

## DevOps – Quality Assurance - QA

Nesse nosso DoJo vamos abordar a **Garantia da Qualidade – QA (Quality Assurance)** no **DevOps**. Como gostamos muito de um conceito, o que é DevOps? Uma **cultura** composta por **boas práticas auxiliada por ferramentas** que permitem **entregar softwares muito mais rápidos e com muito menos custos** do que comparada a metodologias tradicionais de desenvolvimento de software e gerenciamento de infraestrutura.

### Porque utilizar DevOps?

Em uma empresa sem **DevOps** o lançamento de aplicações são atividades estressantes e de alto risco, que envolvem várias equipes diferentes. Por isso são conhecidos com '**a última milha**' tamanho são os esforços e as dificuldades encontradas. Com o advento do movimento Ágil foi cunhado o termo **DevOps** para a **integração entre as equipes de desenvolvimento, qualidade e infraestrutura**.

*Como **DevOps** é um tema muito extenso para um único DoJo vamos focar nossos esforços no **Quality Assurance – QA (Garantia da Qualidade)**.*

Vamos utilizar a temática de questionamentos para o desenvolvimento de nossas atividades, o intuito não é esgotar o nosso assunto, mas sim iniciar um fórum de discussão, despertando a curiosidade, mostrando um norte, gerar aquele 1% de melhora em nossas atividades. Vamos começar!

**Quais são os testes que podemos efetuar para que um código seja integrado em ambiente de produção com qualidade? Frameworks e Ferramentas.**

### BDD

Em nossa abordagem vamos focar nossos estudos em práticas **DevOps** é o primeiro será o **BDD (Behavior-Driven Design/Development ou Design/Desenvolvimento guiado por comportamento)**, onde é uma técnica voltada para o **comportamento da aplicação** e pessoas não técnicas compreendem os testes e podem escrever-los também.

Principal intuito do **BDD** é **manter uma linguagem única com todos os envolvidos no projeto (Stakeholders)**, ou seja, qualquer pessoa seja ele **conhecedora de técnicas de programação ou não consegue entender o código**.

Para mostrar a aplicabilidade do BDD, vamos utilizar uma história de usuário. Intuito dessa história é simples, efetuar uma consulta de projetos (nome e descrição) pré-cadastros e ordenados por nome.

### #ConsultarProjetos

COMO Usuário DESEJO consultar todos os projetos cadastrados.

#### Critérios de Aceitação

- 1) SENDO eu um Usuário EU QUERO consultar todos os projetos ENTÃO o Sistema deverá apresentar os campos de busca.
- 2) DADO que a consulta seja realizada, EU QUERO que os projetos sejam ordenados por ordem alfabética crescente por nome. EU QUERO também que seja apresentado as opções de **#AlterarProjeto**, **#ExcluirProjeto** e **#DetalharProjeto**.

#### Tipos dos campos

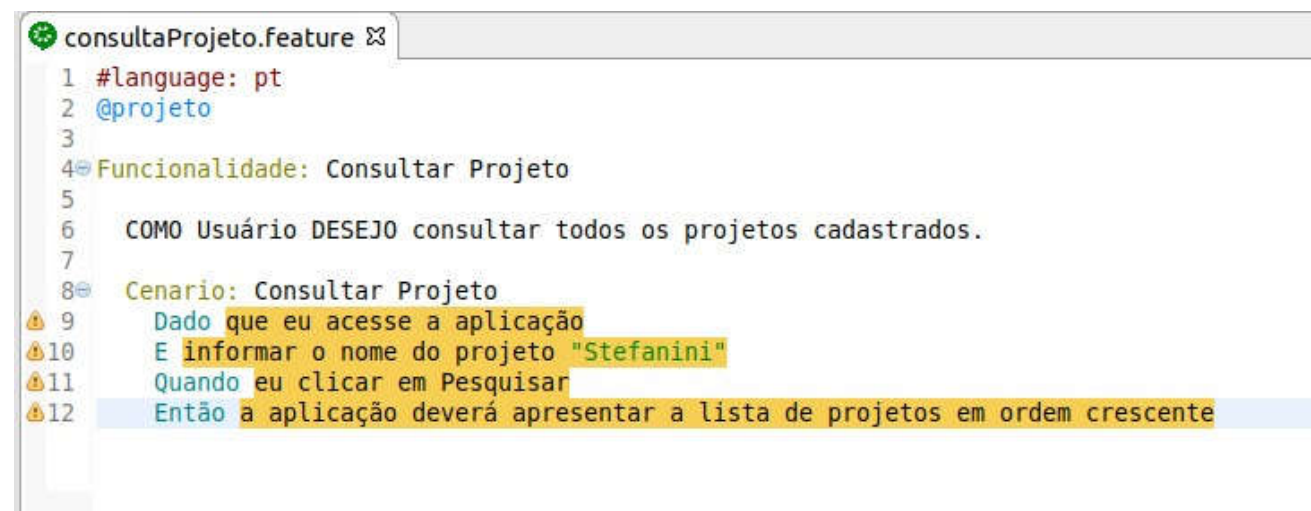
Nome do Campo	Tipo	Tamanho
Ações	Editar/Excluir/Detalhar	Não se aplica
Nome	Alfanumérico	40 caracteres
Descrição	Alfanumérico	200 caracteres

