



JHipster

David THIBAU – 2022

david.thibau@gmail.com



Agenda

- Introduction
 - Objectifs et Yeoman
 - Technologies associées
 - Générateurs et Modules
 - Installation, mise en place
- Tests, CI, Déploiement
 - Tests
 - CI
 - Déploiement / Monitoring
- Application monolithique
 - Création d'application, structure projet
 - Générateur d'entité
 - Syntaxe JDL,
 - Gestions des relations
 - Customisations de l'application CRUD
 - Création de service
 - Création de DTO
- Micro-services
 - Architecture micro-services
 - Spring Cloud, et ses micro-services techniques
 - Création avec Jhipster, Choix de JHipster



Introduction

Objectifs et Yeoman

Technologies associées
Générateurs et Modules
Installation, mise en place



Introduction

Objectifs de *JHipster* : Productivité, Outillage, Qualité

- Permettre le démarrage rapide de projet
- Utiliser les meilleurs frameworks
- Utiliser des bonnes pratiques, design patterns
- Avoir un processus de build sûr et complet

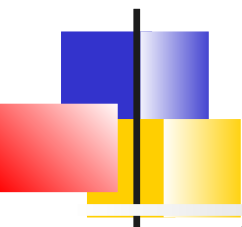
Projet OpenSource très actif

Eco-système afin que des parties tierces s'intègrent



Historique

- Démarré par Julien Dubois en Octobre 2013 (Ippon Technologies)
- 1ère release publique (version 0.3.1) le 7 décembre 2013.
Depuis, plus de ~ 150 releases
- Open-source, Apache 2.0-licensed sur GitHub
<https://github.com/jhipster/generator-jhipster>
- Équipe de 19 développeurs cœur et plus de 350 contributeurs



Yeoman

JHipster est un générateur ***Yeoman***.

Vu de *Yeoman*, c'est un plugin qui peut être exécuté avec la commande ***yo***

Yeoman est une technologie Javascript qui est basée sur 3 types d'outils :

- Le générateur (*yo*)
- L'outil de build (*gradle, npm/yarn, Gulp webpack*)
- Le gestionnaire de paquets (*Bower, npm/yarn*)



Responsabilités de outils

- `yo` pose les fondations d'une nouvelle application, écrit la configuration du build et récupère les plugins et dépendances nécessaires pour le build de l'application.
- Le système de *build* permet de construire, prévisualiser et tester un projet
- Le gestionnaire de paquets gère les dépendances



Générateurs

Un générateur *yeoman* est typiquement un module *Node.js* qui respecte une structure de répertoire.

Typiquement, le générateur principal est présent dans un répertoire *generators/app*, des sous-générateurs sont contenus dans leur propre répertoire

```
|—package.json
|—generators/
  |—app/
  |   |—index.js
  |   |—router/
  |       |—index.js
```

=> *yo name* ET *yo name:router*



Exemple JHipster

generators	27 éléments	Dossier	11 oct.
▸ app	4 éléments	Dossier	11 oct.
▸ aws	4 éléments	Dossier	11 oct.
▸ ci-cd	4 éléments	Dossier	11 oct.
▸ client	6 éléments	Dossier	11 oct.
▸ cloudfoundry	4 éléments	Dossier	11 oct.
▸ docker-compose	5 éléments	Dossier	11 oct.
▸ entity	5 éléments	Dossier	11 oct.
▸ export-jdl	2 éléments	Dossier	11 oct.
▸ heroku	3 éléments	Dossier	11 oct.
▸ import-jdl	2 éléments	Dossier	11 oct.
▸ info	2 éléments	Dossier	11 oct.
▸ kubernetes	5 éléments	Dossier	11 oct.
▸ languages	3 éléments	Dossier	11 oct.
▸ modules	2 éléments	Dossier	11 oct.
▸ openshift	5 éléments	Dossier	11 oct.
▸ rancher-compose	5 éléments	Dossier	11 oct.
▸ server	5 éléments	Dossier	11 oct.
▸ spring-controller	4 éléments	Dossier	11 oct.
▸ spring-service	3 éléments	Dossier	11 oct.
▸ upgrade			

« generators » sélectionné (contenant 27 él



Racine projet

Lors de l'exécution de la commande `yo`, *Yeoman* remonte dans l'arborescence de répertoires à la recherche d'un fichier ***.yo-rc.json***.

- Si il trouve, il considère que c'est la racine du projet et exécute le générateur demandé dans ce répertoire
- Sinon, il crée le fichier *.yo-rc.json* dans le répertoire courant



Introduction

Objectifs et Yeoman
Technologies associées
Générateurs et Modules
Installation, mise en place



Recommandations

En fonction des ses évolutions, *JHipster* recommande les technologies et framework qui lui semble les plus performantes.

