

CloudBees Jenkins Platform: Certification Training

1 - Anatomy of an application

Table of Contents

Lab Introduction	1
1 - The Big Picture: Anatomy of an application	1
Journey Description	1
Exercise: Getting the Code	2
Accessing the WebIDE	2
Accessing Git Server	3
Getting the code in the WebIDE	6
Exercise: Building and Testing the Application	9
Accessing the "Devbox"	9
Build the application	11
Exercise: Running the Application	13
Running "on the metal"	13
Running with Docker	16
Exercise: Updating the Application	17
Changing Code	17
Validating the Change	22
Cleaning Environment	22
Exercise: Defining a CD pipeline	23
Thinking about a CD Pipeline	23
Deployment Environment	23
User Profiles	24
Proposed Pipeline	25
Journey Summary	26

Lab Introduction

Welcome to your CloudBees Lab Exercises.

This workbook supplements your CloudBees University training. It contains a sequence of lab exercises to help prepare you for the [Jenkins Certification Exam](#).

The Lab exercises allow you to practice the concepts presented in the session slide decks.

Before running any of the Lab exercises:

- ✚ If you are running the self-paced version of the training, without an instructor, please follow the "Install Lab Environment" lesson, to ensure that all requirements are met, and that your Lab VM instance is up and running.
- ✚ If you are running the training with an instructor, please carefully follow the instructions you are given to bootstrap and/or access your "Lab Environment".

The next steps assume that you followed those instructions and have full access to the "Lab Environment" services.

1 - The Big Picture: Anatomy of an application

Journey Description

This lab won't cover any Jenkins cases. Instead, the application used for the whole Lab will be covered.

High level challenges will be addressed, to easily map their resolutions to Jenkins capabilities in next labs.

The target application is a simple Hello World:

- ✚ Written in [Java](#)
- ✚ Using the [Dropwizard Framework](#)
- ✚ Compilation chain is [Apache's Maven](#)
- ✚ Target runtime is a [Docker](#) container

IMPORTANT

Do not be scared if you do not know any of the technologies used here. The current Lab is here to let you understand before jumping in Jenkins.

We will cover those steps:

- ✚ Getting the code of the application
- ✚ Building and testing the application
- ✚ Running the application
- ✚ Editing code to change something and see that change in action
- ✚ Defining a "high-level" pipeline that describes this application delivery
- ✚ Setting up security to restrict access to the application source code

¥ Applying a change using a "Pull Request", to validate security

Exercise: Getting the Code

The goal of this exercise is to use the integrated *WebIDE* to fetch the code, from the integrated *git* server.

Accessing the WebIDE

¥ First step is to access the WebIDE of your instance:

! From the instance homepage that list service (<http://localhost:5000/index.html>), click on the WebIDE link

TIP Direct link to WebIDE: <http://localhost:5000/webide>

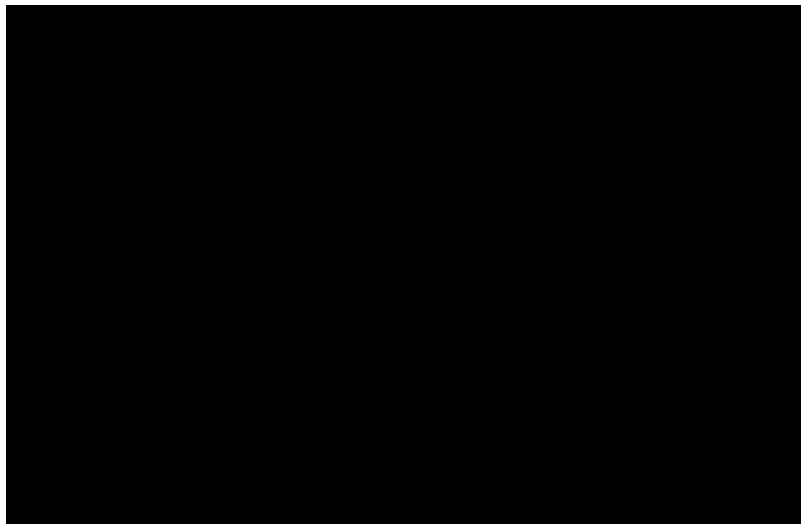


Figure 1. Codiad Login Page

¥ Then, you have to authenticate against this service. This is required to be able to simulate different use cases.

! Username and password are the same : butler

Figure 2. WebIDE Default page

Accessing Git Server

🔗 Next step is to access the git server of your instance in a new tab:

! From the instance homepage that lists services (<http://localhost:5000/index.html>), click on the GitServer link

TIP Direct link to GitServer: <http://localhost:5000/gitserver>

Figure 3. Gogs (Git server) Homepage

¥ Then, browse to the repository:

- ! Use the menu item Explore (top banner)
- ! Select the repository named "butler/demoapp"

Figure 4. Accessing the butler/demoapp repository

¥ You can then browse the project from this page:

- ! Code browsing and viewing
- ! Issue tracking & Pull Requests
- ! Code History with Commits tab
- ! And a lot more !

