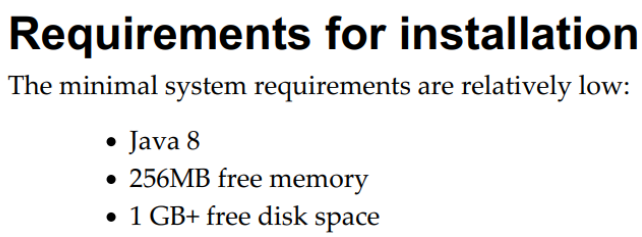
Devops

1. CI/CD with Jenkins and K8s

1.1 Jenkins installation



Install docker runtime (ubuntu)

- Uninstall old version

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

- Install using repo

```
sudo apt-get update


sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release


curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
```

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Install Jenkins:

```
sudo docker run -d -p 30000:8080 --name jenkins jenkins/jenkins:lts
```

Find password:

```
sudo docker exec -u root -it jenkins /bin/bash
```

Password in $JENKINS_HOME/secrets/initialAdminPassword file

```
cat $JENKINS_HOME/secrets/initialAdminPassword
```

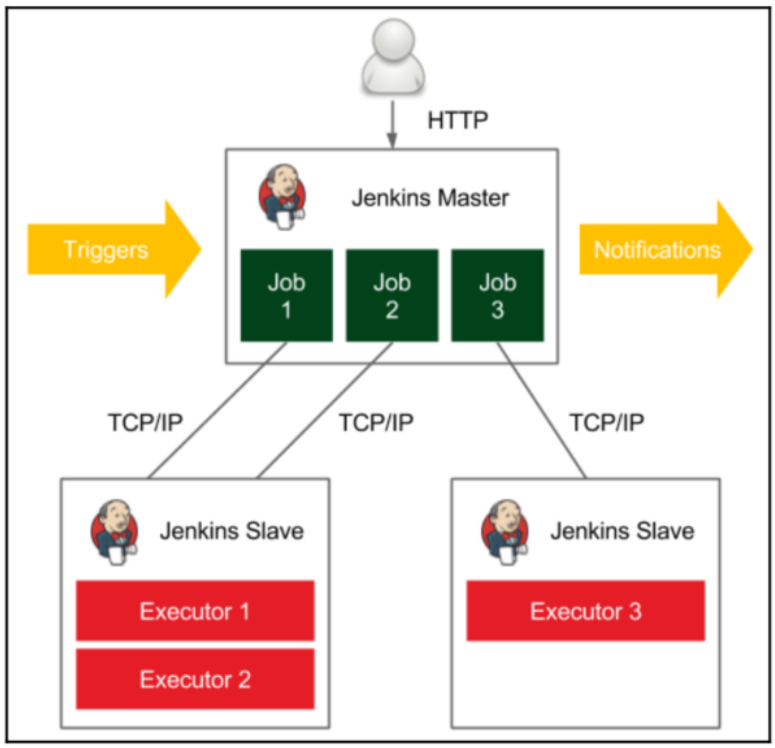
`4fb047675f09428bb72976fa510f026c`

Install VIM

```
apt-get update

apt-get install apt-file

apt-file update

apt-get install vim     # now finally this will work !!!
```

1.2 Jenkins architecture

Master and slaves



- **Master**: This is usually (unless the project is really small) a dedicated machine with RAM ranging from 200 MB for small projects to 70GB plus for huge single-master projects.
- **Slave**: There are no general requirements (other than the fact that it should be capable of executing a single build, for example, if the project is a huge monolith that requires 100 GB of RAM, then the slave machine needs to satisfy these needs).

Setting agents

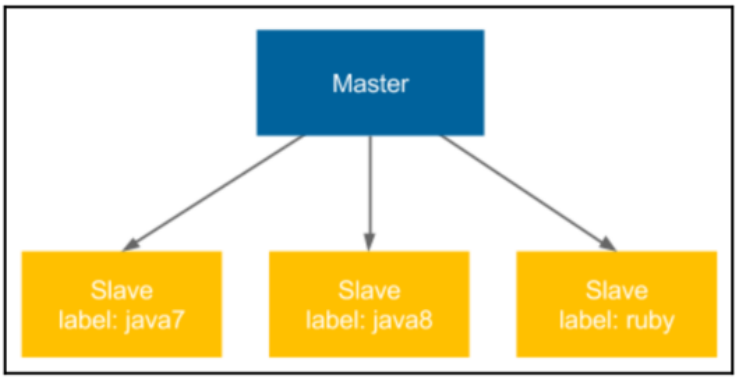
These differences resulted in four common strategies how agents are configured:

- Permanent agents
- Permanent Docker agents
- Jenkins Swarm agents
- Dynamically provisioned Docker agents

- Permanent agents

Understanding permanent agents

As already mentioned, the drawback of such a solution is that we need to maintain multiple slave types (labels) for different project types. Such a situation is presented in the following diagram:



- Permanent Docker agents

Configuring permanent Docker agents

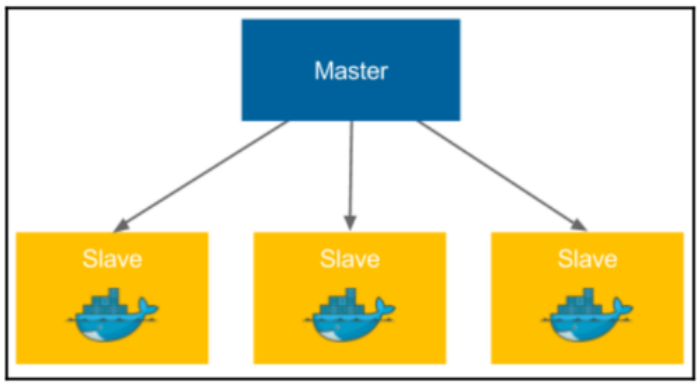
The configuration is static, so it's done exactly the same way as we did for the permanent agents. The only difference is that we need to install Docker on each machine that will be used as a slave. Then, we usually don't need labels because all slaves can be the same. After the slaves are configured, we define the Docker image in each pipeline script.

```
pipeline {
    agent {
        docker {
            image 'openjdk:8-jdk-alpine'
        }
    }
    ...
}
```

When the build is started, the Jenkins slave starts a container from the Docker image `openjdk:8-jdk-alpine` and then executes all pipeline steps inside that container. This way, we always know the execution environment and don't have to configure each slave separately depending on the particular project type.

Understanding permanent Docker agents

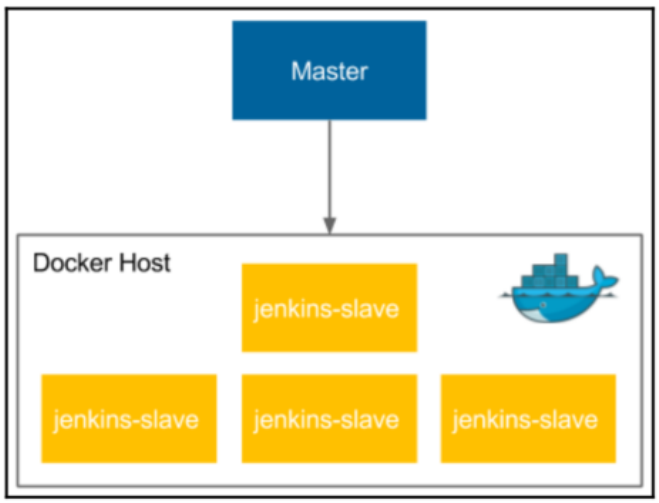
Looking at the same scenario we took for the permanent agents, the diagram looks like this:



Each slave is exactly the same and if we would like to build a project that depends on Java 8, then we define the appropriate Docker image in the pipeline script (instead of specifying the slave label).

- Jenkins Swarm agents
- Dynamically provisioned Docker agents

The following diagram presents the Docker master-slave architecture we've configured:

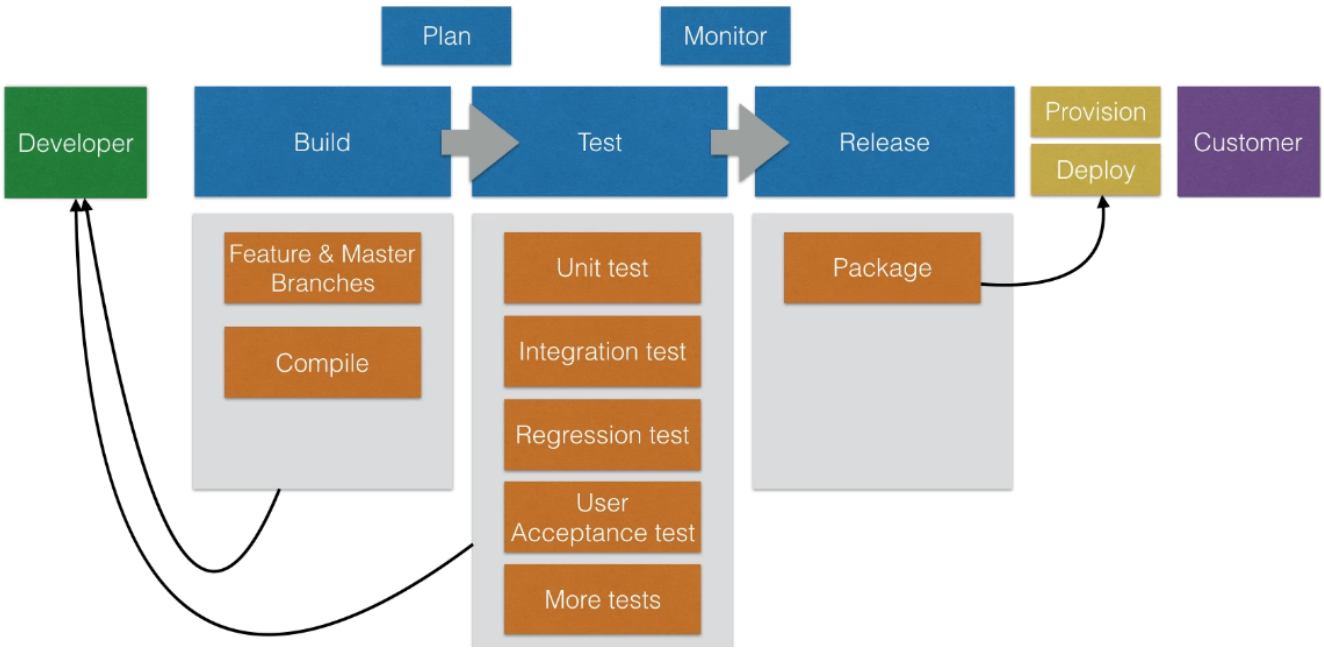


Let's describe step by step how the Docker agent mechanism is used:

1. When the Jenkins job is started, the master runs a new container from the `jenkins-slave` image on the slave Docker host.
2. The jenkins-slave container is, actually, the ubuntu image with the SSHD server installed.
3. The Jenkins master automatically adds the created agent to the agent list (same as what we did manually in the *Setting agents* section).
4. The agent is accessed using the SSH communication protocol to perform the build.
5. After the build, the master stops and removes the slave container.

Running Jenkins master as a Docker container is independent from running Jenkins agents as Docker containers. It's reasonable to do both, but any of them will work separately.

Jenkins summary:



Building docker image with jenkins:

1. Install Cloudbess docker

2. Set up docker jenkins machine

```
sudo yum install git -y
```

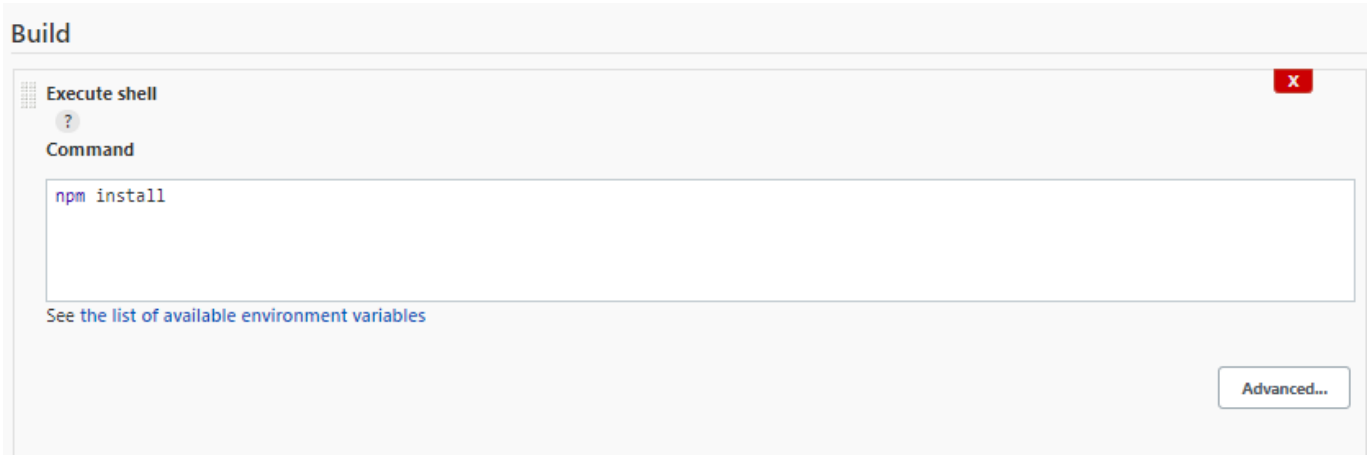
```
git clone https://github.com/duyng2512/jenkins-docker.git
docker build -t jenkins-docker .
```

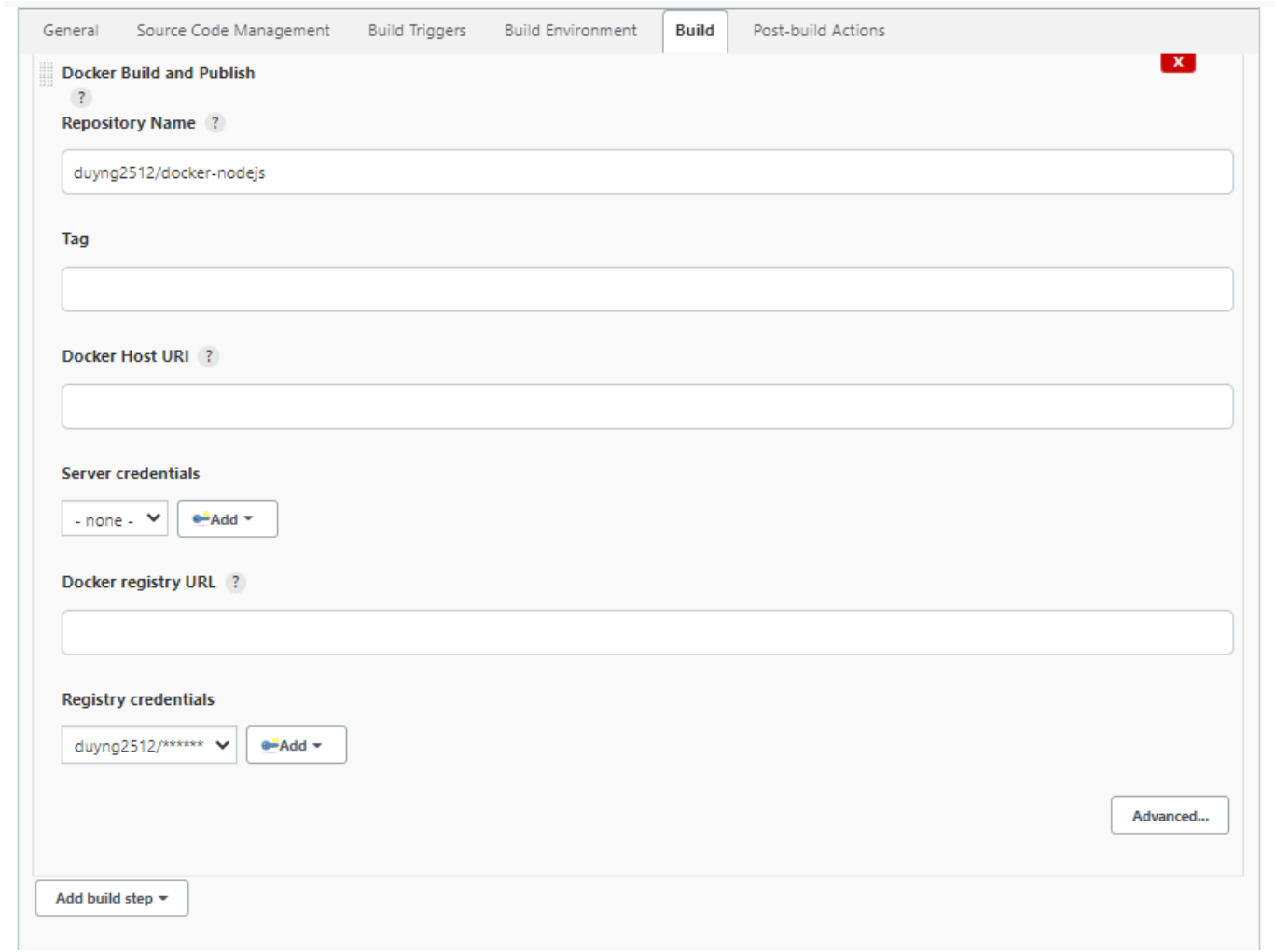
```
docker volume create jenkins_home
```

