Metodologia de Desenvolvimento de Software
da Defensoria Pública de Minas Gerais

Documentação dos projetos de testes automatizados

1 – Introdução

1.1 – Visão Geral

Este documento possui como finalidade definir conceitos a serem aplicados no processo de desenvolvimento de testes realizados nos projetos da instituição, além de padronizar as nomenclaturas empregadas nos projetos. Para tanto, iremos reunir informações necessárias para concluir qualquer demanda relativa ao tema.

1.2 – Escopo

Este documento contempla informações de tecnologias de desenvolvimento de testes, além das tecnologias de implantação e configuração de novos projetos. É também escopo deste documento, orientar os colaboradores sobre os padrões de desenvolvimento definidos pela DPMG além das suas respectivas definições de projetos. Serão descritas as camadas obrigatórias em todos os projetos de testes automatizados, requisitos mínimos para qualidade dos testes, e mecanismos essenciais para proporcionar o máximo de reuso de código, reduzindo substancialmente os custos de manutenção.

2 – Representação da Arquitetura

2.1-Selenium java Existem diversas ferramentas que interagem com aplicações frontend que permitem automatizar os testes desejados. Visando uma alta escalabilidade para funcionamento integrado dos testes, escolheu-se o Selenium em sua versão para Java, pois une a potência do selenium webdriver com a robustez da stack do JAVA. A comunicação dos testes com as aplicações será feita por uma gama de navegadores, que serão posteriormente discutidos, que irão navegar pelos ambientes de desenvolvimento, homologação e produção para garantir o correto funcionamento das aplicações em todo processo produtivo. Um grande diferencial do Selenium JAVA é a possibilidade de integração com o jenkins via maven, além de executar os testes em paralelo, reduzindo consideravelmente o tempo de execução.

2.2-jUnit

O jUnit é o framework escolhido para facilitar a criação e manutenção do código, permitindo as notações de @Test em cada unidade de teste, que posteriormente é utilizada para ser executada nas suites, também definidas pelo JUnit. Com o JUnit, pode-se componentizar os teste, isolá-los e, assim, padronizá-los com entradas parametrizadas. Tal implementação permite um processo de desenvolvimento contínuo, simultâneo entre vários funcionários e de extrema agilidade.

2.3-Apache Maven

O Maven é uma ferramenta de automação de compilação que constrói e gerencia projetos, principalmente projetos Java. Para os projetos de testes, o maven, por meio do arquivo POM.xml, irá organizar as versões do selenium, jUnit, bem como as classes que serão executadas e os plugins necessários para tanto. O Maven irá, antes de executar o projeto, baixar todos os artefatos necessários para o correto funcionamento da aplicação e salvá-los em cache, facilitando a manutenção das configurações.

2.4-Jenkins

O Jenkins é um servidor de automação de código robusto que auxilia na construção e execução dos testes. Por meio dele, os testes serão executados sempre que ocorrer uma alteração nas aplicações, e exibirá o resultado nos displays posicionados nos escritórios da dpMG. As configurações do jenkins devem ser feitas pelo arquivo jenkinsfile, seguindo também os padrões de projeto.

3-Descrição do projeto

3.1-DSLs

O Selenium-java é robusto e muito eficiente. Entretanto, ele possui um problema considerável quando trabalhamos com projetos complexos: a verbosidade. Como os métodos que interagem com as páginas da web são extremamente extensos, utilizamos o DSL para transformá-los em métodos mais intuitivos e simplificados. No DSL, deve-se manter os métodos mais utilizados no dia a dia, como clicar em botões e preencher caixas de texto, preferencialmente recebendo um By genérico.

3.2-Page Objects

Todos os testes serão feitos em páginas web. Sendo assim, deve-se criar os Page Objects que irão conter, para cada página a ser testada, absolutamente todos os objetos que podemos interagir, como botões, text fields, combo boxes e grids. Dessa forma, podendo acessar e interagir com todas as funcionalidades da página, os testes ficarão extremamente descritivos e pouco verbosos. É válido ressaltar que os Page Objects devem utilizar os métodos definidos no DSL para facilitar a manutenção do código.

3.3-Testes

Com os Page Objects estruturados, a construção dos testes é relativamente simples. Estruturando uma sequência de instruções, é preciso apenas criar as assertivas para garantir que o resultado da instrução foi esperado. É recomendável que para testes que necessitam de alterações em entradas, por exemplo, utilizar a função de parametrização do jUnit, possibilitando a reciclagem da estrutura do teste e reduzindo a repetição de código. Por fim, para garantir que um teste não irá influenciar os outros, é obrigatório extender a classe BaseTest para garantir que o próximo teste crie uma nova instância do navegador.

3.4-Suites

Os testes possuirão a notação de @Test, para indicar ao jUnit que eles serão, efetivamente, unidades de teste a serem executadas. Para tanto, é preciso criar classes Suites para organizar sequencialmente quais classes de teste serão executadas pelo maven.

3.5-WebDriver e suas funções

A ferramenta que permite a integração entre o navegador e o projeto é o webdriver. Ele é inicializado em todos os testes, e fechado ao fim dos mesmos, além de ser utilizado para interagir com os elementos web. Dessa forma, é interessante que exista uma classe específica para manipular essas funções, mantendo o padrão de manutenção do código. Logo, a classe DriverFactory possui a funcionalidade de iniciar e fornecer posteriormente o webdriver para os testes, além de finalizá-lo no BaseTest ao fim de cada execução.

4-Estruturas obrigatórias do projeto

4.1-Pacote Core

O pacote Core contém as configurações e propriedades que serão utilizadas em praticamente todas as classes. Nele, deve-se implementar, obrigatoriamente, uma classe contendo dados sensíveis, de acordo com a LDAP, uma classe de BasePage e BaseTest, um DriverFactory, um dicionário do Selenium(DSL) e uma classe de Propriedades.

- A classe de dados sensíveis deve conter getters estáticos para obter os dados a partir das variáveis de ambiente

```
ex:public static String getSenha() {return System.getenv("PASSWORD");
```

- A classe BasePage deve instanciar o DSL.
- A classe BaseTest deve conter um método que fecha o browser ao finalizar um teste.
- A classe DriverFactory deve conter os métodos estáticos para iniciar e finalizar o browser.
- O DSL deve conter os métodos mais utilizados pelo selenium, permitindo uma leitura mais limpa dos testes.
- A classe Propriedades deve conter urls contidas no escopo do projeto, o caminho para os drivers no pacote de recursos e a seleção de qual browser será executado.