Valor para o negócio: Médio

### História 01 - Exemplo de uma história de usuário

No nosso exemplo estamos utilizando o **Cucumber** por ser o **framework mais conhecido para testes de comportamentos e implementação em várias linguagens de programação**.

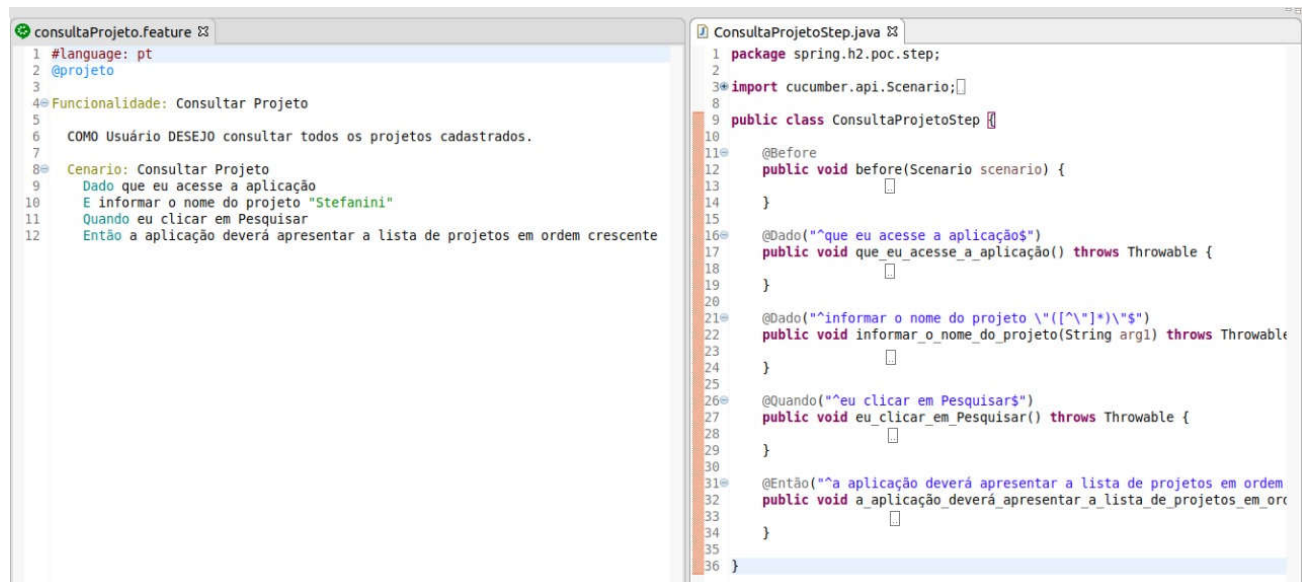
Durante o planejamento das atividades é gerado o código BDD com base na história do usuário **#ConsultaProjeto**.



```
consultaProjeto.feature
1 #language: pt
2 @projeto
3
4 Funcionalidade: Consultar Projeto
5
6     COMO Usuário DESEJO consultar todos os projetos cadastrados.
7
8     Cenário: Consultar Projeto
9         Dado que eu acesse a aplicação
10        E informar o nome do projeto "Stefanini"
11        Quando eu clicar em Pesquisar
12        Então a aplicação deverá apresentar a lista de projetos em ordem crescente
```

Figura 01 - BDD da história consulta de projeto

Após a implementação da funcionalidade o comportamento do BDD é desenvolvido pela equipe de qualidade.



