line, method, ou class) do projeto não seja inferior a 70% (0.70)."

```
<execution>
    <id>trava-de-cobertura-jacoco</id>
    <goals>
        <goal>check</goal>
    </goals>
    <configuration>
        <excludes>
            <exclude>**/dto/**</exclude>
            <exclude>**/model/**</exclude>
            <exclude>**/GqsA3Application.class</exclude>
        </excludes>
        <rules>
            <rule>
                <element>PACKAGE</element>
                <limits>
                    <limit>
                        <counter>LINE</counter>
                        <value>COVEREDRATIO</value>
                        <minimum>0.7</minimum>
                    </limit>
                </limits>
            </rule>
        </rules>
    </configuration>
</execution>
```

```
[INFO] Results:
[INFO]
[INFO] Tests run: 77, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco:0.8.12:report (report) @ A3-GQS-UNIFACS ---
[INFO] Loading execution data file C:\Users\luanc\IdeaProjects\GQS-A3\A3-GQS-UNIFACS\target\jacoco.exec
[INFO] Analyzed bundle 'A3-GQS-UNIFACS' with 28 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  42.674 s
```

O JaCoCo se encontra realizando varreduras de todo o código e executando os testes necessários.

A3-GQS-UNIFACS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.unifacs.GQS_A3.controller		81%		83%	4	30	18	90	3	27	0	6
com.unifacs.GQS_A3.service		92%		83%	11	60	6	117	3	33	0	4
com.unifacs.GQS_A3.security		83%		50%	3	18	9	57	1	16	0	4
com.unifacs.GQS_A3		37%		n/a	1	2	2	3	1	2	0	1
com.unifacs.GQS_A3.model.enums		89%		n/a	1	3	1	7	1	3	0	1
com.unifacs.GQS_A3.dto		0%		n/a	1	1	1	1	1	1	1	1
com.unifacs.GQS_A3.exceptions		100%		n/a	0	9	0	33	0	9	0	5
com.unifacs.GQS_A3.dto.users		100%		n/a	0	3	0	3	0	3	0	3
com.unifacs.GQS_A3.model		100%		100%	0	4	0	5	0	3	0	1
com.unifacs.GQS_A3.dto.auth		100%		n/a	0	2	0	2	0	2	0	2
Total	163 of 1.496	89%	12 of 66	81%	21	132	37	318	10	99	1	28

O relatório de cobertura de código foi gerado pelo JaCoCo para o projeto, exibido em formato HTML. Nele foi relatado e detalhado, pacote por pacote, a eficácia dos seus testes unitários, no qual pode ser visto a garantia da cobertura total de 89% e as coberturas individuais dos pacotes mais críticos e que não caiam abaixo desse valor em futuras alterações.

4. Análise de Processo e Melhoria Contínua

Refletindo sobre o desenvolvimento do projeto, observamos uma clara evolução na maturidade do processo de engenharia de software.

4.1. Comparativo: Início x Final

Aspecto	Início do Projeto	Final do Projeto
Testes	Execução manual e esporádica.	Automatizados via JUnit, rodando a cada PR.
Integração	Merge de código sem validação prévia.	Pipeline bloqueia merges se testes falharem.
Commits	Mensagens livres e despadronizadas.	Padronizados (feat, fix, refactor) e validados por linter.
Confiança	Incerteza sobre novos bugs.	Segurança de que as funcionalidades críticas estão cobertas.

4.2. Lições Aprendidas

A principal lição aprendida foi que a configuração inicial do ambiente DevOps, embora trabalhosa, é convertida rapidamente através da redução de retrabalho. A automação imposta pelo GitHub Actions criou uma cultura de "qualidade em primeiro lugar", onde nenhum código é integrado sem passar pelos critérios de aceite definidos nos testes.

5. Conclusão

O trabalho prático permitiu aplicar os conceitos de Gestão e Qualidade de Software. A entrega não mostra apenas um sistema de restaurante funcional, mas

um repositório configurado com práticas profissionais, pronto para um crescimento sustentável e uma manutenção segura.