4.2-Pacote do projeto

O pacote do projeto possuirá o conjunto de page objects, suites e testes que irão garantir a execução e funcionalidade correta das aplicações que serão testadas. Dentro do pacote do projeto, teremos 3 outros pacotes:

- pages, que conterá os page Objects. O pacote pages deve preferencialmente ser subdividido em outros pacotes, dependendo das funcionalidades do projeto.
- testes, que conterá todos os testes, preferencialmente subdivididos em funcionalidades, assim como as pages.
- suites, que irão definir a sequência de classes testes que serão executadas. As suites devem ser classes divididas de acordo com a funcionalidade, sem a necessidade da subdivisão em pacotes.

5-Criação de um projeto novo

5.1-Criando Projeto Maven

O primeiro passo para criar um novo projeto de testes automatizados é criá-lo como projeto maven. O projeto maven virá com um pacote raiz, o arquivo POM.xml e outros arquivos específicos do seu ambiente de desenvolvimento. É interessante que, após criar essa estrutura, atualizar o projeto base no git com esses arquivos.

5.2-Configurando o POM.xml

Algumas IDEs pedem para configurar partes padrões do POM durante a criação do projeto, outras utilizam um padrão. De qualquer forma, inicialmente o projeto possuirá esta estrutura:

pom.xml (teste)

```xml
<?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>org.example</groupId>
    <artifactId>NomeDoSeuProjeto</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

</project>
```

Inicialmente, é preciso alterar o para o padrão da defensoria.

```
java<groupId>defensoria.mg.def.br</groupId>
```

Em sequência, deve-se adicionar as dependências desejadas, sendo elas o Selenium-java e o jUnit.

```
<dependencies>
   <dependency>
       <groupId>org.seleniumhq.selenium</groupId>
       <artifactId>selenium-java</artifactId>
       <version>3.141.59</version>
   </dependency>
   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.12</version>
       <scope>compile</scope>
   </dependency>
</dependencies>
```

Por fim, é preciso definir as instruções de construção do projeto, podendo ser alterado de acordo com as necessidades e desafios encontrados.

```
<build>
   <defaultGoal>install</defaultGoal>
   <testSourceDirectory>src</testSourceDirectory>
   <plugins>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-surefire-plugin</artifactId>
           <version>2.22.0</version>
           <configuration>
               <includes>seuProjeto.suites/*.java</includes>
               <parallel>suites</parallel>
               <threadCount>4</threadCount>
           </configuration>
       </plugin>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <version>3.8.1</version>
           <configuration>
               <source>1.8</source>
               <target>1.8</target>
           </configuration>
       </plugin>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-surefire-report-plugin</artifactId>
           <version>3.0.0-M5</version>
```

```
        </plugin>
    </plugins>
</build>
```

Os plugins utilizados darão suporte ao jUnit e selenium, sendo o mais importante o maven-surefire-plugin. Em sua parte de configurações, veremos o seguinte trecho

```
<configuration>
    <includes>seuProjeto.suites/*.java</includes>
```

Deve apontar para o pacote que contém todas as suites do projeto.

```
<parallel>suites</parallel>
```

Define as notações do jUnit que irão ser executadas paralelamente.

```
<threadCount>4</threadCount>
```

Define a quantidade de threads do processador destinadas a executar os testes paralelamente (recomenda-se de 4 a 6 threads).

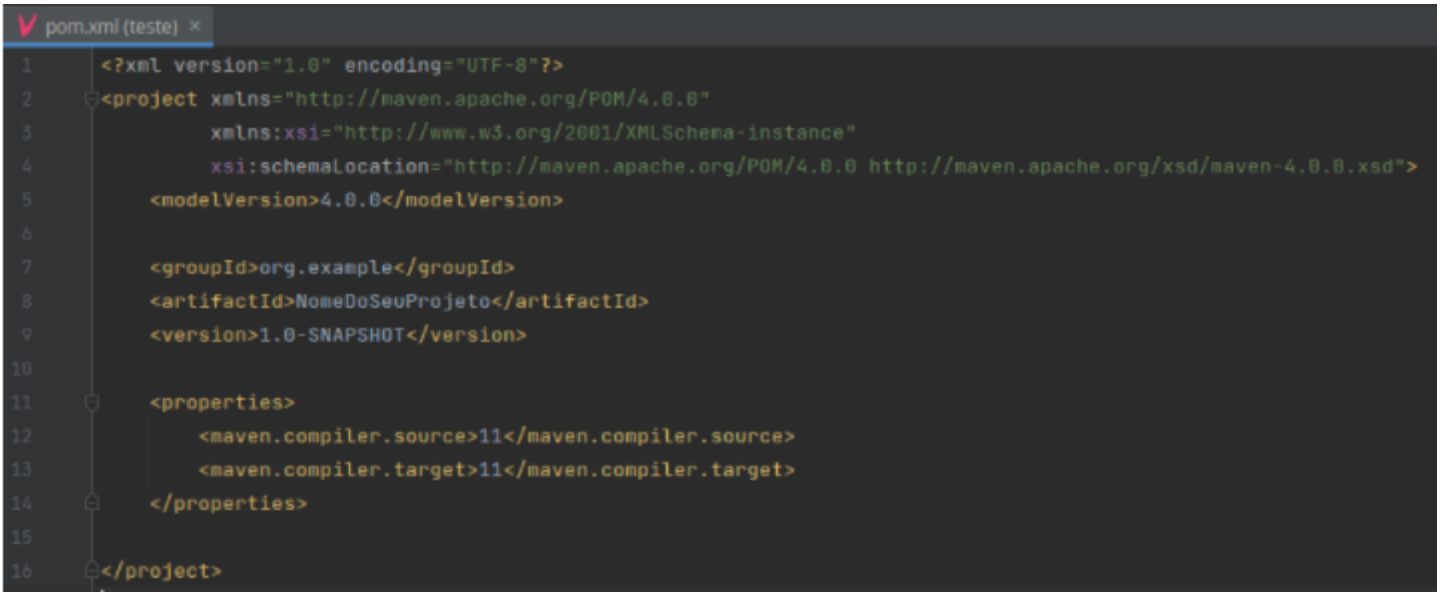
```
</configuration>
```

Essa parte permitirá a execução em paralelo dos testes.

5.3-Configurando o settings.xml

O maven possui um arquivo de configurações padrão, entretanto devemos utilizar as configurações no padrão da defensoria.