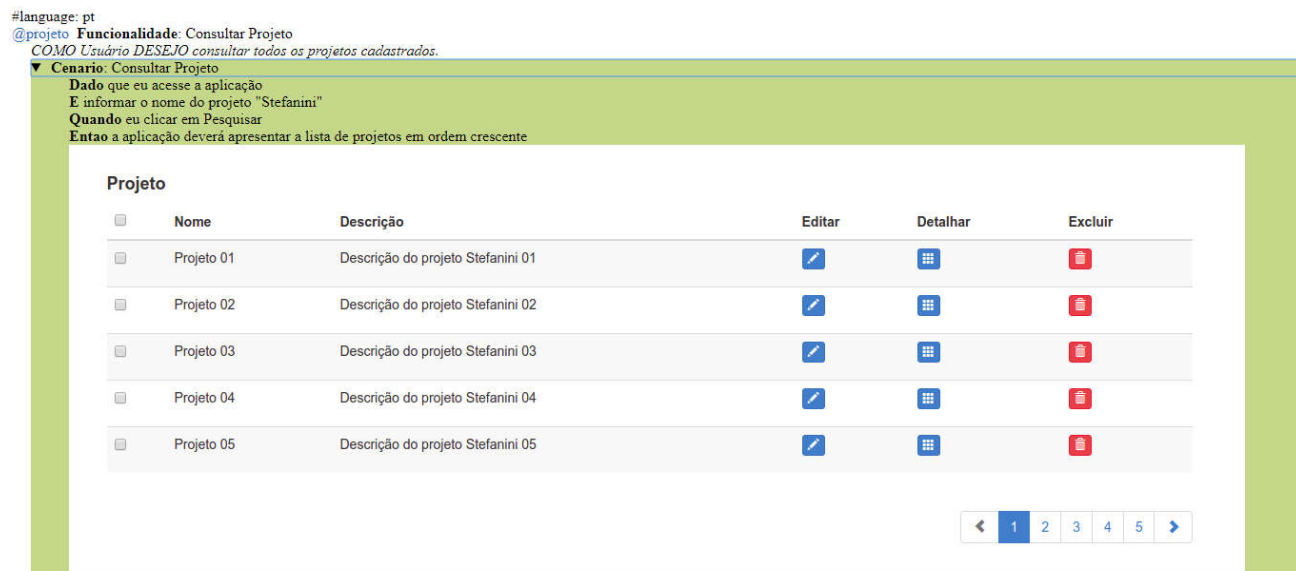
The image shows two side-by-side code editors. The left editor, titled 'consultaProjeto.feature', contains BDD feature and scenario definitions in Portuguese. The right editor, titled 'ConsultaProjetoStep.java', contains the corresponding Java step definitions for the feature.

```
1 #language: pt
2 @projeto
3
4 Funcionalidade: Consultar Projeto
5
6 COMO Usuário DESEJO consultar todos os projetos cadastrados.
7
8 Cenário: Consultar Projeto
9   Dado que eu acesse a aplicação
10  E informar o nome do projeto "Stefanini"
11  Quando eu clicar em Pesquisar
12  Então a aplicação deverá apresentar a lista de projetos em ordem crescente
```

```
1 package spring.h2.poc.step;
2
3 import cucumber.api.Scenario;
4
5 public class ConsultaProjetoStep {
6
7     @Before
8     public void before(Scenario scenario) {
9         //
10    }
11
12    @Dado("^que eu acesse a aplicação$")
13    public void que_eu_acesse_a_aplicação() throws Throwable {
14        //
15    }
16
17    @Dado("^informar o nome do projeto \"([^\"]*)\"$")
18    public void informar_o_nome_do_projeto(String arg1) throws Throwable {
19        //
20    }
21
22    @Quando("^eu clicar em Pesquisar$")
23    public void eu_clicar_em_Pesquisar() throws Throwable {
24        //
25    }
26
27    @Então("^a aplicação deverá apresentar a lista de projetos em ordem crescente$")
28    public void a_aplicação_deverá_apresentar_a_lista_de_projetos_em_ordem_crescente() throws Throwable {
29        //
30    }
31
32 }
```

Figura 02 - Implementação do teste de aceitação com BDD.

No final é gerado um relatório que é o teste de aceitação automatizado.