A l'heure actuelle :

- Front-end : *Angular ou React ou Vue, Bootstrap*
- Back-end : *Spring Boot*



Technologies Front End

CSS :

- *Bootstrap* et peut être dans le futur *Material Design*
- *Saas* : Variables dans le css

Framework MV*

- *Angular, React, Vue*
- *Angular Translate* : Internationalisation
- Eventuellement *Thymeleaf*

Build, Optimisation et minification de code

- *yarn/npm* et *webpack*

Test

- *JTest* et *Cypress* (*Protractor* déprécié)

Productivité

- *Browsersync* : Live-reload + synchronisation de navigateurs



Technologies Back-end

Framework : *Spring Boot* et *Spring Cloud* pour les micro-services

Build :

- *Maven*
- *Gradle*

REST API

- Spring MVC et les Rest Controller
- Spring Cloud Gateway, Eureka ou Consul

Persistence

- Spring Data : JPA
- Liquibase : Suivi des changements de modèle
- Elasticsearch, MongoDB, CouchBase, Cassandra
- Kafka

Authentication

- Spring Security
 - Default : Http Session
 - JWT
 - OAuth2



Tests backend

Pour les tests, *JHipster* par défaut installe des tests unitaires et d'intégration avec *JUnit*

Il propose en option :

- *Gatling* : Test de performance
- *Cucumber* : Test d'acceptation



Intégration continue

JHipster peut générer des pipelines d'intégration continue pour :

- *Jenkins*
- *Travis CI*
- *GitLab CI*
- *CircleCI*
- *GitHub Actions*

Il propose également une intégration avec Sonar pour vérifier la qualité du code



Déploiement

Les applications générées peuvent être

- Au format *.war* et déployées sur un serveur
- Au format fat jar et exécutées par une simple JVM
- Sous la forme d'image Docker

Le déploiement peut se faire facilement vers des solutions de Cloud :

- Images Docker
- CloudFoundry, Heroku, AWS, ...

Deployment Options



Client Side Options



Server Side Options





IDEs

JHipster propose 2 outils pour le développement

- ***JDL*** : DSL permettant de décrire les entités et leurs relations.
 - *JDL-studio* est un éditeur Web
 - *JHipster IDE* est un plugin pour Eclipse ou VisualCode
- ***JHipster-UML*** : Génération d'entités à partir d'UML



Introduction

Objectifs et Yeoman
Technologies associées
Générateurs et Modules
Installation, mise en place



Introduction

Le générateur principal crée le projet

```
mkdir myProject  
cd myProject  
yo jhipster
```

Différents types de projet peuvent être générés :

- Application monolithique
- Micro-service, Gateway

Ensuite des sous-générateurs sont utilisés pour créer du code

- Générateur d'entités
- Générateur de Service
- Générateur de DTO



Modules

Un **module *JHipster*** est un générateur Yeoman additionnel qui *dérive* d'un sous-générateur de *JHipster* afin d'hériter de fonctionnalités *JHipster*.

- Un module peut également s'enregistrer comme Hook du générateur JHipster.
- Les modules peuvent être accéder via le JHipster ***marketplace***.

=> Le système de module permet donc à des tiers de proposer leur propre générateur

=> Cela peut être un moyen (un peu compliqué) de générer du code spécifique dans un projet !!



Introduction

Objectifs et Yeoman
Technologies associées
Générateurs et Modules
Installation, mise en place



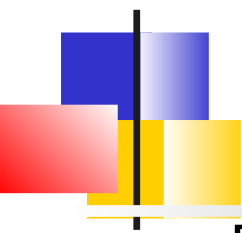
Types d'installation

JHipster Online est un moyen de générer une application sans installer JHipster en premier lieu.

Installation locale avec **NPM**. Méthode recommandée.

Installation locale avec **Yarn**

Conteneur "**Docker**",



Installation *Jhipster* avec NPM

Pré-requis :

- Java 11
- NodeJS

Optionnels :

- Maven, Gradle
- Git
- Docker

Puis

```
npm install -g generator-jhipster
```

Optionnellement pour module complémentaire via le MarketPlace

```
npm install -g yo
```



Eclipse IDE

JHipster permet de démarrer une application mais après avoir généré les entités et vues Angular, il faut coder !

Les projets JHipster sont en fait de simples projets Spring Boot basés sur Maven ou Gradle.

Ils sont importables dans la plupart des IDEs Java

- *Pivotal* propose une suite de plugins Eclipse pour les projets SpringBoot packagé dans **Spring Tools Suite**.
- *JHipster* propose également un plugin JHipster permettant de gérer les fichiers de description des entités



Application monolithique

Création de projet

Génération d'entité

Gestions des relations

Customisations de l'application CRUD

Générations de DTO

Génération de service



Assistant

```
mkdir myProject  
cd myProject  
jhipster
```

L'assistant pose de nombreuses questions. Les choix les plus simples étant :

- Type d'application : Monolithique, Microservice ou Gateway
- Authentification : JWT, OAuth2 ou Session
- Type de BD (Dev et Prod) : SQL, NoSQL,
- Cache : Spring Cache ou Hibernate 2nd level cache si SQL
- Maven ou Gradle
- Autres technologies : Elasticsearch, API first, Kafka, ...
- Framework Client : Angular, React ou Vue
- Bootstrap theme
- LibSaas : Y
- Internationalization : Oui
- Test frameworks : Gatling, Cucumber
- Liquibase ?



