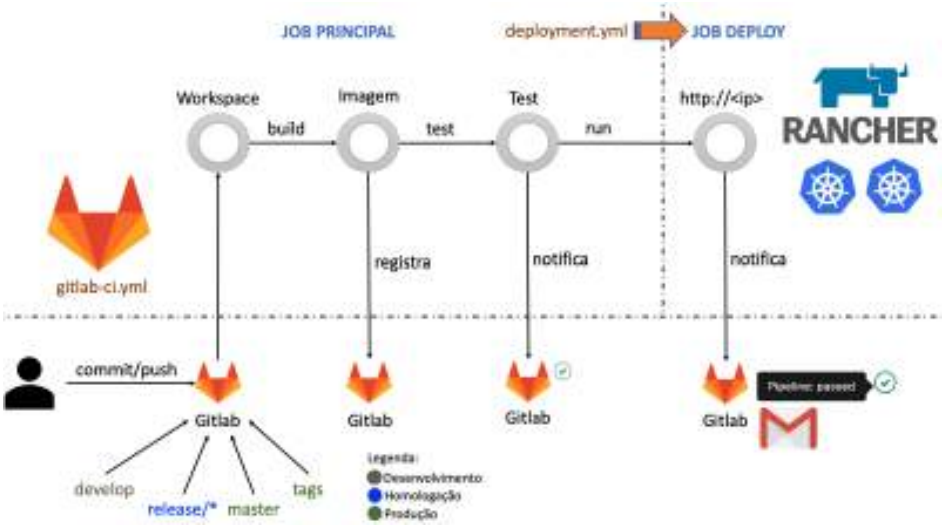
Proposta de Ambiente DevOps no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso

Secretaria de Segurança Pública do Estado de Mato Grosso
Fundação de Amparo a Pesquisa do Estado de Mato Grosso
Instituto Federal de Mato Grosso - Campus Cuiabá Cel. Octayde Jorge da Silva
Edital 003/2020
Execução entre 01/08/2020 - 28/07/2021

Intituto Federal de Mato Grosso

Proposta de Ambiente DevOps no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso



Pesquisadores:
Evandro César Freiberger
João Paulo Delgado Preti
Leonardo Luiz Braun
Ruy de Oliveira
Tiago de Almeida Lacerda
Willian Chito de Souza Pinto

Alunos:
Gabriel José Curvo Honda
Luís Fernando Vieira Sampaio

Membros Integrantes da SESP:
Willer Sondrei Oliveira Marques Silva
Lavídico Alves de Brito Junior

Gestão Operacional da SESP:
Marcos Roberto Weber Hubner
Walmir Akihiro Oribe
Diana Maria de Lima
Bruno de Oliveira Figueiredo

Intituto Federal de Mato Grosso

Dados Internacionais de Catalogação na Publicação (CIP) Câmara Brasileira do Livro, MT, Brasil

Freiberger, Evandro César; Preti, João Paulo D.; Braun, Leonardo Luiz; Lacerda, Tiago A.; Pinho, Willian Chito de Souza

Proposta de Ambiente DevOps no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso.

Cuiabá-MT: Intituto Federal de Mato Grosso, Ltda., 2021.

1. Programas de computador. 2. Engenharia de Software. 3. Integração Contínua. 4. DevOps.

Lista de ilustrações

Figura 1 – Projeto SROP no Gitlab 12
Figura 2 – Modelagem do processo de versionamento e ramificação do projeto srop . . . 13
Figura 3 – Projeto sgapd-view no Gitlab 13
Figura 4 – Modelagem do processo de versionamento e ramificação do projeto sgapd-view 14
Figura 5 – Fluxo de Branches 16
Figura 6 – Fluxo da Branche master para a develop 17
Figura 7 – Fluxo entre as branches develop e feature 18
Figura 8 – Fluxo da branche release 19
Figura 9 – Fluxo da branche hotfix 20
Figura 10 – Exemplo Git Flow 29
Figura 11 – Processo de Build Atual 55
Figura 12 – Metáfora do **Botão de Integração** 57
Figura 13 – Comparação de ferramentas de integração contínua 59
Figura 14 – Branches e o processo de Build 61
Figura 15 – Exemplos de imagens geradas 62
Figura 16 – Exemplos de pacotes gerados 62
Figura 17 – Runnners 64
Figura 18 – Stages 67
Figura 19 – Gestão de Implantação 70
Figura 20 – Dashboard principal 71
Figura 21 – Instalação do Kubernetes 72
Figura 22 – Instalação do Kubernetes, continuação... 72
Figura 23 – Configurações na tela inicial do Rancher 73
Figura 24 – Adicionar Certificado 74
Figura 25 – Load Balancing 74
Figura 26 – configuração do Ingress 75
Figura 27 – Vinculação do Certificado 75
Figura 28 – Configurando Gitlab e Kubernetes 75
Figura 29 – Configurando vínculo com cluster 76
Figura 30 – Configurando vínculo com cluster 77
Figura 31 – Arquivo Kubeconfig do Cluster 77
Figura 32 – Variáveis de ambiente 79
Figura 33 – Modelo de publicação do ambiente 83

Figura 34 – Ambiente de contêineres 83
Figura 35 – Arquitetura da infraestrutura do ambiente 84
Figura 36 – Pipeline Proposta para o Ambiente SESP 85
Figura 37 – Configurando ambientes de trabalho 86
Figura 38 – Ambiente de desenvolvimento 92
Figura 39 – Histórico de commits realizados na versão “v0.5.0” 92
Figura 40 – Versão candidata do software 93
Figura 41 – Migrations 93
Figura 42 – Representação das atualizações do banco de dados 94
Figura 43 – Migration pendente 95
Figura 44 – Refatoração de tabelas 95
Figura 45 – Convenção de nome para migrations 96
Figura 46 – Modelos de versionamento 97
Figura 47 – Execução das migrations na pipeline 99
Figura 48 – Processo de Rollback 99
Figura 49 – Pirâmide de Testes 102
Figura 50 – Pirâmide de Testes na Prática 103
Figura 51 – Pipelines de compilação 104
Figura 52 – Pirâmide de Testes de Mike Cohn 105
Figura 53 – Conjunto de testes de casquinha de sorvete 106
Figura 54 – Estrutura de alto nível do sistema de microsserviço 107
Figura 55 – Estrutura interna do nosso microsserviço 107
Figura 56 – Testing Manifesto 110
Figura 57 – Manifesto dos Testes 111
Figura 58 – Destaque das Tarefas de Testes em um ciclo Ágil 112
Figura 59 – Processo de Testes para a SESP/STI 114
Figura 60 – Arquitetura Base dos Projetos de Microsserviços 115
Figura 61 – Exemplo 1 de User Stories 119
Figura 62 – Exemplo 2 de User Stories 120
Figura 63 – Exemplo de nomenclatura de testes 134
Figura 64 – Sufixo para testes Unitários e Integração com BD 135
Figura 65 – Exemplo ampliado de nomenclatura de casos de testes 135
Figura 66 – Rastreabilidade de falhas nos testes 135
Figura 67 – Exemplo de nomenclatura recomendada 138
Figura 68 – Exemplo de Relatório da ferramenta EclEmma 144

Lista de tabelas

Tabela 1 – Matriz de Decisão da Ferramenta de Versionamento 28
Tabela 2 – Serviços do Ambiente 81
Tabela 3 – Registros do histórico das migrations 94
Tabela 4 – Registros do histórico das migrations 95
Tabela 5 – Interface REST da Aplicação Exemplo 107

Sumário

1 Cenário . . . 9
1.1 Premissas . . . 9
1.2 Visão do Produto . . . 10
2 Processo de Ramificações e Versionamento . . . 11
2.1 Processo atual de versionamento . . . 11
2.2 Ramificações (Branches) . . . 12
2.2.1 Branch master . . . 12
2.2.2 Branch develop . . . 13
2.2.3 Branch feature/* . . . 14
2.2.4 Branch release/* . . . 15
2.2.5 Branch hotfix/* . . . 15
2.3 Descrição do Fluxo . . . 16
2.3.1 Branch master e develop . . . 17
2.3.2 Branch feature . . . 17
2.3.3 Branch Release . . . 18
2.3.4 Brach Hotfix . . . 19
2.4 Padrões de Commits . . . 20
2.4.1 Estrutura . . . 21
2.4.2 Tipos . . . 21
2.4.3 Detalhando commit . . . 22
2.4.4 Exemplos . . . 22
2.4.4.1 Periodicidade e Granularidade dos Commits . . . 23
2.4.4.2 Granularidade . . . 23
2.4.4.3 Periodicidade . . . 24
2.4.5 Estratégias de Alteração de Commits . . . 24
2.4.5.1 Alterações da mensagem de commits . . . 24
2.4.5.2 Alterações do conteúdo de commits . . . 24
2.4.5.3 Considerações sobre alterações de commits . . . 25
2.5 Padrão de Nomenclatura de Versões . . . 25
2.5.1 Níveis . . . 25
2.5.2 Desenvolvimento Inicial . . . 27
2.6 Análise da Ferramenta de Versionamento . . . 27
2.6.1 Critérios de escolha adotados para a ferramenta de versionamento . . . 27
2.6.2 Matriz de Decisão . . . 27
2.6.3 Considerações sobre a escolha . . . 27
2.6.4 Ferramenta de Automação de Fluxo e Ramificações . . . 28
2.7 Exemplo prático com Git flow . . . 28
2.7.1 Instruções . . . 28

2.7.1.1 Configuração do projeto: . . . 28
2.7.1.2 Desenvolvimento do projeto: . . . 29
3 Processo de Build e Gestão de Implantação . . . 45
3.1 Cenário Atual do Processo de Build na SESP . . . 45
3.1.1 Tecnologias e Ambiente . . . 45
3.1.2 Soluções de Automatização Atual . . . 47
3.2 Processo de Build e Gestão da Implantação . . . 55
3.2.1 Conceitos importantes . . . 55
3.2.1.1 Práticas . . . 56
3.2.1.2 Como fazer . . . 56
3.2.1.3 Responsabilidades do time . . . 57
3.2.2 A metáfora do Botão de Integração . . . 57
3.2.3 Ambiente de Integração contínua . . . 58
3.2.3.1 Tamanho e complexidade do ambiente . . . 58
3.2.3.2 Integrações . . . 58
3.2.3.3 Comparação de ferramentas de integração contínua . . . 59
3.2.4 Modelo de build e armazenamento dos artefatos . . . 60
3.2.4.1 Stages . . . 60
3.2.4.2 Branches e o processo de Build . . . 60
3.2.4.3 Nomenclatura de imagens Docker . . . 61
3.2.4.4 Nomenclatura de pacotes . . . 61
3.3 Configurando o processo de Build no Gitlab . . . 63
3.3.1 GitLab CI / CD . . . 63
3.3.2 Primeiros passos . . . 63
3.3.3 Configurando . . . 63
3.3.4 Instale o GitLab Runner . . . 64
3.3.5 Registrando o Runner . . . 64
3.3.6 Stages . . . 67
3.3.7 Referências . . . 69
3.4 Processo de Gestão da Implantação . . . 70
3.4.1 Configurando Rancher e Kubernetes . . . 70
3.4.1.1 Configuração do Cluster Kubernetes . . . 71
3.4.1.2 Certificados . . . 73
3.4.1.3 Load balancer e Ingress . . . 74
3.4.1.4 Configurando Gitlab . . . 74
3.4.1.5 Pipeline de Implantação . . . 76
3.4.1.6 Configuração de Variáveis . . . 79
4 Ambiente de Infraestrutura . . . 81
4.1 Ambiente inicial . . . 81
4.1.1 Máquinas virtuais do projeto . . . 81
4.1.2 Ambiente de contêineres . . . 82
4.1.3 Modelo de publicação do ambiente . . . 82
4.1.4 Ambiente de contêineres . . . 83

4.2 Instalação do Ambiente . . . 84
4.2.1 Gitlab . . . 84
4.2.1.1 Instalação Omnibus . . . 85
4.2.2 Configurações da Pipeline . . . 85
4.2.3 Configurando ambientes de trabalho . . . 85
5 Gestão de Versões de Banco de Dados . . . 91
5.1 Introdução . . . 91
5.2 Migrations . . . 93
5.3 Ferramenta Flyway . . . 96
5.4 Estratégia de versionamento . . . 96
5.4.1 Projeto de Monolito . . . 96
5.4.2 Projeto de Microsserviço . . . 97
5.4.3 Projeto de Microsserviço com compartilhamento de banco de dados com tabelas exclusivas . . . 97
5.4.4 Projeto de Microsserviço com compartilhamento de banco de dados e tabelas não exclusivas . . . 97
5.5 Padrão de Commits e gerenciamento do código . . . 98
5.6 PIPELINE . . . 98
5.7 Referências . . . 100
6 Processo de Testes de Software Proposto . . . 101
6.1 Fundamentação de Testes . . . 101
6.1.1 Pirâmide de Testes . . . 101
6.1.1.1 Notas . . . 103
6.1.2 A pirâmide de testes na prática . . . 103
6.1.2.1 A importância dos testes automatizados . . . 104
6.1.2.2 A Pirâmide de Testes . . . 105
6.1.2.3 Aplicação de exemplo . . . 106
6.1.2.4 Funcionalidades . . . 107
6.1.2.5 Estrutura de alto-nível . . . 107
6.1.2.6 Arquitetura interna . . . 107
6.1.3 O estilo GivenWhenThen . . . 108
6.1.3.1 Notas . . . 109
6.1.4 Manifesto do Teste Ágil, Princípios e Práticas . . . 110
6.1.5 Referência . . . 111
6.2 Processos de Testes SESP/STI . . . 112
6.2.1 Visão Geral do Processo . . . 112
6.2.2 Detalhamento do Processo . . . 113
6.2.2.1 Planejamento de Testes Unitários . . . 115
6.2.2.2 Planejamento de Testes de Integração . . . 116
6.2.3 Documentação Base de Requisitos para o Processo de Testes . . . 116
6.2.3.1 História de Usuário (User Story) . . . 117
6.2.3.2 Padrão INVEST para Construção de User Story . . . 118
6.2.3.3 Critérios de Aceitação . . . 119

6.2.3.4 Desenvolvimento de Planos de Testes . . . 120
6.2.4 Plano de Testes Unitários . . . 120
6.2.4.1 Microsserviço Registro Geral . . . 120
6.2.4.2 Microsserviço Carteira Funcional . . . 121
6.2.4.3 Microsserviço Gerenciamento de Ocorrências . . . 124
6.2.5 Plano de Testes de Integração com o Banco de Dados . . . 127
6.2.5.1 Microsserviço Registro Geral . . . 127
6.2.5.2 Microsserviço Carteira Funcional . . . 127
6.2.5.3 Microsserviço: Gerenciamento de Ocorrências . . . 130
6.3 Análise e Definição de Ferramentas para Automação de Testes . . . 132
6.3.1 Ferramentas e bibliotecas de testes unitários . . . 132
6.4 Convenções no desenvolvimento de testes unitários e testes de integração . . . 133
6.4.1 Nomenclatura das classes de teste . . . 134
6.4.2 Nomenclatura de membros da classe . . . 136
6.4.3 Nomenclatura de variáveis locais . . . 136
6.4.4 Nomenclatura de métodos . . . 137
6.4.4.1 Nome do método, Estado sob teste e Comportamento Esperado 137
6.4.4.2 DEVERIA ter o comportamento esperado QUANDO determinado estado sob teste acontecer . . . 137
6.5 Implementação de testes nos microsserviços . . . 138
6.5.1 Testes Unitário . . . 138
6.5.2 Teste de Integração com o Banco de Dados . . . 140
6.6 Definição de ferramenta de análise de cobertura de testes . . . 142
6.6.1 Ferramentas consideradas para análise . . . 142
6.6.1.1 Cobertura . . . 142
6.6.1.2 EclEmma . . . 142
6.6.1.3 OpenClover . . . 142
6.6.1.4 Visual Studio Enterprise (plugin Code Coverage) . . . 143
6.6.2 Critérios de escolha adotados para a ferramenta . . . 143
6.6.3 Matriz de decisão . . . 143
6.6.4 Considerações sobre a escolha . . . 143
6.7 Estratégias de Geração de Massas de Dados para Testes . . . 144
6.7.1 Geração de Massa de Dados para Testes Unitários . . . 145
6.7.2 Geração de Massa de Dados para Testes de Integração . . . 145
6.7.2.1 Massa de Dados para Testes de Integração com Mecanismo de Persistência . . . 145

CAPÍTULO 1

Cenário

Projeto tem por objetivo entregar como produto um conjunto de métodos e técnicas de desenvolvimento de software que possibilita a implementação dos conceitos de Integração Contínua no desenvolvimento de software da Secretaria de Segurança Pública do Estado de Mato Grosso.

A característica principal do movimento de Integração Contínua é defender fortemente a automação e monitoramento em todas as fases da construção do software, da integração, teste e liberação para implantação. A Integração Contínua pretende fornecer, em ciclos de desenvolvimento menores, liberações mais frequentes de produtos, com maior segura e confiabilidade, em alinhamento próximo com os objetivos de negócio.

1.1 Premissas

- Ser um ambiente operacional validado com um protótipo funcional;
- Utilização de conceitos de DevOps;
- Promover ambiente/modelo automatizado de testes, build e deploy;
- Sincronizar com nosso sistema de controle de versões (gitlab);
- Ser prático e padronizado;
- Baixo custo de implantação e manutenção;
- Permita replicação em outras aplicações;
- Permitir a integração com os demais sistemas da SESP;
- Permitir a extração de dados, análise e estatística;
- Possuir características funcionais adequadas às especificidades da atuação a SESP.

1.2 Visão do Produto

Para Gerência de Sistemas da SESP, que necessita de ambiente de desenvolvimento de software ágil e gerenciado, a Integração Contínua é um projeto para agilizar as entregas de software que através de automação facilita a evolução de forma rastreável e aumenta a produtividade no processo de novas entregas. Ao contrário da abordagem tradicional com entregas longas e alto risco de atendimento às expectativas dos clientes, nosso projeto possui uma única plataforma de compartilhamento de conhecimento obtendo feedback rápido para validar e lançar novas atualizações de software.

CAPÍTULO 2

Processo de Ramificações e Versionamento

Este capítulo descreve os níveis de ramificações, o versionamento de código e o processo de condução de artefatos de um projeto de desenvolvimento considerando as ramificações e os padrões de versionamento adotado. O processo aqui proposto é baseado no fluxo do Git Flow.

Também é apresentada a análise das ferramentas de versionamento e as justificativas da escolha adotada. Por fim é demonstrado um exemplo prático do uso do fluxo Git Flow.

2.1 Processo atual de versionamento

Esse documento descreve um breve entendimento do modelo atual de versionamento e ramificação dos projetos na SESP-MT. Existem duas equipes de desenvolvimento, uma interna composta pelos servidores da SESP e outra externa composta por terceirizados da fábrica de software contratada, que por vezes trabalham em projetos em comum.

Não existe uma convenção ou documentação de referência para o processo de versionamento e ramificação, comprometendo inclusive o histórico de evolução dos projetos.

A SESP utiliza o Gitlab hospedado nas suas dependências para fazer o versionamento gerenciamento do código fonte de seus projetos. O git e o git flow são utilizados para versionar o código e gerenciar as branches. Ambos são integrados a IDE ECLIPSE (plugin).

Os desenvolvedores sempre baixam da branch master ou dev e procuram utilizar o git flow para gerenciar as branches, mas a utilização efetiva varia entre eles. Para cada caso de uso é gerado uma branch e essas são homologadas separadamente ou conforme necessidade. Antes de colocar em produção é gerada uma TAG da versão.

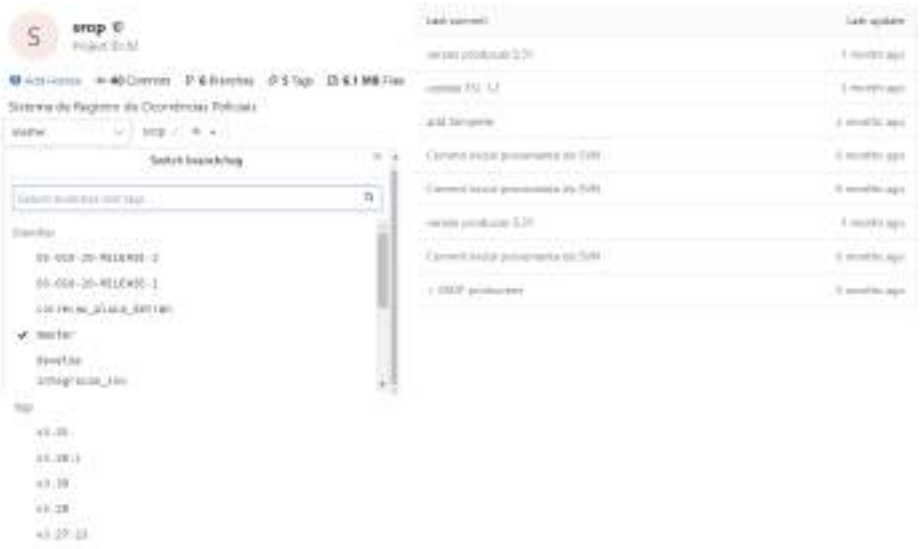
Os commits realizados pelos desenvolvedores também não seguem uma convenção pela equipe. O desenvolvedor comentou de casos onde os commits não possuem nome ou são enviados apenas com um ponto. Além disso já houve casos de commits serem realizados diretamente na branch master.

Outro fator importante apontado pelo desenvolvedor foi a necessidade de controle efetivo do que está sendo commitado pelos desenvolvedores. Atualmente os desenvolvedores não fazem solicitação de merge (Pull request) e não está implementado a análise e aceitação do código (Code review). Já houve caso onde uma correção de bug gerou outro bug e foi liberado para produção.

As imagens a seguir apresentam informações de alguns projetos da SESP. Para fins de exemplificação, utilizou-se os projetos srop e sgapd-view. É possível visualizar que algumas branches não seguem o padrão de ramificações do git flow (integração_sso, correção_placa_detran, etc), o mesmo ocorre com os commits (update TSL 1.2, add template, + SROP printscreen, etc). Já no projeto sgapd-view existem as branches feature/... e release/... o que demonstra a variação de uso do git flow entre os projetos citados pela equipe de desenvolvedores. Essa variação ocorre porque nem todos os desenvolvedores iniciam o git flow ao abrir uma branch.

As modelagens são meramente ilustrativas para fins de análise do processo, tendo seu conteúdo validado junto a um desenvolvedor no dia 14/09/2020.

Figura 1 – Projeto SROP no Gitlab



2.2 Ramificações (Branches)

O processo de ramificações prevê a existência de cinco níveis possíveis, alguns obrigatórios e outros opcionais, segue resumo das branches:

2.2.1 Branch master

Representa o principal ramo e contém o código mais estável do projeto, ou seja, o código que está em produção.

Justificativa: necessidade de ter uma branch permanente que contém o código estável do projeto.

Tipo: Principal

Tempo de vida: infinito

Figura 2 – Modelagem do processo de versionamento e ramificação do projeto srop

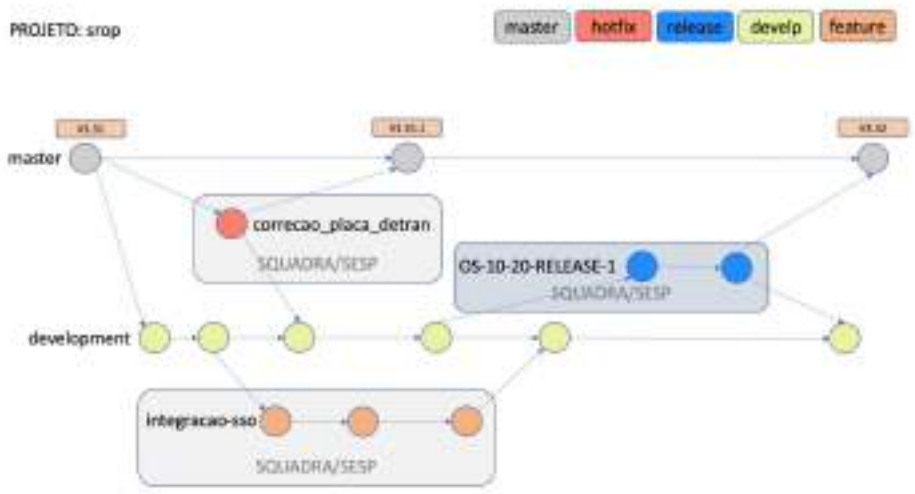
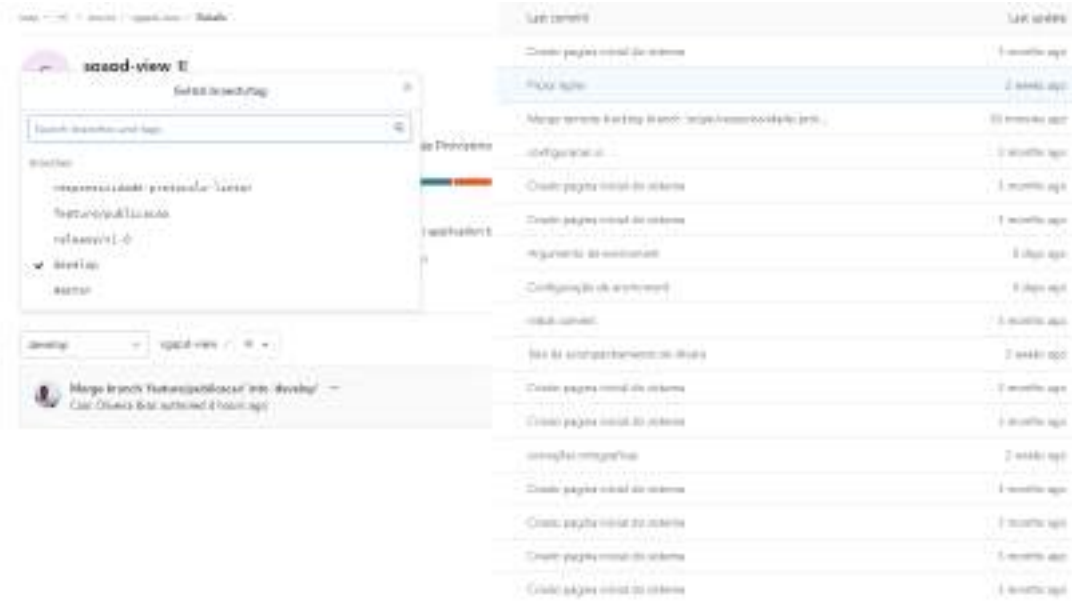


Figura 3 – Projeto sgapd-view no Gitlab



Criado de: N/A

Pode ir para: develop

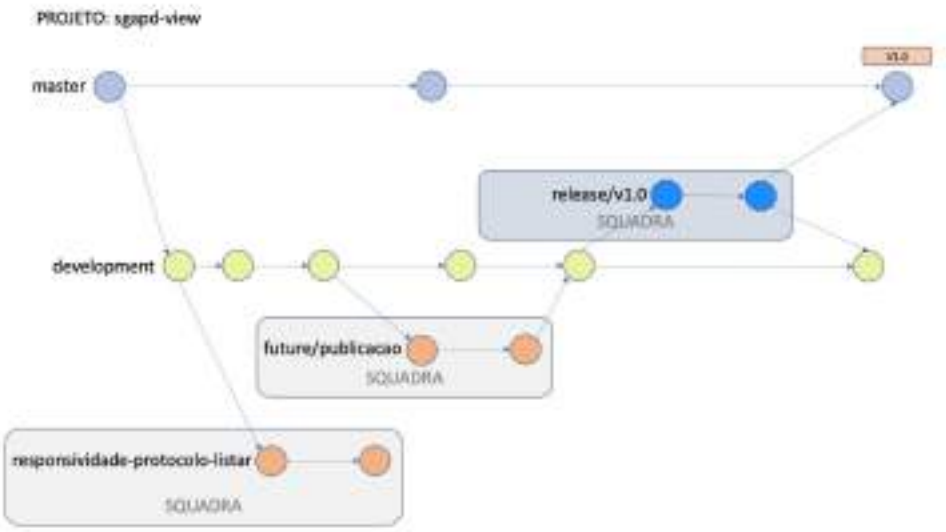
2.2.2 Branch develop

Contém código que irá gerar a próxima versão de implantação. Nessa branch que ocorrem as integrações das linhas de desenvolvimento que estão ocorrendo em paralelo.

Justificativa: permite reunir um conjunto de novas funcionalidades, integrá-las, testá-las e somente depois disso encaminhar essa nova versão para a branch master, com ações intermediárias.

Tipo: Principal

Figura 4 – Modelagem do processo de versionamento e ramificação do projeto sgapd-view



Tempo de vida: infinito

Criado de: master (na primeira vez)

Pode ir para: master

2.2.3 Branch feature/*

É uma branch criada para abrigar o desenvolvimento de uma nova funcionalidade, com tempo de vida correspondente ao desenvolvimento da funcionalidade que a origina. Pode não ser levada ao repositório remoto (opcional). Pode ser cancelada antes de ser incorporada ao branch develop. Tem o objetivo de ser uma ramificação com tempo de vida curto, para evitar que a branch develop fique desatualizada.

Justificativa: permite o isolamento da produção de uma nova característica sem interferir na branch develop. Os testes unitários podem ser realizados no código que está nesta branch, só depois de passar por esses testes, o código será encaminhado para a branch develop, onde testes de integração serão realizados.

Tipo: Suporte

Tempo de vida: finito

Criado de: develop

Pode ir para: develop

Padrão de nomenclatura: feature/nome_que_expressa_nova_funcionalidade

2.2.4 Branch release/*

Permite ajustes e correções finas e prepara os metadados (número de versão, observações, informações complementares, etc) na versão que irá para a produção.

Justificativa: Quando uma versão de código sai da branch develop para a branch release, libera a branch develop para receber código de branches features em evolução que irão compor versões futuras.

Tipo: Suporte

Tempo de vida: finito

Criado de: develop

Pode ir para: develop e master

Padrão de nomenclatura: release/número_próxima_versão

2.2.5 Branch hotfix/*

Correção de erros identificados no código que está em produção (ambiente de produção).

Justificativa: Isola correções de erros que estão em produção, sem interferir na branch develop, até que os erros sejam resolvidos e também permite que as correções sejam realizadas de forma rápida, sem a necessidade de resolução de possíveis conflitos.

Tipo: Suporte

Tempo de vida: finito

Criado de: master

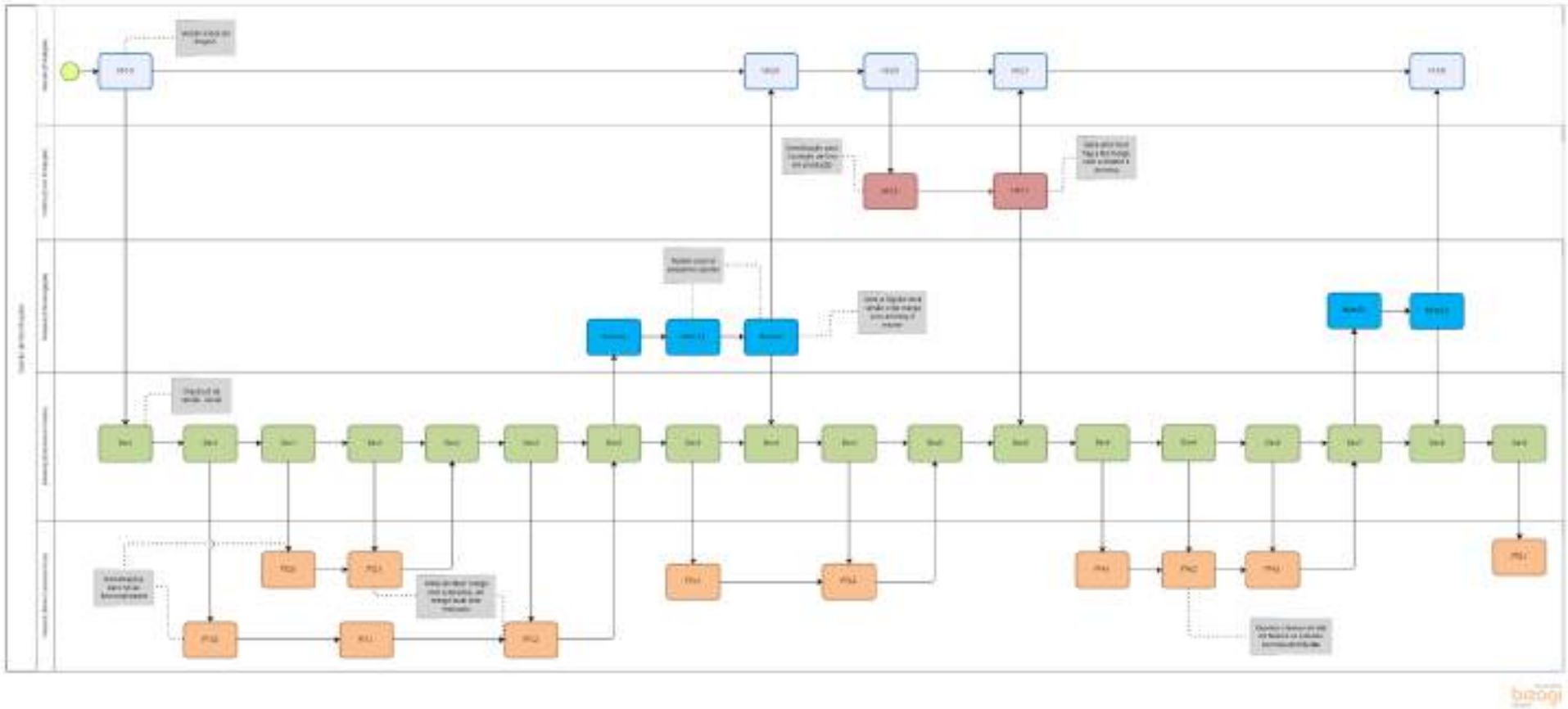
Pode ir para: master e develop

Padrão de nomenclatura: hotfix/sub-número_próxima_versão

2.3 Descrição do Fluxo

Neste modelo proposto, duas branches são obrigatórias: master e develop. As branches feature, release e hotfix são opcionais e serão criadas em situações que dão suporte às ações nas branches master e develop. A figura 5 esboça situações possíveis do fluxo.

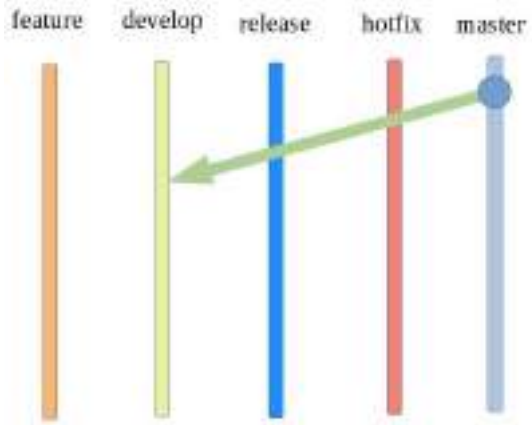
Figura 5 – Fluxo de Branches



2.3.1 Branch master e develop

Neste fluxo, ao invés de uma única branch principal, existem duas ramificações para registrar o histórico do projeto. A branch master armazena o histórico dos lançamentos de novas verões, e a ramificação de develop serve como uma ramificação de integração para novas funcionalidades ou características, conforme ilustrado na figura 6. A marcação de novas versões é realizada na branch master com um número de versão, seguindo a convenção de versionamento semântico.

Figura 6 – Fluxo da Branche master para a develop



O primeiro passo do fluxo é criar uma brach com o nome master, caso o SCM não tenha criado de forma automática. A estrutura inicial do projeto a ser desenvolvido será criada na branch master.

Com o projeto criado na branch master, o segundo passo do fluxo é criar uma branch develop e executar um checkout tendo como origem a branch master.

2.3.2 Branch feature

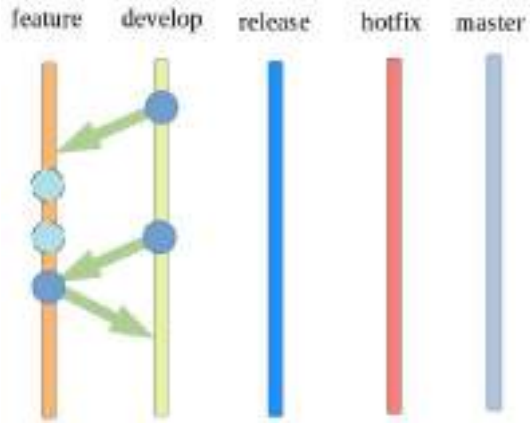
Cada novo recurso ou característica a ser desenvolvida, deverá gerar sua própria ramificação (feature/nome_feature), que tem como origem a ramificação develop. Quando a implementação é concluída, ela é mesclada de volta na ramificação develop, isso é ilustrado na figura 7. As ramificações feature não teriam necessidade de atualizarem o repositório central. Contudo, é conveniente enviar a produção local para o repositório remoto, como garantia de backup ou para permitir a colaboração com os demais desenvolvedores do projeto.

O nome da ramificação feature deve representar de forma geral o recurso/característica que irá agregar ou alterar do projeto. O nome sempre terá o prefixo feature/, seguido do nome do recurso ou característica, por exemplo: feature/crud-cidadao, feature/relatorio-ocorrencias-bairro.

Na branch feature ocorrerão a maioria dos commits do projeto e estes devem ter uma periodicidade regular com recomendação mínima de 1 (uma) vez ao dia, ver em Padrão de Commits . Isso evita que os demais colaboradores da ramificação fiquem sem acesso à implementação conduzida por um implementador.

O trabalho desempenhado em cada branch feature sempre será executado por apenas um desenvolvedor. Essa medida complementa o requisito de atomicidade de funcionalidade e evita que a produção de um desenvolvedor interfira na produção dos demais.

Figura 7 – Fluxo entre as branches develop e feature



Outro procedimento importante é que antes do conteúdo da branch feature retornar para develop, deve ser realizada uma operação de merge na branch feature com o conteúdo da develop. Essa ação minimiza conflitos que ocorreriam na branch develop, se o conteúdo da feature fosse enviado diretamente, visto que, os conflitos serão resolvidos primeiro na feature de origem, para depois ser enviado para a develop.