The image shows a Cucumber report for the 'Consulta Projeto' feature. It includes the BDD definitions and a table of the expected results.

```
#language: pt
@projeto Funcionalidade: Consultar Projeto
COMO Usuário DESEJO consultar todos os projetos cadastrados.
Cenário: Consultar Projeto
  Dado que eu acesse a aplicação
  E informar o nome do projeto "Stefanini"
  Quando eu clicar em Pesquisar
  Então a aplicação deverá apresentar a lista de projetos em ordem crescente
```

Projeto	Nome	Descrição	Editar	Detalhar	Excluir
<input type="checkbox"/>	Projeto 01	Descrição do projeto Stefanini 01			
<input type="checkbox"/>	Projeto 02	Descrição do projeto Stefanini 02			
<input type="checkbox"/>	Projeto 03	Descrição do projeto Stefanini 03			
<input type="checkbox"/>	Projeto 04	Descrição do projeto Stefanini 04			
<input type="checkbox"/>	Projeto 05	Descrição do projeto Stefanini 05			

1 2 3 4 5

Figura 03 - Relatório gerado pelo Cucumber da funcionalidade Consulta Projeto.

## TDD

Prosseguindo na etapa de desenvolvimento da funcionalidade podemos fazer uso de outras técnicas, **antes de implementar a funcionalidade** podemos **escrever testes para guiar o nosso desenvolvimento** os TDD's - Test Driven Development.

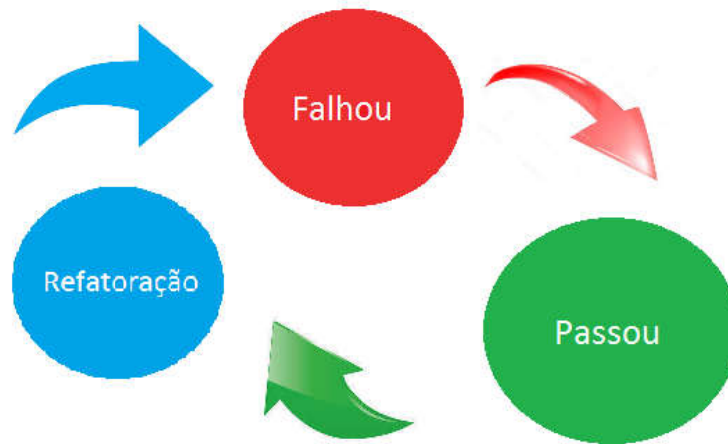


Figura 04 - Ciclo de desenvolvimento guiado por testes

TDD é baseado em **pequenos ciclos de repetição**, onde para cada funcionalidade do sistema um **teste é criado antes**, inicialmente **falhando**, por não existir a implementação, posteriormente criamos a funcionalidade e o teste passa e efetuamos um melhoramento do código com uma **refatoração**.

## Testes Unitários

Os **Testes Unitários** testa uma **única unidade do sistema**, na orientação a objetos, uma classe por exemplo, de maneira isolada e automatizada.

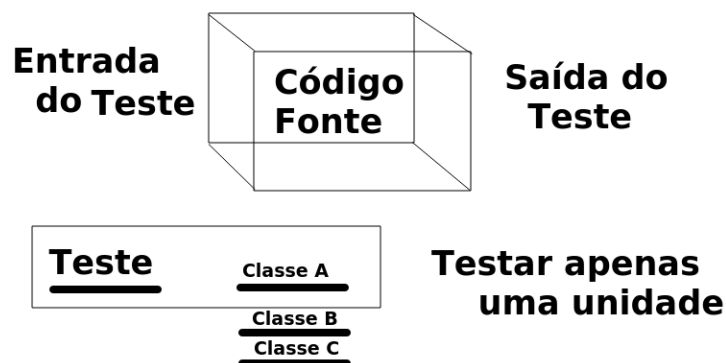


Figura 05 - Estrutura de testes de unidade.

## Testes de Integração

Os **Testes de Integração** testam a **integração entre dois ou mais módulos do sistema**, na orientação a objetos, uma classe de serviço que efetua uma consulta na base de dados, por exemplo, de maneira completa e automatizada.