¥ We have to fetch the code address. This may be required numerous times in the next exercises:

- ! Use the right section
- ! Ensure you are using HTTP (blued, SSH is greyed)
- ! Click the the "COPY button":

Figure 5. Fetching the repository HTTPS Clone URL

Getting the code in the WebIDE

🔗 Going back to the WebIDE, give a focus on the Projects section

! From there, click the "+" button to *Create a new project*:

Figure 6. Create a new project

✚ On the New Project form, use the following settings to import the Git repository:

! Project Name: demoapp

! Folder Name: demoapp (this is the file system directory name)

! Click on the "from Git Repository" button and specify:

" Git Repository: The URL of the git repository you pasted previously from Gogs.

TIP	This URL should be http://localhost:5000/gitserver/butler/demoapp.git
-----	--

" Branch: Let the default value `master`

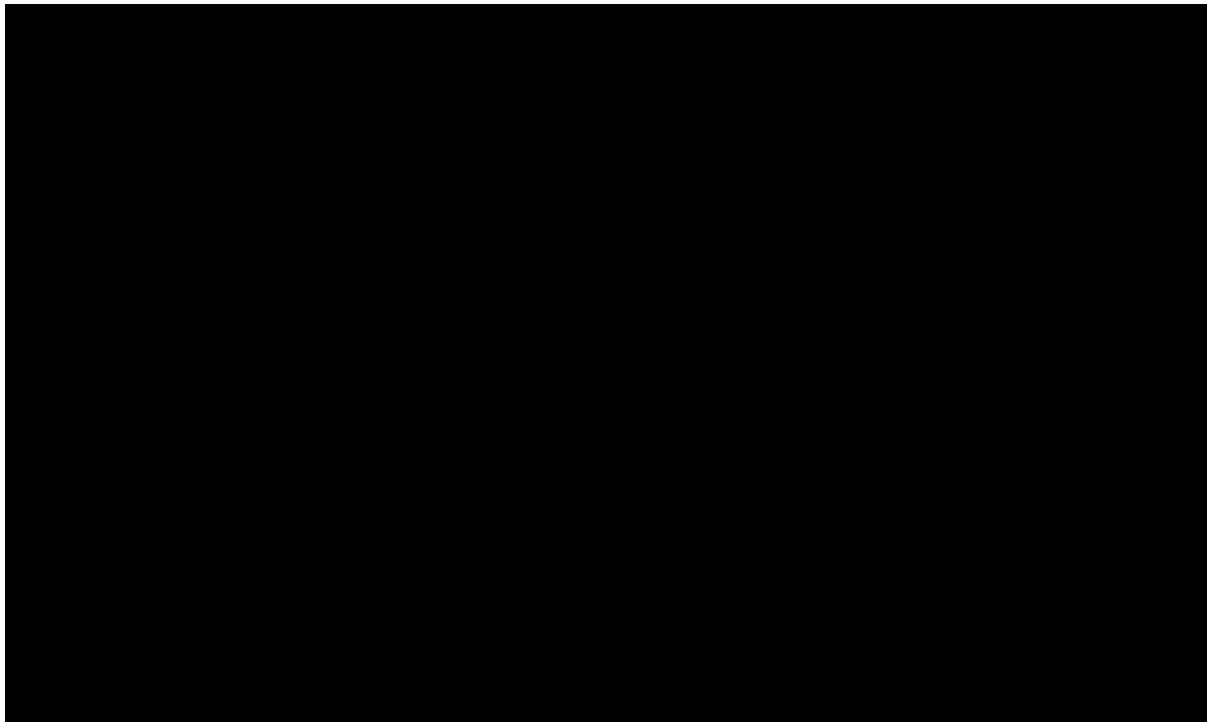


Figure 7. Loading Demo Application from Git

✚ Once validated:

- ! The new project should appear in the Projects section, next to Default, and set as the *Active* project
- ! The repository should be cloned in the WebIDE and be browsable

TIP	Try to browse the project and explore options to master the WebIDE
-----	--

Figure 8. A loaded Demo Project in the WebIDE

And here we are. This is the end of the exercise

Exercise: Building and Testing the Application

The goals of this exercise are:

- ¥ Familiarize yourself with the integrated command line environment (named Devbox)
- ¥ Learn how to build the application using Maven

Accessing the "Devbox"

- ¥ First step is to access the Devbox of your instance:

! From the instance homepage listing all services (<http://localhost:5000/index.html>), click on the Devbox link

TIP Direct link: <http://localhost:5000/devbox>

- ¥ You will see black screen with a white command prompt `c1oudbees-devbox $`. This is a Web-based Command Line Terminal, running inside the Lab machine.