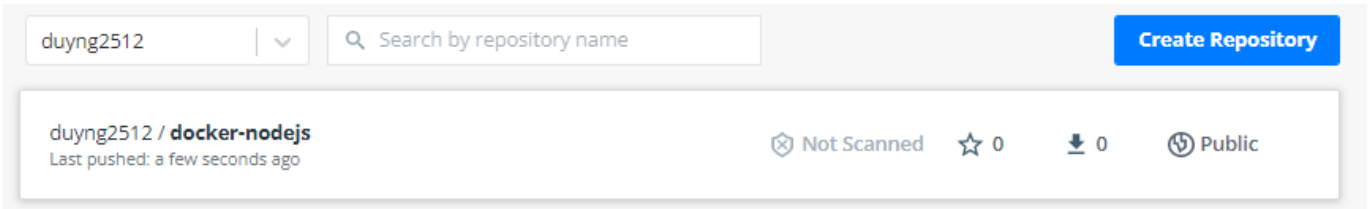
```
docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home -v
/var/run/docker.sock:/var/run/docker.sock --name jenkins-docker -d jenkins-docker
```

```
docker exec -u root -it jenkins-docker bash
chmod 777 /var/run/docker.sock
ls -ahl /var/run/docker.sock
```

3. Setup jobs







Infracstructure as code:

Jenkins jobs DSL

```
job('NodeJS example') {

    scm {
        git('git://github.com/wardviaene/docker-demo.git') {  node ->
            node / gitConfigName('DSL User')
            node / gitConfigEmail('jenkins-dsl@newtech.academy')
        }
    }

    triggers {
        scm('H/5 * * * *')
    }

    wrappers {
        nodejs('nodejs') // this is the name of the NodeJS installation in
                         // Manage Jenkins -> Configure Tools -> NodeJS Installations -> Name
    }

    steps {
        shell("npm install")
    }

}
```

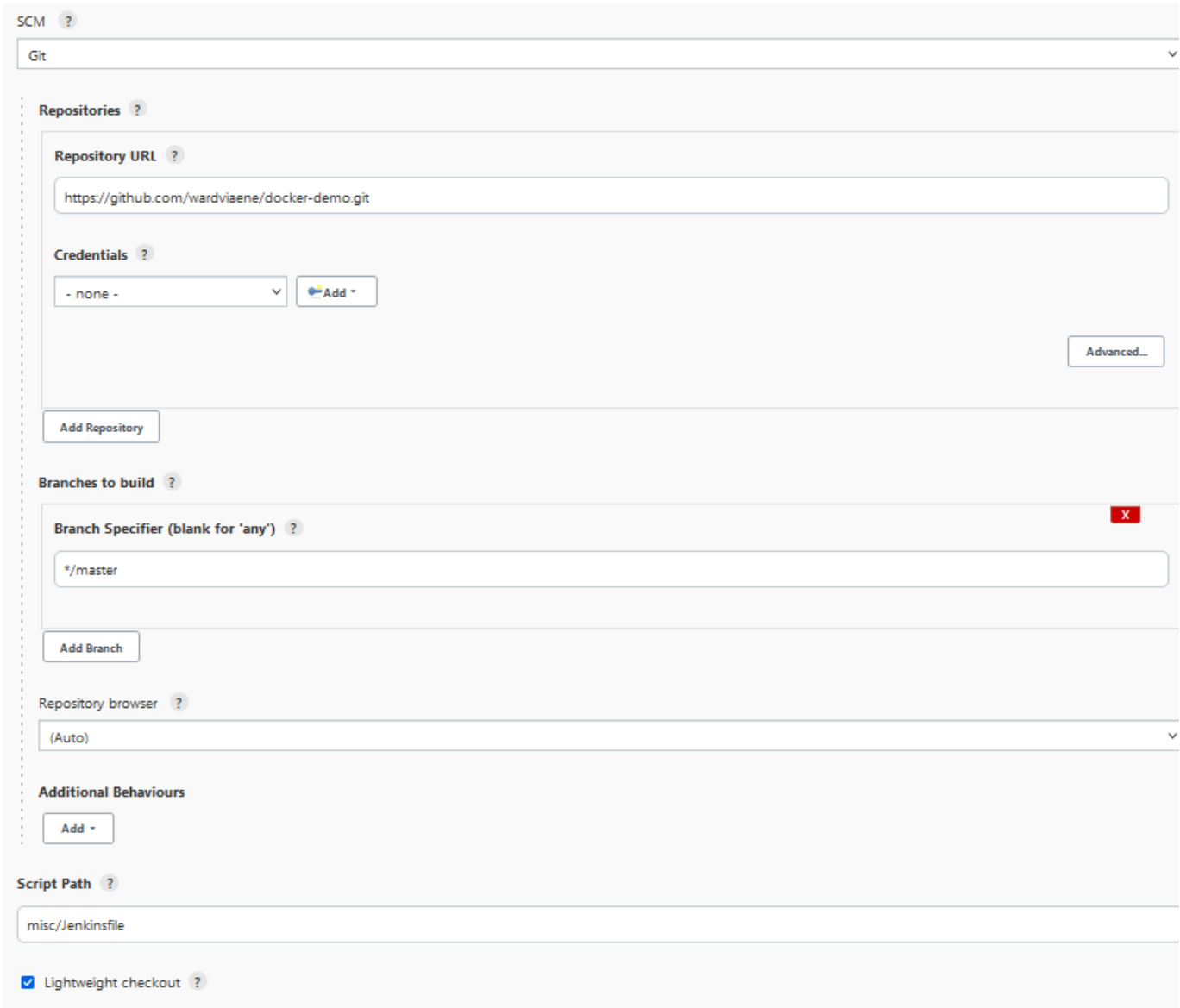
Jenkins Pipelines:

- Jenkins Pipelines allow you to write the **Jenkins build steps in code**
  - **1) Build steps** allow you to write build (compile), test, deploy in code
  - **2) Code** means you can put this code in version control

It's about automating this cycle:

```
node {

   def mvnHome

   stage('Preparation') {
      git 'https://github.com/wardviaene/java-demo.git'
      // Get the Maven tool.
      // ** NOTE: This 'M3' Maven tool must be configured
      // **       in the global configuration.
      mvnHome = tool 'M3'
   }

   stage('Build') {
      // Run the maven build
      if (isUnix()) {
         sh "'${mvnHome}/bin/mvn' -Dmaven.test.failure.ignore
clean package"
      } else {
         bat(/"${mvnHome}\bin\mvn" -Dmaven.test.failure.ignore
clean package/)
      }
   }

   stage('Results') {
      junit '**/target/surefire-reports/TEST-*.xml'
      archive 'target/*.jar'
   }

}
```

- Node: influence on what jenkins worker node the job will be ran (here: any node)
- def: allows you to declare variables
- Stage: defines a building stage: build, test, or deploy
  - Conceptually distinct step
  - Used by other plugins in Jenkins later on to visualize the stages of a job
    - e.g. clean->build->test->publish



```
node {
   def commit_id
   stage('Preparation') {
     checkout scm
     sh "git rev-parse --short HEAD > .git/commit-id"
     commit_id = readFile('.git/commit-id').trim()
   }
   stage('test') {
     nodejs(nodeJSInstallationName: 'nodejs') {
       sh 'npm install --only=dev'
       sh 'npm test'
     }
   }
   stage('docker build/push') {
     docker.withRegistry('https://index.docker.io/v2/', 'dockerhub') {
       def app = docker.build("duyng2512/docker-nodejs-demo:${commit_id}", '.').push()
     }
   }
}
```

Pipeline in containers:

```
node {
   def commit_id
   stage('Preparation') {
     checkout scm
     sh "git rev-parse --short HEAD > .git/commit-id"
     commit_id = readFile('.git/commit-id').trim()
   }
   stage('test') {
     def myTestContainer = docker.image('node:4.6')
     myTestContainer.pull()
     myTestContainer.inside {
       sh 'npm install --only=dev'
       sh 'npm test'
     }
   }
   stage('test with a DB') {
     def mysql = docker.image('mysql').run("-e MYSQL_ALLOW_EMPTY_PASSWORD=yes")
     def myTestContainer = docker.image('node:4.6')
     myTestContainer.pull()
     myTestContainer.inside("--link ${mysql.id}:mysql") { // using linking, mysql will
be available at host: mysql, port: 3306
           sh 'npm install --only=dev'
           sh 'npm test'
     }
     mysql.stop()
   }
   stage('docker build/push') {
     docker.withRegistry('https://registry.hub.docker.com', 'dockerhub') {
       def app = docker.build("duyng2512/docker-nodejs-demo:${commit_id}", '.').push()
     }
   }
}
```

1.3 Jenkins Intergration

1.3.1 Slack intergration

Slack

Workspace ?

Your team's workspace name. If you sign in to slack at https://example.slack.com/, your workspace is 'example'.

It is possible to override this setting per project.

(from Slack Notification Plugin)

Credential ?

slack-secret Add

Default channel / member id ?

#jenkins-test

Enter the channel names or user ids to which notifications should be sent. Note that this can include names of channels OR channel id numbers, e.g. "#builds", and that multiple values may appear comma separated. While names are more readable, channel ids will not change over time and are therefore more resilient.

It is possible to override this setting per project.

(from Slack Notification Plugin)

Save Apply

Icon Emoji ?

Choose a custom emoji to use as the bot's icon in Slack, requires using a bot user, e.g. :white_check_mark:

(from Slack Notification Plugin)

Username ?

jenkins-bot

Send as text ?

Override url ?

https://hooks.slack.com/services/

Your Slack-compatible-chat's (e.g. Mattermost or Rocket Chat) URL. Example: https://chat.mycompany.com/hooks/

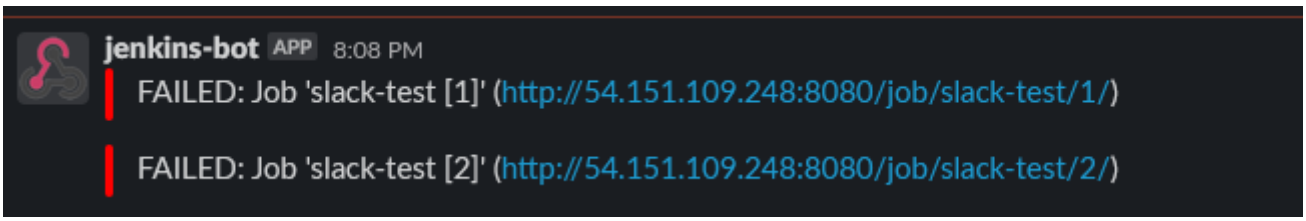
Bot-User is not supported when using this URL.

It is possible to override this setting per project.

(from Slack Notification Plugin)

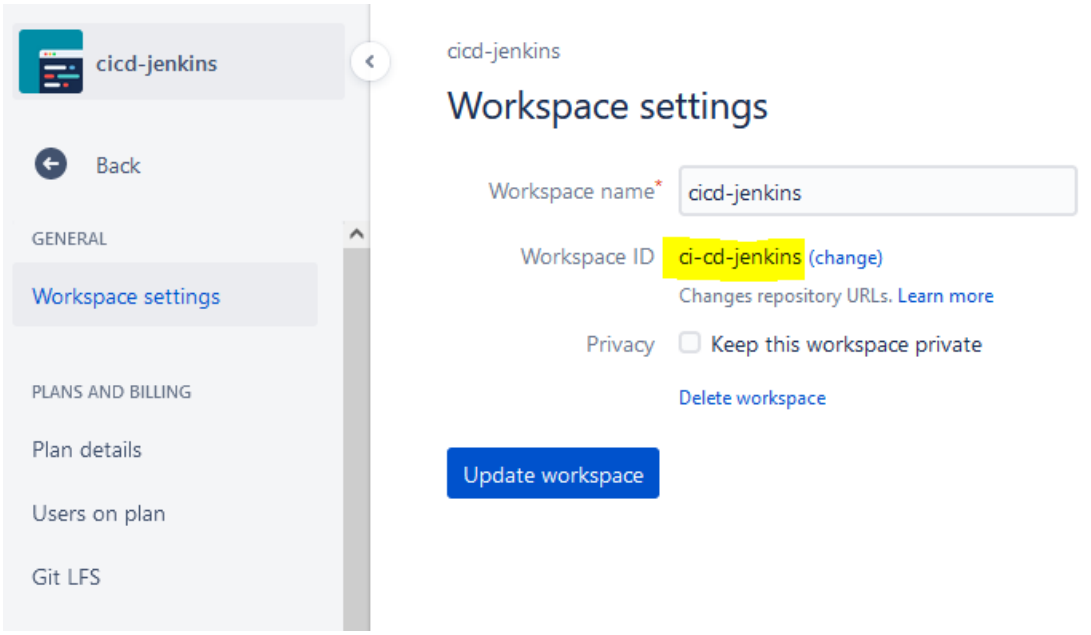
User ID Resolver

```
node {

  // job
  try {
    stage('build') {
      println('so far so good...')
    }
    stage('test') {
      println('A test has failed!')
      sh 'exit 1'
    }
  } catch(e) {
    // mark build as failed
    currentBuild.result = "FAILURE";

    // send slack notification
    slackSend (color: '#FF0000', message: "FAILED: Job '${env.JOB_NAME}
[${env.BUILD_NUMBER}]' (${env.BUILD_URL})")

    // throw the error
    throw e;
  }
}
```

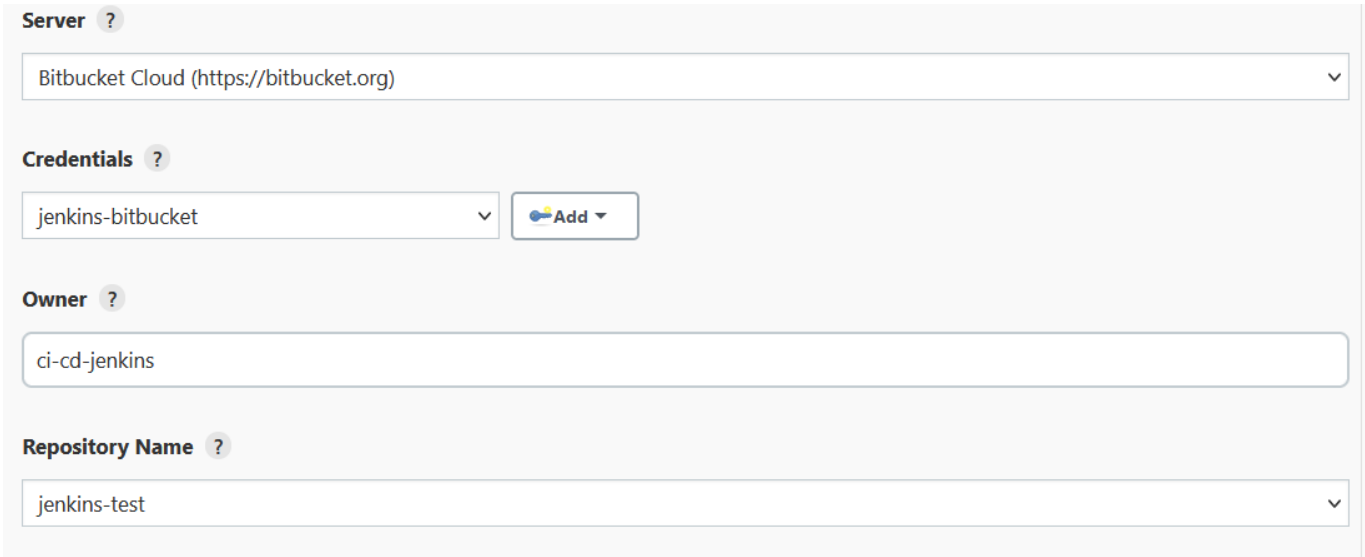


1.3.2 Bitbucket intergration

- Create new workspace



- Install bitbucket branch source plugin
- Setup jobs



1.3.3 Sonarqube integration

- Sonarqube **continuously inspects** your software project on **code quality**
- It can report on:
  - Bugs (code issues)
  - Vulnerabilities (security issues)
  - Code smells (maintainability related issues)
  - Technical debt (estimated time required to fix)
  - Code coverage (test coverage of code)
- Install database and sonarqube as a docker container on the master node
- We can use docker-compose to manage the containers
- Docker-compose is a handy tool from Docker that can spin up containers based on a container definition in yaml format
- We'll have 3 containers, a Jenkins container, a database container, and sonarqube

```
version: '2'
services:
  jenkins:
    image: jenkins-docker
    ports:
      - "8080:8080"
      - "50000:50000"
    networks:
      - jenkins
    volumes:
      - sonarqube:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
  postgres:
    image: postgres:9.6
    networks:
      - jenkins
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonarpasswd
    volumes:
      - /var/postgres-data:/var/lib/postgresql/data
  sonarqube:
    image: sonarqube:lts
    ports:
      - "9000:9000"
      - "9092:9092"
    networks:
      - jenkins
    environment:
      SONARQUBE_JDBC_USERNAME: sonar
      SONARQUBE_JDBC_PASSWORD: sonarpasswd
      SONARQUBE_JDBC_URL: "jdbc:postgresql://postgres:5432/sonar"
    depends_on:
      - postgres
volumes:
  sonarqube:
networks:
  jenkins:
```