1. Clone ou baixe o projeto CONFIGURACOES-PROJETOS em alguma pasta da sua máquina.
2. Na sua IDE, abra as configurações e pesquise por “maven”. Sua tela deve ser semelhante a seguinte:



3. Sobrescreva o arquivo de configurações de usuário -no caso da imagem acima, clique em “override” do “User settings file”.
4. Busque pela pasta de configurações de projetos>maven e selecione settings.xml.
5. Aplique as alterações.

5.4-Configurando o jenkins

O jenkinsfile é o arquivo que explica para o jenkins como o projeto deve ser executado em pipeline, portanto é a conexão do projeto com a pipeline. Inicialmente, crie na camada mais externa do projeto um arquivo chamado “jenkinsfile”. O código a seguir é o mínimo para executar um projeto:

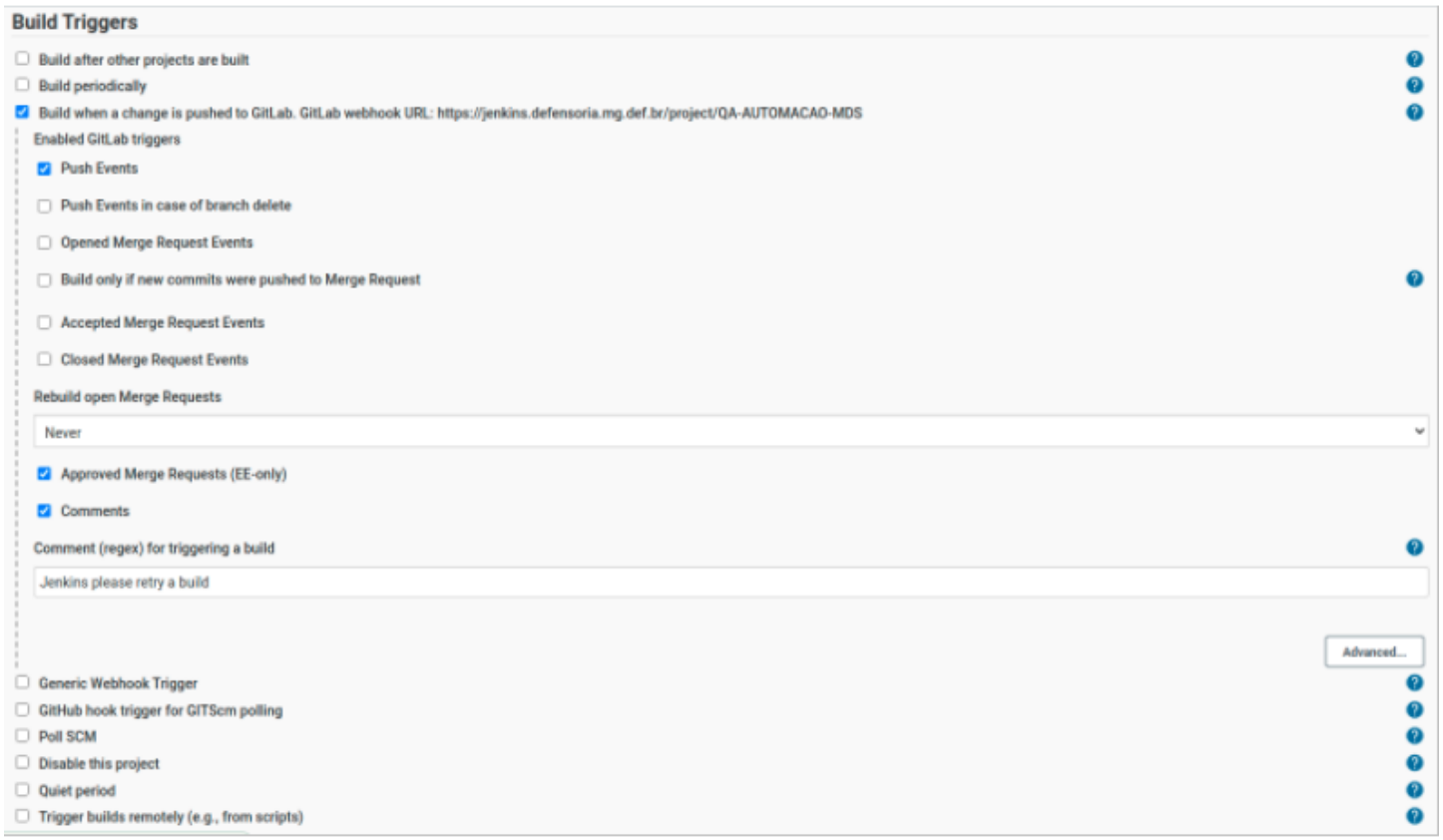
```
pipeline {
   agent any
   tools{
      jdk 'JAVA_11'
      maven 'MAVEN 3.6.3'

   stages{
       // este stage builda o projeto para poder rodar
       stage('Build'){
           steps {
               sh 'mvn install -DskipTests -Dselenide.browserBinary="chrome/usr
           }
       }
       stage ('Executar testes gerais'){
           steps {
               // este é o comando em ShellScript que roda os testes automatiza
               sh "mvn clean test"
           }
```

```
        }
    }
    // este passo roda ao final do build
    post {
        // em caso de sucesso, limpa o diretorio para manter o workspace limpo e
        success {
            deleteDir()
        }
    }
}
```

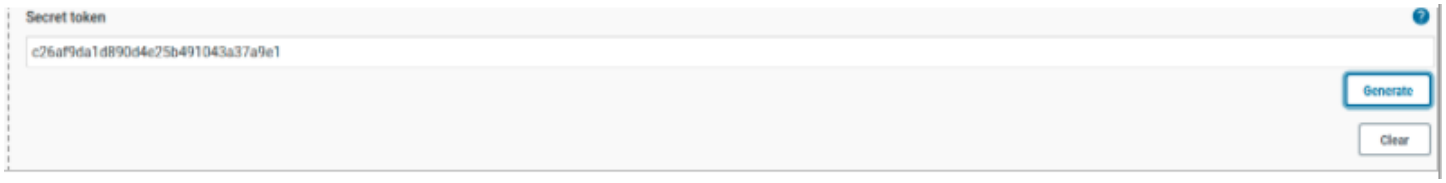
Dessa forma, precisamos configurar o projeto no jenkins e conectá-lo ao git.