Depois que uma branch feature for encaminhada para a develop (via merge), esta poderá ser excluída. É conveniente usar a opção **–no-ff** na ação do merge. A opção **–no-ff** indica ao git que não é para realizar uma mesclagem fast forward, e que um novo objeto commit sempre será criado . Isso pode ser desejável se você quiser que o git mantenha um histórico de ramificações de features.

Considerando o fluxo ideal, o tempo de vida de uma feature não deveria se estender por mais de um ou dois dias. Contudo, em alguns casos isso pode se estender por um prazo maior. Nestes casos a recomendação é que a cada três dias de vida de uma feature seja realizada uma operação de merge da branch develop para a feature em questão. Essa medida irá atualizar o progresso dos outros desenvolvedores e evitará que a feature em questão não se distancie do desenvolvimento da equipe como um todo.

2.3.3 Branch Release

As branches desse tipo serão criadas quando uma versão do código na branch develop atingir o nível desejado para uma nova versão de implantação.

A criação de uma branch release se justifica por três motivos:

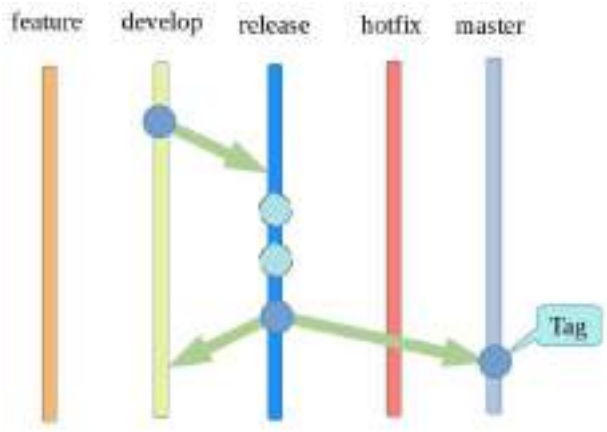
1. liberar a branch develop para receber atualizações vindo das features, com códigos que irão compor versões futuras do software, mas que não devem pertencer à próxima versão em curso;

2. a branch release é caracterizada por conter um nível de confiança maior do que a branch develop, e que se encontram em nível de preparação para ser juntada com a branch master, mas que ainda estão em sofrendo algum tipo de teste ou avaliação;

3. a branch release pode ser usada para a realização de configurações da nova versão (número de versão, informações de log, etc).

Os nomes dessas branches começam com release/ e termina com o número da próxima versão do software, seguindo as regras do versionamento semântico, por exemplo: release/2.3.0. A figura 8 ilustra esse fluxo.

Figura 8 – Fluxo da branche release



Quando o código de uma branch release atinge o nível de produção, este será enviado para a branch master. Caso tenha ocorrido alguma correção ou ajuste no código enquanto ele estava na branch release, este deve ser enviado também para a branch develop, para que as mudanças sejam incorporadas no código das próximas versões em desenvolvimento.

O código que chega na branch master a partir de uma branch release, deve receber uma tag, com o mesmo número contido no sulfixo da release de origem.

Considerando os motivos que levam a criação de uma branch release, ela pode ser considerada opcional, quando não há a necessidade de liberar a branch develop para receber códigos de features que irão compor versões futuras ou quando o código da develop já esteja totalmente maduro para ser encaminhado para a produção. Nesses casos então o código da develop poderá ser enviado diretamente para a branch master. A tag será gerada a partir da última versão em produção, seguindo o padrão de versionamento semântico.

Depois que uma branch release for encaminhada para a master (via merge), esta poderá ser excluída. Vale a mesma observação feita sobre a opção **–no-ff** do merge, descrita na brach feature.

2.3.4 Brach Hotfix

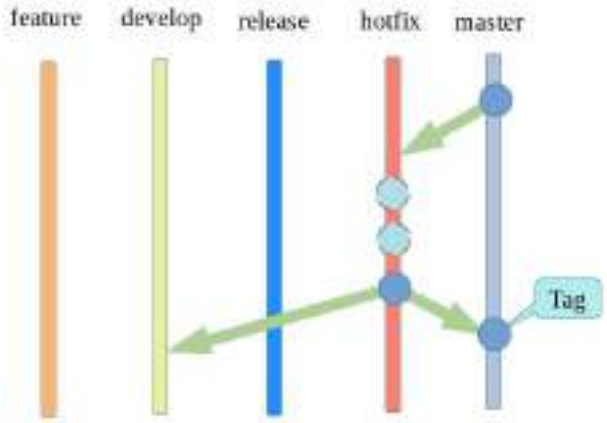
É usada para a realização de correções de bugs críticos encontrados no ambiente de produção, o que justifica sua criação a partir da branch master. Após as correções, o código deve voltar para a branch master e atualizar também a branch develop, visto que as próximas versões que estão em desenvolvimento devem conter as correções realizadas na hotfix.

A branch hotfix não pode ser usada para a inclusão de uma nova funcionalidade ou característica ao software, mas sim corrigir algo já existente e que impede ou não atende a versão que está em produção. Isso justifica o fato que de o tempo de vida de uma branch hotfix deve ser curto, como horas ou no máximo um dia.

Os nomes dessas branches começam com hotfix/ e terminam com o próximo sub-número de versão, por exemplo, se a versão atual em produção é 2.3.0, a branch deve se criada com o nome

hotfix/2.3.1), seguindo as regras do padrão de versionamento semântico. A figura 9 ilustra esse fluxo.

Figura 9 – Fluxo da branche hotfix



Quando o código retornar para a branch master, será gerada uma tag com o mesmo número do sulfixo da branch hotfix de origem.

Depois que uma branch hotfix for encaminhada para a master (via merge), esta poderá ser excluída. Vale a mesma observação feita sobre a opção **–no-ff** do merge, descrita na brach feature.

2.4 Padrões de Commits

Esta seção contém boas práticas para realização de commits nos projetos da SESP. As recomendações aqui contidas são uma adaptação da especificação do Conventional Commits e manter este nível de organização e coordenação no versionamento do código ajuda a:

- Melhorar a legibilidade do histórico de versionamento do Git;
- Aumentar a velocidade na hora de procurar por mudanças específicas no código;
- Entender, facilmente, quais mudanças estão sendo publicadas dentro de um pipeline;
- Possibilitar a geração de um CHANGELOG ou de release notes de maneira totalmente automatizada;
- Incentivar os desenvolvedores a realizarem commits de maneira pensada e específica, sem realizar commits grandes e cheios de mudanças;
- Possibilita a utilização de ferramentas de automação de commits.

Este documento define um conjunto de regras para criar um histórico de commit explícito, o que facilita a criação de ferramentas automatizadas. Esta convenção segue o Padrão de Nomenclatura de Versões, descrevendo os recursos, correções e modificações que quebram a compatibilidade nas mensagens de commit.

2.4.1 Estrutura

A mensagem do commit deve ser estruturada da seguinte forma:

```
[tipo][escopo opcional]: [descrição]

[corpo opcional]

[rodapé opcional]
```

O commit contém os seguintes elementos estruturais, para comunicar a intenção ao utilizador da sua biblioteca.

2.4.2 Tipos

Os tipos são a descrição inicial de o que o commit está realizando, sendo obrigatórios e devendo seguir um padrão bem definido.

1. **fix**: um commit do tipo fix soluciona um problema na sua base de código (isso se correlaciona com PATCH do versionamento semântico).

2. **feat**: um commit do tipo feat inclui um novo recurso na sua base de código (isso se correlaciona com MINOR do versionamento semântico).

3. **BREAKING CHANGE**: um commit que contém o texto BREAKING CHANGE:, no começo do texto do corpo opcional ou do rodapé opcional, inclui uma modificação que quebra a compatibilidade da API (isso se correlaciona com MAJOR do versionamento semântico). Uma BREAKING CHANGE pode fazer parte de commits de qualquer tipo.

4. **db**: um commit do tipo db é destinado a commits relacionados as Migrations utilizadas para versionar o banco de dados juntamente com a versão do software. Esse recurso foi utilizado durante o projeto como uma forma de iniciar o versionamento do banco de dados e melhorar a rastreabilidade das alterações realizadas.

5. **Outros**: tipos adicionais são permitidos além de **fix**: e **feat**:, recomenda-se **draft**:, **chore**:, **docs**:, **style**:, **refactor**:, **perf** :, **test**:, entre outros.

Também recomendamos *improvement* para *commits* que melhoram uma implementação atual sem adicionar um novo recurso ou consertar um bug. Observe que esses tipos não são obrigatórios pela especificação do **Conventional Commits** e não têm efeito implícito no versionamento semântico (a menos que incluam uma BREAKING CHANGE). Um escopo pode ser adicionado ao tipo do commit, para fornecer informações contextuais adicionais e está contido entre parênteses, por exemplo feat(parser): adiciona capacidade de interpretar arrays.

2.4.3 Detalhando commit

Nesta seção são detalhados os elementos que compoõem o padrão de documentação dos commits.

Escopo

O escopo do commit é uma parte opcional, curta e de fácil compreensão. É nela que iremos dizer qual parte do código foi modificada, como indicar que fizemos alterações em uma determinada funcionalidade.

Descrição

A descrição, juntamente com o tipo, é uma das partes mais importantes do padrão: é aqui que deve ser descrito, de maneira clara, sucinta e simplificada, o que foi realizado no commit. É recomendado que essa parte tenha, no máximo, 70 caracteres, para que não se estenda muito.

Corpo

O corpo do commit é também opcional. Nele, pode-se realizar uma descrição mais detalhada do commit, indicar razões para a realização dele e consequências que ele pode vir a causar, além de alguma outra observação que seja pertinente.

Rodapé

O rodapé, assim como o corpo, é opcional e informativo. Ele pode ser usado como uma finalização do commit, informando o encerramento de uma issue ou o pertencimento/associação a uma task também.

2.4.4 Exemplos

Mensagem de commit com descrição e modificação que quebra a compatibilidade no corpo

```
feat: permitir que o objeto de configuração fornecido estenda
      outras configurações

BREAKING CHANGE: a chave 'extends', no arquivo de configuração,
                 agora é utilizada para estender outro arquivo de
                 configuração
```

Mensagem de commit com ! opcional para chamar a atenção para quebra a compatibilidade

```
chore!: remove Node 6 da matriz de testes

BREAKING CHANGE: removendo Node 6 que atinge o final de vida em Abril
```

Mensagem de commit sem corpo

```
docs: ortografia correta de CHANGELOG
```

Mensagem de commit com escopo

```
feat(lang): adiciona tradução para português brasileiro
```

Mensagem de commit de uma correção utilizando número de ticket (opcional)

```
fix: corrige pequenos erros de digitação no código

veja o ticket para detalhes sobre os erros de digitação corrigidos

finaliza o ticket #12
```

Mensagem de commit de versionamento do banco de dados

```
db: script de criação da tabela cargo
```

2.4.4.1 Periodicidade e Granularidade dos Commits

Definir um padrão que estabelece uma periodicidade de operações de commits ajuda a uniformizar o histórico do projeto. Outro aspecto é que em projetos com vários desenvolvedores é importante um padrão de commits para que todos saibam de forma periódica quando as ações ocorrerão, haja visto que existirão dependência entre as várias frentes de desenvolvimento.

A granularidade do conteúdo dos commits também é um fator que deve seguir um padrão. A granularidade também interfere na rastreabilidade do histórico de mudanças e na colaboração entre os desenvolvedores.

2.4.4.2 Granularidade

A recomendação mais aceita para a granularidade é que cada commit seja atômico, considerando o escopo de uma única funcionalidade. Assim, cada commit só pode conter incremento ou correções de uma funcionalidade. Essa diretriz desconsidera o número de arquivos envolvidos, ou seja, não importa quantos arquivos foram afetados, desde que todos representem uma única funcionalidade ou correção.

O modelo de ramificação e fluxo **Processo de Ramificações e Versionamento** adotado contribui para a adoção desta recomendação, considerando que a produção de novas funcionalidades (feature) ou correções (hotfix) serão realizadas em branches específicas.

Benefícios:

- Fácil de reverter sem afetar outras mudanças
- Fácil de fazer outras alterações na hora
- Fácil de mesclar recursos com outros ramos
- Fácil de rastrear com as Task em um sistema de Issue Tracking

As recomendações estendem-se ainda aos commits relacionados as migrations, onde sugere-se que a cada alteração do banco, relacionado às alterações da versão do software, seja gerado um commit exclusivo para os arquivos de migrations criados. Mais informações estão disponíveis no documento **Migrations**.

2.4.4.3 Periodicidade

A recomendação é que seja realizado pelo menos um commit por dia de trabalho, seguido de um push, desde que o código não esteja com erro de compilação/execução.

Benefícios:

- Gera uma cópia de segurança do esforço de desenvolvimento individual
- Permite que o progresso de um desenvolvedor seja compartilhado o mais cedo possível

2.4.5 Estratégias de Alteração de Commits

Apesar do objetivo ser de fazer commits consistentes, é inevitável situações onde haja a necessidade de alterações. Contudo, é importante estabelecer um padrão de alterações, garantindo também uma boa rastreabilidade das alterações no histórico.

O comando **git commit –amend** é uma forma conveniente de modificar o commit mais recente. Ele permite combinar alterações na área de staging com o commit anterior em vez de criar um novo commit.

2.4.5.1 Alterações da mensagem de commits

Algo indesejável, porém não tão incomum, são os erros cometidos na mensagem do commit, quer seja um texto que ficou incompleto ou na tipificação do commit.

git commit –amend -m ”nova mensagem do commit”

Executar este comando quando não existe nada na área de staging permite que você edite a mensagem do último commit, sem criar uma nova instância de commit.

2.4.5.2 Alterações do conteúdo de commits

Existem situações em que um determinado conjunto de arquivos foram alterados e confirmados em um commit. Contudo, logo após o commit, verifica-se que uma alteração necessária em um ou mais arquivos não foi realiza e não deseja-se criar um novo commit. Para corrigir o erro, basta fazer o staging dos arquivos alterados e confirmar com o sinalizador –amend.

git commit –amend –no-edit

O sinalizador **–no-edit** vai permitir fazer a correção no commit sem alterar a mensagem de confirmação. O commit resultante vai substituir o commit incompleto e vai parecer que as alterações foram confirmadas em um único commit.

Essas alterações são recomendadas apenas em branch que um único desenvolvedor está atuando. Assim, não corrija commits que outros desenvolvedores tomaram como base seu trabalho.

2.4.5.3 Considerações sobre alterações de commits

Apesar de existir a possibilidade de alteração em commits mais antigos ou alterações envolvendo vários commits ao mesmo tempo, por meio do comando git rebase, o padrão sugerido neste documento não encoraja o uso de tal comando, visto que, poderá provocar perdas do histórico e consequentemente dificuldade de rastreabilidade ou retorno a versões antigas.

2.5 Padrão de Nomenclatura de Versões

O padrão de nomenclatura das versões dos sistemas desenvolvidos pela STI-SESP/MT é uma adaptação do sistema de versionamento semântico de autoria de Tom Preston-Werner, criador do Gravatars e co-fundador do GitHub.

Este processo de versionamento irá marcar commits específicos como estáveis e atribuir um rótulo aos mesmos, integrando o padrão de nomenclatura de versionamento ao repositório de código fonte(GitLab).

2.5.1 Níveis

O padrão de nomenclatura de versões de software está dividido em três níveis: Maior, Menor e Correção.

Podemos de forma sucinta representar como **[maior]**.**[menor]**.**[correção]**:

- **[correção]**: otimização e/ou correção de bugs;
- **[menor]**: novas funcionalidades compatíveis com versões anteriores;
- **[maior]**: novas funcionalidades incompatíveis com versões anteriores. Pode envolver uma mudança arquitetural, de stack de tecnologias adotadas, uma reformulação da interface de usuário e/ou da experiência do usuário com o software, por exemplo.

Abaixo apresenta-se o detalhamento de regras quanto aos diferentes níveis:

1. Um número de versão normal DEVE ter o formato de X.Y.Z, onde X, Y, e Z são inteiros não negativos, e NÃO DEVE conter zeros à esquerda. X é a versão Maior, Y é a versão Menor, e Z é a versão de Correção. Cada elemento DEVE aumentar numericamente. Por exemplo: 1.9.0 - 1.10.0 - 1.11.0.
2. Uma vez que um pacote versionado foi lançado(released) em Produção, o conteúdo desta versão NÃO DEVE ser modificado. Qualquer modificação DEVE ser lançado como uma nova versão.
3. No início do desenvolvimento, a versão Maior DEVE ser zero (0.y.z). Qualquer coisa pode mudar a qualquer momento. O software não deve ser considerado estável.
4. Versão 1.0.0 define a primeira versão disponibilizada em Produção.

5. Versão Menor Y (x.Y.z — x maior que 0) DEVE ser incrementada quando houverem novas funcionalidades introduzidas e/ou alteradas no software. DEVE ser incrementada se qualquer funcionalidade do software for descontinuada(retirada).

6. Versão de Correção Z (x.y.Z — x maior que 0) DEVE ser incrementado com a correção de bugs. Uma correção de bug é definida como uma mudança interna que corrige um comportamento incorreto. A versão de Correção deve ser redefinida para 0(zero) quando a versão Menor for incrementada.

7. Versão Maior X (X.y.z — X maior que 0) DEVE ser incrementada se forem introduzidas mudanças substanciais no software[descrever o que são mudanças substanciais]. Versão Menor e Versão de Correção devem ser redefinidas para 0(zero) quando a versão Maior for incrementada.

8. Uma versão de Pré-Lançamento (pre-release) PODE ser identificada adicionando um hífen (dash) e uma série de identificadores separados por ponto (dot) imediatamente após a versão de Correção. Identificador DEVE incluir apenas caracteres alfanuméricos e hífen [0-9A-Za-z-]. Identificador NÃO DEVE ser vazio. Indicador numérico NÃO DEVE incluir zeros à esquerda. Versão de Pré-Lançamento tem precedência inferior à versão normal a que está associada. Uma versão de Pré-Lançamento (pre-release) indica que a versão é instável e pode não satisfazer os requisitos de compatibilidade pretendidos, como indicado por sua versão normal associada. Exemplos: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.

9. Metadados de construção(Build) PODE ser identificada por adicionar um sinal de adição (+) e uma série de identificadores separados por ponto imediatamente após a Correção ou Pré-Lançamento. Identificador DEVE ser composto apenas por caracteres alfanuméricos e hífen [0-9A-Za-z-]. Identificador NÃO DEVE ser vazio. Metadados de construção PODEM ser ignorados quando se determina a versão de precedência. Assim, duas versões que diferem apenas nos metadados de construção, têm a mesma precedência. Exemplos: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.

10. A precedência refere como as versões são comparadas com cada outra quando solicitado. A precedência DEVE ser calculada separando identificadores de versão em Maior, Menor, Correção e Pré-lançamento, nesta ordem (Metadados de construção não figuram na precedência). A precedência é determinada pela primeira diferença quando se compara cada identificador da esquerda para direita, como se segue: Versões Maior, Menor e Correção são sempre comparadas numericamente. Example: 1.0.0 menor 2.0.0 menor 2.1.0 menor 2.1.1. Quando Maior, Menor e Correção são iguais, a versão de Pré-Lançamento tem precedência menor que a versão normal. Example: 1.0.0-alpha menor 1.0.0. A precedência entre duas versões de Pré-lançamento com mesma versão Maior, Menor e Correção DEVE ser determinada comparando cada identificador separado por ponto da esquerda para direita até que seja encontrada diferença da seguinte forma: identificadores consistindo apenas dígitos são comparados numericamente e identificadores com letras ou hífen são comparados lexicalmente na ordem de classificação ASCII. Identificadores numéricos sempre têm menor precedência do que os não numéricos. Um conjunto maior de campos de pré-lançamento

tem uma precedência maior do que um conjunto menor, se todos os identificadores anteriores são iguais. Example: 1.0.0-alpha menor 1.0.0-alpha.1 menor 1.0.0-alpha.beta menor 1.0.0-beta menor 1.0.0-beta.2 menor 1.0.0-beta.11 menor 1.0.0-rc.1 menor 1.0.0.

2.5.2 Desenvolvimento Inicial

A versão de desenvolvimento inicial DEVE ser 0.1.0 e, então, incrementar a uma versão 'menor' em cada lançamento subsequente.

2.6 Análise da Ferramenta de Versionamento

Esta seção contém as informações a respeito da análise e decisões a respeito de ferramentas de versionamento e gerenciamento de fluxo e ramificações a serem adotadas na área de desenvolvimento de software da Secretaria de Segurança Pública do Estado de Mato Grosso (SESPMT).

2.6.1 Critérios de escolha adotados para a ferramenta de versionamento

Considerando o edital Editais e Normas SESP/FAPEMAT que estabelece os itens:

1. Sincronizar com o sistema de controle de versões (gitlab) da SESP
2. Possuir características funcionais adequadas às especificidades de atuação da SESP
3. Ser de baixo custo de implantação e manutenção
4. Utilizar conceitos DevOps

Acrescido do levantamento das características do ambiente de desenvolvimento atual da SESP, foram adotados os seguintes critérios para a escolha de ferramentas de versionamento:

1. Versionamento baseado no padrão Git
2. Baixo custo, preferencialmente de custo zero de licença e implantação
3. Permita a implementação ou integração de um pipeline de Integração Contínua
4. Permite o rastreamento de task com os códigos no repositório
5. Permite a instalação e configura em servidor próprio da SESP ou fornecido pela Infraestrutura do Estado

2.6.2 Matriz de Decisão

2.6.3 Considerações sobre a escolha

Diante da pesquisa e comparações das ferramentas para a gestão de repositórios Git, a ferramenta Gitlab[1] foi a que atendeu o maior número de critérios de escolha.

[1] https://about.gitlab.com/

Tabela 1 – Matriz de Decisão da Ferramenta de Versionamento

Ferramenta	C1	C2	C3	C4	C5
Gitlab	X	X	X	X	X
GitHub Enterprise	X	-	-	X	X
Bitbucket	X	-	-	X	X
Phabricator	X	X	-	X	X
Gitblit	X	-	-	X	X

Além disso, um critério que não foi considerado em função de não eliminar automaticamente as outras ferramentas analisadas, mas que reforça a escolha, é que a SESP já utiliza a ferramenta Gitlab para o versionamento de seus artefatos de projetos de desenvolvimento de software.

Outra característica que foi considerada como ponto decisivo é a capacidade que o Gitlab tem de automação da Integração Contínua, objetivo do atual projeto em execução.

2.6.4 Ferramenta de Automação de Fluxo e Ramificações

Considerando que uma ferramenta ou extensão de automação fluxo e ramificações pode estar intrinsecamente ligada à ferramenta de gestão de repositório Git, foi adotada a estratégia de primeiro escolher e decidir a ferramenta de gestão de repositório, discutida no tópico anterior.

Uma vez adotada a ferramenta Gitlab como ferramenta de gestão de repositório Git, será realizada um aprofundamento das características que permitem a automação de fluxo e ramificações. Contudo, estas soluções podem ser afetadas ou interferidas pelas decisões que ocorrerão no próximo ciclo de desenvolvimento do projeto (Sprint), que é o de definição de estratégias de Build de código.

Assim, a decisão de quais recursos ou características do Gitlab serão usadas para apoiar as atividades relacionadas ao fluxo e ramificações do repositórios serão escolhidas e definidas no próximo ciclo, juntamente com as decisões relacionadas ao Build de código.

2.7 Exemplo prático com Git flow

O roteiro abaixo apresenta o modelo de versionamento e ramificação para fins de exemplificação prática dos conceitos discutidos nesse cápitulo.

2.7.1 Instruções

2.7.1.1 Configuração do projeto:

1. Acessar https://gitlab.fabricadesoftwareifmt.com.br e criar projeto
2. Copiar Informações para clonar o repositório
3. No Eclipse mudar a perspectiva para Git

Figura 10 – Exemplo Git Flow

4. Clonar o projeto no Eclipse
5. Como não temos nenhum projeto dentro do repositório vamos criar um e conectar com nosso repositorio.
6. Criar um novo projeto utilizando o Java 1.5
7. Após criar o projeto vamos conectar ele com o repositório Git:
    a) Botão direito em cima do projeto, team - share project
    b) Selecionar o repositório Git
    c) Finalizar
8. Para utilizar o Git Flow, instalar o plugin Gitflow Nightly:
    a) No Eclipse acessar o Market Place em Help
    b) Procure por git flow
    c) Instale o plugin Gitflow Nightly
9. Após instalar o plugin vamos iniciar o Gitflow no repositório:
    a) Botão direito em cima do projeto, team - Init Gitflow
    b) Confirmar a criação
    c) Configurar os pre-fixos das branches
    d) Finalizar

2.7.1.2 Desenvolvimento do projeto:

1. Adicionar o pacote br.com.fabricadesoftwareifmt
2. Adicionar a primeira classe e código inicial do projeto:

a) Criar classe com nome Editor.java com o seguinte código:

```
public class Editor {

  public static void main(String[] args) {
    // TODO Auto-generated method stub
  }

}
```

Código 2.1 – Criando a Classe Editor.java

3. Realizar o primeiro commit e push do projeto

   a) Adicionar arquivos pertinentes (Editor.java e .gitignore)

   b) Commitar com a mensagem "**Estrutura inicial do projeto**"

4. Adicionar a versão inicial do Editor de Texto

   a) Adicionar o código

```
package br.com.fabricadesoftwareifmt;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;

public class Editor extends JFrame implements ActionListener {

  public static void main(String[] args) {
    new Editor();
  }

  public Editor(){
    configuraJanela();
    setVisible(true);
  }

  private void configuraJanela() {
    setSize(840, 600);
    this.setTitle("Editor GPL");
    this.setLayout(new BorderLayout());
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }

  public void actionPerformed(ActionEvent arg) {
    // TODO Auto-generated method stub
  }

}
```

Código 2.2 – Implementação da Classe Editor.java

   b) Commitar com a seguinte mensagem "**feat: Implementado a janela do editor de texto**"

   c) Fazer push da branch **develop** para origin

5. No Eclipse iniciar a branch **feature/adiciona-textarea** no Gitflow

a) Botão direito em cima do projeto, **team - Gitflow - Start Feature**

b) Fazer push da branch para origin (Gitlab)

c) Substituir o código conforme trecho abaixo

```
package br.com.fabricadesoftwareifmt;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;

public class Editor extends JFrame implements ActionListener {

  public static void main(String[] args) {
    new Editor();
  }

  public Editor(){
    configuraJanela();
    setVisible(true);
  }

  private void configuraJanela() {
    setSize(840, 600);
    this.setTitle("Editor GPL");
    this.setLayout(new BorderLayout());
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }

  public void actionPerformed(ActionEvent arg) {
    // TODO Auto-generated method stub
  }

}
```

Código 2.3 – Estrutura inicial da Classe Editor.java

d) Commitar com a seguinte mensagem: "**draft: Implementação do componente de texto do editor**"

e) Commitar e fazer push para Origin (Gitlab)

6. Finalizar a implementação da **feature/adiciona-textarea**

a) Substituir o código conforme trecho abaixo

```
package br.com.fabricadesoftwareifmt;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;

```

```
14 public class Editor extends JFrame implements ActionListener {
15
16   private JPanel jPanelCampos;
17   private JPanel jpanelEstilos;
18
19   private JTextPane textArea;
20   private JButton bt2Sair;
21   private JScrollPane scroll;
22
23   public static void main(String[] args) {
24     new Editor();
25   }
26
27   public Editor() {
28     configuraJanela();
29     inicializar();
30     scroll.add(textArea);
31     scroll.setViewportView(textArea);
32     this.add(scroll);
33     setVisible(true);
34   }
35
36   private void configuraJanela() {
37     setSize(840, 600);
38     this.setTitle("Editor de texto");
39     this.setLayout(new BorderLayout());
40     this.setDefaultCloseOperation(EXIT_ON_CLOSE);
41   }
42
43   private void inicializar() {
44     this.add(getPanelCampos(), BorderLayout.SOUTH);
45     this.add(getPanelEstilos(), BorderLayout.NORTH);
46   }
47
48   private Component getPanelEstilos() {
49     if (jpanelEstilos == null)
50     {
51       jpanelEstilos = new JPanel();
52       scroll = new JScrollPane();
53     }
54     return jpanelEstilos;
55   }
56
57   private Component getPanelCampos() {
58     if (jPanelCampos == null) {
59       jPanelCampos = new JPanel();
60       textArea = new JTextPane();
61       jPanelCampos.add(textArea);
62
63     }
64     return jPanelCampos;
65   }
66
67   public void actionPerformed(ActionEvent arg) {
68     // TODO Auto-generated method stub
69   }
70
71 }
```

Código 2.4 – Alteração da Classe Editor.java

b) Commitar e fazer push para origin com o seguinte comentário: "**feat: Implementado o componente textarea**"

7. No Eclipse finalizar a **feature/adiciona-textarea** no Git flow

   a) Botão direito em cima do projeto, **team - Gitflow - Finish Feature**

   b) Ao finalizar a feature o plugin do Gitflow faz o merge da branch **feature** para a branch **develop** no repositório local do Git

8. Fazer push da branch **develop** para origin

   a) Os commits gerados na branch **feature/adiciona-textarea** são enviados para origin

9. No Eclipse, iniciar outra branch **feature** no Gitflow chamada **feature/manipulacao-arquivo**

   a) Fazer push da branch **feature/manipulacao-arquivopara** origin

   b) Voltar para a branch **develop**

10. No Eclipse, iniciar outra branch feature no Gitflow chamada **feature/estilos-texto**

    a) Fazer push da branch **feature/estilos-texto** para origin

11. Implementar a branch **feature/estilos-texto**.

    a) Substituir o código conforme trecho abaixo

```
1  package br.com.fabricadesoftwareifmt;
2
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7
8  import javax.swing.JCheckBox;
9  import javax.swing.JFrame;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextPane;
13
14 public class Editor extends JFrame implements ActionListener {
15
16   private JPanel jPanelCampos;
17   private JPanel jpanelEstilos;
18
19   private JTextPane textArea;
20   private JCheckBox negrito;
21   private JCheckBox italico;
22   private JCheckBox sublinhado;
23   private JScrollPane scroll;
24
25   public static void main(String[] args) {
26     new Editor();
27   }
28
29   public Editor() {
30     configuraJanela();
31     inicializar();
```

```
32      scroll.add(textArea);
33      scroll.setViewportView(textArea);
34      this.add(scroll);
35      setVisible(true);
36    }
37
38    private void configuraJanela() {
39      setSize(840, 600);
40      this.setTitle("Editor de texto");
41      this.setLayout(new BorderLayout());
42      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
43    }
44
45    private void inicializar() {
46      this.add(getPanelCampos(), BorderLayout.SOUTH);
47      this.add(getPanelEstilos(), BorderLayout.NORTH);
48    }
49
50
51    private Component getPanelEstilos() {
52      if (jpanelEstilos == null)
53      {
54        jpanelEstilos = new JPanel();
55        scroll = new JScrollPane();
56        negrito = new JCheckBox("Negrito");
57        italico = new JCheckBox("Italico");
58        sublinhado = new JCheckBox("Sublinhado");
59
60        negrito.addActionListener(this);
61        italico.addActionListener(this);
62        sublinhado.addActionListener(this);
63
64        jpanelEstilos.add(negrito);
65        jpanelEstilos.add(italico);
66        jpanelEstilos.add(sublinhado);
67      }
68      return jpanelEstilos;
69    }
70
71    private Component getPanelCampos() {
72      if (jPanelCampos == null) {
73        jPanelCampos = new JPanel();
74        textArea = new JTextPane();
75        jPanelCampos.add(textArea);
76
77      }
78      return jPanelCampos;
79    }
80
81    public void actionPerformed(ActionEvent arg) {
82      // TODO Auto-generated method stub
83    }
84
85  }
```

Código 2.5 – Alteração da Classe Editor.java

b) Commitar e fazer push para origin com o seguinte comentário: "**draft: Adicionado botões dos estilos**"

12. Mudar para a branch **feature/manipulacao-arquivo** para implementar a branch

a) Substituir o código conforme trecho abaixo

```
package br.com.fabricadesoftwareifmt;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;

public class Editor extends JFrame implements ActionListener {

  private JPanel jPanelCampos;
  private JPanel jpanelEstilos;

  private JTextPane textArea;
  private JPanel jPanelBotoes;
  private JButton btAbrir;
  private JButton bt2Gravar;
  private JButton bt2Sair;
  private JScrollPane scroll;

  public static void main(String[] args) {
    new Editor();
  }

  public Editor() {
    configuraJanela();
    inicializar();
    scroll.add(textArea);
    scroll.setViewportView(textArea);
    this.add(scroll);
    setVisible(true);
  }

  private void configuraJanela() {
    setSize(840, 600);
    this.setTitle("Editor de texto");
    this.setLayout(new BorderLayout());
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }

  private void inicializar() {
    this.add(getPanelCampos(), BorderLayout.SOUTH);
    this.add(getPanelEstilos(), BorderLayout.NORTH);
  }


  private Component getPanelEstilos() {
    if (jpanelEstilos == null)
    {
      jpanelEstilos = new JPanel();
      scroll = new JScrollPane();

      jPanelBotoes = new JPanel();
      btAbrir = new JButton("Abrir");
      bt2Gravar = new JButton("Gravar");
```

```
61        bt2Sair = new JButton("Sair");
62
63        jpanelEstilos.add(btAbrir);
64        jpanelEstilos.add(bt2Gravar);
65        jpanelEstilos.add(bt2Sair);
66      }
67      return jpanelEstilos;
68    }
69
70    private Component getPanelCampos() {
71      if (jPanelCampos == null) {
72        jPanelCampos = new JPanel();
73        textArea = new JTextPane();
74        jPanelCampos.add(textArea);
75
76      }
77      return jPanelCampos;
78    }
79
80    public void actionPerformed(ActionEvent arg) {
81      // TODO Auto-generated method stub
82    }
83
84  }
```

Código 2.6 – Alteração da Classe Editor.java

b) Commitar e fazer push para origin com o seguinte comentário: "**draft: Adicionado botões de manipulacao**"

13. Finalizar a implementação dos botões da feature/manipulacao-arquivo

a) Substituir o código conforme trecho abaixo

```
1  package br.com.fabricadesoftwareifmt;
2
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.io.BufferedReader;
8  import java.io.File;
9  import java.io.FileNotFoundException;
10 import java.io.FileOutputStream;
11 import java.io.FileReader;
12 import java.io.IOException;
13
14 import javax.swing.JButton;
15 import javax.swing.JFileChooser;
16 import javax.swing.JFrame;
17 import javax.swing.JPanel;
18 import javax.swing.JScrollPane;
19 import javax.swing.JTextPane;
20
21 public class Editor extends JFrame implements ActionListener {
22
23   private JPanel jPanelCampos;
24   private JPanel jpanelEstilos;
25
26   private JTextPane textArea;
27   private JPanel jPanelBotoes;
```

```
28  private JButton btAbrir;
29  private JButton bt2Gravar;
30  private JButton bt2Sair;
31  private JScrollPane scroll;
32
33  public static void main(String[] args) {
34    new Editor();
35  }
36
37  public Editor() {
38    configuraJanela();
39    inicializar();
40    scroll.add(textArea);
41    scroll.setViewportView(textArea);
42    this.add(scroll);
43    setVisible(true);
44  }
45
46  private void configuraJanela() {
47    setSize(840, 600);
48    this.setTitle("Editor de texto");
49    this.setLayout(new BorderLayout());
50    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
51  }
52
53  private void inicializar()
54  {
55    this.add(getPanelCampos(), BorderLayout.SOUTH);
56    this.add(getPanelEstilos(), BorderLayout.NORTH);
57  }
58
59  private Component getPanelEstilos() {
60    if (jpanelEstilos == null) {
61      jpanelEstilos = new JPanel();
62      scroll = new JScrollPane();
63
64      jPanelBotoes = new JPanel();
65      btAbrir = new JButton("Abrir");
66      bt2Gravar = new JButton("Gravar");
67      bt2Sair = new JButton("Sair");
68
69      jpanelEstilos.add(btAbrir);
70      btAbrir.addActionListener(new ActionListener() {
71
72        public void actionPerformed(ActionEvent arg0) {
73
74        JFileChooser chooserAbrirArquivo = new JFileChooser();
75        chooserAbrirArquivo.showOpenDialog(null);
76        FileReader fr = null;
77        try {
78          fr = new FileReader(chooserAbrirArquivo
79          .getSelectedFile());
80        } catch (FileNotFoundException e) {
81          e.printStackTrace();
82        }
83        BufferedReader br = new BufferedReader(fr);
84        String linha;
85        StringBuffer sb = new StringBuffer();
86        try {
87          while ((linha = br.readLine()) != null) {
88            sb.append(linha).append("\n");
```

```
89            }
90          } catch (IOException e) {
91            e.printStackTrace();
92          }
93          try {
94            fr.close();
95          } catch (IOException e) {
96            e.printStackTrace();
97          }
98            textArea.setText(sb.toString());
99          }
100
101       });
102
103       jpanelEstilos.add(bt2Gravar);
104       bt2Gravar.addActionListener(new ActionListener() {
105
106         public void actionPerformed(ActionEvent arg0) {
107
108           File salvarArquivo = null;
109
110           JFileChooser escolherCaminhoSalvar = new JFileChooser();
111           escolherCaminhoSalvar
112               .setFileSelectionMode(JFileChooser.FILES_ONLY);
113           // file.set
114           int i = escolherCaminhoSalvar.showSaveDialog(null);
115           if (i == 1) {
116             textArea.setText("");
117           } else {
118             salvarArquivo = escolherCaminhoSalvar
119                 .getSelectedFile();
120           }
121
122           FileOutputStream fos = null;
123           try {
124             fos = new FileOutputStream(salvarArquivo);
125           } catch (FileNotFoundException e1) {
126             // TODO Auto-generated catch block
127             e1.printStackTrace();
128           }
129           try {
130             fos.write(textArea.getText().getBytes());
131           } catch (IOException e) {
132             // TODO Auto-generated catch block
133             e.printStackTrace();
134           }
135           try {
136             fos.close();
137           } catch (IOException e) {
138             // TODO Auto-generated catch block
139             e.printStackTrace();
140           }
141
142         }
143       });
144
145       jpanelEstilos.add(bt2Sair);
146       bt2Sair.addActionListener(new ActionListener() {
147         public void actionPerformed(ActionEvent arg0) {
148
149         }
```

```
150         });
151       }
152       return jpanelEstilos;
153     }
154
155     private Component getPanelCampos() {
156       if (jPanelCampos == null) {
157         jPanelCampos = new JPanel();
158         textArea = new JTextPane();
159         jPanelCampos.add(textArea);
160
161       }
162       return jPanelCampos;
163     }
164
165     public void actionPerformed(ActionEvent arg) {
166       // TODO Auto-generated method stub
167     }
168
169 }
```