Figure 9. Devbox Default Prompt

⚡ Unless specified differently, all command line have to be typed in the Devbox web terminal.

TIP | You can open multiple web-browser tabs or windows to have different command lines.

⚡ Try the following Linux/Unix commands:

! Print the current user you are impersonating inside the Devbox:

```
cloudbees-devbox $ whoami
```

! Current folder is `/workspace`: this is a shared folder with your IDE

```
cloudbees-devbox $ pwd
```

! Browse to the `demoapp` project and check content:

```
cloudbees-devbox $ cd demoapp
cloudbees-devbox $ ls -l
```

⚡ Check the tools pre-installed:

! Java

```
cloudbees-devbox $ java -version
```

! Git

```
cloudbees-devbox $ git --version
```

! Maven

```
cloudbees-devbox $ mvn -version
```

! Docker

```
cloudbees-devbox $ docker -v  
cloudbees-devbox $ docker info
```

Build the application

¥ The application needs to be built using Maven.

¥ Central entry-points for Maven are:

! The Maven command line: `mvn`

! The Maven CLI will parse the file `pom.xml` that describes the project to build

TIP | POM stands for Project Object Modeling

¥ Maven implements the concept of build steps, named "Goals"

! A set of predefined goals are already defined and bound by convention to some actions, depending on the `pom.xml` content

! `Pom.xml` can also set up some triggered actions before or after the standards goals

TIP | We are going to use only standard Goals

! The `pom.xml` also describes the application dependencies. Maven will download them and make them available to the build chain.

" Maven stores, by default, its dependencies in `~/ .m2/repository` folder

¥ Start by building the application with this command, from the root of the `demoapp` folder:

```
cloudbees-devbox $ cd /workspace/demoapp  
cloudbees-devbox $ mvn clean install  
...
```

! You should end on a "Build Success" ascii banner if everything runs smoothly:

```
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] -----
[INFO] Total time: 01:38 min
[INFO] Finished at: 2016-08-31T14:07:38+00:00
[INFO] Final Memory: 62M/269M
[INFO] -----
```

! The application will be generated in the `target` folder, next to `pom.xml`.

" This folder contains compiled classes, intermediate generated artifacts and the packaged application (a `jar` file in our use case.)

" See the [next chapter](#) for more details about packaged applications.

¥ You can then try different (explained) goals:

! Cleaning the generated artifacts (`target` folder):

```
cloudbees-devbox $ mvn clean
```

! Compiling the application (classes and intermediate artifacts):

```
cloudbees-devbox $ mvn compile
```

! Run the Unit Tests of the application:

```
cloudbees-devbox $ mvn test
```

TIP This goal will run `compile` goal, which is an "implied" goal, even if you don't specify it.

" Takes time to read the test-related output:

```
Results :
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

! Package the compiled classes in an (or multiple) artifacts

```
cloudbees-devbox $ mvn package
```

! Run the Integration Tests of the application (slow):

```
cloudbees-devbox $ mvn verify
```

WARNING

The tests take more than 30s to run and will do also a basic "Smoke Test" by starting the application.

! Installing the application in the target folder:

```
cloudbees-devbox $ mvn install
```

TIP

"Installing" the application means copying the artifacts in the local Maven repository (~/.m2), if any other Maven projects have a dependency on this one

IMPORTANT

Takes time to run all those targets, to understand constraints for each.

That's all for this exercise !

Exercise: Running the Application

The generated application will be a standalone jar file.

TIP

jar stands for "Java Archive". It's a ZIP archived, with a ".jar" extension, that contains Java compiled classes and metadata

This jar-file already contains an application server appliance, so you do not need a Java application server like Tomcat or Jetty.

TIP

This is also known as an "Uber-JAR"

We are going to run the application with 2 methods:

- ¥ Running "on the metal": Launch the embedded application server "as-it".
- ¥ Running "with Docker": build and run a Docker image to run the application

Running "on the metal"

We will launch the application "as is" inside the Devbox. We won't be able to reach it using the web-browser, given the lab instance architecture. We'll use command line instead.

IMPORTANT

The goal there is to understand how Integration and Smoke tests are behaving on an headless server.

- ¥ The [DropWizard Documentation](#) states that you have to call the java binary, providing:

- ! The path to the jar file to the flag '-jar'
- ! The keyword "server" as first argument

! The path to a DropWizard YAML configuration file

¥ The resulting command is:

```
cloudbees-devbox $ java -jar ./target/demoapp.jar server ./hello-world.yml
```

¥ Once started, it should print the log interactively on the command line.

! Take time to read the latest lines:

