UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

CI with Jenkins

How to set up a CI Jenkins server for a .NET Core Pipeline

Supervisore
Fabio Massacci

Laureando
Luzzara Marco

Anno accademico 2018/2019

Ringraziamenti

Un ringraziamento speciale ai miei familiari e ai parenti più stretti, a Chiara Riccò, ai miei amici e compagni di squadra senza i quali non avrei mai potuto raggiungere questo traguardo.

Desidero anche ringraziare i colleghi di lavoro che mi hanno insegnato tanto fin dal primo giorno in Aliaslab, ormai quasi 3 anni fa.

In particolare ringrazio Sandro Bozzoli, Michael Perdomini e il docente Fabio Massacci per aver contribuito alla realizzazione di questo elaborato e per il supporto datomi durante l'esperienza di tirocinio.

Contents

Summary

This degree thesis is about the solution I found to a widespread problem in the company I have been working for almost 3 years: the slow and error-prone process of releasing software. Given the high number of microservices and the attempt to approach an Agile methodology, releasing new versions by hand had become unpracticable and extremely stressing. During my internship period and the next months, I focused on partially automating this process with the help of a CI server, Jenkins, and multiple other technologies I described here.

I basically defined a new way to structure a project and some rules that improve the quality of the project itself. I had the opportunity to discuss with several company's figures, from developers to CTOs, to continuously refine the process. The outcome is a solid foundation on which a Continuous Delivery/Deployment process can be built, the next necessary step in order to give more value to this project.

After the first learning period, I started with the setup of a Jenkins server on `localhost`, then a slave built on .NET Core image, the Artifactory server and the versioning server, all of them as Docker containers.

Next part was the implementation of a shared library and a general *Jenkinsfile*, common to all .NET Core projects. Certainly not the least was the reasoning on making a generic pipeline as configurable as possible, achieved essentially through configuration files and Dependency Injection.

After testing the pipeline, I merged my local Jenkins with the one already on the server, used by the Mobile Team. Fortunately, it took me a couple of hours, thanks to how easily configuration properties are stored. Working with docker volumes and moving them to the server through SSH was actually the longest part. Monitoring the performances and resources consumption of container involved, I was able to find an efficient grouping for them: Jenkins and its slaves are on a server, whereas Artifactory and the versioning server are on another one.

Considering the initial absence of a versioning system and disorganized projects, the CI phase has reached a good level overall. Even though, it is undeniable that it needs some improvements, especially in preparation of the CD process, that completely automates the software's release.

1 Overview

This chapter focuses on explaining what is the continuous integration by point out the correct way to adopt it in a project's lifecycle and, specifically, how Aliaslab approached it considering the requirement to quickly deploying new releases for a vast range of microservices.

1.1 What is the Continuous Integration?

CI is a software development practice where members of a team integrate their work frequently, usually at least once a day [3]. Each integration is accepted only if the automated build is successful. This procedure offers a double advantage: less time on trying to solve integration mistakes thanks to short pieces of code that need to be merged, which leads to a stable software for its almost entire lifetime. The term "Continuous Integration" originated with Kent Beck's Extreme Programming [1] development process, as one of its original twelve practices.

CI implies solid skills with the use of VCS (Versioning Control System), which helps and clearly simplifies one of the tasks at the base of this methodology: committing frequently. On the other hand, extremely long and complex branch only postpone the next release of the software and force the developer to put effort into hard-working and risky merge.

Other main principles of CI, taken from [3] and readjusted, are:

- **Maintain a Single Source Repository** – Everyone should know how to get all the necessary source code to make a build, from configuration and property files to install and testing scripts.
- **Automate the build** – In order to build a Complex project, the developer has to carry out a sequence of operations that, when handmade, are definitely error-prone and a time-consuming activity.
- **Make your build Self-Testing** – New code must not compromise working code, so a best practice is to always run the entire test battery to guarantee old functionalities. The developer is sometimes lazy and over-confident that even a small change is not going to disrupt other's work.
- **Fix broken builds as soon as possible** – it is not a bad thing for the mainline build to break, but when it happens, fixing it is a priority. In these situations, the correct way to proceed is to revert to the previous build and checking out the broken code on a development workstation.
- **Test on Multiple Environment** – You can test your software on a clone of the production environment for example, albeit it has some limitations, like If you're writing desktop software then it's not practicable to test in a clone of every possible desktop with all the third party software that different people are running. Similarly, some production environments may be prohibitively expensive to duplicate.

1.2 Aliaslab Technologies and Why CI?

Aliaslab's products are distributed in 2 ways:

- **Cloud-based**: API and services are exposed, keeping all data in proprietary data center. The payment method is a PAYG (Pay-as-you-go) cloud computing: the charge is based on usage.
- **On-premises**: the customer reserves resources and servers for the installation of our products, the company provides support and maintenance.

I am working on the digital signature team and for years I have been helping with the development of microservices based products. Thanks to this architecture, modules are loosely coupled and highly maintainable and testable. Moreover, they are deployed and shipped independently: the outcome is a high number of projects suitable for a CI process.

I have been working with the digital signature team but focused only on .NET microservices, which are further subdivided into .NET Framework and .NET Core microservices. The first set is fully supported only on Windows machines, while .NET Core projects are cross-platform. Since the company's next milestone is to port all .NET microservices on .NET Core, these ones have been the subject of my studies and researches to design an extendible, flexible and easily configurable pipeline. Because of this variety of microservices, team members belonging to customer support and delivery often spent lot of time on keeping company's products updated to last version but also compatible with customer's machine requirements and dependencies.

Another big problem that for too long concerned our team was the absence of a well-defined versioning system, essential to make sure the customer's product is compatible with a new release of a microservice.

Last but not least, the team I am working with deploys multiple times the same software. New code is initially deployed on the development environment, where everyone usually uploads changes that stay here until the next stable version is established. The stable release is then moved to the testing environment, where customers make a first attempt of integration. If the integration succeeds, the released software shifts to certification and immediately after to production environment. These last 2 environments are specular, both highly reliable and cannot allow any disservice. The difference between them is that the first one is usually for demos, whereas the second is the one the customer interacts with.

2 Technologies

In this chapter I have described the various technologies involved in my thesis, stressing out its core, namely Jenkins, and the infrastructure holding up the system, realized thanks to the flexibility that Docker provides.

2.1 Jenkins

Jenkins is an open source automation server that can be used as a simple CI server or turned into the continuous delivery hub for any projects. It became extremely popular for several reasons:

- It provides a high number of plugins
- Easy to install and configure
- Fixes and improvements are constantly released
- An active community supports it

But why did Aliaslab decide to choose Jenkins instead of other tools like CruiseControl or TeamCity? First of all, there is a team inside the company that already used Jenkins and know at least running a build and doing basic operations, it would have not been efficient to set up a new CI server when there is an already installed one. The second main reason is that Jenkins is vastly used among large companies like Facebook or Netflix, and this also means that it scales very well, both vertically (when master's load increases) and horizontally (to accommodate new teams or projects). Jenkins' solution to vertically growth is adding slaves, or agent, that run builds in the place of master. For the horizontally growth, it is possible to add multiple masters, but following certain rules to avoid depleting resources and making sure to keep them updated as often as possible (See [4]).

2.2 Docker and Containers

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. In accordance with the company, I decided to use Docker for multiple reasons:

- It supports both Linux and Windows containers and it easily allow you to build complex images, like Jenkins slaves, or start multiple services sharing volumes with docker-compose.
- IaC (Infrastructure as Code): it means managing your IT infrastructure using configuration files. Especially with Docker I saved lot of time and minimized the error rate, considering the long-lasting procedure of building images and starting virtual machines all by hand. Scripting all the steps to run a container also means changing a single file if you need to modify the setup of a machine.
- Containers are by definition isolated, a feature very useful in a unit testing environment, and they can be updated from a new image easily.

Docker has much other interesting functionalities like swarm mode or docker-machine that the company plans to use in the future, for the next phase: Continuous Delivery.

2.3 Languages

I used several programming and scripting languages:

- Groovy – it is a powerful, optionally typed and dynamic language, with static-typing and static compilation capabilities, for the Java platform. Jenkins added an embedded Groovy engine making it Pipeline's domain specific language.
- Powershell and Bash – the first one has been helpful to create aliases (executable modules) of sequences of docker commands especially when building images and containers. The second one was necessary in order to operate on the Jenkins slave.
- C# and .NET Core – installation script has been developed using .NET Core framework with C# to make it cross-platform.
- JS and Node.js – the versioning system has been centralized to a single Node.js web server exposing a single endpoint and few APIs.

3 New Rules and Guidelines

In this chapter I have listed the new guidelines a developer must follow to successfully and correctly integrate a project with Jenkins, from the project structure, an installation script, the versioning system up to the configuration of a Jenkins build.

3.1 A new Project Structure

To enable CI on a project, some rules relating the project structure have been established, mainly to standardize the creation of a project and keep it neat and tidy. The following is the basic folder tree:

```
.sln file
    CI
        installScript/
            InstallScript.dll
        Resources/
            githooks/
                commit-msg
    src/
        proj1/
            proj1.csproj file
        proj2/
            proj2.csproj file
    test/
        test_proj1/
            test_proj1.csproj file
        NewmanTests/
            Collections/
            Env/
```

The .sln file is the solution file and describes where single projects can be found and list their possible configurations. CI folder is where install script and its corresponding resources are placed.

src is the folder containing source projects, each one placed in a directory named with the same name of the .csproj itself. Test folder has the same purpose of the *src* folder, but containing only test projects. Always inside this folder, there is *NewmanTests* directory, better explained on the Test stage section.

3.2 Installation Script and Git Hooks

Git hooks have a main role on the pipeline. They can be grouped into **client-side** hooks, which are triggered when certain types of operations are executed on a local machine (like committing or merging), and **server-side** hooks that run on network operations like a pushed commit. I used two of them:

- *post-receive*: it is a server-side hook triggered when a new commit is pushed on the remote repository. It must be configured for each project's repo so that jenkins is going to be notified of a new commit instead of continuously polling the internal git server. The outcome is an outstanding reduction of a build start's delay from minutes to seconds.

  ```
  curl http://jenkins_server:port/git/notifyCommit?url=<Git repository>
  ```

- *commit-msg*: it is a client-side hook that check if the commit message complies with the regular expression

  ```
  ^(?:[rR]if(?:\.? +))?#(\d+) *(?::|-| )
  ```

  The most important part is the sequence of digits following '#'. This number is a reference to a task created on the company's project management portal.

  ```
  #!/bin/sh
  test "" != "$(grep -E '^(([rR]if(\.? +) *)?#[[:digit:]]+)' "$1")"
  || {
      echo 'Cannot commit with a message inconsistent with
          the pattern "rif. #0000: commit message"'
      exit 1
  }
  ```

The biggest problem with git hooks is that their default folder (*.git/hooks/*) cannot be versioned, although it is possible to select a local folder. This brought me to a more complicated solution involving a necessary installation script. The idea behind it was to define a build target on a *.csproj* file (an xml listing project's properties and its dependencies) doing essentially 2 things:

1. Try pulling changes from *CI\installScript*, if it does not exist then clone it.
2. Run *InstallScript.dll* inside the folder just cloned (Note: that *dotnet* command is cross-platform). This script first checks if *Resources* directory already exists and, if not, it creates both *Resources* folder and its subfolder *githooks* where *commit-msg* resides. After that, it executes

   ```
   git config core.hooksPath 'CI/Resources/githooks'
   ```

   To change the default folder for git hooks.