Código 2.7 – Alteração da Classe Editor.java

b) Commitar e fazer push para origin com o seguinte comentário: "**feat: Implementado a manipulação de arquivos no editor de texto**"

14. Finalizar a branch **feature/manipulacao-arquivo**

a) Botão direito em cima do projeto, **team - Gitflow - Finish Feature**

b) Ao finalizar a feature o plugin do Gitflow faz o merge da branch **feature** para a branch **develop** no repositório local do Git

15. Fazer push da branch **develop** para origin

a) Os commits gerados na branch **feature/adiciona-textarea** são enviados para origin

16. No Eclipse mudar para a branch **feature/estilos-texto**

17. Finalizar a implementação dos botões da **feature/estilos-texto**

a) Substituir o código conforme trecho abaixo

```
1  package br.com.fabricadesoftwareifmt;
2
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7
8  import javax.swing.JCheckBox;
9  import javax.swing.JFrame;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextPane;
13 import javax.swing.text.BadLocationException;
14 import javax.swing.text.Style;
15 import javax.swing.text.StyleConstants;
16 import javax.swing.text.StyledDocument;
```

```
17
18 public class Editor extends JFrame implements ActionListener {
19
20   private JPanel jPanelCampos;
21   private JPanel jpanelEstilos;
22
23   private JTextPane textArea;
24   private JCheckBox negrito;
25   private JCheckBox italico;
26   private JCheckBox sublinhado;
27   private JScrollPane scroll;
28
29   public static void main(String[] args) {
30     new Editor();
31   }
32
33   public Editor() {
34     configuraJanela();
35     inicializar();
36     scroll.add(textArea);
37     scroll.setViewportView(textArea);
38     this.add(scroll);
39     setVisible(true);
40   }
41
42   private void configuraJanela() {
43     setSize(840, 600);
44     this.setTitle("Editor de texto");
45     this.setLayout(new BorderLayout());
46     this.setDefaultCloseOperation(EXIT_ON_CLOSE);
47   }
48
49   private void inicializar() {
50     this.add(getPanelCampos(), BorderLayout.SOUTH);
51     this.add(getPanelEstilos(), BorderLayout.NORTH);
52   }
53
54
55   private Component getPanelEstilos() {
56     if (jpanelEstilos == null)
57     {
58       jpanelEstilos = new JPanel();
59       scroll = new JScrollPane();
60       negrito = new JCheckBox("Negrito");
61       italico = new JCheckBox("Italico");
62       sublinhado = new JCheckBox("Sublinhado");
63
64       negrito.addActionListener(this);
65       italico.addActionListener(this);
66       sublinhado.addActionListener(this);
67
68       negrito.addActionListener(this);
69       italico.addActionListener(this);
70       sublinhado.addActionListener(this);
71
72       jpanelEstilos.add(negrito);
73       jpanelEstilos.add(italico);
74       jpanelEstilos.add(sublinhado);
75     }
76     return jpanelEstilos;
77   }
```

```
78
79    private Component getPanelCampos() {
80      if (jPanelCampos == null) {
81        jPanelCampos = new JPanel();
82        textArea = new JTextPane();
83        jPanelCampos.add(textArea);
84
85      }
86      return jPanelCampos;
87    }
88
89    public void modificaEstilo() {
90      StyledDocument documento = (StyledDocument) textArea.getDocument();
91      Style estilo = documento.getStyle(documento.addStyle("StyleAdd", null)
92      .getName());
93
94
95      StyleConstants.setBold(estilo, negrito.isSelected());
96      StyleConstants.setItalic(estilo, italico.isSelected());
97      StyleConstants.setUnderline(estilo, sublinhado.isSelected());
98
99      String textoFormatado = textArea.getText();
100     textArea.setText("");
101     documento.addStyle("Style", estilo);
102
103     try {
104       documento.insertString(documento.getLength(), textoFormatado,
105         estilo);
106     } catch (BadLocationException e) {
107       // TODO Auto-generated catch block
108       e.printStackTrace();
109     }
110   }
111
112   public void actionPerformed(ActionEvent arg) {
113     modificaEstilo();
114   }
115
116 }
```

Código 2.8 – Alteração da Classe Editor.java

b) Commitar e fazer push para origin com o seguinte comentário: "**feat: Implementado o componente dos estilos do texto (negrito, itálico e sublinhado)**"

c) Fazer merge da branch **develop** para a branch **feature/estilos-texto**

d) Resolver os conflitos gerados pelo merge na branch **feature/estilos-texto**

e) Commitar e fazer push para origin com o seguinte comentário: "**fix: Resolvido conflitos de merge 'origin/develop'em feature/estilos-texto**"

18. Finalizar a branch **feature/estilos-texto**

a) Botão direito em cima do projeto, **team - Gitflow - Finish Feature**

b) Ao finalizar a feature o plugin do Gitflow faz o merge da branch **feature** para a branch **develop** no repositório local do Git

19. Fazer push da branch **develop** para origin

a) Os commits gerados na branch **feature/adiciona-textarea** são enviados para origin

20. No Eclipse, iniciar outra branch release no Gitflow chamada **release/1.0.0**

    a) Substituir o código do texto do editor conforme trecho abaixo

```
this.setTitle("Editor de texto 1.0");
```

Código 2.9 – Alteração da Classe Editor.java

    b) Commitar e fazer push para origin com o seguinte comentário: "**fix: Adicionado a versão do editor no título do painel**"

21. Finalizar a branch **release/1.0.0**

    a) Botão direito em cima do projeto, **team - Gitflow - Finish Release**

    b) Ao finalizar a release o Git flow gera uma tag com o nome da release

    c) Ao finalizar a release o plugin do Gitflow faz o merge da branch **release** para a branch **develop** e para a branch **master** no repositório local do Git

22. Fazer push da branch **develop** para origin

    a) Os commits gerados na branch **feature/adiciona-textarea** são enviados para origin

23. Após a homologação da branch **release/1.0.0** enviar a tag pelo Eclipse para o Gitlab

24. Fazer merge request da branch **release/1.0.0** para branch **master**

25. No Eclipse, mudar para a branch **master**

    a) Fazer pull da origin

    b) Iniciar branch hotfix no Gitflow chamada **hotfix/corrige-botao-sair**

    c) Fazer push da branch **hotfix/corrige-botao-sair**

    d) Adicionar o seguinte código ao método actionPerformed() do botão sair

```
bt2Sair.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
      System.exit(0);
    }
});
```

Código 2.10 – Alteração da Classe Editor.java

    e) Commitar e fazer push para origin com o seguinte comentário: "**fix: Corrigido evento do botão sair Ao clicar no botão sair do editor de texto o evento era disparado mas o editor de texto não era encerrado**"

26. Finalizar a branch **hotfix/corrige-botao-sair**

    a) Botão direito em cima do projeto, **team - Gitflow - Finish Hotfix**

    b) Ao finalizar a hotfix o plugin do Gitflow faz o merge da branch **release** para a branch **develop** e para a branch **master** no repositório local do Git

27. azer push da branch **develop** para origin

    a) Os commits gerados na branch **hotfix/corrige-botao-sair** são enviados para origin

28. Fazer merge request da branch **hotfix/corrige-botao-sair** para branch **master**

29. No Eclipse, mudar para a branch **master**

    a) Fazer pull da origin

CAPÍTULO 3

Processo de Build e Gestão de Implantação

Este capítulo descreve o processo de build (construção/empacotamento/distribuição) proposto para a SESP-MT, visando conciliar com as principais características de soluções existentes e acrescentar novos elementos, técnicas e ferramentas que dão suporte a fase de Build e entrega de componentes de software para a implantação, aderentes aos conceitos de Integração Contínua.

3.1 Cenário Atual do Processo de Build na SESP

3.1.1 Tecnologias e Ambiente

No processo atual de Build identificou-se as principais tecnologias utilizadas, sendo:

- IDE Eclipse para desenvolvimento
- Maven para controle de dependências e compilação do projeto
- Giltab para gerenciamento do código fonte
- Jenkins para construção dos artefatos e publicação no repositório
- Nexus para armazenamento de artefatos (Bibliotecas, Imagens Docker, etc)
- Rancher para gerenciamento de containers

Inicialmente verificou-se que todos os desenvolvedores recebem um arquivo para realizar as configurações do ambiente de desenvolvimento.

Dentre as informações de configuração do Java e do Eclipse, a mais relevante para essa análise foi a configuração do Maven apontando para o repositório do Nexus obrigatoriamente. Essa configuração é realizada no arquivo **settings.xml** do Maven.

Todas as dependências do Maven precisam ser baixadas obrigatoriamente do repositório Nexus da SESP.

Os projetos possuem o POM.XML para gerenciamento das dependências e envio dos projetos e bibliotecas para o Nexus. Todos os jars, wars, etc. podem ser enviados para o Nexus, porém, usualmente apenas as libs e imagens do Docker que são enviadas. Não existe uma regra específica para isso. Detalhes do POM.XML estão disponíveis no código a seguir:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
      maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>br.gov.mt.sesp.pessoa</groupId>
  <artifactId>pessoa_fonetica</artifactId>
  <version>0.0.3-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>pessoa_fonetica</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.2.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>biz.paluch.logging</groupId>
        <artifactId>logstash-gelf</artifactId>
        <version>1.13.0</version>
    </dependency>

    <!-- drive JDBC oracle -->
    <dependency>
      <groupId>com.oracle</groupId>
      <artifactId>ojdbc7</artifactId>
```

```
53        <version>12.1.0.2</version>
54      </dependency>
55
56      <dependency>
57        <groupId>org.springframework.boot</groupId>
58        <artifactId>spring-boot-starter-test</artifactId>
59        <scope>test</scope>
60      </dependency>
61
62    </dependencies>
63
64    <build>
65      <pluginManagement>
66        <plugins>
67          <plugin>
68            <groupId>org.apache.maven.plugins</groupId>
69            <artifactId>maven-deploy-plugin</artifactId>
70            <version>3.0.0-M1</version>
71          </plugin>
72        </plugins>
73      </pluginManagement>
74      <plugins>
75        <plugin>
76          <groupId>org.springframework.boot</groupId>
77          <artifactId>spring-boot-maven-plugin</artifactId>
78        </plugin>
79        <plugin>
80          <groupId>org.apache.maven.plugins</groupId>
81          <artifactId>maven-deploy-plugin</artifactId>
82        </plugin>
83      </plugins>
84    </build>
85
86    <distributionManagement>
87      <repository>
88        <id>maven-snapshots</id>
89        <name>maven-snapshots</name>
90        <url>http://nexus.seguranca.local:8081/repository/maven-snapshots/</url>
91      </repository>
92    </distributionManagement>
93
94  </project>
```

Código 3.1 – Arquivo pom.xml da SESP

O Projeto apresentado foi desenvolvido em Spring boot. Para esse tipo de projeto, os desenvolvedores podem compilar o projeto com o Maven e testar na própria máquina com servidor local do Spring boot.

3.1.2 Soluções de Automatização Atual

Para o processo automatizado de build é utilizado um servidor Jenkins. Nesse servidor foi configurado um Job do tipo “Multibranch Pipeline”. Esse tipo de pipeline detecta alterações nas branches Master, Develop, Hotfix e Release do respectivo projeto gerenciado no Gitlab e inicia o processo de build em caso de alguma alteração.

O processo de build é gerenciado por um script Groovy e separado em 3 stages sendo “Build desenvolvimento”, “Build Homologação”e “Build Producao”. Todos esses stages executam o Maven

para compilar e resolver dependências, cosntruir uma imagem de container, armazenar a imagem do container no Nexus e finalmente publicar a imagem no Rancher. Os detalhes estão disponíveis no arquivo jenkinsfile a seguir:

```
/* Arquivo modelo de Jenkinsfile script */

pipeline {

    agent any

    tools {
        maven 'MAVEN_3'
        jdk 'JDK8u221'
    }

    environment {
        ARTIFACTID_POM = readMavenPom().getArtifactId()
    }

    stages {

        /* build branch release, hotfix, feature,  */
        stage ('Build Desenvolvimento') {

            when {
                anyOf { branch 'release/*'; branch 'hotfix/*'; branch 'feature/*';
                    branch 'bugfix/*' }
            }

            steps {

                withMaven() {
                    sh 'echo "maven package..."'
                    sh 'mvn clean package -B -P desenvolvimento  -DskipTests=true'
                }

                withCredentials([usernamePassword(credentialsId: 'docker-registry',
                    passwordVariable: 'PASSWORD', usernameVariable: 'USERNAME')]) {
                    sh 'echo "Docker Builder..."'
                    sh 'docker login -u $USERNAME -p $PASSWORD nexus.seguranca.
                        local:8083'
                    sh 'docker build -t nexus.seguranca.local:8083/sesp/$
                        ARTIFACTID_POM:dev .'
                    sh 'docker push nexus.seguranca.local:8083/sesp/$
                        ARTIFACTID_POM:dev'
                }

              rancherRedeploy alwaysPull: true, credential: 'rancher-api-key', images:
                  'nexus.seguranca.local:8083/sesp/pessoa_fonetica:dev',
                workload: '/project/c-dv6lp:p-h9jhj/workloads/deployment:pessoa-
                    fonetica:pessoa-fonetica'

            }

        }

        stage ('Build Homologacao') {

            when {
                branch 'develop'
            }
```

```
51
52              steps {
53
54                  withMaven() {
55                      sh 'echo "maven package..."'
56                      sh 'mvn clean package -B -P homologacao  -DskipTests=true'
57                  }
58
59                  withCredentials([usernamePassword(credentialsId: 'docker-registry',
                        passwordVariable: 'PASSWORD', usernameVariable: 'USERNAME')]) {
60                      sh 'echo Docker Builder'
61                      sh 'docker login -u $USERNAME -p $PASSWORD nexus.seguranca.
                            local:8083'
62                      sh 'docker build -t nexus.seguranca.local:8083/sesp/$
                            ARTIFACTID_POM:hmg .'
63                      sh 'docker push nexus.seguranca.local:8083/sesp/$
                            ARTIFACTID_POM:hmg'
64                  }
65
66                  rancherRedeploy alwaysPull: true, credential: 'rancher-api-key',
                        images: 'nexus.seguranca.local:8083/sesp/pessoa_fonetica:hmg',
67                      workload: '/project/c-l85kn:p-bdg65/workloads/deployment:pessoa-
                            fonetica-hmg:pessoa-fonetica'
68
69              }
70          }
71
72
73          /* build somente master, producao */
74          stage ('Build Producao') {
75
76              when {
77                  branch 'master'
78              }
79
80              steps {
81
82                  withMaven() {
83                      sh 'mvn clean package -B -P producao -DskipTests=true'
84                  }
85
86                  withMaven() {
87                      sh 'mvn deploy -DskipTests=true'
88                  }
89
90                  withCredentials([usernamePassword(credentialsId: 'docker-registry',
                        passwordVariable: 'PASSWORD', usernameVariable: 'USERNAME')]) {
91                      sh 'docker login -u $USERNAME -p $PASSWORD nexus.seguranca.
                            local:8083'
92                      sh 'docker build -t nexus.seguranca.local:8083/sesp/$
                            ARTIFACTID_POM:pro .'
93                      sh 'docker push nexus.seguranca.local:8083/sesp/$
                            ARTIFACTID_POM:pro'
94                  }
95
96                rancherRedeploy alwaysPull: true, credential: 'rancher-api-key', images:
                      'nexus.seguranca.local:8083/sesp/pessoa_fonetica:pro',
97                  workload: '/project/c-dv6lp:p-h9jhj/workloads/deployment:pessoa-
                        fonetica:pessoa-fonetica'
98
99              }
```

```
100
101         }
102
103     }
104
105     post {
106
107         success {
108
109       gitlabCommitStatus(builds: [[connection: gitLabConnection('GitLab Sesp Server'),
              projectId: '35', revisionHash: GIT_COMMIT]],
110         name: 'build', connection: gitLabConnection('GitLab Sesp Server')) {
111           echo 'success'
112       }
113
114         }
115
116         failure {
117
118       gitlabCommitStatus(builds: [[connection: gitLabConnection('GitLab Sesp Server'),
              projectId: '35', revisionHash: GIT_COMMIT]],
119         name: 'build', connection: gitLabConnection('GitLab Sesp Server')) {
120           echo 'failure'
121       }
122
123         }
124
125     }
126
127 }
```

Código 3.2 – Arquivo jenkinsfile da SESP

A imagem do Docker é construída com a partir do seguinte Dockerfile:

```
1 FROM openjdk:8-jdk-alpine
2
3 ENV JAVA_OPTIONS="-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap"
4
5 EXPOSE 8080
6 COPY target/*.jar app.jar
7 ENTRYPOINT ["sh", "-c", "java $JAVA_OPTIONS -jar -Dspring.profiles.active=rancher /app
      .jar"]
```

Código 3.3 – Arquivo Dockerfile da SESP

Para fins de análise do processo de Build, foi solicitado à equipe de desenvolvimento da SESP o log do processo de Build do projeto apresentado. O log está no arquivo consoleText.txt abaixo:

```
1 Branch indexing
2  > git rev-parse --is-inside-work-tree # timeout=10
3 Setting origin to http://git.seguranca.local/20666041865/pessoafonetica.git
4  > git config remote.origin.url http://git.seguranca.local/20666041865/pessoafonetica.
       git # timeout=10
5 Fetching origin...
6 Fetching upstream changes from origin
7  > git --version # timeout=10
8  > git config --get remote.origin.url # timeout=10
9 using GIT_ASKPASS to set credentials GitLab da Secretaria de Seguranca Publica
10  > git fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/* # timeout=10
11 Seen branch in repository origin/develop
12 Seen branch in repository origin/master
```

```
Seen 2 remote branches
Obtained Jenkinsfile from 022c29299fb7362b878ee5371d400ac6ea75d59b
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pessoa-fonetica_develop
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
using credential gitlab-seguranca
Cloning the remote Git repository
Cloning with configured refspecs honoured and without tags
Cloning repository http://git.seguranca.local/20666041865/pessoafonetica.git
 > git init /var/lib/jenkins/workspace/pessoa-fonetica_develop # timeout=10
Fetching upstream changes from http://git.seguranca.local/20666041865/pessoafonetica.
    git
 > git --version # timeout=10
using GIT_ASKPASS to set credentials GitLab da Secretaria de Seguranca Publica
 > git fetch --no-tags --progress http://git.seguranca.local/20666041865/
     pessoafonetica.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url http://git.seguranca.local/20666041865/pessoafonetica.
     git # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout
     =10
 > git config remote.origin.url http://git.seguranca.local/20666041865/pessoafonetica.
     git # timeout=10
Fetching without tags
Fetching upstream changes from http://git.seguranca.local/20666041865/pessoafonetica.
    git
using GIT_ASKPASS to set credentials GitLab da Secretaria de Seguranca Publica
 > git fetch --no-tags --progress http://git.seguranca.local/20666041865/
     pessoafonetica.git +refs/heads/*:refs/remotes/origin/* # timeout=10
Checking out Revision 022c29299fb7362b878ee5371d400ac6ea75d59b (develop)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 022c29299fb7362b878ee5371d400ac6ea75d59b # timeout=10
Commit message: "fix: versao homogacao"
 > git rev-list --no-walk 8065386a8f1fd216ff1eb2f640a73799a173eb84 # timeout=10
Cleaning workspace
 > git rev-parse --verify HEAD # timeout=10
Resetting working tree
 > git reset --hard # timeout=10
 > git clean -fdx # timeout=10
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] readMavenPom
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Tool Install)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
```

```
[Pipeline] { (Build Desenvolvimento)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withMaven
[withMaven] Options: []
[withMaven] Available options:
[withMaven] using JDK installation provided by the build agent
$ /bin/sh -c "which mvn"
[withMaven] Maven installation not specified in the 'withMaven()' step and not found on the build agent but 'mvnw' script found in the workspace.
[Pipeline] {
[Pipeline] sh
+ echo 'maven package...'
maven package...
[Pipeline] sh
+ mvn clean package -B -P desenvolvimento -DskipTests=true
Picked up JAVA_TOOL_OPTIONS: -Dmaven.ext.class.path="/var/lib/jenkins/workspace/pessoa-fonetica_develop@tmp/withMaven39c366ea/pipeline-maven-spy.jar" -Dorg.jenkinsci.plugins.pipeline.maven.reportsFolder="/var/lib/jenkins/workspace/pessoa-fonetica_develop@tmp/withMaven39c366ea"
[INFO] [jenkins-event-spy] Generate /var/lib/jenkins/workspace/pessoa-fonetica_develop@tmp/withMaven39c366ea/maven-spy-20201001-101554-5103757712019318538569.log.tmp ...
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------< br.gov.mt.sesp.pessoa:pessoa_fonetica >----------------
[INFO] Building pessoa_fonetica 0.0.3-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.0.0:clean (default-clean) @ pessoa_fonetica ---
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:resources (default-resources) @ pessoa_fonetica ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 3 resources
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:compile (default-compile) @ pessoa_fonetica ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 20 source files to /var/lib/jenkins/workspace/pessoa-fonetica_develop/target/classes
[INFO] /var/lib/jenkins/workspace/pessoa-fonetica_develop/src/main/java/br/com/p2d/phonetizer/StringUtils.java: Some input files use unchecked or unsafe operations.
[INFO] /var/lib/jenkins/workspace/pessoa-fonetica_develop/src/main/java/br/com/p2d/phonetizer/StringUtils.java: Recompile with -Xlint:unchecked for details.
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:testResources (default-testResources) @ pessoa_fonetica ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /var/lib/jenkins/workspace/pessoa-fonetica_develop/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:testCompile (default-testCompile) @ pessoa_fonetica ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /var/lib/jenkins/workspace/pessoa-fonetica_develop/target/test-classes
```

```
[INFO]
[INFO] --- maven-surefire-plugin:2.21.0:test (default-test) @ pessoa_fonetica ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ pessoa_fonetica ---
[INFO] Building jar: /var/lib/jenkins/workspace/pessoa-fonetica_develop/target/
    pessoa_fonetica-0.0.3-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.2.RELEASE:repackage (default) @
    pessoa_fonetica ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14.093 s
[INFO] Finished at: 2020-10-01T10:16:09-04:00
[INFO] ------------------------------------------------------------------------
[WARNING] The requested profile "desenvolvimento" could not be activated because it
    does not exist.
[INFO] [jenkins-event-spy] Generated /var/lib/jenkins/workspace/pessoa-
    fonetica_develop@tmp/withMaven39c366ea/maven-spy
    -20201001-101554-5103757712019318538569.log
[Pipeline] }
[withMaven] artifactsPublisher - Archive artifact pom.xml under br/gov/mt/sesp/pessoa/
    pessoa_fonetica/0.0.3-SNAPSHOT/pessoa_fonetica-0.0.3-SNAPSHOT.pom
[withMaven] artifactsPublisher - Archive artifact target/pessoa_fonetica-0.0.3-
    SNAPSHOT.jar under br/gov/mt/sesp/pessoa/pessoa_fonetica/0.0.3-SNAPSHOT/
    pessoa_fonetica-0.0.3-SNAPSHOT.jar
[withMaven] junitPublisher - Archive test results for Maven artifact br.gov.mt.sesp.
    pessoa:pessoa_fonetica:jar:0.0.3-SNAPSHOT generated by maven-surefire-plugin:test (
    default-test): target/surefire-reports/*.xml
[withMaven] junitPublisher - Jenkins JUnit Attachments Plugin not found, can't publish
     test attachments.Gravando o resultado dos testes
Nenhum dos relatorios de teste contem resultado
[withMaven] Jenkins Task Scanner Plugin not found, don't display results of source
    code scanning for 'TODO' and 'FIXME' in pipeline screen.
[withMaven] Publishers: Pipeline Graph Publisher: 32 ms, Generated Artifacts
    Publisher: 222 ms, Junit Publisher: 36 ms, Open Task Scanner Publisher: 1 ms
[Pipeline] // withMaven
[Pipeline] withCredentials
Masking supported pattern matches of $USERNAME or $PASSWORD
[Pipeline] {
[Pipeline] sh
+ echo 'Docker Builder...'
Docker Builder...
[Pipeline] sh
+ docker login -u **** -p **** nexus.seguranca.local:8083
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /var/lib/****/.docker/config.json
    .
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[Pipeline] sh
+ docker build -t nexus.seguranca.local:8083/sesp/pessoa_fonetica:dev .
Sending build context to Docker daemon  33.52MB

Step 1/5 : FROM openjdk:8-jdk-alpine
8-jdk-alpine: Pulling from library/openjdk
e7c96db7181b: Pulling fs layer
f910a506b6cb: Pulling fs layer
```

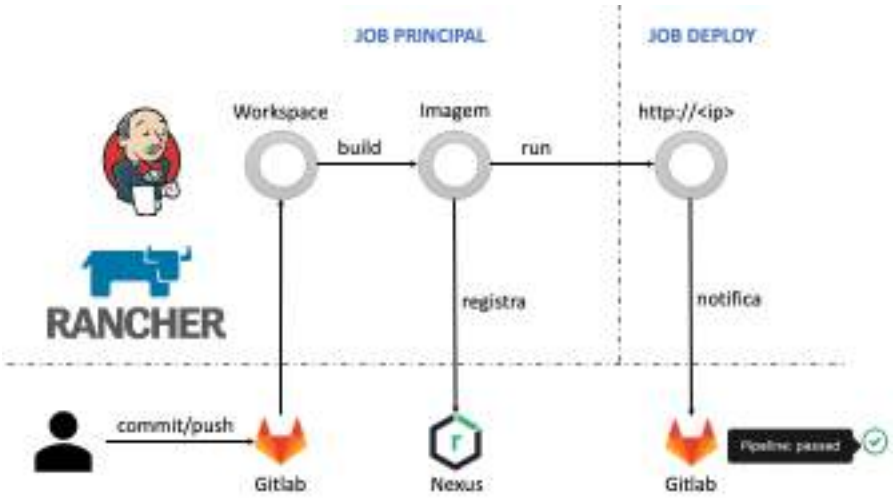
```
c2274a1a0e27: Pulling fs layer
f910a506b6cb: Verifying Checksum
f910a506b6cb: Download complete
e7c96db7181b: Verifying Checksum
e7c96db7181b: Download complete
e7c96db7181b: Pull complete
f910a506b6cb: Pull complete
c2274a1a0e27: Verifying Checksum
c2274a1a0e27: Download complete
c2274a1a0e27: Pull complete
Digest: sha256:94792824df2df33402f201713f932b58cb9de94a0cd524164a0f2283343547b3
Status: Downloaded newer image for openjdk:8-jdk-alpine
 ---> a3562aa0b991
Step 2/5 : ENV JAVA_OPTIONS="-XX:+UnlockExperimentalVMOptions -XX:+
    UseCGroupMemoryLimitForHeap"
 ---> Running in cbd9eaa81dd0
Removing intermediate container cbd9eaa81dd0
 ---> 95b8c556b6d3
Step 3/5 : EXPOSE 8080
 ---> Running in e1237126febe
Removing intermediate container e1237126febe
 ---> f12eb943be5f
Step 4/5 : COPY target/*.jar app.jar
 ---> f4891befe4d2
Step 5/5 : ENTRYPOINT ["sh", "-c", "java $JAVA_OPTIONS -jar -Dspring.profiles.active=
    rancher /app.jar"]
 ---> Running in 4360f9451588
Removing intermediate container 4360f9451588
 ---> 20fcb010eeb0
Successfully built 20fcb010eeb0
Successfully tagged nexus.seguranca.local:8083/sesp/pessoa_fonetica:dev
[Pipeline] sh
+ docker push nexus.seguranca.local:8083/sesp/pessoa_fonetica:dev
The push refers to repository [nexus.seguranca.local:8083/sesp/pessoa_fonetica]
9a29e68bff15: Preparing
ceaf9e1ebef5: Preparing
9b9b7f3d56a0: Preparing
f1b5933fe4b5: Preparing
ceaf9e1ebef5: Layer already exists
9b9b7f3d56a0: Layer already exists
f1b5933fe4b5: Layer already exists
9a29e68bff15: Pushed
dev: digest: sha256:0b1da46d0281b44a72737da49b85560a572dbe4b25b8ce50c1136c2581ab63c6
    size: 1159
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] step
set image tag from "nexus.seguranca.local:8083/sesp/pessoa_fonetica:dev" to "nexus.
    seguranca.local:8083/sesp/pessoa_fonetica:dev"
redeploy Rancher2.x workload succeed!
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] gitlabCommitStatus
[Pipeline] {
[Pipeline] echo
success
[Pipeline] }
```

```
[Pipeline] // gitlabCommitStatus
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Código 3.4 – Arquivo Log do Build da SESP

A figura 11 foi elaborada para representação do processo atual:

Figura 11 – Processo de Build Atual



3.2 Processo de Build e Gestão da Implantação

Este documento descreve o processo de build e publicação dos artefatos específicos em repositório centralizado. O processo foi elaborado considerando o ambiente atual de Integração Contínua da SESP, descrita no documento Processo de Build atual - SESP, e em práticas de integração contínua descritas pela Thougtworks e por Martin Fowler.

3.2.1 Conceitos importantes

O sucesso da build de um projeto depende de diversos fatores intrínsecos ao conceito de Integração Continua (IC). Buscou-se portanto evidenciar os conceitos relevantes à segmentação do processo de build, test e deploy de um projeto de software. Tal processo pode ser definido como a pipeline do projeto.

A definição de integração contínua é apresentada abaixo:

Integração contínua é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho com frequência, geralmente cada um faz a integração uma vez ao dia. Cada integração é verificada

por uma build automatizada (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times encontram nessa abordagem uma redução significativa nos problemas de integração e permite ao time desenvolver software coeso mais rapidamente. Martin Fowler

Além da definição, existem conceitos inerentes ao processo de integração continua que devem ser adotados pelo time em suas rotinas. Em um post da Thougtworks, é definido e descrito três conceitos principais na rotina de integração continua sendo Práticas, Como fazer e Responsabilidades do time, descritos a seguir:

3.2.1.1 Práticas

- Repositório único
- Automatizar a build
- Build auto testável
- Cada commit deve iniciar uma build
- Manter a build rápida
- Testar em um ambiente idêntico ao de produção
- Todos devem ter acesso fácil a última versão executável
- Todo mundo pode ver o que está acontecendo
- Automatizar o processo de deploy

3.2.1.2 Como fazer

- Desenvolvedor fazem checkout do código para seus workspaces
- Quando finalizarem, commitam as mudanças para o repositório
- O Servidor de IC monitora o repositório e checa se houve mudanças
- O Servidor de IC faz o build e roda os testes unitários e de integração
- O Servidor de IC libera uma release do artefato publicável para teste
- O Servidor de IC atribui um rótulo de build para a versão do código
- O Servidor de IC comunica o time do sucesso da build
- Se a build ou os testes falharem, o servidor IC comunica o time
- O time precisa corrigir problemas na primeira oportunidade
- Continue integrando e testando continuamente o projeto

3.2.1.3 Responsabilidades do time

- Enviar código frequentemente
- Não enviar código quebrado
- Não enviar código não testado
- Não vá pra casa antes de verificar se a build passou

3.2.2 A metáfora do Botão de Integração

Integrar código desenvolvido por diversos desenvolvedor ao longo do tempo não é algo trivial, especialmente pelo fato desses processos possuírem diversas tarefas repetitivas e maçantes. Essa complexidade ainda é expandida a medida que outros processos são inseridos como testes, deploy e notificação. A metáfora do **Botão de Integração** de Duvall et al. (2007), ilustrada na figura 12 trata justamente da necessidade de automatizar o processo de integração a partir de um processo que contemple as principais fases da entrega de um software.

Figura 12 – Metáfora do **Botão de Integração**

Planejar e definir o processo de entrega de um produto é fator crucial em ambientes de integração continua. A ideia é que esse planejamento, aliado as boas práticas de desenvolvimento de software, integração e entrega contínua, melhoram a qualidade do software e o feedback com a equipe, além de reduzir os riscos da entrega.

3.2.3 Ambiente de Integração contínua

Para a análise e sugestão da ferramenta de integração continua, foram considerados aspectos importantes do ambiente de integração continua e as dificuldades comuns ao gerenciamentos da infraestrutura de Tecnologia da Informação.

3.2.3.1 Tamanho e complexidade do ambiente

Um dos grandes desafios em se gerenciar ambientes de tecnologia da informação é a quantidade de hardware, serviços, configurações, atualizações, backups, etc. Esse cenário acaba sendo agravado em ambientes onde os recursos humanos não são suficientes para suprir o volume de atividades.

A redução do número de atividades relativas a qualquer uma das atividades citadas anteriormente, implica em redução de esforços, economia de tempo e recursos. Portanto, a titulo de exemplo, os seguintes itens foram considerados na análise das ferramentas de integração continua a serem utilizadas:

1. Diminuição do número de servidores
2. Economia de memória, armazenamento e processamento
3. Diminuição de softwares gerenciados
    a) Plugins (91 instalados para simular o ambiente no Jenkins) - Plugin hell
    b) Atualizações de versão e de segurança
    c) Clients
    d) etc
4. Diminuição da infraestrutura de rede
    - IP
    - DNS
    - NTP
    - Firewall
    - etc

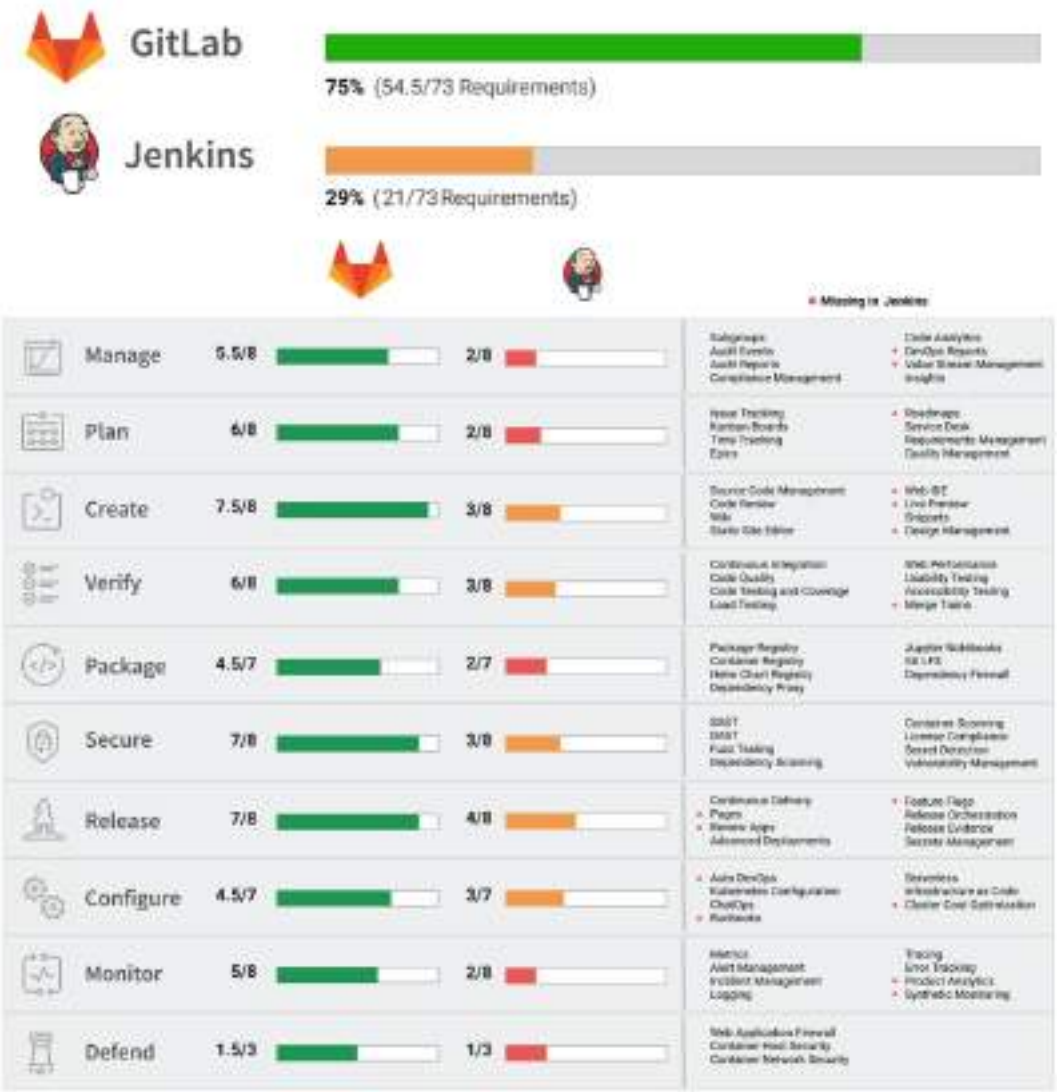
3.2.3.2 Integrações

As integrações entre serviços também constituem fator de complexidade no gerenciamento de recursos de tecnologia da informação.

1. Autenticação e Autorização;
2. Gerenciamento de código e automação da build no mesmo sistema;
3. Status da pipeline;
4. Webhooks;
5. Gitlab mais produtivo com projetos menores - exemplo: microsserviços.

