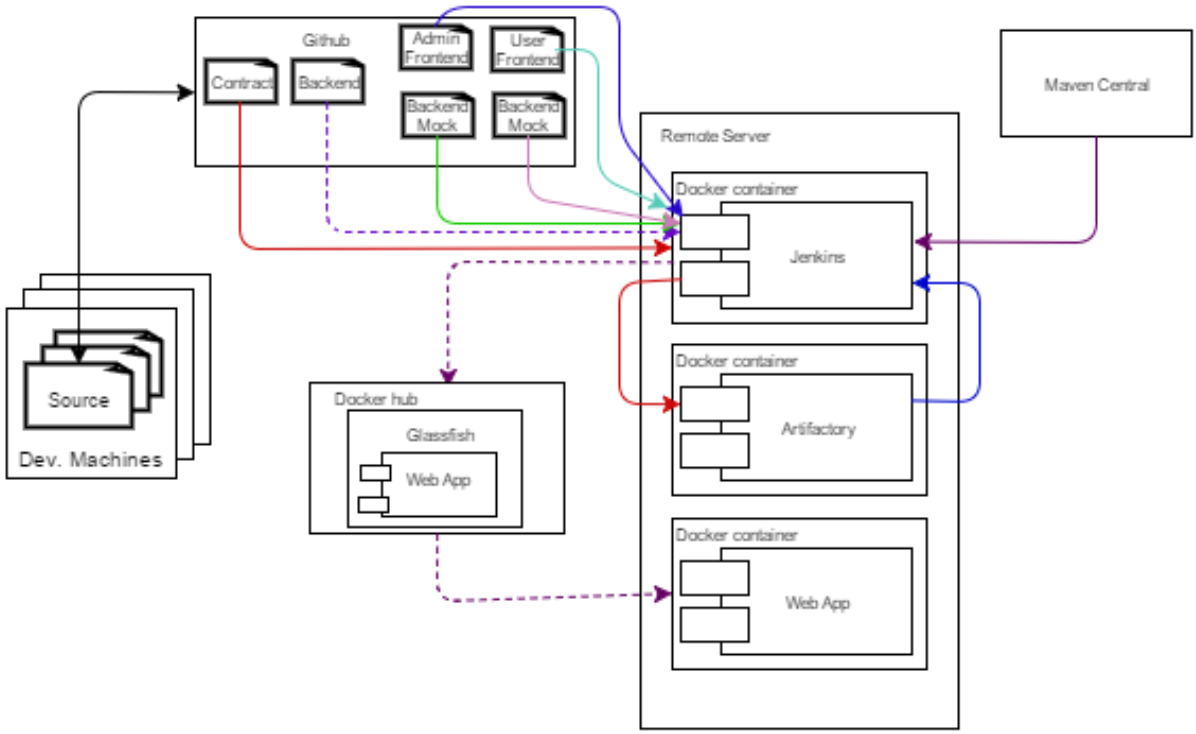
Ferry case - Group B

Continuous integration Setup



Our intended Continuous integration setup in the ferry case project look as above.

Due to lack of time, cause by communication errors and disagreements between the groups in the cluster, we are not able to deploy the build directly from jenkins to dockerhub in time. Instead we are manually uploading the build to Docker.

We could have run all the docker containers on one big server, but to avoid paying for servers and instead use the free server money granted to students. It also gave more people a chance to work with the setup of servers and server connection.

Our final setup has the following elements:

GitHub Repositories for

Ferry case Contract:
https://github.com/Legendslayer/FerryProjectContract

Contract Test:
https://github.com/Infernosaint/ContractTest

Customer Fronted:
https://github.com/tompet815/frontend_customer

Backend:
https://github.com/Infernosaint/FerryBackend

Backend mockup:
https://github.com/Madalina1994/BackendMockFerry

Then we have a 3 separate servers that runs:
Jenkins in a docker container:
http://46.101.194.147:8080/

Artifactory in a docker container:
http://104.236.119.119:8082/artifactory/webapp/browserepo.html?4

Server to run our project dockerfile and war file:
46.101.223.183:8080/ferryproject

The Report for this project is uploaded to the FerryBackend Repository
https://github.com/Infernosaint/FerryBackend

The Toolbox Contract is in the Contract repository
https://github.com/Legendslayer/FerryProjectContract

Flow of our Continuous integration

Every developer upload their code to their particular groups github repository.
All the repositories are connected to jenkins and when a change in a repository is detected, jenkins is notified and will pull the maven project from the repository and start to build the project, pull the needed dependencies from maven central, and pull any other dependencies, like the contract for the backend and fronted project from the Artifactory.
All dependencies are written in the maven project pom.xml file.
If jenkins has build the project successfully it will run the tests made in the ferry contract test.
If successful it will send the project builds to the artifactory.