```
...
INFO [2016-08-31 16:38:12,271] org.eclipse.jetty.server.ServerConnector:
Started application@674658f7{HTTP/1.1}{0.0.0.0:8080}
INFO [2016-08-31 16:38:12,272] org.eclipse.jetty.server.ServerConnector:
Started admin@5c8eee0f{HTTP/1.1}{0.0.0.0:8081}
INFO [2016-08-31 16:38:12,272] org.eclipse.jetty.server.Server: Started
@4018ms
```

! The application is listening on the port 8080, and the administration panel of DropWizard on 8081 of the Devbox

IMPORTANT

Those ports are private: you cannot reach the application using your web browser YET.

¥ Validate this by opening in a new tab a 2nd Devbox command line, and use the curl command:

TIP

The `-v` flag of curl will print HTTP headers of both response and request, before printing the content


```

cloudbees-devbox $ curl -v http://127.0.0.1:8080/
*   Trying ::1...
* Connected to 127.0.0.1 (::1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.50.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Wed, 31 Aug 2016 16:44:37 GMT
< Last-Modified: Wed, 31 Aug 2016 14:07:30 GMT
< Content-Type: text/html; charset=UTF-8
< Vary: Accept-Encoding
< ETag: "8f2181178186417ce600f7d5716bb124"
< Content-Length: 345
<
<!--doctype html-->
<html ng-app>
<head>
  <title>Hello Butler</title>
  <script src="angular.min.js"></script>
  <script src="hello.js"></script>
</head>

<body>
  <div ng-controller="Hello">
    <p>The ID is {{greeting.id}}</p>
    <p>The content is {{greeting.content}}</p>
  </div>

  
</body>
</html>
* Connection #0 to host 127.0.0.1 left intact

```

¥ Back on the first terminal, you should be able to see the access log of the application:

```

...
0:0:0:0:0:0:0:1 - - [31/Aug/2016:16:44:37 +0000] "GET / HTTP/1.1" 200 345 "-"
"curl/7.50.1" 5
...

```

¥ Finally, terminate the application using the CTRL-C combination with your keyboard: it will send a SIGTERM signal to stop the application:

```

...
^C
INFO [2016-08-31 16:47:38,233] org.eclipse.jetty.server.ServerConnector:
Stopped application@674658f7{HTTP/1.1}{0.0.0.0:8080}
INFO [2016-08-31 16:47:38,236] org.eclipse.jetty.server.ServerConnector:
Stopped admin@5c8eee0f{HTTP/1.1}{0.0.0.0:8081}
INFO [2016-08-31 16:47:38,237]
org.eclipse.jetty.server.handler.ContextHandler: Stopped
i.d.j.MutableServletContextHandler@66e889df{/,null,UNAVAILABLE}
INFO [2016-08-31 16:47:38,239]
org.eclipse.jetty.server.handler.ContextHandler: Stopped
i.d.j.MutableServletContextHandler@383790cf{/,null,UNAVAILABLE}

```

Running with Docker

Running the application using Docker will allow:

- ✚ Better portability of the application: we are going to provide everything needed (outside Docker of course) to run the application everywhere
- ✚ Given the lab's architecture, which uses Docker itself, it will allow us to check the application using a web-browser

TIP We'll just cover the needed parts. For more Docker information, please read the Docker documentation [here](#) !

The "Docker" workflow will be as follows:

- ✚ Using a `Dockerfile`, we are going to build a "Docker image" for the application
 - ! A `Dockerfile` is a text file, which serves as a "recipe"
 - ! A "Docker image" is an immutable template
- ✚ Then, we are going to run a "Docker container", based on the image
 - ! Docker will manage the port forwarding to let us access the application externally

TIP A `Dockerfile` is already present at the root of the repository? Let's use it !

- ✚ Let's run the command line below, from the `demoapp` directory, to build and run our application docker container:
 - ! `-d` flag is for "Daemonize", aka. "Running in background"
 - ! `-p 30000:8080` flag makes Docker to publish container's private 8080 port to the external interface's public 30000 port.

TIP A long string will be printed by the command `docker run`. It will be the "Unique ID" (UUID) of the launched container. It is not needed for the exercise, but you can use it for any docker command. You can try it with `docker logs <UUID>`.