3.2.3.3 Comparação de ferramentas de integração contínua

Figura 13 – Comparação de ferramentas de integração contínua



3.2.4 Modelo de build e armazenamento dos artefatos

3.2.4.1 Stages

build

- **Descrição:** Nessa stage é executado o processo de build do Maven (Clean package)
- **Jobs:** build_maven

deploy

- **Descrição:** Nessa stage é executado o processo de deploy do Maven
- **Jobs:** deploy_maven

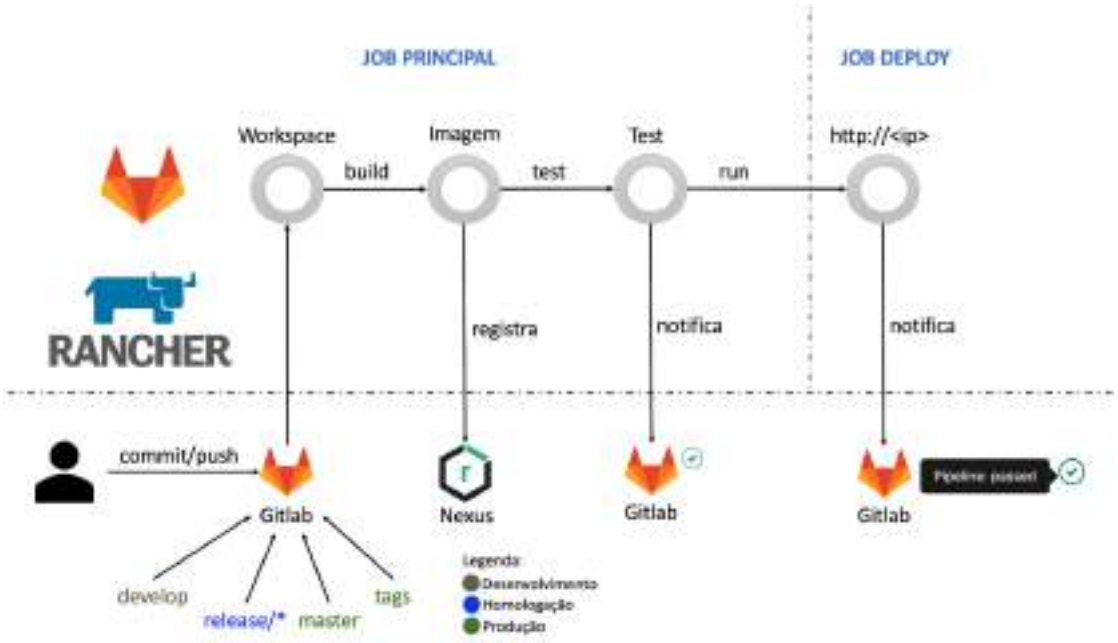
build_docker

- **Descrição:** Nessa stage é executado o processo de build e registro da imagem Docker no repositório Nexus.
- **Jobs:** build_desenv, build_homologa, build_prod

3.2.4.2 Branches e o processo de Build

- A branch **develop** vai iniciar a pipeline para testar a build sempre que houver um commit.
- Quando houver funcionalidades suficientes é gerado uma **release/*** da branch **develop**.
- A branch **release/*** após criada irá iniciar a pipeline e gerar uma imagem Docker de homologação com a versão da release.
- Após a conclusão da **release/*** será feito o merge na branch **master** e a criação de uma **tag/***.
- Será iniciado o processo de build da imagem Docker de produção com a versão da **tag/***.
- Após o merge na **branch** master, será realizado o deploy dos artefatos Maven no Nexus.

Figura 14 – Branches e o processo de Build



3.2.4.3 Nomenclatura de imagens Docker

Desenvolvimento

- **develop** : Última imagem docker gerada da branch develop

Homologação

- **hmg** : Imagem docker gerada a partir da última release. Semelhante a imagem latest
- **release-*-*-*** : Imagem docker gerada a partir da última release

Produção

- **latest** : Última imagem docker gerada da branch master
- **tags-*-*-*** : Imagem docker gerada a partir de uma tag

Apenas as imagens das tags e releases ficam armazenadas, ou seja, não serão excluídas após novo build.

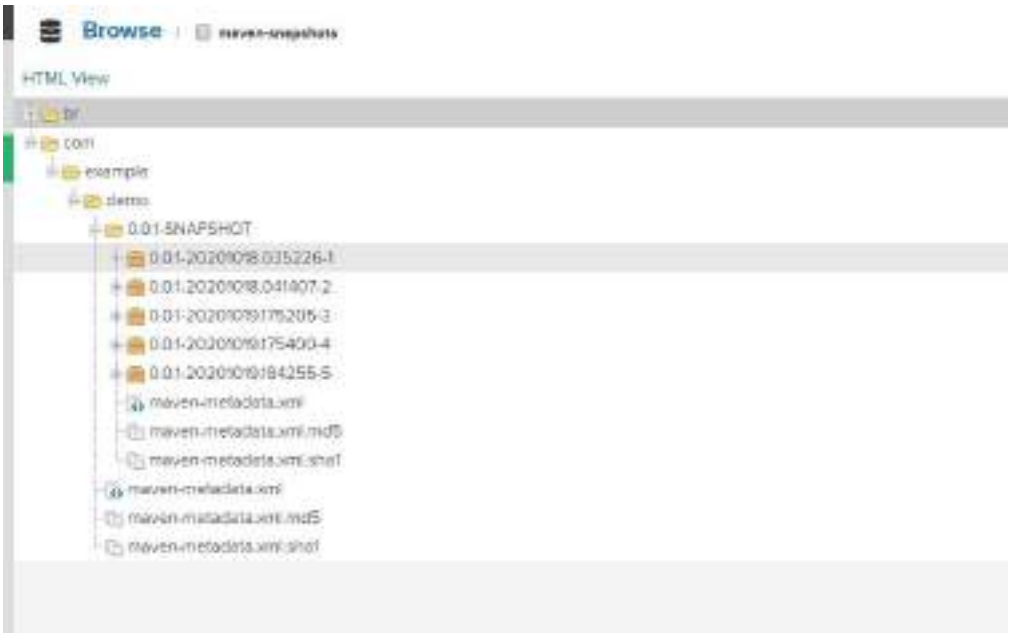
3.2.4.4 Nomenclatura de pacotes

Apenas os jars, wars, etc. gerados pela branch master vão para o Nexus.

- Última imagem docker gerada da branch develop
- 1.2.0-SNAPSHOT
- Pacotes expirarão em 10 dias

Figura 15 – Exemplos de imagens geradas

Figura 16 – Exemplos de pacotes gerados



3.3 Configurando o processo de Build no Gitlab

3.3.1 GitLab CI / CD

CI / CD é uma ferramenta incorporada ao GitLab para desenvolvimento de software por meio de metodologias contínuas:

- Integração Contínua (CI)
- Entrega Contínua (CD)
- Implantação Contínua (CD)

A integração contínua funciona enviando pequenos pedaços de código para a base de código do seu aplicativo hospedada em um repositório Git e, para cada push, execute um pipeline de scripts para construir, testar e validar as alterações de código antes de mesclá-los no branch principal. Esse recurso é denominado *Auto DevOps* no Giltab.

A entrega e implantação contínuas consistem em uma etapa adicional de CI, implantando seu aplicativo para produção a cada push para a branch padrão do repositório (Develop, Homologação, Produção, etc).

Essas metodologias permitem detectar bugs e erros no início do ciclo de desenvolvimento, garantindo que todo o código implantado para produção esteja em conformidade com os padrões de código que você estabeleceu para seu aplicativo.

3.3.2 Primeiros passos

O GitLab CI / CD é configurado por um arquivo chamado .gitlab-ci.yml colocado na raiz do repositório. Este arquivo cria um pipeline, que é executado para alterações no código no repositório. Os pipelines consistem em um ou mais estágios executados em ordem e cada um pode conter uma ou mais tarefas executadas em paralelo. Esses jobs (ou scripts) são executados pelo agente GitLab Runner .

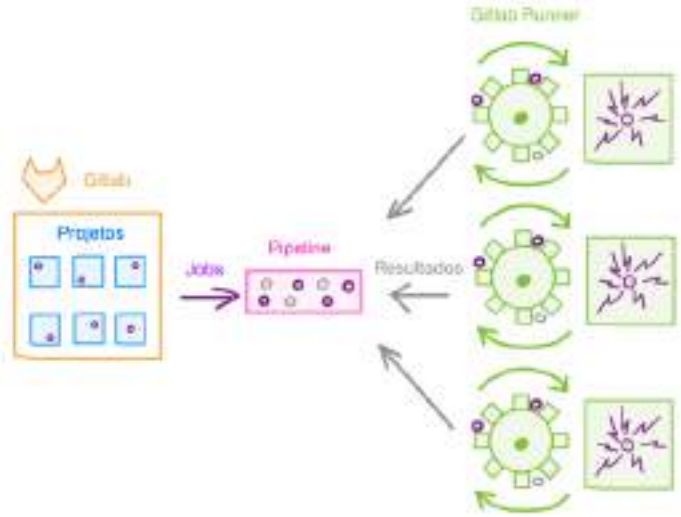
Em um ambiente *"Self-hosted"*é necessário a configuração de um Runner para executar nossas pipelines.

3.3.3 Configurando

No GitLab CI / CD, os *runners* executam o código definido em **.gitlab-ci.yml**. Um *runner* é um agente leve e altamente escalonável que pega um job de CI por meio da API do coordenador do GitLab CI / CD, executa o job e envia o resultado de volta à instância do GitLab.

Os *runners* são Runnnerscriados por um administrador e são visíveis na IU do GitLab. Os *runners* podem ser específicos para determinados projetos ou disponíveis para todos os projetos.

Figura 17 – Runnners



3.3.4 Instale o GitLab Runner

O GitLab Runner pode ser instalado e usado em GNU / Linux, macOS, FreeBSD e Windows. Existem três maneiras de instalá-lo. Utilizando Docker, baixando o binário manualmente ou utilizando o repositório de pacotes rpm/deb.

Para nossa configuração vamos instalar como um docker o primeiro passo é criar um volume para que nossas informações não sejam perdidas quando o runner for reiniciado.

1. Crie o volume do Docker:

```
docker volume create gitlab-runner-config
```

Código 3.5 – Volume do Docker

2. Inicie o contêiner Runner usando o volume que acabamos de criar

```
docker run -d --name gitlab-runner --restart always \

  -v /var/run/docker.sock:/var/run/docker.sock \

  -v gitlab-runner-config:/etc/gitlab-runner \

  gitlab/gitlab-runner:latest
```

Código 3.6 – Inicie o contêiner Runner

3.3.5 Registrando o Runner

Antes de registrar um runner, você precisa primeiro:

Instala-lo em um servidor separado daquele onde o GitLab está instalado

Obtenha um token:

- Para um **runner** compartilhado , peça a um administrador que vá para a área administrativa do GitLab e clique em Visão geral - Runners

- Para um Runners de grupo , vá para Configurações - CI / CD e expanda a seção Runners
- Para um runner específico do projeto , vá para Configurações - CI / CD e expanda a seção Runners

Para registrar um *runner* usando um contêiner Docker com montagem de volumes como no nosso exemplo.

1. Execute o comando de registro:

```
docker run --rm -it -v gitlab-runner-config:/etc/gitlab-runner gitlab/gitlab-
    runner:latest register
```

Código 3.7 – Comando de registro

2. Insira o URL da instância do GitLab (também conhecido como gitlab-ci coordinator URL).
3. Digite o token que você obteve para registrar o runner.
4. Insira uma descrição para o runner. Você pode alterar esse valor posteriormente na interface do usuário do GitLab.
5. Insira as tags associadas ao runner , separadas por vírgulas. Você pode alterar esse valor posteriormente na interface do usuário do GitLab.

Fornece o executor do *runner*. Para a maioria dos casos de uso, digite docker.
Ou use o comando estruturado:

```
docker run --rm -v /srv/gitlab-runner/config:/etc/gitlab-runner gitlab/gitlab-runner
    register \
  --non-interactive \
  --executor "docker" \
  --docker-image alpine:latest \
  --url "https://gitlab.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --description "docker-runner" \
  --tag-list "docker,aws" \
  --run-untagged="true" \
  --locked="false" \
  --access-level="not_protected"
```

Código 3.8 – Comando estruturado

O **.gitlab-ci.yml** define a estrutura e a ordem dos pipelines e determina:

- O que executar usando o GitLab Runner.
- Que decisões tomar quando forem encontradas condições específicas. Por exemplo, quando um processo é bem-sucedido ou falha.

A configuração da pipeline começa com jobs. Os jobs são o elemento mais fundamental de um **.gitlab-ci.yml**.
Os jobs são:

1. Definido com restrições indicando sob quais condições eles devem ser executados.
2. Elementos de nível superior com um nome arbitrário e devem conter pelo menos a script cláusula.
3. Não limitado em quantos podem ser definidos.

Por exemplo:

```
docker run --rm -v /srv/gitlab-runner/config:/etc/gitlab-runner gitlab/gitlab-runner
    register \
  --non-interactive \
  --executor "docker" \
  --docker-image alpine:latest \
  --url "https://gitlab.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --description "docker-runner" \
  --tag-list "docker,aws" \
  --run-untagged="true" \
  --locked="false" \
  --access-level="not_protected"
```

Código 3.9 – Exemplo de jobs

O exemplo acima é a configuração CI / CD mais simples possível com dois jobs separados, em que cada um dos jobs executa um comando diferente. É claro que um comando pode executar código diretamente (**/configure;clean;package install**) ou executar um script (**test.sh**) no repositório.

Os jobs são escolhidos pelos runners e executados dentro do ambiente do runner. O importante é que cada job seja executado independentemente um do outro.

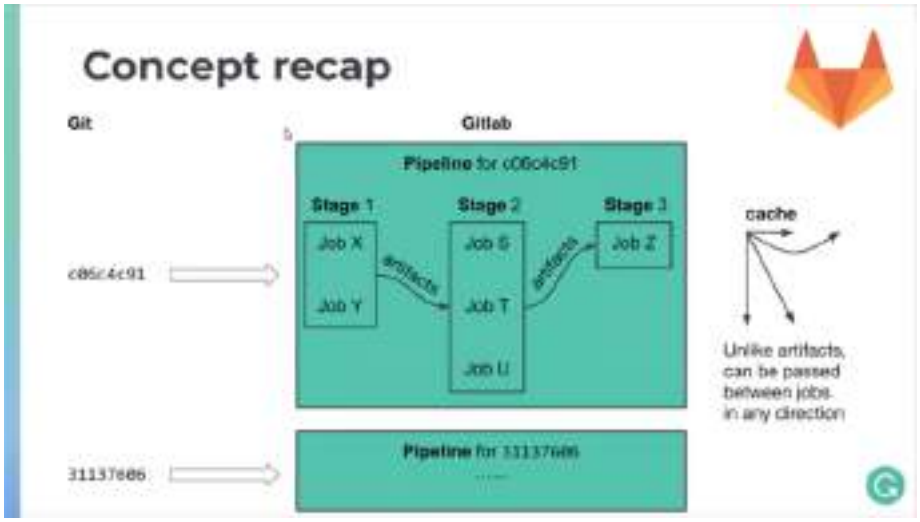
3.3.6 Stages

Stages são usado para definir estágios que contêm jobs e é definido globalmente para o pipeline.

A especificação de *stages* permite ter pipelines de múltiplos estágios flexíveis. A ordem dos elementos em *stages* define a ordem da execução dos jobs:

- Jobs do mesmo estágio são executados em paralelo.
- Os jobs do próximo estágio são executados depois que os jobs do estágio anterior são concluídos com sucesso.

Figura 18 – Stages



Vamos considerar o exemplo a seguir, que define 3 estágios: build, test, deploy
Primeiro, todas as tarefas de build são executadas em paralelo.

- Se todos os jobs de build forem bem-sucedidos, os jobs test serão executados em paralelo.
- Se todos os jobs de test forem bem-sucedidos, os jobs deploy serão executados em paralelo.
- Se todos os jobs forem deploy bem-sucedidos, o commit será marcado como passed.
- Se qualquer uma das tarefas anteriores falhar, a confirmação será marcada como failed e nenhuma tarefa de estágio posterior será executada.

Para o **deploy** de micro-serviços desenvolvidos em Spring Boot desenvolvemos um arquivo de **gitlab-ci.yml**

```
image: docker:stable



services:
docker:dind



variables:
```

```
  MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/
      repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.
      Slf4jMavenTransferListener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -
      Djava.awt.headless=true"
  MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=
      true -DdeployAtEnd=true"
  CONTAINER_IMAGE: fabricadesoftwareifmt.com.br/registry/$CI_PROJECT_PATH

stages:
build
deploy
build_docker


build_maven:

  stage: build
  image: maven:3.6.3-openjdk-11-slim
  cache:
    paths:
.m2/repository
  script:
'mvn $MAVEN_CLI_OPTS clean package -s $mavenSettings'
  artifacts:
      expire_in: 10 day
      paths:
target/*.jar
  only:
    refs:
develop
/^release/[0-9]+(?:.[0-9]+)+$/
tags



deploy_maven:
  stage: deploy
  image: maven:3.6.3-openjdk-11-slim
  cache:
    paths:
target/
  script:
'mvn $MAVEN_CLI_OPTS deploy -s $mavenSettings'
  only:
    refs:
master



##DESENVOLVIMENTO
build_desenv:
  stage: build_docker
  image: docker:latest
  cache:
    paths:
target/

before_script:
docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
  script:
docker pull $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME || true
```

```
docker build --cache-from $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME --tag $CONTAINER_IMAGE:
    $CI_COMMIT_REF_NAME .
docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME
  only:
    refs:
develop



##HOMOLOCAGAO
build_homologa:
  stage: build_docker
  image: docker:latest
  cache:
    paths:
target/
  before_script:

docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
  script:
docker pull $CONTAINER_IMAGE:hmg || true
docker build --cache-from $CONTAINER_IMAGE:hmg --tag $CONTAINER_IMAGE:$
    CI_COMMIT_REF_SLUG --tag $CONTAINER_IMAGE:hmg .
docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_SLUG
docker push $CONTAINER_IMAGE:hmg
  only:
    refs:
/^release/[0-9]+(?:.[0-9]+)+$/

##PRODUCAO
build_prod:
  stage: build_docker
  image: docker:latest
  before_script:

docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
  script:
docker pull $CONTAINER_IMAGE:latest || true
docker build --cache-from $CONTAINER_IMAGE:latest --tag $CONTAINER_IMAGE:$
    CI_COMMIT_REF_SLUG --tag $CONTAINER_IMAGE:latest .
docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_SLUG
docker push $CONTAINER_IMAGE:latest

  only:
    refs:
tags
```

Código 3.10 – Arquivo gitlab-ci.yml

3.3.7 Referências

- Install GitLab Runner — GitLab[1]
- Registering runners — GitLab[2]
- GitLab CI/CD — GitLab[3]

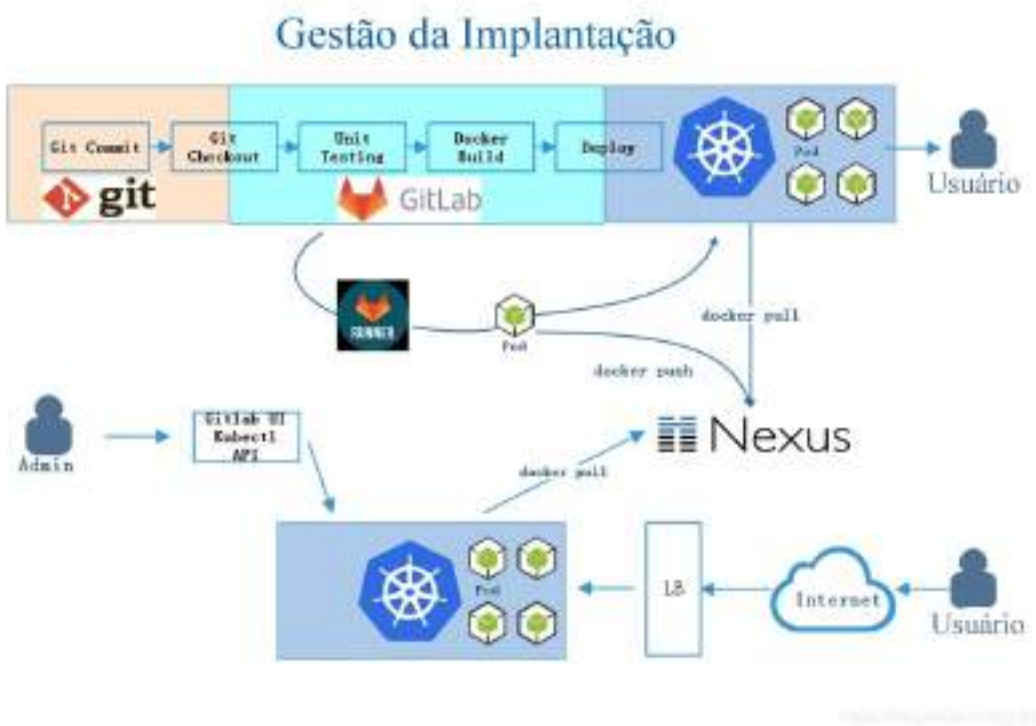
[1] https://docs.gitlab.com/runner/install/
[2] https://docs.gitlab.com/runner/register/
[3] https://docs.gitlab.com/ee/ci/

- https://docs.gitlab.com/ee/ci/yaml/README.html

3.4 Processo de Gestão da Implantação

Esta seção descreve o processo de gestão da implantação. O processo foi elaborado considerando o ambiente de Integração Contínua da SESP. A figura 19 ilustra esse ambiente.

Figura 19 – Gestão de Implantação



3.4.1 Configurando Rancher e Kubernetes

O ambiente de implantação foi concebido em arquitetura de containers utilizando o Docker Compose. O arquivo yaml com as configurações está disponível no seguinte projeto do Gitlab:

Para configurar o Rancher basta adicionar o seguinte trecho de código:

```
rancher:
        image: rancher/rancher:stable
        container_name: rancher-container
        networks:
            - fabrica-software-network
        volumes:
            - ./data/rancher:/var/lib/rancher
```

Código 3.11 – Arquivo de Configuração do Rancher

Para a publicação do Rancher na internet, foi realizada a configuração do NGINX para expor os serviços que estão rodando em container Docker. O trecho de código apresenta a configuração realizada no NGINX:

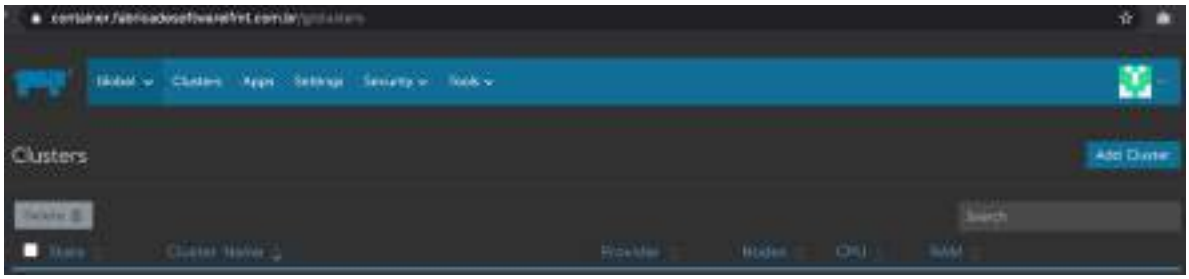
```
map $http_upgrade $connection_upgrade {
    default Upgrade;
    ''      close;
}

server {
    listen 443 ssl;
    server_name container.fabricadesoftwareifmt.com.br;

    ssl_certificate /etc/letsencrypt/live/fabricadesoftwareifmt.com.br/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/fabricadesoftwareifmt.com.br/privkey.pem
        ;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    location / {
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://rancher-container;
        proxy_http_version 1.1;
  proxy_set_header Connection $connection_upgrade;
        proxy_set_header Upgrade $http_upgrade;
    }
}

server {
    listen 80;
    server_name container.fabricadesoftwareifmt.com.br;
    server_tokens off;
    client_max_body_size 0;

    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }
}
```

Código 3.12 – Arquivo de Configuração do NGINX

3.4.1.1 Configuração do Cluster Kubernetes

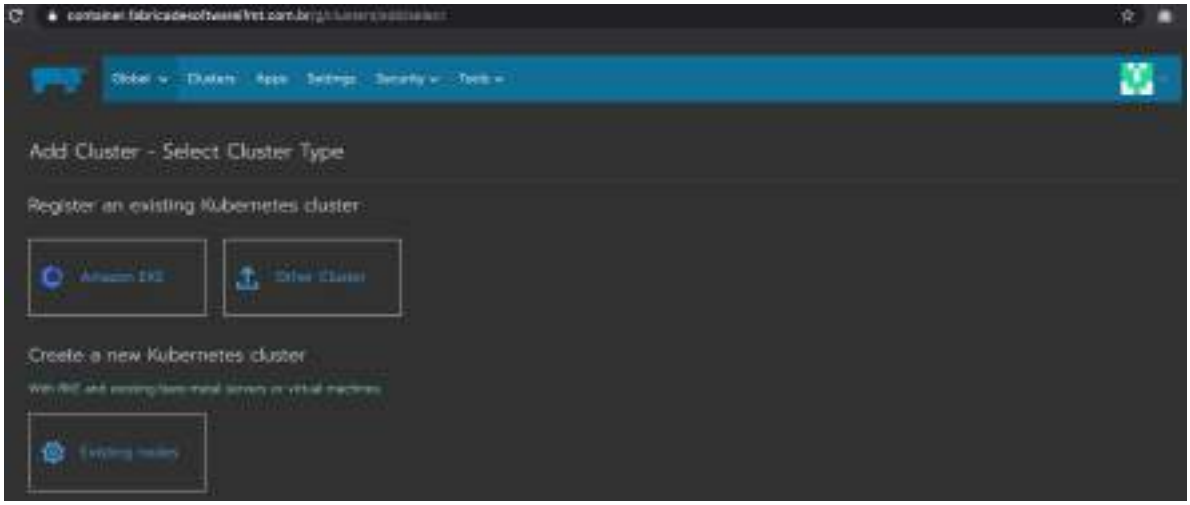
O Rancher possibilita a automatização da instalação de um cluster Kubernetes, basta ir no dashboard principal e adicionar um Cluster, conforme figura 20.

Figura 20 – Dashboard principal



A partir de um nó (servidor) existente, é possível iniciar uma instalação do Kubernetes. Para isso, a opção utilizada é a Existing nodes, conforme figura 21.

Figura 21 – Instalação do Kubernetes

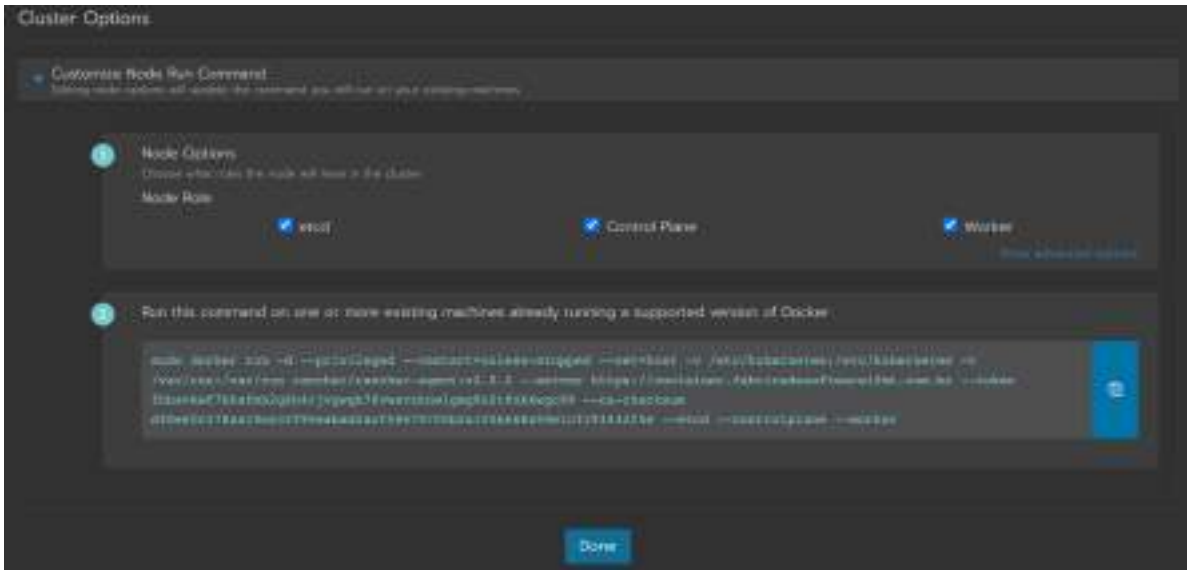


A partir dessa etapa, no cenário configurado para o ambiente do protótipo, as seguintes configurações foram realizadas:

- Nome de cluster
- Registro privado - Foi informado o registry do Nexus para utilização das imagens Docker
- Habilitado o ingress do NGINX para exposição dos serviços na Internet
- A opção Authorized Endpoint foi habilitada pois o Gitlab necessita se conectar ao Kubernetes

Na tela ilustrada na figura 22 basta informar as opções do cluster e copiar o comando disponibilizado. Esse link deverá ser copiado e executado em um servidor onde o Kubernetes será instalado.

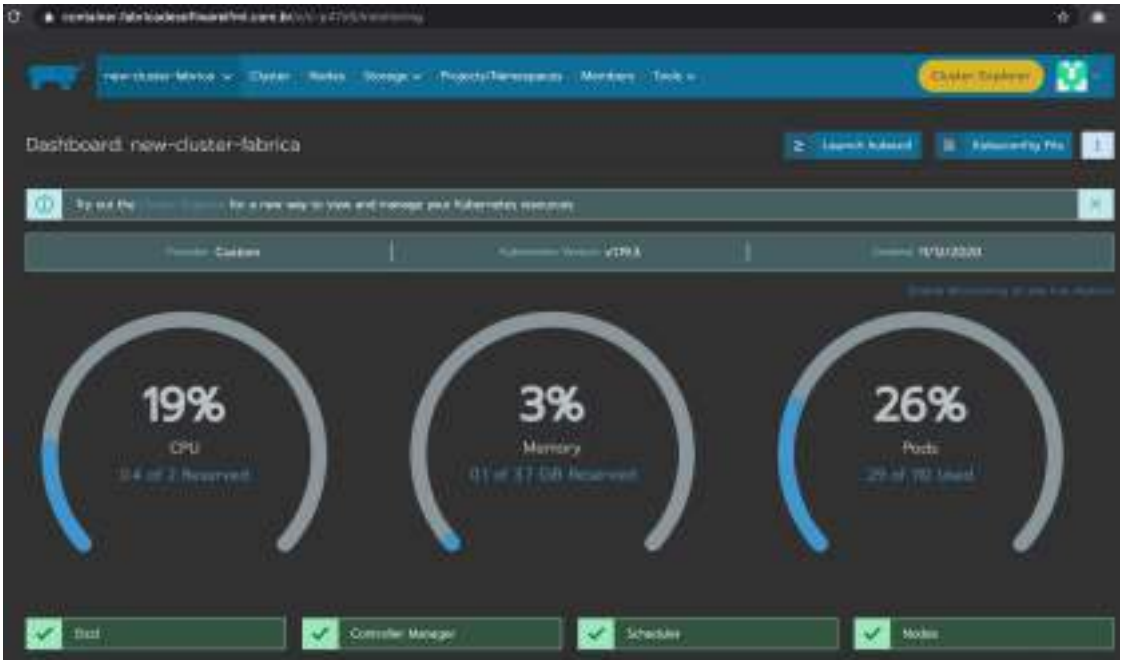
Figura 22 – Instalação do Kubernetes, continuação...



Importante destacar que esse processo fará o vinculo com o Rancher, ou seja, será gerenciado pelo Rancher.

Após a conclusão da instalação do cluster Kubernetes, é possível acessar as configurações na tela inicial do Rancher, conforme ilustrado na figura 23.

Figura 23 – Configurações na tela inicial do Rancher



Essa tela possibilita a execução de um terminal com o kubectl para gestão do Kubernetes e também acessar as configurações que podem ser copiadas para execução remota de comandos do kubectl.

Para acessar essas configurações, clique no botão **Kubeconfig File**. Siga as instruções de configuração no painel apresentado.

3.4.1.2 Certificados

O Rancher permite a configuração de certificados digitais que serão utilizados na publicação de serviços. Para o ambiente do protótipo, foi utilizado o certificado da CA Let's Encrypt com os seguintes domínios configurados:

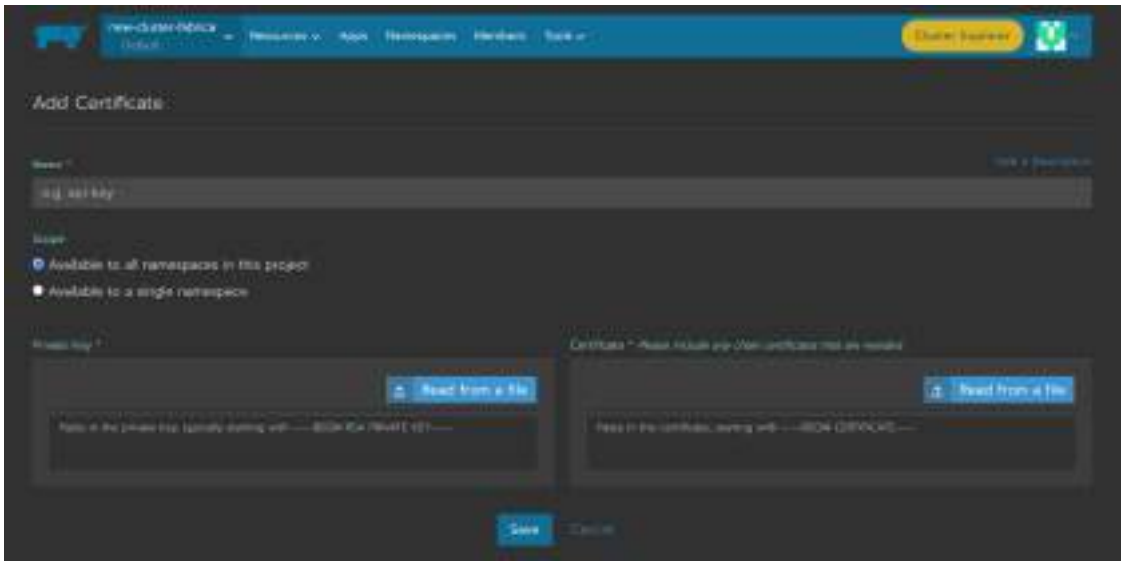
- fabricadesoftwareifmt.com.br (CN)[4]
- container.fabricadesoftwareifmt.com.br [5]
- prototipo.fabricadesoftwareifmt.com.br [6]
- registry.fabricadesoftwareifmt.com.br [7]

Utilizando o Namespace Default, acesse o menu Resources - Secrets. Após abrir a tela inicial, clique na aba Certificates.

[4] https://fabricadesoftwareifmt.com.br/
[5] https://container.fabricadesoftwareifmt.com.br/
[6] https://prototipo.fabricadesoftwareifmt.com.br/
[7] https://registry.fabricadesoftwareifmt.com.br/

Para adicionar um certificado basta colar o código Certificado e da chave privada conforme abaixo, conforme ilustrado na figura 24.

Figura 24 – Adicionar Certificado



Os certificados podem estar disponíveis a um ou a todos os namespaces.

3.4.1.3 Load balancer e Ingress

O Load Balancer é um recurso que possibilita a publicação dos pods do cluster Kuberntes para o mundo externo, ou seja, a comunicação de um computador na rede interna ou na internet com os serviços publicados no cluster.

Para configurar um Load balancer, utilizando o Namespace *Default*, acesse o menu *Resources - Workloads*. Após abrir a tela inicial, clique na aba *Load Balancing*, conforme figura 25.

Figura 25 – Load Balancing

Para o ambiente do protótipo, foi configurado um *Ingress* com target para o pod **gerenciamento-roubo-web** na porta 8080, conforme figura 26.

Logo a seguir, o certificado do Let's Encrypt foi vinculado a esse serviço, conforme 27.

3.4.1.4 Configurando Gitlab

Acesso o menu Kubernetes do contexto desejado e vamos fazer a configuração da integração com o Kubernets, selecione "Integrate with a cluster certificate" conforme figura 28.

Figura 26 – configuração do Ingress

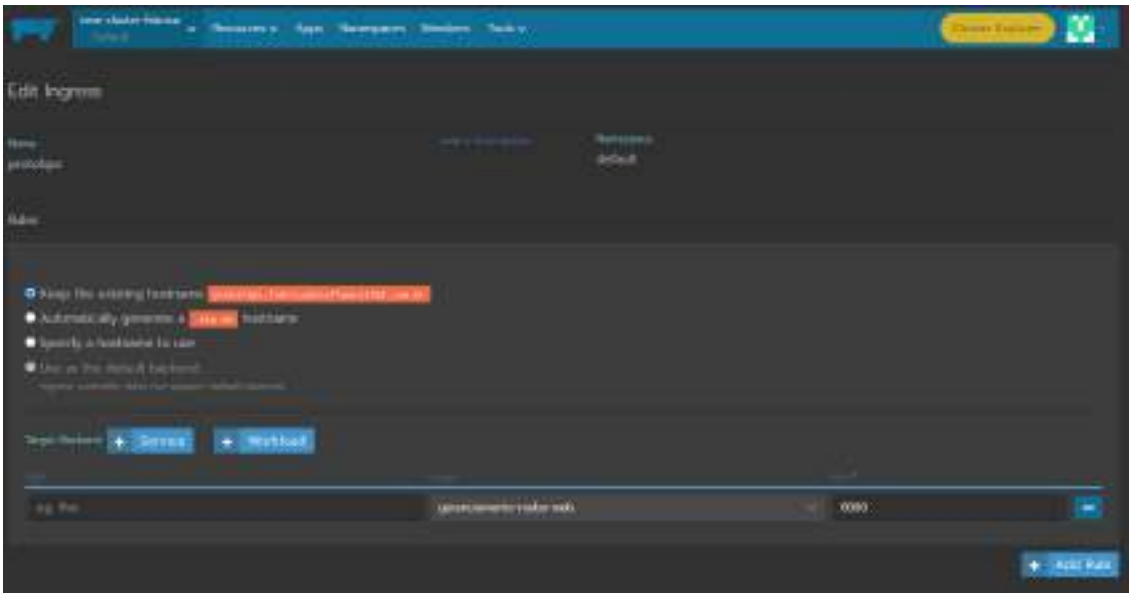
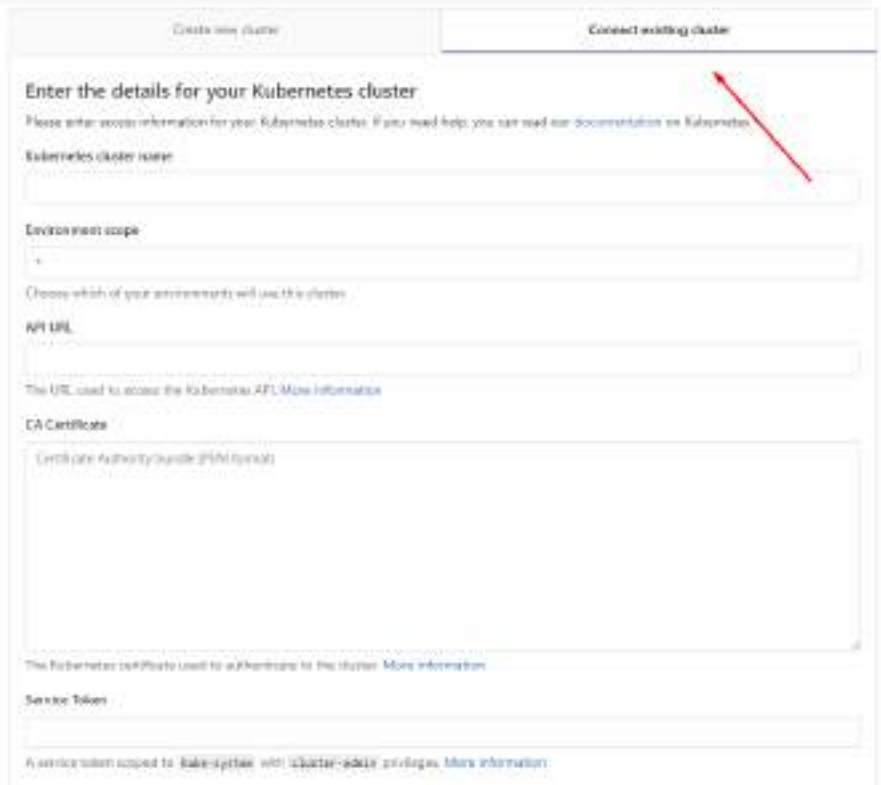


Figura 27 – Vinculação do Certificado

Figura 28 – Configurando Gitlab e Kubernetes

Figura 29 – Configurando vínculo com cluster



Selecione a segunda aba para vincular um cluster existente, conforme ilustrado na figura 29.