1. Entre em http://jenkins.defensoria.mg.def.br/ e faça login.
2. Selecione no canto superior esquerdo a opção “New Item”
3. Em “Enter an item name”, coloque o nome do projeto seguindo o padrão da dpmg “QA-AUTOMACAO-”
4. Selecione “Pipeline” e clique em OK. As configurações são extensas, então siga os passos com atenção
   1-)Selecione “Discard old builds” e preencha da seguinte forma

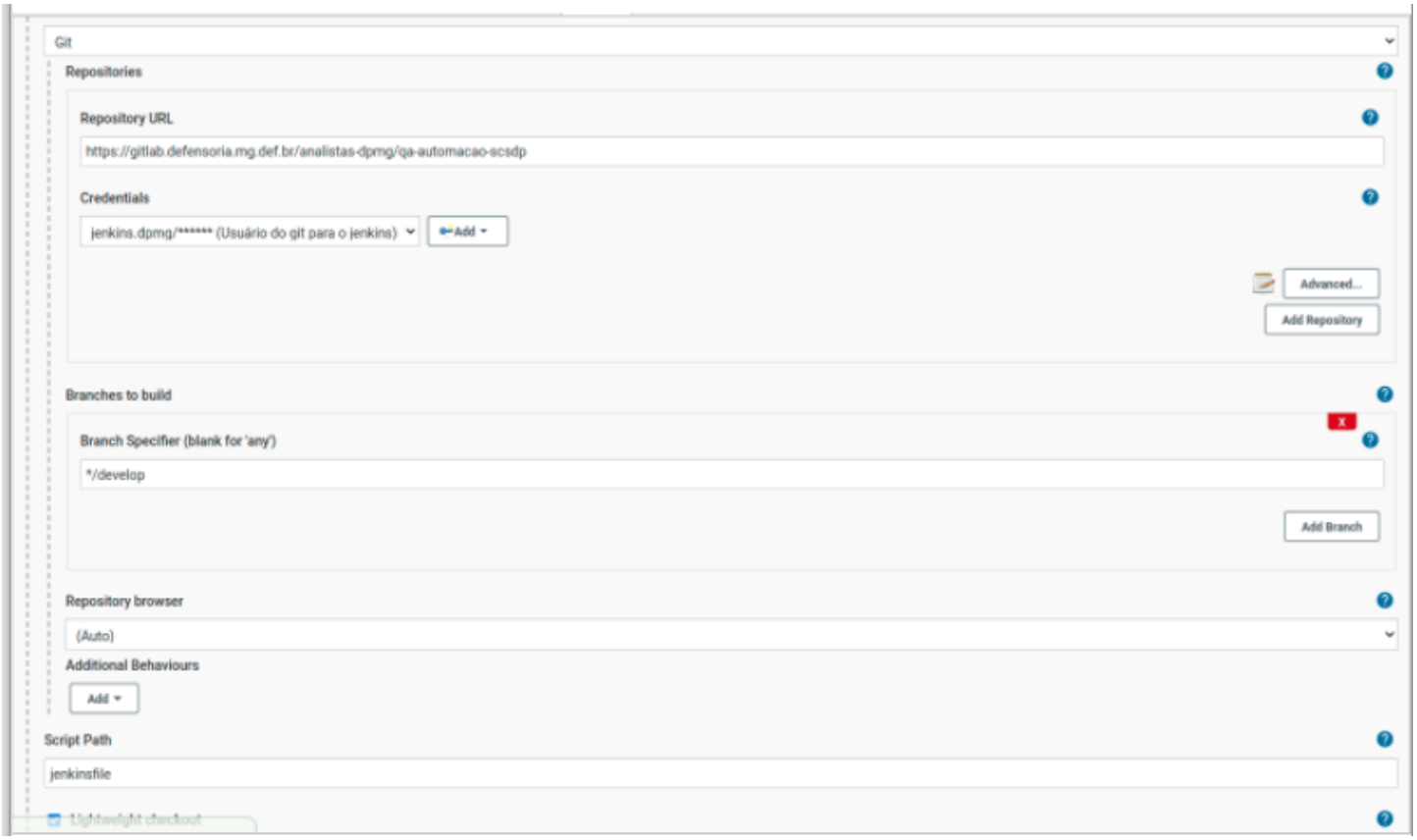
2-)Em “Build Triggers” selecione as seguintes opções:



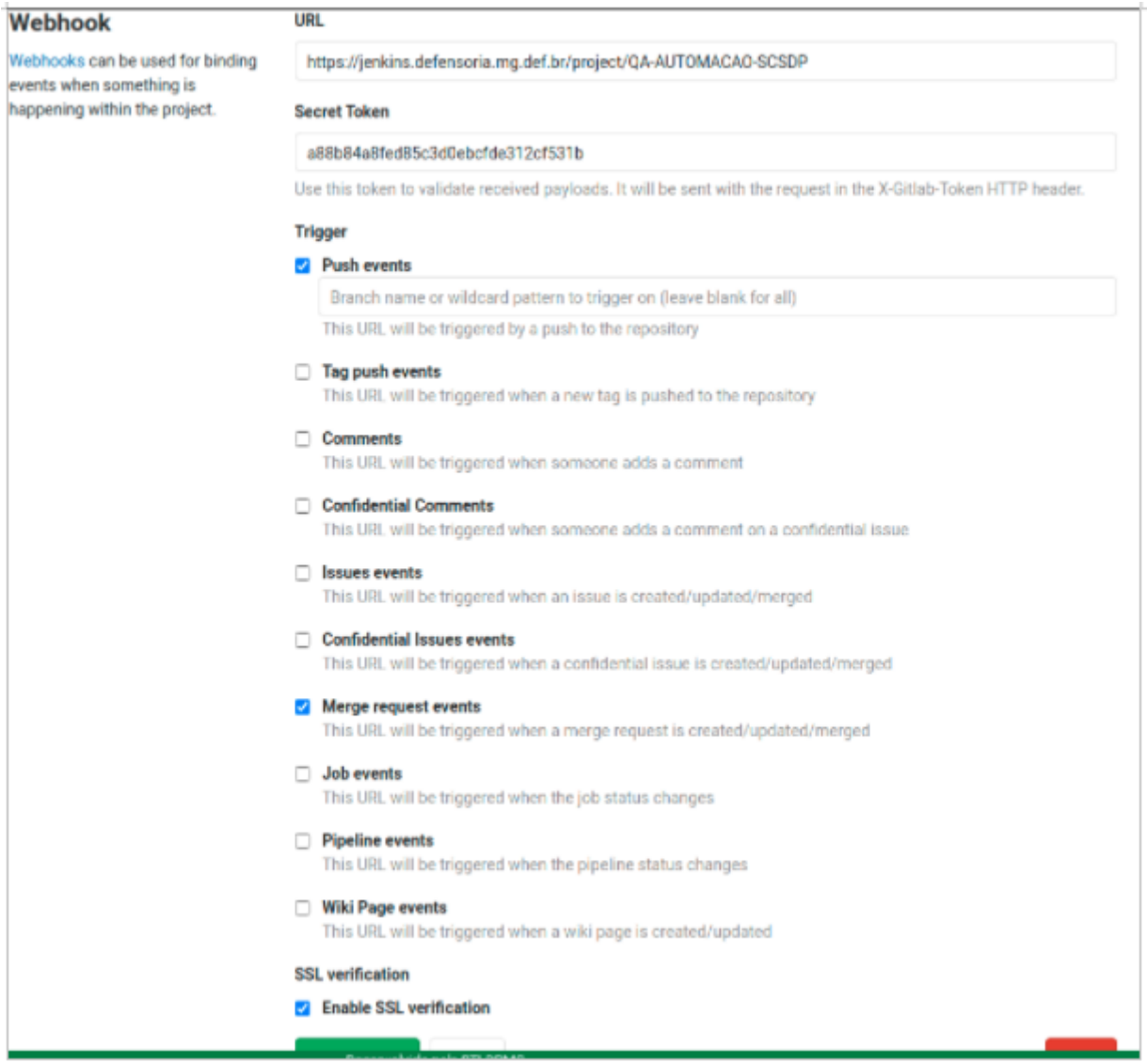
Antes de progredir, salve em algum local o Gitlab webhook URL, no caso acima http://jenkins.defensoria.mg.def.br/project/QA-AUTOMACAO-MDS Clique em “Advanced…” e gere um token secreto que também deve ser salvo/anotado



