Classic
Installation Manual

by Team Classic

Contents

1. Deploying the web application

This chapter explains how to deploy the Classic web application in a live environment. The Classic web application is primarily developed in Java and AngularJS so it can run on multiple platforms. This manual will only cover the deployment on Linux.

1.1 Requirements

- A server with Linux installed. (In the examples we will be using Ubuntu Server 14.04 LTS)
- Git, to pull the code from the Git repository.
- Gradle, to build the backend.
- Java JRE or JDK (Version 8+).
- The Apache2 web server.
- A PostgreSQL database.
- A web browser for accessing the Classic web application. The officially supported browsers are Chrome (Version 45+) and Mozilla Firefox (Version 43+).

1.2 Download the files

To pull the application files from the Git repository you will need Git. This can be installed with the command

```
apt-get install git
```

Once you have git you can pull the files to a desired location

```
git clone https://github.ugent.be/jloverme/design-project.git
```

1.3 Set up your server

1.3.1 Set up you database

1. If you haven't alredy install Postgres this can be done by running

   ```
   apt-get install postgresql
   ```

2. Now open a database terminal by executing

   ```
   sudo -u postgres psql postgres
   ```

3. Here you can change the password for the default user "postgres" by using the command

```
\backslash password postgres
```

4. Now you can create a new database

```
CREATE DATABASE classic WITH OWNER postgres;
```

5. The postgres terminal can now be closed by typing \q

6. To create the tables in the Classic database the Classic run the SQL-script "/design-project/database/fillClassic_6-Release1.0.sql"

```
sudo -u postgres psql -d classic -a -f fillClassic\_6-
   Release1.0.sql
```

1.3.2 Set up The Classic API

To install the Classic API you will only need Java installed on your server. The other dependencies will be fetched by gradle.

Install Java and Gradle

Installing Java on linux can be done by executing the following commando.

```
apt-get install default-jre
```

Gradle has to be installed manually. Firstly fetch the Gradle files.

```
mkdir -p ~/opt/packages/gradle && cd \$\_
wget https://services.gradle.org/distributions/gradle-2.3-bin.zip
unzip gradle-2.3-bin.zip
```

Next, we will create a symlink that provides a shorter path to the specific Gradle version. The symlink will allow us to upgrade Gradle later without changing any other configuration.

```
ln -s ~/opt/packages/gradle/gradle-2.3/ ~/opt/gradle
```

Open your .profile file in your favourite text editor and paste the following at the bottom of your .profile file.

```
# Gradle
if [ -d "\$HOME/opt/gradle" ]; then
    export GRADLE_HOME="\$HOME/opt/gradle"
    PATH="\$PATH:\$GRADLE_HOME/bin"
fi
```

Finally, source your .profile and test gradle.

```
source ~/.profile
which gradle
gradle -version
```

Installing the API

The API files are located in the folder "design-project/Backend". Firstly we will need to edit the properties file "design-project/Backend/persistance/src/resources/dbconfig.properties". Fill in database user, password and name.

Also in the file "design-project/Backend/src/config/Config.java" change the HOME_URL to the url you will be using. In the same file you can choose a directory that will be used to upload Coursenotes too"

Now we can build the Backend. Build the JAR by executing the command.

```
gradle assemble
```

The JAR can be found in the folder "/design-project/Backend/build/libs". Copy the JAR to a desired location and execute it by running the command

```
java -Dserver.port=8092 -jar classic-1.0.0.jar &
```

The port can be replaced by any free port you want.

1.3.3 Set up Apache2 and the Frontend

Set up Apache2

First we will need to install Apache2.

```
apt-get install apache2
```

Edit your apache config to proxy the backend to "yourdomain.com/api". The apache config can usually be found in "/etc/apache2/sites-available/". The config file should look like this

```
<VirtualHost *:80>
        ServerName student-dp8.intec.ugent.be
        ServerAlias classic
        ServerAdmin classic@ugent.be
        DocumentRoot /var/www/html

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined

        ProxyRequests Off
        <Proxy *>
                Order deny,allow
                Allow from all
        </Proxy>
        ProxyPreserveHost on

        ProxyPass /api http://localhost:8092 nocanon
        ProxyPassReverse  /api http://localhost:8092
        AllowEncodedSlashes NoDecode
</VirtualHost>
```

To allow the poxy to work we have to enable the proxy module en restart apache.

```
a2enmod proxy
a2enmod proxy_http
a2enmod proxy_ajp
a2enmod rewrite
a2enmod deflate
a2enmod headers
a2enmod proxy_balancer
a2enmod proxy_connect
a2enmod proxy_html
service apache2 restart
```

Serve the frontend to apache

The last thing to do is set up the frontend. This is done by simply copying the contents of the folder "design-project/Frontend/" to the apache documentroot which usually is "/var/www/html". To make the frontend consume our API edit the file "js/global.js" and change the url of the REST service to your url.

Congratulations, you have just set up the Classic web application!

2. Setting up a development environment

First of to you will want to set up a second Classic instance with the development build. This is done in the same way as setting up a normal Classic instance. When setting up the development instance on the same server as your release instance, you need to make sure you pick another port number for the backend. This can be chosen when running the JAR. Then you can just edit the apache config to set up a second proxy to the development api. You will also have to put the development frontend in a separate folder in the documentroot. The apache config will look something like this

```
<VirtualHost *:80>
        ServerName student-dp8.intec.ugent.be
        ServerAlias classic
        ServerAdmin classic@ugent.be
        DocumentRoot /var/www/html

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined

        ProxyRequests Off
        <Proxy *>
                Order deny,allow
                Allow from all
        </Proxy>
```

```
        ProxyPreserveHost on

        ## Jenkins ##

        ProxyPass /jenkins http://localhost:8080/jenkins nocanon
        ProxyPassReverse  /jenkins  http://localhost:8080/jenkins

        ## Sonar ##

        ProxyPass /sonar http://localhost:9000/sonar nocanon
        ProxyPassReverse  /sonar  http://localhost:9000/sonar

        ## Development ##

        ProxyPass /development/api http://localhost:8091 nocanon
        ProxyPassReverse  /development/api http://localhost:8091

        ## Release ##

        RedirectMatch ^/$ /release/

        ProxyPass /release/api http://localhost:8090 nocanon
        ProxyPassReverse  /release/api  http://localhost:8090

        AllowEncodedSlashes NoDecode
</VirtualHost>
```

In this config file, the proxies for the Jenkins and Sonar have already been added.

For the integration test you should edit "design-project/Backend/integtestsrc/config/ITConfig.java". This file is the same as "design-project/Backend/src/config/Config.java" but is used in the integration test.

2.0.1 Creating a test database

To be able to run JUnit tests and integration tests we will need a separate test database. It is easy to set up this database since it is an exact copy of the release database.

```
sudo -u postgres psql postgres
CREATE DATABASE classictest WITH OWNER postgres;
```

```
sudo -u postgres psql -d classic -a -f fillClassic\_6-Release1.0.
   sql
```

The next step is editing the properties files of the 3 major backend modules to point to the test database. These files can be found in

- "design-project/Backend/src/resources/testdbconfig.properties"
- "design-project/Backend/controllers/src/resources/testdbconfig.properties"
- "design-project/Backend/persistance/src/resources/testdbconfig.properties"

2.0.2 Installing and configuring Jenkins

Installation

Jenkins can be installed by running the following commands.

```
wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key |
   sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable
   binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

By default jenkins will run on port 8080. this can be changed by editing "/etc/default/jenkins"

To be able to reach jenkins you will have to add a proxy to apache. In the apache config add the folowing lines.

```
## Jenkins ##

ProxyPass /jenkins http://localhost:8080/jenkins nocanon
ProxyPassReverse  /jenkins  http://localhost:8080/jenkins
```

Configuration

Open the Github plugin manager (manage jenkins → manage plugins) and install the following plugins:

- Github plugin
- Gradle plugin
- Slack notification plugin
- SonarQube plugin

We want to create a job that runs on every commit to our github repository. On the jenkins home page, click "new item". Name the job "classic" and select "freestyle project".

Jenkins will present you a configuration form. The project name should be "classic". Tick the box next to "github project" and fill in the classic repository url.

In the section "source code management", select "git" and enter the project url again. For private repositories, select "add credentials" and fill add your github credentials. Then fill in "*/development" for branches to build.

In the section "build triggers", tick the box "trigger builds remotely" and fill in "classicpush" for the authentication token. This way, a build can be triggered on github using the following url:

```
http://student-dp8.intec.ugent.be/jenkins/job/Classic/build?token
   =classicpush
```

In the github repository, this should be configured by going to "settings" → "webhooks and services" → "add webhook".

Back in the jenkins config form, click "add build step" and select "execute shell". Paste the following script:

```
cd Backend
gradle clean
gradle build
```

For post-build actions, add "publish junit test results". Fill in "**/build/test-results/*.xml" and amplication factor "1.0". Also add "slack notifications" and check all boxes. Lastly, add "post build task". Log text should be "BUILD SUCCESFUL", check the "Run script only if all previous steps were successful" box and paste the following script:

```
cd /srv/classic/deployement
./deploy_development.sh
```

Ofcourse, we sill need to create this "/srv/classic/deployement/deploy_development.sh" file. Paste the following contents:

```
echo "** Deploying development..."
## Stop the running backend
echo "** Stopping running backend instance"
screen -S "classic_development" -p 0 -X stuff "^C"
screen -S "classic_test" -p 0 -X stuff "^C"
cd /srv/classic/development/design-project
## Pull the files from github
echo "** Pulling files from development branch"
git checkout development
git pull origin development
## Copy json file ##
echo "** Set up swagger file"
cd /srv/classic/development/design-project/Backend
cp classic.json /var/www/classic/resources
## Build
echo "** Building the backend"
cd /srv/classic/development/design-project/Backend
gradle assemble
## Start the new backend
echo "** Starting a new backend instance"
screen -S "classic_development" -p 0 -X stuff "cd /srv/classic/
   development/desig

   n-project/Backend/build/libs\n"
screen -S "classic_development" -p 0 -X stuff "java -Dserver.port
   =8091 -jar clas

   sic-1.0.0.jar\n"
## Overwrite the frontend files
echo "** Overwriting frontend files"
cd /srv/classic/development/design-project/Frontend
```

```
cp -r * /var/www/classic/development
## Set up /test
echo "** Setting up test"
cp -r /srv/classic/development/design-project/Backend /srv/
   classic/test
cd /srv/classic/test/Backend/persistence/src/resources
rm dbconfig.properties
cp testdbconfig.properties dbconfig.properties
rm /srv/classic/test/Backend/src/resources/dbconfig.properties
cp testdbconfig.properties /srv/classic/test/Backend/src/
   resources/dbconfig.prop

   erties
cd /srv/classic/test/Backend
gradle assemble
screen -S "classic_test" -p 0 -X stuff "cd /srv/classic/test/
   Backend/build/libs\

   n"
screen -S "classic_test" -p 0 -X stuff "java -Dserver.port=8092 -
   jar classic-1.0

   .0.jar\n"
cd /srv/classic/development/design-project/Frontend
cp -r * /var/www/classic/test
perl -pi -e 's/development/test/g' /var/www/classic/test/js/
   globals.js
```

Afterwards, go back to jenkins and click save. You created the classic jenkins job!

The second job we want to create is the integrationtests job, fill in the name "integrationtests" and configure the git repository as we did above. In "build triggers", select "build periodically" and fill in the following schedule:

```
H 2 * * *
```

As build script, use:

```
cd Backend
gradle clean
gradle build
gradle integtest
```

Add JUnit test reports and slack notifications as we did above, then click "save".

2.0.3 Installing and configuring SonarQube

Installation

First we will have to create a database user and a database for SonarQube.

```
sudo -u postgres psql postgres
```

```
CREATE USER sonar WITH PASSWORD 'sonar';
CREATE DATABASE classic WITH OWNER = sonar;
```

Next we get the latest SonarQube version.

```
wget http://dist.sonar.codehaus.org/sonarqube-5.1.zip
unzip sonarqube-5.1.zip
mv sonarqube-5.1 /opt/sonar
```

Open /opt/sonar/conf/sonar.properties with your favourite text editor, and modify it. Uncomment the user credentials and the postgresql related settings:

```
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar

sonar.jdbc.url=jdbc:postgresql://localhost/sonar
```

Since we want to proxy SonarQube also edit

```
sonar.web.host=127.0.0.1
sonar.web.context=/sonar
sonar.web.port=9000
```

To proxy to the SonarQube instance add the following lines to the apache config:

```
## Sonar ##

ProxyPass /sonar http://localhost:9000/sonar nocanon
ProxyPassReverse  /sonar  http://localhost:9000/sonar
```

Finally we can start the SonarQube server:

```
sudo /opt/sonar/bin/linux-x86-64/sonar.sh start
```

Configuration

We also want to configure a jenkins job for the sonar analysis. So, back in jenkins, the last job we want to create is called the "sonar" job. Configure the github repository exactly like before. Build periodically with schedule:

```
H 4 * * *
```

Use the build script:

```
cd Backend
gradle clean
gradle build
```

Next, add the build step "invoke sonarqube analysis" with the analysis properties:

```
sonar.projectKey=classic
sonar.projectName=classic
sonar.projectVersion=0.1
sonar.modules=Backend,Backend/controllers,Backend/models,Backend/
   persistence
sonar.tests=junit
sonar.language=java
sonar.java.coveragePlugin=jacoco
```

```
sonar.dynamicAnalysis=reuseReports
Backend.sonar.projectKey=Backend
Backend.sonar.sources=src/
Backend.sonar..tests=testsrc/
Backend.sonar.java.binaries=build/classes/
Backend.sonar.java.libraries=build/libs/*.jar
Backend.sonar.projectBaseDir=Backend/
Backend/controllers.sonar.projectKey=controllers
Backend/controllers.sonar.sources=src/
Backend/controllers.sonar.tests=testsrc/
Backend/controllers.sonar.java.binaries=build/classes/
Backend/controller.sonar.java.libraries=build/libs/*.jar
Backend/controllers.projectBaseDir=controllers/
Backend/models.sonar.projectKey=models
Backend/models.sonar.sources=src/
Backend/models.sonar.tests=testsrc/
Backend/models.sonar.java.binaries=build/classes/
Backend/models.sonar.java.libraries=build/libs/*.jar
Backend/models.projectBaseDir=models/
Backend/persistence.sonar.projectKey=persistence
Backend/persistence.sonar.sources=src/
Backend/persistence.sonar.tests=testsrc/
Backend/persistence.sonar.java.binaries=build/classes/
Backend/persistence.sonar.java.libraries=build/libs/*.jar
Backend/persistence.projectBaseDir=persistence/
sonar.jacoco.reportPath=build/jacoco/test.exec
sonar.jacoco.itReportPath=build/jacoco/integtest.exec
```

Leave the other fields empty and click ”save”.