Dentro do formulário vamos informar um nome para identificar o Cluster, determinar um escopo para esse cluster, a URL da api do cluster Kubernetes, o certificado CA e o Token de serviço, conforme ilustrado na figura 30.

As informações solicitadas podem ser encontradas no arquivo Kubeconfig do Cluster que estamos integrando, sendo a API URL na propriedade "server" o certificado codificado em base64 na propriedade "certificate-authority-data" e Service Token na propriedade token, conforme ilustrado na figura 31.

3.4.1.5 Pipeline de Implantação

Para configurarmos o acesso ao cluster Kubernetes precisamos incialmente especificar algumas configurações de autenticação. Quando você estiver em seu repositório, use o menu à esquerda para abrir as Configurações - CI / CD - Variáveis de ambiente.

Vamos adicionar a seguinte configuração:

CERTIFICATE_AUTHORITY_DATA: Essa é a configuração de CA para o cluster Kubernetes.

SERVER: Esse é endpoint da API Kubernetes para nosso cluster.

USER_TOKEN: Esse é o token do usuário que usaremos para se conectar ao cluster do Kubernetes.

Nos arquivos **.gitlab-ci.yml**, desenvolvidos na última sprint, consta a configuração para a criação de imagens docker. Agora vamos trabalhar para publicá-las no cluster Kubernetes gerenciado pelo Rancher.

Figura 30 – Configurando vínculo com cluster

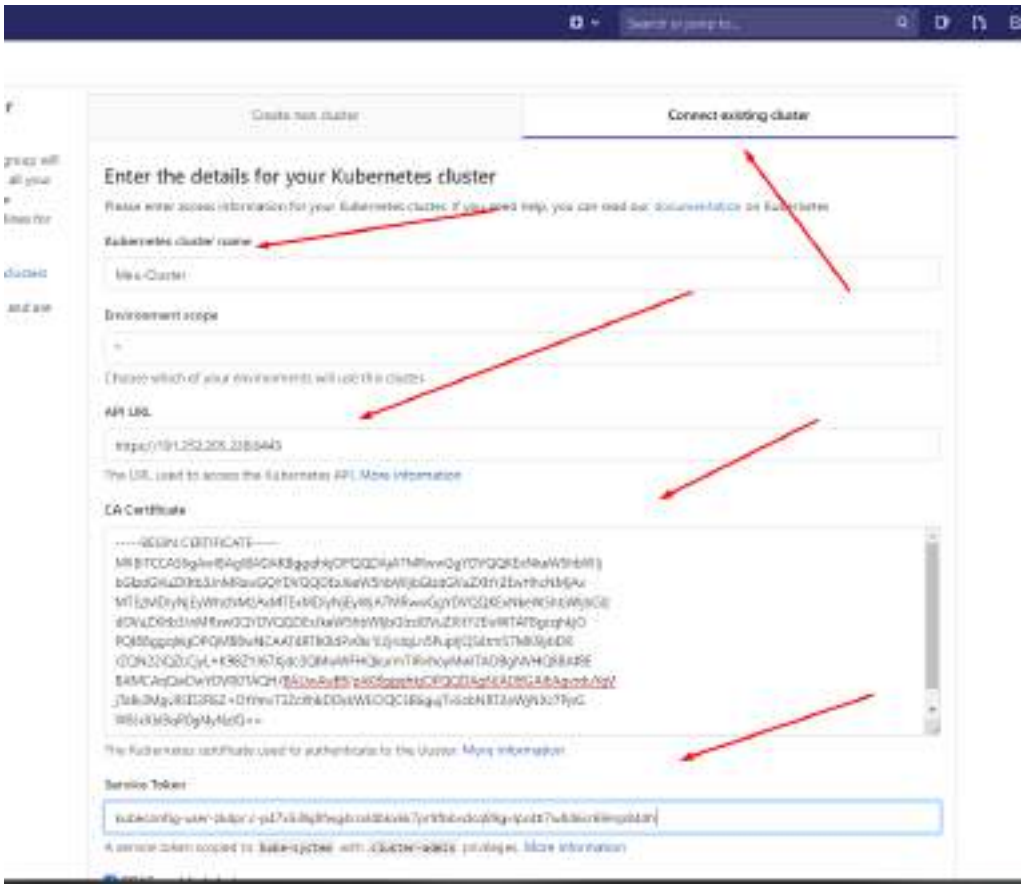


Figura 31 – Arquivo Kubeconfig do Cluster

Vamos criar uma implantação (*deployment*) do Kubernetes. Este é um recurso do Kubernetes que envolve os contêineres do Docker e gerencia seu ciclo de vida. Isso garante sua publicação, a reinicialização dos contêineres se eles forem interrompidos e também que a quantidade certa de contêineres esteja em execução. Ele também pode realizar atualizações contínuas e usar verificações de integridade para verificar se os contêineres continuam funcionando.

Crie um novo arquivo chamado **deployment.yaml** e adicione o seguinte conteúdo:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: registro-geral
  labels:
    app: registro-geral
spec:
  replicas: 3
  selector:
    matchLabels:
      app: registro-geral
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 33%
  template:
    metadata:
      labels:
        app: registro-geral
    spec:
      containers:
        - name: registro-geral-container
          image: fabricadesoftwareifmt.com.br/registry/sesp-mt/registro-geral:<VERSION
              >
          ports:
            - containerPort: 8080
```

Código 3.13 – Arquivo deployment.yam

Neste exemplo estamos configurando a publicação do microsserviço **Registro Geral** com 3 pods, porém, esse script pode ser generalizado utilizando o mesmo conceito para outros serviços, bastando alterar o nome do mesmo e o endereço da imagem utilizada.

Agora vamos fazer a implantação por meio do GitLab e, em seguida, executar essa implantação. Adicione o novo estágio a seguir ao final do arquivo **.gitlab-ci.yml**:

```
deploy_rancher_develop:
  stage: deploy_rancher
  image: dtzar/helm-kubectl
  script:
    - kubectl config set-cluster k8s --server="${SERVER}"
    - kubectl config set clusters.k8s.certificate-authority-data ${
        CERTIFICATE_AUTHORITY_DATA}
    - kubectl config set-credentials gitlab --token="${USER_TOKEN}"
    - kubectl config set-context default --cluster=k8s --user=gitlab
    - kubectl config use-context default
    - sed -i "s/<VERSION>/develop/g" deployment.yaml
    - kubectl apply -f deployment.yaml
  only:
    refs:
      - develop
```

```
15
16
17 deploy_rancher_hmg:
18   stage: deploy_rancher
19   image: dtzar/helm-kubectl
20   script:
21     - kubectl config set-cluster k8s --server="${SERVER}"
22     - kubectl config set clusters.k8s.certificate-authority-data ${
         CERTIFICATE_AUTHORITY_DATA}
23     - kubectl config set-credentials gitlab --token="${USER_TOKEN}"
24     - kubectl config set-context default --cluster=k8s --user=gitlab
25     - kubectl config use-context default
26     - sed -i "s/<VERSION>/hmg/g" deployment.yaml
27     - kubectl apply -f deployment.yaml
28   only:
29     refs:
30       - /^release/[0-9]+(?:.[0-9]+)+$/
```

Código 3.14 – Novo estágio no arquivo .gitlab-ci.yml

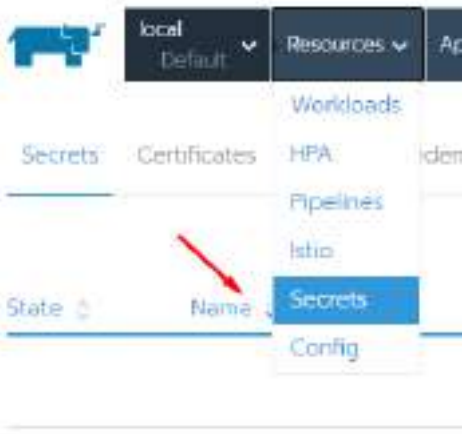
Envie essas alterações para seu repositório GitLab (**git commit && git push**). A *pipeline* do Gitlab é iniciada automaticamente e a nova etapa de implantação fará a implantação no Kubernetes. Verifique o Rancher para ver o novo pod em execução e seus detalhes.

Com essas configurações criamos uma *pipeline* de implantação totalmente automatizada utilizando o Gitlab e o Kubernetes. Embora a *pipeline* não contemple nenhum teste automatizado (unidade, integração, etc) ou promoções do aplicativo em diferentes ambientes, ela apresenta a ideia sobre a construção de uma pipeline com recursos para implantação dos projetos no Kubernetes.

3.4.1.6 Configuração de Variáveis

Variáveis que contem dados sensíveis podem ser armazenadas utilizando o recurso de secrets do Rancher, conforme ilustrado na figura 32.

Figura 32 – Variáveis de ambiente



CAPÍTULO 4

Ambiente de Infraestrutura

Este capítulo apresenta as configurações do ambiente de infraestrutura organizado para o desenvolvimento da Integração Contínua no setor de desenvolvimento de software da Secretaria de Segurança Público do Estado de Mato Grosso. Esse ambiente também serviu de suporte para a implementação do protótipo funcional.

4.1 Ambiente inicial

Para a definição da máquina virtual foi levantado os requisitos de hardware para os serviços iniciais do setup do projeto conforme tabela 2:

Tabela 2 – Serviços do Ambiente

Binário	CPU	Memória Ram (GB)	Espaço em disco (GB)
Docker	1	0,512	6
Jenkins	2	0,256	1,024
Postgree	2	0,512	0,512
Ubuntu 18.04	1	0,512	2,5
JDK	1	0,128	0,124
Maven	1	0	0,5
Gitlab	2	3,75	4

4.1.1 Máquinas virtuais do projeto

Características da máquina virtual **VPS24175**

HOSTNAME: @srv.fabricadesoftwareifmt.com.br

OS: Ubuntu 18.04 bionic

Kernel: x86_64 Linux 4.15.0-39-generic

Packages: 606

CPU: Intel Xeon Gold 5218 2x

RAM: 3414MiB / 3913MiB

A máquina virtual está publicada com o endereço de IP 191.252.202.213 e com o domínio fabricadesoftwareifmt.com.br

- Acesso SSH liberado na porta 22

4.1.2 Ambiente de contêineres

O projeto está versionado no Gitllab. O endereço [1] deve ser usado para clonar a infraestrutura.

Este projeto depende da configuração do Certbot[2] e do LetsEncrypt[3] para publicar o certificado para o domínio fabricadesoftwareifmt.com.br[4]. Portanto, após clonagem do projeto é necessário executar o script **init.letsencrypt.sh**. Os seguintes passos devem ser realizados:

1. Clonar o projeto;
2. Dar permissão de execução para o **init.letsencrypt.sh: chmod +x init.letsencrypt.sh**
3. O script vem configurado para rodar em modo de stage para não ultrapassar o limite de geração do LetsEncrypt. Nesse caso os subdomínios atrás do proxy reverso não ficam com SSL válido. Para alterar isso, edite o init.letsencrypt.sh e mude de 1 para 0 a seguinte linha:
    - De staging=1 para staging=0
4. Rodar o o script: **./init.letsencrypt.sh**

O script já roda o **docker-compose up**. Como o serviço **nginx** no **docker-compose.yml** foi configurado com o **depends_on**, todo o ambiente já é configurado.

O **depends_on** é necessário pois o Nginx está fazendo proxy para os serviços /jenkins e sem ele o Nginx não sobe.

4.1.3 Modelo de publicação do ambiente

Características da máquina virtual **VPS24695**

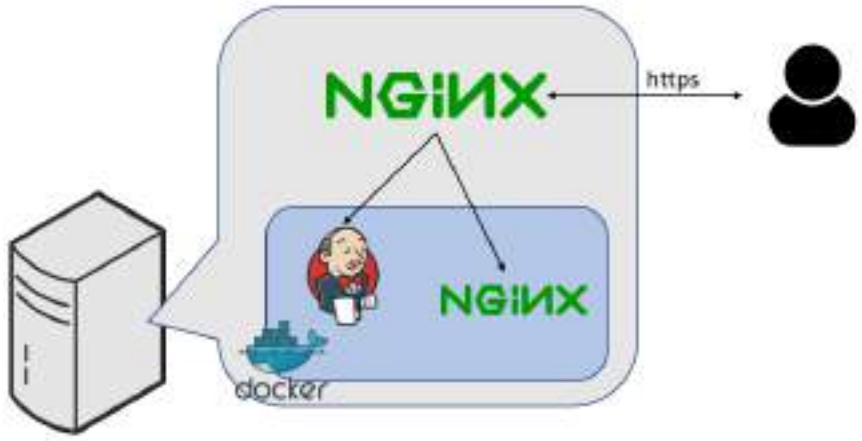
HOSTNAME: @gitlab.fabricadesoftwareifmt.com.br

OS: Ubuntu 18.04 bionic

Kernel: x86_64 Linux 4.15.0-39-generic

[1] https://gitlab.fabricadesoftwareifmt.com.br/leonardobraun/fabrica-software-docker
[2] https://certbot.eff.org/
[3] https://letsencrypt.org/pt-br/
[4] http://fabricadesoftwareifmt.com.br/

Figura 33 – Modelo de publicação do ambiente



Packages: 606

CPU: Intel Xeon CPU E5-2630 v4 @ 2x 2.2 GHz

RAM: 3196MiB / 3913MiB

A máquina virtual está publicada com o endereço de IP 191.252.205.228 e com o domínio gitlab.fabricadesoftwareifmt.com.br

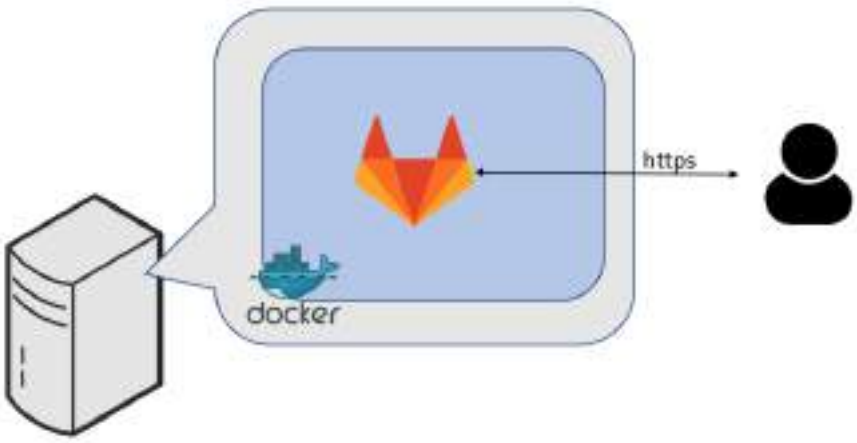
- Acesso SSH liberado na porta 22

4.1.4 Ambiente de contêineres

O projeto está versionado no Gitllab. Acesse o endereço[5] para clonar a infraestrutura.

Modelo de publicação do ambiente é ilustrado na figura 34.

Figura 34 – Ambiente de contêineres



[5] https://gitlab.fabricadesoftwareifmt.com.br/leonardobraun/gitlab-docker

4.2 Instalação do Ambiente

4.2.1 Gitlab

Inicialmente é necessário analisar o hardware necessário para o ambiente que está sendo configurado. Utilize o link[6] para customizar o hardware conforme necessidade:

No ambiente do IFMT utilizamos e recomendamos as seguintes especificações:

- 4 vCPUS
- 8 GB de memória RAM
- 120 GB de armazenamento SSD

Portas utilizadas no ambiente:

- 80 → HTTP Gitlab e Rancher
- 443 → HTTP Gitlab e Rancher
- 6443 → API Kubernetes
- 5000 → Container Registry Gitlab

Pode ser necessário a liberação das portas do firewall da rede e também no firewall das VMS. Utilize o exemplo abaixo para liberar as portas no Firewalld:

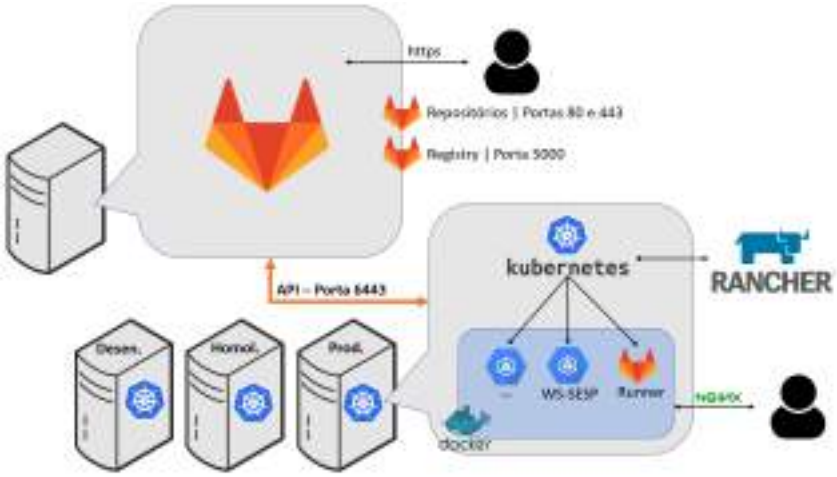
sudo firewall-cmd –zone=public –add-service=5005 –permanent

Verifique também no IPTABLES se as portas estão liberadas:

iptables -L

A figura 35 apresenta a arquitetura da infraestrutura do ambiente:

Figura 35 – Arquitetura da infraestrutura do ambiente



[6] https://docs.gitlab.com/ee/install/requirements.html

4.2.1.1 Instalação Omnibus

No ambiente configurado na SESP-MT foi utilizado o método de instalação denominado Omnibus. Nesse método a instalação é realizada diretamente no sistema operacional de uma máquina virtual ou servidor físico.

Para maiores informações acesse https://about.gitlab.com/install/

É possível utilizar outros métodos de instalação, consulte o site do Gitlab para verificar.

4.2.2 Configurações da Pipeline

As definições, conceitos e o processo de build utilizado no projeto estão descritas no capítulo Processo de Build.

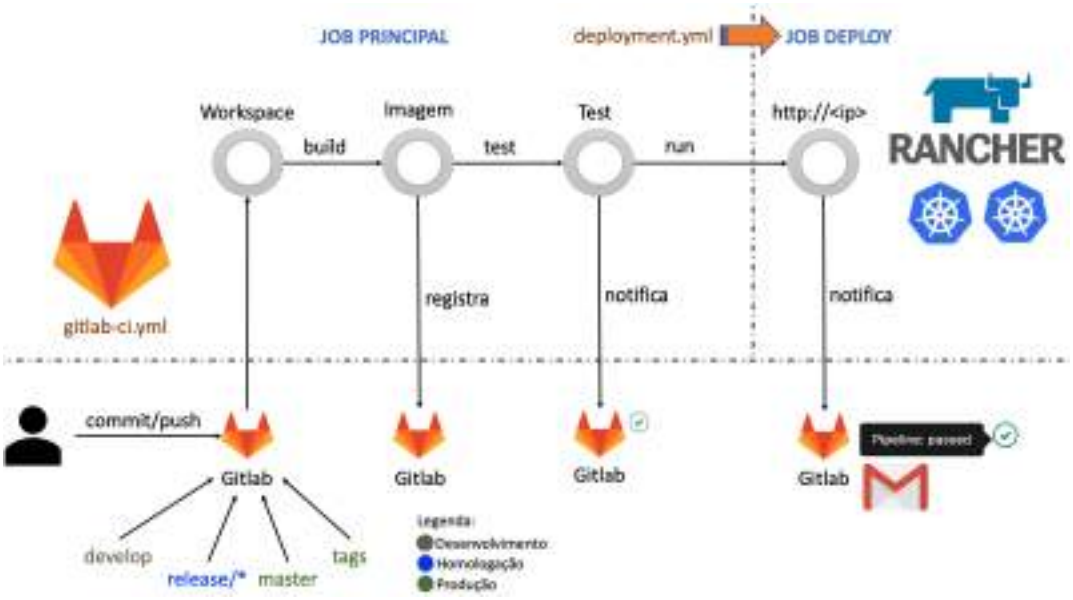
As configurações necessárias para implementar a Pipeline no Giltab estão descritas no capítulo Configurações Gitlab CI.

A documentação da arquitetura do Container Registry do Gitlab está disponível no link https://docs.gitlab.com/c

O processo de gestão da implantação e as configurações necessárias para integração entre Gitlab e Kubernetes estão descritas no caítulo Mapeamento e Modelagem do Processo de Gestão da Implantação

A figura 36 apresenta a versão da PIPELINE proposta pela Fabrica de Software do IFMT. Nessa PIPELINE formam substituídos o Jenkins para Integração Contínua e o Nexus para Container e Package Registry, ambos substituídos pelo Gitlab.

Figura 36 – Pipeline Proposta para o Ambiente SESP

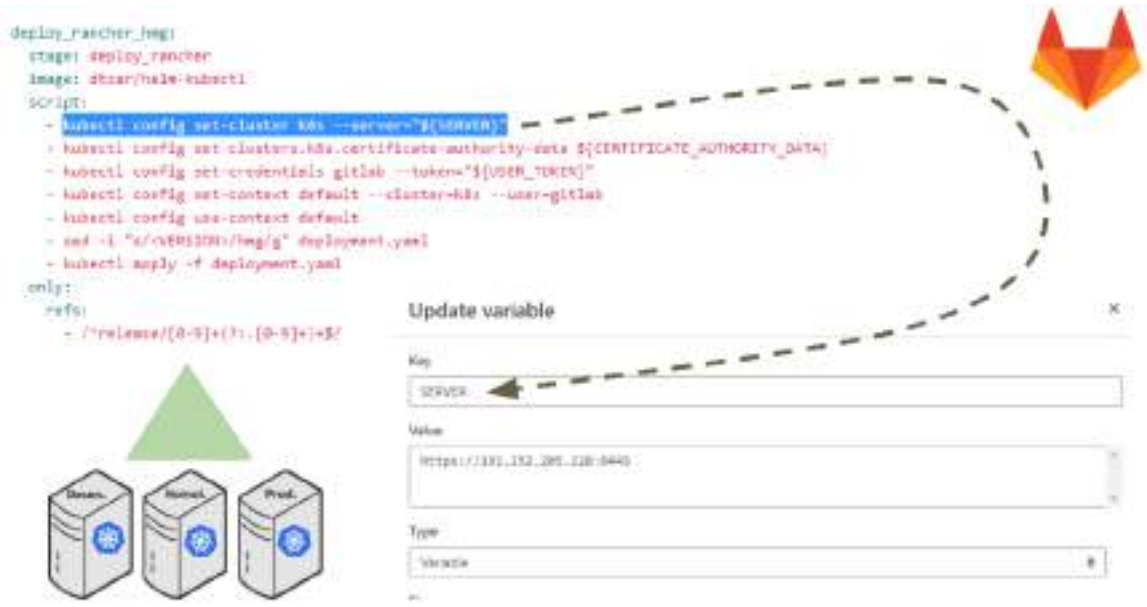


4.2.3 Configurando ambientes de trabalho

No ambiente que foi configurado na SESP-MT utilizamos apenas um cluster para fazer os testes, no exemplo abaixo, ilustrado na figura 37, foi configurado para o stage de deploy em homologação. Para isso, criou-se uma variável no Gitlab denominada “SERVER“ com o IP do cluster Kubernetes.

Ocorre que em cada STAGE definido no **gitlab-ci.yml** é informado por meio dessa variável em qual cluster o container será publicado:

Figura 37 – Configurando ambientes de trabalho



Para diferentes ambientes, tais como PRODUÇÃO, HOMOLOGAÇÃO E DESENVOLVIMENTO, será necessário seguir os passos:

1. Criar variáveis no Gitlab, exemplo (SERVER_PROD, SERVER_HMG, SERVER_DEV)
2. Atribuir o IP do cluster Kubernetes para a variável do respectivo ambiente
3. Configurar os STAGEs de cada ambiente com sua respectiva variável

Exemplos de configuração do Gitlab CI e do Deployment são apresentados a seguir:

```
image: docker:stable



services:
docker:dind



variables:
    MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/
        repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.
        Slf4jMavenTransferListener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -
        Djava.awt.headless=true"
    MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=
        true -DdeployAtEnd=true"
    CONTAINER_IMAGE: fabricadesoftwareifmt.com.br/registry/$CI_PROJECT_PATH

stages:
build
deploy
build_docker
```

```
19
20
21 build_maven:
22
23   stage: build
24   image: maven:3.6.3-openjdk-11-slim
25   cache:
26     paths:
27 .m2/repository
28   script:
29 'mvn $MAVEN_CLI_OPTS clean package -s $mavenSettings'
30   artifacts:
31       expire_in: 10 day
32       paths:
33 target/*.jar
34   only:
35     refs:
36 develop
37 /^release/[0-9]+(?:.[0-9]+)+$/
38 tags
39
40
41
42 deploy_maven:
43   stage: deploy
44   image: maven:3.6.3-openjdk-11-slim
45   cache:
46     paths:
47 target/
48   script:
49 'mvn $MAVEN_CLI_OPTS deploy -s $mavenSettings'
50   only:
51     refs:
52 master
53
54
55
56 ##DESENVOLVIMENTO
57 build_desenv:
58   stage: build_docker
59   image: docker:latest
60   cache:
61     paths:
62 target/
63
64 before_script:
65 docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
66   script:
67 docker pull $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME || true
68 docker build --cache-from $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME --tag $CONTAINER_IMAGE:
     $CI_COMMIT_REF_NAME .
69 docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME
70   only:
71     refs:
72 develop
73
74
75
76 ##HOMOLOCAGAO
77 build_homologa:
78   stage: build_docker
```

```
  image: docker:latest
  cache:
    paths:
target/
  before_script:

docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
  script:
docker pull $CONTAINER_IMAGE:hmg || true
docker build --cache-from $CONTAINER_IMAGE:hmg --tag $CONTAINER_IMAGE:$
    CI_COMMIT_REF_SLUG --tag $CONTAINER_IMAGE:hmg .
docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_SLUG
docker push $CONTAINER_IMAGE:hmg
  only:
    refs:
/^release/[0-9]+(?:.[0-9]+)+$/

##PRODUCAO
build_prod:
  stage: build_docker
  image: docker:latest
  before_script:

docker login -u $REGISTRY_ADMIN -p $REGISTRY_PASSWORD $REGISTRY_URL
  script:
docker pull $CONTAINER_IMAGE:latest || true
docker build --cache-from $CONTAINER_IMAGE:latest --tag $CONTAINER_IMAGE:$
    CI_COMMIT_REF_SLUG --tag $CONTAINER_IMAGE:latest .
docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_SLUG
docker push $CONTAINER_IMAGE:latest

  only:
    refs:
tags
```

Código 4.1 – Arquivo gitlab-ci.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: registro-geral
  labels:
    app: registro-geral
spec:
  replicas: 3
  selector:
    matchLabels:
      app: registro-geral
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 33%
  template:
    metadata:
      labels:
        app: registro-geral
    spec:
      containers:
        - name: registro-geral-container
          image: fabricadesoftwareifmt.com.br/registry/sesp-mt/registro-geral:<VERSION
              >
          ports:
            - containerPort: 8080
```

Código 4.2 – Arquivo deployment.yam

Trabalhando com certificados auto assinados

Para casos onde o Gitlab e o Rancher não estão expostos na internet, é possível trabalhar com certificados auto assinados. Embora seja altamente recomendável proteger seu registro usando um certificado TLS emitido por uma CA conhecida, você pode escolher usar certificados auto-assinados ou usar seu registro em uma conexão HTTP não criptografada. Qualquer uma dessas opções envolve trocas de segurança e etapas de configuração adicionais.

Os links abaixo descrevem as configurações necessárias:

- https://docs.docker.com/registry/insecure/
- https://kubernetes.io/pt/docs/concepts/cluster-administration/certificates/

Depois de efetuada as configurações restart o serviço do Docker é tudo deverá funcionar.

CAPÍTULO 5

Gestão de Versões de Banco de Dados

Este capítulo apresenta o conceito de Migrations e descreve o processo de gerenciamento e versionamento de scripts de banco de dados para projetos de software e sua implementação na pipeline de integração contínua proposta para a Secretaria de Segurança Publica do Estado de Mato Grosso.

5.1 Introdução

Tradicionalmente, as equipes de desenvolvimento de software utilizam ferramentas para gerenciar e versionar os software que estão desenvolvendo, especialmente em ambientes com diversos desenvolvedores e que necessitam integrar códigos com frequência. Mas isso acaba não sendo uma realidade para o gerenciamento dos scripts de banco de dados.

Como exemplo, vejamos a figura 38 que apresenta um ambiente de desenvolvimento onde os desenvolvedores estão trabalhando em um software chamado “Shiny Soft“. Esse ambiente está organizado em uma pipeline de integração contínua. Junto a esse software, podemos perceber em cada figura que o banco de dados “Shiny DB“ também evolui constantemente, a cada versão é possível que ocorra alguma alteração.

Toda versão nova desse software ocasiona em novas linhas de código e alterações nas que existem. Nesses momentos que as ferramentas de gerenciamento e versionamento de código mostram-se importantíssimas. Elas revelam o que foi alterado a partir de um ponto anterior, ou seja, quais mudanças ocorreram entre as versões anteriores e a que estou comparando. Na figura 39 é possível verificar um recorte do histórico de commits realizados na versão “v0.5.0” do microsserviço Gerenciamento Ocorrência. Essa é uma versão de produção.

Figura 38 – Ambiente de desenvolvimento

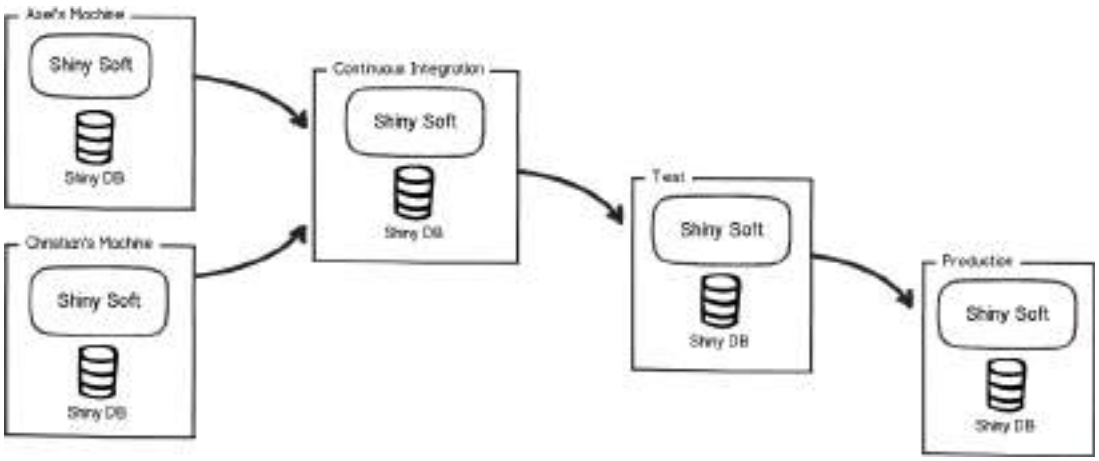
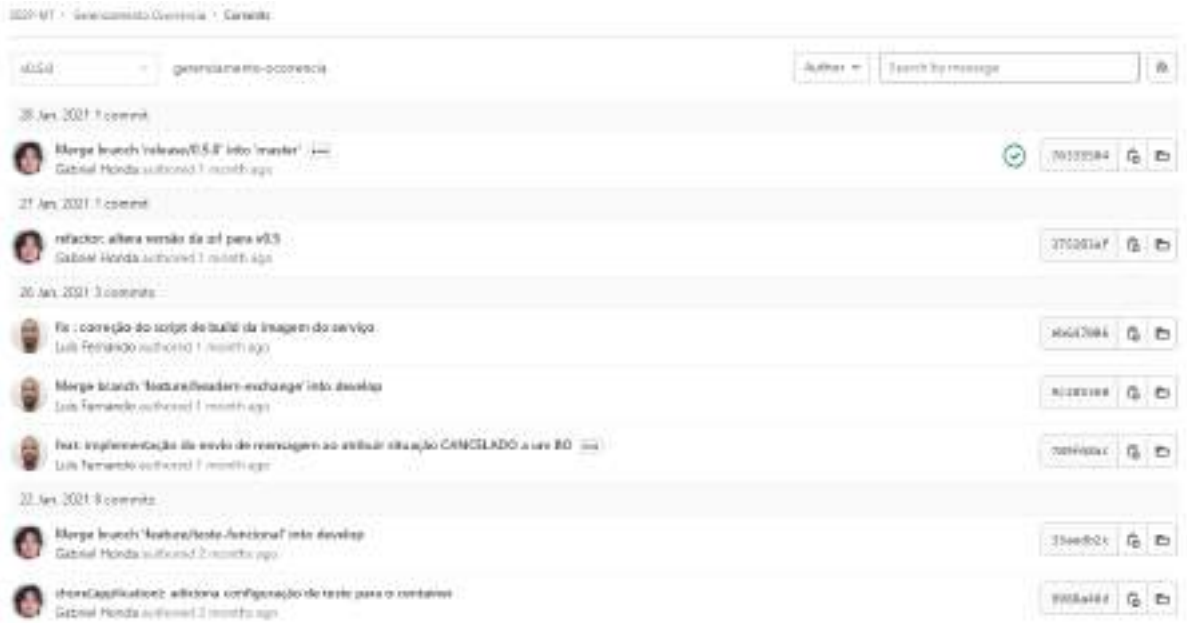


Figura 39 – Histórico de commits realizados na versão “v0.5.0”



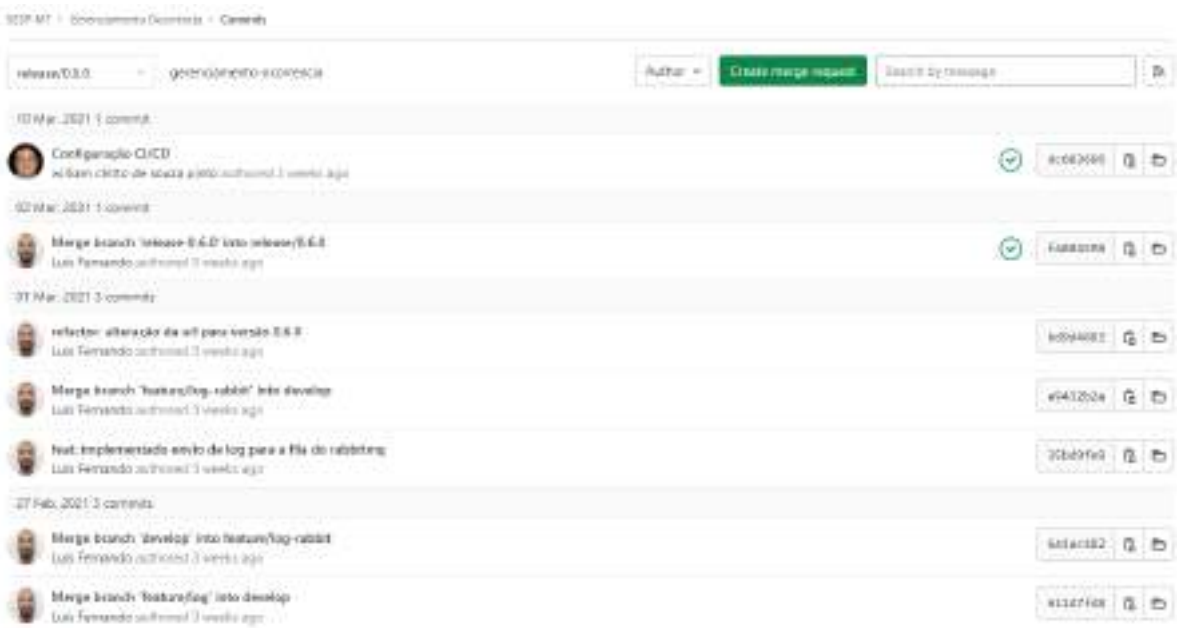
É natural que novas implementações gerem funcionalidades, correções e melhorias de código e consequentemente, novas versões do software. A figura 40 apresenta uma nova versão candidata desse software. Novamente é possível navegar pela versão e verificar todo o histórico de commits com suas respectivas alterações no código.

Todas essas informações da ferramenta de gerenciamento de código fonte possibilitam saber o que mudou no código do software, **mas e no banco de dados, o que mudou**?