3-)Em “Pipeline, selecione “Pipeline script from SCM” e preencha o restante das opções de acordo com o seu projeto



4-)Clique em salvar. Agora devemos criar o webhook no jenkins. 1-)Entre no projeto pelo git 2-)Na parte esquerda da tela, clique em configurações>Webhooks (atenção que é preciso ter permissões necessárias para visualizar essas opções) 3-)Em URL, cole o link fornecido pelo jenkins “Gitlab webhook URL” 4-)Em Secret Token, cole o token secreto fornecido pelo jenkins 5-)Nos Triggers, selecione: Push events (não é necessário definir uma branch específica). Merge request events. 6-)Selecione “Enable SSL verification” e clique em Save changes.



Se tudo ocorreu como deveria, o projeto já está configurado na pipeline e deve executar os testes sempre que ocorra uma alteração no git. É importante adicionar o webhook acima, da mesma forma, em todos os projetos que podem afetar as funcionalidades contidas no escopo dos seus testes.

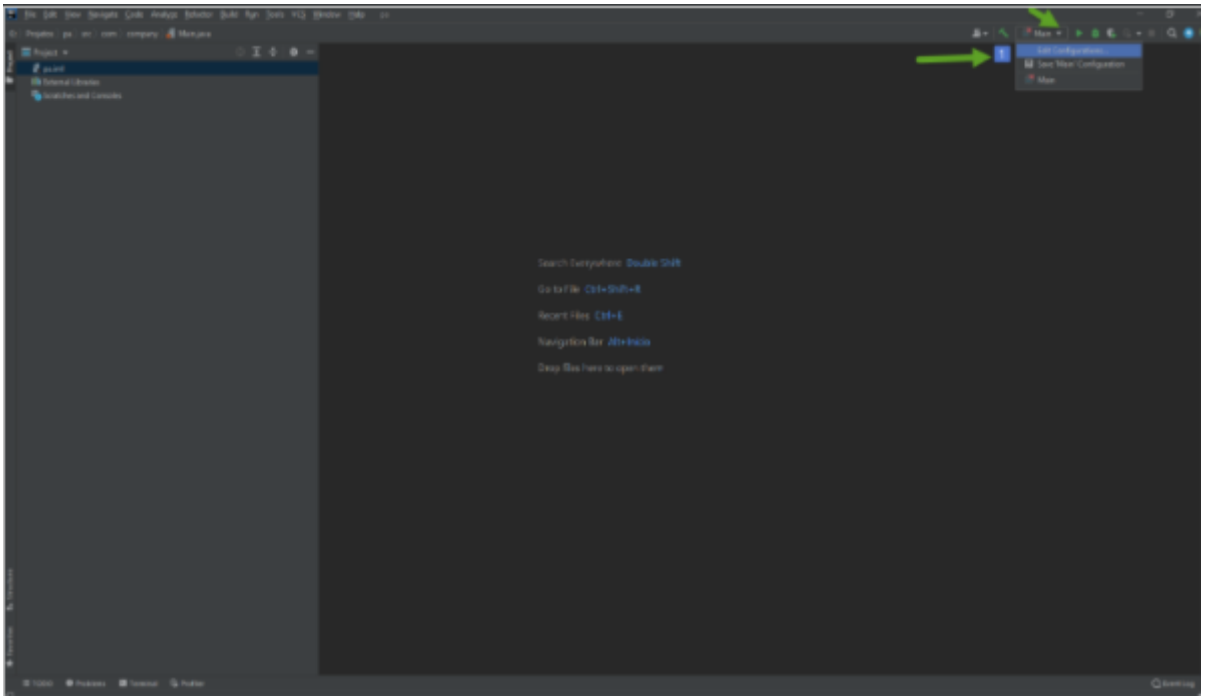
**5.5-Executando o projeto localmente## **

O projeto pode ser executado tanto pela IDE, quanto pela linha de código. Como cada IDE configura a execução do projeto de forma distinta, iremos ilustrar apenas por linha de código, sendo possível transcrever para a IDE. Como o projeto será executado com auxílio do maven, abriremos ele no terminal e executaremos o seguinte comando: mvn clean install -DskipTests Dessa forma, iremos instalar as dependências e limpar qualquer dado salvo de execuções anteriores. Em sequência, executaremos o seguinte comando: mvn test Assim, executaremos os testes indicados no arquivo POM.xml.

