# AMIS SIG – Introduction to JHipster

July 2018

In this workshop you will experience how JHipster can boost your projects by generating a full stack application. You will build an application which has a data store, server-side rest API and browser-based client. You will explore the different technologies which are used for the server and client applications. After that you can deploy this application to your preferred cloud provider.

Installing JHipster

To have a great experience with JHipster you need a few things installed.

java, node.js, yarn, git and docker

My preferred way is to install it with a package manager, but also manual is fine

### Installation types

JHipster provides 6 ways of working with JHipster. If in doubt, choose our 2nd option, “Local installation with Yarn”:

* [JHipster Online](https://start.jhipster.tech/) is a simple way to generate an application without installing JHipster in the first place.
* “Local installation with Yarn” is the classical way of working with JHipster. Everything is installed on your machine, which can be a little complex to set up, but that’s how most people usually work. In case of doubt, choose this installation.
* “Installation with a package manager” is only available for Mac OS X and Windows. This is a very simple installation method, if you use a package manager, but it is still in BETA.
* The Vagrant-based “[development box](https://github.com/jhipster/jhipster-devbox)”, with all tools already set up in a Ubuntu-based virtual machine.
* The “[Docker](https://www.docker.io/)” container, which brings you a lightweight container with JHipster installed.

### Manual Local installation with Yarn

1. Install Java 8 from [the Oracle website](http://www.oracle.com/technetwork/java/javase/downloads/index.html).
2. Install Node.js from [the Node.js website](https://nodejs.org/) (prefer an LTS 64-bit version)
3. Install Yarn from [the Yarn website](https://yarnpkg.com/en/docs/install)
4. If you want to use the JHipster Marketplace, install Yeoman: yarn global add yo
5. Install JHipster: yarn global add generator-jhipster
6. Install Git from [the Git website](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
7. Install docker from [the Docker website](https://www.docker.com/community-edition)

### Installation with a package manager

#### Installation with Homebrew on Mac OS X

JHipster provides a [Homebrew](https://brew.sh/) package, available on <http://formulae.brew.sh/formula/jhipster>.

To install JHipster (as well as Node and Yarn), just type:

brew install jhipster

Also install git and docker

#### Installation with Chocolatey on Windows

JHipster provides a [Chocolatey](https://chocolatey.org/) package, available on <https://chocolatey.org/packages/jhipster>.

To install JHipster (as well as Node, Yarn, Yeoman, Java and Git), just type:

choco install jhipster

Also install git and docker: choco git docker-for-windows

## Creating an application

Now that JHipster is installed, your next step is to create an application

First of all, create an empty directory in which you will create your application:

mkdir myapplication

Go to that directory:

cd myapplication/

To generate your application, type:

Jhipster

Answer the questions asked by the generator to create an application tailored to your needs. Those options are described in [the “creating an application” section on the JHipster site](https://www.jhipster.tech/creating-an-app/#2). Add multiple languages to see the internationalization in action. .

Once the application is generated, you can launch it using

Maven (./mvnw on Linux/MacOS/Windows PowerShell, mvnw on Windows Cmd)

or Gradle (./gradlew on Linux/MacOS/Windows PowerShell, gradlew on Windows Cmd).

The application will be available on [http://localhost:8080](http://localhost:8080/)

If you want to sign in, you can try the default accounts:

- Administrator (login="admin" and password="admin")

- User (login="user" and password="user").

Notice the functionality which is already there:

1. The site is marked as development in the top-left corner with a ribbon. This only shows in the development profile, but it is easily extended to show ribbons in other environments
2. The version of the application is show in the header next to the name of the application. This version can be used for bug reports.
3. The site is responsive when you switch to mobile and tablet sizes
4. The top-right contains the main menu with languages, sign-in and register
5. Switching the language will change the site, but also the language in emails sent to you.
6. The site has a registration form to create a new account. Notice the validations on the fields and the hints shown for the password.
7. When you register the log in the console shows an error “Connection refused: connect”. The server application tried to send an activation email, but failed to connect to a mail server on localhost.
8. Use the activation link from the logging which is just above the exception stacktrace. It will look similar to <http://127.0.0.1:8080/#/activate?key=01873986589764932995>
9. The sign in page has the expected functionality for local user management. A form, remember me token and password reset.
10. Login with your new account or use the “user” account with password “user”.
11. The menu contains new items. The “Entities” menu is still empty, but we are going to fill that later in this workshop. The “Account” menu has new items for “Settings”, “Password” and “Sign out”.
12. Sign out and Sign back in with the “admin” account with password “admin”
13. Now there is one more menu item “Administration” with many sub items.
14. In “User Management” you can administer the user. Assigning profiles/roles and disable an user.
15. The Metrics dashboard shows all statistics of the different layers (http, service, cache, data store) of the application and system. .The only thing that is missing are the graphs over time. These graphs should be made by the monitoring system of your platform. This application has JMX, [Graphite](https://graphiteapp.org/) and [Prometheus](https://prometheus.io/) exporters for the monitoring system.
16. The health dashboard gives health information on various parts of the application. Many health checks are provided out-of-the-box by Spring Boot Actuator, and it’s also very easy to add application-specific health checks.
17. The Configuration dashboard shows all configuration properties of you application. These are the final values after all the different property locations (global, profile and environment) are merged.
18. The Audits dashboard shows all login attempts.
19. The logs dashboard allows to manage at runtime the Logback configuration of the running application. Changing the log level of a Java package is as simple as clicking on a button, which is very convenient both in development and in production.
20. The API dashboard uses Swagger-UI to document and test your exposed REST endpoints.
21. Depending on your database you will have a menu item to browse the data in the database.

## Explore the sources

Let’s have a look at the code which has been generated. Open the directory in your favorite editor or IDE. The root of the project has many files

* .editorconfig
  + The editorconfig helps developers define and maintain consistent coding styles between different editors and IDEs. Visit [editorconfig.org](https://editorconfig.org) for more information. This file makes it easy to work in one team with different editor. Not forcing all members to work with the same editor.
* .gitattributes
  + A gitattributes file is a simple text file that gives attributes to pathnames. A file which matches a pathname is processed by Git based on these attributes. The attributes control if it is a binary/text file and which line ending characters should be used. This file makes it possible for team members to be on different operating systems.
* .gitignore
  + A gitignore file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected.
* .prettierrc
  + Prettier is an opinionated code formatter with support for many languages. It is supported by many editors to prettify your code. This is the configuration file for prettier. Visit [prettier.io](https://prettier.io)
* .prettierignore
  + A prettierignore specifies which paths should not be prettified when prettier is run on the full project.
* .project
  + This Eclipse project file is used to setup Eclipse when you import this directory as a project..
* .yo-rc.json
  + Storing user configuration options for yeoman. JHipster uses yeoman to generate the project and stores you answers in here.
* angular.json
  + The configuration file of an Angular Workspace. An Angular Workspace is a directory which is generated via Angular CLI and able to contain multiple projects or libraries.
* Mvnw/gradlew
  + Shell script for linux/osx to run the maven/gradle wrapper. The wrapper is the easiest way to run maven/gradle. It will download and install maven/gradle when it is not installed yet.
* mvnw.cmd/gradlew.cmd
  + Windows script to run the maven/gradle wrapper
* package.json
  + This file is used to give information to npm (Node Package Manager) and Yarn to identify the project as well as handle the project's dependencies. Everything for the frontend is configured here. The scripts section shows which commands are available to run.
* pom.xml
  + This file is used to give information to maven to identify the project as well as handle the project's dependencies. Everything for the backend is configured here.
* proxy.conf.json
  + Using the [proxying support](https://webpack.js.org/configuration/dev-server/" \l "devserver-proxy) in webpack's dev server to highjack certain URLs and send them to a backend server. The yarn start command starts the webpack dev server and uses this configuration.
* README.md
  + Contains a lot of useful information how to work with this JHipster project.
* tsconfig.json
  + The frontend is written with Typescript. This is the configuration file for the Typescript compiler.
* tsconfig-aot.json
  + The typescript compiler configuration, but then setup for the Angular Ahead-of-Time (AOT) compiler.
* tslint.json
  + Typescript Linter configuration. A linter is a tool that analyze source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.
* yarn.lock
  + In this file yarn keeps track of the actual installed dependencies in the node\_modules directory. All team members will get the exact same dependency versions when this is checked into version control.
* Webpack
  + This directory contains the webpack configuration for the frontend.