The real advantage of this solution is that even the script is now versioned and after a first configuration (the build target), the developer could forget of everything that concerns CI folder and git hooks. Should I change the installation script, after the next build, the project will be automatically ready to the updated CI process.

```
<Target Name="InstallScript" AfterTargets="Build">
    <Exec Command="git -C $(SolutionDir)CI\installScript pull ||
        git clone https://url_repo_with_installscript
```

```
                $(SolutionDir)CI\installScript\ ||
            echo 'repo cannot be cloned'" IgnoreExitCode="true" />
        <Exec Command="dotnet
            $(SolutionDir)CI\installScript\InstallScript.dll"
            IgnoreExitCode="true" />
    </Target>
```

(*build target*)

I obviously had to version both installation script project and its bin folder, containing the dlls, in 2 separate repositories. To avoid pushing the project's dlls each time I change something in it, I defined another build target so that every time I build with the *Release* configuration type, dlls are automatically pushed to the other repository. Here is the second build target:

```
<PropertyGroup>
    <DllsPath>
        &quot;$(SolutionDir)src\InstallScript\bin\Release\netcoreapp2.2&quot;
    </DllsPath>
    <InstallScriptDllOrigin>
        &quot;https://url_repo_with_installscript&quot;
    </InstallScriptDllOrigin>
</PropertyGroup>

<Target Name="PushDlls" AfterTargets="Build"
     Condition="'$(Configuration)' == 'Release'">
    <Exec Command="git -C $(DllsPath) init" />
    <Exec Command="git -C $(DllsPath)
        remote add origin $(InstallScriptDllOrigin) ||
        git -C $(DllsPath) remote set-url origin $(InstallScriptDllOrigin)"
        />
    <Exec Command="git -C $(DllsPath) add ." />
    <Exec Command="git -C $(DllsPath) commit -m
        &quot;push from csproj's target&quot;" />
    <Exec Command="git -C $(DllsPath) push -f origin master" />
    <Exec Command="echo 'pushed of dlls complete'"/>
</Target>
```

3.3 Software Versioning

Versioning was a critical issue for our team and definitely confusing. I was inspired by the .NET way to manage NuGet packages using Semantic Versioning. I preferred to adapt it to our requirements, so I simplified it a little. It follows the pattern `{MAJOR}.{MINOR}.{PATCH}` where, from [5]:

- **MAJOR** – when you make incompatible API changes
- **MINOR** – when you add functionality in a backwards compatible manner
- **PATCH** – when you make backwards compatible bug fixes.

But how to decide the next version for a new build on Jenkins? Each task on the project management portal (EasyRedmine for Aliaslab) has a "*Release Type*" field initialized to *Major*, *Minor*, *Patch* or *None*. *None* is the default value so that if you forget to set this field, *None* is going to trigger the build failure. When a commit is pushed, its message is matched against the previous explained regular expression to retrieve the task number. Then the Jenkins node gets from EasyRedmine the *Release Type* field. For instance:

```
Release Type: Minor
Current version: 1.2.23
Next version: 1.3.0
```

This system helps keeping the commit history clean and understandable, since the developer is essentially forced to commit after a bugfix or functionality, without merging multiple activities. This turns out to be a developer's "responsibility" towards the team, just for this reason he must be aware of what he is doing.

How to keep track of the current project's version? A Node.js web server manages all operations, including GET last version, POST a new version or DELETE an existing one (used for rollbacking if something goes wrong further during pipeline execution). In particular, the web server keeps a json file containing changes from version to version for each project. The model is:

```
{
    "versions": [
        {
            "version": "1.1.0",
            "tasks": [
                {
                    "taskId": "taskId1",
                    "commits": [
                        "message1 for taskId1"
                    ]
                }
                {
                    "taskId": "taskId2",
                    "commits": [
                        "message1 for taskId2"
                    ]
                }
            ]
        },
        {
            "version": "1.0.0",
            "tasks": [
            ]
        }
    ]
}
```

Most recent versions are on top of the list, each version has its associated tasks, each task has its commit messages that reference it. A note for the increment of the MAJOR version. The team decided that letting Jenkins taking care of advancing this version would have been too much, at least during the first period testing all these changes on the workflow. In accordance with my colleagues, I opted for a more controlled way to increment the Major, described later on the Versioning paragraph.

3.4 Pipeline's Configuration

The last step for the integration of a project with Jenkins is filling out a JSON, fed to the pipeline itself, containing configuration and endpoint to external services (like the versioning web server or binary repository).

```
{
    "slnConfiguration": "Release",
    "environment": "Sviluppo",
    "doesManuallyIncrementMajor": false ,
    "packageFolder": "Package",
    "deployFolder": "Deploy",
    "repoUrl": "https://redmine.aliaslab.net/git/prj -00774.git/",
    "testResultsPath": "./TestResults ",
    "versioningServerHostName": "172.25.2.58:9001",
    "internalRepoUrl": "http://nuget.aliaslab.net",
    "prjManagePortalUrl": "https://redmine.aliaslab.net",
    "basePathOnBinaryRepo": "IDSign",
    "internalProjects": [
        "IDSign.Snap.Integration.Utility"
    ],
    "deployProjects": [
        "IDSign.Snap.Integration"
    ],
    "artifactoryServerId": "artifactory_00"
}
```

`slnConfiguration`: the build configuration (Like Debug, Release, Docker or those declared on solution config file)
`environment`: used mainly on the integration test stage
`doesManuallyIncrementMajor`: used on versioning stage to trigger a major
`packageFolder`: folder name containing .nupkg file
`deployFolder`: folder name containing the deliverable
`repoUrl`: git repository url
`testResultsPath`: relative path from prj folder where test results are stored
`versioningServerHostName`: versioning web server's end point
`internalRepoUrl`: url of the internal packages' repository
`prjManagePortalUrl`: url of the company's project management portal
`basePathOnBinaryRepo`: base path for artifact stored in binary repo
`artifactoryServerId`: server id, configured directly on Jenkins settings
`internalProjects`: names of internal projects
`deployProjects`: names of projects to deliver

Credentials like username and password for repositories and Jenkins account are in a separate configuration file. From a security perspective it is a serious vulnerability (See [2]), this is only a temporary solution to ease the edit and access to credentials.

```
    {
        "apiKey": "apikey",
        "credentialsId": "your-credentialsid",
        "binaryRepoUser": "user",
        "binaryRepoPwd": "password",
        "jenkinsUserName": "jenkins",
        "jenkinsUserPwd": "jenkins_password"
    }
```

These JSON have been the key to add flexibility to a pipeline common to all .NET Core Projects, stages are the same as well as the code, but each project's configuration is now versioned, visible and editable by anyone.

At the start of each CI build, Jenkins checks if the build parameter *manual_build* is true: if it is the pipeline configuration json is taken from build parameters, otherwise it is pulled from git. This option has been added to let developers run customized build, like when you have to publish a *Major* version or for other purposes like testing.

4 Environment Setup

Below I have put the technical details to dockerize Jenkins (both master and agent), the Artifactory repository and the versioning server. I have prepared Dockerfile and docker-compose scripts to easily replicate all the containers.

4.1 Dockerize Jenkins

Both Jenkins master and its slaves are mounted on Linux containers since they are definitely smaller and faster than Windows container. First of all, a shared volume must be created with
`docker volume create volume_name`
so that all the Jenkins data are saved there and can be transferred on another container. The master node can be run using docker-compose command on the following file:

```
version: "2.3"
services:
  jenkins:
    image: jenkins/jenkins:lts
    user: root
    ports:
      - "8080:8080"
      - "8443:8443"
      - "50000:50000"
    volumes:
      - type: volume
        source: jenkins_data
        target: /var/jenkins_home
        volume:
          nocopy: true

volumes:
  jenkins_data:
    external: true
```

The long-term support version of Jenkins is pulled from the public docker hub. 3 port mappings are declared, so that both the user and its nodes can interact with it. If the external volume already contains Jenkins data, `external: true` must be specified not to let them be overwritten. Now that the master is running, below the steps to build a slave, a Debian container extended with the .NET Core SDK:

1. Go to *Manage Jenkins >Manage Nodes >New Node* and choose how many executors you need. An executor is a process running on the slave agent that actually executes a job. These executors are directly managed by the Jenkins master, so it is better to have multiple executors on a single slave than multiple slaves with less executors because running a slave is in any case costly.

2. Download the *.jar* file and write down both the secret and the url for the *slave-agent.jnlp*.

3. Build the slave image (Dockerfile).

4. Tag the slave with a label for .NET Core support: it is possible to tag a slave and tell the Jenkins Master to run a pipeline on a node having only a certain label. In this way even if you

have nodes built on different images, Jenkins is taking care of choosing the most work-free node compatible with the pipeline commands.

5. Before running the slave, two volumes must be created: one that is mounted on the node's *workspace* folder and one mounted on nuget cache folder. The main reason is to share data between the node itself and the container realized for unit testing.

6. Run the slave with:

```
docker run -d -v {nuget_cache_volume}:/root/.nuget/packages -v
{slave_data_volume}:{slave_working_directory}/workspace -v
/var/run/docker.sock:/var/run/docker.sock --name {container_name}
{docker_image} -jnlpUrl {jnlp_url} -secret {secret} -workDir
{slave_working_directory}
```

   The mounted file `/var/run/docker.sock` is necessary to let the slave share the socket on which the Docker daemon is listening to. The slave can therefore use the host's Docker daemon when calling Docker's API. In this way volumes are shared too.

4.2 Dockerize Artifactory and Versioning Server

Artifactory and the versioning server are dockerized too. Like Jenkins master image, a volume for each of them must be created earlier so that if the image has to be updated, all data remain in the volume. The structure of the docker-compose is almost the same, but the volume target for Artifactory is the folder */var/opt/jfrog/artifactory/data/filestore*, whereas for the versioning server is the project folder, where there are json files listing the history of each project.