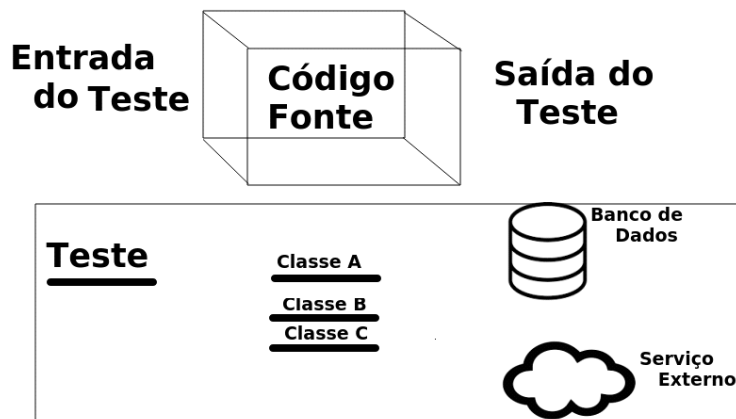


Figura 06 - Estrutura de testes automatizados

## Testes Exploratórios

Sim, mesmo com BDD, TDD, testes unitários e de testes de integração são necessários os testes exploratórios. Para você que ainda não sabe o que é um **Teste Exploratório**, é um **estilo de teste** no qual **você explora o software** enquanto, **simultaneamente**, **escreve e executa os testes usando o feedback do último teste para executar o próximo**.

## Teste de Performance

São os testes em que **submetemos o sistema a uma avaliação de carga, stress ou desempenho, avaliando os resultados para comparar com as especificações iniciais**.

Podemos citar como ferramenta o famoso JMeter, observe o relatório gerado pela aplicação.

A captura de tela mostra a interface do JMeter com o relatório 'Aggregate Report' aberto. O relatório contém uma tabela com os seguintes dados:

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	kB/s
HTTP Re...	10	56	16	407	15	407	0.00%	12.3/sec	4
TOTAL	10	56	16	407	15	407	0.00%	12.3/sec	4

Figura 07 - Relatório JMeter

## Como é possível simular o ambiente de produção no ambiente de testes?

Inicialmente era muito difícil **simular o ambiente de produção** em nossos testes, não eram as mesmas configurações de hardware, bibliotecas, as configurações em geral eram no máximo próximos, mas dificilmente idênticas, entretanto, com o **advento da virtualização** foi possível ter um **ambiente similar ao outro**. É possível clonar a máquina virtual e temos as mesmas configurações, bibliotecas e até mesmo hardware.

Além das soluções de mercado amplamente conhecidas de virtualização: Oracle Virtualbox e VMware. Atualmente é muito mais interessante utilizamos o **Docker** uma tecnologia que nós oferece Containers, sendo **uma camada de abstração entre o sistema operacional e nossas aplicações**, permitindo uma **portabilidade** mais independente que as soluções supra citadas.

Observe como o Docker fica entre o sistema operacional e nossas aplicações.