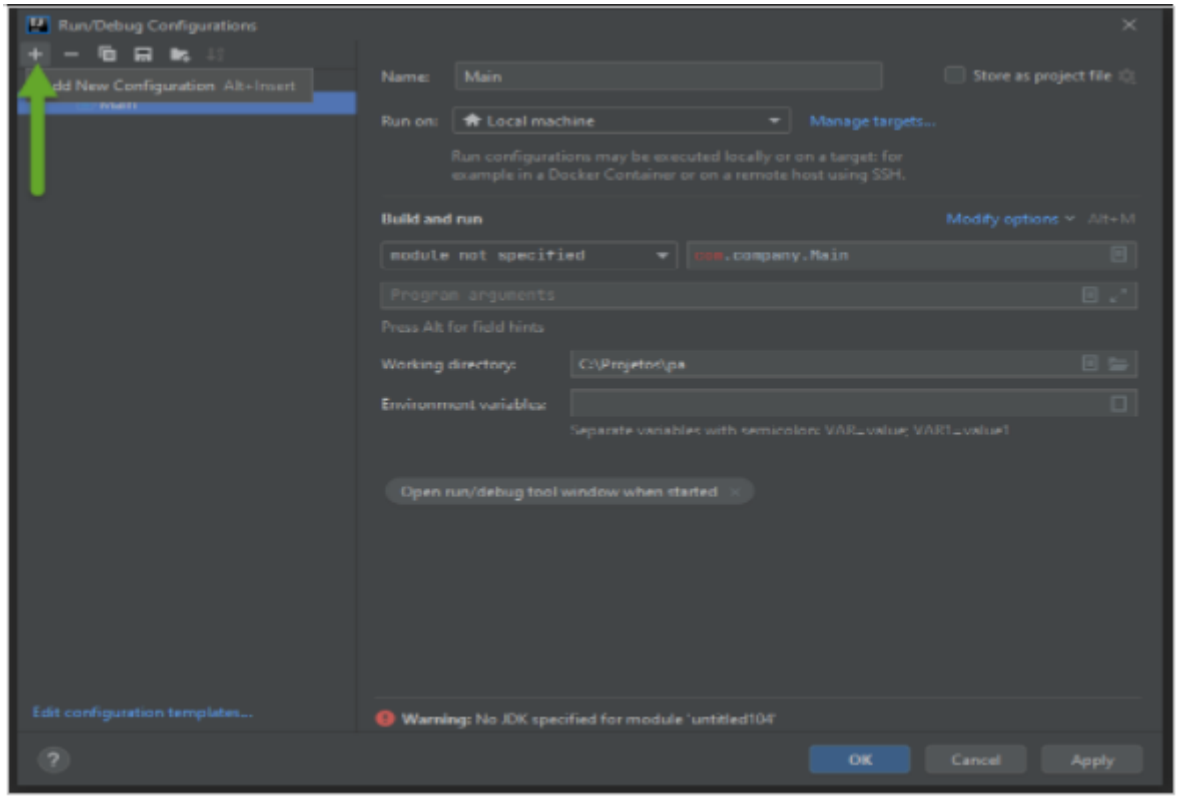
6-Exemplificando as configurações locais na IDE IntelliJ

6.1-Configurando o maven no projeto

- Clicar em Edit Configurations

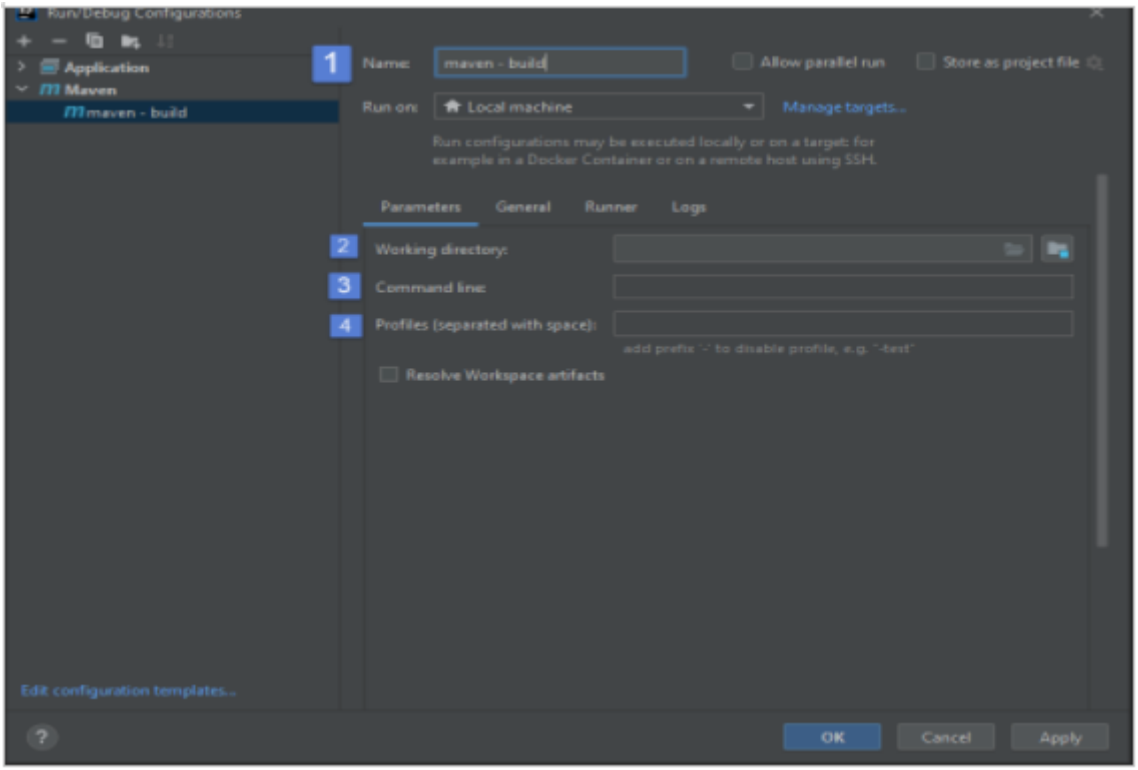


- Clicar no +
- Ao abrir procurar por maven e Clicar para abrir

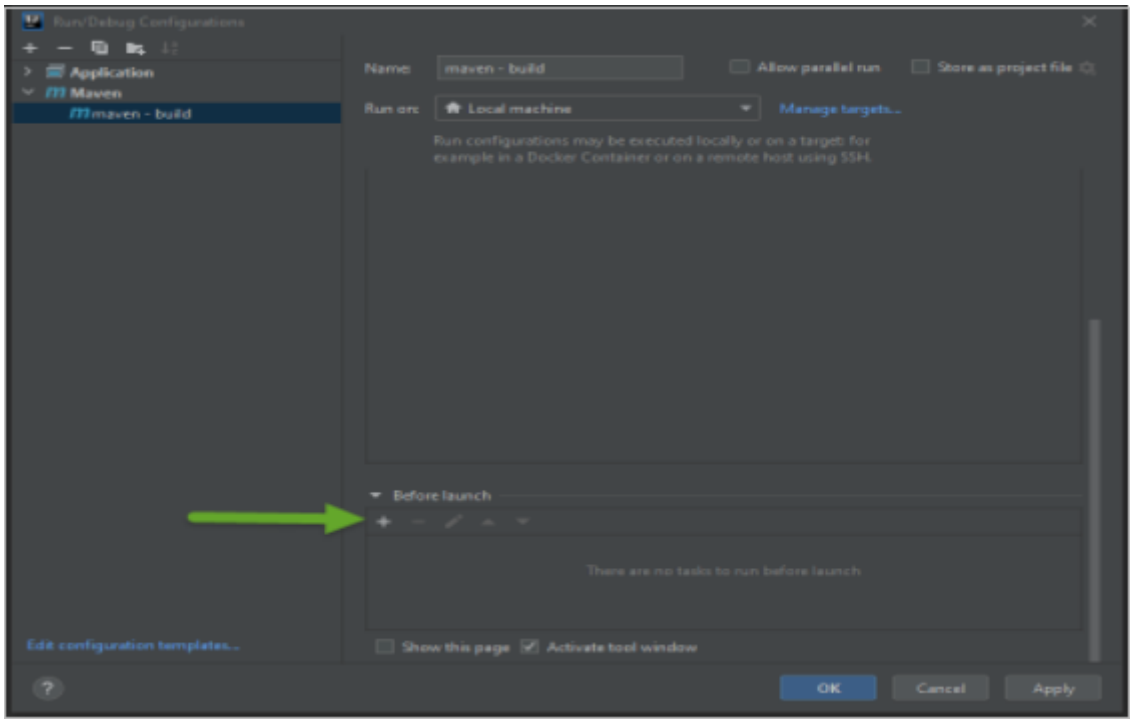


Em Parameters

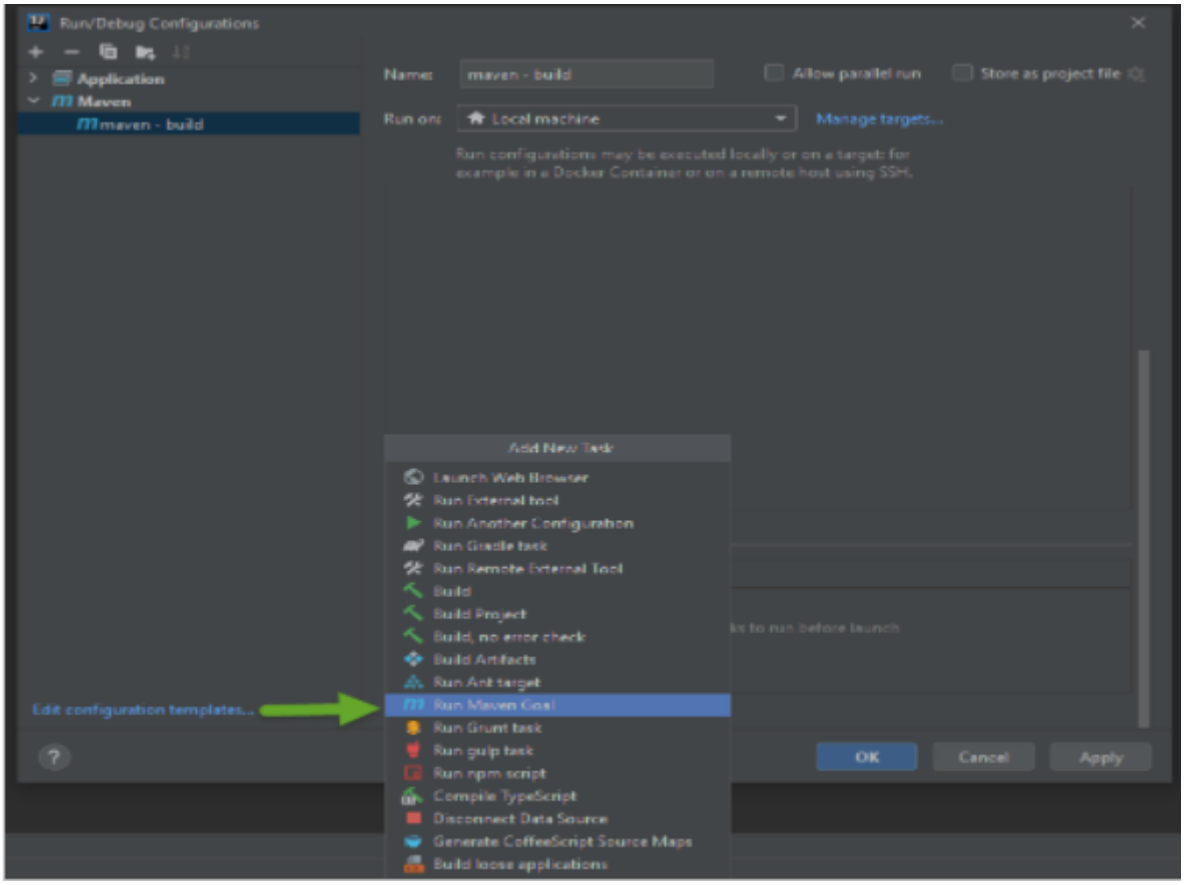
1. Preencher Name (maven – build)
2. Parameters – Working directory (clicar na pasta ao lado e escolher “pasta principal do projeto”)
3. command line (“test”)
4. Profiles “dev”



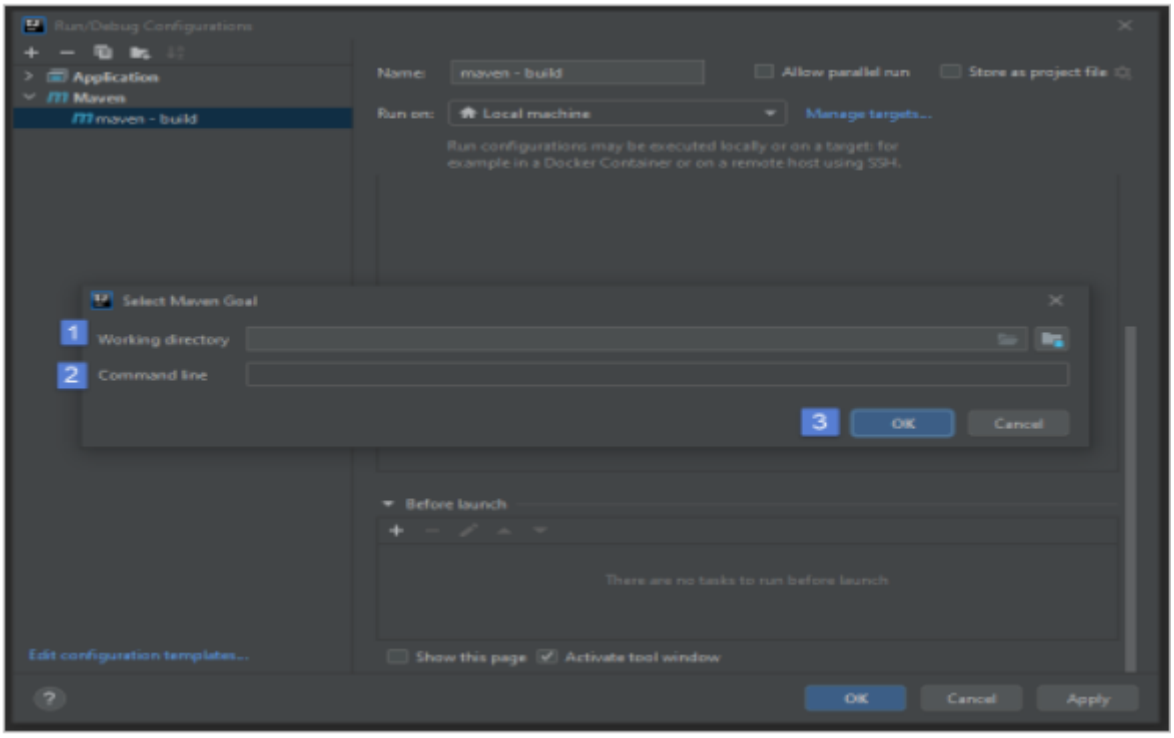
*1 – Em Before launch clicar no +



*2 – Selecionar Run Maven Goal

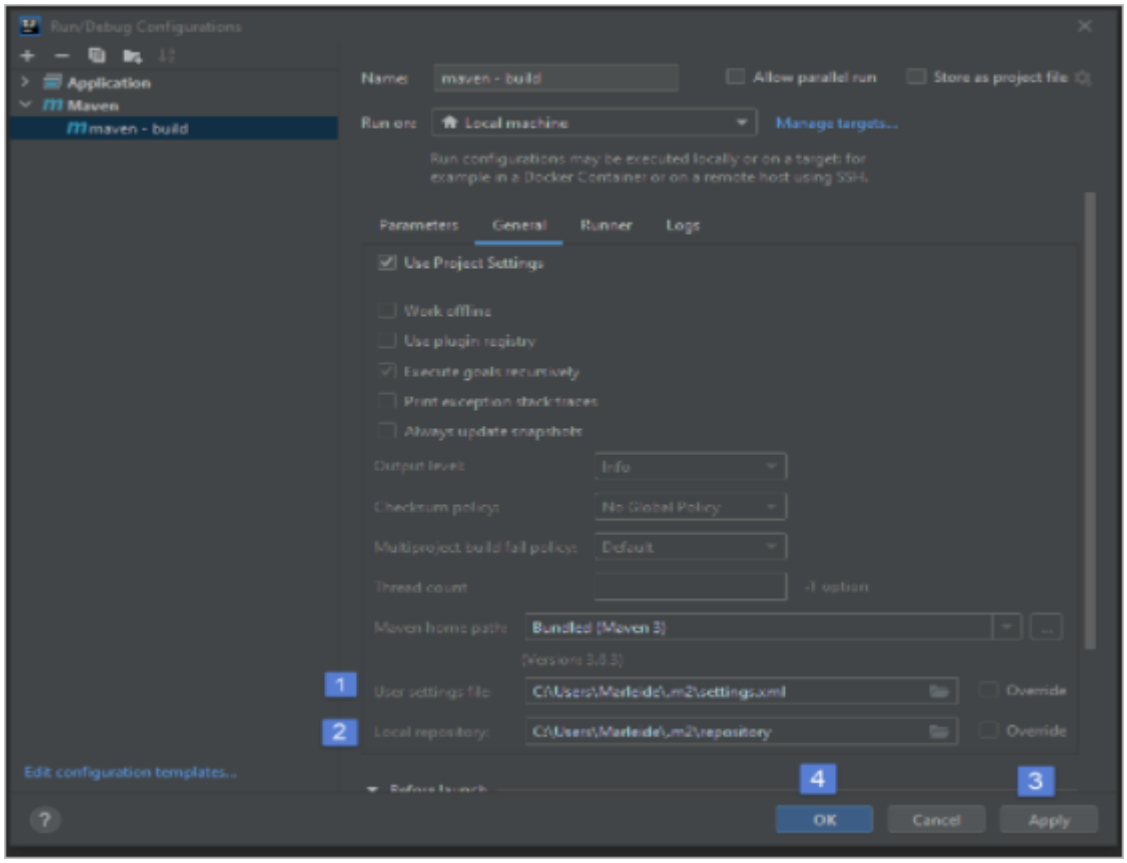


*1 – Working directory (escolher pasta do projeto) *2 – Command line (clean install -DSkipTests) *3 – ok



Em General

*1 User settings file (escolher pasta do projeto settings) *2 – Marcar a opção “Override”
*3 – Clicar Apply *4 – Clicar ok


Run/Debug Configurations
Application
Maven
maven - build
Name: maven - build
Allow parallel run
Store as project file
Run on: Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.
Parameters
General
Runner
Logs
Use Project Settings
Work offline
Use plugin registry
Execute goals recursively
Print exception stack traces
Always update snapshots
Output level: Info
Checksum policy: No Global Policy
Multiproject build fail policy: Default
Thread count:
-T option
Maven home path: Bundled (Maven 3)
(Version: 3.6.3)
1
User settings file: C:\Users\Marleide\.m2\settings.xml
Override
2
Local repository: C:\Users\Marleide\.m2\repository
Override
Edit configuration templates...
4
3
OK
Cancel
Apply