Jenkins as well as Artifactory are resource-intensive containers, therefore placing them on 2 different servers was probably the best choice. For this reason, inbound firewall rules must be defined for each port mapped, otherwise they would be unreachable. On the other hand, the versioning server is quite light-weight, at least for now.

4.3 Some noteworthy Observations

About this phase I want to point out some noteworthy things:

- **Docker on Windows with Hyper-V**: for my experience, sometimes, containers are not able to communicate with the host machine even using `host.docker.internal`. My solution was creating an external virtual switch on Hyper-V and specify it through the option `--network='network_interface'` when running the container. I am probably going to need native Windows Container too because of the .NET Framework legacy.

- **NTLM security protocol**: thanks to Wireshark I found out about this protocol after several attempts trying to restore dependencies from the company's NuGet server. I solved with the installation of `gss-ntlmssp`, a package that takes care of NTLM authentication procedure.

- **Firewall rules**: running multiple containers simultaneously put an enormous strain on my local machine, so that I had to use another host where I put versioning and Artifactory containers. In order to let them communicate and accept connections from each other, I created some inbound firewall rules.

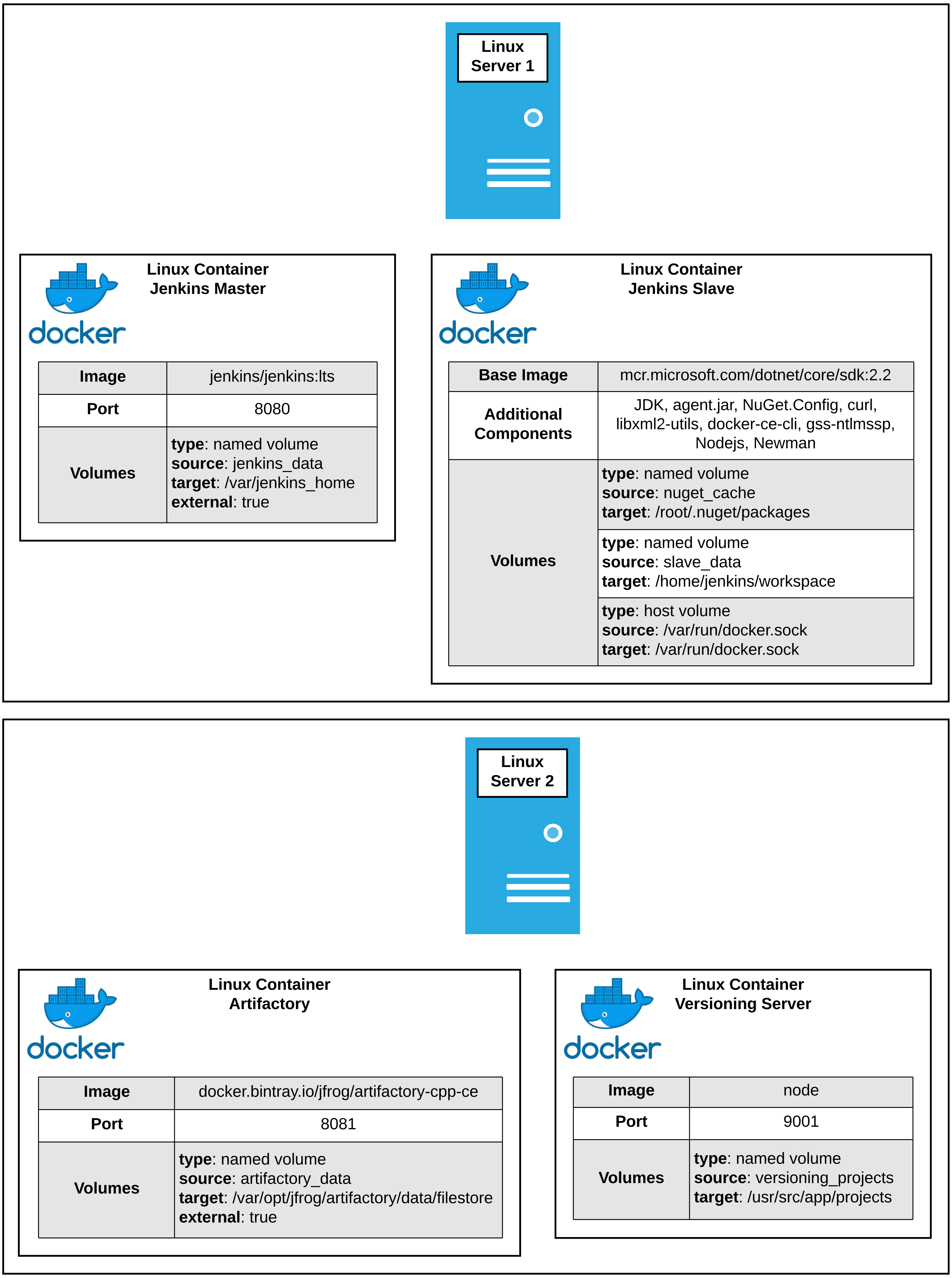

Linux Server 1
Linux Container Jenkins Master
docker
Linux Container Jenkins Slave
docker
Linux Server 2
Linux Container Artifactory
docker
Linux Container Versioning Server
docker

Image	jenkins/jenkins:lts
Port	8080
Volumes	**type**: named volume **source**: jenkins_data **target**: /var/jenkins_home **external**: true

Base Image	mcr.microsoft.com/dotnet/core/sdk:2.2
Additional Components	JDK, agent.jar, NuGet.Config, curl, libxml2-utils, docker-ce-cli, gss-ntlmssp, Nodejs, Newman
Volumes	**type**: named volume **source**: nuget_cache **target**: /root/.nuget/packages
	type: named volume **source**: slave_data **target**: /home/jenkins/workspace
	type: host volume **source**: /var/run/docker.sock **target**: /var/run/docker.sock

Image	docker.bintray.io/jfrog/artifactory-cpp-ce
Port	8081
Volumes	**type**: named volume **source**: artifactory_data **target**: /var/opt/jfrog/artifactory/data/filestore **external**: true

Image	node
Port	9001
Volumes	**type**: named volume **source**: versioning_projects **target**: /usr/src/app/projects

5 Pipeline Stages and Implementations

This chapter contains the implementation details of a generic .NET Core pipeline. Single steps can be found in Shared Libraries and are described in order of their executions, starting from the checkout of the project from the git repository, ending in a deliverable zipped and pushed on Artifactory. As well as these stages, the versioning system and several Jenkins plugins are involved.

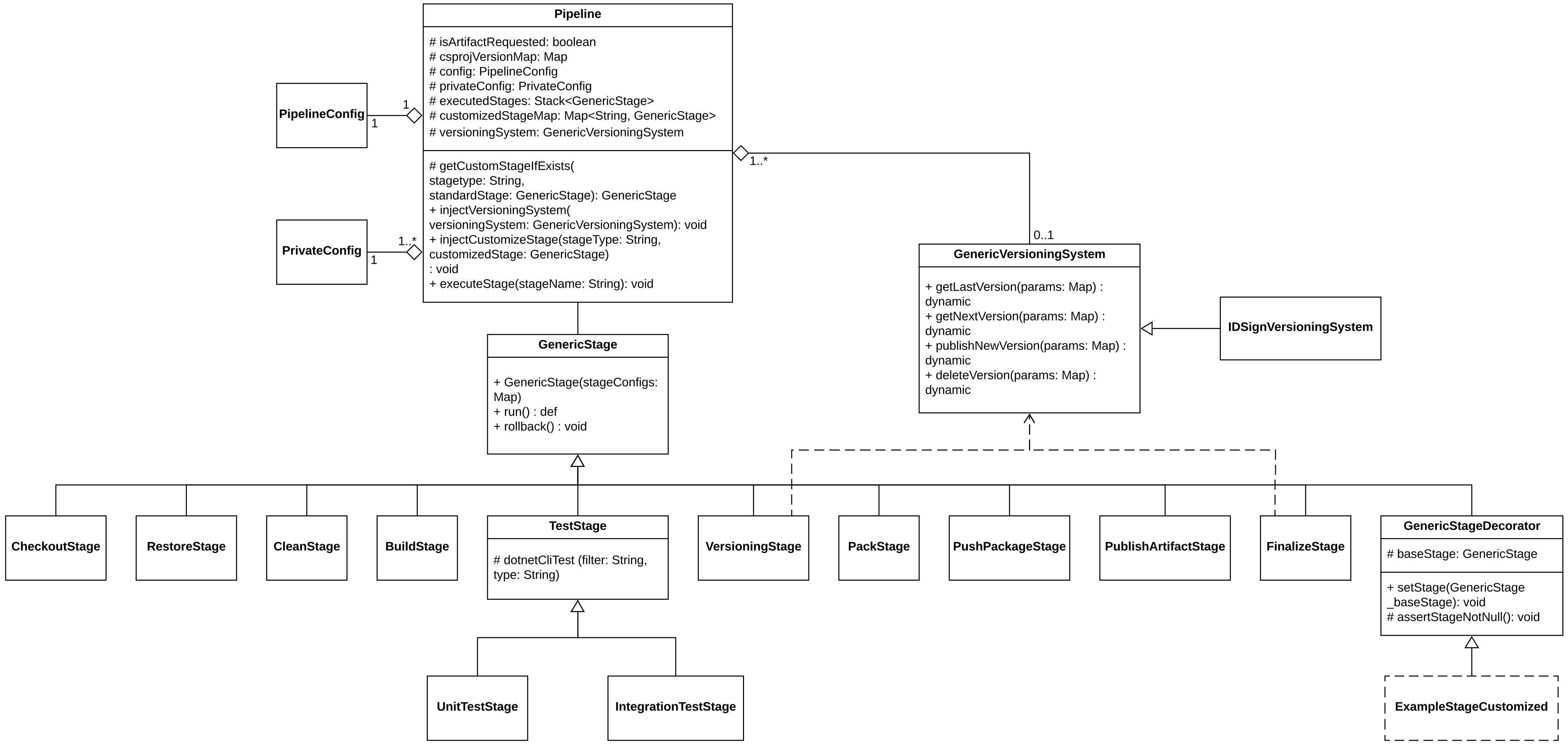
5.1 Shared Libraries

Jenkins provides a really convenient way to write code once and use it on every pipeline: shared libraries. They include modules organized on 2 subfolders, *vars* and *src*. *src* contains *.groovy* source files, added to classpath when executing pipelines, while *vars* folder hosts script files that are exposed as variables on *Jenkinsfile* (the only file that jenkins runs).