Frequentemente, os desenvolvedores se deparam com as seguintes questões:

- Qual o estado do banco?
- Os SQL já foi aplicado ou ainda não?
- Os scripts foram aplicados em todos os ambientes (Dev, Homologa e Produção)?
- Como configurar uma nova instância do banco?

Figura 40 – Versão candidata do software

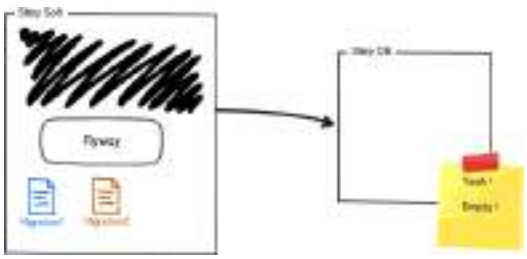


5.2 Migrations

Migrations é um modelo de gerenciamento de scripts de banco de dados que permitem a criação e manipulação de bancos de dados, fornecendo uma série de recursos que possibilitam o gerenciamento e o versionamento dos códigos SQL possibilitando assim a manutenção de um histórico de alterações que ocorreram na base de dados conforme elas ocorreram ao longo do tempo. A partir desse históricos de alterações é possível reverter qualquer alteração realizada no banco de dados, otimizando assim o gerenciamento das alterações realizadas e o controle das versões do banco, que passam a estar atreladas às versões do software.

Cada migration é um arquivo que contém declarações que irão manipular o banco de dados. A figura 41 apresenta esse modelo.

Figura 41 – Migrations

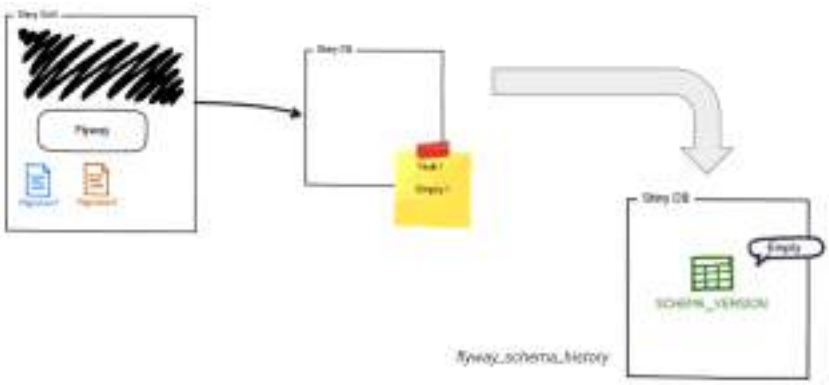


Portanto, as migrations são responsáveis pelo gerenciamento do banco de dados a partir de arquivos que são criados e versionados junto às versões do software e possibilitam:

- Recriar o banco de dados do zero
- Deixar sempre claro o estado do banco de dados
- Migrar de forma determinística da versão atual do software para uma mais recente

Após a criação do banco, uma tabela é criada para gerenciar a execução das migrations no tempo, conforme demonstrado na figura 42:

Figura 42 – Representação das atualizações do banco de dados



Cada migrations que é criada representa um nova versão do banco de dados, ou seja, uma alteração registrada no histórico da evolução do banco de dados. No exemplo da figura 42 é possível perceber que o banco está vazio no primeiro quadro e, logo após, na migration de "Versão 1" três tabelas foram criadas. Uma segunda migration foi aplicada (Versão 2) que cria mais 2 tabelas no banco de dados, totalizando 6 tabelas, se considerarmos a tabela de versionamento.

Se olharmos a tabela 3 de versionamento do banco de dados, ela terá as seguintes informações:

Tabela 3 – Registros do histórico das migrations

flyway_schema_history

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1996767037	axel	2016-02-04 22:23:00.0	546	true
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-06 09:18:00.0	127	true

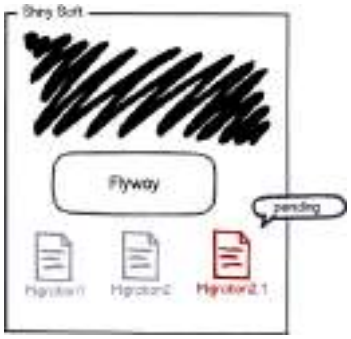
Cada linha refere-se a uma migration que foi executada e as colunas possuem as seguintes informações:

- **installed_rank**: ordem da execução
- **version**: versão da migration
- **description**: descrição da migration
- **type**: tipo do código declarado
- **script**: nome do arquivo
- **checksum**: verificado criado para checar se houve alteração no migration após sua execução
- **installed_by**: usuário que executou a migration
- **installed_on**: data de execução da migration

- **execution_time**: tempo de execução da migration
- **success**: informação referente a confirmação de execução da migration

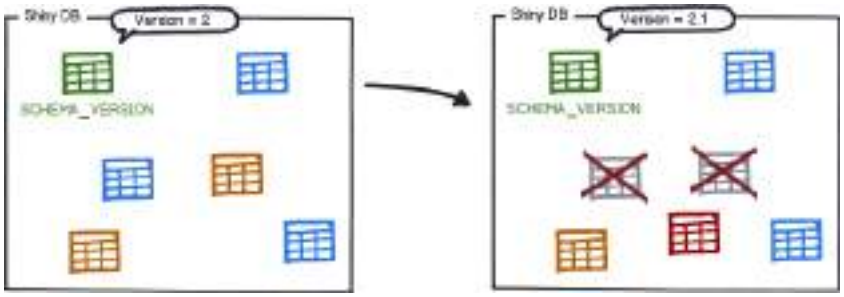
Toda vez que uma nova migration é criada, essa fica como pendente até que seja executada durante a fase de inicio do software. No exemplo da figura 43 um arquivo (migration) foi criado com a versão 2.1:

Figura 43 – Migration pendente



Essa migration refatora o banco de dados e consolida duas tabelas em 1, conforme ilustrado na figura .

Figura 44 – Refatoração de tabelas



Uma nova linha foi acrescentada na tabela 4 de versionamento das migrations no banco de dados:

Tabela 4 – Registros do histórico das migrations

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1998767037	axel	2016-02-04 22:25:00.0	546	true
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-06 09:18:00.0	127	true
3	2.1	Refactoring	JDBC	V2_1__Refactoring		axel	2016-02-10 17:45:05.4	251	true

44

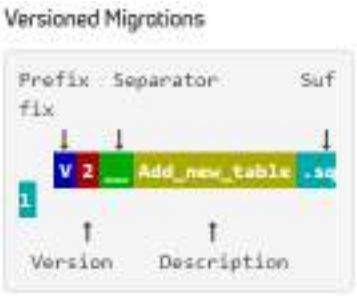
IMPORTANTE!!! Toda versão (arquivo/migration) é válida se está conforme a notação adotada. Na maioria dos casos basta utilizar a incrementação de um inteiro.

A figura 45 apresenta a conversão comumente utilizada para o versionsamento das migrations.

É possível estabelecer um padrão próprio de nomes para os arquivos. Acesse os link abaixo para mais informações:

https://flywaydb.org/documentation/concepts/migrations#versioned-migrations

Figura 45 – Convenção de nome para migrations



5.3 Ferramenta Flyway

O Flyway é a principal ferramenta de Migrations utilizada para ambientes de desenvolvimento que utilizam Java. Ela possui as seguintes características:

- Ferramenta open-source
- As migrations podem ser escritas em SQL ou Java
- É possível utilizar a ferramenta em linha de comando ou Maven plugin
- Suporta as principais bases de dados (Oracle, PostgreSQL e MariaDB)
- Utiliza basicamente 7 comandos (Migrate, Clean, Info, Validate, Undo, Baseline e Repair)

Mais informações em https://flywaydb.org/documentation/

Entre diversas funcionalidades, as principais são:

- Sincronizar o banco de dados com a versão da aplicação
- Saber quais scripts SQL foram executados ou não
- Automatizar a execução dos scripts
- Criar um banco de dados do zero

5.4 Estratégia de versionamento

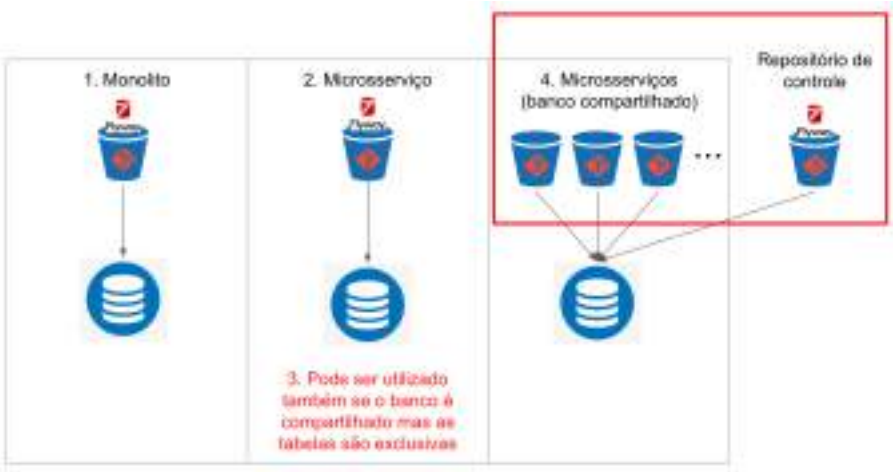
Durante a análise dos modelos de versionamento utilizados, foram identificados 4 modelos usuais das migrations, conforme apresentado na figura 46:

5.4.1 Projeto de Monolito

Nesse modelo as migrations são gerenciadas e versionadas junto ao software. Cada versão do software possui também em seu projeto o arquivos das migrations que representam as alterações realizadas no banco para aquelas versão.

Comumente utiliza-se o Flyway como plugin do projeto para execução em fase de inicio da aplicação.

Figura 46 – Modelos de versionamento



5.4.2 Projeto de Microsserviço

Nesse modelo cada microsserviço possui seu respectivo banco de dados e, assim como em um projeto monolito, as migrations são gerenciadas e versionadas junto ao software.

Cada versão do software possui também em seu projeto o arquivos das migrations que representam as alterações realizadas no banco para aquelas versão.

Comumente utiliza-se o Flyway como plugin do projeto para execução em fase de inicio da aplicação.

5.4.3 Projeto de Microsserviço com compartilhamento de banco de dados com tabelas exclusivas

Nesse modelo os microsserviços compartilham um banco de dados mas cada microsserviço possui sua respectiva tabela, não interferindo ou alterando tabelas de outros microsserviços.

As migrations são gerenciadas e versionadas junto ao software. Cada versão do software possui também em seu projeto o arquivos das migrations que representam as alterações realizadas no banco para aquelas versão.

Comumente utiliza-se o Flyway como plugin do projeto para execução em fase de inicio da aplicação.

IMPORTANTE!!! Esse modelo foi adotado no Projeto de Microsserviços do IFMT e implementado na pipeline no Projeto de DevOps.

5.4.4 Projeto de Microsserviço com compartilhamento de banco de dados e tabelas não exclusivas

Nesse modelo os microsserviços compartilham um banco de dados e também tabelas e não existe restrições de uso ou alteração de tabelas entre os microsserviços.

As migrations são gerenciadas e versionadas em um repositório exclusivo para o versionamento do banco de dados. Esse repositório fará a gestão das alterações de todos os microsserviços que compartilham o banco.

A cada versão do software é necessário atualizar também o projeto específico do banco de dados e, no caso de uma pipeline de integração e entrega contínua, é necessário rodas tanto a pipeline do software quanto a do banco de dados.

IMPORTANTE!!! Nesse modelo os microsserviços não possuem o Flyway ativo.

5.5 Padrão de Commits e gerenciamento do código

Para facilitar a gestão das alterações, sugere-se a utilização do prefixo “db:“ em todos commits referentes a criação de migrations, como no exemplo abaixo:

“db: Script de criação da tabela cargo”

Esse modelo facilita a rastreabilidade das alterações referentes ao banco de dados em uma determinada versão. Sugere-se também que os commits referentes a alterações no banco de dados sejam exclusivas para criação das migrations.

Assim como nas alterações e implementações de código, os desenvolvedores são responsáveis por resolver problemas que novas migrations possam ocasionar quando realizarem merge entre branches.

5.6 PIPELINE

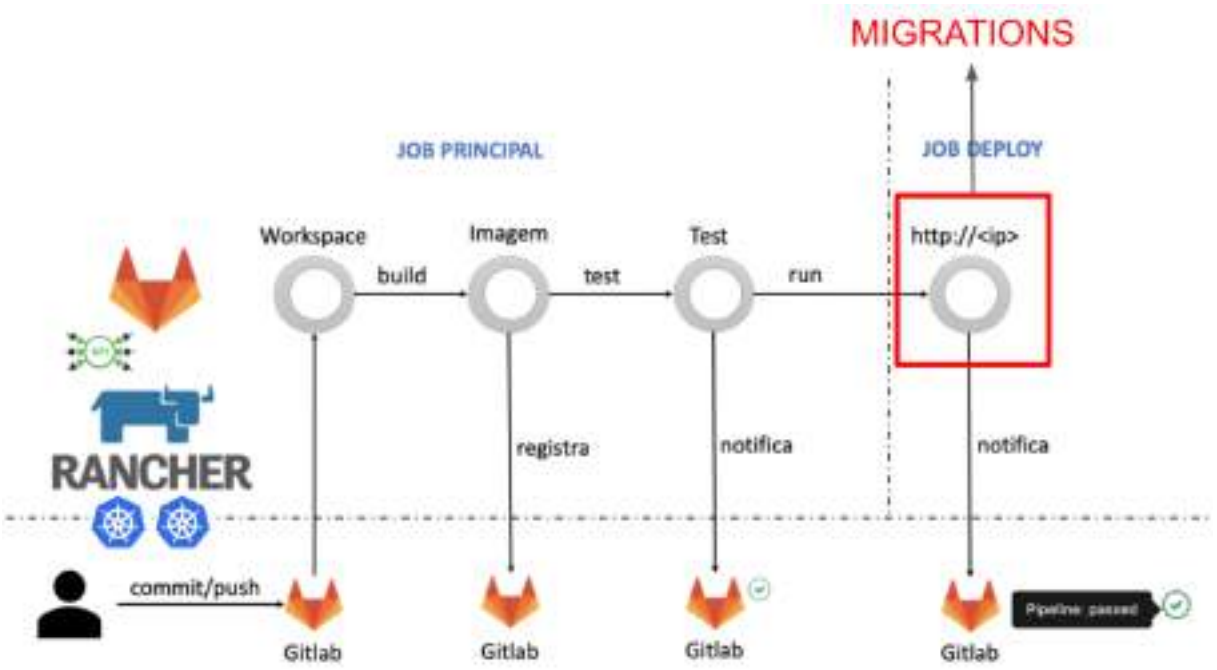
Na PIPELINE de integração e entrega contínua, as migrations são executadas em fase de deploy (figura 47). Isso ocorre porque é necessário realizar todas as etapas anteriores e disponibilizar a imagem do sistema pronta para ser iniciada. O banco de dados só será atualizado se essa imagem for construída e testada com sucesso, além de instalada em um servidor Kubernetes. Após o inicio do container no Kubernetes é que o sistema será iniciado e a aplicação das migrations pendentes será efetivada.

Na pipeline esse processo ocorre de forma tardia, ou seja, ocorre na execução do stage de deploy no Kubernetes e, em caso de sucesso na execução, a pipeline é concluída com sucesso. Como a execução do FlyWay é durante o inicio do sistema, por isso tardia (após a finalização da pipeline do Gitlab), em caso de erros na execução das Migrations o serviço irá quebrar e isso acarretará em divergência de estados dos serviços (A pipeline do Gitlab passou com sucesso mas o container no Kubernetes está inoperante).

Para resolver esse problema foi implementado um nova stage na pipeline do Gitlab denominado “Rollback“. Durante o stage de Deploy é checado se a nova versão foi iniciada com sucesso, e isso envolve o sucesso na execução do Flyway. Portanto, temos as seguintes fases:

1. Execução do deployment.yaml (pipeline Gitlab)
2. Criação do POD no Kubernetes
3. Iniciação do sistema
4. Execução das migrations

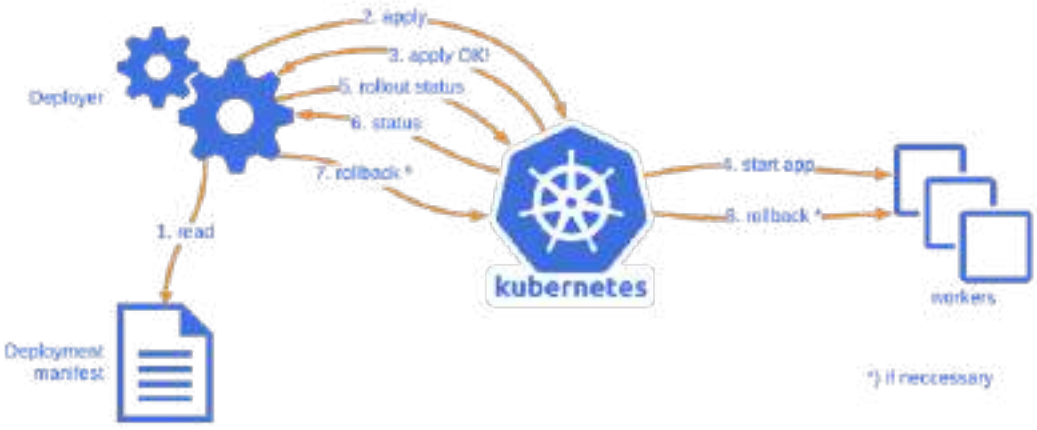
Figura 47 – Execução das migrations na pipeline



5. Checagem de status do POD
6. Rollback (se necessário)

A figura 48 apresenta mais detalhes do processo de Rollback:

Figura 48 – Processo de Rollback



Os seguintes trechos de código foram inseridos no arquivo **gitlab-ci.yaml**:

- No stage deploy_rancher - script: para definição do tempo de espera para checagem de status do POD e verficação do status.

```
1  - kubectl patch deployment.v1.apps/gerenciamento-patrulha -p '{"spec":{"
       progressDeadlineSeconds":10}}'
2  - kubectl rollout status deployment gerenciamento-patrulha
```

Código 5.1 – Configuração do Processo de Rollback

IMPORTANTE!!! O nome que está sendo utilizado, no exemplo gerenciamento-patrulha, deve ser o mesmo definido na tag "app" do deployment.yaml do projeto.

- Novo stage criado para realização do rollback em caso de erro na execução do FlyWay

```
stages:
    - build
    - deploy
    - build_desenv
    - build_homologa
    - build_prod
    - deploy_rancher
    - rollback

##ROLLABCK
rollback:
  stage: rollback
  image: dtzar/helm-kubectl
  when: on_failure
  script:
    - kubectl config set-cluster k8s --server="${SERVER}"
    - kubectl config set clusters.k8s.certificate-authority-data ${
        CERTIFICATE_AUTHORITY_DATA}
    - kubectl config set-credentials gitlab --token="${USER_TOKEN}"
    - kubectl config set-context default --cluster=k8s --user=gitlab
    - kubectl config use-context default
    - kubectl rollout undo deployment gerenciamento-patrulha
```

Código 5.2 – Novo Stage para o Processo de Rollback

5.7 Referências

- https://flywaydb.org/
- https://medium.com/polarsquad/check-your-kubernetes-deployments-46dbfbc47a7c
- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds

CAPÍTULO 6

Processo de Testes de Software Proposto

Este capítulo apresenta os conceitos e técnicas de um processo de testes de software proposto para o cenário de desenvolvimento de software da Secretaria de Segurança Pública do Estado de Mato Grosso.

6.1 Fundamentação de Testes

Nesta seção são apresentados conceitos e técnicas defendidas pelos principais autores da área de desemvolvimento de testes, que deram sustentação à proposta apresentada como processo de testes.

6.1.1 Pirâmide de Testes

A pirâmide de testes é uma maneira de pensar sobre como diferentes tipos de testes automatizados devem ser usados para criar um portfólio equilibrado de testes. Seu ponto essencial é que você deve ter muito mais Testes de Unidade de baixo nível do que testes de grandes partes do sistema (BroadStackTests) de alto nível em execução por meio de uma Interface Gráfica do Usuário (IGU), conforme ilustrado na figura 49.

Martin Fowler relata que em grande parte de sua carreira a automação de testes significava testes que direcionavam uma aplicação por meio de sua interface de usuário (IU). Essas ferramentas geralmente fornecem a facilidade de gravar uma interação com a aplicação e, em seguida, permitir a reprodução dessa interação, verificando se a aplicação retornou os mesmos resultados. Essa abordagem funciona bem inicialmente. É fácil gravar testes, e os testes podem ser gravados por pessoas sem conhecimento de programação.

Figura 49 – Pirâmide de Testes

Mas esse tipo de abordagem rapidamente se torna um problema, tornando-se uma casquinha de sorvete[1]. Testar por meio da IU é lento, aumentando o tempo de construção do software. Frequentemente são requeridas licenças para o software de automação de teste, o que significa que só pode ser feito em máquinas específicas. Normalmente eles não podem ser executados facilmente em um modo sem interface gráfica, rodando em "background" (modo "headless"), monitorado por scripts para ser implementado em um pipeline de implantação adequado.

Mais importante ainda, esses testes são muito frágeis. Uma melhoria no sistema pode facilmente acabar interrompendo muitos desses testes, que então precisam ser regravados. Você pode reduzir esse problema abandonando as ferramentas de reprodução de gravação, mas isso torna os testes mais difíceis de escrever. [1] Mesmo com boas práticas ao escrevê-los, os testes de ponta a ponta são mais propensos a problemas de não determinismo[2], o que pode minar a confiança neles. Resumindo, os testes executados de ponta a ponta por meio da IU são: frágeis, caros para escrever e demorados para executar. Portanto, a pirâmide argumenta que você deve fazer muito mais testes automatizados por meio de testes de unidade do que por meio de testes tradicionais baseados em IGU. [2]

A pirâmide também defende uma camada intermediária de testes que atuam por meio de uma camada de serviço de uma aplicação, os chamados Testes Subcutâneos[3]. Eles podem fornecer muitas das vantagens dos testes de ponta a ponta, mas evitam muitas das complexidades de lidar com estruturas de IU. Em aplicações da web, isso corresponderia a testes por meio de uma camada de API, enquanto a parte superior da IU da pirâmide corresponderia a testes usando algo como Selenium[4] ou Sahi.

A pirâmide de testes é muito comum no meio dos testes ágeis e, embora sua mensagem central seja sólida, há muito mais a dizer sobre a construção de um portfólio de teste bem equilibrado. Um problema comum é que as equipes combinam os conceitos de testes ponta a ponta, testes de IU e testes voltados para o cliente. Todas essas são características ortogonais. Por exemplo, uma interface de usuário javascript rica deve ter a maior parte de seu comportamento de interface do usuário testado com testes de unidade de javascript usando algo como Jasmine. Um conjunto complexo de regras de negócios pode ter testes capturados em um formulário voltado para o cliente, mas executado apenas no módulo relevante, assim como os testes de unidade.

1 https://watirmelon.blog/testing-pyramids
2 https://martinfowler.com/articles/nonDeterminism.html
3 https://martinfowler.com/bliki/SubcutaneousTest.html
4 http://seleniumhq.org/

Martin Fowler sempre defendeu que os testes de alto nível existem como uma segunda linha de defesa de teste. Se você obtiver uma falha em um teste de alto nível, não apenas terá um bug em seu código funcional, mas também um teste de unidade ausente ou incorreto. Portanto, aconselho que antes de consertar um bug exposto por um teste de alto nível, você deve replicar o bug com um teste de unidade. Em seguida, o teste de unidade garante que o bug permaneça morto.

6.1.1.1 Notas

1. Ferramentas de reprodução de gravação quase sempre são uma má ideia para qualquer tipo de automação, uma vez que resistem à mutabilidade e obstruem abstrações úteis. Elas só valem a pena como ferramenta para gerar fragmentos de scripts que você pode editar como uma linguagem de programação adequada, na forma de Twist ou Emacs.

2. A pirâmide é baseada na suposição de que os testes de pilha ampla são caros, lentos e frágeis em comparação com testes mais focados, como testes de unidade. Embora isso geralmente seja verdade, há exceções. Se meus testes de alto nível são rápidos, confiáveis e baratos para modificar, os testes de nível inferior não são necessários.

Autor: Martin Fowler
Texto original: https://martinfowler.com/bliki/TestPyramid.html
Tradução e adaptações: Leonardo Braun

6.1.2 A pirâmide de testes na prática

A "Pirâmide de Testes"é uma metáfora sobre agrupar os testes de software em baldes de granularidade diferente. Também dá uma ideia de quantos testes devemos ter em cada um desses grupos. Embora o conceito da Pirâmide de Testes já exista há algum tempo, as equipes ainda lutam para colocá-la em prática de maneira adequada. Este artigo revisita o conceito original da Pirâmide de Testes e demonstra como colocá-la em prática. Mostra também quais tipos de testes você deve procurar nos diferentes níveis da pirâmide e exemplos práticos de como eles podem ser implementados.

Figura 50 – Pirâmide de Testes na Prática

O software pronto para utilização precisa ser testado antes de entrar em produção. Conforme a disciplina de desenvolvimento de software amadureceu, as abordagens de testes de software também amadureceram. Ao contrário de uma multidão de testadores manuais de software, as equipes de desenvolvimento passaram a automatizar a maior parte de seus esforços em testar novas versões de um produto. Automatizar os testes possibilita às equipes identificar o que está quebrado no software em questão de segundos e minutos, em vez de dias e semanas.

Um ciclo de feedback reduzido guiado por testes automatizados anda de mãos dadas com as práticas de desenvolvimento ágil, entrega contínua e cultura DevOps. Ter uma abordagem de testes de software eficaz permite que as equipes se movam com rapidez e confiança.

Este artigo explora o que um portfólio de testes completo necessita para ser responsivo, confiável e de fácil manutenção, independentemente de sua finalidade ou se sua arquitetura é de microsserviços, aplicativos móveis ou ecossistemas IoT. Também será detalhado a criação de testes automatizados eficazes e legíveis.

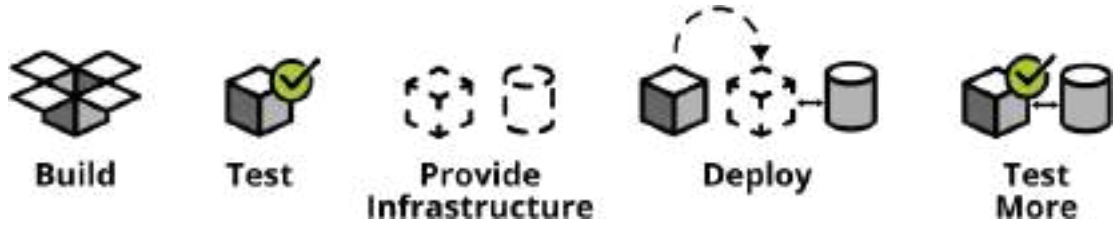
6.1.2.1 A importância dos testes automatizados

O software se tornou uma parte essencial do mundo em que vivemos. Ele superou seu propósito inicial de tornar os negócios mais eficientes. Hoje as empresas tentam encontrar maneiras de se tornarem empresas digitais (first-class digital companies). Como usuários, cada um de nós interage com uma quantidade cada vez maior de softwares todos os dias. As rodas da inovação estão girando rapidamente.

Se você quiser acompanhar o ritmo terá que buscar maneiras de entregar seu software com mais rapidez, sem sacrificar sua qualidade. A **entrega contínua**, prática que garante que seu software seja colocado em produção automaticamente a qualquer momento, pode ajudá-lo com isso. Na entrega contínua você usa uma **pipeline de construção** para testar automaticamente seu software e implantá-lo em seus ambientes de teste e produção.

Construir, testar e implantar manualmente uma quantidade cada vez maior de software se tornou impossível, a menos que você queira gastar todo o seu tempo com trabalho manual e repetitivo, em vez de entregar software funcional. Automatizar tudo, da construção aos testes, implantação e infraestrutura, é o seu único caminho a seguir, ilustrado na figura 51.

Figura 51 – Pipelines de compilação



Os testes de software eram tradicionalmente um trabalho excessivamente manual feito ao implantar sua aplicação em um ambiente de testes e, em seguida, realizar alguns testes do tipo caixa preta, como por exemplo, quando se clica em uma interface de usuário para ver se algo está quebrado. Frequentemente esses testes são especificados por scripts de teste para garantir que os testadores façam verificações consistentes.

É óbvio que testar todas as alterações manualmente é demorado, repetitivo e tedioso. Tarefas repetitivas são chatas e monótonas, o que acaba por levar a erros e ainda faz com que você procure um trabalho diferente no final da semana.

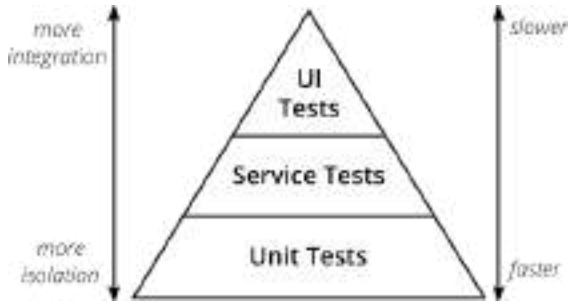
Felizmente existe um remédio para as tarefas repetitivas: **automação**.

Automatizar os testes repetitivos pode ter um impacto significante na sua vida como desenvolvedor de software. Automatize os testes e você não precisará mais seguir os protocolos de clique sem pensar para verificar se o software ainda está funcionando corretamente. Automatize os testes e você poderá alterar seu código fonte sem surpresas. Se você já tentou refatorar seu código em grande escala sem um conjunto de testes adequado, aposto que sabe como isso pode ser uma experiência assustadora. Como saber se algo quebrou acidentalmente ao longo do caminho? Bem, você clica todos os seus casos de teste manuais e é assim. Mas vamos ser honestos, você realmente gosta disso?

6.1.2.2 A Pirâmide de Testes

Se você deseja levar a sério os testes automatizados para o seu software, há um conceito-chave que você deve conhecer: a **pirâmide de testes**. Mike Cohn cunhou esse conceito em seu livro Succeeding with Agile. É uma ótima metáfora visual que nos faz pensar sobre diferentes camadas de teste, conforme ilustrado na figura 52. Também demonstra quanto testar em cada camada.

Figura 52 – Pirâmide de Testes de Mike Cohn



A pirâmide de testes original de Mike Cohn consiste em três camadas que o conjunto de testes deve consistir (de baixo para cima):

1. Testes Unitários
2. Testes de serviço
3. Testes de interface do usuário

Infelizmente o conceito da pirâmide de teste falha um pouco se você olhar mais de perto. Alguns argumentam que a nomenclatura ou alguns aspectos conceituais da pirâmide de Mike Cohn não são ideais, e eu tenho que concordar. De um ponto de vista moderno, a pirâmide parece excessivamente simplista e pode, portanto, ser enganosa.

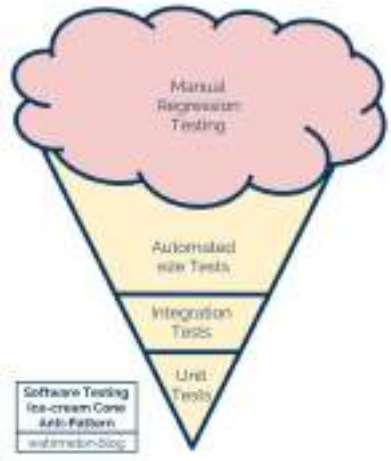
Ainda assim, devido à sua simplicidade, a essência da pirâmide de testes serve como uma boa regra quando se trata de estabelecer seu próprio conjunto de testes. Sua melhor aposta é lembrar duas coisas da pirâmide original de Cohn:

1. Escreva testes com granularidade diferente

2. Quanto mais alto nível você obtém, menos testes você deve ter

A pirâmide é um modelo a ser seguido para chegar a um conjunto de testes saudável, rápido e sustentável: Escreva muitos testes de unidade pequenos e rápidos. Escreva alguns testes mais granulares e pouquíssimos testes de alto nível que testem seu aplicativo de ponta a ponta. Cuidado para não acabar com um conjunto de testes de casquinha de sorvete que será um pesadelo para manter e levará muito tempo para ser executada, ilustrada na figura 53.

Figura 53 – Conjunto de testes de casquinha de sorvete



Fonte: https://alisterbscott.com/kb/testing-pyramids/

Não se apegue muito aos nomes das camadas individuais da pirâmide de testes de Cohn. Na verdade, eles podem ser bastante enganosos: teste de serviço é um termo difícil de entender, o próprio Cohn descreve que muitos desenvolvedores ignoram completamente essa camada. Fica evidente que atualmente os testes de User Interface (UI) não precisam estar no nível mais alto da pirâmide para aplicações de página única (Single page application) em frameworks como React, Angular, Ember.js e outros, é perfeitamente possível testar a unidade de UI em todos esses frameworks.

Dadas as deficiências dos nomes originais, não há problema em sugerir outros nomes para suas camadas de testes, desde que você as mantenha consistentes em suas bases de código e nas discussões com a equipe.

6.1.2.3 Aplicação de exemplo

Vamos utilizar um microsserviço simples, incluindo um conjunto de testes para as diferentes camadas da pirâmide de testes.

A aplicação de exemplo mostra características de um microsserviço. Ele fornece uma interface REST, comunica-se com um banco de dados e busca informações de um serviço REST de terceiros. Ele foi implementado em Spring Boot e deve ser compreensível mesmo se você nunca trabalhou com o Spring Boot antes.

Certifique-se de verificar o código no Github[5]. O arquivo “README” contém as instruções necessárias para executar a aplicação e seus testes automatizados em sua máquina.

6.1.2.4 Funcionalidades

A funcionalidade da aplicação é simples. Ele fornece uma interface REST com três endpoints, conforme listado na tabela 5.

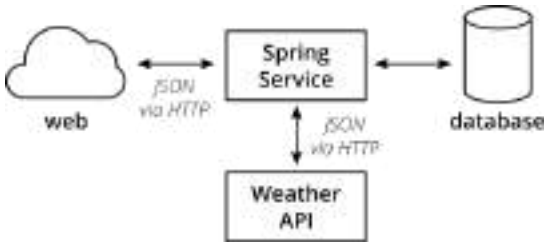
Tabela 5 – Interface REST da Aplicação Exemplo

GET /hello	Retorna "Hello World". Sempre.
GET /hello/{lastname}	Procura a pessoa com o sobrenome fornecido. Se a pessoa for conhecida, retorna "Olá {Firstname} {Lastname}".
GET /weather	Retorna as condições meteorológicas atuais para Hamburgo, Alemanha.

6.1.2.5 Estrutura de alto-nível

O sistema tem a seguinte estrutura, ilustrada na figura 54:

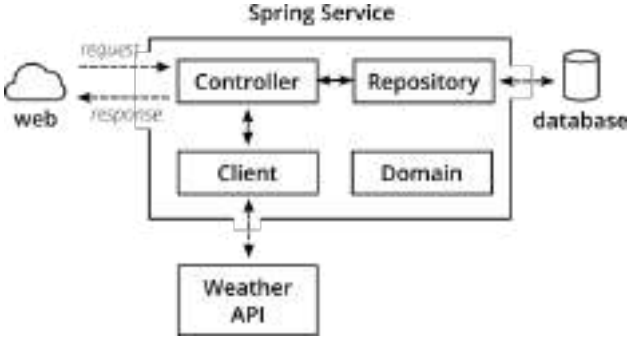
Figura 54 – Estrutura de alto nível do sistema de microsserviço



6.1.2.6 Arquitetura interna

Internamente, o serviço tem uma arquitetura típica do Spring, ilustrada na figura 55:

Figura 55 – Estrutura interna do nosso microsserviço



[5] https://github.com/hamvocke/spring-testing

- As classes **controlller** fornecem endpoints REST e lidam com solicitações e respostas HTTP
- As classes **repository** são as interfaces com o banco de dados e cuidam da gravação e leitura de dados (de/para) o armazenamento persistente
- As classes **client** se comunicam com outras APIs, no nosso caso, busca JSON via HTTPS da API de clima do site darksky.net
- As classes **domain** capturam nosso modelo de domínio, incluindo a lógica do domínio (o que, para ser justo, é muito importante em nosso caso).

Desenvolvedores experientes do Spring podem notar que uma camada frequentemente usada está faltando aqui: Inspirado pelo Domain-Driven Design, muitos desenvolvedores criam uma camada de serviço que consiste em classes de serviço. Decidi não incluir uma camada de serviço nessa aplicação. Um dos motivos é que nossa aplicação é bastante simples, uma camada de serviço seria um nível desnecessário. A outra é que acho que as pessoas exageram nas camadas de serviço.

Nossos repositórios são diretos e fornecem funcionalidade CRUD simples. Para manter o código simples, usei Spring Data. Spring Data nos dá uma implementação de repositório CRUD simples e genérica que podemos usar em vez de fazer o nosso próprio. Ele também se encarrega de criar um banco de dados na memória para nossos testes, em vez de usar um banco de dados PostgreSQL real como faria em produção.

Autor: Ham Vocke
Texto original: https://martinfowler.com/articles/practical-test-pyramid.html
Tradução e adaptações: Leonardo Braun

6.1.3 O estilo GivenWhenThen

Dado-Quando-Então (*GivenWhenThen*) é um estilo de representar testes utilizado para especificar o comportamento de um sistema utilizando a estrutura de Especificação por exemplo (SpecificationByExample[6]). É uma abordagem desenvolvida por Daniel Terhorst-North[7] e Chris Matts como parte do *Behavior-Driven Development* (BDD)[8]. [1] É uma abordagem estruturante utilizada em muitos frameworks de teste como o Cucumber. É possivel considerá-lo ainda como uma reformulação do padrão de testes de quatro fases (*Four-Phase Test pattern*[9]).

A ideia essencial é dividir a escrita de um cenário (ou teste) em três seções:

- A seção **Dado (Given)** descreve o estado do que se pretende testar antes de iniciar o comportamento que está especificando neste cenário. Você pode imaginar isso como um pré-requisito para o teste.
- A seção **Quando (When)** é o comportamento que você está especificando.