The src folder of the project contains the standard folder structure of a maven/gradle java project

* main - Application sources
  + docker
    - .dockerignore – Ignore files for docker image creation
    - app.yml – docker-compose file to build and start the application in docker
    - Dockerfile – Docker build file for the jhipster application
    - entrypoint.sh – start script of the docker image -
    - mysql.yml – docker-compose file to start mysql for development
    - sonar.yml – docker-compose file to start the static analyzer server from SonarQube
  + java/nl/amis/sigdemo/
    - aop – Contains an Aspect for logging execution of service and repository of Spring components. Visit [Aspect Oriented Programming with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop)
    - config - [Spring Framework configuration files](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html)
    - domain – Domain Model types (entities and relations) eg. Authority, User
    - repository – Data store repositories for JPA or NoSql. Visit [Spring Data](https://spring.io/projects/spring-data)
    - security – [Spring security](https://spring.io/projects/spring-security) implementation with JWT tokens
    - service – Business services
    - web – REST api endpoints implemented with [Spring Data Rest](https://projects.spring.io/spring-data-rest/)
    - ApplicationWebXml.java - helper that provides an alternative to creating a web.xml
    - MyapplicationApp.java – Main class for starting a [Spring Boot](https://spring.io/projects/spring-boot) application
  + resources
    - config – yaml configuration files for spring boot
    - templates – Email templates
    - banner.txt – ASCII art banner for the log
    - logback-spring.xml – logging configuration
  + webapp – Angular/React source . The yarn/webpack buld tool for the frontend is configured to use this directory.
* test - Test sources
  + java – Unit tests for all generated java application sources
  + javascript – Test for all generated javascript application sources
  + resources – configuration and other files required to run the tests

## Create a data model

It is time to add a data model to the application. The entities in a data model needs to be added on several places In the application.

* A database table
* A Liquibase change set (src\main\resources\config\liquibase)
* A JPA Entity (src\main\java\nl\amis\sigdemo\domain)
* A Spring Data JPA Repository (src\main\java\nl\amis\sigdemo\repository)
* A Spring MVC REST Controller, which has the basic CRUD operations (src\main\java\nl\amis\sigdemo\web\rest)
* An Angular router, a component and a service (src\main\webapp\app)
* An HTML view (src\main\webapp\app\entities)
* Integration tests, to validate everything works as expected (src\test)
* Performance tests, to see if everything works smoothly

There are two ways to add entities and relations to the model with JHipster. You can use the command line tool and answer several questions. The JHipster site describes this on the [“Creating an Entity”](https://www.jhipster.tech/creating-an-entity/) page. A better experience is using the online JDL Studio and importing a JDL file.

### JDL Studio

Open <https://start.jhipster.tech/jdl-studio/> in the browser.

The JDL Studio opens with an elaborate example model. You can use this to see how the rest of the steps work or you remove everything and build your own model. Notice that all the way at the bottom, there are some instructions for pagination, dto’s (Data Transfer Objects) services and angular.

Use the options at the top right to explore the help and language reference. Notice that you can also download a snapshot image of the diagram.

Entity fields

entity A {

myField String required minlength(1) maxlength(42) pattern(/[A-Z]+/)

}

For each entity, you can add as many fields as you want. You will need to input the field names and their types, and JHipster will generate for you all the required code and configuration, from the Angular HTML view to the Liquibase changelog.

Those fields cannot contain reserved keywords in the technologies you are using. For example, if you use MySQL:

* You cannot use Java reserved keywords (as your code will not compile)
* You cannot use MySQL reserved keywords (as your database schema update will fail)

Field types

JHipster supports many field types. This support depends on your database backend, JDL uses Java types to describe them: a Java String will be stored differently in Oracle or Cassandra, and it is one of JHipster’s strengths to generate the correct database access code for you.

Validation

Validation can be set up for each field. Depending on the field type, different validation options will be available.

Validation will be automatically generated on:

* the HTML views, using the Angular or React validation mechanism
* the Java domain objects, using [Bean Validation](http://beanvalidation.org/)

Bean validation will then be used to automatically validate domain objects when they are used in:

* Spring MVC REST controllers (using the @Valid annotation)
* Hibernate/JPA (entities are automatically validated before being saved)

Validation information will also be used to generate more precise database column metadata:

* Required fields will be marked non-nullable
* Unique fields will create a unique constraint
* Fields which have a maximum length will have the same column length

Validation has a few limitations:

* We don’t support all validation options from Angular, React and Bean Validation, as we only support those which are common to both client and server APIs
* Regular Expression patterns don’t work the same in JavaScript and in Java, so if you configure one, you might need to tweak one of the generated patterns
* JHipster generates unit tests that work for generic entities, without knowing your validation rules: it is possible that the generated tests do not pass the validation rules. In that case, you will need to update the sample values used in your unit tests, so that they pass the validation rules.