In my case I put most part of code inside *src* folder, so that I could design it as if it was a standard Java project. It contains all the stages, included the generic decorator for a customized stage, the *Pipeline* object, classes for pipeline configurations, utils and the versioning system. This last one is actually injected from outside, so that classes that depend on it reference the abstract class the versioning system depends on. On the other hand, *vars* folder includes scripts that help reading configuration files or utility for plugins, since only on scripts file "*this*" is a reference to a *Script* object, able to access plugin methods. That is why it has also been used to make Slack notifications and the creation of pipelines easier.

About utils classes, those that needs a reference to the *Script* object can be accessed through a sort of adapter that does nothing but instantiate the *Script* object in the static constructor and then expose its methods.

Shared libraries are versioned and downloaded from the git repository every time a build starts. They are configurable from *Manage Jenkins → Configure System → Global Pipeline Libraries.*


Pipeline
# isArtifactRequested: boolean
# csprojVersionMap: Map
# config: PipelineConfig
# privateConfig: PrivateConfig
# executedStages: Stack<GenericStage>
# customizedStageMap: Map<String, GenericStage>
# versioningSystem: GenericVersioningSystem
# getCustomStageIfExists(
stagetype: String,
standardStage: GenericStage): GenericStage
+ injectVersioningSystem(
versioningSystem: GenericVersioningSystem): void
+ injectCustomizeStage(stageType: String,
customizedStage: GenericStage)
: void
+ executeStage(stageName: String): void
PipelineConfig
1
1
PrivateConfig
1..*
1
1..*
0..1
GenericVersioningSystem
+ getLastVersion(params: Map) :
dynamic
+ getNextVersion(params: Map) :
dynamic
+ publishNewVersion(params: Map) :
dynamic
+ deleteVersion(params: Map) :
dynamic
IDSignVersioningSystem
GenericStage
+ GenericStage(stageConfigs:
Map)
+ run() : def
+ rollback() : void
CheckoutStage
RestoreStage
CleanStage
BuildStage
TestStage
# dotnetCliTest (filter: String,
type: String)
UnitTestStage
IntegrationTestStage
VersioningStage
PackStage
PushPackageStage
PublishArtifactStage
FinalizeStage
GenericStageDecorator
# baseStage: GenericStage
+ setStage(GenericStage
_baseStage): void
# assertStageNotNull(): void
ExampleStageCustomized

5.2 Jenkins Plugins

Below are listed the most important plugins I have used:

- *Artifactory*: to upload on Artifactory packages to deploy. Unfortunately, it does not support package deletion for example, therefore I had to integrate it with API calls.
- *Git Plugin*: used for pulling from git repo. Particularly useful is the possibility to perform a sparse checkout: instead of cloning the entire repository, if supported, only the requested file or folder is copied.
- *JUnit Plugin*: used to parse and read test reports and display results
- *MSTest Plugin*: unluckily .NET test results have the .trx file extension. This plugin converts them into .xml files, following the schema definition to be JUnit readable.
- *Pipeline Utility Steps*: it provides a lot of handy functions to parse json or zip folders.
- *Slack Notification Plugin*: I created a channel on our Slack workspace and added Jenkins CI app. After a short configuration phase through an integration token I managed to notify all members of these channel through messages like:

  ```
  Pipeline: ${env.JOB_NAME}
  Build: #${env.BUILD_NUMBER} on ${env.NODE_NAME}
  Commit of: ${authorOfLastCommit}
  Status: ${"Started"|"Failed"|"Success"|"Aborted"}
  Url: ${env.BUILD_URL}
  Additional Info: ${additional info like exception message}
  ```

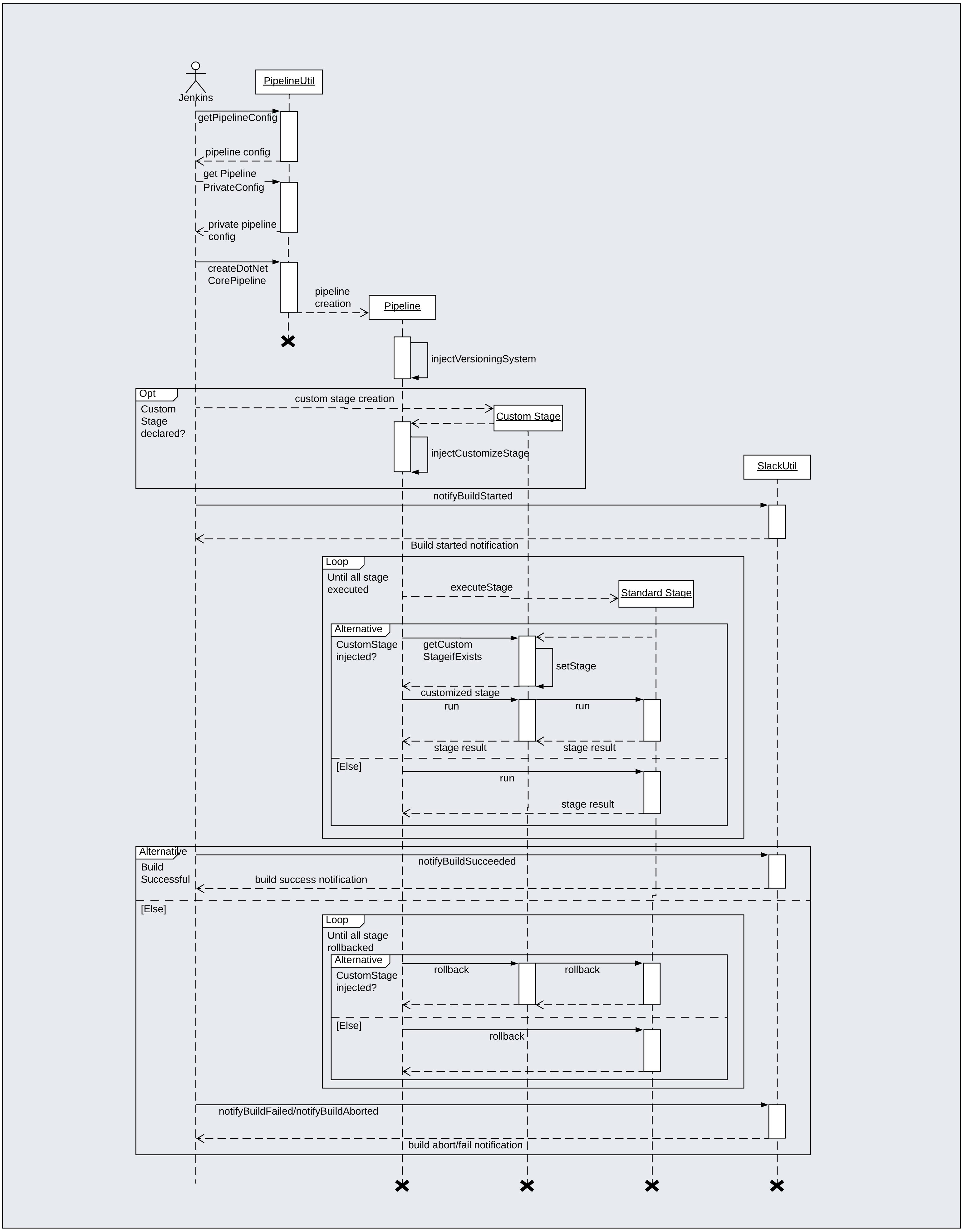
  Where in Commit of I put the author of the last commit, the Url field contains the url of notified build while in Additional info there is the exception message if the build failed.
- *Test Results Analyzer*: it shows the history of test execution results through line, pie and bar charts.

5.3 Pipeline Class and Stages

A pipeline object is responsible for calling a stage when user asks for it. It is built from configuration properties, credentials and a Script object to invoke plugins functions if necessary. Each stage shares a constructor accepting a Map, two methods: `run` containing the steps to execute and `rollback` that undo all the work for the current step if the build fails. Little by little that stages complete their work, they are added to a stack so that rollback procedures are popped from it in reverse order.

In order to keep a low level of complexity, I chose to create an abstract class from which all the stages inherit from. In the future, if I needed multiple implementations of the same stage, a possible alternative solution would be to use a Strategy Pattern keeping a unique pipeline but multiple interfaces as many as the number of stages. In this way I can choose the implementation of a single stage in the `Pipeline` class. What could not be missing is a way to extend a single stage if a project needs to carry out more complex operations. I opted for a Decorator pattern: new stages are declared on *Jenkinsfile* and “injected” in the current pipeline. After the initialization of the standard stage, the decorated stage is built, ready to use the inherited `run` method. Unfortunately, I could not extend old stages by inheritance because only the `Pipeline` object should create the next stage since it has all the needed informations stored from the results of previous stages.

Reflection would be the only other choice, storing the canonical name and then instantiate it through the `newInstance` method from a `Constructor` object. I decided to avoid this approach because in this case I would have been using Reflection to change a stage behavior that I could have not known at compile-time. This is not logically correct since I am aware of which is the implementation of a stage before running the *JenkinsFile*. I would have only simplified the design, avoiding to use the Decorator pattern, but reducing the flexibility I gain with it. A second drawback of Reflection, almost negligible in these circumstances, is its performance penalty. Stages certainly cannot be decorated at runtime, but it may be useful to furtherly decorate already customized stages.


Jenkins
PipelineUtil
getPipelineConfig
pipeline config
get Pipeline PrivateConfig
private pipeline config
createDotNet CorePipeline
pipeline creation
Pipeline
injectVersioningSystem
Opt
Custom Stage declared?
custom stage creation
Custom Stage
injectCustomizeStage
SlackUtil
notifyBuildStarted
Build started notification
Loop
Until all stage executed
executeStage
Standard Stage
Alternative
CustomStage injected?
getCustom StageifExists
setStage
customized stage
run
run
stage result
stage result
[Else]
run
stage result
Alternative
Build Successful
notifyBuildSucceeded
build success notification
[Else]
Loop
Until all stage rollbacked
Alternative
CustomStage injected?
rollback
rollback
[Else]
rollback
notifyBuildFailed/notifyBuildAborted
build abort/fail notification

5.4 Checkout, Restore, Clean and Build

First step after initialization is obviously the checkout of the project. The team I am working with never adopted any particular git workflow, so the *master* branch is pulled. Restore stage takes care of retrieving all dependencies. It is a company's convention to move in a private NuGet repository all the packages from which others could depend on. Each solution can have multiple projects and each one of them is described by a .csproj file. This file contains, among other items, references to packages:

```
<PackageReference Include="IDSign.Base.Service.Standard"
    Version="1.48.82" />
```

Our internal and public packages taken from the above XML elements are downloaded and placed in a cache under */root/.nuget/packages/*. When a project is built, the cached packages DLL are then placed in the bin folder. Upgrading the version of a DLL is a risky operation because it may introduce a bugfix that could trigger other unpleasant situations or even worse, a new functionality, maybe not properly tested that returns wrong results, leading to a successful but unstable build. For this reason it has been decided that, when such an operation is necessary, a new separate task must be created while the commit that upgrade the DLL version references it.