Copy the appropriate `docker-compose` binary from GitHub:

```
sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-
compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose
```

NOTE: to get the latest version (thanks @spodnet): `sudo curl -L https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose`

Fix permissions after download:

`sudo chmod +x /usr/local/bin/docker-compose`

Verify success:

`docker-compose version`

Run yaml file:

```
docker volume create sonarqube
docker-compose up -d
```

```
docker rm -f $(docker ps -a -q --filter="name=jenkins-compose")
```

```
docker ps --format "table {{.ID}}\\t{{.Image}}\\t{{.Status}}\\t{{.Names}}"
```

1.3.4 Jenkins slave

Jenkins Slaves

- Currently, only one node (one droplet) is hosting the jenkins web UI and doing all the builds
- In production environments, you typically want to host the **Jenkins web UI on a small master node**, and have **one or more worker nodes** (Jenkins Slaves)
- Using worker nodes, you can easily **expand your build capacity**
- Typically one worker has one or more **build executors** (building slots)
  - If a Jenkins node has **2 executors**, only **2 builds** can run in **parallel**
  - Other builds will get **queued**

Dynamic worker scaling

- You have plugins that can scale Jenkins slaves for you
  - **The Amazon EC2 Plugin**: if your jenkins build cluster gets overloaded, the plugin will start new slave nodes automatically using the AWS EC2 API. If after some time the nodes are idle, they'll automatically get killed
  - **Docker plugin**: This plugin uses a docker host to spin up a slave container, run a jenkins build in it, and tear it down
  - **Amazon ECS Plugin**: Same as docker plugin, but the host is now a docker orchestrator, the EC2 Container Engine, which can host the docker containers and scale out when necessary

Jenkins Slaves Benefits

- **Reduced cost**: only have the capacity you really need
- Slaves are easily **replaceable**: if a slave crashes, in can be spun up again
- The master can run on a separate node that isn't affected by the CPU/Memory load that builds generate
  - i.e. the UI will always be responsive
- You can **respond to sudden surges** in builds, capacity can be added on the fly
  - Even with manual scaling, you can quickly spin up a new machine as a Jenkins slave

Setup aws using Ansible:

- Inventory file:

```
[webserver]
172.31.5.113 ansible_user=ec2-user
ansible_ssh_private_key_file=/home/provision/ansible/jenkins-server.pem
```

- Playbook:

```
REM test connection
ansible all -m ping
```

```yaml
# Ansible playbook AWS - install docker (2021)
---
- name: "AWS - Install docker"
  hosts: jenkins-slave
  become: yes
  tasks:
    - name: Update all packages
      yum:
        name: '*'
        state: latest
        update_only: yes

    - name: Ensure a list of yum packages are installed
      yum:
        name: "{{ packages }}"
        state: latest
        update_cache: yes
      vars:
        packages:
        - python-pip
        - yum-utils
        - device-mapper-persistent-data
        - lvm2
        - amazon-linux-extras

    - name: Add extras repository
      shell: yum-config-manager --enable extras

    - name: Enable Some packages from amazon-linux-extras packages
      shell: "amazon-linux-extras enable python3.8 ansible2 docker"

    - name: clean yum metadata cache
      command: yum clean metadata
      args:
        warn: false

    - name: Ensure a list of yum packages are installed
      yum:
        name: "{{ packages }}"
        state: latest
        update_cache: yes
      vars:
        packages:
        - python3.8
        - ansible
        - docker

    - name: Enable Docker CE service at startup
      service:
        name: docker
```

```
        state: started
        enabled: yes

    - name: Upgrade pip3
      shell: "python3.8 -m pip install pip --upgrade"

    - name: Ensure Python pip packages are installed
      pip:
        name: "{{ packages }}"
        executable: /usr/local/bin/pip3.8
      vars:
        packages:
        - boto
        - boto3
        - docker-compose
```

- Run playbook

```
sudo ansible-playbook deploy-ssh.yml
sudo ansible-playbook install-docker.yml
```

```
sudo ansible-playbook setup-jenkins.yml -e 'create_file=1'
```

Slave test

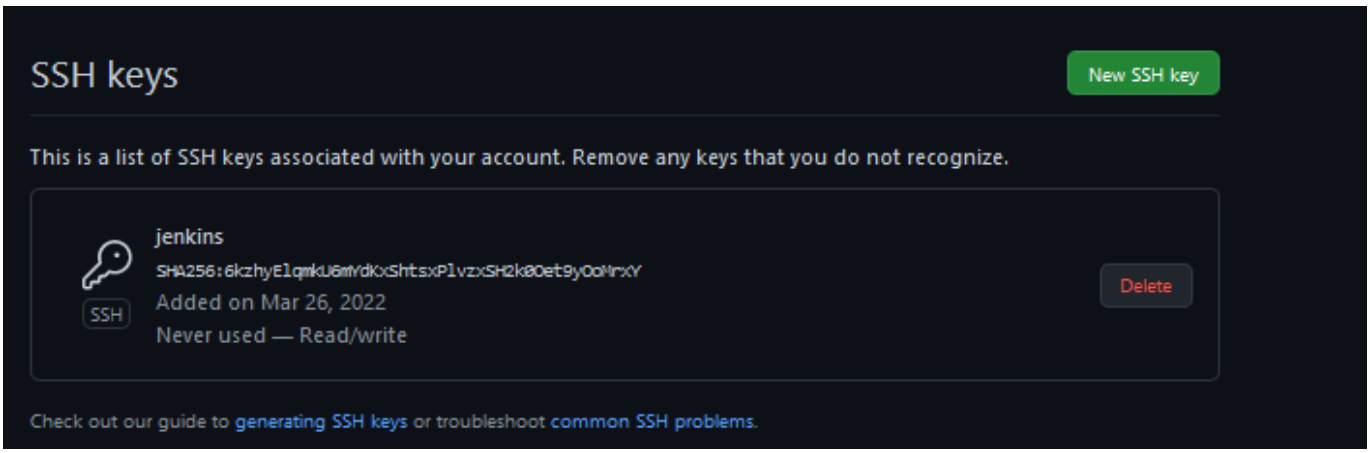
```
node(label:'builder') {
    stage('prep') {
        git 'https://github.com/duyng2512/docker-demo.git'
    }

    stage('build') {
        def con = docker.image('node:4.6')
        con.pull()
        con.inside{
            sh 'npm install'
        }
    }
}
```

1.3.5 ssh-agent

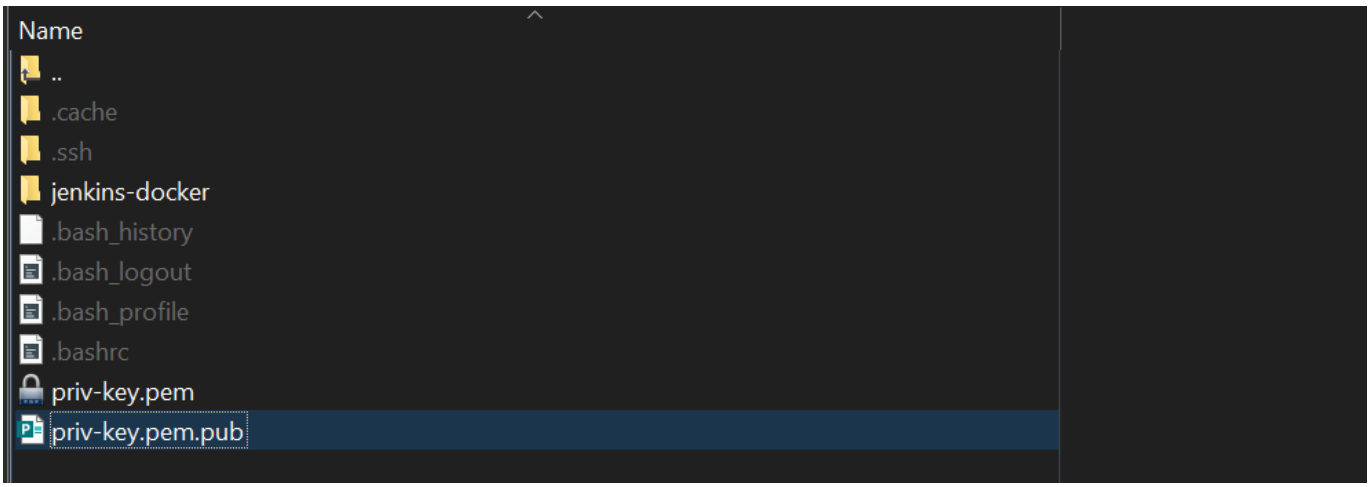
```
ssh-keygen -t ed25519 -C "duyng2512@example.com"
```

```
-----BEGIN EC PRIVATE KEY-----
MIHcAgEBBEIAJuAAzLQ3UKVUF87GuWp8CfcBhN2UstT44pzv347TQAEWpCe25wvS
/pbPpPWU3dMuZuG617HbHTy/NYzt6zjhq8egBwYFK4EEACOhgYkDgYYABABStiZ0
hDLsB1vu33vgwDTiuGG9uQBNSgLMj3a0z9v+/I5Z7trl9tUDXAZnJCqC8VlfnZ3O
c8RT8LuXcnTZ3uXCLQDxEoAYrNJx394S6dK1mt9Jbw9ikdLrl+PIItNxSLmyk9+Y
JFIJtu5kiOGsklbjbJjalInV+mq1MkH7+2nCzHc1Iw==
-----END EC PRIVATE KEY-----
```

Put the private key to Mange Jenkins credentials:



Add public key to git



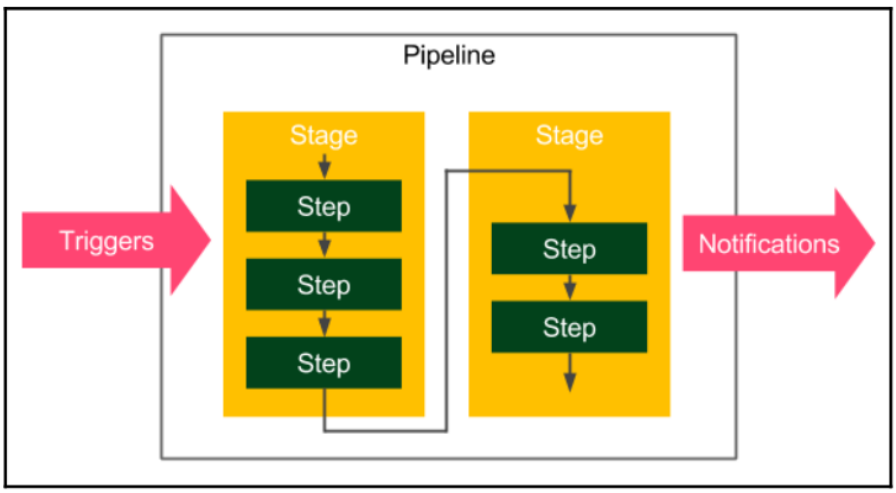
1.3.5 Jenkins best practice

- Best practices for Jenkins:
  - Try to keep your Jenkins shielded from the internet: using firewall rules and behind a **VPN**
    - You'll need to **whitelist** the bitbucket/github IP addresses for push requests
    - In the past there were **security vulnerabilities** discovered that can be exploited without being logged in, so it's better to keep Jenkins shielded away from the internet
  - Rather than managing users locally, it's best to manage users **elsewhere**
  - All enterprise companies use a central user database, often a **directory service** like Active Directory or LDAP
    - If you already have this setup, then it's quite easy for you
    - At the "Configure Global Security" page, select **LDAP** and input the settings provided to you by the AD group

1.4 Continuous Integration Pipeline

Pipeline structure

A Jenkins pipeline consists of two kinds of elements: stages and steps. The following figure shows how they are used:



Pipeline check code at github

```
pipeline {
    agent any
    stages {
        stage("Checkout") {
            steps {
                git url: 'https://github.com/leszko/calculator.git'
            }
        }
    }
}
```

Compile & test

```
stage("Compile") {
    steps {
        sh "./gradlew compileJava"
    }
}

stage("Unit test") {
    steps {
        sh "./gradlew test"
    }
}
```

Jenkinsfile

```
pipeline {
    agent any
    stages {
        stage("Compile") {
            steps {
                sh "./gradlew compileJava"
            }
        }
        stage("Unit test") {
            steps {
                sh "./gradlew test"
            }
        }
    }
}
```

Code coverage

```
stage("Code coverage") {
    steps {
        sh "./gradlew jacocoTestReport"
        sh "./gradlew jacocoTestCoverageVerification"
    }
}
```

Publishing the code coverage report

When the coverage is low and the pipeline fails, it would be useful to look at the code coverage report and find what parts are not yet covered with tests. We could run Gradle locally and generate the coverage report; however, it is more convenient if Jenkins shows the report for us.