```

cloudbees-devbox $ docker build -t "demo-app:1.0.0" ./
...
Successfully built 8e7e2115adb8
cloudbees-devbox $ docker run -d --name demoapp-1 -p 30000:8080 demo-
app:1.0.0

```

¥ Now, open the application on your web browser at this URL: <http://localhost:30000>

TIP	Reload the page to check what is changing on the application page
-----	---

Figure 10. Demo App Main Page

That's all for this exercise !

Exercise: Updating the Application

The goal of this exercise is to change something in the application, and then rebuild it and run it again to see the change.

Changing Code

¥ Using the WebIDE:

- ! Open the file `hello-world.yml`, located at the root of `demoapp`
- ! Edit it, by changing the value associated with the key `defaultName` by something else:

Figure 11. Editing hello-world.yml

✚ Save the change, using your favorite keystroke, or the right panel:

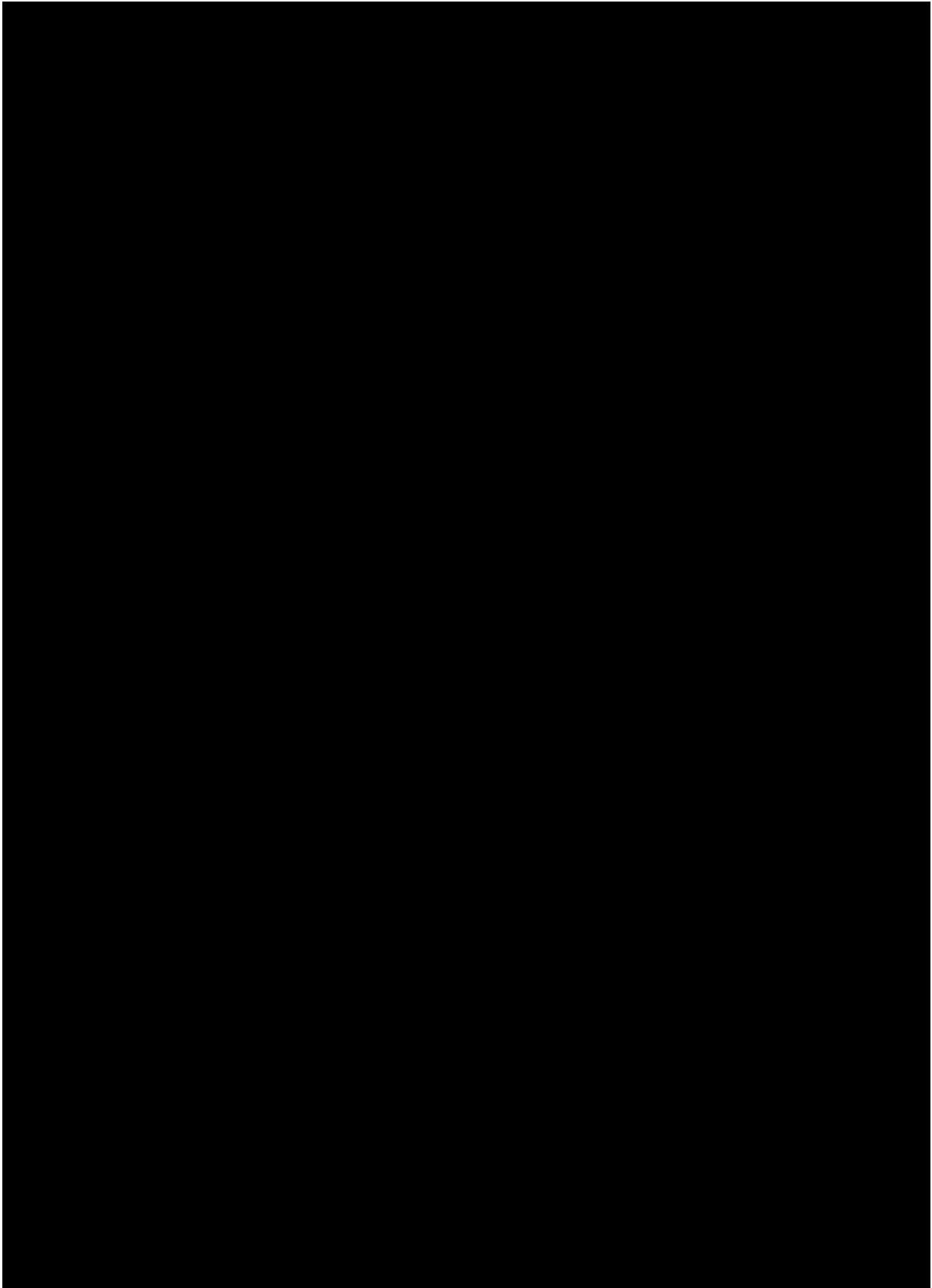


Figure 12. Saving the hello-world.yml file

¥ "Commit" the change to the *local* git repository, using "CodeGit" from the right panel:

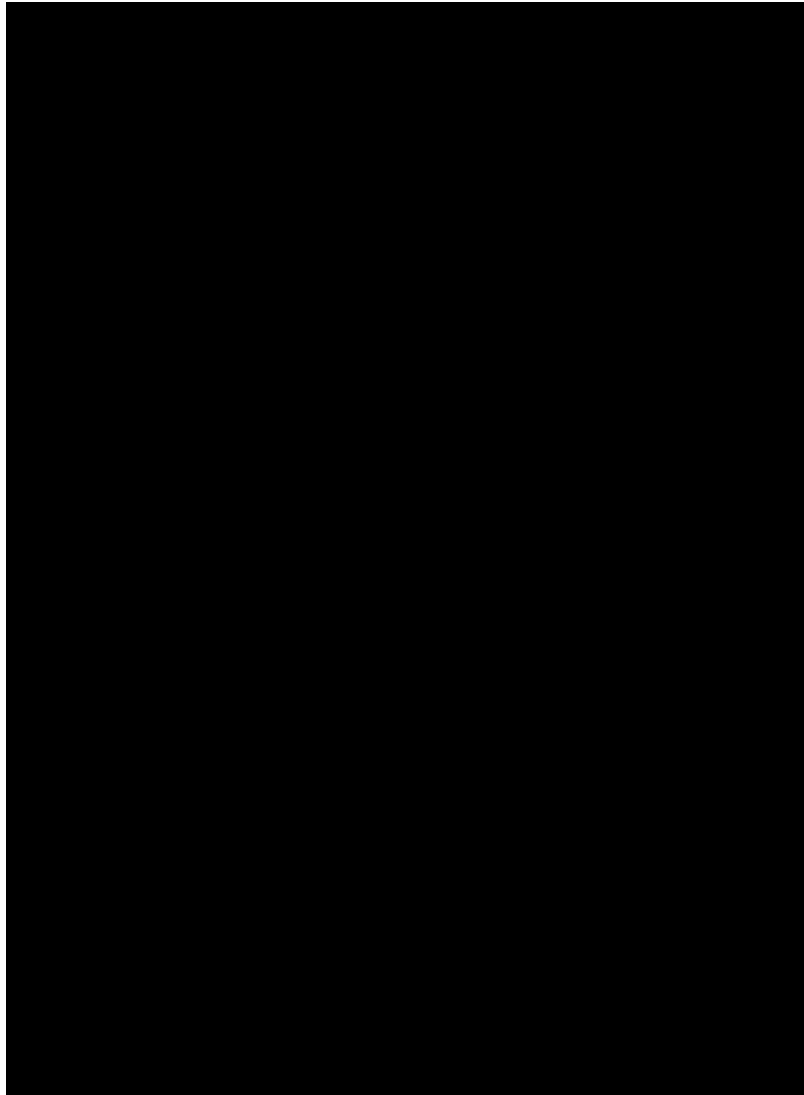


Figure 13. Accessing CodeGit

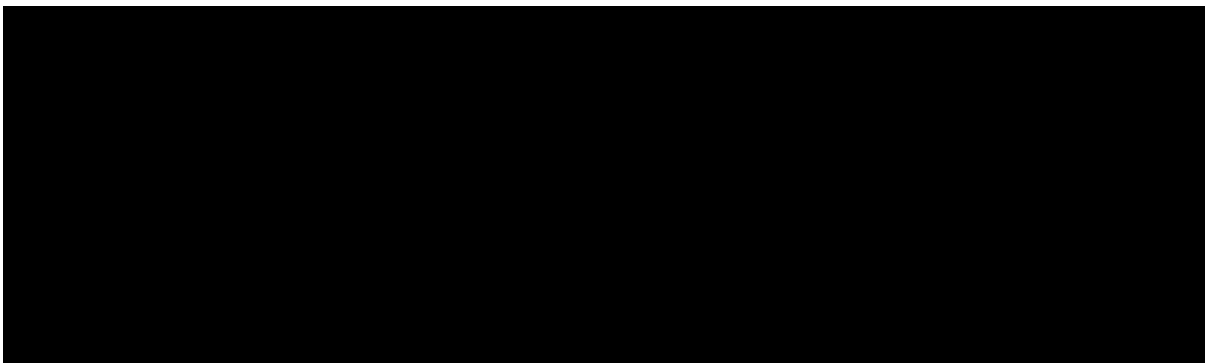


Figure 14. Git-committing hello-world.yml

¥ Write down a comprehensive commit message:

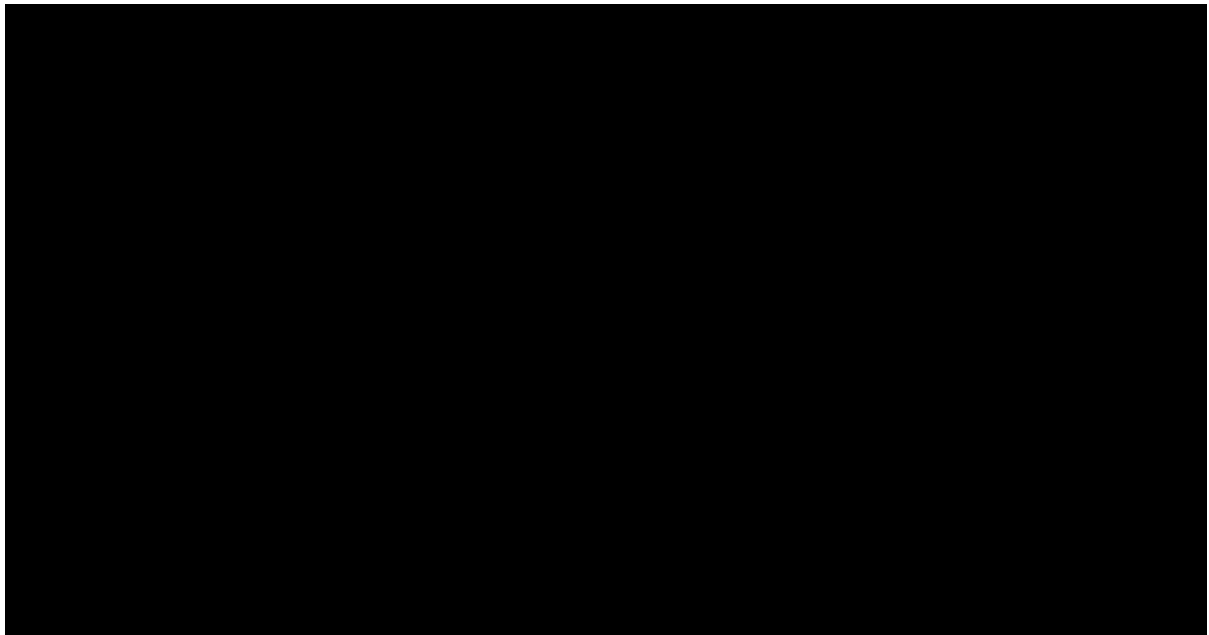


Figure 15. Git commit message

¥ "Push" the change to the *remote* git repository, still using CodeGit:

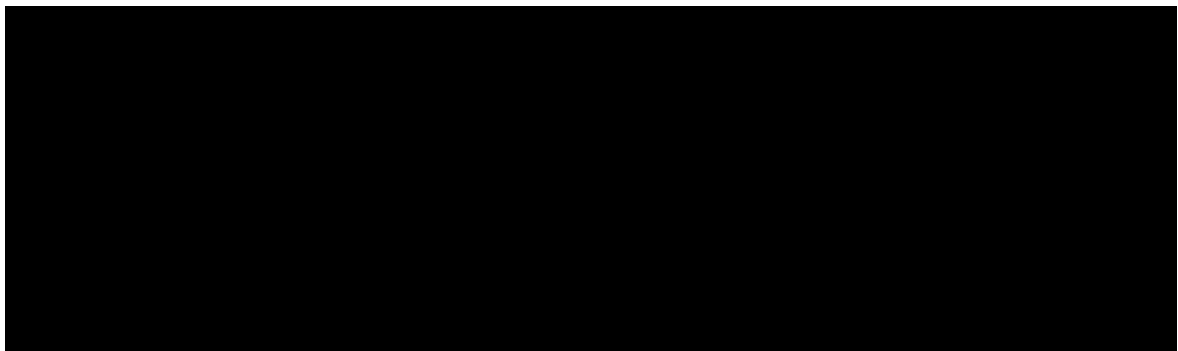


Figure 16. Accessing the Git push

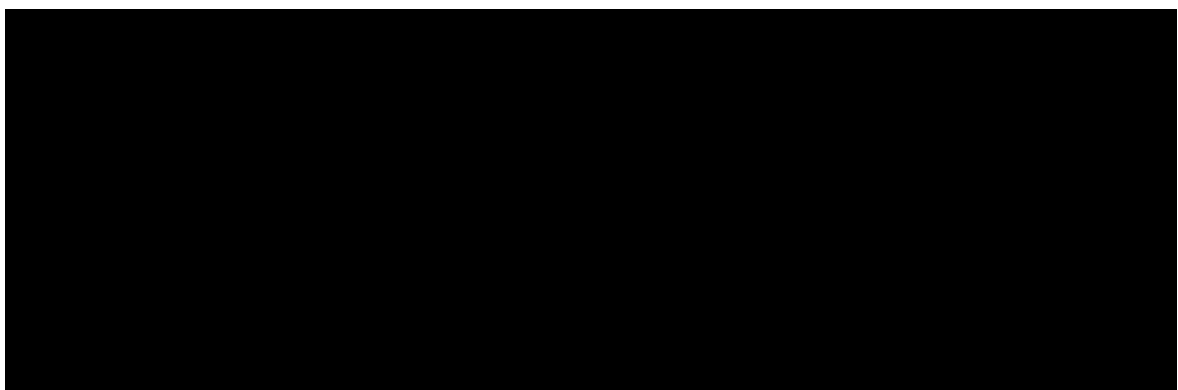


Figure 17. Git push

IMPORTANT

You will be asked to login. Use the value `butler` for both username and password.

! Browse the repository on your GitServer to ensure that your change is there.

TIP | Use the Commits tabs on the repository.

Wait a minute. Did you just push a change without validating it ?

Validating the Change

Let's test this change:

¥ Using the Devbox command line:

! Build a new Docker image with the changed application source code, named `demo-app:1.0.1`, with the following command, to run from the `demoapp` directory:

```
cloudbees-devbox $ docker build -t "demo-app:1.0.1" ./
```

! Run a new instance of the application based on this new image:

```
cloudbees-devbox $ docker run -d --name demoapp-2 -p 30001:8080 demo-app:1.0.1
```

¥ Validate your change:

! Browse to this new application on this URL: <http://localhost:30001>

! Compare to the older deployed version on <http://localhost:30000>

Cleaning Environment

We need to clean all the running Docker containers that we have launched for the demo application.

Use the following commands on the Devbox Command Line prompt:

¥ Check the `demo-app` containers currently running with this command:

```
cloudbees-devbox $ docker ps | grep demo-app
```

¥ Stop the running containers related to `demo-app`:

```
cloudbees-devbox $ docker ps | grep demo-app \
| awk '{print $1}' | xargs docker stop
```

¥ Check again the 2 applications version: you should see an HTTP 502 error page instead.

That's all for this exercise !

Exercise: Defining a CD pipeline

Thinking about a CD Pipeline

The goal of this exercise is to have a clear idea of the CD Pipeline we want to build for our demo application.

Deployment Environment

We define 4 environments for the demo application:

¥ Development Environment:

- ! Represents developer's environment
- ! *Devbox + WebIDE* services

¥ Forge Environment:

- ! This is the centralized place where we build and test the application.
- ! *Jenkins + GitServer + Docker registry*
- ! *Docker* containers on public ports ≥ 32768

¥ Staging Environment:

- ! This is a staging environment that looks like production, aimed at validating applications
- ! *Docker* containers on public ports {20000,30000,30001}

¥ Production Environment

- ! This is the final environment your customers are using.
- ! *Docker* containers on public port 10000
- ! This environment may contains other services not covered here in the next labs.

Figure 18. Cartography of Environments and Services

User Profiles

We define 3 user profiles:

- ¥ End-Users: People who use the application in production.
- ¥ Product Managers: People who validate application behavior before delivering to *End-Users*
- ¥ Engineers: Set of developers, sysadmin, etc. who write, maintain and operate the application

Proposed Pipeline

This is the proposal for the CD pipeline to achieve:

Figure 19. CD Pipeline for demo app

✂ Steps details:

- ! Checkout: Getting the source code. Dedicated step since it can fail due to network latency or bad credentials
 - " Feedback: When failing, this step will provide feedback to Engineers
- ! Build: From a *given* checked-out source code, solving dependencies and compilation phase of the application
 - " Feedback: When failing, this step will provide feedback to Engineers
- ! Unit Tests: From a compiled application, run a set of Unit Tests that must be *fast*
 - " Feedback: When failing, this step will provide feedback to Engineers
- ! Smoke & Acceptance Tests: From an application passing Unit Tests, it runs all smoke and acceptance tests that may be slow but complete.
 - " Feedback: When failing, this step will provide feedback to both Engineers and Product Managers
- ! Staging Deployment: From an application passing ALL tests, makes the *Automated* deployment to the Staging environment.
 - " Feedback:
 - " When deployment is a success, notification is send to Product Managers with the URL of the deployed application.
 - " When failing, feedback is only sent to Engineers
- ! Production Deployment Request: From an application deployed in *Staging*, if the *Product Managers* have manually approved the application, a notification is provided to Engineers to request a Production Deployment
 - " Feedback: the steps themselves are feedback.
- ! Production Deployment: From a *Production Deployment Request*, Engineers can push-button to push-production.
 - " Feedback: Everyone is notified when production deployment happens

TIP

Take time to discuss it with all attendees and the trainer if you do not understand. It will be the "model" for the next labs

That's all for this exercise !

Journey Summary

After this set of exercises, we covered:

- ✚ Accessing and using basics of your Lab Instance
- ✚ How to use the Development Environment: editing code, using command lines
- ✚ How to build, test and run the demo application
- ✚ How to manage a bit of AAA security by using Pull Request system by user profiles
- ✚ A proposed implementation of the CD pipeline