The clean stage only removes object and binary files in the possibility they slipped in with a commit.

During the build stage the project is compiled and output files (generally dll) are generated and placed in bin folder. For these last 2 operations and for the next, the *Configuration* property should be set to *Release*, since in *Debug* there are debugging symbols and the output folder grows in size.

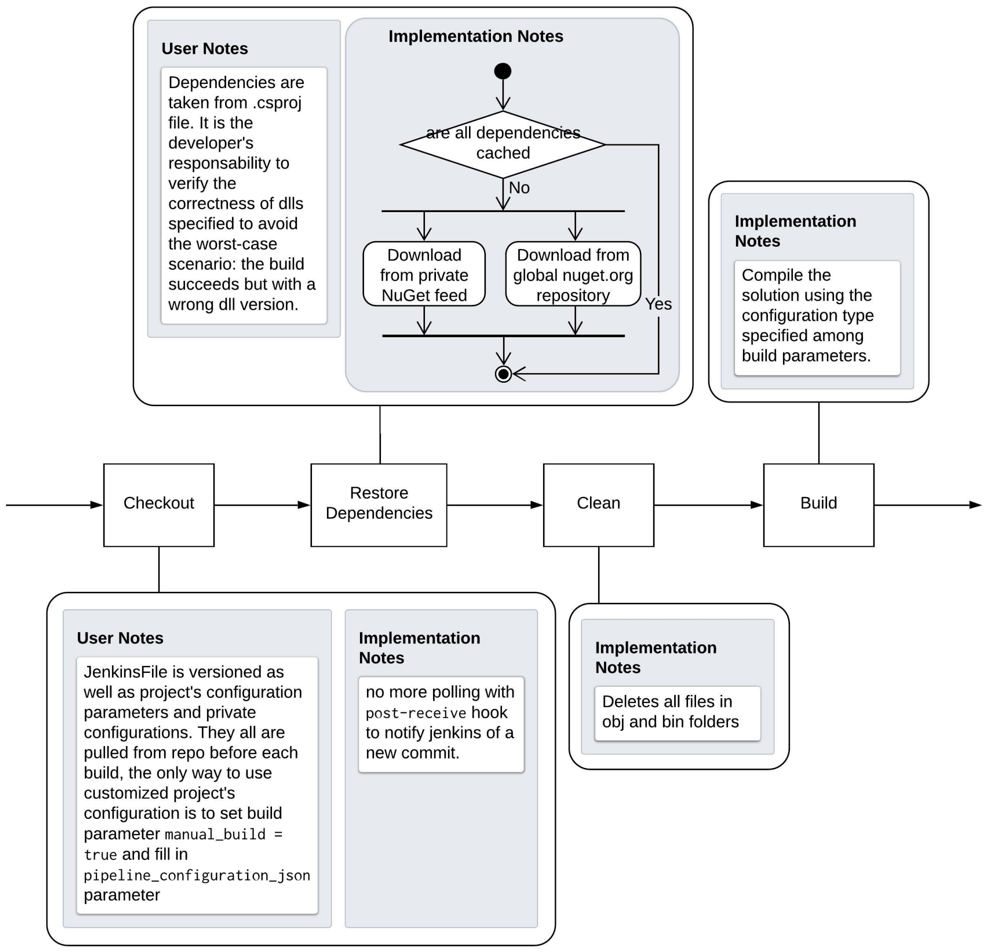

User Notes
Dependencies are taken from .csproj file. It is the developer's responsability to verify the correctness of dlls specified to avoid the worst-case scenario: the build succeeds but with a wrong dll version.
Implementation Notes
are all dependencies cached
No
Yes
Download from private NuGet feed
Download from global nuget.org repository
Implementation Notes
Compile the solution using the configuration type specified among build parameters.
Checkout
Restore Dependencies
Clean
Build
User Notes
JenkinsFile is versioned as well as project's configuration parameters and private configurations. They all are pulled from repo before each build, the only way to use customized project's configuration is to set build parameter manual_build = true and fill in pipeline_configuration_json parameter
Implementation Notes
no more polling with post-receive hook to notify jenkins of a new commit.
Implementation Notes
Deletes all files in obj and bin folders

Figure 5.1: Checkout, Restore, Clean and Build stages

5.5 Testing Stage

As for now only two types of tests are planned: **unit** and **integration** tests. In the eventuality my team is going to deal with bigger projects, a possible improvement would be to add even acceptance tests before the unit test step. Who writes tests has the duty to group his tests into the two categories just mentioned. Several ways are available to do this, but I chose two of them:

- Filtering by *classname*: if a class containing tests includes `IT_` in its name, then it is considered an integration test, otherwise a unit test. It would be better to check if it started with `IT_` but, as for now, it is not supported by dotnet CLI.

- Filtering by *test category*: methods and classes can have attributes on .NET, essentially the same term to indicate annotations in Java. If a method has a `TestCategory` attribute equals to *IntegrationTest*, then it is considered an integration test, otherwise a unit test.

But how to be sure that a developer did not confuse a unit test with an integration test? The only practicable solution I thought is using a completely isolated container where unit tests can be run. With "completely isolated" I mean that it is not even on the same network of the host, therefore unable to communicate with any network. This is achieved through the option `--network="none"` of the `run` command:

```
docker run --rm --network="none" -v
{nuget_cache_volume}:/root/.nuget/packages -v
{slave_data_volume}:/csproj mcr.microsoft.com/dotnet/core/sdk:2.2
/bin/bash -c "cd /csproj/{project_folder} && dotnet test
{project_name}.csproj -c Release ..."
```

It has to share the NuGet cache location with the slave, otherwise it would need to restore dependencies, that implies an internet connection. The correct differentiation of unit and integration tests is really important for two reasons:

- Reliability of metrics, like how much time tests take.

- The isolation of unit tests is an additional assurance for the developer that what he labeled as unit test is actually a unit test and does not involve other components but the module itself.

Test reports produced are cast into JUnit reports through MSTest. Integration test stage has an additional phase: postman tests. From Postman it is possible to test your api and its responses. All tested api collections must be put in *./NewmanTests/Collections* folder, so that the npm package *postman-combine-collections* can aggregate them into a single collection.

Another npm package, called Newman, then takes this resulting collection, calls the listed API and runs the corresponding tests, generating JUnit and CLI (Command Line Interface) reports. What if I needed to test on different environments, for example because an endpoint of my service changed? Newman command accepts, through *-e* parameter, a json describing the environment variables. This json file resides in *./NewmanTests/Env/{environment_name}.json*.

Code Coverage can be enabled through the option `--collect` with the command `dotnet test`. However, as for now, it is supported only on Windows.

5.6 Versioning Stage

This stage is essential for all the next ones because it returns a map containing the last and next version for each project. These properties are needed across stages, especially when pushing on NuGet and Artifactory repositories and for the creation of the release note file. The returned object:

```
[
    "isArtifactRequested": true,
    "projectVersionMap": [
```

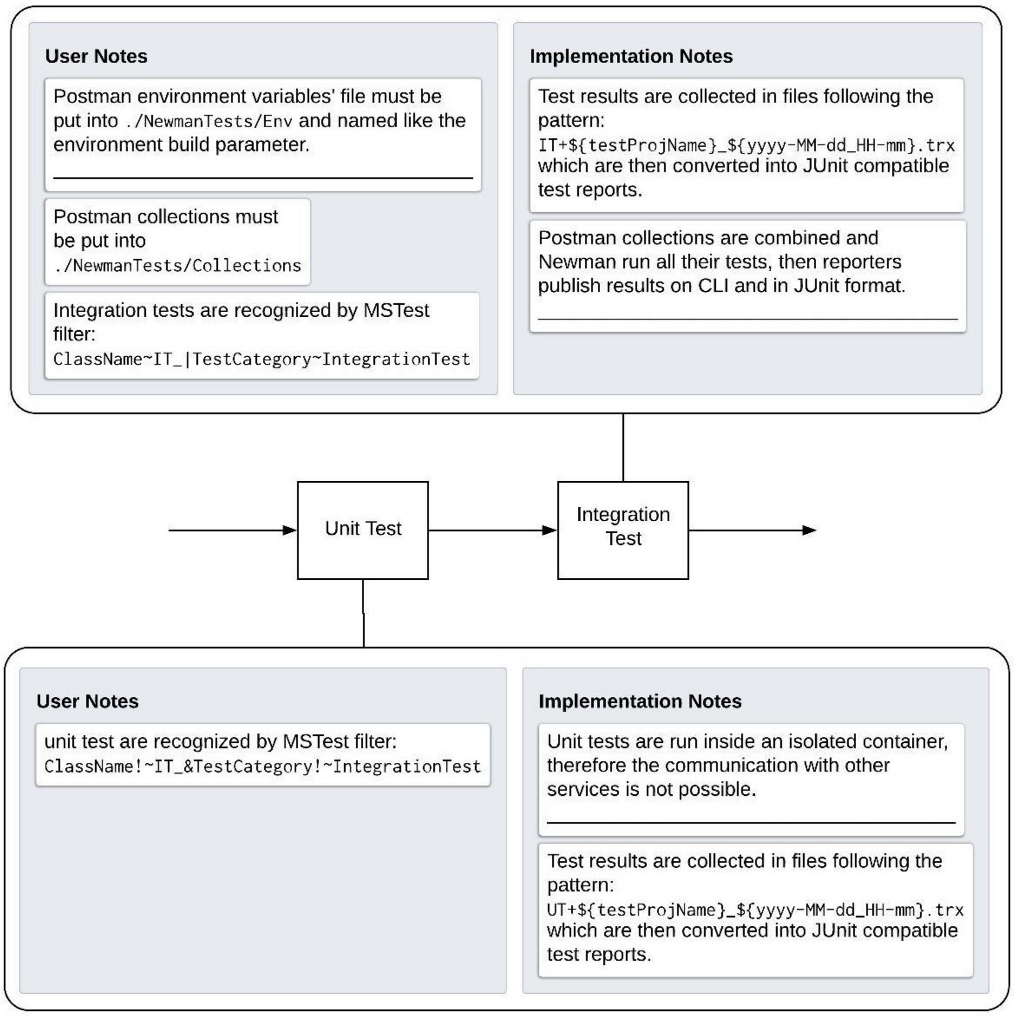

User Notes
Postman environment variables' file must be put into ./NewmanTests/Env and named like the environment build parameter.
Postman collections must be put into ./NewmanTests/Collections
Integration tests are recognized by MSTest filter: ClassName~IT_|TestCategory~IntegrationTest
Implementation Notes
Test results are collected in files following the pattern: IT+${testProjName}_${yyyy-MM-dd_HH-mm}.trx which are then converted into JUnit compatible test reports.
Postman collections are combined and Newman run all their tests, then reporters publish results on CLI and in JUnit format.
Unit Test
Integration Test
User Notes
unit test are recognized by MSTest filter: ClassName!~IT_&TestCategory!~IntegrationTest
Implementation Notes
Unit tests are run inside an isolated container, therefore the communication with other services is not possible.
Test results are collected in files following the pattern: UT+${testProjName}_${yyyy-MM-dd_HH-mm}.trx which are then converted into JUnit compatible test reports.