In order to publish the code coverage report in Jenkins, we need the following stage definition:

```
stage("Code coverage") {
     steps {
          sh "./gradlew jacocoTestReport"
          publishHTML (target: [
               reportDir: 'build/reports/jacoco/test/html',
               reportFiles: 'index.html',
               reportName: "JaCoCo Report"
          ])
          sh "./gradlew jacocoTestCoverageVerification"
     }
}
```

Static code analysis

Adding the Checkstyle configuration

In order to add the Checkstyle configuration, we need to define the rules against which the code is checked. We can do this by specifying the `config/checkstyle/checkstyle.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
     "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
     "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">

<module name="Checker">
     <module name="TreeWalker">
          <module name="JavadocType">
               <property name="scope" value="public"/>
          </module>
     </module>
</module>
```

Adding a static code analysis stage

We can add a `Static code analysis` stage to the pipeline:

```
stage("Static code analysis") {
     steps {
          sh "./gradlew checkstyleMain"
     }
}
```

Now, if anyone commits a file with a public class without Javadoc, the build will fail.

Triggers and notifications

Triggers

An automatic action to start the build is called the pipeline trigger. In Jenkins, there are many options to choose from; however, they all boil down to three types:

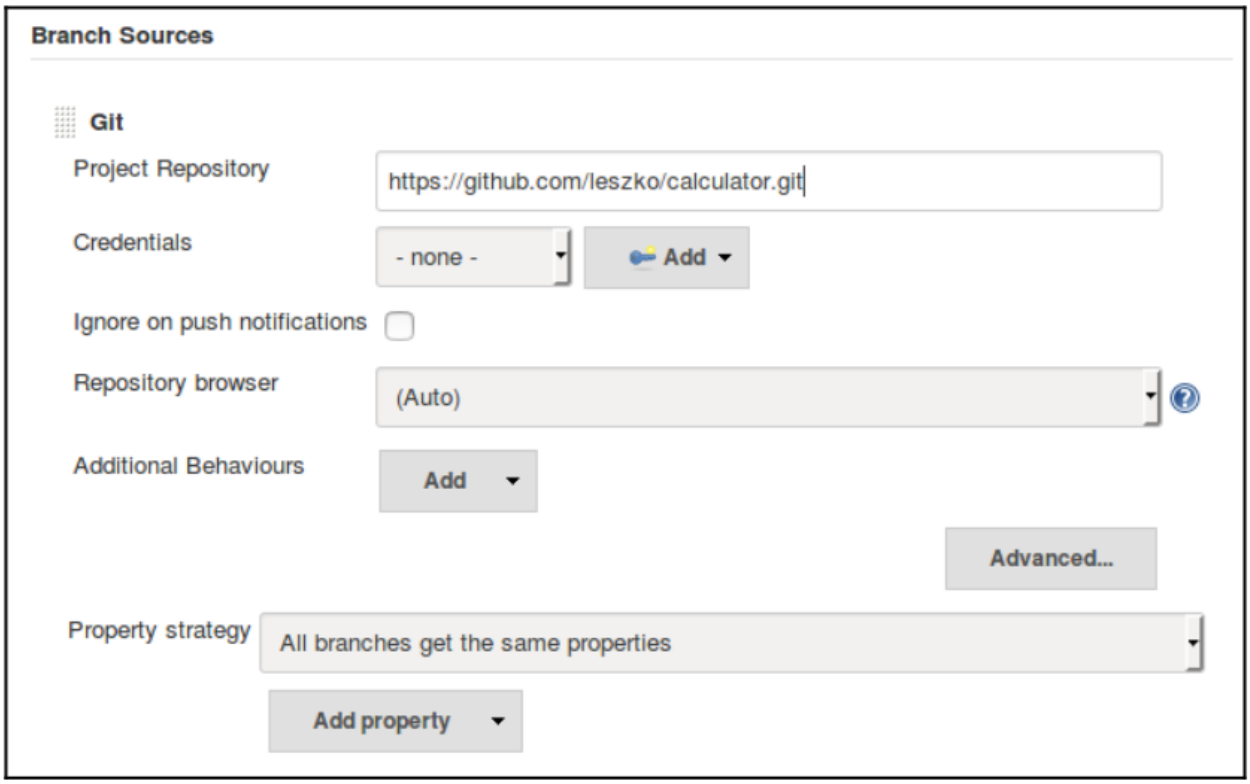
- External
- Polling SCM (Source Control Management)
- Scheduled build

Jenkins Multibranch

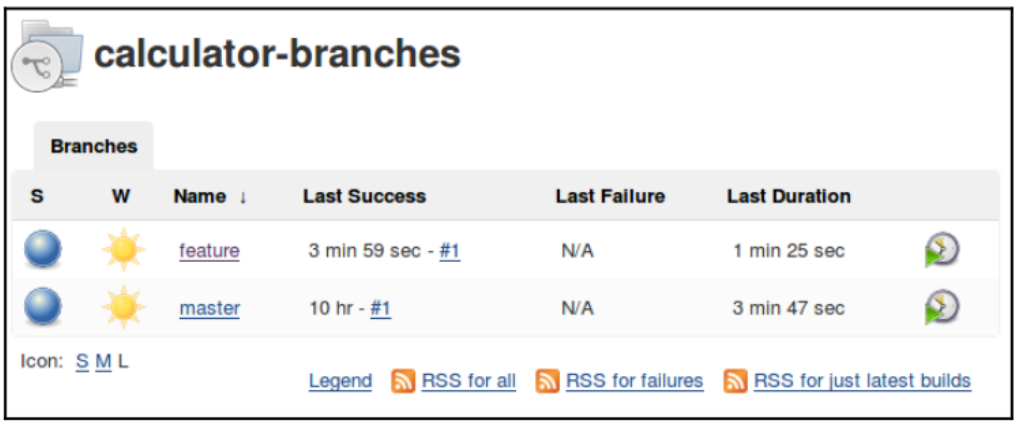
If you decide to use branches in any form, the long feature branches or the recommended short-lived branches, then it is convenient to know that the code is healthy before merging it into master. This approach results in always keeping the main codebase green and, luckily, there is an easy way to do it with Jenkins.

In order to use Multibranch in our calculator project, let's proceed with the following steps:

1. Open the main Jenkins page.
2. Click on **New Item**.
3. Enter `calculator-branches` as the item name, select **Multibranch Pipeline**, and click on **OK**.
4. In the **Branch Sources** section, click on **Add source**, and select **Git**.
5. Enter the repository address into **Project Repository**.



After a moment, you should see a new branch pipeline automatically created and run:

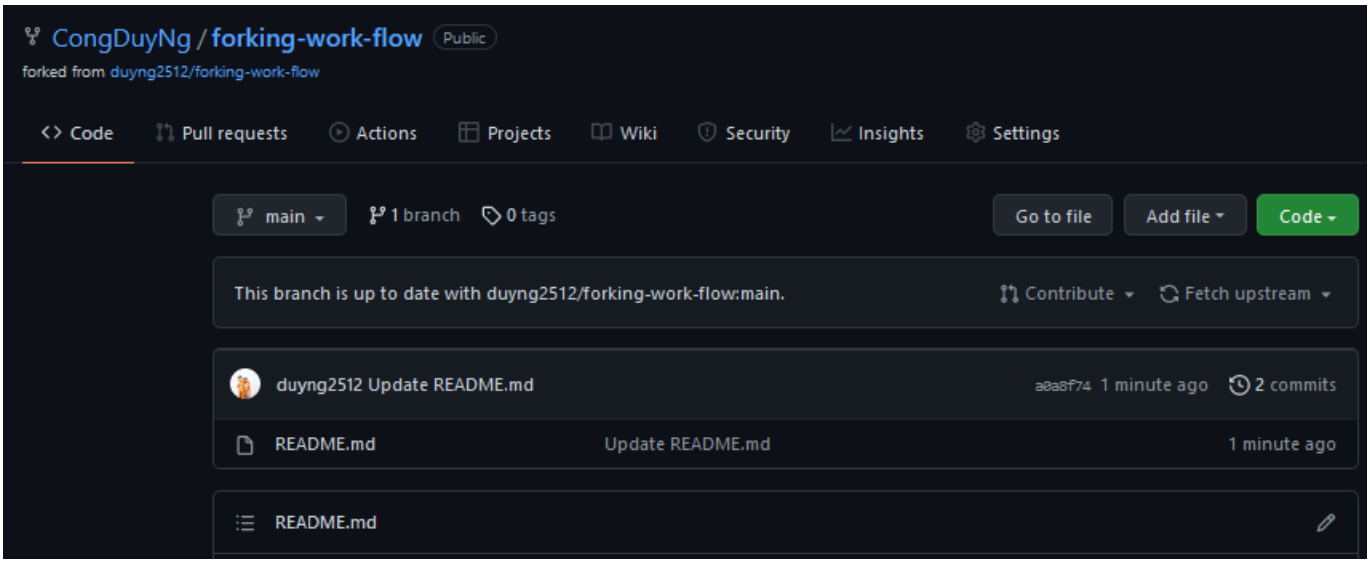


Git flow:

Forking workflow

> The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open source projects.

- Developer A fork repository



> Instead, they fork the official repository to create a copy of it on the server. This new copy serves as their personal public repository—no other developers are allowed to push to it, but they can pull changes from it (we'll see why this is important in a moment). After they have created their server-side copy, the developer performs a git clone to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

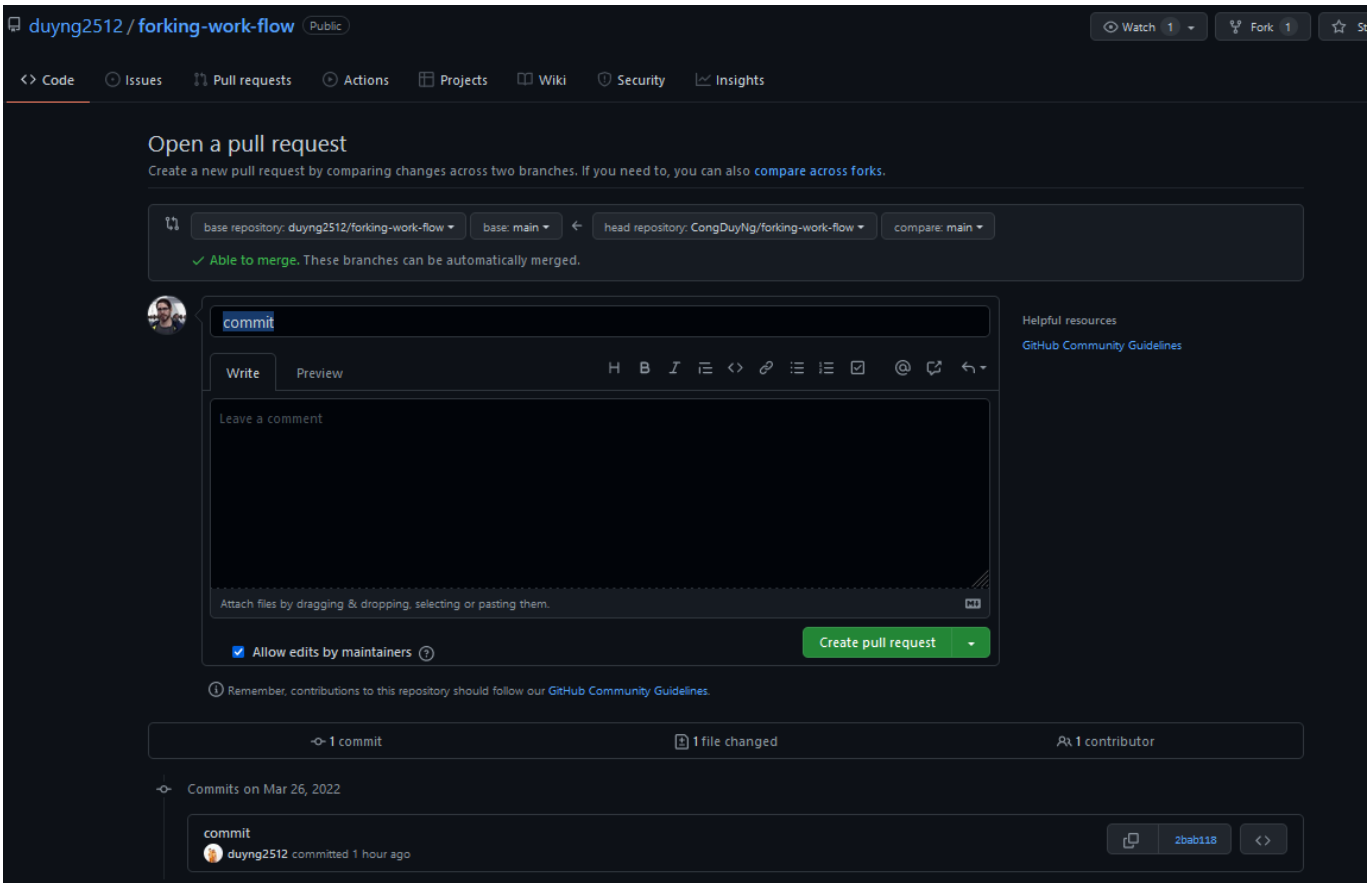
- Clone to local machine

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The pull request also serves as a convenient discussion thread if there are issues with the contributed code. The following is a step-by-step example of this workflow.

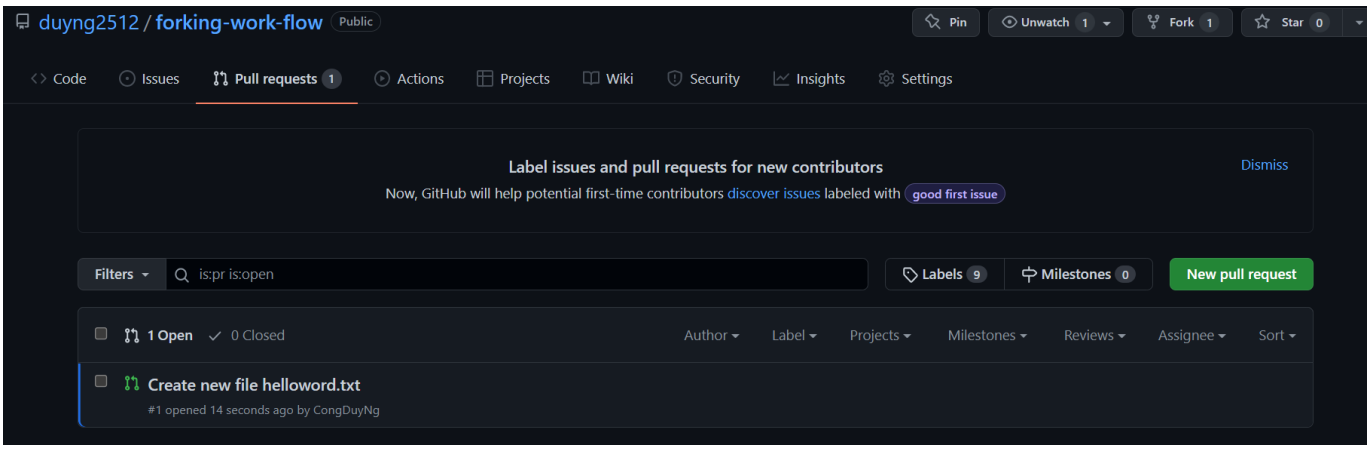
- Change in local machine, add and commit and push to developers's repository

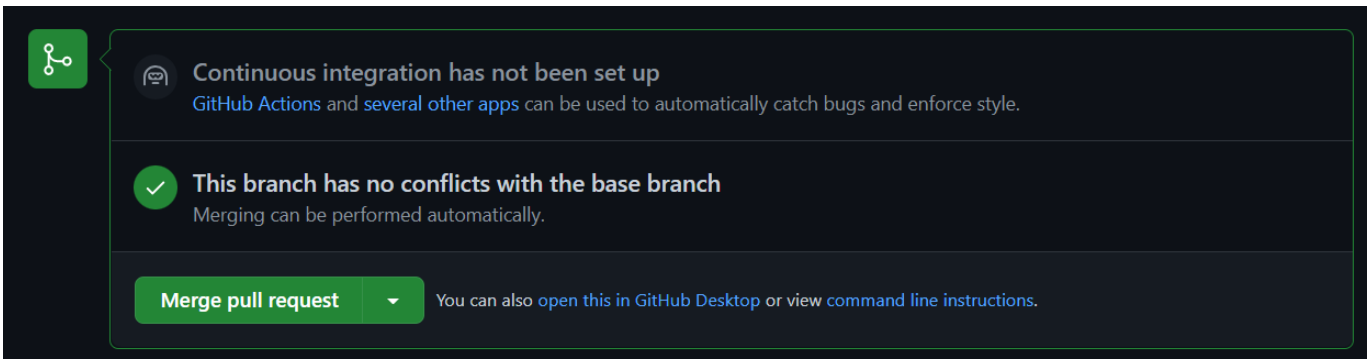
```
git push
https://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@github.com/CongDuyNg/forking-work-flow.git
```

- Create PR



- Checking in own repository

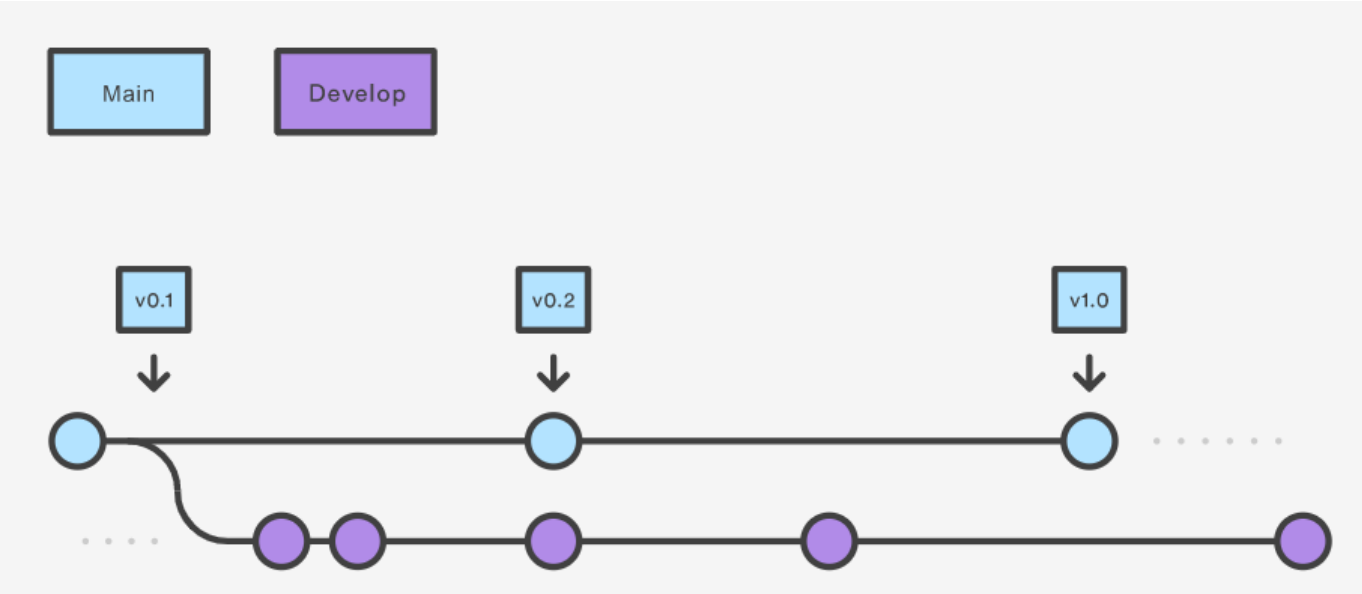


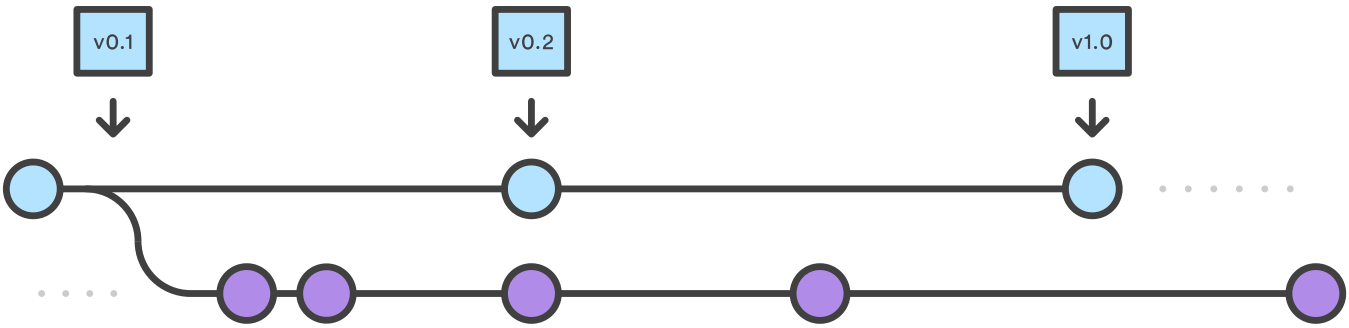


All of these personal public repositories are really just a convenient way to share branches with other developers. Everybody should still be using branches to isolate individual features, just like in the Feature Branch Workflow and the Gitflow Workflow. The only difference is how those branches get shared. In the Forking Workflow, they are pulled into another developer's local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