Dockerhub and glassfish server:

We chose to go with Glassfish server as it is open source and free and we have the most experience with it. Now in the docker file or ssh session we had to manipulate it through the command line. An example of that would be

as-install/bin/asadmin deploy ferryproject.war

Which deploys a war file with the name ferryproject.war taken from our Artifactory and uses port 8080 as we left it unchanged.

The war file is the final version of the front end project, including all dependencies, ie. contract and backend, as well as all external dependencies.

Our Docker setup includes 2 Docker files - 1 to setup Java, Maven , Glassfish , JVM, PostgreSQL and the other to use that image as basis and make a get request to our Artifactory to get the war file that needs to be deployed on Glassfish.
This way we can easily change what we need and we stay very agile.

Example of this is the following Docker file:

```
FROM ubuntu:latest

RUN apt-get -y update


RUN wget
http://104.236.119.119:8082/artifactory/simple/libs-snapshot-local/B/frontendCopy/1.0-SNAP
SHOT/frontendcopy-1.0.20170105.101241-1.war

RUN mv frontendcopy-1.0.20170105.101241-1.war ferryproject.war
RUN systemctl enable postgresql

RUN service postgresql start

RUN as-install/bin/asadmin deploy ferryproject.war



EXPOSE 5432

EXPOSE 8080
```

The other file is looks like this :

```
RUN docker pull ubuntu

RUN sudo apt-get update -y && sudo apt-get upgrade -y

RUN apt-get install openjdk-8-jre -y

RUN apt-get install openjdk-8-jdk -y

RUN apt-get install maven -y

RUN apt-get install postgresql postgresql-contrib -y

RUN systemctl enable postgresql

RUN service postgresql start

RUN apt-get install wget -y

RUN apt-get install unzip -y

RUN wget http://download.java.net/glassfish/4.1.1/release/glassfish-4.1.1.zip

RUN unzip glassfish-4.1.1.zip

RUN glassfish4/bin/asadmin start-domain

EXPOSE 5432
EXPOSE 8080
```

Jenkins setup:

Generated ssh key pair

We use ssh keys to make a safe and easy connect to our jenkins.
We recommend you use a linux based(like ubuntu) machine to create ssh keys and to connect to the server. This is far easier than windows and since our server also runs ubuntu it be more simple to runs commands on the server.

To Generate a ssh key pair is pretty simple just type the following line in the terminal.

```
ssh-keygen -t rsa
```

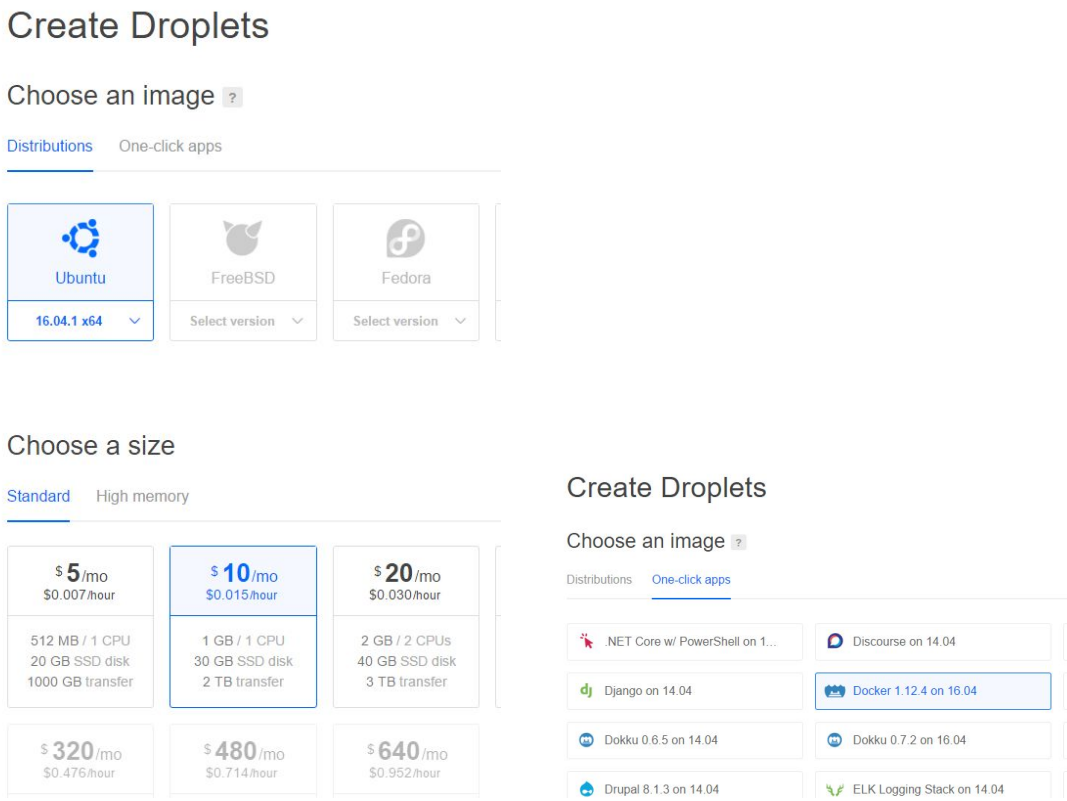
To get the public key type this in the terminal

```
$ cat ~/.ssh/id_rsa.pub
```

Copy the key for later use.

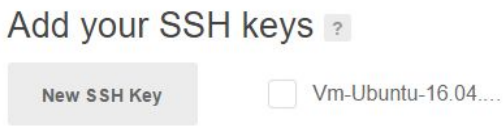
Create Droplet on Digital ocean

When we have generated the ssh key, we can setup a server at Digital ocean, that runs the newest ubuntu and we set it to preinstall docker on the server.



Before we create the droplet we are gonna add the ssh public key we have created earlier

To the droplet, remember to mark the key you just added and then create the droplet.



When the server starts running, you will be able to connect to the server from the machine that you created the ssh key on, the machine will have the private key that match the server's public key.

Connect to the jenkins server

When you connect to the server for the first time write this in the terminal, remember to change it to your servers ip address, which you can is on digital ocean.

```
ssh root@<your_ip>
adduser builder
usermod -aG sudo builder
exit
```

Now you have made a user on the server and set it as a admin user and exit the server again.
Now you can connect to the server with the following line

```
ssh builder@<your_ip>
```

If you made a password for the user you created it will be asking for it.

Install Jenkins on the server

When you are login on the server you can now install jinkins.
We have done it via docker and installed the office jenkins docker container.

```
docker pull jenkins
```

Pulls the jenkins container from dockerhub

```
docker run -p 8080:8080 -p 50000:50000 jenkins
```

Setup and runs the jenkins container
Look out for a initial admin password that will be print out in the terminal doing the setup process. You need to copy it and use it later when you go to the jenkins webpage.
You can check to see if the jenkins container is running by typing

```
$ docker ps
```

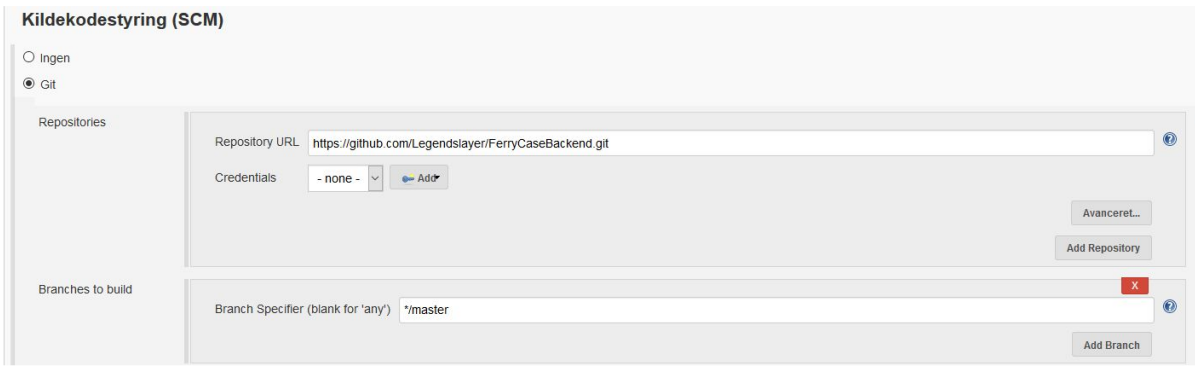
And it will show a list of all running containers. There should be one named jenkins.
Now you can exit the server.

And then type the ip address of the server with port 8080 in your browser. First time your access the webpage it will ask for the initial admin password you copy from the terminal doing setup of jenkins.

Follow the setup process to create a new user.

Setup of our projects in jenkins

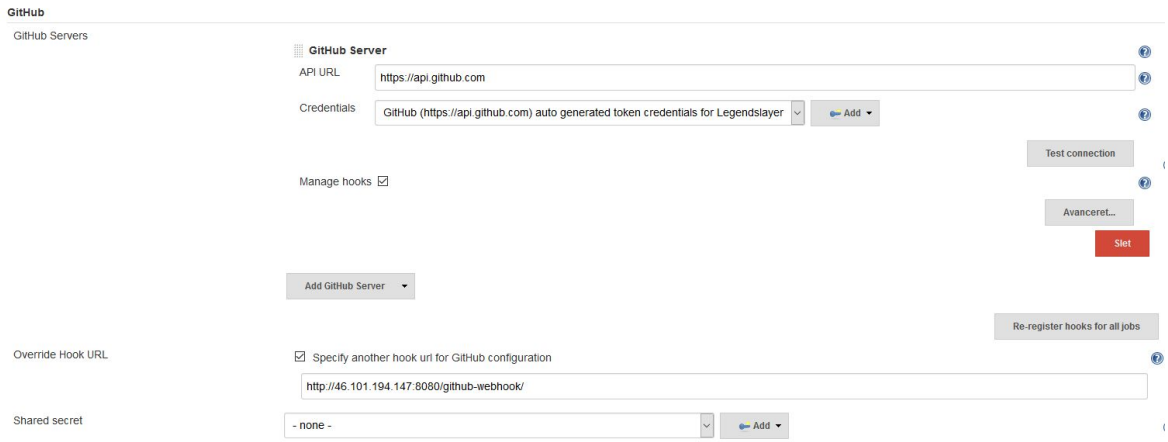
Now we have jenkins up and running, we create a new item, and set it to be a maven project. Now we can click on the new item and configure it to pull the code from github when building. Here we is configuring the item for the backend project.



Then we add a post-process to the item, that will deploy the successful build to our artifactory. To let jenkins access the artifactory you need to given jenkins a username and password for the artifactory. This can be set as default credentials under config jenkins and under artifactory. It is recommend that you use the hashed password and not the plain password for security reasons. You can get the hashed password in from the artifactory website and under user profile, that you are using.

Now jenkins will send the successful builds to the artifactory.

For jenkins to be able to automatically build when something is push to a github repository, you need to add a webhook in the repository.
You can get the webhook in manage jenkins and look on the section github, then click advanced, and check “Specify another hook url for GitHub configuration” on, then you can see the link. Copy it and uncheck it again since do not want to change but just see the link.



Then you go into your github repository setting and under webhook adds a new webhook and paste the link in.

Now you just have to check some variables on in the jenkins build config you want to automatically build.

These steps is then repeated for every project (contract, backed, frontend-customer).

Testing

After building the contract, and somewhat agreeing on it, we wrote contract tests:
https://github.com/Infernosaint/ContractTest

These tests were mainly focused on verifying the returned enitites from the backend, both mock and development. The test was added to jenkins and built to the artifactory, so it could be added as dependencies from the different instances of the backend.

When you test an instance of a backend, you just add the Contract Test to your test suite, and it will run the tests on your relevant backend.

This test first approach gave us a really clear understanding of the flow of data in the big picture, so our code could be written with fulfilling these tests in mind.

Every time you push changes to the backend, Jenkins will pull from github, then build and test the new version, giving you a clear overview of how the development is going over time.

One such test makes sure that a Reservation is actually updated on updateReservation(), and not just the same local version that you just made

```
@Test
   public void testUpdateReservation() {
      assumeThat(manager, not(nullValue()));

      Collection<DepartureDetail> departures = manager.getDepartures(new LineIdentifier(1), new
Date());

      assumeThat(departures, not(nullValue()));
      assumeTrue(departures.size() >= 1);
      String customerName = "Kaloyan";
      DepartureIdentifier depId = new DepartureIdentifier(departures.iterator().next().getId());
      ReservationDetail resDet = manager.saveReservation(depId, 2, 3, 1, 0, 0, customerName);
      System.out.println(resDet.getId());
      assertEquals(resDet.getCustomerName(), customerName);
      customerName = "Mads";
      resDet = manager.updateReservation(new ReservationIdentifier(resDet.getId()), depId, 0, 0, 0, 0, 0,
customerName);
      assertEquals(resDet.getCustomerName(), customerName);
      assertEquals(customerName, manager.getReservation(new
ReservationIdentifier(resDet.getId())).getCustomerName());
   }
```