Figure 5.2: Unit and Integration test stages

```
        "proj1": [
            lastVersion: "1.0.0",
            nextVersion: "1.0.1"
        ],
        "proj2": [
            lastVersion: "1.0.0",
            nextVersion: "1.0.1"
        ],
        ...
    ]
]
```

`IsArtifactRequested` is evaluated to true only if the last commit message contains the *no-artifact* tag. As I already mentioned, each commit message must match a regex in order to be accepted, but user can also specify special tags inside curly brackets:

- *no-artifact*: `projectVersionMap` is empty and none of the following stages are executed. It is meant for situations where you do not need to deploy anything new, like when you are working on an uncompleted task or when you just need to run metrics on you code.
- *major*: incremementing major version is not possible unless one of the following 2 conditions are satisfied. The first one imposes this tag, while the second one is a build with *doesManually-IncrementMajor* configuration property set to true. The general rule is:

```
def checkMajor = commitMsgParsed.doesIncrementMajor
    || doesManuallyIncrementMajor;

def versionType = taskVersionType == SemanticVersionType.MAJOR ?
    (checkMajor ? SemanticVersionType.MAJOR :
        SemanticVersionType.MINOR) :
    taskVersionType;
```

If `checkMajor` fails, the next version is going to be a MINOR.

`lastVersion` could be null when versioning server does not have the entry referencing the current project yet, so it must be aligned with others if other projects have already been versioned. This delicate situation occurs when new projects must be added to versioning: It is mandatory that they were not versioned before, otherwise they cannot neither be aligned with already versioned ones. As a matter of fact, all projects are having the same `lastVersion` and `nextVersion` after this stage. Even if, as it is implemented, the forced alignment is kind of a contradiction with the existing logic, it is also the only way to keep track of versions using git tags, as better explained soon.

About the versioning system used, it inherits from `GenericVersioningSystem` class and, as I said, is injected in the Pipeline object immediately after its creation. It is not in the constructor for the same reason I may not want a new artifact. It implements both methods `getLastVersion` and `getNextVersion`. In particular this last one takes as parameters the `lastVersion` and "*Release Type*", which is evaluated by requesting from the project management portal the XML describing the task and obtaining the value through the following XPath query:

```
/issue/custom_fields/custom_field[@name='Release type']/value/text()
```

5.7 Pack and Push of utility Packages

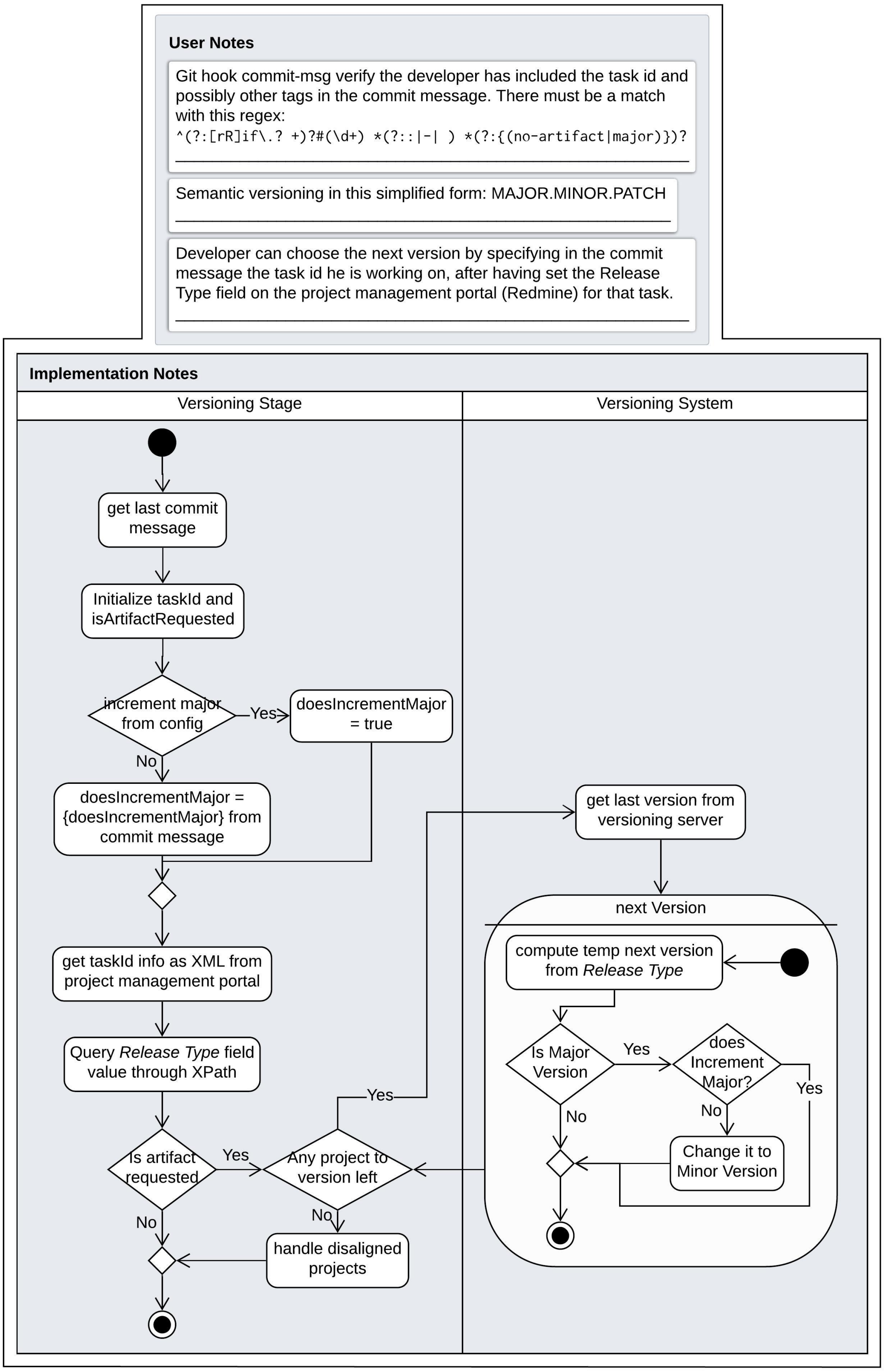
Pack stage creates a file with *.nupkg* extension, the only supported extension in a NuGet server. The new package is tagged with `-p:PackageVersion=nextVersion`. The older versions of the generated package are obviously not discarded. They all are accessible since the new version's changes might not be supported by older users of this library.

This .nupkg file simply contains the project's packaged source code, so it is usually quite light. `PushPackage` stage just takes these files and push them on NuGet private repository using a secret *ApiKey* found in the private configuration json. This is the first time a rollback is fundamental: without it, if the build fails, packages already pushed are staying there and prevent the success of the next build because it is not possible to overwrite existing packages (returning a `409 Conflict`), which means having the same version. Rollback procedure is a guarantee that they are going to be indeed deleted, not resulting in the described conflict.

5.8 Publishing Artifact

In this stage the deliverable is defined and moved on Artifactory. Artifactory is a repository manager that I, in accordance with the company, chose for:

- Designed to be integrated in a CI/CD ecosystem with an optimal scaling strategy.
- Powerful REST APIs.
- It is a repository manager on premise, in cloud or in a hybrid model.
- It published plugins for the most famous CI servers.
- With its premium subscription it manages other repository types, such as Maven, besides the binary one included in the free plan.


User Notes

Git hook commit-msg verify the developer has included the task id and possibly other tags in the commit message. There must be a match with this regex:

```
^(?:[rR]if\.? +)?#(\d+) *(?::|-| ) *(?:{(no-artifact|major)})?
```

Semantic versioning in this simplified form: MAJOR.MINOR.PATCH

Developer can choose the next version by specifying in the commit message the task id he is working on, after having set the Release Type field on the project management portal (Redmine) for that task.

Implementation Notes
Versioning Stage
Versioning System
get last commit message
Initialize taskId and isArtifactRequested
increment major from config
Yes
doesIncrementMajor = true
No
doesIncrementMajor = {doesIncrementMajor} from commit message
get taskId info as XML from project management portal
Query *Release Type* field value through XPath
Is artifact requested
Yes
Any project to version left
Yes
No
No
handle disaligned projects
get last version from versioning server
next Version
compute temp next version from *Release Type*
Is Major Version
Yes
does Increment Major?
Yes
No
No
Change it to Minor Version

Figure 5.3: Versioning stage

First of all, it is created for each project a bundle containing all DLLs, including the dependencies, and configuration files the project itself needs in order to run. This bundle is then zipped in a *dist*

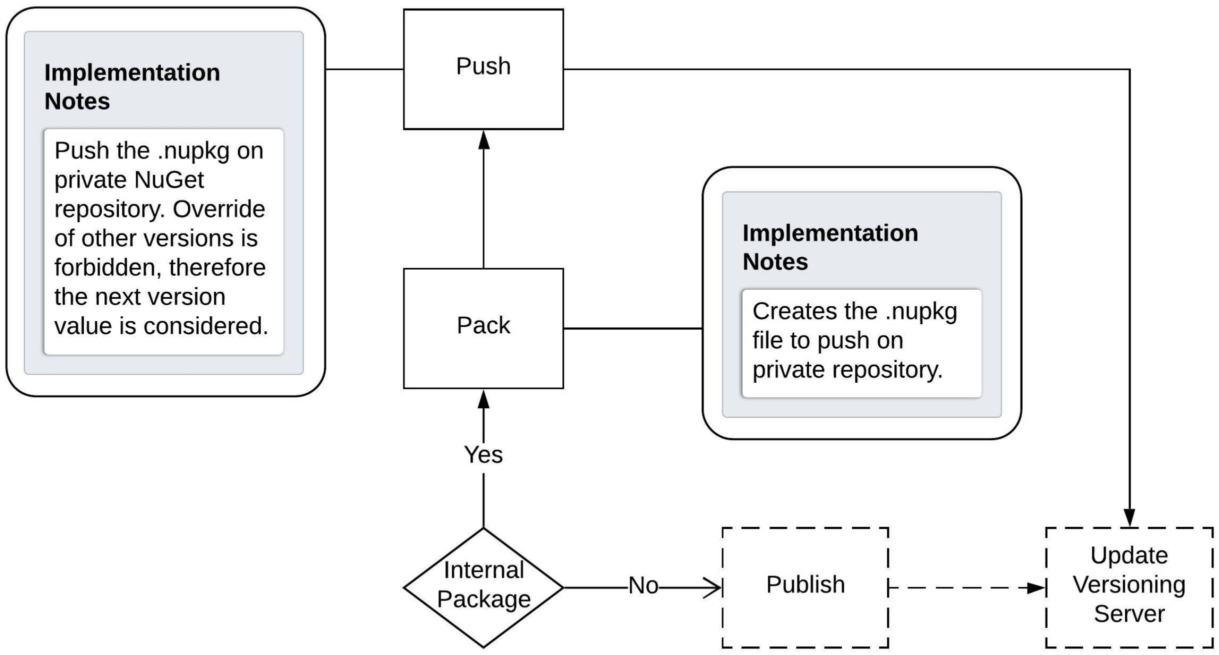

Implementation Notes
Push the .nupkg on private NuGet repository. Override of other versions is forbidden, therefore the next version value is considered.
Push
Implementation Notes
Creates the .nupkg file to push on private repository.
Pack
Yes
Internal Package
No
Publish
Update Versioning Server