```
git clone https://user@bitbucket.org/user/repo.git
git remote add upstream https://bitbucket.org/maintainer/repo
git checkout -b some-feature
 # Edit some code git commit -a -m "Add first draft of some feature"
git pull upstream main
git push origin feature-branch
```

Git workflow

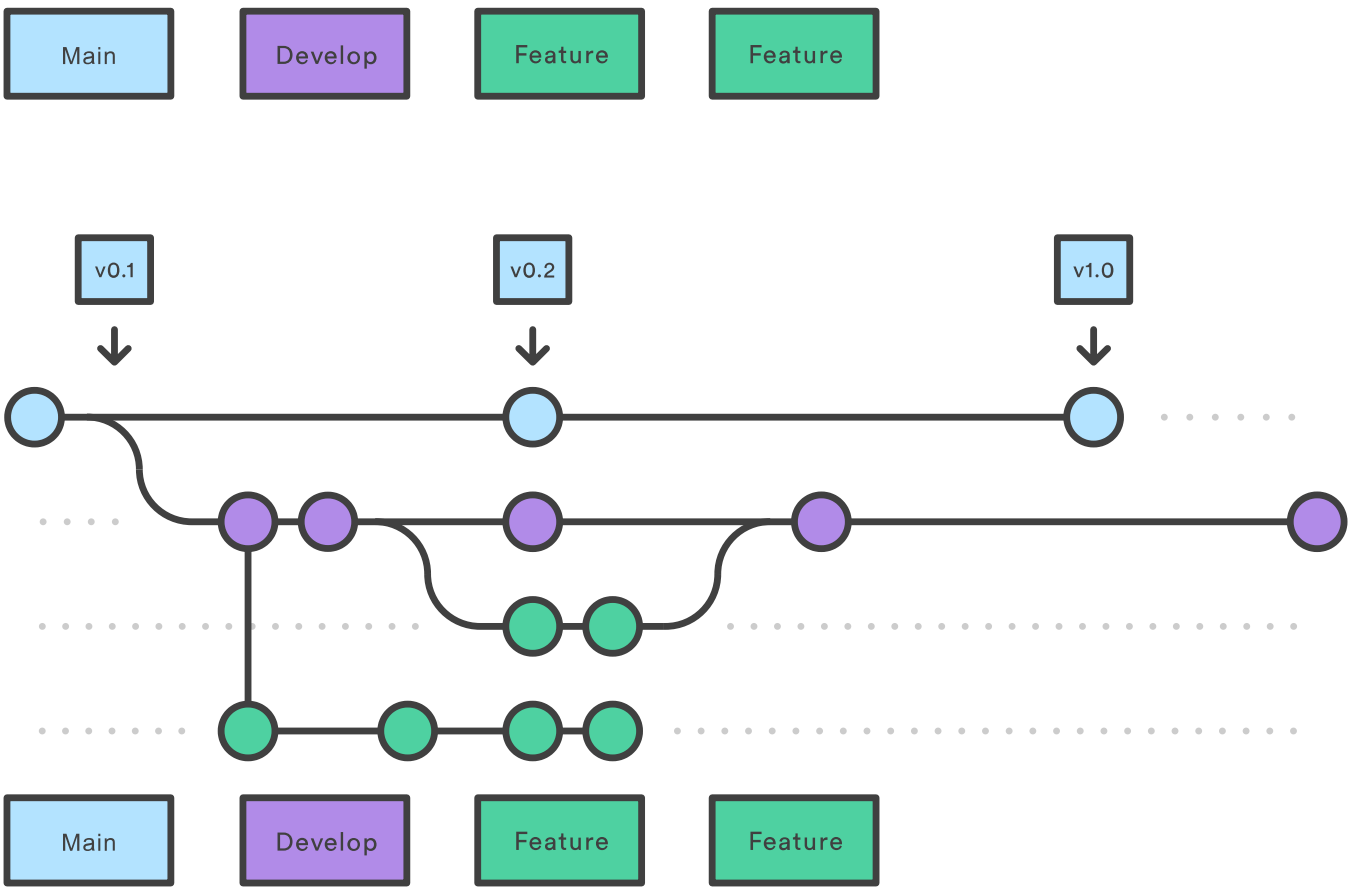


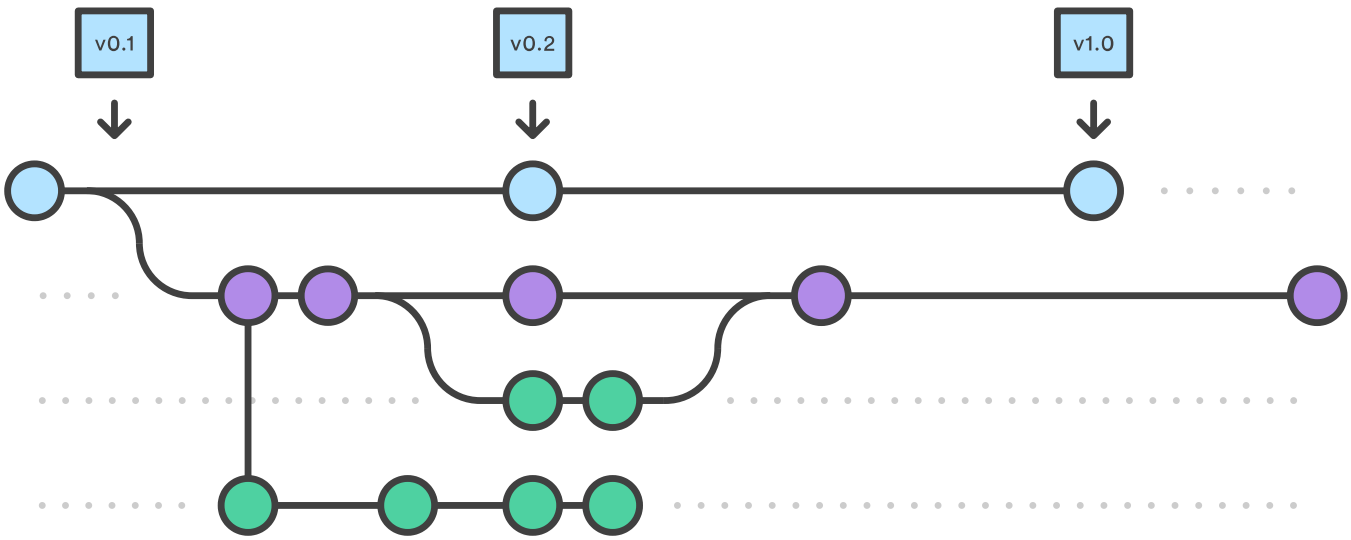


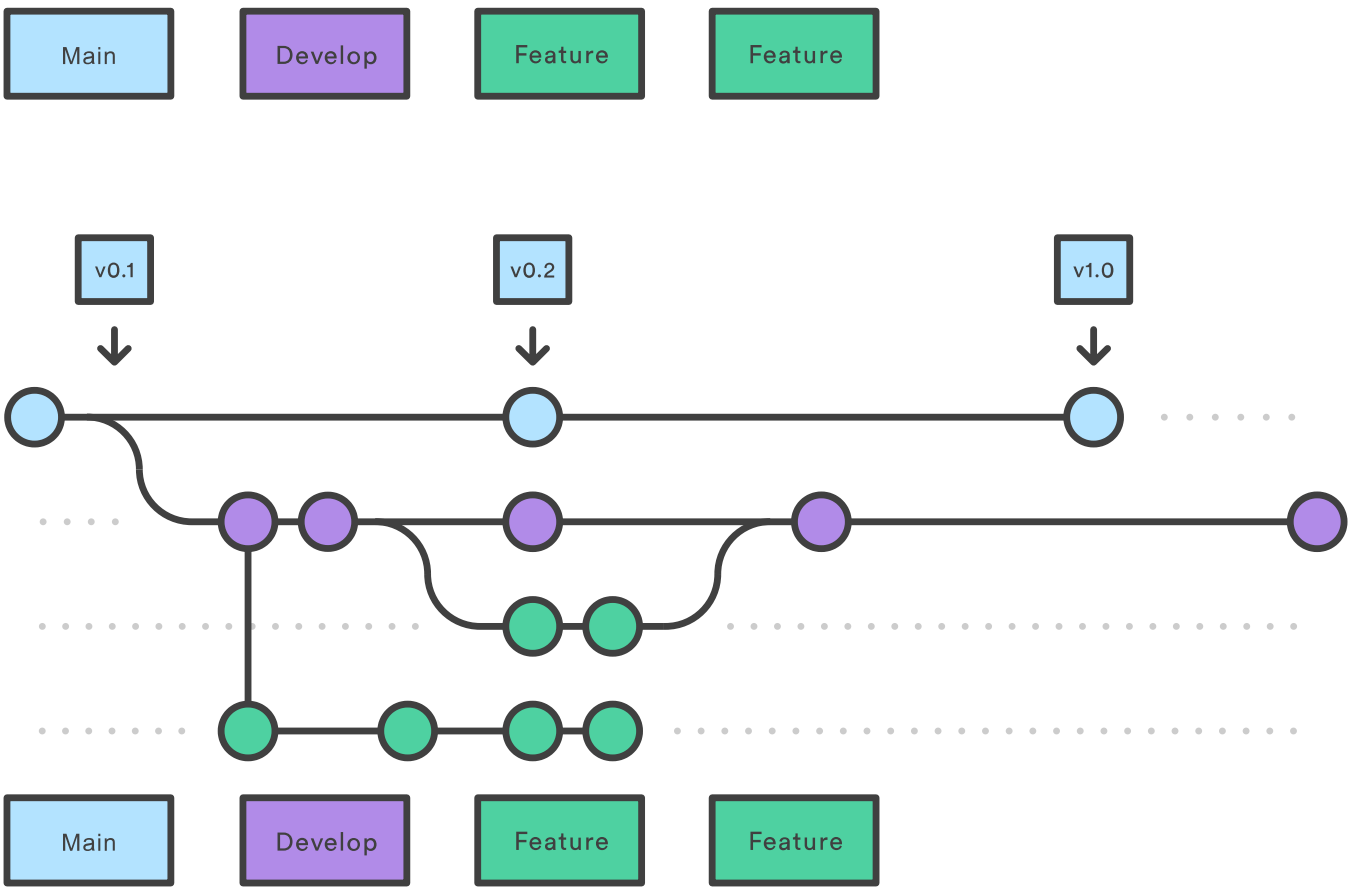
- Create development branching

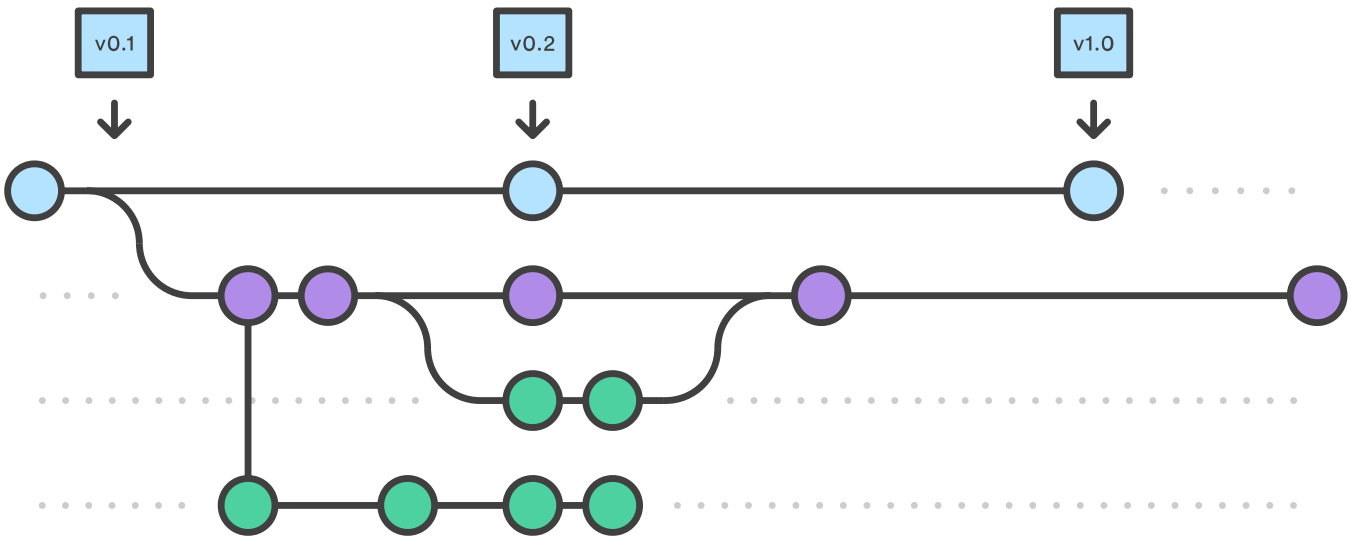
```
git branch develop
git push -u origin develop
```

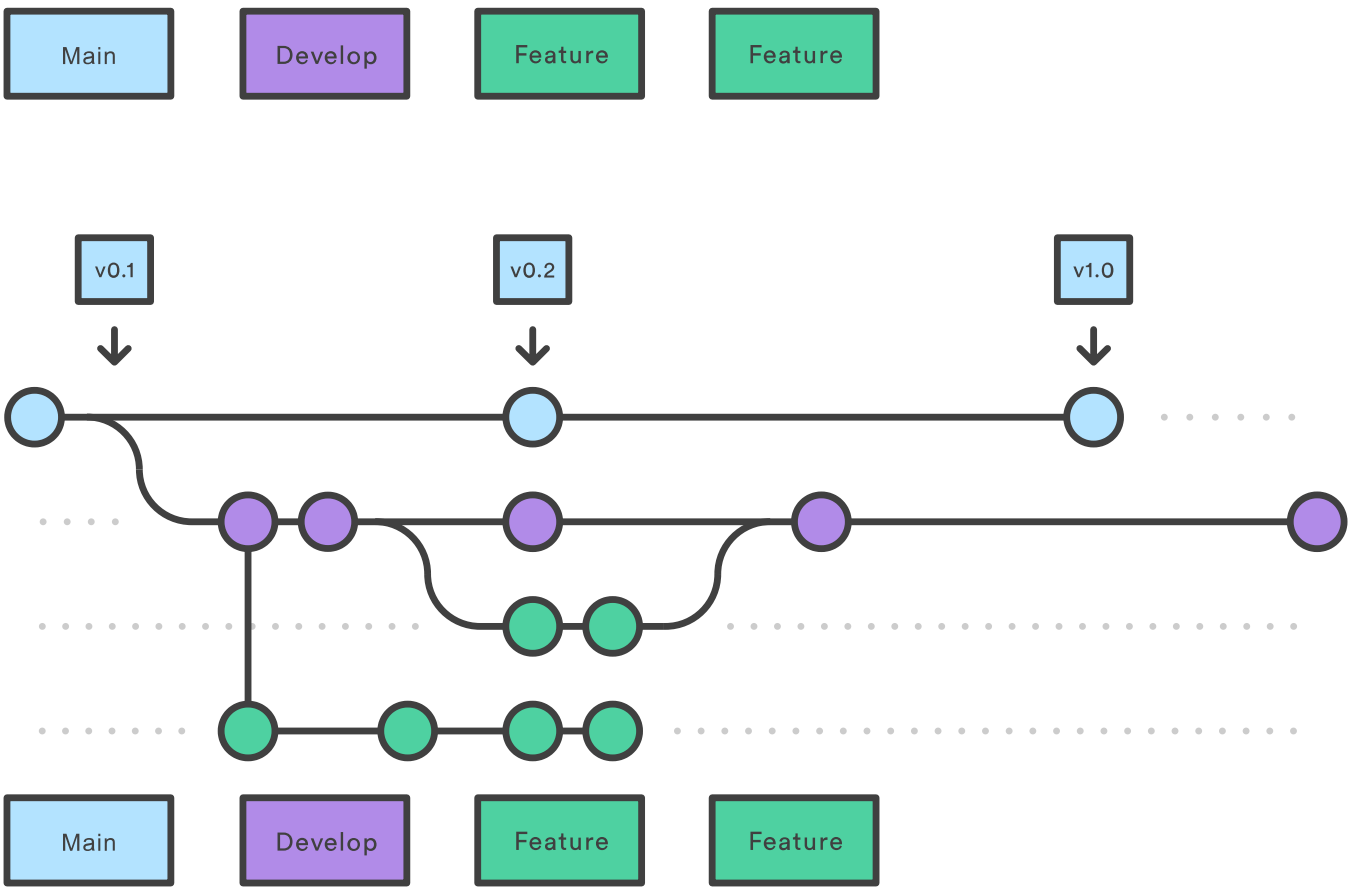
```
git flow init
git branch
git checkout develop
```

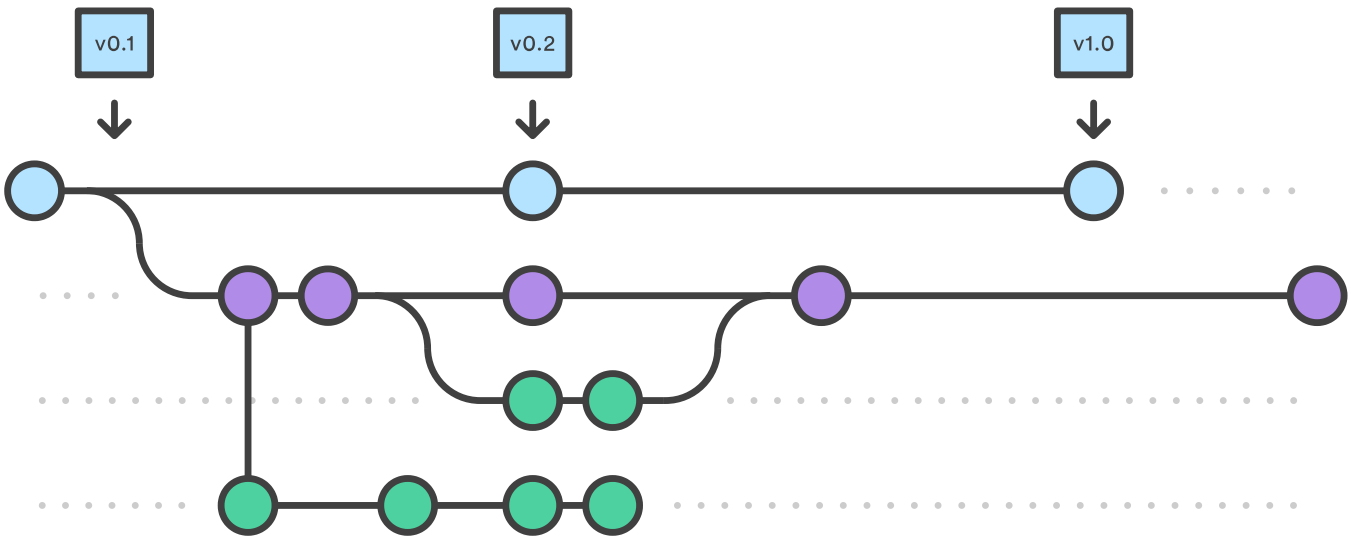

Main
Develop
Feature
Feature
v0.1
v0.2
v1.0
Main
Develop
Feature
Feature

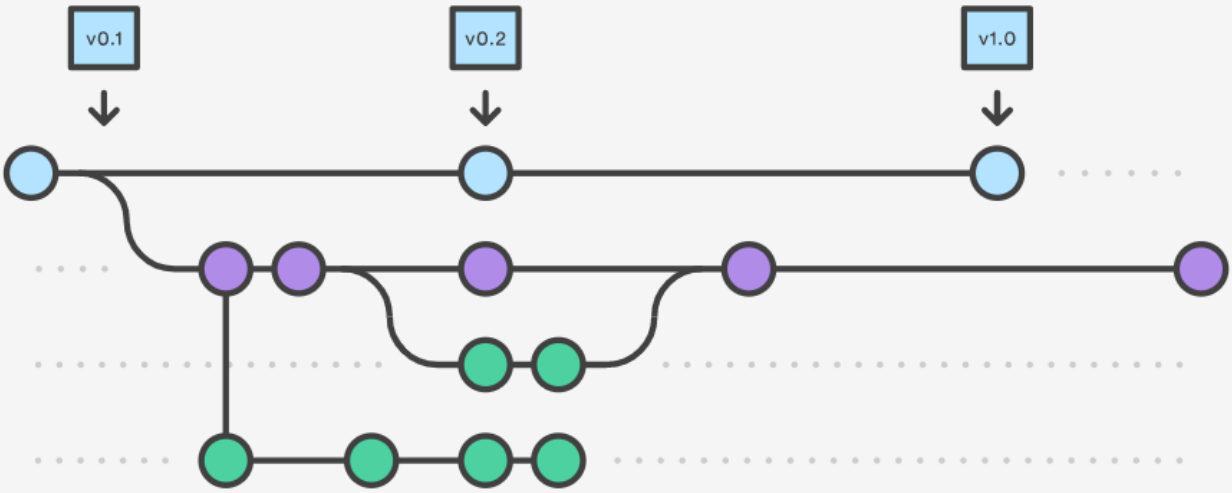

v0.1
v0.2
v1.0


Main
Develop
Feature
Feature
v0.1
v0.2
v1.0
Main
Develop
Feature
Feature


v0.1
v0.2
v1.0


Main
Develop
Feature
Feature
v0.1
v0.2
v1.0
Main
Develop
Feature
Feature


v0.1
v0.2
v1.0



Creating a feature branch

Without the git-flow extensions:

```
git checkout develop
git checkout -b feature_branch
```

When using the git-flow extension:

```
git flow feature start feature_branch
```

Continue your work and use Git like you normally would.

```
git clone https://duyshinnosuke:0903858818Duy@github.com/CongDuyNg/git-workflow.git
```

```
git flow init
git checkout develop
REM create new branch -b
git checkout -b feature_branch
```

```
REM After develop in feature_branch
git commit -am "message"

git checkout develop
git merge feature_branch
```

```
git push https://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@github.com/CongDuyNg/git-workflow.git
```

```
$gitURL = 'https://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@github.com/CongDuyNg/git-workflow.git'
```

```
git push $gitURL '*:*'
```

Standard flow:

```
git branch
git checkout -b feature_xxx
echo "fuck off" >> data.txt
git commit -am "message"
```

```
$gitURL = 'https://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@github.com/CongDuyNg/git-workflow.git'
```

```
git push -u $gitURL feature_xxx
```

```
git checkout develop
git merge feature_xxx
git push -u $gitURL develop
```

```
git remote add git-workflow https://github.com/CongDuyNg/git-workflow.git
git remote set-url git-workflow https://github.com/CongDuyNg/git-workflow.git
git push -u git-workflow feature_xxx
```

Exercises

You've learned a lot about how to configure the Continuous Integration process. Since *practice makes man perfect*, we recommend doing the following exercises:

1. Create a Python program that multiplies two numbers passed as the command-line parameters. Add unit tests and publish the project on GitHub:
    - Create two files `calculator.py` and `test_calculator.py`
    - You can use the `unittest` library at `https://docs.python.org/library/unittest.html`
    - Run the program and the unit test

2. Build the Continuous Integration pipeline for the Python calculator project:
    - Use Jenkinsfile for specifying the pipeline
    - Configure the trigger so that the pipeline runs automatically in case of any commit to the repository
    - The pipeline doesn't need the `Compile` step since Python is an interpretable language
    - Run the pipeline and observe the results
    - Try to commit the code that breaks each stage of the pipeline and observe how it is visualized in Jenkins

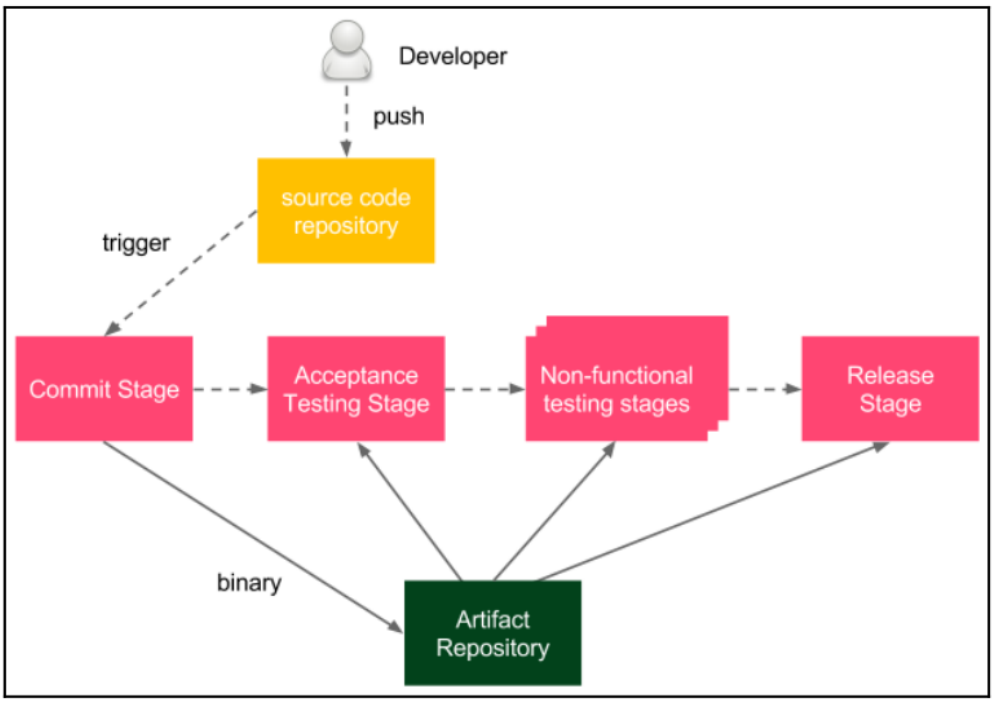
1.5 Automated Acceptance Testing

Introducing acceptance testing

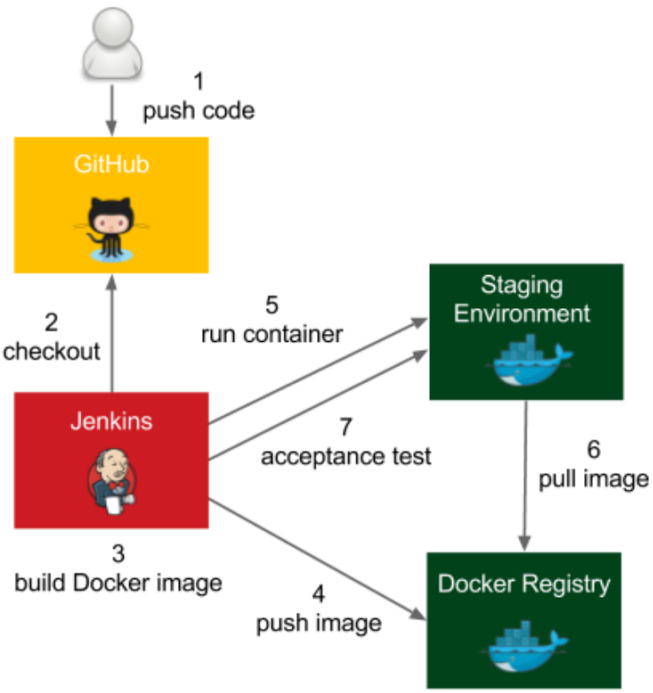
Acceptance testing is a test performed to determine if the business requirements or contracts are met. It involves black-box testing against a complete system from a user perspective and its positive result should imply the acceptance of the software delivery. Sometimes, also called **UAT (user acceptance testing)**, end user testing, or beta testing, it is a phase of the development process when software meets the *real-world* audience.

Many projects rely on manual steps performed by QAs or users to verify the functional and nonfunctional requirements, but still, it's way more reasonable to run them as programmed repeatable operations.

Artifact repository



Acceptance test in pipeline



Let's continue the pipeline we started in the previous chapter and add three more stages:

- `Docker build`
- `Docker push`
- `Acceptance test`

Adding an acceptance test to the pipeline

Acceptance testing usually requires running a dedicated black-box test suite that checks the behavior of the system. We will cover it in the *Writing acceptance tests* section. At the moment, for the sake of simplicity, let's perform acceptance testing simply by calling the web service endpoint with the `curl` tool and checking the result using the `test` command.

In the root directory of the project, let's create the `acceptance_test.sh` file:

```
#!/bin/bash
test $(curl localhost:8765/sum?a=1\&b=2) -eq 3
```

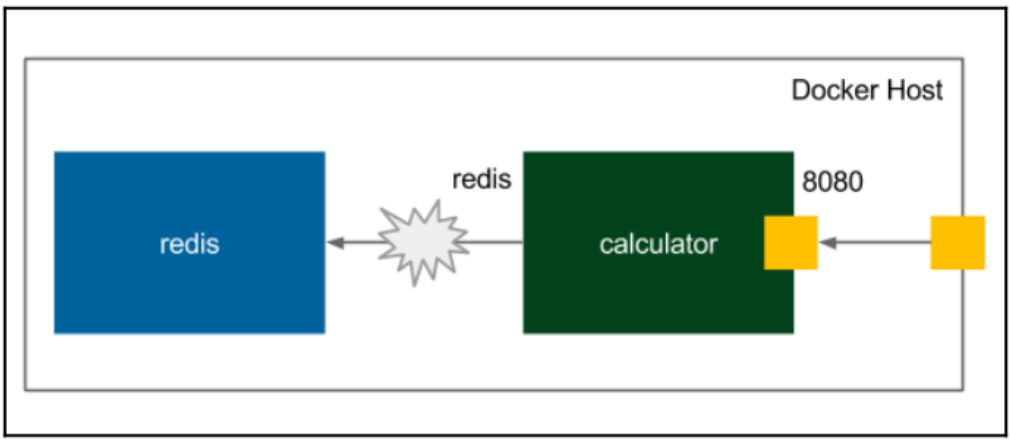
We call the `sum` endpoint with parameters `a=1` and `b=2` and expect to receive `3` in response.

Then, the `Acceptance test` stage can look as follows:

```
stage("Acceptance test") {
     steps {
          sleep 60
          sh "./acceptance_test.sh"
     }
}
```

Docker Compose

The environment configuration is presented in the following figure:



Let's see the definition of the two containers:

- **redis**: A container from the latest version of the `redis` image pulled from the official Docker Hub.
- **calculator**: A container from the latest version of the `calculator` image built locally. It publishes the `8080` port to the Docker host (which is a substitute for the `-p` option of the `docker` command). The container links to the `redis` container, which means that they share the same network and the `redis` IP address is visible under the `redis` hostname from inside the `calculator` container.

The command started two containers, `calculator` and `redis` in the background (`-d` option). We can check that the containers are running:

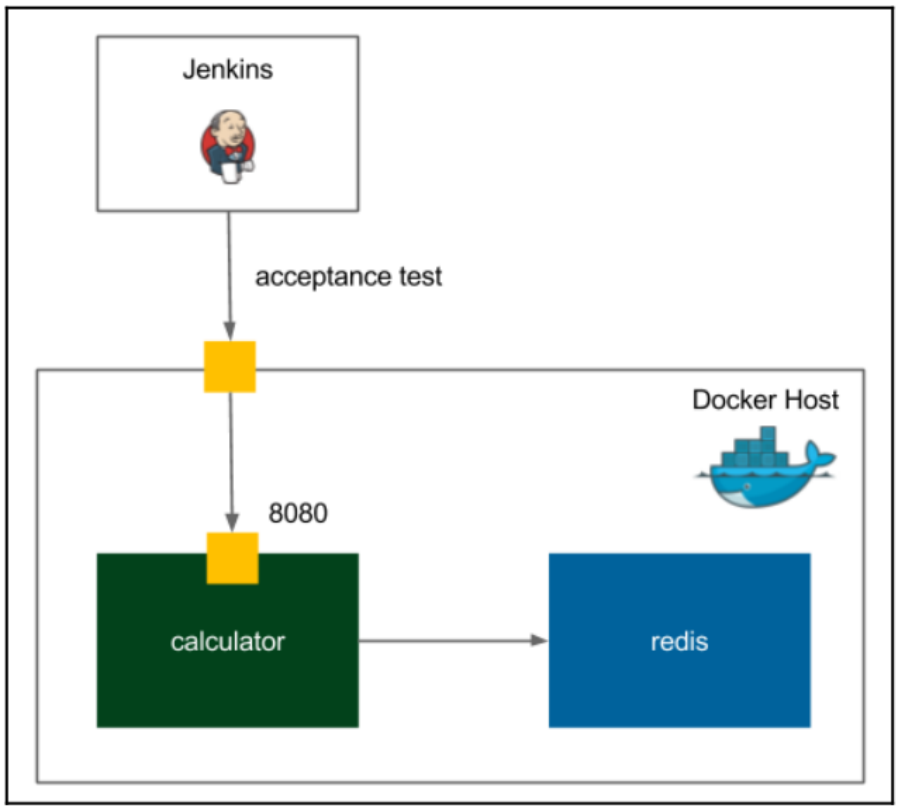
```
$ docker-compose ps
 Name                    Command             State          Ports
-------------------------------------------------------------------------------
project_calculator_1   java -jar app.jar     Up      0.0.0.0:8080->8080/tcp
project_redis_1        docker-entrypoint.sh redis ... Up 6379/tcp
```

The container names are prefixed with the project name `project`, which is taken from the name of the directory in which the `docker-compose.yml` file is placed. We could specify the project name manually using the `-p <project_name>` option. Since Docker Compose is run on top of Docker, we can also use the `docker` command to confirm that the containers are running:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND                  PORTS
360518e46bd3   calculator:latest "java -jar app.jar"
0.0.0.0:8080->8080/tcp
2268b9f1e14b   redis:latest       "docker-entrypoint..."   6379/tcp
```

Method 1 – Jenkins-first acceptance testing

The first method is to perform acceptance testing in the same way we did in the case of a single container application. The only difference is that now we have two containers running as presented in the following figure:



The `redis` container is not visible from a user perspective, so as a result, the only difference between single-container and multi-container acceptance testing is that we use the `docker-compose up` command instead of `docker run`.

Changing the staging deployment stage

Let's change the `Deploy to staging` stage to use Docker Compose:

```
stage("Deploy to staging") {
    steps {
        sh "docker-compose up -d"
    }
}
```

We must change the clean up in exactly the same way:

```
post {
    always {
        sh "docker-compose down"
    }
}
```

Changing the acceptance test stage

For the purpose of using `docker-compose scale`, we didn't specify the port number under which our web service would be published. If we did, then the scaling process would fail because all clones would try to publish under the same port number. On the contrary, we let Docker choose the port. Therefore, we need to change the `acceptance_test.sh` script to first find what the port number is and then run `curl` with the correct port number.

```
#!/bin/bash
CALCULATOR_PORT=$(docker-compose port calculator 8080 | cut -d: -f2)
test $(curl localhost:$CALCULATOR_PORT/sum?a=1\&b=2) -eq 3
```

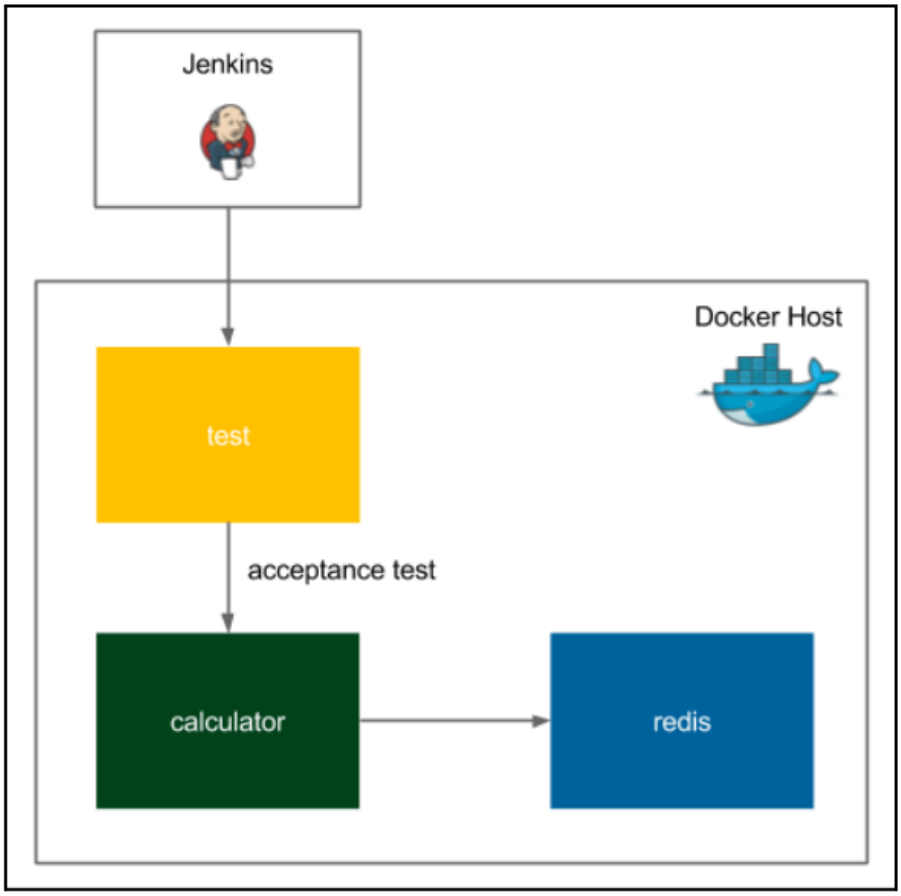
Let's figure out how we found the port number:

1. The `docker-compose port calculator 8080` command checks under which IP and port address the web service is published (it returns, for example, `127.0.0.1:57648`).
2. `cut -d: -f2` selects only port (for example, for `127.0.0.1:57648`, it returns `57648`).

We can push the change to GitHub and observe the Jenkins results. The idea is still the same as with the single-container application, set up the environment, run the acceptance test suite, and tear down the environment. Even though this acceptance testing method is good and works well, let's look at the alternative solution.

Method 2 – Docker-first acceptance testing

In the Docker-first approach, we create an additional `test` container that performs testing from inside the Docker host, as presented in the following figure:



This approach facilitates the acceptance test script in terms of retrieving the port number and can be easily run without Jenkins. It's also much more in the Docker style.

Creating a Dockerfile for acceptance test

We will start by creating a separate Dockerfile for acceptance testing. Let's create a new directory `acceptance` in the calculator project and a Dockerfile inside:

```
FROM ubuntu:trusty
RUN apt-get update && \
    apt-get install -yq curl
COPY test.sh .
CMD ["bash", "test.sh"]
```

It creates an image that runs the acceptance test.

Creating docker-compose.yml for acceptance test

In the same directory, let's create `docker-compose-acceptance.yml` to provide the testing orchestration:

```
version: "3"
services:
    test:
        build: ./acceptance
```

It creates a new container that is linked to the container being tested: `calculator`. What's more, internally it's always 8080 so that eliminates the need for the tricky part of port finding.

Creating an acceptance test script

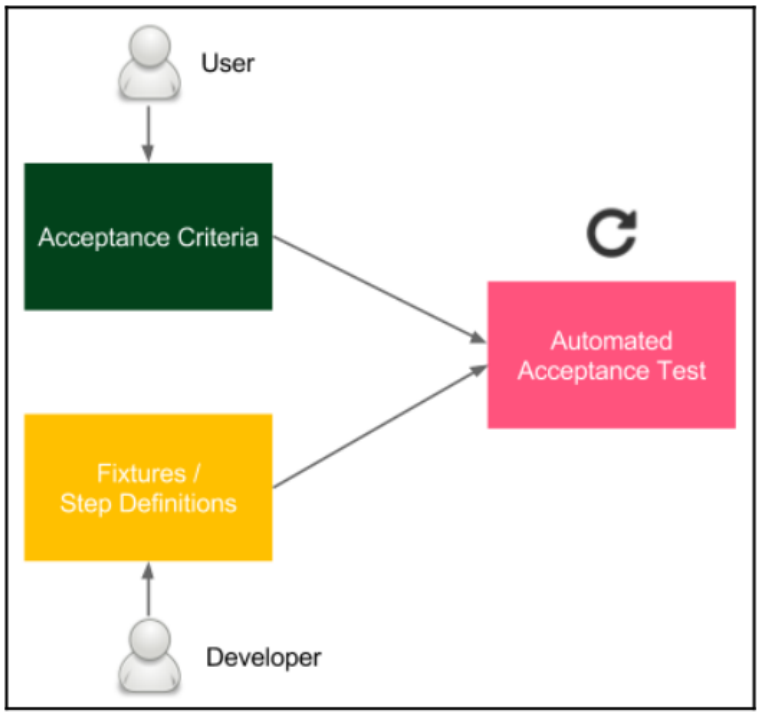
The last missing part is the test script. In the same directory, let's create the `test.sh` file that represents the acceptance test:

```
#!/bin/bash
sleep 60
test $(curl calculator:8080/sum?a=1\&b=2) -eq 3
```

It's very similar to the previous acceptance test script, the only difference is that we can address the calculator service by the `calculator` hostname and that the port number is always `8080`. Also, in this case, we wait inside the script, not in the Jenkinsfile.

Writing user-facing tests

In real life, most software is written to deliver a specific business value, and that business value is defined by non-developers. Therefore, we need a common language to collaborate. On one side, there is the business who understands what is needed but not how to do it; on the other side, the development team who knows how but doesn't know what. Luckily, there are a number of frameworks that help to connect these two worlds, for instance, Cucumber, FitNesse, JBehave, Capybara, and many more. They differ from each other, and each of them may be a subject for a separate book; however, the general idea of writing acceptance tests is the same and can be presented in the following figure:



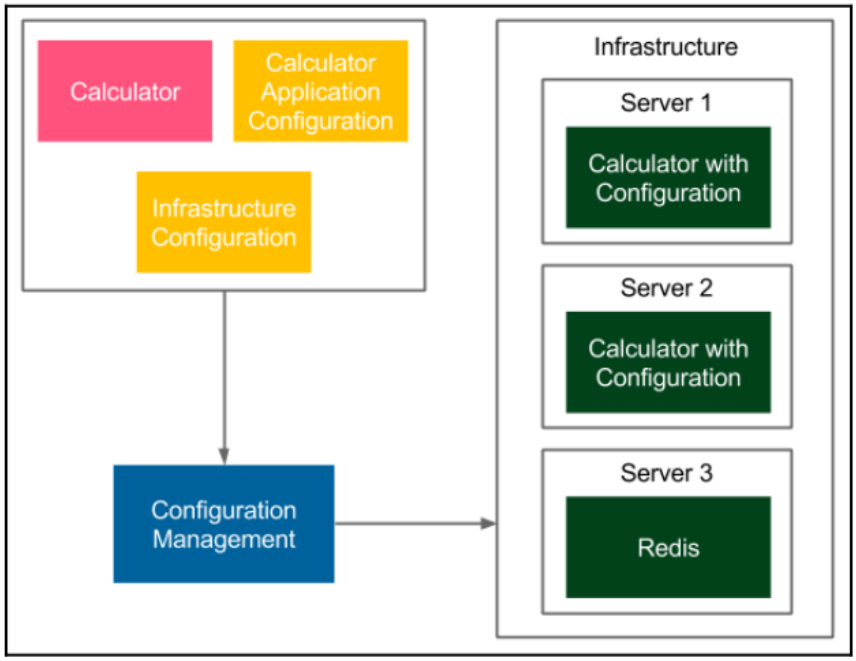
Creating acceptance criteria

Let's put the business specification in `src/test/resources/feature/calculator.feature`:

```
Feature: Calculator
    Scenario: Sum two numbers
        Given I have two numbers: 1 and 2
        When the calculator sums them
        Then I receive 3 as a result
```

This file should be created by users with the help of developers. Note that it is written in a way that non-technical people can understand it.

1.6 Configuration Management with Ansible



Overview of configuration management tools

The most popular configuration management tools are Ansible, Puppet, and Chef. Each of them is a good choice; they are all open source products with free basic versions and paid enterprise editions. The most important differences between them are:

- **Configuration Language**: Chef uses Ruby, Puppet uses its own DSL (based on Ruby), and Ansible uses YAML.
- **Agent-based**: Puppet and Chef use agents for communication, which means that each managed server needs to have a special tool installed. Ansible, on the contrary, is agentless and uses the standard SSH protocol for communication.

The agentless feature is a significant advantage because it implies no need to install anything on servers. What's more, Ansible is quickly trending upwards, which is why it was chosen for this book. Nevertheless, other tools can also be successfully used for the Continuous Delivery process.

Creating inventory

An inventory is a list of all the servers that are managed by Ansible. Each server requires nothing more than the Python interpreter and the SSH server installed. By default, Ansible assumes that the SSH keys are used for authentication; however, it is also possible to use the username and the password by adding the `--ask-pass` option to the Ansible commands.

The inventory is defined in the `/etc/ansible/hosts` file and it has the following structure:

```
[group_name]
<server1_address>
<server2_address>
...
```

The inventory syntax also accepts ranges of servers, for example, `www[01-22].company.com`. The SSH port should also be specified if it's anything other than 22 (the default one). You can read more on the official Ansible page at: `http://docs.ansible.com/ansible/intro_inventory.html`.

There may be 0 or many groups in the inventory file. As an example, let's define two machines in one group of servers.

```
[webservers]
192.168.0.241
192.168.0.242
```

We can also create the configuration with server aliases and specify the remote user:

```
[webservers]
web1 ansible_host=192.168.0.241 ansible_user=admin
web2 ansible_host=192.168.0.242 ansible_user=admin
```

Ad hoc commands

The simplest command we can run is a ping on all servers.

```
$ ansible all -m ping
web1 | SUCCESS => {
      "changed": false,
      "ping": "pong"
}
web2 | SUCCESS => {
      "changed": false,
      "ping": "pong"
}
```

Playbooks

An Ansible playbook is a configuration file, which describes how servers should be configured. It provides a way to define a sequence of tasks that should be performed on each of the machines. A playbook is expressed in the YAML configuration language, which makes it human-readable and easy to understand. Let's start with a sample playbook and then see how we can use it.