## Import JDL

When you are done with the JDL Studio then click on the download button on the top right. Or even easier press CTRL+S to download the jhipster-jdl.jh.

You can keep the server and client still running while doing the JDL import, but that sometimes fails. For now stop the maven/gradle/yarn processes to be sure the sources are generated correctly. For some files it will ask to regenerate/overwrite the file. You can safely choose yes.

You can (re-)generate entities from a JDL file using the import-jdl sub-generator, by running

jhipster import-jdl ~/Downloads/jhipster-jdl.jh

* If you do not want to regenerate your entities, while importing a JDL, you can use the --json-only flag to skip entity creation part and create only the json files in .jhipster folder.
* By default import-jdl regenerates only entities which have changed, if you want all your entities to be regenerated then pass in the --force flag. Please note that this will overwrite all your local changes to the entity files

jhipster import-jdl ~/Downloads/jhipster-jdl.jh --force

Start the application again and mvnw or gradlew

Notice in the log “Liquibase has updated your database”

Login with the “admin” and observe all the changes

* The “Entities” menu has sub items. Explore the list and CRUD pages
* The Metrics dashboard is keeping an eye on entity services and rest endpoints
* The Logs dashboard is showing the loggers for the entity services and rest endpoints
* The API dashboard has the new rest endpoints which you can try out.
* And the database console has all the tables to browse through.

Now, you will have many new files. Let’s review them for a moment

* .jhipster/
  + This is a new directory which contains all the definitions of the JHipster model. These files will be used continuously by JHipster as reference. You can manually change them and generate the sources again.
* src/main/java/nl/amis/sigdemo/domain
  + New java entities are created. These classes have the correct annotations for JPA or the NoSQL database you selected when the application was created.
  + The validation annotations are present for server side validation
* src/main/java/nl/amis/sigdemo/repository
  + New java repositories are created, These interfaces extend the correct repository for JPA or the NoSQL database you selected when the application was created.
  + Some interfaces have method declarations based on the pagination settings
  + Notice these are interfaces. The spring-data libraries will generate the implementations on the fly when the application starts.
* src/main/java/nl/amis/sigdemo/service
  + This directory is created when you have instructed in the JDL to create dto’s and services
  + The DTO’s (Data Transfer Objects) have the same fields as the domain entities
  + The service interfaces have the CRUD methods for the entities
  + The service implementations convert the dto’s to domain entities
  + The mappers are doing the actual work of the conversion
* src/main/java/nl/amis/sigdemo/web/rest
  + The REST resources implemented with Spring Data Rest
  + Swagger-UI will display these new endpoints in the Administration API dashboard
* src/main/resources/config/liquibase/changelog
  + Liquibase changelog entries are used to alter the database to have the new tables
* src/main/webapp/app/entities/
  + All CRUD forms in Angular/React for all the entities are created.
* src/main/webapp/app/shared/model
  + The domain model entities created on the frontend
* src/main/webapp/i18n/
  + The language files for the entity forms
  + You still have to translate all entities for the non-English languages
* src\main\webapp\app\layouts\navbar\navbar.component.html
  + The entities are added to the “Entities” menu
* src/test/java/nl/amis/sigdemo/web/rest
  + Test classes for the REST controllers
* src/test/javascript/spec/app/entities
  + Component tests for the Angular/React frontend

## Test and QA

JHipster comes with an extensive set of tests, and each generated application has:

* Integration tests using the Spring Test Context framework.
* UI tests with [Jest](https://facebook.github.io/jest/).

### Running the test

Run the commands

./mvnw clean test

yarn test

This will run all the tests and the end should say “BUILD SUCCESS”

### Code quality

Code quality can be easily analyzed using [SonarCloud](https://sonarcloud.io/), which is automatically configured by JHipster.

Using Sonar with JHipster

For this you need docker and docker-compose installed

JHipster provides a specific Docker Compose configuration for Sonar ([here is the JHipster Docker Compose documentation](https://www.jhipster.tech/docker-compose/)) that provides an out-of-the box Sonar instance. At the root of your project, please run:

docker-compose -f src/main/docker/sonar.yml up –d

If you use Maven, it has been automatically configured:

./mvnw clean test sonar:sonar

If you use Gradle, it has also been automatically configured:

./gradlew clean test sonarqube

Once the analysis completes, it will be available on the Sonar dashboard, which by default is available on <http://127.0.0.1:9000/>.

## Setting up Continuous Integration

The generated application does not come with Continuous Integration files by default. A JHipster application is harder to setup than a normal spring-boot or frontend project. The CI systems needs to build two software stacks and bundle them together.

* the Java back-end code with Maven or Gradle
* the JavaScript front-end with NodeJS, NPM and Webpack

Each stack comes with its own dependency management (Maven artifacts, NPM packages) with potential conflicts to solve.

JHipster should support the following CI systems out of the box:

* Jenkins:
  + [Setting up Jenkins 1](https://www.jhipster.tech/setting-up-ci-jenkins1/)
  + [Setting up Jenkins 2](https://www.jhipster.tech/setting-up-ci-jenkins2/) (recommended)
* Travis: refer to the [Travis Documentation](https://docs.travis-ci.com/user/getting-started/)
* CircleCI: refer to the [CircleCI Documentation](https://circleci.com/docs/getting-started/)
* GitLab CI: refer to the [GitLab CI Documentation](https://about.gitlab.com/gitlab-ci/)

To generate these config files, run this command in your project folder:

jhipster ci-cd

Then answer all the questions.

**Note**: when you select Jenkins pipeline, a new src/main/docker/jenkins.yml file will be generated. So you can test Jenkins locally by running:

docker-compose -f src/main/docker/jenkins.yml up –d

Then create a Pipeline project as described on [Setting up Jenkins 2](https://www.jhipster.tech/setting-up-ci-jenkins2/)

## Using JHipster in production

JHipster generates a fully production-ready, optimized and secured application.

### Building an executable WAR file

To package the application as a “production” WAR, with Maven please type:

./mvnw -Pprod package

Or when using Gradle, please type:

./gradlew -Pprod bootWar

This will generate two files in the “target” or “build” directory (if your application is called “myapplicaton”):

* myapplicaton -0.0.1-SNAPSHOT.war
* myapplicaton -0.0.1-SNAPSHOT.war.original

The first one is an executable WAR file (see next section to run it). It can also be deployed on an application server, but as it includes runtime libraries, we recommend you use the second, .original file if you want to deploy JHipster on an application server like Tomcat, Weblogic or Websphere.

### Running in production

Instead of deploying to an application server, many people find it easier to just have an executable WAR file. The first WAR file generated in the previous step is such a WAR, so you can run it in “production” mode by typing (on Mac OS X or Linux):

./jhipster-0.0.1-SNAPSHOT.war

If you are on Windows, use:

java –jar -Dspring.profiles.active=prod jhipster-0.0.1-SNAPSHOT.war

**Please note** that this WAR file uses the profile we selected when building it. As it was built using the prod file in the previous section, it will therefore run with the prod profile.

The War fill only run successfully in production when your selected data store is up and running. JHIpster generates a docker-compose file for your data store. To run the data store execute

docker-compose -f src/main/docker/mysql.yml up –d

### Running the application in a Docker container

When generating your application, JHipster generates for you:

* A Dockerfile for building a Docker image and running your application inside a container
* Several Docker Compose configurations to help you run your application with third-party services, for example a database

Those files are located inside folder src/main/docker/.

To create a Docker image of your application, and push it into your Docker registry:

* With Maven, type: ./mvnw package -Pprod dockerfile:build
* With Gradle, type: ./gradlew bootWar -Pprod buildDocker

This will package your application with the prod profile, and install the image.

On Windows, due to [lack of named pipes](https://github.com/spotify/docker-client/issues/875), you may have to tune settings for Docker and turn on “Expose daemon on tcp://localhost:2375 without TLS”.

To run this image, use the Docker Compose configuration located in the src/main/docker folder of your application:

docker-compose -f src/main/docker/app.yml up

This command will start up your application and the services it relies on (database, search engine …).

### Running in the cloud

JHipster can help to move your application into a cloud.

* Cloud Foundry
* Heroku
* Kubernetes
* OpenShift
* Rancher
* AWS

Visit <https://www.jhipster.tech/production> and checkout the different section how to deploy to each cloud.