Figure 5.4: Pack and Push stages

folder and named:

```
"${yyyy_MM_dd}_${csprojName}_${version}.zip"
```

First part is the current date, from years to days to ease sorting; then there is the project name and `nextVersion`. To move this zip on Artifactory, an upload specification json is needed, taking the zip and the path where it must be placed.

```
{
    "files": [
        {
            "pattern": "${zipFilePath}",
            "target": "${basePath}/${slnName}/${projectName}/"
        }
    ]
}
```

This json is then fed to the Artifactory plugin which, after been configured by setting the *Artifactory Server Id*, eventually uploads the deliverable.

As before and for the same reasons, a rollback strategy is necessary in this stage too. Unfortunately, Artifactory plugin does not let you delete file from the repository, leaving me with the only other option: calling Artifactory public API and delete it “manually”. Username and password are necessary for the API's authentication, so they are taken from the private configuration json (like I said, temporarily).

5.9 Update Versioning Server and Add Git Tag

If both push on NuGet and Artifactory are successful, next step is to update the versioning server with the current version (`nextVersion` basically). To do this, both `lastVersion` and `nextVersion` are considered, following these operations:

1. after each successful build, the most recent commit is tagged and named as the `nextVersion`.

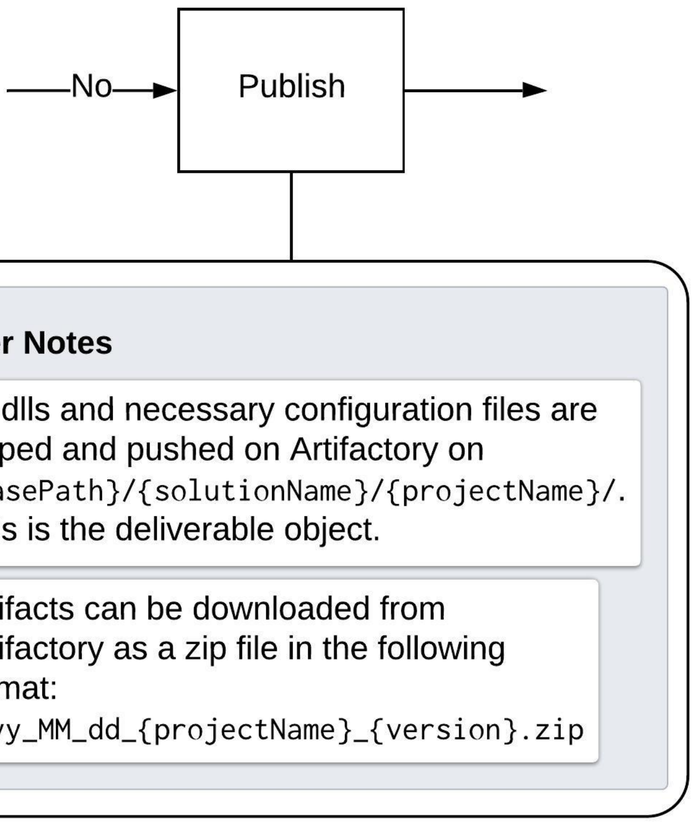

No
Publish
User Notes
All dlls and necessary configuration files are zipped and pushed on Artifactory on {basePath}/{solutionName}/{projectName}/. This is the deliverable object.
Artifacts can be downloaded from Artifactory as a zip file in the following format: yyyy_MM_dd_{projectName}_{version}.zip

Figure 5.5: Publish stage

2. To find out what changed from the old version to the current one, the hash of the tagged commit named as `lastVersion` is caught and used as starting point to collect all the following ones until the HEAD commit. For this step `git rev-list` gets to help. In the case when there is no commit tagged as `lastVersion`, meaning that it is null and therefore it is the first build with Jenkins, then the first commit is located always with `git rev-list` but with the option "`--max-parents=0`, obtaining the only commit without a parent, in other words the first commit.

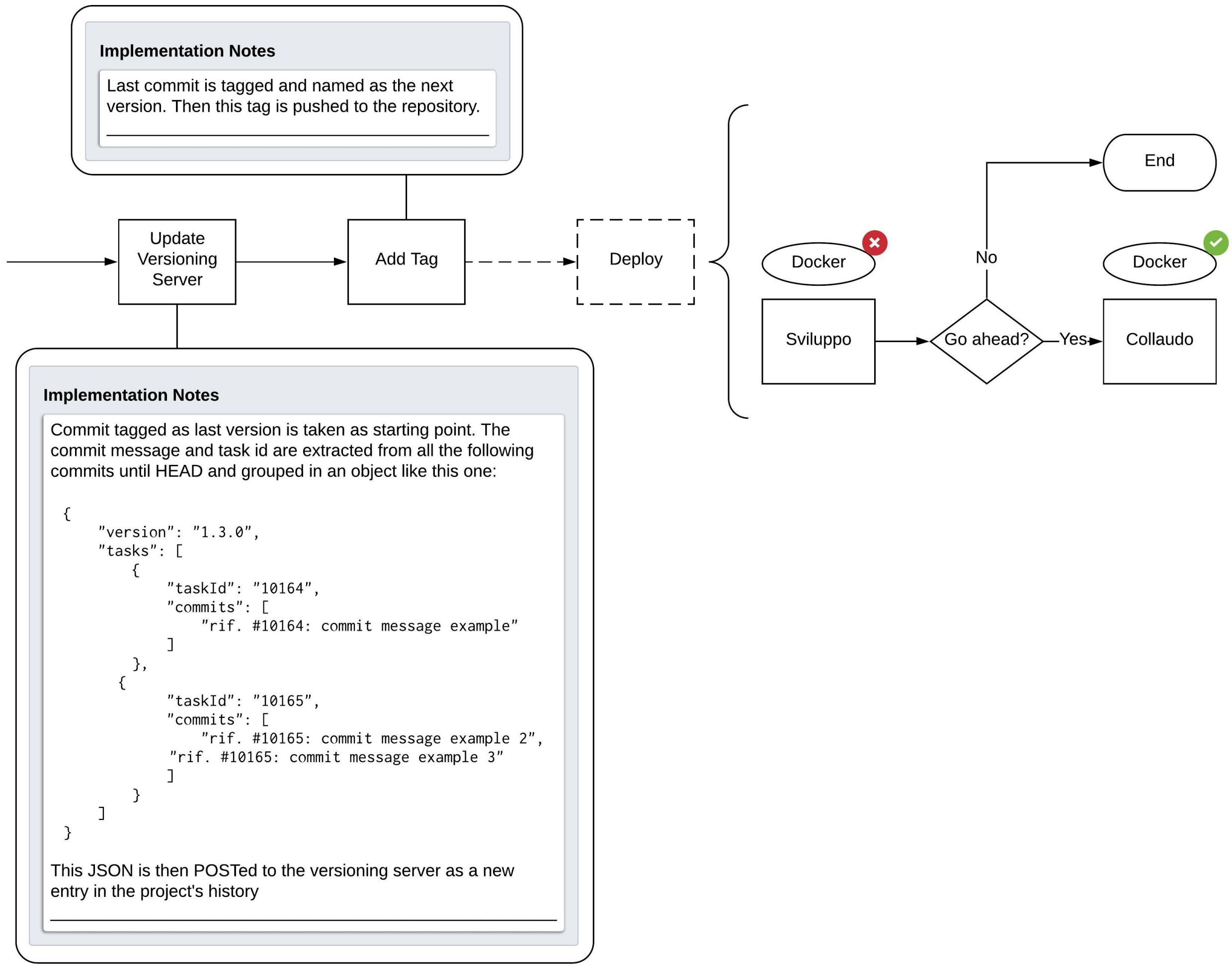
3. From each commit, the message is extracted with `git show --quiet --format='%s' ${commit}` and from each message the `taskId` is isolated and used as key in a map that groups all `taskIds` with their respective messages.

4. This map is then parsed as JSON and sent in the body of a POST request to the versioning server.

Now the last thing left is tagging the current commit. Git supports two types of tags: lightweight and annotated. Lightweight tags should be use for development purposes, essentially to note down important commits, whereas annotated tags can be easily pushed to the remote repository using `--follow-tags` option and this is why I chose the second type.

Again, a rollback strategy is necessary both for the versioning server update, to keep it aligned with Artifatory and NuGet, and to delete the possibly pushed tags, since git blocks duplicated tags.

Note: the versioning server is the core of the whole system, so finding a way to restore its state after a crash is a priority, therefore doing frequent backups would be a possible way to keep it aligned.



Last commit is tagged and named as the next version. Then this tag is pushed to the repository.

Commit tagged as last version is taken as starting point. The commit message and task id are extracted from all the following commits until HEAD and grouped in an object like this one:

```
{
    "version": "1.3.0",
    "tasks": [
        {
            "taskId": "10164",
            "commits": [
                "rif. #10164: commit message example"
            ]
        },
      {
            "taskId": "10165",
            "commits": [
                "rif. #10165: commit message example 2",
            "rif. #10165: commit message example 3"
            ]
        }
    ]
}
```

This JSON is then POSTed to the versioning server as a new entry in the project's history

Figure 5.6: Updating Versioning Server and Tagged last commit

6 Conclusions

To give a broader perspective, I have wanted to emphasize the actual changes after the integration with Jenkins, both in terms of time gained and in terms of quality of project's management. However, a necessary future development is the Continuous Delivery/Continuous Deployment integration.