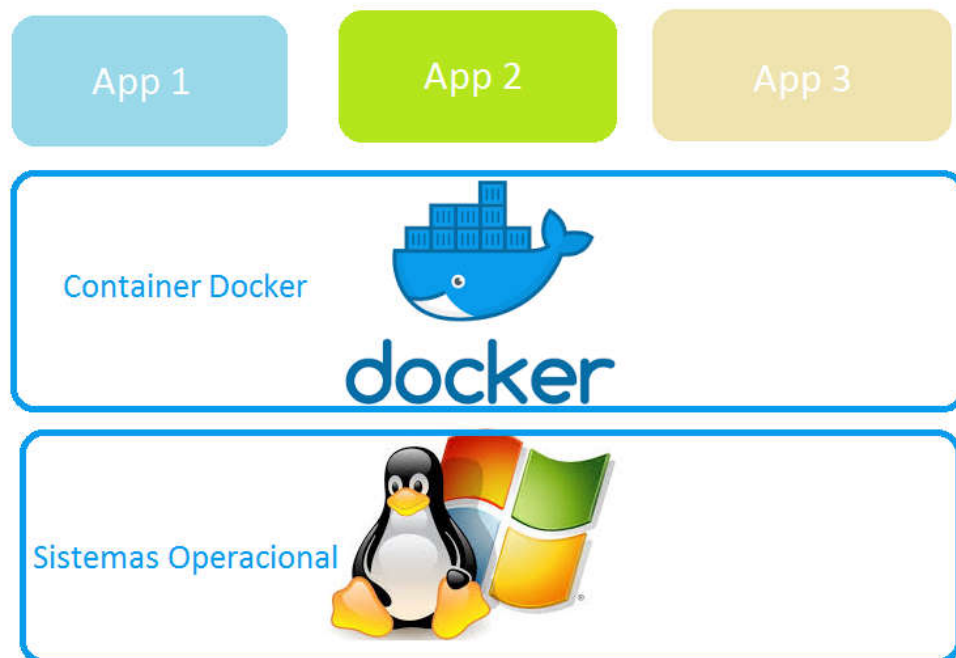
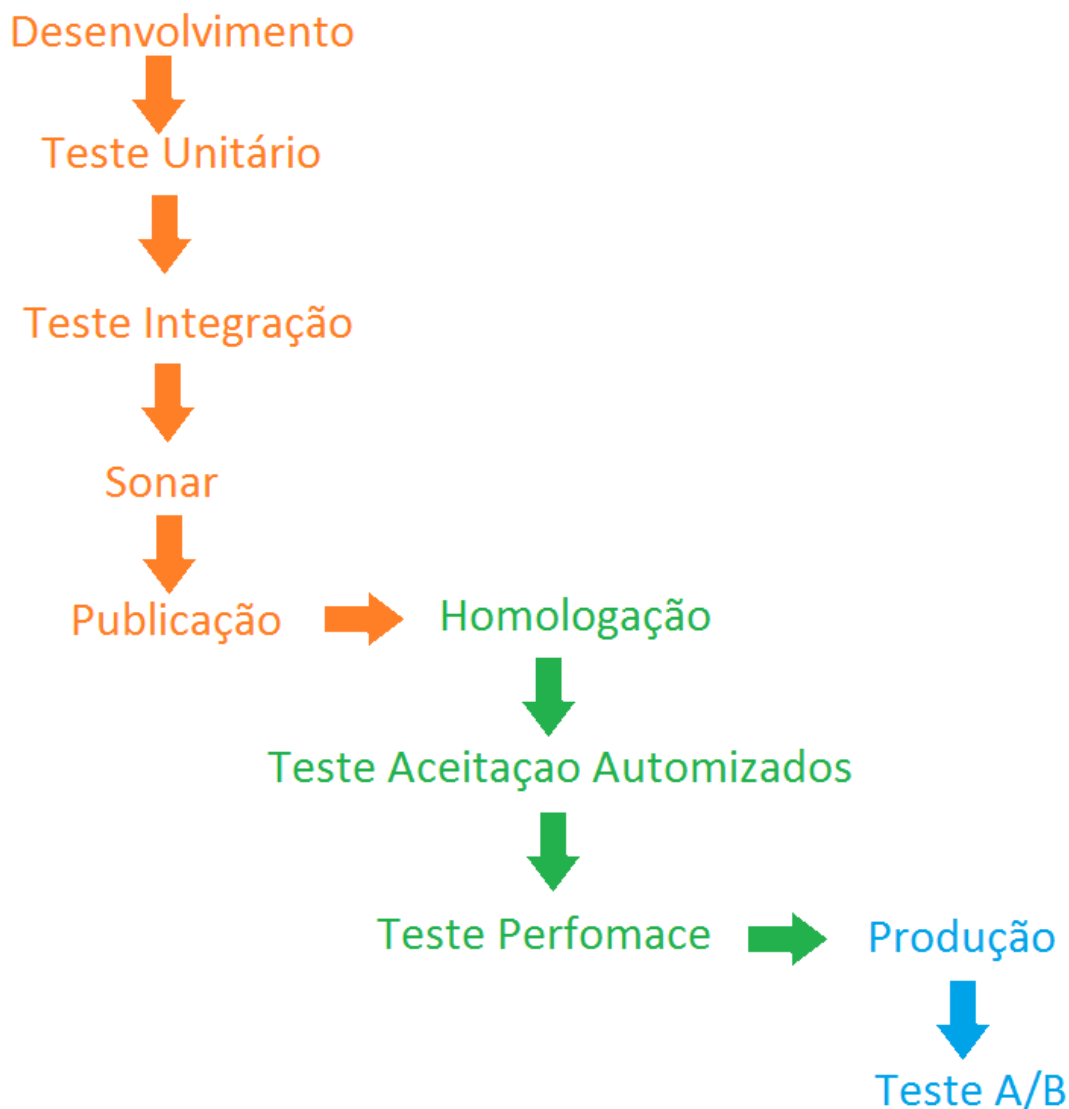


Figura 08 - Estrutura Container Docker.