[6] https://martinfowler.com/bliki/SpecificationByExample.html
[7] http://dannorth.net/about/
[8] http://dannorth.net/introducing-bdd/
[9] http://xunitpatterns.com/Four%20Phase%20Test.html

- Finalmente, a seção **Então (Then)** descreve as mudanças esperadas decorrentes do comportamento especificado.

Vamos analisar um exemplo utilizando a estrutura de Especificação por exemplo: [2]

```
Recurso: o usuario negocia acoes

  Cenario: O usuario solicita uma venda antes do fechamento das negociacoes
    Dado que tenho 100 acoes da MSFT
    E tenho 150 acoes da APPL
    E o horario e anterior ao fechamento das negociacoes

  Quando peco para vender 20 acoes da MSFT

    Entao devo ter 80 acoes da MSFT
    E devo ter 150 acoes da APPL
    E uma ordem de venda de 20 acoes da MSFT deve ter sido executada
```

Código 6.1 – Exemplo utilizando GivenWhenThen

O exemplo acima utiliza Cucumber [3], que é uma maneira popular de escrever BusinessFacingTests[10], mas é possível utilizar a estrutura do Dado-Quando-Então com qualquer tipo de teste. Alguns desenvolvedores gostam de colocar a estrutura do Dado-Quando-Então como comentários para marcar blocos informais nos testes de unidade [4].

É comum nessa abordagem utilizar o conectivo "**E**"para combinar várias expressões dentro de cada cláusula.

Martin Fowler destaca que prefere caracterizar a seção "Dado"(Given) como uma descrição do estado de pré-requisito. Um Framework de testes, contudo, interpreta os "Dado" (Givens) com um conjunto de comandos necessários para preparar o sistema que está sendo testado para o estado necessário ou correto para executar os comandos "Quando"(When) (É por isso que outras convenções de nomenclatura geralmente chamam isso de "configuração"(setup)). Frameworks de teste fornecem vários métodos de consulta (query) para os comandos "Então" (then), mas esses devem estar livres de efeitos colaterais.

Embora o estilo Dado-Quando-Então seja característico do BDD, a ideia básica é muito comum ao escrever testes ou especificações por exemplo (SpecificationByExample). Meszaros[11] descreve o padrão como Teste de Quatro Fases (Four-Phase Test[12]). Suas quatro fases são Setup (Dado), Exercício (Quando), Verificar (Então) e Desmontar (Limpar) [5]. Bill Wake propôs a formulação Arrange, Act, Assert[13].

6.1.3.1 Notas

1. Em comentários de revisão sobre essa questão, Dan credita a Ivan Moore parte relevante de sua inspiração ao chegar a essa definição.

2. De Pete Hodgson[14]

[10] https://martinfowler.com/bliki/BusinessFacingTest.html
[11] https://martinfowler.com/books/xunit
[12] http://xunitpatterns.com/Four%20Phase%20Test.html
[13] http://xp123.com/articles/3a-arrange-act-assert/
[14] http://blog.thepete.net/

3. Ou para ser rigoroso, ele usa Gherkin, que é o nome do DSL do Cucumber.

4. Os frameworks de teste tendem a seguir o estilo de nomenclatura de xunit[15] ou BDD, o último tende a nomear métodos no estilo Dado-Quando-Então.

5. Não é sempre que Teardown é necessário ao implementar testes (principalmente se você estiver usando Teardown automatizado[16]) e não acrescenta muito ao aspecto de comunicação da especificação por exemplo. Portanto, é aceitável que não esteja contemplado no estilo BDD.

Autor: Martin Fowler
Texto original: https://martinfowler.com/bliki/GivenWhenThen.html
Tradução e adaptações: Leonardo Braun

6.1.4 Manifesto do Teste Ágil, Princípios e Práticas

Em meados de 2013, Sam Laing e Karen Greaves criaram sua versão de um manifesto de testes, como um resumo rápido da mentalidade a ser adotada quando pensamos em testes ágeis e também como uma analogia ao Manifesto Ágil, ilustrado na figura 56.

Figura 56 – Testing Manifesto

A versão traduzida do manifesto em está na figura 57:

Importante destacar também os princípios e práticas apresentados por Elisabeth Hendrickson em 2008 em sua palestra "Agile Testing, Nine Principles and Six Concrete Practices for Testing on Agile Teams 2008":

Nove princípios para testes em equipes ágeis:

- O teste leva o projeto adiante;
- Os testes não podem ser vistos apenas como uma fase do projeto;
- Todos testam, não apenas o especialista em testes;

[15] https://xunit.net/
[16] http://xunitpatterns.com/Automated%20Teardown.html

Figura 57 – Manifesto dos Testes

- Diminua o tempo dos feedbacks;
- Mantenha o código limpo, erros encontrados devem ser resolvidos ASAP;
- Testes descobrem requisitos não explicitados;
- Reduza a documentação;
- "Feito Pronto!", não apenas Feito;
- Adote, sempre que possível, a técnica de Test Driven Development (TDD).

Seis práticas concretas para testes em equipes ágeis:

- Testes Automatizados de Unidade e Integração;
- Desenvolvimento Orientado a Testes (TDD, Testes Unitários, especificamente);
- Testes Automatizados de Regressão no Nível do Sistema;
- Desenvolvimento Orientado a Testes de Aceitação (ATDD, discutir - destilar - desenvolver - demonstrar);
- Testes Exploratórios (baseado em histórias de usuário, se for o caso);
- Testes de versão COM código (repositório de controle de origem inclui código-fonte, testes de unidade e testes de sistema).

6.1.5 Referência

- https://www.growingagile.co.za/2015/04/the-testing-manifesto/
- https://agilizei.com/2017/03/20/manifesto-dos-testes/
- http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf
- https://medium.com/beelabacademy/planejamento-em-testes-ágeis-3055dad670a7

6.2 Processos de Testes SESP/STI

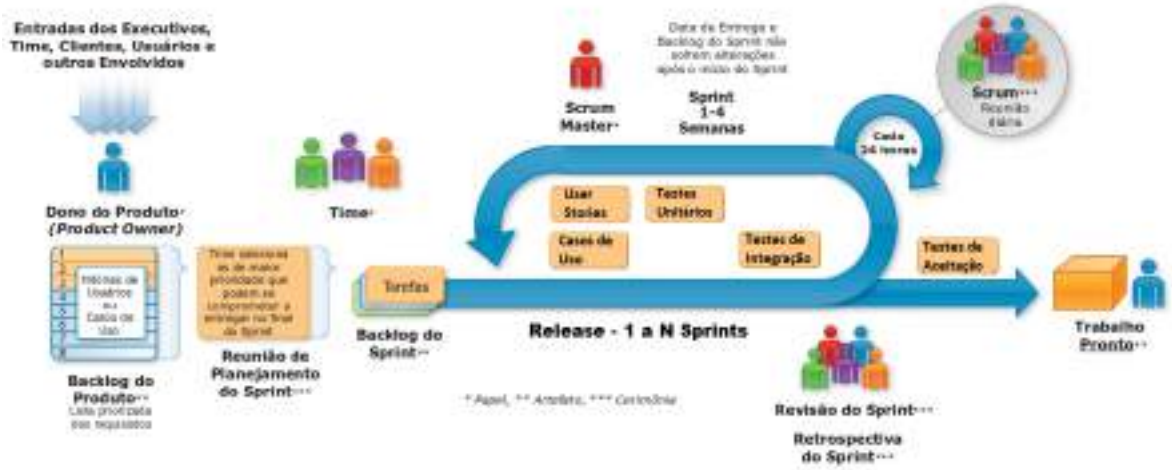
Esta seção tem como objetivo apresentar uma proposta de processo de testes para o setor de desenvolvimento de software da SESP-MT. Nesta proposta são previstos três níveis de testes: testes unitários, testes de integração e testes de aceitação.

Uma característica predominante e que impacta diretamente na forma como o processo foi organizado é a presença de contratação de fábricas de softwares para a prestação de serviços de desenvolvimento, manutenção e sustentação de software, como forma de viabilizar essas atividades frente ao baixo número de profissionais de TI efetivos atualmente na SESP.

6.2.1 Visão Geral do Processo

Todo o processo de testes aqui proposto está baseado na condução das atividades por meio de uma metodologia ágil, ou um modelo de desenvolvimento cuja entrega é realizada em partes fracionadas (releases). A figura 58 a seguir demonstra às atividades relevantes para o processo de testes proposto, na visão da equipe de gestão de projetos de desenvolvimento de software da SESP.

Figura 58 – Destaque das Tarefas de Testes em um ciclo Ágil



Numa visão geral, às atividades de definição e execução dos testes acontecem dentro de um ciclo de desenvolvimento (Sprint/Iteração). Dessa forma, as ações de implementação de testes unitários e testes de integração e suas respectivas execuções ocorrerão repetidas vezes, uma por Sprint.

Na Figura 01, pode-se observar que os testes de aceitação está fora do ciclo (Sprint). Essa representação foi utilizada para demonstrar que os testes de aceitação serão produzidos e executados uma vez para cada entrega para o usuário final (release). Apesar de que as metodologias ágeis têm como princípio a entrega de um artefato de software testável e integrável a cada Sprint, também é possível que sejam executadas N Sprints para que haja uma entrega formal (release) ao usuário final, ou que estamos considerando um processo de software intermediário, com algumas práticas ágeis.

Por outro lado, caso o produto resultante de uma Sprint gere uma entrega formal, os testes de aceitação irão ser produzidos e executados para cada ciclo. Caso seja necessário a execução de N

Sprints para uma entrega formal, os testes de aceitação poderão ser produzidos durante os ciclos e executados apenas uma vez, quando a entrega formal ocorrerá.

6.2.2 Detalhamento do Processo

As contratações de serviços de desenvolvimento, manutenção e sustentação de software realizadas por órgãos do estado de Mato Grosso são geridas por um conjunto de normativas PDS_MT (Processo de Desenvolvimento de Software do Estado de Mato Grosso). Nessas normativas são estabelecidos, dentre outras coisas, quais artefatos de documentação do processo de desenvolvimento de software devem ser atendidos, bem como os modelos desses documentos.

No que se refere a especificação de requisitos, as normativas do PDS_MT estabelecem como base o uso do documento de especificação de casos de uso.

Dessa forma, este processo considera que o levantamento de requisitos e a produção de especificações de casos de uso devem ocorrer paralelamente à produção de User Strories e um Backlog (documentos de especificação de requisitos apoiado pelas metodologias ágeis), onde para cada User Story deve haver uma lista de critérios de aceitação estabelecida.

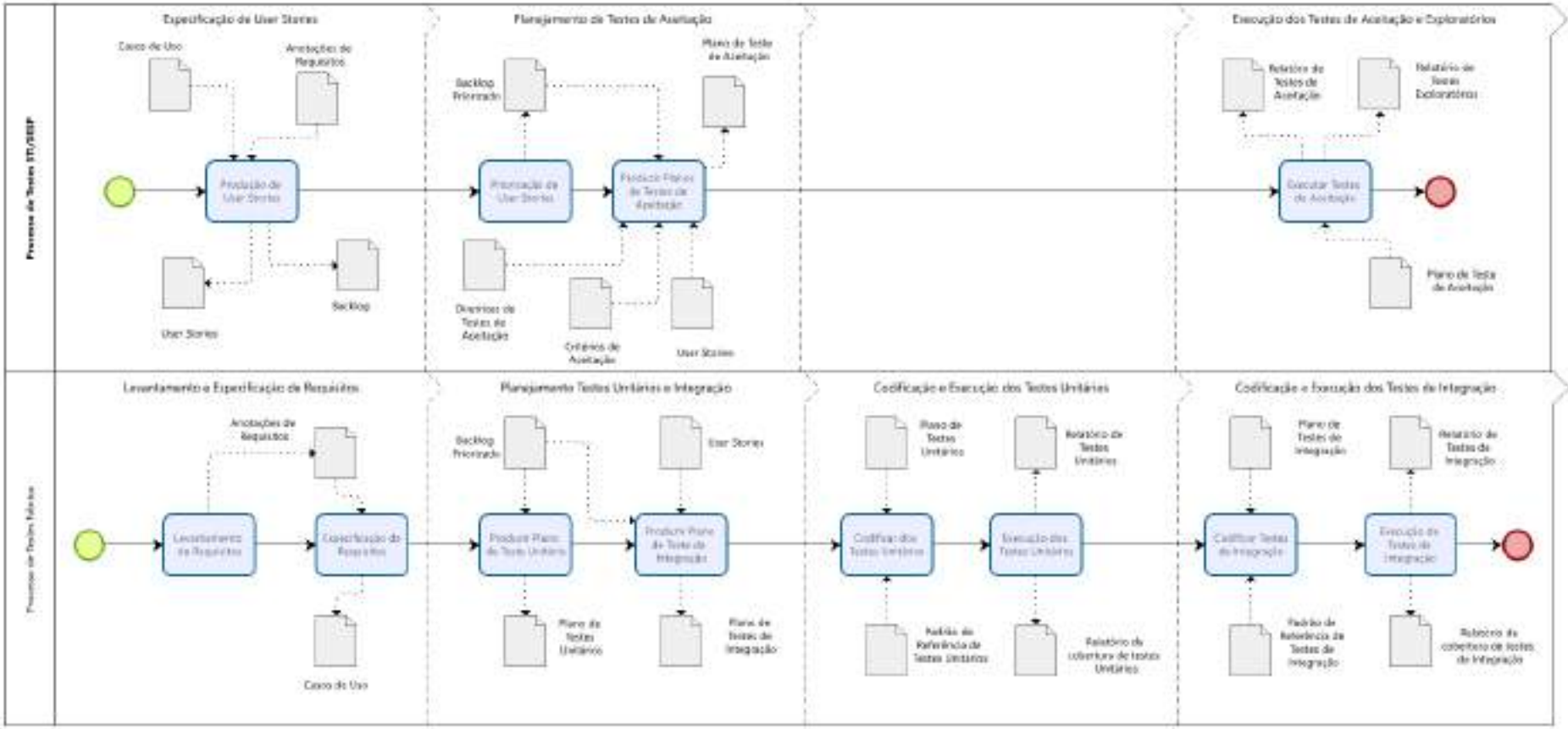
A adoção de User Stories e Backlog, como base da especificação de requisitos e de organização do trabalho, são necessários em virtude do estudo e aplicação de um modelo de desenvolvimento de software em curso na SESP, que tem como objetivo a implantação de práticas DevOps e um Pipeline automatizado para várias fazes da produção de software. Como as práticas DevOps estão calcadas nos princípios e práticas das metodologias ágeis, o uso de User Stories e Backlog é inerente a esse processo proposto.

Outro fator importante do uso de User Stories, associado com uma lista de critérios de aceitação e Backlog se dá pelo fato de que caracterizam como documentos que usam uma linguagem próxima ao usuário final do sistema, com granularidade baixa, tornado a tarefa de produção de cenários de testes, execução e verificação dos testes, muito mais simples, em comparação aos complexos documentos de especificação de casos de uso.

A partir do levantamento de informações a respeito do processo de desenvolvimento de software, e em especial ao processo de testes existente atualmente na SESP, foram elaborados dois fluxos: o primeiro mapeia as ações e produtos relacionados às responsabilidades de testes da equipe efetiva da SESP; o segundo mapeia as ações e produtos relacionados às responsabilidades dos testes das equipes formadas pelas fábricas de software contratadas, ambos ilustrados na figura 59.

A figura 59 ilustra o fluxo do processo de testes para a equipe interna da SESP/STI e o fluxo para o processo de testes que devem ocorrer sob responsabilidade das fábricas de software contratadas. Nestes fluxos a primeiras ações: Levantamento de Requisitos, Especificação de Requisitos e Produção de User Stories poderão ocorrer de forma paralela. O objetivo é que a partir ou durante o levantamento de requisitos realizados pelas equipes das fábricas de softwares contratadas, o responsável técnico (Product Owner), que será um representante da equipe efetiva da SESP/STI, produzirá as User Stories e um Backlog do produto.

Figura 59 – Processo de Testes para a SESP/STI



Considerando que a SESP/STI e as fábricas contratadas não trabalham efetivamente com metodologias ágeis, o fluxo foi idealizado para que se repita a cada ciclo de entrega (Release), independente da metodologia de gestão do projeto. Dessa forma, a partir de um levantamento preliminar de requisitos, será produzida uma lista de User Story e um Backlog correspondente, que servirão de base para a construção dos planos de testes. A cada início de ciclo de entrega deve-se garantir que as User Story serão priorizadas por meio do Backlog, e que essas serão detalhadas até o ponto de todos possuírem suas respectivas listas de critérios de aceitação.

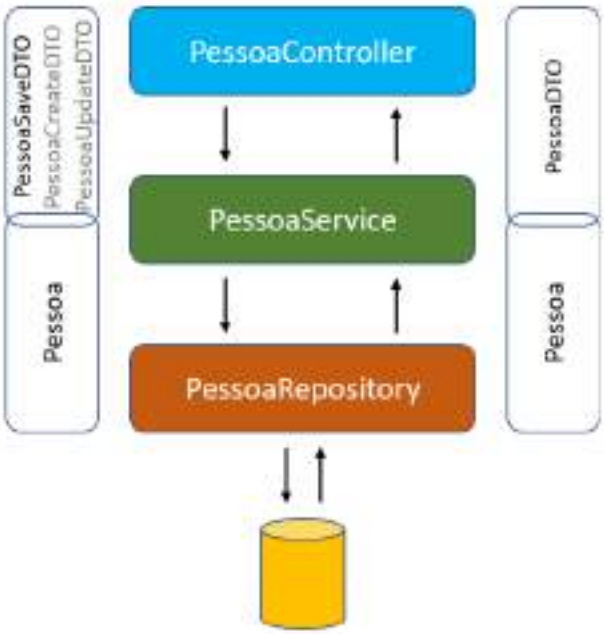
Os planos de testes para testes Unitário e de Integração devem ser o mais simples possíveis, visto que, o objetivo maior é a partir do Backlog priorizado, defina-se para quais elementos de software serão produzidos os testes Unitários e os testes de Integração automatizados, seguindo os padrões de referência estabelecidos para ambos os tipos de testes.

6.2.2.1 Planejamento de Testes Unitários

Os testes Unitários caracterizam principalmente pelo fato de que o elemento de software (método, classe ou componente) deve ser testado de forma isolada em relação as suas dependências. Em uma arquitetura de software existem inúmeros níveis de abstrações, e um dos primeiros passos para se definir um plano de testes unitário é definir em quais níveis de abstrações serão aplicados os testes unitários.

Considerando a arquitetura base proposta para os projetos de microsserviços, a recomendação é a produção de testes unitários automatizados para a abstração das classes que representam as regras de negócio, no caso, as classes com sufixo Service, ilustrada na figura 60.

Figura 60 – Arquitetura Base dos Projetos de Microsserviços



A justificativa para a implementação de testes unitários apenas para as classes de negócio são:

1. As regras de negócio são pontos críticos de qualquer sistema e devem ser garantidas durante o desenvolvimento do software e em manutenções futuras;

2. Implementar e manter testes unitários para todas os níveis de abstrações de um software requer um esforço muito grande;

3. Considerando que esta proposta contempla testes de integração, que serão produzidos a partir dos endpoint dos microsserviços, o restante dos requisitos serão testados na forma de verificações das funcionalidades de persistências e comunicações com outros elementos de softwares distribuídos.

Essa recomendação pode ser estendida para outras arquiteturas, como por exemplo, em projetos que usem os frameworks JSF e Spring, a recomendação seria a mesma, implementar testes unitários para as classes que representam as abstrações das regras de negócio.

6.2.2.2 Planejamento de Testes de Integração

Os testes de Integração tem como principal objetivo a garantia de que todas as dependências externas de um artefato de software em teste, estejam funcionando de forma adequada.

Em uma arquitetura de software baseada em microsserviços, existem vários componentes de software distribuídos e que precisam colaborar entre si para que os processos de negócio implementados a partir dessa rede de microsserviços sejam executados de forma adequada.

Os componentes de software distribuídos em uma arquitetura de microsserviços, vão desde a SGBDs, outros microsserviços e serviços de apoio, como por exemplo: serviços de autenticação, serviços de log, serviços de mensagem, etc.

Nesta proposta, os testes de integração serão implementados por meio da automação de testes envolvendo as classes com o sufixo Controller, previstas na arquitetura base dos projetos de microsserviços (Figura 03). A recomendação é a codificação de classes de testes que envolvam a chamada dos endpoints dos microsserviços, testando não apenas a integração, mas também as funcionalidades em um nível de abstração alto.

Nesse nível de abstrações as chamadas dos endpoints percorrerão todo o caminho das integrações entre os microsserviços, chegando até as bases de dados e as dependências entre os microsserviços. Nenhum conjunto de dados ou requisição deverá ser criado artificialmente (mokados) para isolamento, visto que, o que ser deseja é testar as funcionalidades e as integrações necessárias para que elas sejam realizadas.

6.2.3 Documentação Base de Requisitos para o Processo de Testes

Esta seção tem como objetivo apresentar uma proposta documentação de requisitos para sistemas de software como suporte ao processo de testes para o setor de desenvolvimento de software da SESP-MT. O objetivo é dar suporte à padronização de produção de testes em projetos implementados pela própria equipe efetiva da STI ou quando o desenvolvimento é terceirizado.

Como o cenário do trabalho no Projeto de Integração Contínua considera que o processo de desenvolvimento de software adequado ao modelo de integração contínua são as abordagens ágeis, o processo de testes também foi baseado em metodologias ágeis, mais fortemente ao Scrum. Sendo assim, a sugestão mínima para a documentação de requisitos, produção de critérios de aceitação e planos de testes está sendo proposto com base em histórias de usuários. A seguir esse assunto será

apresentados como base para a produção de cenários de testes, quer seja para fins de automação ou não.

6.2.3.1 História de Usuário (User Story)

Um projeto Scrum tem seus requisitos organizados em uma lista chamada Product Backlog. Essa lista representa os requisitos de um projeto. A forma mais utilizada para descrever os itens do backlog são as Histórias de Usuário (User Stories), ou apenas Histórias. Uma História pode ser escrita da seguinte forma (Cohn, 2004): Um Ator quer Algo por um Motivo.

Estrutura básica de uma história de usuário:

```
Eu como [ator]

Quero um/uma [algo]

Para que [motivo]
```

Ou

```
Eu como [ator]

Quando [gatilho acionador da necessidade]

Quero um/uma [algo]

Para que [motivo]
```

O **Ator** é a representação de um papel de usuário, ou seja, de um tipo de usuário, como Secretária, Cidadão, Diretor, Coordenador, Gestor, Policial, Vítima, etc.

O **Algo** é a meta do usuário ao utilizar o produto a ser desenvolvido. Deixa claro o que o usuário espera ao utilizar o produto. Algo como "emitir um relatório de sinistros de trânsito" ou "verificar média de ocorrências por bairro".

O **Motivo** funciona como uma justificativa e ajuda o Time Scrum a entender a História e a necessidade do Ator. "Para estabelecer medidas educativas de trânsito" e "definir medidas de segurança por demanda" são exemplos de motivos.

Esse formato deixa claro a ideia de História: comunicar ao Time um "requisito", ou seja, algo que precisa ser feito durante o desenvolvimento do produto ou projeto. Dessa forma, o time deve ler a história e entender o que deve ser feito, eventualmente esclarecendo alguma dúvida com o PO.

Contudo, somente uma descrição simples não é suficiente para o total entendimento e produção da funcionalidade ou ação. Dependendo do tipo de produto/projeto em desenvolvimento, do conhecimento sobre o negócio e do tempo que o time tem com o mesmo, uma simples frase pode ser o suficiente. Mas na maioria das vezes isso não ocorre.

O Scrum também tem uma solução para isso: Definição de Preparada (Definition of Ready).

Considera-se que uma história está preparada quando já foi refinada e detalhada o suficiente para que o Time possa trabalhar nela. Considerando que o próprio Time de Desenvolvimento

deverá estimar e atender o que está prescrito na história, nada mais justo do que o detalhamento ser realizado pelo Time de Desenvolvimento com o PO. Isso é exatamente a Definição de Preparada – uma combinação entre PO e Time de Desenvolvimento sobre o que é necessário para uma história ser considerada preparada e consequentemente poder ser trabalhada em uma Sprint. Essa definição promove a transparência, deixando claro para todo o Time Scrum o que é necessário em termos de preparação e alinhando as expectativas do Time de Desenvolvimento e do PO.

Caso seja necessário, além da Definição de Preparada muito simples, outros itens podem ser incluídos:

- Artefatos complementares, como Modelos de Caos de Uso e Modelos de Negócio, Esboços de Telas, Maquetes, etc.;
- Critérios para a aceitação;
- Capacidade do time (ter os profissionais adequados para a realização da história, etc).

6.2.3.2 Padrão INVEST para Construção de User Story

A palavra INVEST é resultado da união das iniciais de cada aspecto de uma boa user story.

Independente

Um incremento de qualquer tipo deve ser independente, uma vez que estamos buscando o desenvolvimento incremental e ágil.

Negociável

A história de usuário deve ser passível de negociação, para que o time tenha sempre a opção de atuar em fatias menores do que se está pedindo e opinar nas mesmas.

Valiosa

Evite entregas não valiosas. Não entregar serviços e produtos apenas por entregar algo. Foque no valor daquela fatia (user story) e não na fábrica de entregas.

Estimável

Se não é possível estimar o incremento a ser produzido é sinal que ainda temos pouco conhecimento sobre o que deve ser feito. Considere cortar em uma entrega menor e de conhecimento da equipe. Por isso, tudo deve ser negociável.

Pequena (Small)

O ideal é sempre entregar pequenas coisas com alto valor. Então, ser pequena é uma característica básica de uma User Story. Busque constantemente isso!

Testável

O que foi produzido pelo time de desenvolvimento deve ser passível de testes. É testando que o cliente vai atestar que o que foi feito, está de acordo com o que ele pediu. Aqui, é fundamental que os critérios de aceite estejam escritos de forma a complementar a User Story.

6.2.3.3 Critérios de Aceitação

Os critérios de aceitação são expressos por uma lista de itens para verificar se a User Story foi implementada de acordo com o que o PO queria e se atende aos requisitos do usuário.

É um desafio para o Product Owner criar e descrever os limites de User Stories com base em informações limitadas de seus usuários, mas entender quais são as entregas esperadas do produto é muito importante para alinhar com o cliente e time de desenvolvimento aquilo que está sendo feito.

No final de cada Sprint, na reunião de Revisão, o Product Owner utiliza os critérios de aceitação para entender se as entregas concluídas estão dentro do esperado e se pode aceitar ou rejeitar as entregas individuais e suas histórias de usuário.

Esses Critérios (itens) surgem de perguntas que a equipe faz ao PO no momento em que a User Story está sendo descrita, na busca por obter mais detalhes do que deve ser implementado.

Os critérios de aceitação costumam incluir itens de usabilidade, desempenho, funcionalidade, tratamento de erros, etc.

Critérios de Aceitação são expressos por enunciados pequenos e de fácil entendimento. São utilizados para determinar quando a funcionalidade produzida pelo Time de Desenvolvimento está completa e, assim, nada mais deve ser adicionado a ela.

Como base nos Critérios de Aceitação a equipe deve escrever os Testes de Aceitação, pois ao final do desenvolvimento são esses testes que dirão se a funcionalidade foi implementada de forma a ser aceita ou não pelo PO.

Também é possível que as User Stories e os critérios de aceitação apoiem a automação dos testes unitários e os testes de integração, considerando a hipótese dos testes unitários cobrirem as regras de negócio e critérios de validação dos dados, e os testes de integração podem validas as operações de persistência e recuperação de informações relacionadas com os critérios de aceitação.

Se os testes ainda forem criados com uso de ferramentas específicas para automatização da execução dos mesmos, isso garantirá maior agilidade e qualidade do produto.

Trabalhar com User Stories e com Critérios de Aceitação bem definidos garante que possamos chegar no melhor possível a ser entregue com o projeto, gerando qualidade e aceitação daquilo que a equipe de desenvolvimento produziu.

Exemplos de User Stories são ilustrados nas figuras 61 e 62:

Figura 61 – Exemplo 1 de User Stories

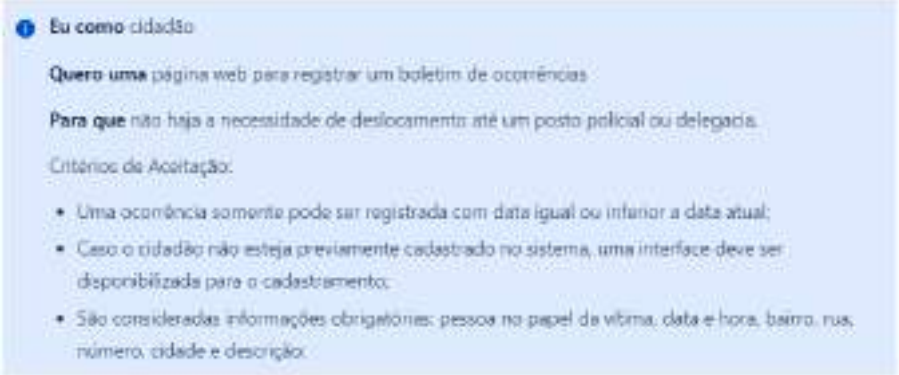
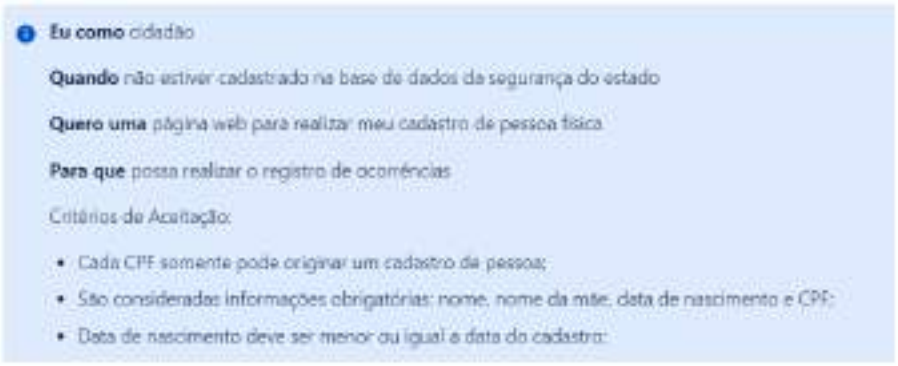


Figura 62 – Exemplo 2 de User Stories



6.2.3.4 Desenvolvimento de Planos de Testes

A partir das histórias de usuário produzidas e refinadas ao ponto serem consideradas Preparadas, o passo seguinte no processo de testes é produzir os planos de testes. Como nas metodologias ágeis, e especificamente no Scrum, a ideia é produzir documentação suficiente para o time de desenvolvimento, os planos de testes devem seguir o mesmo objetivo. Dessa forma, a sugestão é produzir planos de testes estritamente enxutos e de fácil entendimento, quer sejam para testes que serão automatizados ou não.

6.2.4 Plano de Testes Unitários

Este documento tem como objetivo apresentar uma proposta de plano de testes para o setor de desenvolvimento de software da SESP-MT. Nesta proposta são previstos os planos para testes unitários e testes de integração.

Os testes unitários são produzidos pela automação de testes das classes de serviços, que representam as classes de negócio dos projetos dos microsserviços desenvolvidos para o protótipo funcional.

6.2.4.1 Microsserviço Registro Geral

Entidade: Pessoa

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recebida como confirmação da operação
Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante
Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada
Entidade com ausência de dados obrigatórios (nome, nome da mãe, data de nascimento, cpf)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade
Data de Nascimento menor ou igual a data atual	Mensagem de exceção informando que data de nascimento deve ser inferior ou igual a data atual.

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de pessoa não localizada
ID Nulo	Mensagem de exceção de inconsistência de ID

6.2.4.2 Microsserviço Carteira Funcional

Entidade: Policial

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recebida como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada
Entidade com ausência de dados obrigatórios (referência a uma instância de Batalhão + os campos obrigatórios de pessoa)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de entidade não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Policial não localizado
ID Nulo	Mensagem de exceção de inconsistência de ID

Entidade: Administrativo

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recebida como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de inclusão realizada
Entidade com ausência de dados obrigatórios (referência a uma instância de Setor + os campos obrigatórios de pessoa)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Administrativo não localizado
ID Nulo	Mensagem de exceção de inconsistência de ID

6.2.4.3 Microsserviço Gerenciamento de Ocorrências

Entidade: Boletim

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recebida como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada
Entidade com ausência de dados obrigatórios (data e hora, bairro, rua, numero, descricao)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade
Lista de vítimas vazia	Mensagem de exceção informando que a informação vítimas não pode ser ausente

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Boletim não localizado
ID Nulo	Mensagem de exceção de inconsistência de ID

Entidade: Suspeito

Operação: Inclusão de Suspeito em Boletim
Pré-condição: nenhuma
Pós-condição: Suspeito presente na lista de suspeitos de um boletim de ocorrência

Variações	Resultado Esperado
Entidade correta	Mensagem de inclusão realizada
Entidade com ausência de dados obrigatórios (características, referência a uma pessoa, referência a um boletim)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
ID Boletim e ID Suspeito existente	Entidade conhecida recebida
ID Boletim inexistente e ID Suspeito existente	Mensagem de Boletim não inexistente
ID Boletim existente e ID Suspeito inexistente	Mensagem de Suspeito não inexistente
ID Boletim ou ID Suspeito nulo	Mensagem de exceção de inconsistência de ID

Entidade: Vítima

Operação: Inclusão de Vítima em Boletim
Pré-condição: nenhuma
Pós-condição: Vítima presente na lista de vítimas de um boletim de ocorrência

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada
Entidade com ausência de dados obrigatórios (características, referência a uma pessoa, referência a um boletim)	Mensagem de exceção listando os campos com ausência de valores
Entidade nula	Mensagem de exceção contendo mensagem de obrigatoriedade

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade resultante

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada
Entidade Correta e Chave Nula	Mensagem de exceção de chave obrigatória
Entidade com dados inconsistentes e Chave Correta	Mensagem de exceção de dados inconsistentes

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida

Variações	Resultado Esperado
ID Boletim e ID Vítima existente	Entidade conhecida recebida
ID Boletim inexistente e ID Vítima existente	Mensagem de Boletim não inexistente
ID Boletim existente e ID Vítima inexistente	Mensagem de Vítima não inexistente
ID Boletim ou ID Vítima nulo	Mensagem de exceção de inconsistência de ID

6.2.5 Plano de Testes de Integração com o Banco de Dados

Este documento tem como objetivo apresentar uma proposta de plano de testes para o setor de desenvolvimento de software da SESP-MT, neste caso os testes critérios para testes de integração com o banco de dados. O objetivo é verificar ser as operações de persistência e recuperação das informações estejam e se mantém corretas.

6.2.5.1 Microsserviço Registro Geral

Entidade: Pessoa

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recuperada do banco como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade recuperada do banco após alteração

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de Pessoa não localizada

Operação: Exclusão
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade não presente na base de dados após operação

Variações	Resultado Esperado
ID correto e existente	Mensagem de Exclusão realizada
ID inexistente	Mensagem de exceção de exclusão não realizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

6.2.5.2 Microsserviço Carteira Funcional

Entidade: Policial

Operação: Inclusão

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de pessoa não localizada

Pré-condição: nenhuma
Pós-condição: entidade recuperada do banco como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade recuperada do banco após alteração

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de entidade não localizada

Operação: Exclusão
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade não presente na base de dados após operação

	Resultado Esperado
ID correto e existente	Mensagem de Exclusão realizada
ID inexistente	Mensagem de exceção de exclusão não realizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Policial não localizado
ID Nulo	Mensagem de exceção de inconsistência de ID

Entidade: Administrativo

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recebida como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade recuperada do banco após alteração

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada

Operação: Exclusão
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade não presente na base de dados após operação

Variações	Resultado Esperado
ID correto e existente	Mensagem de Exclusão realizada
ID inexistente	Mensagem de exceção de exclusão não realizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Administrativo não localizado
ID Nulo	Mensagem de exceção de inconsistência de ID

6.2.5.3 Microsserviço: Gerenciamento de Ocorrências

Entidade: Boletim

Operação: Inclusão
Pré-condição: nenhuma
Pós-condição: entidade recuperada do banco como confirmação da operação

Variações	Resultado Esperado
Entidade correta	Mensagem de inclusão realizada

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados confirmados na entidade recuperada do banco após alteração

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada

Operação: Exclusão
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade não presente na base de dados após operação

Variações	Resultado Esperado
ID correto e existente	Mensagem de Exclusão realizada
ID inexistente	Mensagem de exceção de exclusão não realizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

Variações	Resultado Esperado
ID correto e existente	Entidade conhecida recebida
ID inexistente	Mensagem de Boletim não localizado

Entidade: Suspeito

Operação: Inclusão de Suspeito em um Boletim
Pré-condição: nenhuma
Pós-condição: Suspeito presente na lista de suspeitos de um boletim de ocorrência recuperado do banco

Variações	Resultado Esperado
Entidade correta	Mensagem de Inclusão realizada

Operação: Alteração de Suspeito de um Boletim
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados de um Suspeito pertencente a um Boletim

Variações	Resultado Esperado
Entidade Correta e Chave Correta (boletim e suspeito)	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta (boletim ou suspeito)	Mensagem de exceção de chave não localizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

Variações	Resultado Esperado
ID Boletim e ID Suspeito existente	Entidade conhecida recebida
ID Boletim inexistente e ID Suspeito existente	Mensagem de Boletim não inexistente
ID Boletim existente e ID Suspeito inexistente	Mensagem de Suspeito não inexistente

Entidade: Vítima

Operação: Inclusão de Vítima em Boletim
Pré-condição: nenhuma
Pós-condição: Vítima presente na lista de vítimas de um boletim de ocorrência recuperado do banco

Variações	Resultado Esperado
Entidade correta	Mensagem de inclusão realizada

Operação: Alteração
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Dados alterados de uma Vítima pertencente a um Boletim

Variações	Resultado Esperado
Entidade Correta e Chave Correta	Mensagem de Alteração realizada
Entidade Correta e Chave Incorreta	Mensagem de exceção de chave não localizada

Operação: Consulta por ID
Pré-condição: Inclusão de uma entidade conhecida
Pós-condição: Entidade conhecida recuperada do banco de dados

Variações	Resultado Esperado
ID Boletim e ID Vítima existente	Entidade conhecida recebida
ID Boletim inexistente e ID Vítima existente	Mensagem de Boletim não inexistente
ID Boletim existente e ID Vítima inexistente	Mensagem de Vítima não inexistente

6.3 Análise e Definição de Ferramentas para Automação de Testes

Esta seção contém as informações a respeito da análise e definições a respeito das ferramentas de testes unitários a serem adotadas na área de desenvolvimento de software da Secretaria de Segurança Pública do Estado de Mato Grosso (SESPMT).