6.1 What changed after CI?

Before CI, the release process was really confused: the standard was to create a package, namely a zip folder containing the new deliverable, including release notes, that explain how to complete the process, and database migrations, scripts that must be executed on the production database, if present. In practice, this model was rarely followed and in critical situations it happened that only modified files were taken and replaced on the customer environment.

One of the few advantages of adapting to the package-based release was the tracking of which package is actually installed on the customer, since each package contained the version number. The main problem is that this last version did not correspond to any commit on the git repository, but it was defined at user discretion, definitely an obstacle when dealing with compatibility issues.

After CI introduction, there are several noticeable advantages:

- A build is stable and can be released only if it has been run on Jenkins machine and it is successful there. Developers' workspaces do not count. Moreover, our implementation based on IaC with Docker makes the CI build environment reproducible and the only source of truth.
- Test metrics: history of single test results and code coverage (not supported yet) are like a health check for the project.
- Artifacts are now stored in a binary repository (Artifactory) and everyone can deploy on a different environment by just downloading the last (not necessarily) zipped artifact.
- Commit messages have to contain the task id as already described. It is immediate finding out the last person who worked on a bug fix or functionality.
- Changes from version to version are now accessible through a simple HTTP GET request to the versioning server. It returns an html page displaying the history of releases arranged by expandable lists of tasks and commit messages with the possibility to filter by version.

6.2 Timing and Numbers with and without CI

Even at this level, integrating with Jenkins could significantly shorten the software release time. Before CI, the release of a single microservice could take from 4 up to 8 hours: this time segment includes the testing phase and the creation of the package too, which are covered and automated using Jenkins.

Regarding the percentage of Jenkins-integrated projects, the expectations are that, by the end of the year, at least 70% of the projects are going to migrate. The following achievement is the 100% of the company's main product, namely IDSign, that is going to be integrated with Jenkins at least as for its core. Instead, it is preferred to manage customer ad-hoc services and plugins separately.

6.3 Future Developments

Reached this point, my project certainly cannot be considered complete:

- Until now only one pipeline has been completely implemented, namely the one for .NET Core projects. However, the Java team too may find a CI process useful and even Mobile team needs a hand to automate those steps that are still launched by hand.
- Continuous Integration without Continuous Delivery means exploiting only half of its real potential. It is enough to take the issue of managing configurations for each environment: automate this part makes a difference. Not to mention the gain in time and quality of a process that replaces the human being on these stressful activities.
- As I said before, projects' versions are always aligned. This implementation should not be interpreted as a real issue because in this way it is clear the last release of a "solution", meanwhile it is easy to build the history of changes thanks to git tag. In few words, even if a project has not been touched, its version is increased, which is logically incorrect and could take to more software updates than necessary. As it is now, it has room for improvement even if it definitely means adding a remarkable level of complexity.
- Even if the version of untouched and independent projects is incremented, that does not imply they have to be rebuilt during each Jenkins build, as it is happening. Lot of time could be saved if they were not built and their test results processed.
- Full integration of Docker in the pipeline, like scaling Jenkins executors through Docker based on the number of planned jobs.

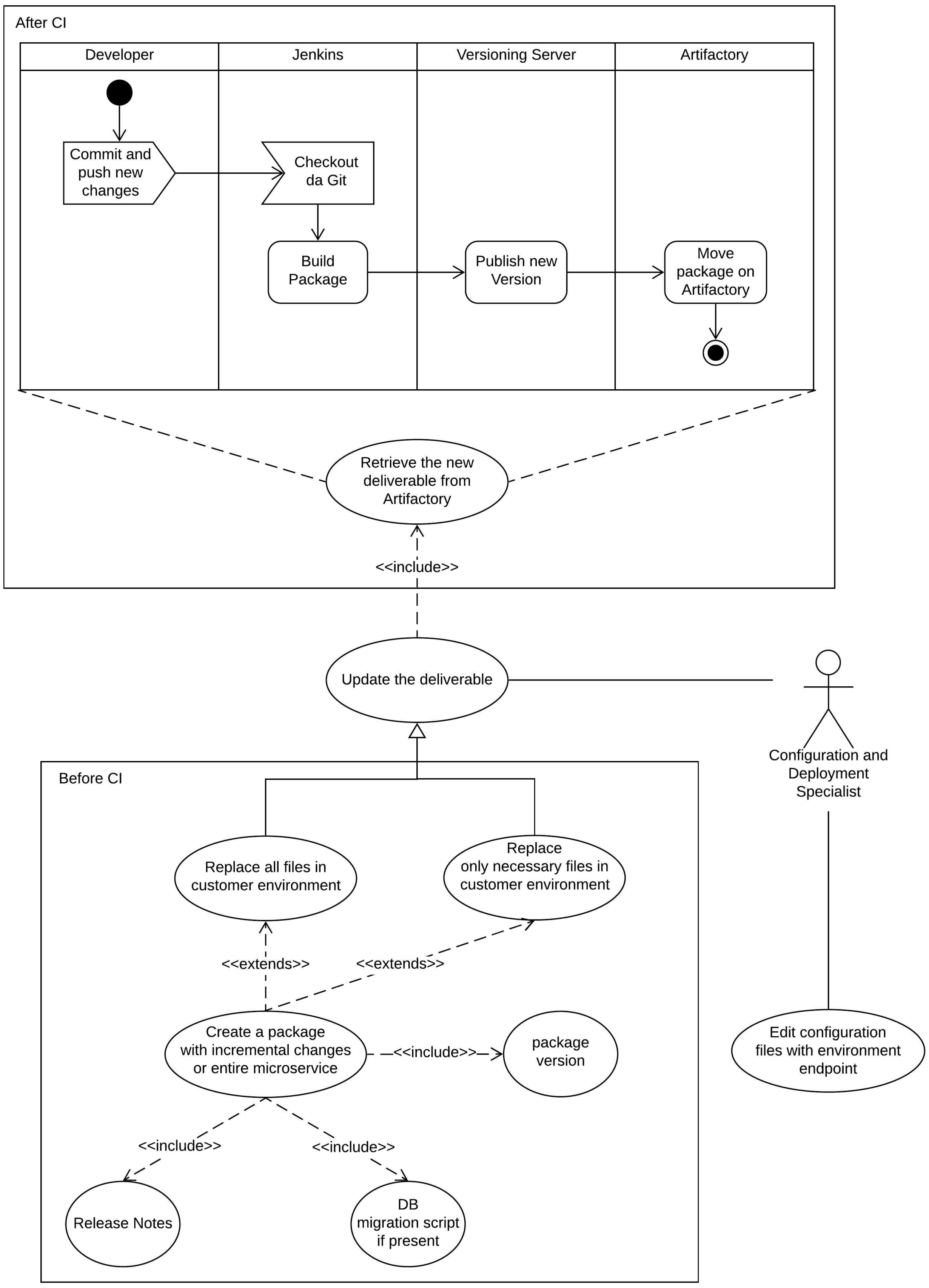

After CI
Developer
Jenkins
Versioning Server
Artifactory
Commit and push new changes
Checkout da Git
Build Package
Publish new Version
Move package on Artifactory
Retrieve the new deliverable from Artifactory
<<include>>
Update the deliverable
Configuration and Deployment Specialist
Before CI
Replace all files in customer environment
Replace only necessary files in customer environment
<<extends>>
<<extends>>
Create a package with incremental changes or entire microservice
<<include>>
package version
<<include>>
<<include>>
Release Notes
DB migration script if present
Edit configuration files with environment endpoint

Figure 6.1: Software Release before CI (and what it covered)

Bibliography

[1] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2nd edition edition, 2004.

[2] OWASP Foundation. A2-broken authentication. https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A2-Broken_Authentication. Last visit on 02/03/20.

[3] Martin Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, May 2006.

[4] Jenkins. Architecting for scale. https://jenkins.io/doc/book/architecting-for-scale#calculating-how-many-jobs-masters-and-executors-are-needed. Last visit on 11/02/20.

[5] Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org/. Last visit on 11/02/20.

Attachment A Dockerfile for Jenkins Slave

Here the *Dockerfile* for a Jenkins slave built on .NET Core image:

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2

# Install OpenJDK-8
RUN apt-get update && \
    apt-get install -y openjdk-8-jdk && \
    apt-get install -y ant && \
    apt-get clean;

# Fix certificate issues
RUN apt-get update && \
    apt-get install ca-certificates-java && \
    apt-get clean && \
    update-ca-certificates -f;

# Setup JAVA_HOME -- useful for docker commandline
ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64/
RUN export JAVA_HOME

RUN apt-get -y update && \
        apt-get -y install curl

# read here: https://pagure.io/fork/simo/gssntlmssp
RUN apt-get -y install gss-ntlmssp && \
        touch /etc/gss/mech.d/credentials && \
        chmod 777 /etc/gss/mech.d/credentials && \
        echo :user:pwd > /etc/gss/mech.d/credentials

ENV NTLM_USER_FILE /etc/gss/mech.d/credentials
RUN export NTLM_USER_FILE

RUN apt-get -y install curl software-properties-common && \
        curl -sL https://deb.nodesource.com/setup_12.x | \
        bash - && \
        apt-get -y install nodejs

RUN npm install -g newman
RUN npm install -g postman-combine-collections

# to run xmllint
RUN apt-get -y install libxml2-utils
```

```
# docker support
RUN apt−get −y update && \
        apt−get −y install apt−transport−https ca−certificates curl gnupg2 software−
        curl −fsSL https://download.docker.com/linux/debian/gpg | apt−key add − && \
        add−apt−repository "deb [arch=amd64] https://download.docker.com/linux/debian
        apt−get −y update && \
        apt−get −y install docker−ce−cli

COPY /install_files/agent.jar /agent.jar

COPY /install_files/NuGet.Config /root/.config/NuGet/

ENTRYPOINT ["java", "−jar", "/agent.jar"]
CMD ["−jnlpUrl",
    "http://host.docker.internal:8080/computer/unix00/slave−agent.jnlp",
    "−secret", "yoursecret", "−workDir", "'/home/jenkins'"]
```

Attachment B Source Code

All projects and documentation are open-source and uploaded on GitHub:

- Dockerfiles, docker-compose.yml and powershell scripts as utilities for docker operations: `https://github.com/marco-luzzara/Jenkins-CI-Docker`
- Install script for .NET Core projects: `https://github.com/marco-luzzara/DotNetCore-Jenkins-Inst`
- Generic pipeline and its json configurations: `https://github.com/marco-luzzara/DotNetCore-Jenkin`
- Web server for Versioning, including a Postman collection and OpenApi documentation: `https://github.com/marco-luzzara/project-versions-history`
- Jenkins Shared Libraries: `https://github.com/marco-luzzara/DotNetCore-Jenkins-shared-librar`