Bom, agora vamos entender **como automatizar o processo de entrega com Pipeline e como o Jenkins pode auxiliar na automação de testes?**

Antes mesmo da **cultura DevOps** já trabalhávamos com **entrega contínua e execução dos diferentes tipos de teste** pelo **Jenkins**, todavia, precisamos explorar ao máximo esses recursos, primeiro vamos abordar a questão de **Pipeline que são processos automatizados para cada etapa da nossa implantação e entrega automatizada** em diferentes ambientes (desenvolvimento, teste, homologação e produção, por exemplo).

Uma aplicabilidade de Pipeline seria no ambiente de desenvolvimento queremos que sejam executados a build do projeto, a execução dos testes unitários e de integração, a avaliação da qualidade do código com o Sonar e a publicação no ambiente de desenvolvimento. Já no ambiente de homologação não vamos nos atentar aos testes unitários e de integração, mas sim, queremos executar os testes de aceitação automatizados e os testes de performance, e no ambiente de produção queremos que sejam executados apenas os testes de canários (Testes A/B) que são testes de experimento das funcionalidades ou também conhecidos como testes de fumaça. Cada um desses processos supra citados podem ser agrupado em uma Pipeline para cada ambiente, invés de serem executados separados e controlados manualmente. Garantindo que a entrega e implementação no ambiente só se daria se todas as suas pré-condições fossem satisfeitas.



**Figura 09 - Exemplo de Pipeline - Desenvolvimento, Homologação e Produção.**

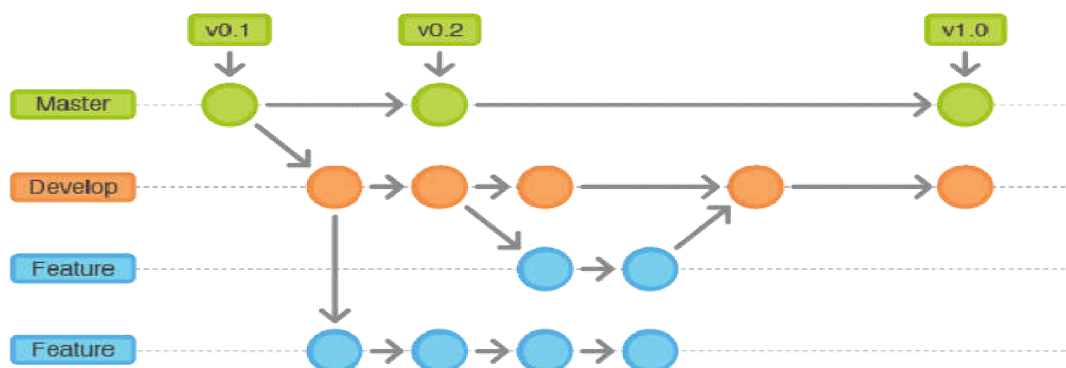
O Jenkins também possui inúmeros plugins para auxiliar na qualidade, separamos alguns relevantes: **JunitPlugin**(Tendência de Resultados de Testes), **Test Results Analyser**(Consolida e apresenta o resultado de todos os testes), **Cucumber Living Documentation**(Gera documentação completa em PDF e HTML baseado nas User Stories do Cucumber).

### Por fim vamos falar um pouco de **Git Flow** - Como utilizar o Git Flow na Garantia da Qualidade?

Como amamos definições, Git Flow é um modelo de organização de branch, estabelece regras de nomenclatura para cada tipo de branch.

Resumidamente temos:

- **Master:** contém o código de produção;
- **Develop:** contém o código para o próximo deploy em produção e serve de base para novas funcionalidades, sempre é atualizado por uma feature;
- **Feature:** são as branches(ramos) onde novos recursos são desenvolvidos;
- **Hotfix:** são utilizadas para corrigir o código de produção, ao ser finalizada sua função é atualizar tanto a branch master como a branch develop;
- **Release:** são marcos extraídos da develop antes da incorporação com a branch master.



**Figura 10 - Repositório Git organizado com Git Flow**

**Git Flow** permite que uma **funcionalidade seja testada isoladamente** na sua feature pela equipe de qualidade, sem afetar e ser afetada pelos demais testes nas demais branches.

Outra ponto interessante é que é possível testar uma correção apontando o código para uma hotfix.

Grande ganho de se utilizar o Git Flow na garantia da qualidade é que podemos efetuar a liberação de partes menores e ter um maior controle das versões e das correções.