6.3.1 Ferramentas e bibliotecas de testes unitários

Existem diversas ferramentas de testes unitários para cada linguagem de programação.

Para linguagem Java, que é o caso de nossas aplicações, as principais ferramentas de testes do mercado são:

- **JUnit:** executor de teste

- **Mockito:** para simular dependências
- **Wiremock:** para eliminar serviços externos
- **Pact:** para escrever testes CDC (Consumer driven contract)
- **Selenium:** para escrever testes ponta a ponta orientados pela UI

No cenário onde este projeto se desenvolve, é caracterizado por aplicações com arquitetura de microsserviços utilizando Spring Boot, cujos microsserviços fornecem uma interface REST que pode ser chamada via HTTP. Em nossos endpoints a camada service irá buscar informações de um banco de dados.

As tecnologias escolhidas para serem utilizadas no desenvolvimento desse projeto foram:

- Spring Boot – Versão 2.3.4
- IDE Eclipse - para desenvolvimento
- Maven para controle de dependências e compilação do projeto
- Giltab para gerenciamento do código fonte

Considerações sobre a definição

Dado o contexto do nosso cenário as ferramentas a serem utilizadas que melhor atendem ao ao proposito são:

- **JUnit**
- **Mockito**
- **Wiremock**

A Ferramenta Pact não será necessária pois não realizaremos testes de Contrato no momento e a Selenium não se aplica pois não teremos testes de UI.

6.4 Convenções no desenvolvimento de testes unitários e testes de integração

Esta seção tem como objetivo definir convenções no desenvolvimento de testes unitários, testes de integração afim de padronizar e simplificar sua documentação e manutenção para os desenvolvedores da Secretária de Segurança Pública do Estado de Mato Grosso (SESPMT).

6.4.1 Nomenclatura das classes de teste

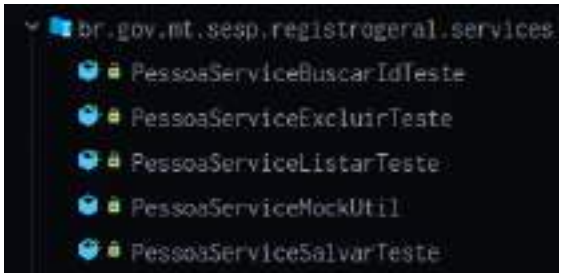
Durante a implementação de testes seja qual seu tipo, é necessário ter em mente que o mesmo será consultado por desenvolvedores que não participaram de sua implementação sendo essencial para identificar falhas durante os testes desses recursos.

Ao implementar testes seja unitário, funcional ou de integração recomenda-se utilizar uma classe de teste por funcionalidade. Para isso, é preciso utilizar a seguinte fórmula para nomenclatura das classes de teste nome da classe testada + nome do método testado + sufixo test. Por exemplo, a classe responsável pelas regras de negócio da entidade Pessoa possui as seguintes funcionalidades:

- Salvar uma pessoa
- Buscar uma pessoa por identificador
- Listar pessoas
- Excluir uma pessoa

Desse modo, para implementar testes unitários relacionados à esse conjunto de regras será necessário criar uma classe de teste para cada recurso como demonstrado na figura 63.

Figura 63 – Exemplo de nomenclatura de testes



Contudo, caso haja a necessidade de aplicar testes de integração e testes unitários na mesma classe deve-se utilizar um sufixo para que seja mantido a semântica dos testes afim de obedecer a premissa na simplicidade na gerencia dos testes. Assim, no cenário do microsserviço de carteira funcional os testes de unitários e de integração com o banco foram separados através dos sufixos **IBD** e **Unit**, onde **IBD** corresponde a abreviação para Integração Banco de Dados e **Unit** para testes unitários, conforme exemplificado na figura 64.

Assim, não haverá grande esforço para dar manutenção ou corrigir testes já existentes, pois cada caso de teste possui uma única funcionalidade que está sendo testada relacionada ao nome da classe. Portanto, utilizando apenas as mensagens de execução dos testes é possível inferir quais funcionalidades estão sendo testadas, ilustrado na figura 65.

Além disso, seguir essa convenção facilita na criação de novos testes e simplifica para o desenvolvedor no planejamento de quais são as funcionalidades críticas que devem ter uma maior cobertura de teste.

Ainda, combinar essas práticas com as convenções de nomenclatura de métodos apresentadas nesse documento auxilia, por exemplo, um desenvolvedor que não está envolvido no desenvolvimento de testes reportar e localizar um possível erro nos testes. Isso foi ilustrado na figura 66.

Figura 64 – Sufixo para testes Unitários e Integração com BD

Figura 65 – Exemplo ampliado de nomenclatura de casos de testes

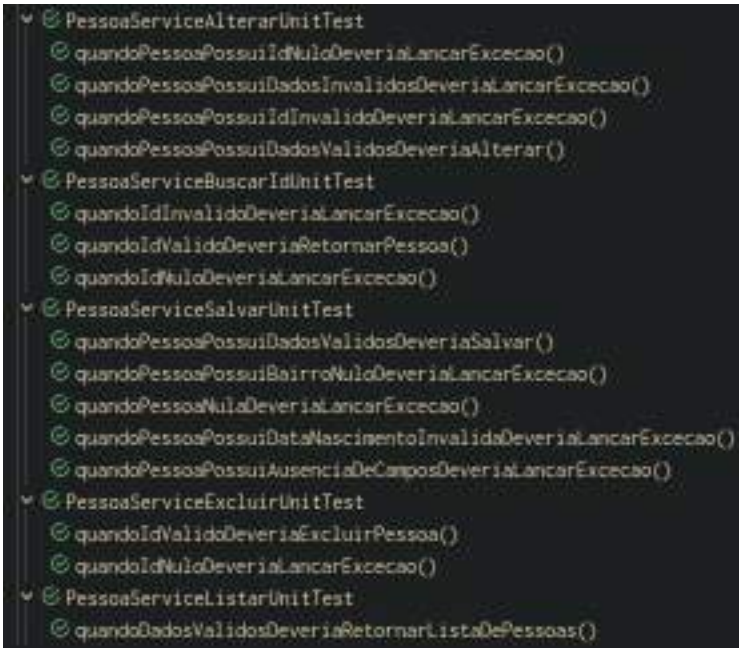


Figura 66 – Rastreabilidade de falhas nos testes

Desse modo, utilizando apenas a pilha de erros do Java é possível inferir que o erro ocorrido está relacionado a funcionalidade de busca por id da classe de serviço da entidade Pessoa.

6.4.2 Nomenclatura de membros da classe

Para membros de classe é recomendado nomear de modo a descrever sua função no teste como demonstrado no exemplo abaixo utilizar o sufixo **mock** facilita a leitura e manutenção futura do código.

```
@ExtendWith(MockitoExtension.class)
public class PessoaServiceTest {
  @InjectMocks
  private PessoaService pessoaService;
  @Mock
  private BairroRepository bairroRepositoryMock;
  @Mock
  private ModelMapper modelMapperMock;
  @Mock
  private PessoaRepository pessoaRepositoryMock;
}
```

Código 6.2 – Uso do sufixo mock

Ainda, uma nomenclatura alternativa para a instância do objeto que está sendo testado, nesse exemplo a instância de pessoaService, de SUT (System Under Test).

```
@ExtendWith(MockitoExtension.class)
public class PessoaServiceTest {
  @InjectMocks
  private PessoaService sut;
  @Mock
  private BairroRepository bairroRepositoryMock;
  @Mock
  private ModelMapper modelMapperMock;
  @Mock
  private PessoaRepository pessoaRepositoryMock;
}
```

Código 6.3 – Uso da abreviação sut

6.4.3 Nomenclatura de variáveis locais

Para nomear variáveis locais deve-se evitar nomes genéricos e os chamados números mágicos como no exemplo abaixo:

```
  @Test
  public void quandoIdValidoDeveriaRetornarPessoa() {
    var pessoaDTOMock = umaPessoaDTO();
    when(pessoaRepository.findById(anyLong())).thenReturn(Optional.of(umaPessoa()));
    when(modelMapper.map(any(), eq(PessoaDTO.class))).thenReturn(pessoaDTOMock);

    var expected = pessoaService.buscar(1L);

    verify(modelMapper, times(1)).map(any(), eq(PessoaDTO.class));
    verify(pessoaRepository, times(1)).findById(anyLong());
  }
```

Código 6.4 – Nomenclatura de variáveis locais

Nesse exemplo, nas linha 7 o número inserido nas chamadas dos métodos não possui um significado claro para quem está lendo o código. Para melhorar a legibilidade pode-se substituir esse número com a definição de uma constante.

```
1  @Test
2  public void quandoIdValidoDeveriaRetornarPessoa() {
3    var pessoaDTOMock = umaPessoaDTO();
4    when(pessoaRepository.findById(anyLong())).thenReturn(Optional.of(umaPessoa()));
5    when(modelMapper.map(any(), eq(PessoaDTO.class))).thenReturn(pessoaDTOMock);
6
7    long ID_VALIDO = 1L;
8
9    var expected = pessoaService.buscar(ID_VALIDO);
10
11   verify(modelMapper, times(1)).map(any(), eq(PessoaDTO.class));
12   verify(pessoaRepository, times(1)).findById(anyLong());
13 }
```

Código 6.5 – Uso de constantes

6.4.4 Nomenclatura de métodos

Existem várias convenções para a nomenclatura de métodos, nesse documento serão apresentada duas opções sendo que cada uma delas possui sua própria variação.

6.4.4.1 Nome do método, Estado sob teste e Comportamento Esperado

Essa convenção visa descrever qual é o comportamento do método testado sob um determinado estado. Assim, o nome do teste serve de documentação para quem for dar suporte ou precisar altera-lo futuramente. Seguindo o exemplo abaixo é possível inferir que o método findById deve retornar uma pessoa quando um identificador válido for inserido.

```
1 @Test
2 public void findById_idValido_retornaPessoa(){}
```

Código 6.6 – Nome de método não recomendado

Contudo, a desvantagem dessa convenção é a possível refatoração caso o nome do método seja alterado no futuro.

6.4.4.2 DEVERIA ter o comportamento esperado QUANDO determinado estado sob teste acontecer

Assim como na convenção anterior o nome do método auxilia na nomenclatura, legibilidade e manutenção dos testes.

```
1 @Test
2 public void quandoIdValidoDeveriaRetornarUmaPessoa(){}
```

Código 6.7 – Nome de método recomendado

Ou ainda, utilizando a convenção em inglês:

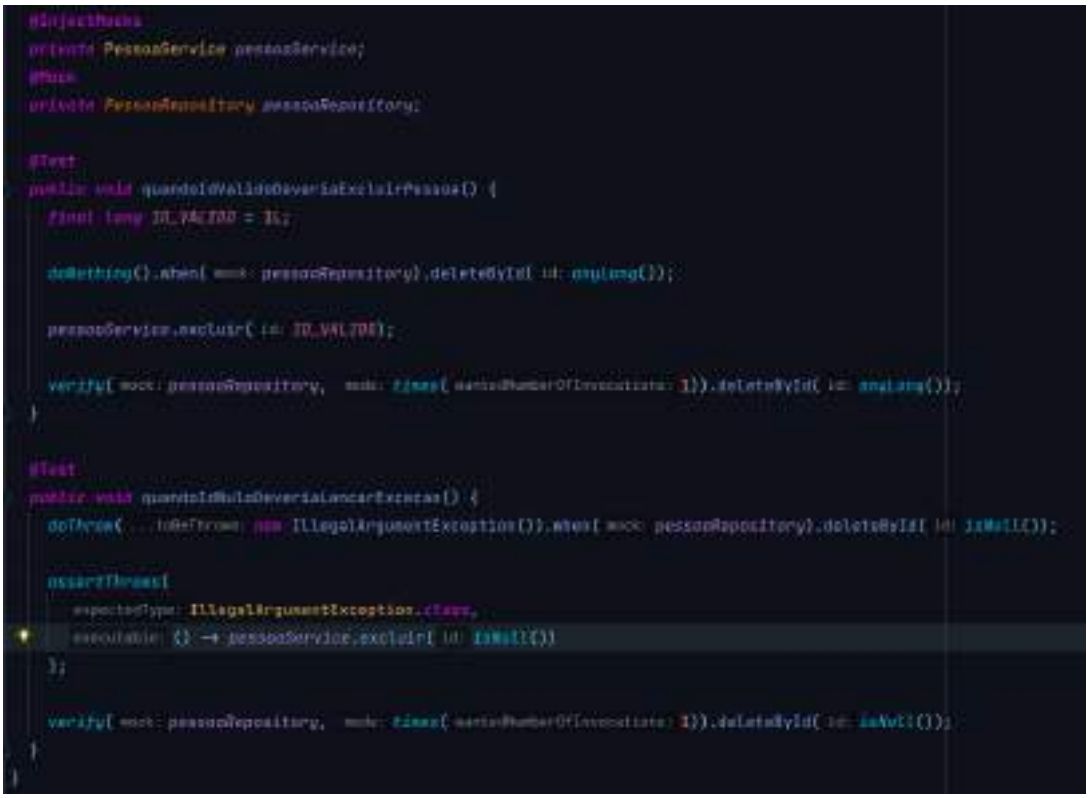
```
1 @Test
2 public void whenValidIdShouldReturnPessoa(){}
```

Código 6.8 – Nome de método recomendado em Inglês

Assim, recomenda-se utilizar este padrão pela facilidade em cumprir os requisitos do sistema dado que o próprio nome do método convém como documentação. Ainda, para um desenvolvedor com pouca experiência em testes o nome do método irá simplificar a implementação já que possui tanto o estado que está sendo testado quanto seu retorno.

Desse modo, ao aplicar essas regras à classe de negócio **PessoaService** deve-se obter o resultado apresentado na figura 67.

Figura 67 – Exemplo de nomenclatura recomendada



6.5 Implementação de testes nos microsserviços

Esta seção tem como objetivo expor detalhes da implementação de testes unitários e de integração desenvolvidos para os microsserviços Registro Geral e Carteira Funcional.

6.5.1 Testes Unitário

No desenvolvimento de testes para o microsserviço Registro Geral, foi escolhido o teste unitário como um exemplo de implementação. Para isso, a classe **PessoaService** responsável pelas regras de negócio relacionadas à entidade Pessoa foi selecionada para a realização dos testes.

Assim, foram utilizadas as seguintes ferramentas também definidas no documento Análise e Definição da Ferramenta de Testes Unitários:

- JUnit
- Mockito

Assim, foi implementado os seguintes casos de testes seguindo o documento de convenção de testes unitários e integração para a classe **PessoaService**:

Classe de teste	Testes
PessoaServiceBuscarIdUnitTest	• quandoIdInvalidoDeveriaLancarExcecao • quandoIdNuloDeveriaLancarExcecao • quandoIdValidoDeveriaRetornarPessoa
PessoaServiceExcluirUnitTest	• quandoIdValidoDeveriaExcluirPessoa • quandoIdNuloDeveriaLancarExcecao
PessoaServiceListarUnitTest	• quandoDadosValidosDeveriaRetornarListaDePessoas
PessoaServiceSalvarUnitTest	• quandoDadosValidosDeveriaSalvarPessoa • quandoDataNascimentoInvalidaDeveriaLancarExcecao • quandoPessoaPossuiBairroNuloDeveriaLancarExcecao

Ainda, de modo a demonstrar a separação de testes através da nomeação de sua classe foram implementados testes unitários para o microsserviço de Carteira Funcional que também possui testes de integração. Desse modo, foram gerados os seguintes testes para o microsserviço **AdministrativoService**.

Classe de teste	Testes
AdministrativoBuscarIdUnitTest	• quandoIdValidoDeveriaRetornarAdministrativo • quandoAdministrativoPossuiIdInvalidoDeveriaLancarExcecao • quandoAdministrativoPossuiIdNuloDeveriaLancarExcecao
AdministrativoSalvarUnitTest	• quandoAdministrativoValidoDeveriaSalvar • quandoAdministrativoPossuiAusenciaDeCamposDeveriaLancarExcecao • quandoAdministrativoNuloDeveriaLancarExcecao
AdministrativoAlterarUnitTest	• quandoAdministrativoSalvoDeveriaAlterar • quandoAdministrativoPossuiIdNuloDeveriaLancarExcecao • quandoAdministrativoPossuiDadosInvalidosDeveriaLancarExcecao

Também foram gerados os seguintes testes para o microsserviço **PolicialService**.

Por ser um teste de unidade é necessário que a mesma esteja completamente isolada do sistema para garantir a integridade do teste e a aderência ao conceito de Teste Unitário. Para isso, foi utilizado a ferramenta Mockito, para gerar dados oriundo de dependências que não fazem parte da unidade em teste, como demonstrado no código abaixo:

```
1  @Test
2  public void quandoIdValidoDeveriaRetornarPessoa() {
3    var pessoaDTOMock = umaPessoaDTO();
4    when(pessoaRepository.findById(anyLong())).thenReturn(Optional.of(umaPessoa()));
5    when(modelMapper.map(any(), eq(PessoaDTO.class))).thenReturn(pessoaDTOMock);
6
```

Classe de teste	Testes
PolicialBuscarUnitTest	• quandoIdValidoDeveriaRetornarPolicial • quandoPolicialPossuiIdInvalidoDeveriaLancarExcecao • quandoPolicialPossuiIdNuloDeveriaLancarExcecao
PolicialSalvarUnitTest	• quandoPolicialValidoDeveriaSalvar • quandoPolicialPossuiAusenciaDeCamposDeveriaLancarExcecao • quandoPolicialNuloDeveriaLancarExcecao
PolicialAlterarUnitTest	• quandoPolicialSalvoDeveriaAlterar • quandoPolicialPossuiIdNuloDeveriaLancarExcecao • quandoPolicialPossuiDadosInvalidosDeveriaLancarExcecao

```
7     long ID_VALIDO = 1L;
8
9     var expected = pessoaService.buscar(ID_VALIDO);
10
11    verify(modelMapper, times(1)).map(any(), eq(PessoaDTO.class));
12    verify(pessoaRepository, times(1)).findById(anyLong());
13  }
```

Código 6.9 – Uso do Mockito

Nesse exemplo, o método que está sendo testado é o **buscar** cuja funcionalidade consiste em consultar o banco de dados utilizando um **id** inserido em seu argumento. Para isso, o método utiliza duas dependências: **pessoaRepository** e **modelMapper**.

A primeira é responsável pela persistência de dados no banco e o método utilizado pelo **buscar()** é o **findById()**, assim, como o intuito do teste unitário é verificar se a unidade está executando corretamente sua funcionalidade o retorno do método **findById()** não é relevante para o teste. O mesmo se aplica para a dependência do **modelMapper** utilizado para converter o objeto **Pessoa** para **PessoaDTO**. Desse modo, nas linhas 4 e 5 são definidas a entrada e saída das dependências que serão chamadas internamente pelo Mockito.

Após a chamada do método é utilizado o método **verify()** para garantir que a chamada das dependências foi feita corretamente.

6.5.2 Teste de Integração com o Banco de Dados

No microsserviço Carteira Funcional foi implementado testes de integração com o banco de dados através de sua camada de serviço. Desse modo, esses testes tem como objetivo garantir que haja conexão com o banco de dados.

Assim, os testes são aplicados na camada de serviço utilizando a chamada do método testado. Este teste de integração, diferente do teste aplicado no serviço de Registro Geral, é chamado de teste de caixa preta por priorizar apenas a saída do sistema em teste desconsiderando seu funcionamento interno.

```
1   @Test
2   public void quandoAdministrativoValidoDeveriaSalvar() {
3
4     var administrativo = criaAdministrativoSaveDTO();
```

```
5
6       var expected = sut.salvar(administrativo);
7
8       Assertions.assertNotNull(expected.getId(), "Administrativo deveria ter sido criado
            ");
9   }
```

Código 6.10 – Exemplo de Teste de Integração com BD

Nesse exemplo é aplicado um teste de criação de administrativo no microsserviço de Carteira Funcional. Como mencionado antes, as ações e comportamentos internos do microsserviço são desprezados admitindo como sucesso apenas o retorno esperado.

Desse modo, ao final de sua implementação foram gerados os seguintes testes de integração para a classe **AdministrativoService**:

Classe de teste	Testes
`AdministrativoBuscarIdIBDTest`	• `quandoIdValidoDeveriaRetornarAdministrativo` • `quandoAdministrativoPossuiIdInvalidoDeveriaLancarExcecao` • `quandoAdministrativoPossuiIdNuloDeveriaLancarExcecao`
`AdministrativoExcluirIBDTest`	• `quandoAdministrativoPossuiIdValidoDeveriaExcluir` • `quandoAdministrativoPossuiIdNuloValidoDeveriaExcluir`
`AdministrativoListarIBDTest`	• `quandoDadosValidosDeveriaListarAdministrativos`
`AdministrativoSalvarIBDTest`	• `quandoAdministrativoValidoDeveriaSalvar`
`AdministrativoAlterarIBDTest`	• `quandoAdministrativoSalvoDeveriaAlterar` • `quandoAdministrativoPossuirIdInvalidoDeveriaLancarExcecao`

Para a classe **PolicialService**, forema gerados os seguintes testes:

Classe de teste	Testes
`PolicialBuscarIBDTest`	• `quandoIdValidoDeveriaRetornarPolicial` • `quandoPolicialPossuiIdInvalidoDeveriaLancarExcecao` • `quandoPolicialPossuiIdNuloDeveriaLancarExcecao`
`PolicialExcluirIBDTest`	• `quandoPolicialPossuiIdValidoDeveriaExcluir` • `quandoPolicialPossuiIdNuloValidoDeveriaExcluir`
`PolicialListarIBDTest`	• `quandoDadosValidosDeveriaListarPoliciais`
`PolicialSalvarIBDTest`	• `quandoPolicialValidoDeveriaSalvar`
`PolicialAlterarIBDTest`	• `quandoPolicialSalvoDeveriaAlterar` • `quandoPolicialPossuirIdInvalidoDeveriaLancarExcecao`

6.6 Definição de ferramenta de análise de cobertura de testes

Utilização de testes automatizados e metodologias de testes tem ganhando força nos últimos anos e um grande aliado na hora de analisar a necessidade de novos testes em seu ambiente é uma ferramenta de cobertura de testes. O percentual de cobertura de testes é uma medida importante que quantifica o grau em que o código-fonte do programa foi testado.

De forma resumida, ele demostra qual a porcentagem do seu código os testes estão cobrindo. Essa categoria de ferramentas auxilia o desenvolvedor e os testadores a descobrir que um determinador cenário X que pensamos na hora de codificar (o famoso if no meio do caminho), não está sendo coberto em nenhum dos testes unitários. Interessante, não?

6.6.1 Ferramentas consideradas para análise

Há várias ferramentas de cobertura de código no mercado e selecionar uma para o projeto pode ser um desafio, vamos analisar agora as principais do mercado:

6.6.1.1 Cobertura

Cobertura é uma ferramenta Java gratuita que calcula a porcentagem de código acessado pelos testes. Ele pode ser usado para identificar quais partes de seu programa Java estão sem cobertura de teste. Baseia-se em jcoverage.

Disponível em: https://cobertura.github.io/cobertura/

6.6.1.2 EclEmma

O EclEmma é uma ferramenta de cobertura de código para Java. O EclEmma é um plugin gratuito, disponível para o Eclipse sob a licença Eclipse Public License. A ferramenta foi inspirada na biblioteca EMMA desenvolvida por Vlad Roubtsov.

EclEmma atualmente usa como core a biblioteca jaCoCo que fornece a tecnologia padrão para análise de cobertura de código em ambientes baseados em Java VM. O foco é fornecer uma biblioteca leve, flexível e bem documentada para integração com várias ferramentas de construção e desenvolvimento.

Disponível em: https://www.eclemma.org/

6.6.1.3 OpenClover

A ferramenta OpenClover ajuda a medir a cobertura de código para Java e Groovy e coleta mais de 20 métricas de código. Ajuda a exibir áreas não testadas de seu aplicativo. É uma das melhores ferramentas de cobertura de teste que ajuda você a combinar cobertura e métricas para encontrar o código mais arriscado.

Disponível em: http://openclover.org/

6.6.1.4 Visual Studio Enterprise (plugin Code Coverage)

O recurso de cobertura de código do Visual Studio ajuda você a determinar qual parte do código do seu projeto é testada por testes codificados como testes de unidade. A ferramenta permite que você visualize o conjunto anterior de resultados.

Disponível em: https://visualstudio.microsoft.com/pt-br/vs/features/testing-tools/

6.6.2 Critérios de escolha adotados para a ferramenta

Considerando nosso cenário com micro serviços, IDE de desenvolvimento Eclipse, utilizando a pipeline go gitlab com multi branchs os seguintes critérios foram definidos para a escolha das ferramentas.

1. Integração com o Maven
2. Plugins para eclipse
3. Relatório de Cobertura
4. Analise por Branch
5. Atualização constante

6.6.3 Matriz de decisão

Ferramenta	C1	C2	C3	C4	C5
Cobertura	X		X	X	
EclEmma(JaCoCo)	X	X	X	X	X
OpenClover	X	X	X	X	
Visual Studio			X	X	X

6.6.4 Considerações sobre a escolha

Diante da pesquisa e comparações das ferramentas para análise de cobertura de testes, a ferramenta EclEmma a que atendeu o maior número de critérios de escolha, além de recebe atualizações constante e foi de fácil instalação e configuração para utilizar

A seguir uma analise rápida sobre as principais características desfavoráveis das outras ferramentas.

Cobertura: apesar de muito boa não esta recebendo atualizações consideráveis a alguns anos e contem no momento da escrita desse artigo 196 Issues abertos no github (https://github.com/cobertura/cobertura/i para correção, e provavelmente a mesma não seja atualizada para novas versões do Java.

Figura 68 – Exemplo de Relatório da ferramenta EclEmma

OpenClover: Essa ferramenta esta a cerca de um ano sem receber novas atualizações que é o principal problema da mesma, além de ser mais complicada de configurar e utilizar.

Visual Studio Enterprise: Essa ferramenta é muito boa, porem ela só está disponível na versão paga do Visual Studio, o que necessita de uma troca de IDE pela equipe.

6.7 Estratégias de Geração de Massas de Dados para Testes

Esta seção tem como objetivo apresentar uma proposta de geração automática de dados para testes para o setor de desenvolvimento de software da SESP-MT. Nesta proposta são previstas operações de geração de massa de dados para três níveis de testes: testes unitários, testes de integração e testes de aceitação, sendo que o teste de integração é subdividido em dois estágios: integração com banco de dados e integração com o ambiente distribuído.

Para a formulação desta proposta foram analisadas duas estratégias para geração de massa de dados automática para testes:

- A primeira é a produção de dados em arquivos por meio de ferramentas que a partir de parâmetros estabelecidos geram arquivos contendo os dados em formatos como XML, JSON, CSV, HTML ou scripts SQL com comandos INSERT para a inclusão de dados diretamente nas tabelas do banco de dados;

- A segunda é a produção de dados por meio do uso de API que a partir de parametrizações e customizações produzem instâncias de entidades que representam objetos que podem ser persistidos no banco de dados.

A primeira estratégia tem como principal característica a facilidade de produção e inserção dos dados em uma base de dados, contudo é muito volátil às alterações de estruturas de dados ou relacionamentos, exigindo que os arquivos sejam gerados todas as vezes que acontece uma alteração na estrutura dos dados presentes nos arquivos. Os dados gerados em arquivos ou scripts SQL em arquivos não são validados por nenhum mecanismo depois que eles foram gerados.

A segunda estratégia exige um maior esforço inicial para a produção de dados para testes, visto que, necessitam da produção de código fonte que implemente as rotinas para a produção de tais dados. Contudo, nesta estratégia as alterações de estrutura de dados são facilmente incorporadas, visto que, basta alterar a rotina específica que produz os dados conforme as alterações das entidades que representam tais informações. Outra característica importante é que, como os dados são

gerados por meio da implementação no código fonte e este é versionado e gerenciado no fluxo de commits, as rotinas de geração de massa de dados farão parte desse versionamento, visto que, será realizado por meio de código fonte.

Considerando as características das duas estratégias possíveis, foi adotada a segunda estratégia, visto que na nossa avaliação, é uma estratégia que possibilita a alteração e evolução da produção das massas de dados conforme o software evolui.

6.7.1 Geração de Massa de Dados para Testes Unitários

Nos testes Unitários, a abordagem adotada foi a produção de testes caixa-preta nas classes de negócio. Essa decisão remete ao uso de dados produzidos internamente nos casos de testes, gerados em memória. A automação na geração de dados para os testes Unitários se dá pelo uso da biblioteca Mokito, incorporada no Framework Spring. Esse mecanismos de geração de dados para os testes Unitários está demonstrado no documento Implementação de testes nos microsserviços .

6.7.2 Geração de Massa de Dados para Testes de Integração

Como foi descrito no início deste documento, os teste de Integração foram divididos em dois níveis:

- Testes de Integração com mecanismos de persistência;
- Testes de Integração entre microsserviços dependentes e ambiente de execução;

A justificativa para tal divisão se dá pelo fato do Protótipo Funcional em desenvolvimento envolver a implementação de vários projetos de microsserviços, descrito no documento Projetos do Protótipo Funcional. Dessa forma, em cada projeto de um microsserviço existem as operações de persistência de dados pertinentes ao contexto do microsserviço em questão, mas também existem uma série de dependências externas, tais como: dependências de dados ou funcionalidades de outros microsserviços ou dependências com outros serviços necessários à execução, como por exemplo, serviços de mensageria, log, autenticação e autorização.

O protótipo funcional em desenvolvimento é constituído por cinco microsserviços concebidos sob a arquitetura em camadas e implementados com o framework Spring Boot, conforme descrito no documento Projetos do Protótipo Funcional. Nessa arquitetura, cada microsserviço possui um conjunto de classes de negócio (Service) e um conjunto de classes responsáveis pela persistência dos em banco de dados (Repository), que atuam no conjunto de tabelas restritas ao contexto do microsserviço em questão.

6.7.2.1 Massa de Dados para Testes de Integração com Mecanismo de Persistência

Em função da arquitetura adotada nos projetos dos microsserviços, que isolam as tabelas de dados no contexto particular de cada projeto, implica que cada microsserviço tem acesso de leitura e escrita somente nas tabelas do seu contexto de negócio, mesmo que a base de dados seja centralizada.

A automação dos testes de Integração demanda duas tarefas:

- Criar as rotinas de testes automatizados para verificar a integração do microsserviço com o mecanismo de persistência, descritos no documento Implementação de testes nos microsserviços;
- Gerar massa de dados automatizada para os testes de operações de persistência.

Todas as dependências de dados relacionados com outros microsserviços são resolvidas mediante invocação de serviços.

A geração de massa de dados para os testes de integração foi implementada por meio da ferramenta **Java Faker**[17], cujas principais características são:

1. Permite a produção de grandes quantidades de dados de forma automática;
2. Produz dados contextualizados em domínios de negócio;
3. Possui parametrização de localidade, inclusive para Português;
4. É uma API cujas rotinas de geração de dados são versionadas juntamente com o código fonte do projeto;

Nas rotinas de testes de Integração com banco de dados, a ferramenta Java Faker é usada para criar dados mais realistas e contextualizados, além de permitir gerar grandes quantidades de dados.

Os trechos de código a seguir demonstram o uso da ferramenta JavaFaker, para gerar dados que são usados para compor uma ocorrência (microsserviço de Ocorrências). O a seguir apresenta a implementação da classe VitimaMock, que produz instâncias da classe VitimaSaveDTO já com dados produzidos pelo **Java Faker**.

```
1  package br.gov.mt.sesp.gerenciamentoocorrencia.mock;
2
3  import br.gov.mt.sesp.gerenciamentoocorrencia.dtos.VitimaSaveDTO;
4  import com.github.javafaker.Faker;
5  import java.util.Locale;
6
7  public class VitimaMock {
8    private static final Faker faker;
9    static {
10     faker = new Faker(new Locale("pt-BR"));
11   }
12
13   public static VitimaSaveDTO criaVitimaSaveDto() {
14     var vitimaSaveDto = new VitimaSaveDTO();
15     vitimaSaveDto.setIdPessoa(faker.number().numberBetween(1L, 20L));
16     vitimaSaveDto.setCaracteristicas(faker.lorem().sentence(5));
17     return vitimaSaveDto;
18   }
19 }
```

Código 6.11 – Exemplo de uso do Faker para gerar dados de Vitima

O código seguinte apresenta a implementação da classe SuspeitoMock, que produz instâncias da classe **SuspeitoSaveDTO** já com dados produzidos pelo Java Faker.

[17] https://faker.readthedocs.io/en/master/index.html

```java
package br.gov.mt.sesp.gerenciamentoocorrencia.mock;

import br.gov.mt.sesp.gerenciamentoocorrencia.dtos.SuspeitoSaveDTO;
import com.github.javafaker.Faker;
import java.util.Locale;

public class SuspeitoMock {

  private static final Faker faker;

  static {
    faker = new Faker(new Locale("pt-BR"));
  }

  public static SuspeitoSaveDTO criaSuspeitoSaveDto() {
    var suspeitoSaveDto = new SuspeitoSaveDTO();
    suspeitoSaveDto.setIdPessoa(faker.number().numberBetween(1L, 20L));
    suspeitoSaveDto.setCaracteristicas(faker.lorem().sentence(5));
    return suspeitoSaveDto;
  }
}
```

Código 6.12 – Exemplo de uso do Faker para gerar dados de Suspeito

O código seguinte apresenta a implementação da classe BoletimOcorrenciaMock, que produz instâncias da classe BoletimOcorrenciaSaveDTO já com dados produzidos pelo Java Faker, inclusive agregando objetos produzidos pelas duas classes anteriores.

```java
package br.gov.mt.sesp.gerenciamentoocorrencia.mock;

import br.gov.mt.sesp.gerenciamentoocorrencia.dtos.BoletimOcorrenciaSaveDTO;
import br.gov.mt.sesp.gerenciamentoocorrencia.dtos.SuspeitoSaveDTO;
import br.gov.mt.sesp.gerenciamentoocorrencia.dtos.VitimaSaveDTO;
import com.github.javafaker.Faker;
import java.util.Arrays;
import java.util.Locale;
import java.util.concurrent.TimeUnit;
import static br.gov.mt.sesp.gerenciamentoocorrencia.mock.SuspeitoMock.
    criaSuspeitoSaveDto;
import static br.gov.mt.sesp.gerenciamentoocorrencia.mock.VitimaMock.criaVitimaSaveDto
    ;

public class BoletimOcorrenciaMock {
  private static final Faker faker;

  static {
    faker = new Faker(new Locale("pt-BR"));
  }

  public static BoletimOcorrenciaSaveDTO criaBoletimOcorrenciaSaveDto() {
    var boletimOcorrenciaSaveDTO = new BoletimOcorrenciaSaveDTO();
    boletimOcorrenciaSaveDTO.setSituacao("REGISTRADO");
    boletimOcorrenciaSaveDTO.setCep(faker.address().zipCode());
    boletimOcorrenciaSaveDTO.setIdBairro(1L);
    boletimOcorrenciaSaveDTO.setIdAdministrativo(1L);
    boletimOcorrenciaSaveDTO.setDataHora("29/10/2020 20:13:00");
    boletimOcorrenciaSaveDTO.setNumero(faker.number().digits(4));
    boletimOcorrenciaSaveDTO.setComplemento(faker.address().streetAddress());
    boletimOcorrenciaSaveDTO.setRua(faker.address().streetName());
    boletimOcorrenciaSaveDTO.setDescricao(faker.lorem().sentence(5));
```

```
    boletimOcorrenciaSaveDTO.setSuspeitos(
      Arrays.asList(criaSuspeitoSaveDto(),
                    criaSuspeitoSaveDto(),
                    criaSuspeitoSaveDto()
      )
    );
    boletimOcorrenciaSaveDTO.setVitimas(
      Arrays.asList(criaVitimaSaveDto(),
                    criaVitimaSaveDto()
      )
    );

    return boletimOcorrenciaSaveDTO;
  }

  public static BoletimOcorrenciaSaveDTO criaBoletimOcorrenciaSaveDtoComId() {
    var dto = criaBoletimOcorrenciaSaveDto();
    dto.setId(1L);
    return dto;
  }
}
```

Código 6.13 – Exemplo de uso do Faker para gerar dados de Vitima

É importante destacar que a ferramenta Java Faker foi encapsulada em classes Mock para que ferramenta não fique exposta para o restante do código de testes, visto que, se for necessário mudar a ferramenta de geração de dados, isso pode ser realizado de forma transparente.

```
@Test
  public void quandoVitimaValidaDeveriaSalvar() throws Exception {
    var suspeitoSaveDto = VitimaMock.criaVitimaSaveDto();

    long ID_BOLETIM_OCORRENCIA = 1;
    this.mvc.perform(
      MockMvcRequestBuilders.post(
        "/ocorrencias/" + ID_BOLETIM_OCORRENCIA + "/vitimas").contentType(
        MediaType.APPLICATION_JSON)
        .content(JSONConverter.asJsonString(suspeitoSaveDto)))
      .andExpect(status().isCreated());
  }
```

Código 6.14 – Exemplo de Teste que Salva os dados de Vitima

```
@Test
  public void quandoBoletimOcorrenciaValidoDeveriaSalvar() throws Exception {
    var boletimOcorrencia = BoletimOcorrenciaMock.criaBoletimOcorrenciaSaveDto();

    this.mvc.perform(
      MockMvcRequestBuilders.post("/ocorrencias").contentType(MediaType.
          APPLICATION_JSON)
        .content(JSONConverter.asJsonString(boletimOcorrencia)))
      .andExpect(status().isCreated());
  }
```

Código 6.15 – Exemplo de Teste que Salva os dados de Boletim de Ocorrência

Nos dois trechos de Código anteriores, os dados criados nas classes Mock são usados para testar as operações de persistência do microsserviço de Ocorrência.