Defining a playbook

A playbook is composed of one or many plays. Each play contains a host group name, tasks to perform, and configuration details (for example, remote username or access rights). An example playbook might look like this:

```
---
- hosts: web1
  become: yes
  become_method: sudo
  tasks:
  - name: ensure apache is at the latest version
    apt: name=apache2 state=latest
  - name: ensure apache is running
    service: name=apache2 state=started enabled=yes
```

This configuration contains one play which:

- Is executed only on the host `web1`
- Gains root access using the `sudo` command

Playbook's idempotency

We can execute the command again.

```
$ ansible-playbook playbook.yml

PLAY [web1] ********************************************************************

TASK [setup] *******************************************************************
ok: [web1]

TASK [ensure apache is at the latest version] **********************************
ok: [web1]

TASK [ensure apache is running] ************************************************
ok: [web1]

PLAY RECAP *********************************************************************
web1: ok=3 changed=0 unreachable=0 failed=0
```

Note that the output is slightly different. This time the command didn't change anything on the server. That is because each Ansible module is designed to be idempotent. In other words, executing the same module many times in a sequence should have the same effect as executing it only once.

The simplest way to achieve idempotency is to always first check if the task hasn't been executed yet, and execute it only if it hasn't. Idempotency is a powerful feature and we should always write our Ansible tasks this way.

If all tasks are idempotent, then we can execute them as many times as we want. In that context, we can think of the playbook as a description of the desired state of remote machines. Then, the `ansible-playbook` command takes care of bringing the machine (or group of machines) into that state.

Handlers

Some operations should be executed only if some other task changed. For example, imagine that you copy the configuration file to the remote machine and the Apache server should be restarted only if the configuration file has changed. How to approach such a case?

For example, imagine that you copy the configuration file to the remote machine and the Apache server should be restarted only if the configuration file has changed. How to approach such a case?

Ansible provides an event-oriented mechanism to notify about the changes. In order to use it, we need to know two keywords:

- `handlers`: This specifies the tasks executed when notified
- `notify`: This specifies the handlers that should be executed

Example:

Let's look at an example of how we could copy the configuration to the server and restart Apache only if the configuration has changed.

```
tasks:
- name: copy configuration
  copy:
    src: foo.conf
    dest: /etc/foo.conf
  notify:
  - restart apache
handlers:
- name: restart apache
  service:
    name: apache2
    state: restarted
```

Handler only activate if foo.conf content change

Ansible copied the file and restarted the Apache server. It's important to understand that if we run the command again, nothing will happen. However, if we change the content of the `foo.conf` file and then run the `ansible-playbook` command, the file will be copied again (and the Apache server will be restarted).

```
$ echo "something" > foo.conf
$ ansible-playbook playbook.yml

...

TASK [copy configuration] ***************************************************
```

Variables

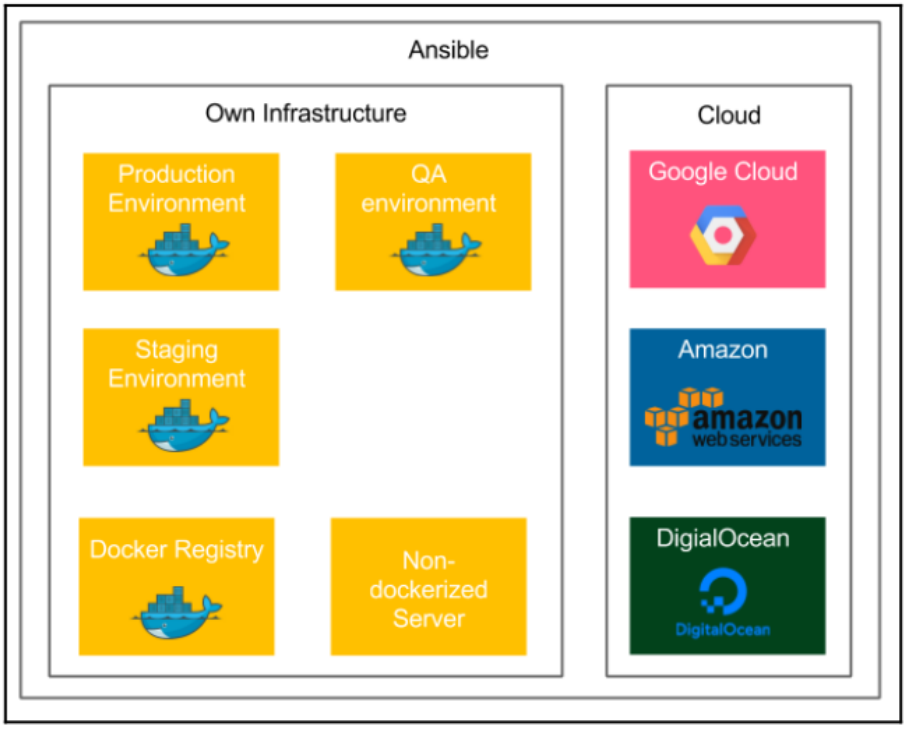
While the Ansible automation makes things identical and repeatable for multiple hosts, it is inevitable that servers may require some differences. For example, think of the application port number. It can be different depending on the machine. Luckily, Ansible provides variables, which is a good mechanism to deal with server differences. Let's create a new playbook and define a variable.

For example, think of the application port number. It can be different depending on the machine. Luckily, Ansible provides variables, which is a good mechanism to deal with server differences. Let's create a new playbook and define a variable.

```
---
- hosts: web1
  vars:
    http_port: 8080
```

The configuration defines the `http_port` variable with the value `8080`. Now, we can use it using the Jinja2 syntax.

```
tasks:
- name: print port number
  debug:
    msg: "Port number: {{http_port}}"
```



1.7 Continuous Delivery Pipeline

Types of environment

There are four most common environment types: production, staging, QA (testing), and development. Let's discuss each of them and its infrastructure.

Production

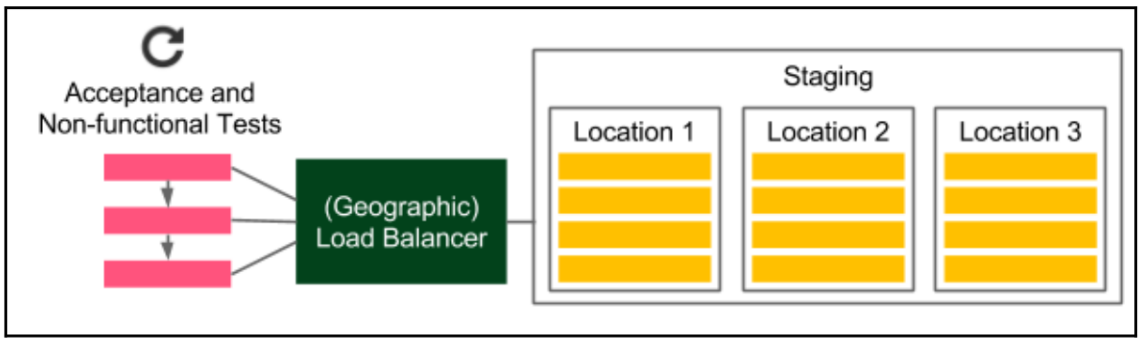
Production is the environment that is used by the end user. It exists in every company and, of course, it is the most important environment.

Let's look at the following diagram and see how most production environments are organized:

Staging

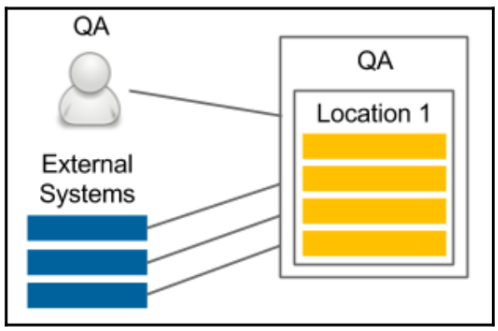
The staging environment is the place where the release candidate is deployed in order to perform the final tests before going live. Ideally, this environment is a mirror of the production.

Let's look at the following to see how such an environment should look in the context of the delivery process:



QA

The QA environment (also called the testing environment) is intended for the QA team to perform exploratory testing and for external applications (which depend on our service) to perform integration testing. The use cases and the infrastructure of the QA environment are presented in the following diagram:



Development

The development environment can be created as a shared server for all developers or each developer can have his/her own development environment. A simple diagram is presented here:

The development environment always contains the latest version of the code. It is used to enable integration between developers and can be treated the same way as the QA environment, but is used by developers, not QAs.

Environments in Continuous Delivery

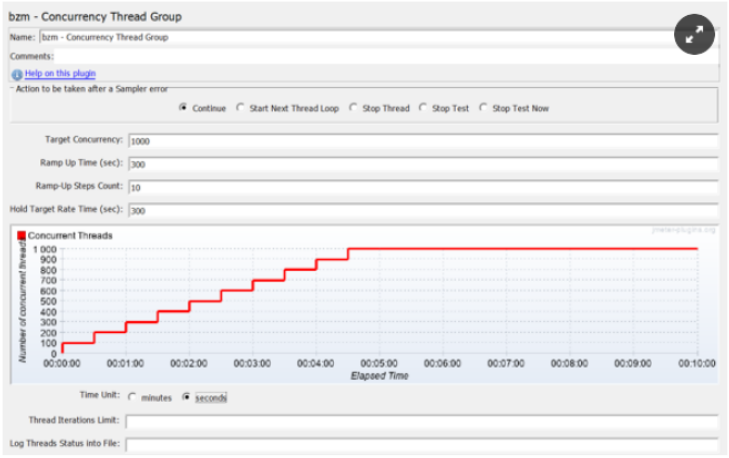
For the purpose of the Continuous Delivery process, the staging environment is indispensable. In some very rare cases, when the performance is not important and the project does not have many dependencies, we could perform the acceptance tests on the local (development) Docker host (like we did in the previous chapter), but that should be an exception, not a rule. In such a case, we always risk some production issues related to the environment.

Performance Testing:

Load testing

Load tests are used to check how the system functions when there are a lot of concurrent requests. While a system can be very fast with a single request, it does not mean that it works fast enough with 1,000 requests at the same time. During load testing, we measure the average request-response time of many concurrent calls, usually performed from many machines. Load testing is a very common QA phase in the release cycle. To automate it, we can use the same tools as with the simple performance test; however, in the case of larger systems, we may need a separate client environment to perform a large number of concurrent requests.

The following figure shows what a load test can look like in JMeter. This test analyzes adding 100 users every 30 seconds until reaching 1,000 users. The entire stepping process takes 300 seconds. After reaching 1,000 threads, all of them will continue running and hitting the server together for 5 minutes.

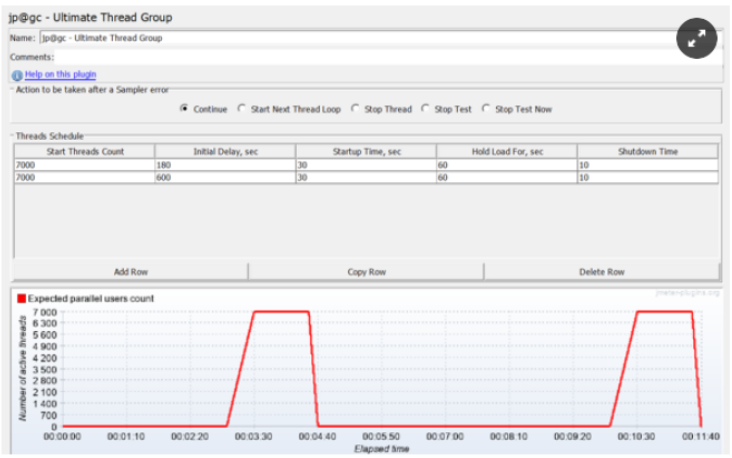


Stress testing

Stress testing, also called capacity testing or throughput testing, is a test that determines how many concurrent users can access our service. It may sound the same as load testing; however, in the case of load testing, we set the number of concurrent users (throughput) to a given number, check the response time (latency), and make the build fail if the limit is exceeded. During stress testing, however, we keep the latency constant and increase the throughput to discover the maximum number of concurrent calls when the system is still operable. So the result of a stress test may be notification that our system can handle 10,000 concurrent users, which helps us prepare for the peak usage time.

Stress testing is not well suited for the Continuous Delivery process, because it requires long tests with an increasing number of concurrent requests. It should be prepared as a separate script of a separate Jenkins pipeline and triggered on demand, when we know that the code change can cause performance issues.

The following example shows how to create a traffic spike using JMeter's "Ultimate Thread Group" component. We presume the system will be under traffic three minutes into the test. We define more threads to be added within fixed time windows using the "Initial Delay" setting.



Endurance testing

Endurance tests, also called longevity tests, run the system for a long time to see if the performance drops after a certain period of time. They detect memory leaks and stability issues. Since they require a system running for a long time, it doesn't make sense to run them inside the Continuous Delivery pipeline.

Security testing

Security testing deals with different aspects related to security mechanisms and data protection. Some security aspects are purely functional requirements such as authentication, authorization, or role assignment. These parts should be checked the same way as any other functional requirement—during the acceptance test phase. There are also other security aspects that are nonfunctional; for example, the system should be protected against SQL injection. No client would probably specify such a requirement, but it's implicit.

Security tests should be included in Continuous Delivery as a pipeline stage. They can be written using the same frameworks as the acceptance tests or with dedicated security testing frameworks, for example, BDD security.

Recovery testing

Recovery testing is a technique to determine how quickly the system can recover after it crashed because of a software or hardware failure. The best case would be if the system does not fail at all, even if a part of its services is down. Some companies even perform production failures on purpose to check if they can survive a disaster. The best known example is Netflix and their Chaos Monkey tool, which randomly terminates random instances of the production environment. Such an approach forces engineers to write code that makes systems resilient to failures.

Complete Continuous Delivery pipeline

- Create the inventory of staging and production environments

```
# inventory/staging
[webservers]
web1 ansible_host=192.168.0.241 ansible_user=admin


# inventory/production
[webservers]
web2 ansible_host=192.168.0.242 ansible_user=admin
```

- Update acceptance tests to use the remote host (instead of local)

```
stage("Deploy to staging") {
    steps {
        sh "ansible-playbook playbook.yml -i inventory/staging"
    }
}
```

If `playbook.yml` and docker-compose.yml look the same as in the *Ansible with Docker* section, then it should be enough to deploy the application with dependencies into the staging environment.

- Release the application to the production environment

```
stage("Release") {
    steps {
        sh "ansible-playbook playbook.yml -i inventory/production"
    }
}
```

- Add a smoke test which makes sure the application was successfully released

```
stage("Smoke test") {
    steps {
        sleep 60
        sh "./smoke_test.sh"
    }
}
```

Full jenkins flow:

```
pipeline {
    agent any
    triggers {
        pollSCM('* * * * *')
    }
    stages {
        stage("Compile") {
            steps {
                sh "./gradlew compileJava"
            }
        }
        stage("Unit test") {
            steps {
                sh "./gradlew test"
            }
        }
        stage("Code coverage") {
            steps {
                sh "./gradlew jacocoTestReport"
                publishHTML(target: [
                        reportDir: 'build/reports/jacoco/test/html',
                        reportFiles: 'index.html',
                        reportName: "JaCoCo Report"])
                sh "./gradlew jacocoTestCoverageVerification"
            }
        }
        stage("Static code analysis") {
            steps {
                sh "./gradlew checkstyleMain"
                publishHTML(target: [
                        reportDir: 'build/reports/checkstyle/',
                        reportFiles: 'main.html',
                        reportName: "Checkstyle Report"])
            }
        }
        stage("Build") {
            steps {
                sh "./gradlew build"
            }
        }
        stage("Docker build") {
            steps {
                sh "docker build -t leszko/calculator:${BUILD_TIMESTAMP} ."
            }
        }
        stage("Docker push") {
            steps {
                sh "docker push leszko/calculator:${BUILD_TIMESTAMP}"
            }
        }
```

```
        stage("Deploy to staging") {
            steps {
                sh "ansible-playbook playbook.yml -i inventory/staging"
                sleep 60
            }
        }
        stage("Acceptance test") {
            steps {
                sh "./acceptance_test.sh"
            }
        }
        // Performance test stages
        stage("Release") {
            steps {
                sh "ansible-playbook playbook.yml -i inventory/production"
                sleep 60
            }
        }
        stage("Smoke test") {
            steps {
                sh "./smoke_test.sh"
            }
        }
    }
}
```

1.8 Advanced Continuous Delivery

Managing database changes

Challeges of database schema change:

- Compatibility of web service
- Zero downtime
- Rollback
- Test data

Introducing database migrations

Database schema migration is a process of incremental changes to the relational database structure. Let's have a look at the following diagram to understand it better: