JHipster 4 Workshop

Introduction

Welcome

- Introduction
- Agenda
- Practical details

Introduction

- Who we are

Introduction

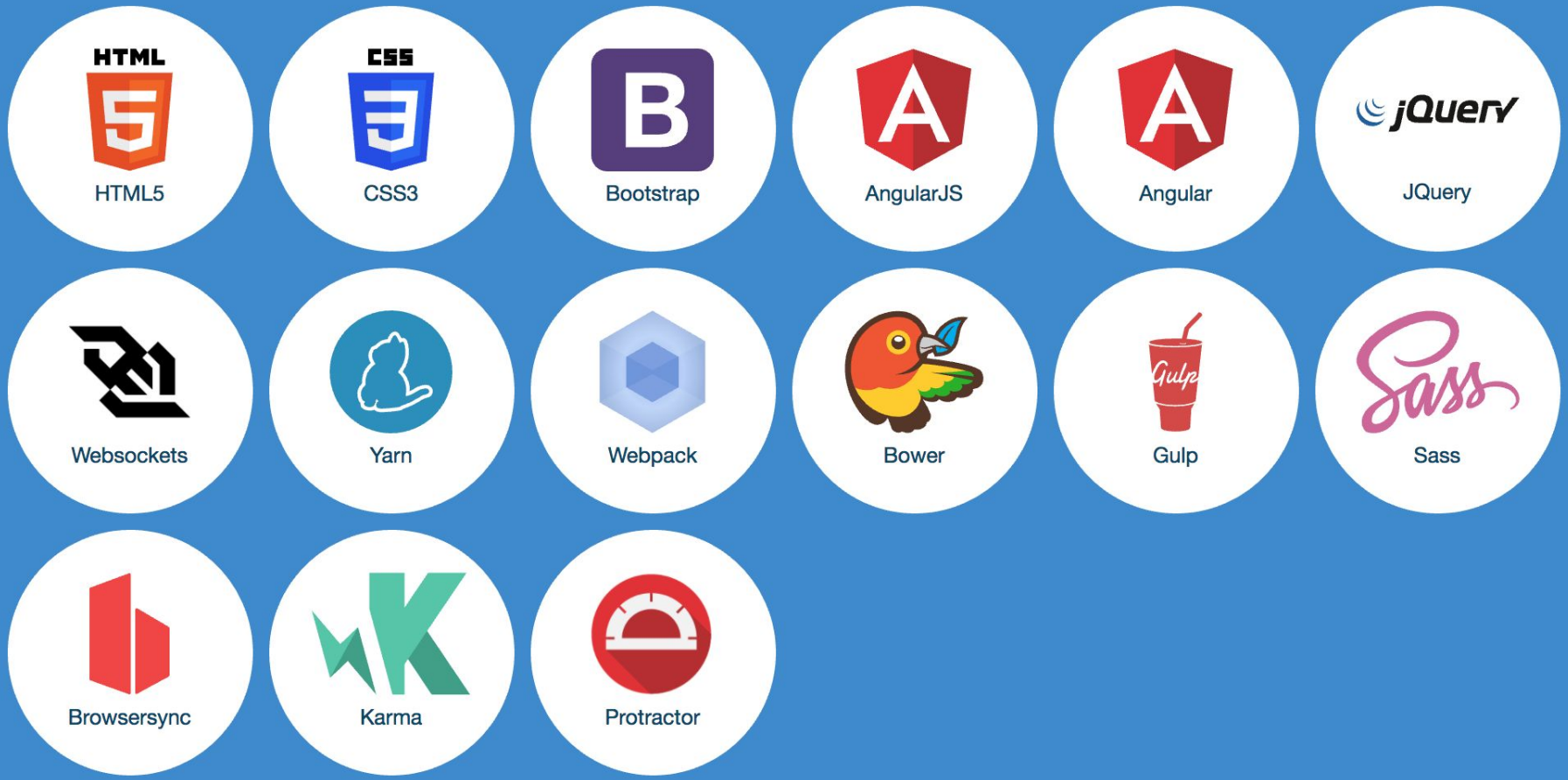
- Who you are

Overview of the workshop

- Goal: being able to set up a complete application with JHipster
  - ➔ With a good understanding of the concepts
- Theory (slides) and hands-on labs

Server side frameworks

Client side frameworks



Versions of the frameworks & tools

- This workshop is based on the following versions
  - ➔ JHipster 4.x
  - ➔ Spring Boot 1.5.x with Spring Data
  - ➔ Yeoman 1.8.x
  - ➔ Webpack 2.2.x
  - ➔ Angular 2

Agenda

- Angular 2 Intro
- Introduction to JHipster
- Basic technologies used
- The main generator
- The project structure
- The “entity” sub-generator
- Available development workflows
- Database access
- Security
- Testing
- Deploying to production
- Doing microservices

Practical details - The VM

- Warning: setting up your environment can take some time
  - ➔ You should do this before the workshop
  - ➔ You need a good network connection
- JHipster can be installed on your machine (better but more complicated)
  - ➔ Follow the instructions on https://jhipster.github.io/installation.html
  - ➔ We will use MySQL during the workshop, so you need to install it from http://dev.mysql.com/downloads/
- JHipster can be used within a Virtual Machine (easier to setup)
  - ➔ Install VirtualBox https://www.virtualbox.org/
  - ➔ Install Vagrant from https://www.vagrantup.com/
  - ➔ Follow the setup instructions from https://github.com/jhipster/jhipster-devbox
    - Use the “Quick setup”, using the distribution available on Atlas
    - Modify your VirtualBox configuration to have as much RAM and CPU as possible
    - Change your keyboard settings if you need to

Practical details - checking your development machine

- Do you have an IDE working?
  - ➔ If you use the JHipster “devbox” Virtual Machine, you have Spring Tool Suite installed, just click on the “STS” link on your Desktop
- Do you have MySQL installed?
- Does “yo doctor” work?
- Does “java -version” give you a Java version >= 1.8.0?

Introduction to JHipster

What is JHipster

- An application generator
- Spring Boot + Angular + tooling
- 100% free and Open Source
- Available on Github at https://github.com/jhipster/generator-jhipster

  ➔ Come and join us!

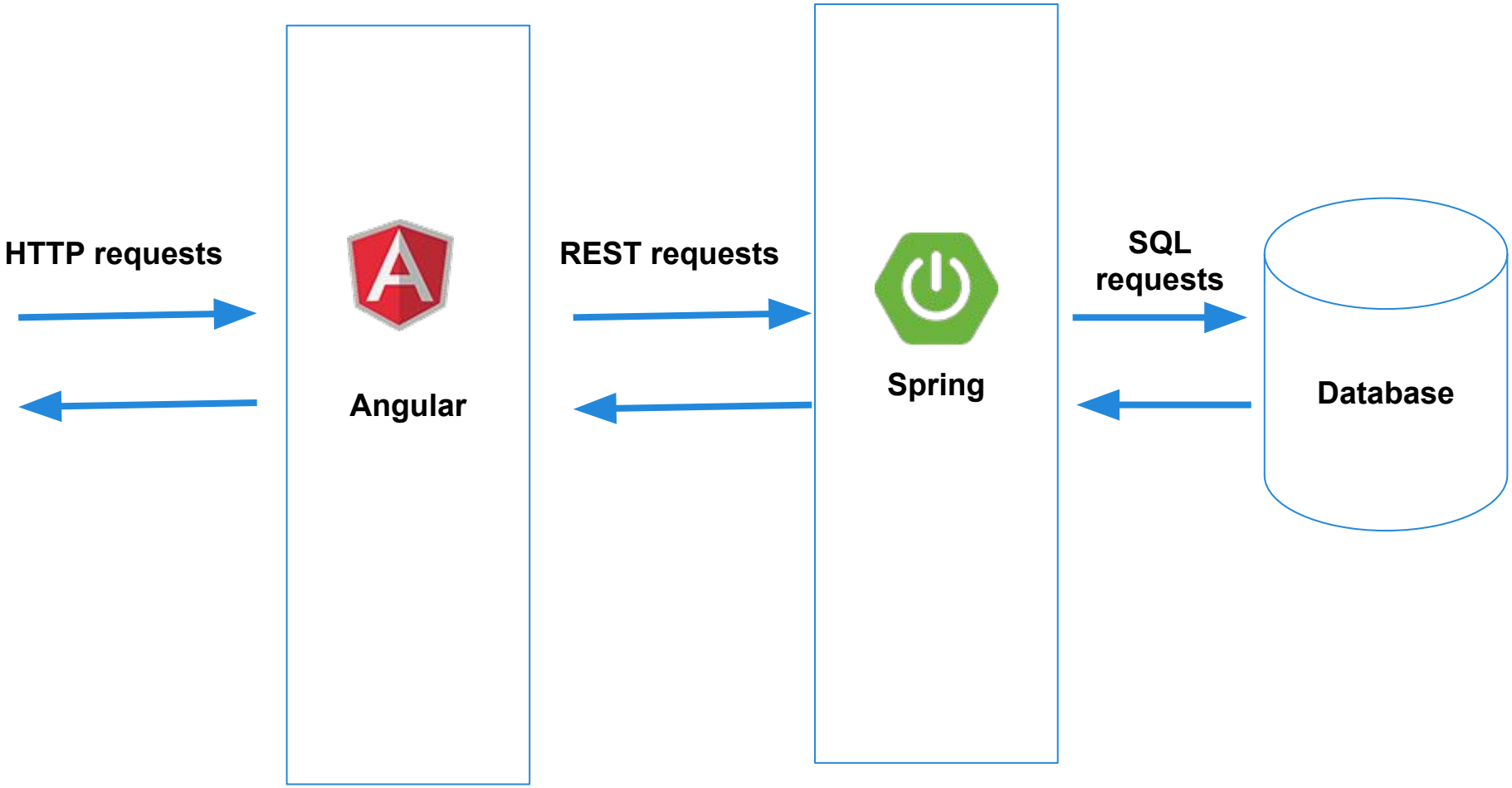
What is it good for?

- Business applications
  - ➔ Initial focus
  - ➔ Single Web Page Application
  - ➔ Form-based, CRUD applications
- Web applications
  - ➔ More and more used for B2C Webapps
  - ➔ Great UI and back-end performance make it a compelling solution
- Mobile applications
  - ➔ Responsive UI
  - ➔ With a native UI accessing the REST API

Architecture

- The classical JHipster architecture is that of a Single Page Application
  - ➔ Your database may vary (supports SQL, MongoDB, Cassandra)
  - ➔ Your services may vary (Elasticsearch, caching…)
- A main, static HTML page is loaded
  - ➔ Launches an Angular application
  - ➔ This application does REST requests to the back-end
  - ➔ The back-end handles those requests, add business services and security
  - ➔ And data is stored in a database

Architecture



What is being generated?

- The application skeleton
  - ➔ Spring Boot on the back
  - ➔ Angular on the front
  - ➔ Tooling is configured so you can get started quickly
- You can also use “sub-generators” to add more features
  - ➔ The main one is the “entity sub-generator” that generates CRUD interfaces

Configured tooling

- JHipster comes packed with tools to ease your development workflow
  - ➔ The main goal of this masterclass is to learn how to use them efficiently
- Server-side tools
  - ➔ Spring Boot adds a lot of features, including “hot reload”!
  - ➔ Liquibase for database updates
- Client-side tools
  - ➔ Yarn, Webpack & BrowserSync for “live reload” of your front-end application
- Many “dotfiles” for your IDE or Git
- Docker files for running services in a container

Generating your application graphically

- JHipster builds on top of Yeoman
  - ➔ A question/answer system to generate your application
  - ➔ Very powerful, but can be annoying if you have a lot of things to generate
- Applications can be generated from a standard .xmi file (coming from your own UML editor)
  - ➔ You develop your model graphically using an UML editor, and export it as an .xmi file
  - ➔ The JHipster UML project is then able to generate the entities automatically
  - ➔ More information at: http://jhipster.github.io/jhipster-uml/
- The free, online "JDL Studio" can be used to generate the application using a domain-specific language
  - ➔ http://jhipster.github.io/jdl-studio/
  - ➔ Download the generated file and run it with JHipster

JHipster best practices

- Follow each underlying technology “best practices”
  - ➔ Angular best practices, Spring best practices, etc…
  - ➔ Only change them if there is a good reason to
- Use the provided library versions
  - ➔ It’s hard work to have them all working together
- Follow the workflows provided by JHipster
  - ➔ They are here to help you
  - ➔ There is usually a very good reason to use them in the recommended way
- You have a great working environment out-of-the-box, don’t break it!
  - ➔ Front-end and back-end updates should be automatic and fast
  - ➔ Production deployment should be easy
- Git is your friend
  - ➔ Every mistake (including in the database!) should be easy to rollback with Git

Basic technologies used

Spring Boot

- Hugely popular, on top of the hugely popular Spring Framework
- Spring Boot helps configuring and getting started with Spring
  - ➔ Convention over configuration
  - ➔ Automatic configuration, with clean and documented ways to adapt it to your needs
- JHipster doesn't get in the way of Spring Boot
  - ➔ Configures it, so that it works great with Angular
  - ➔ Builds on top of it and adds new features: mainly for management and security
  - ➔ Very scarcely, corrects some Spring Boot issues, but tries as much as possible to give those back to Spring Boot

Spring Data

- One of the many Spring sub-projects, integrated in a Spring Boot starter
- Greatly simplifies the way JPA works with a dedicated module
  - ➔ Automatic query generation, based on interface method names
- In a typical JHipster application, you will need to create those queries yourself
  - ➔ It's important to understand Hibernate/JPA
  - ➔ At first Spring Data JPA looks a bit complex, but when you are used to it, it's really quick and efficient
- If you use a NoSQL database
  - ➔ MongoDB: Spring Data MongoDB works the same (there even are portable methods, which work for both technologies!)
  - ➔ Cassandra: it was deliberately chosen not to use Spring Data Cassandra, mainly for performance and stability reasons

Spring MVC REST

- The de-facto standard REST framework in Java
  - ➔ Great performance, lots of documentation on the Internet
  - ➔ Very close to most other REST frameworks anyway
- JHipster creates “CRUD” operations automatically
  - ➔ GET, PUT, POST, DELETE verbs from HTTP
  - ➔ JSON objects with Jackson

Spring Security

- One of the oldest Security frameworks in Java
  - ➔ Widely used
  - ➔ Lots of plugins and features
- Secures both URLs and Java methods
  - ➔ Use both features together for extra security!
- JHipster adds a lot of features on top of it
  - ➔ Several different authentication schemes
  - ➔ Great UI with Angular
  - ➔ Complete database schema
  - ➔ Added features, such as invalidation of the persistent tokens

Angular

- Hugely popular JavaScript framework
- Easy to use at first, but hard to have a good modular design
- JHipster provides the architecture and many building blocks
  - ➔ Clean and modular design, which allows your application to grow without trouble
  - ➔ REST clients that connect to what Spring MVC REST provides
  - ➔ Built-in support for Spring Security
  - ➔ i18n, both on the client and on the server side
- By default, JHipster generates a Single Page Application that provides a rich and fluid user experience

Bootstrap

- Hugely popular JS/CSS framework
- JHipster uses its classes extensively
  - ➔ Responsive UI out-of-the-box
  - ➔ Tables, alerts, popups, menus… all come from Bootstrap
- Great for Web Designers
  - ➔ As JHipster uses Bootstrap correctly, it’s easy to add a new theme on top of a JHipster application
  - ➔ Modify the provided theme or install a new one

Yeoman

- The tool that is used to create JHipster
- As a JHipster user, you will mostly see it when you call “yo”
- Provides many interesting features for advanced users
  - ➔ Configuration files are easy to modify
  - ➔ Re-generating a project is seamless
  - ➔ Templates are great (and have been improved by the JHipster team!)

Maven & Gradle

- Your build tools for the Java world
- JHipster provides wrappers for both, so you don't have to install them
  - ➔ Best practice: each build tool has its own version, to guarantee an immutable build
  - ➔ Of course you can still install them if you want to!
- JHipster provides libraries and plugins versions, all configured
  - ➔ Extends Spring Boot, which also provides those versions
  - ➔ Only modifies a version on top of Spring Boot when there is a good reason to do so
  - ➔ Think carefully before upgrading a version, as both the Spring Boot team and the JHipster team have validated it!
- Public repositories can be searched for installing/updating libraries
  - ➔ http://search.maven.org/
  - ➔ https://repository.sonatype.org/

The main generator

Available options

- JHipster provides many options when creating an application
  - ➔ It can generate an application tailored to your specific needs
  - ➔ When in doubt, use the default options
- The most important options are:
  - ➔ Client framework: AngularJS or Angular
  - ➔ Type of authentication
  - ➔ Database to be used in production and in development
  - ➔ Hibernate cache
  - ➔ Usage of Elasticsearch
  - ➔ Build tools that should be configured: Maven/Gradle
  - ➔ i18n
  - ➔ Testing frameworks

Generated files

- JHipster generates a lot of files
  - ➔ The node_modules directory is used internally by NPM, you should ignore it
  - ➔ Beside this, it is a classical Maven project structure
    - With “src/main/java” and “target” directories, as usual
  - ➔ In the “src/main/webapp” folder you will find the whole Web application, including the Angular 2 application
- The Spring Boot application has the usual Java package hierarchy
  - ➔ “domain” for your domain objects, “repository” for your repositories, “web/rest” for you REST endpoints
  - ➔ Layers are separated by technical concerns, and not by business concerns
- The Angular application is separated in modules
  - ➔ Follows the Angular style guide
  - ➔ https://angular.io/docs/ts/latest/guide/style-guide.html

The .yo-rc.json file

- This is the internal Yeoman configuration file
  - ➔ You can tweak it manually if you want to re-generate your application
  - ➔ You should send it to the team if you have an issue
- JHipster also has its own configuration files in the .jhipster directory
  - ➔ At the moment, this is only for generating entities
  - ➔ This is what JHipster UML generates

How to start correctly a new project

- Create an empty directory in which you will work
- Generate the application
- Test if everything is OK
  - ➔ run “mvn test” for the back-end
  - ➔ run “yarn test” for the front-end
- Commit it to Git
  - ➔ JHipster generates a .gitignore file for you, so you can commit your files straight after generation
- Import it into your IDE
  - ➔ Spring Tool Suite, Eclipse, Intellij IDEA, should all work
  - ➔ More information on your IDE configuration at http://jhipster.github.io/configuring_ide.html

How to update an existing project

- If you run again “yo jhipster” or “yo jhipster:entity” on an existing project, JHipster will re-use its configuration files to re-generate the project
  - ➔ If you have upgraded to a new JHipster version, you will get newer versions of your files
- The trouble when updating an existing project is that JHipster will overwrite all files
  - ➔ Yeoman provides a mechanism for this, that will warn you and allow you to make a diff to see what’s being changed
  - ➔ But for big updates, this isn’t enough
- The best solution is to use Git and the Git tools to do the merging
  - ➔ We recommend using a graphical tool, like SourceTree: https://www.sourcetreeapp.com/

Lab 1: setting up a project

The sample application

- The sample application is a Bug Tracker
- It can handle several projects
- Each project has several tickets
- Tickets can have labels, like “bug” or “enhancement”
- Tickets are assigned to users

Your first JHipster project (1/2)

- Create a new directory called “BugTracker”
- Run “yo jhipster” in that directory
  - ➔ When the database is selected, choose a MySQL in development and in production
  - ➔ When the client framework is selected, choose Angular 2.x
  - ➔ Do NOT enable internationalization support
  - ➔ Keep all other options as default
- Validate your project
  - ➔ “./mvnw test” for the server side
  - ➔ “yarn test” for the client side
- Open up your application on your IDE
  - ➔ If you use the JHipster “devbox”, the shortcut is on your desktop
  - ➔ Configure your IDE in order to import the project

Your first JHipster project (2/2)

- Run MySQL
  - ➔ The recommended way is to use Docker:

    docker-compose -f src/main/docker/mysql.yml up
  - ➔ You can also install MySQL manually:

    apt-get install mysql
- Create a MySQL schema
  - ➔ The schema name is by default your application's name
- Configure your "application-dev.yml" to point to your MySQL schema
- Run your application
  - ➔ Run the "Application" class
  - ➔ Go to your application at http://127.0.0.1:8080
  - ➔ Check your database schema

Project structure

Java project structure

- JHipster generates a “standard” Maven project
  - ➔ Usual Maven project structure
- Java packages match the usual technical layers
  - ➔ Configuration
  - ➔ Repositories
  - ➔ Domain objects
  - ➔ Services
  - ➔ REST endpoints

Configuration files

- Configuration files are generated in the “src/main/resources/config” directory
- Spring Boot is configured using the “application.yml” files
  - ➔ Uses the YAML format
  - ➔ Fully documented (and your IDE might even be able to autocomplete them!)
    - See http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html
  - ➔ One base “application.yml” file, and then one file per Spring profile
  - ➔ Each key can be overridden using an environment variable
- Liquibase is configured in the “src/main/resources/config/liquibase” subdirectory
  - ➔ One master file that points to several changelogs
  - ➔ Automatically updated when running “yo jhipster:entity”
  - ➔ Will require editing when running “mvn liquibase:diff”

Web project structure

- The Web project is in the “src/main/webapp” folder
  - ➔ Contains many best practices for Web development
- “app” contains the Angular application
- “content” contains static resources, such as images and CSS files
- Note that a “target/www” folder will be created during the project’s production packaging, and will contain a minified version of the web project

Angular project structure

- “app” contains the main application
- “app/account” contains the user account management UI
- “app/admin” contains the administration UI
- “app/blocks” contains common building blocks like configuration and interceptors
- “app/home” contains the home page
- “app/shared” contains common services like authentication and internationalization
- “app/entities” is where your entities will get generated
- “app/layouts” contains UI code such as the navigation bar and error pages

The “entity” sub-generator

Generating a simple CRUD entity

- The “entity” sub-generator can be run in the application folder, to add entities to the application
  - ➔ Run “yo jhipster:entity Ticket” to generate the “Ticket” entity
- It generates a “domain object”, as well as all the required infrastructure, back-end and front-end code
  - ➔ Liquibase changelog
  - ➔ JPA entity
  - ➔ JPA repository
  - ➔ Spring MVC REST controller with GET, PUT, POST, DELETE methods
  - ➔ Angular and HTML code
  - ➔ Tests
- This generation depends on your application’s main configuration
  - ➔ Supports SQL, Cassandra, MongoDB
  - ➔ Supports Elasticsearch…

Field types

- Many field types are provided out of the box
  - ➔ For each field, you will be asked to define its type, validation rules, pagination…
  - ➔ Each type is configured to be used in HTML, in Java, and in the database
- Available types are:
  - ➔ String
  - ➔ Numeric types: Integer, Long, Float, Double, BigDecimal
  - ➔ Date types (from the Java 8 Date/Time API): LocalDate, ZonedDateTime
  - ➔ Boolean
  - ➔ Enum (which generates a Java enum type)
  - ➔ Blob, which can handle both a generic binary file or an image
- If you need a type that doesn't exist yet, use the closest available type (or use "String") and then do a refactoring with your IDE

Validation

- For each field type, validation is available
- On the client-side, the validation system from Angular 2 is used
  - ➔ Works with Bootstrap to provide a nice UI
- On the server-side, Bean Validation works with Spring MVC REST
  - ➔ Bean validation is executed automatically by Spring MVC REST when the @Valid annotation is present at the controller level
- We try to have a coherent validation between the client and the server side
  - ➔ Not always possible, by example for Regexp patterns
  - ➔ Forces JHipster to use the “lowest common denominator”
  - ➔ For advanced usage, you will need to tweak the generated code manually

Relationship types

- Relationships only work if you have selected an SQL database
- JHipster supports relationships generation between entities
  - ➔ one-to-one
  - ➔ one-to-many
  - ➔ many-to-one
  - ➔ many-to-many
- Those relationships can be bidirectional or not
- Relationships rely on database (and JPA) relationships
  - ➔ generates the Liquibase and JPA code
  - ➔ also generates specific Angular 2/HTML front-end code
- Warning! Don't forget JPA relationships are lazy by default
  - ➔ JHipster uses the underlying technology's default configuration
  - ➔ Use eager relationships or transactions to solve this common issue with JPA

Entity creation order

- Relationships force the creation of some entities before others
- This comes from the database
  - ➔ You can't add a foreign key pointing to a table that doesn't exist yet
- For a one-to-many relationship
  - ➔ You must first do the "one" side (as the foreign key is on the "many" side)
- For a many-to-many relationship
  - ➔ You must first build the entity that doesn't own the relationship

The “User” entity

- The “User” entity is a specific entity, as it is already created by JHipster
  - ➔ It is used by JHipster for authentication (with Spring Security)
  - ➔ It can be modified as any generated Java code
  - ➔ There is an administration UI specifically provided out of the box for this entity, to manage users
- A “many-to-one” relationship can be added to the “User” entity
  - ➔ Example: several tickets are assigned to one user

Pagination options

- Pagination is optional
- On the server-side, a “GET” request to get all objects has by default a pagination of 20 objects
  - ➔ Can be configured through Spring MVC REST/Spring Data
- On the client-side, 3 different options are available
  - ➔ Simple pager
  - ➔ Pagination links
  - ➔ Infinite scrolling

Lab 2: generating entities

Entity 1: the “Project”

- Create an entity called “Project”: yo jhipster:entity project
- It has only one field, “name”, of type “String”
- Hit “compile” on your IDE, everything should be automatically updated
- Run your application
  - ➔ Either from your IDE like in the previous lab, or type “./mvnw”
  - ➔ Run Browsersync by typing “yarn start”, this will open up a Web browser using http://localhost:9000

Entity 1 solution

.jhipster/Project.json

```
{
   “fluentMethods”: true,
   "relationships": [],
   "fields": [
      {
         "fieldName": "name",
         "fieldType": "String"
      }
   ],
   "changelogDate": "20151118144708",
   "dto": "no",
   “service”: “no”,
   “entityTableName”: “project”,
   "pagination": "no"
}
```

Entity 2: the “Label”

- Create an entity called “Label”: yo jhipster:entity label
- It has only one field, “label”
  - ➔ it is of type “String”
  - ➔ it must be validated: it is a required field, with a minimum length of 3
- It has a relationship to the entity “Ticket”
  - ➔ It is a many-to-many relationship
  - ➔ The label is the non-owning side
- Hit “compile” on your IDE, **it is going to FAIL** as the “Ticket” entity doesn’t exist yet

Entity 2 solution

.jhipster/Label.json

```
{
  “fluentMethods”: true,
  "relationships": [
    {
      "relationshipName": "ticket",
      "otherEntityName": "ticket",
      "relationshipType": "many-to-many",
      "ownerSide": false,
      "otherEntityRelationshipName": "label"
    }
  ],
  "fields": [
    {
      "fieldName": "label",
      "fieldType": "String",
      "fieldValidateRules": [
        "required",
        "minlength"
      ],
      "fieldValidateRulesMinlength": "3"
    }
  ],
  "changelogDate": "20151118145609",
  "dto": "no",
  “service”: “no”,
  “entityTableName”: “label”,
  "pagination": "no"
}
```

Entity 3: the "Ticket"

- Create an entity called "Ticket"
  - ➔ If you make a mistake during those questions/answers, you will need to restart the entity creation
- It has several fields
  - ➔ "title", of type String, which is required
  - ➔ "description", of type String
  - ➔ "dueDate", of type LocalDate
  - ➔ "done", of type Boolean
- It has several relationships
  - ➔ with "Project", it has a many-to-one relationship, using the Project "name" to display the relationship
  - ➔ with "User", it has a many-to-one relationship named "assignedTo", using the User "login" to display the relationship
    - The "User" entity is generated by default by JHipster
  - ➔ with "Label", it has a many-to-many relationship, which it owns, using the Label "label" to display the relationship
- It uses pagination links

Entity 3 solution

```
{
  “fluentMethods”: true,
  "relationships": [
    {
      "relationshipName": "project",
      "otherEntityName": "project",
      "relationshipType": "many-to-one",
      "otherEntityField": "name"
    },
    {
      "relationshipName": "assignedTo",
      "otherEntityName": "user",
      "relationshipType": "many-to-one",
      "otherEntityField": "login",
      “ownerSide”: true,
      “otherEntityRelationshipName”: “ticket”
    },
    {
      "relationshipName": "label",
      "otherEntityName": "label",
      "relationshipType": "many-to-many",
      "otherEntityField": "label",
      "ownerSide": true,
      “otherEntityRelationshipName”: “ticket”
    }
  ],
  "fields": [
    {
      "fieldName": "title",
      "fieldType": "String",
      "fieldValidateRules": [
        "required"
      ]
    },
    {
      "fieldName": "description",
      "fieldType": "String"
    },
    {
      "fieldName": "dueDate",
      "fieldType": "LocalDate"
    },
    {
      "fieldName": "done",
      "fieldType": "Boolean"
    }
  ],
  "changelogDate": "20151118154200",
  "dto": "no",
  “service”: “no”,
  “entityTableName”: “ticket”,
  "pagination": "pagination"
}
```

.jhipster/Ticket.json

Available development workflows

Running the Java application

- The Java application can be run in three ways
  - ➔ mvn
  - ➔ run the “Application” class
  - ➔ run the application as a “Spring Boot application” if your IDE supports Spring Boot
- It can be run with 2 profiles
  - ➔ A profile is a Spring Profile as well as a Maven Profile (JHipster generates those consistently)
  - ➔ The “dev” profile is the one selected by default, and is used for development
  - ➔ The “prod” profile is for production use
- The “dev” profile has Spring Boot devtools enabled
  - ➔ Uses a specific classloader to restart the application faster
  - ➔ This is a must-have feature, you have to use it!

Using Maven (or Gradle)

- “mvn” runs the default “mvn spring-boot:run” goal
  - ➔ Runs the application
- “mvn clean” will cleans the application’s “target” directory
  - ➔ Cleans up Elasticsearch indexes and H2 disk-based persistence database
- “mvn compile” compiles the classes
  - ➔ Works with Spring Boot devtools if you have “mvn” running in another terminal
- “mvn test” runs the tests
  - ➔ Unit tests and integration tests

Using Yarn

- “yarn start” runs the browser on port 9000 by default
  - ➔ Automatically watch files
  - ➔ Runs the front-end application with Browsersync
- “yarn test” runs the front-end tests
  - ➔ With karma.js and phantom.js

Working with BrowserSync

- Browsersync is a great tool for Web development
  - ➔ http://www.browsersync.io/
- It synchronizes different browsers
  - ➔ For testing different browsers or resolutions at the same time
  - ➔ Also works from external devices: control one or several phones automatically
- Synchronizes your behavior: clicks, scrolls, form inputs…
- Refreshes the application when a file changes
  - ➔ HTML, CSS, and JavaScript
  - ➔ No more manual browser refresh!

Database updates with Liquibase

- Liquibase versions databases changes
  - ➔ Very handy to update and maintain a database's structure
  - ➔ Allows to have several people work on the same database together
- JHipster generates Liquibase changelogs for you
  - ➔ When the main generator is run
  - ➔ When the entity sub-generator is run
- JHipster also configures the Maven Liquibase Hibernate plugin according to your current database state
  - ➔ The plugin will connect to your database and generate a changelog between your current JPA configuration and your database schema
  - ➔ Works well with MySQL or Postgresql database (support for disk-based H2 too)
  - ➔ Edit a JPA entity, compile it, and run "mvn liquibase:diff" to have a new changelog in "src/main/resources/config/liquibase/changelog". You will need to include it in the "src/main/resources/config/liquibase/master.xml" file.

Lab 3: common workflows

Workflow 1: updating client-side code

- Go to your application's tickets page at http://localhost:9000/#/tickets
- With your IDE, look for the ticket.component.html page
  - ➔ Remove the "Description" column
  - ➔ Change the format of the "DueDate" to "MM/dd/yyyy"
  - ➔ Each time you change this file, your browser should refresh automatically

Workflow 2: Liquibase

- Add a new field to the “Ticket” entity
  - ➔ A ticket has a “priority” which is an integer
  - ➔ Add this field to the Ticket entity, and generate its getter/setter methods
  - ➔ Compile the Ticket class
  - ➔ Open up your “pom.xml” file and configure your Liquibase plugin to be sure it points to your MySQL database
    - Check your schema name, login and password
  - ➔ Run “mvn liquibase:diff”
  - ➔ Have a look at the generated changelog, and add it to the master.xml
  - ➔ Compile the application again to make Liquibase execute the changelog
  - ➔ Use MySQL workbench to check that the field has been added to the database
- (optional) modify the user interface to see the priority field
  - ➔ In the “ticket-dialog.component.html” file, add a new text input (re-use description)
  - ➔ In the “ticket.component.html” file, add a new “priority” column which will display the priority

Database access

Spring Data JPA

- Spring Data JPA provides many helpers methods to ease working with JPA/Hibernate
  - ➔ Simple CRUD repositories are automatically provided
  - ➔ More complex queries can be generated dynamically using method names
  - ➔ Provides pagination and ordering API
- For example, to have all the “Foos” ordered by the “bar” field:
  - ➔ fooRepository.findAllByOrderByBarDesc
- More information on the Spring Data JPA project at http://projects.spring.io/spring-data-jpa/

Hibernate 2nd level cache

- Using a 2nd level cache is the top priority if you want good performance using JPA
- JHipster provides 3 options
  - ➔ No cache
  - ➔ Ehcache (the default)
  - ➔ Hazelcast (a distributed cache)
- Caches will be generated using default values, which are quite low for production
  - ➔ You need to tune them, depending on your usage and available RAM
  - ➔ Ehcache is monitored, and can be observed in the application's administration/monitoring screen: use it to tune your cache configuration accordingly

MongoDB

- MongoDB is a NoSQL database, mainly used to store JSON documents
  - ➔ More information at https://www.mongodb.org/
- JHipster supports MongoDB
  - ➔ Using Spring Data MongoDB
  - ➔ Using Mongeez for database upgrades (instead of Liquibase)

Cassandra

- Cassandra is a NoSQL database providing high scalability and availability
  - ➔ More information at http://cassandra.apache.org/
- JHipster supports Cassandra
  - ➔ Using directly the DataStax Java Driver for Cassandra

Lab 4: database access

Working with Spring Data JPA

- We want to have the most urgent tickets on top of the list
  - ➔ Currently they are not ordered
- Modify the TicketResource REST controller to use a new Spring Data repository method
  - ➔ Find the right method name to have all the entities, ordered by due date (descending)
  - ➔ Use your IDE to generate this method in the repository
  - ➔ Thanks to the Spring Boot devtools, your application should restart automatically (and very quickly!) with your new repository method

Solution

TicketResource.java

```
Page<Ticket> page =

    ticketRepository.findAllByOrderByDueDateDesc(pageable)
```

Security

Available options

- For authentication and authorization, JHipster configures Spring Security, the corresponding database schema and user interface
- 4 options are currently available:
  - ➔ Session-based authentication
  - ➔ Session-based authentication with social login enabled
  - ➔ OAuth2
  - ➔ JWT
- OAuth2 and JWT are stateless mechanisms, and can be easier to scale on a cluster

Remember-me

- "Remember-me" is a Spring Security option that allows to a user to stay connected for 30 days
  - ➔ Works for session-based and JWT authentications
- For session-based authentication, JHipster adds many features on top of what is provided by Spring Security
  - ➔ Tokens are far more secure than the default Spring Security implementation
  - ➔ Database table to store the user's tokens
  - ➔ User interface in Angular 2, including support for invalidating current tokens

CSRF protection

- CSRF attacks use a forged link to your application in order to execute commands, using your current authentication
  - ➔ With JHipster, this could happen if you have session-based authentication: the attacker would use your session cookie to be authenticated
- JHipster uses Spring Security's CSRF protection to avoid this issue
  - ➔ A token is sent in the browser's headers, and must be validated
  - ➔ Requests using the HTTP GET verb are not validated, as they are not modifying anything
  - ➔ Angular 2 is specifically configured to understand the same token as JHipster

Using the “User” entity

- The User entity is a specific entity generated by JHipster
  - ➔ It represents a “User” domain object
  - ➔ It is used by Spring Security to check the user’s authentication and authorizations
- It can be used throughout your application
  - ➔ More fields can be added
- A many-to-one relationship to the “User” can be created
  - ➔ A specific Spring Data JPA query is generated so the other entity can be filtered on the current user (a common use case)

Lab 5: security

Securing methods

- Only the “admin” users should have the permission to delete existing tickets
  - ➔ Add a security annotation on the TicketResource REST controller for users with the “ROLE_ADMIN” permission only can delete the resource
  - ➔ Hit “compile”, your application should restart automatically
  - ➔ Log in the application with “user/user”, and try to delete a ticket: you should have an HTTP 403 (forbidden) error
- (Optional) Remove the “delete” button when the user doesn’t have the ROLE_ADMIN permission
  - ➔ Hint: use “jhiHasAnyAuthority”

Solution

TicketResource.java

```
@Secured(AuthoritiesConstants.ADMIN)
 public ResponseEntity<Void> deleteTicket(...)
```

ticket.component.html

```
<button type="submit"
                [routerLink]="['/', { outlets: { popup: 'ticket/'+ ticket.id + '/delete'} }]"
                replaceUrl="true"
                class="btn btn-danger btn-sm"
                jhiHasAnyAuthority="ROLE_ADMIN" >
```

Testing

Testing with Spring

- On the server side, JHipster provides
  - ➔ Unit tests using vanilla JUnit
  - ➔ Integration tests using Spring's test context framework
- Integration tests provide
  - ➔ An in-memory database, with the Liquibase changelogs applied
  - ➔ Tooling to test the application's REST endpoints
- Those tests are located in "src/test/java"
  - ➔ They can be run inside an IDE
  - ➔ "mvn test" runs all the Java tests

UI testing with Karma.js

- Karma.js is a framework to do unit tests in JavaScript
  - ➔ Uses mocks to simulate the back-end
  - ➔ Uses Phantom.js to run a browser from the command line
- Those tests are located in “src/test/javascript”
  - ➔ “yarn test” runs all the JavaScript tests

Performance testing with Gatling

- Gatling is used to do performance tests on the back-end
  - ➔ Can simulate a high number of concurrent users
  - ➔ Provides a Scala DSL to code the test scenarios
- Those tests are located in “src/test/gatling”
  - ➔ “mvn gatling:execute” allows to select which test to run
  - ➔ A report is generated at the end of the test, with detailed performance results

Lab 6: testing

IPPON

Testing on the Java side

- A ticket's priority should not be negative: add a "@Min(0)" Bean Validation annotation on the "priority" field we have created
- Test this new validation rule
  - ➔ Add a new method to TicketResourceIntTest
  - ➔ This method should test that if the priority is "-1", a "400 Bad Request" is returned by the server
  - ➔ (optional) Also test that a priority of "1" returns a "201 Created"

Solution

Ticket.java

```
@Min(0)
private int priority;
```

TicketResourceIntTest.java

```
@Test
@Transactional
public void checkPriorityIsNotNegative() throws Exception {
    int databaseSizeBeforeTest = ticketRepository.findAll().size();
    ticket.setPriority(-1);
    restTicketMockMvc.perform(post("/api/tickets")
        .contentType(TestUtil.APPLICATION_JSON_UTF8)
        .content(TestUtil.convertObjectToJsonBytes(ticket)))
        .andExpect(status().isBadRequest());

    List<Ticket> tickets = ticketRepository.findAll();
    assertThat(tickets).hasSize(databaseSizeBeforeTest);
}
```

Deploying to production

The “prod” profile

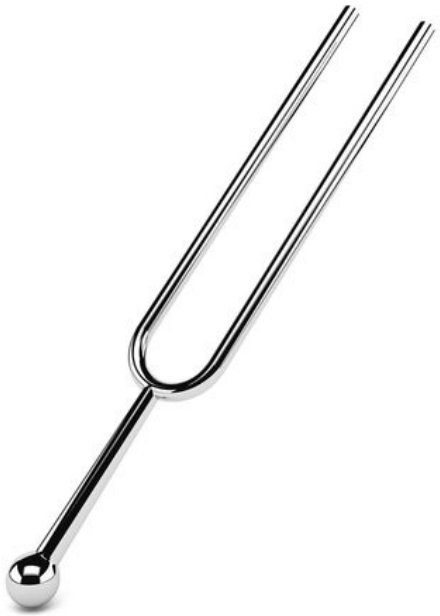
- JHipster has a “prod” profile, that packages an application for production use
  - ➔ It will optimize the front-end for production
  - ➔ It will tune the back-end for production, consistent with the front-end
- It is run with Maven
  - ➔ “mvn package -Pprod” will generate a production WAR file in the “target/” directory
  - ➔ “mvn -Pprod” will run directly the application in “prod” mode
- The generated WAR file contains the production code
  - ➔ To run it with the Spring “prod” profile, use the standard Spring Boot configuration: “./jhipster-0.0.1-SNAPSHOT.war --spring.profiles.active=prod”

Front-end minification

- JHipster will use Yarn to minify the application front-end
- All JS/CSS/HTML files will be minified, and the result will appear in the "target/www" folder
  - ➔ You can compare the "src/main/webapp/index.html" file used in development with the "target/www/index.html" file used in production
  - ➔ Minified files will have a checksum in front of their name, so they can be cached effectively
  - ➔ You can use the Chrome Dev Tools, including the audit tool, to check that everything has been minified

Server-side tuning

- In production mode, JHipster optimizes the web application
  - ➔ HTTP requests are GZipped (if they are >1kb)
  - ➔ Cache headers are added to all static files (see the previous slide about the checksum added in front of each file)
- The application uses the “application-prod.yml” configuration file
  - ➔ Including the specific configuration properties for using the production database
- Spring Boot devtools are disabled



Deploying the WAR file

- The result of the “production build” is a WAR file
- It is an executable WAR file, that contains an embedded Tomcat server
  - ➔ It can be run in production with “./jhipster-0.0.1-SNAPSHOT.war --spring.profiles.active=prod”
  - ➔ It can be deployed in a Java EE application server that supports the Servlet 3.0 specification, using the “spring.profiles.active=prod” environment variable

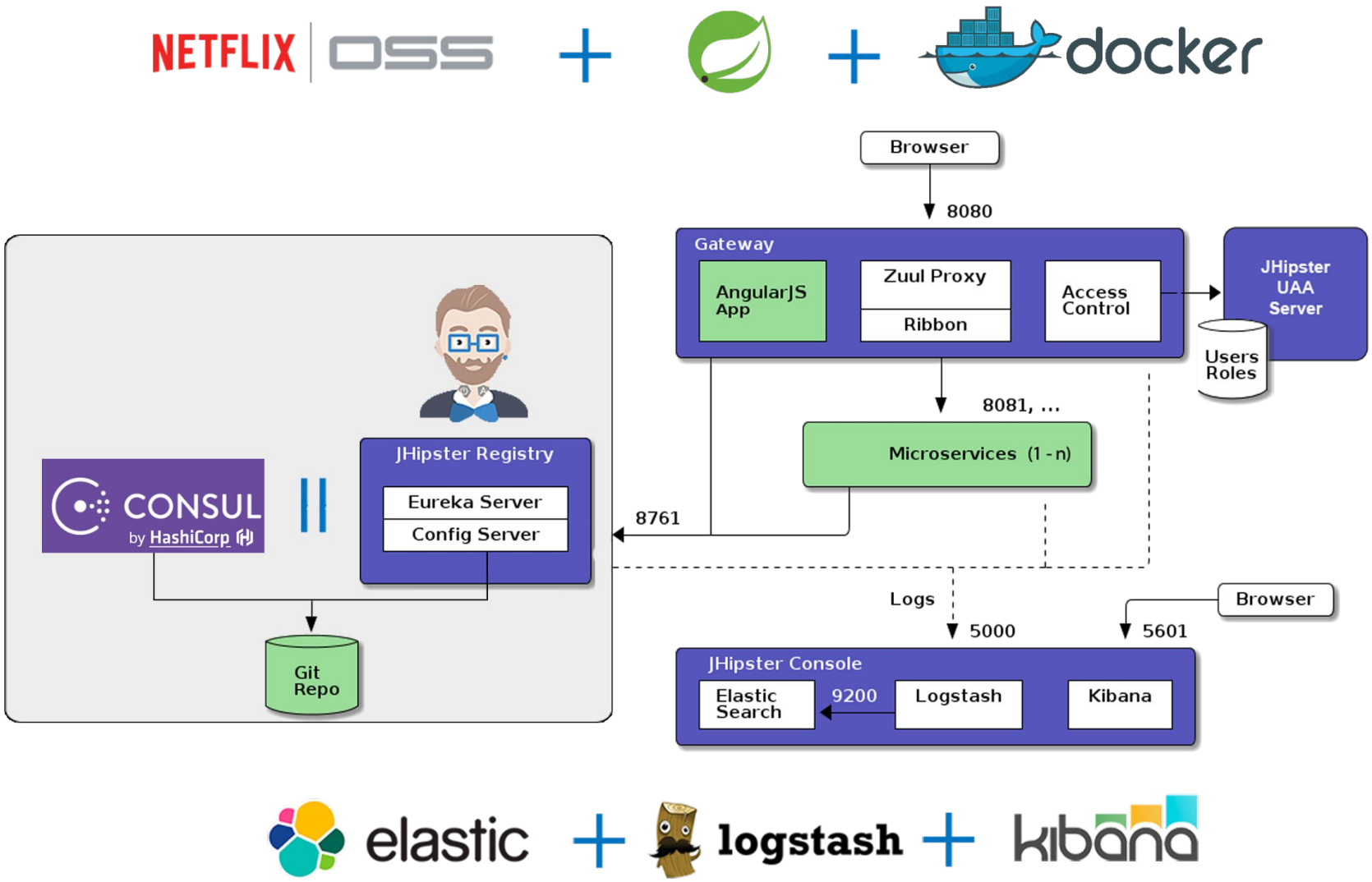
Deploying to a cloud provider

- JHipster provides several sub-generators to deploy the application to the cloud
  - ➔ Cloud Foundry
  - ➔ Heroku
  - ➔ Amazon Web Services
- Each of those providers have specific prices and features
- If you selected “MySQL” and “Elasticsearch” options when generating your application, your cloud provider must also provide options for both

Deploying to the Cloud Demo

Doing microservices

The big picture



A microservice with JHipster

- Normal JHipster application, focused on the back-end
  - ➔ All databases work as expected
  - ➔ Main JHipster options work as usual
  - ➔ Sub-generators work
- No graphical user interface (no Angular)
- No user management code

A gateway with JHipster

- Normal JHipster application, focused on the front-end
  - ➔ Can also have a back-end: it has a database, and can have full-fledged entities
  - ➔ All usual options work as usual
  - ➔ Sub-generators work
- But it is also a “router” to microservices
  - ➔ Does load balancing and circuit breaking
  - ➔ Very high performance
  - ➔ Handles security and user management for microservices
  - ➔ Optionally, can have rate limiting features
- And it has a specific UI
  - ➔ Can generate front-end code (Angular) based on an existing microservice

The JHipster Registry

- Runtime component provided by JHipster
- Fully Open Source (Apache 2 license)
- Service Registry based on Spring Cloud Eureka
  - ➔ All services register themselves on the JHipster Registry
  - ➔ Allows load balancing on the gateways
  - ➔ Allows microservice scalability and cluster configuration
- Configuration server based on Spring Cloud Config
  - ➔ Sends configuration data to all services
  - ➔ Useful to version, tag, rollback configurations
  - ➔ Allows to store “sensitive” information like database passwords
- NEW: Can choose to use Hashicorp’s Consul instead of the JHipster Registry

JHipster Console

- Monitoring console based on the ELK stack
  - ➔ Elasticsearch, Logstash, Kibana
  - ➔ Aggregates logs from microservices and gateways
  - ➔ Provides pre-defined dashboards
- Logs are sent by each JHipster application
  - ➔ Log messages sent by using the logback API: “log.debug()”
  - ➔ Dropwizard Metrics data dumped regularly to the logs, with detailed information from the JVM, Spring Beans, etc.
- Alerting is also available
  - ➔ Using Elastalert from Yelp

Running everything with Docker Compose

- JHipster applications are always configured to run with Docker
  - ➔ Services like databases or Elasticsearch have a pre-configured Docker Compose configuration in the “src/main/docker” directory
  - ➔ Applications can be packaged as Docker images using Maven or Gradle
  - ➔ Run “mvn -Pprod package docker:build” to generate an image of your current application in production
    - Uses Alpine Linux and Open JRE
    - Very light and secured
- When doing microservices, JHipster provides the “docker-compose” sub-generator
  - ➔ Allows us to have a Docker Compose configuration of everything, so running the whole architecture is very easy
  - ➔ Includes the JHipster Registry and (optionally) the JHipster Console
  - ➔ Allows us to scale services easily using “docker-compose scale”

Lab 7: creating a simple microservice

Generating a microservice

- Create a new directory called “microservices”
- In that directory, create an application called “catalog”
  - ➔ Create a “catalog” sub-directory
  - ➔ Generate a “microservice application” in that directory
    - Choose a MySQL database
    - Choose the (default) HazelCast cache
    - Add an Elasticsearch search engine
  - ➔ Create a “product” entity in that application
    - Add the entity using “yo jhipster:entity Product”
    - A product has a name, description, price….
  - ➔ Run “mvn test” to test your project
- Generate a Docker image of your project
  - ➔ Run “mvn -Pprod package docker:build”

Generating a gateway

- Go to the “microservices” directory
- In that directory, create an application called “gateway”
  - ➔ Create a “gateway” sub-directory
  - ➔ Generate a “microservice gateway” in that directory
    - Choose a MySQL database
  - ➔ Create a “product” entity in that application, that is the front-end to the microservice we have generated
    - Add the entity using “yo jhipster:entity Product”
    - Confirm that you will generate that entity from an existing microservice
    - Write the path to your existing “Catalog” microservice
  - ➔ Run “mvn test” and “yarn test” to test your project
- Generate a Docker image of your project
  - ➔ Run “mvn -Pprod package docker:build”

Building and running with Docker

- Go to the “microservices” directory
- In that directory, create a sub-directory called “docker-compose”
  - ➔ Go that directory and run “yo jhipster:docker-compose”
  - ➔ Confirm you want to use the parent (“../”) directory
  - ➔ Choose to use both the Catalog and the Gateway applications
  - ➔ Select ELK monitoring
- Run everything
  - ➔ Just type “docker-compose up -d”
- Test
  - ➔ Go to http://127.0.0.1:8080/ to have access to the gateway
  - ➔ Create some entities, and test the Elasticsearch is working
  - ➔ Go to http://127.0.0.1:8761/ to have access to the JHipster Registry
  - ➔ Go to http://127.0.0.1:5601 to have access to the JHipster Console

Scaling with Docker

- Scale the “catalog” microservice with Docker
  - ➔ Run “docker-compose scale catalog-app=2”
- A second instance of “catalog” is running
  - ➔ As it uses HazelCast, a distributed cache will be automatically configured between both instances
  - ➔ This second instance will be available in the JHipster Registry and in the gateway’s admin screen
  - ➔ It will also be automatically monitored by the JHipster Console
- You can launch new instances, and kill existing ones, to see how the architecture handles failure, circuit breaking and load balancing
- When you have finished, just run “docker-compose down” to destroy everything!

PARIS
BORDEAUX
NANTES
LYON
WASHINGTON DC
NEW-YORK
RICHMOND
MELBOURNE
MARRAKECH

contact@ippon.fr
www.ippon.fr - www.ippon-hosting.com - www.ippon-digital.fr
@ippontech
-
01 46 12 48 48

support@stormpath.com
www.stormpath.com
@goStormpath