README

En fonction des réponses, JHipster crée un fichier ***README.md*** à la racine du projet détaillant les commandes de build disponibles

Les principales commandes proposées sont :

- Développement : Démarrage de l'appli et de *yarn* pour avoir des fonctionnalités de reload automatique du navigateur
- Ajoute de dépendances via *yarn*
- Commande de génération de composant Angular
- Utilisation du profil de production
- Lancement des tests back-end, client et Gatling
- Utilisation de docker
- Configuration pour l'intégration continue



Vérifications

Maven

`./mvnw #démarré Spring Boot sur le port 8080`

`./mvnw clean test #Exécute les tests`

`yarn test #Exécute les tests client`

Gradle

`./gradlew #démarré Spring Boot`

`./gradlew clean test #Exécute les tests`

TP : Création du projet monolithique forum => Initial commit



Importation dans STS

Après la création du projet, il est possible de l'importer dans l'IDE

Import → Existing Maven Project

- Exclure le répertoire *node_modules* pour enlever les erreurs Javascript
- Inclure *src/main/webapp* dans les chemins Javascript et exclure app
- Activer les profils *dev* et *IDE* dans les propriétés Maven
- Tester le démarrage de l'application et accéder à la documentation *swagger*
(<http://localhost:8080/swagger/index.html>)

TP : Importation dans Eclipse et démarrage de l'appli :

<http://www.jhipster.tech/configuring-ide-eclipse/>



Structure du projet

JHpister crée une structure de projet Maven adaptée à SpringBoot et Angular

En particulier à la racine du projet, on trouve :

- ***pom.xml*** : héritage du *pom.xml* de SpringBoot fixant les versions, nombreuses propriétés de build, les dépendances requises par le projet, la configuration des plugins, plusieurs profils (dev, IDE, prod, ...)
- ***package.json*** : Dépendances, dépendances pour le développement, la version de *node* requise, les commandes associées au cycle de build



Fichiers back-end

Le code Java est dans *src/main/java*

- Dans le **package principal**, la classe principale SpringBoot et une classe permettant d'initialiser le contexte Spring au déploiement d'un war
- **<rootPackage>.config** : Les classes *@Configuration* de SpringBoot
- **<rootPackage>.domain** : Les classes du modèle
- **<rootPackage>.repository** : Les classes Repository au sens SpringBoot
- **<rootPackage>.security** : La classe effectuant l'authentification,
- **<rootPackage>.security.jwt** : Gestion du token JSON (JWT)
- **<rootPackage>.service** : Couche service
- **<rootPackage>.service.dto** : Classes d'encapsulation DTO
- **<rootPackage>.service.mapper** : Transformation entité ↔ DTO
- **<rootPackage>.web.rest** : API Rest
- **<rootPackage>.web.rest.vm** : Modèle des composant d'UI



Fichiers ressources

Dans *src/main/resources* on trouve les fichiers statiques utilisés par le code Java :

- Les fichiers de configuration SpringBoot ***application*.yml***
- Les fichiers de configuration de *Liquibase* contenant les mises à jours BD
- Les fichiers de labels pour l'internationalisation
- Les gabarits pour les notifications
- Le gabarit pour la page d'erreur



Tests

Les tests unitaires et d'intégration sont situés dans ***src/test/java***

Les tests de performance sont dans ***src/test/gatling***

Les tests d'acceptance sont dans ***src/test/features***



Fichiers *front-end*

La racine du *front-end* est dans
src/main/webapp

- Le répertoire ***app*** contient les modules Angular
- ***i18n*** contient les fichiers pour l'internationalisation
- Les tests unitaires Karma sont dans ***src/test/javascript/spec***
- Les tests fonctionnels (Protractor) sont dans ***src/test/javascript/e2e***



Application monolithique

Création de projet
Génération d'entité
Gestions des relations
Customisations de l'application CRUD
Générations de DTO
Génération de service



Sous-générateur entité

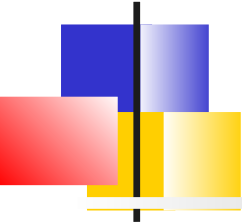
```
yo jhipster:entity <my_entity>
```

Le générateur démarre un assistant permettant de créer une classe entité.

Pour chaque entité, on précise :

- Les champs : types et validation
- Les relations avec les autres entités
- La possibilité de générer une classe DTO
- La possibilité de générer une classe service associée
- L'utilisation de la pagination

JHipster a par défaut une entité *User* avec laquelle il est possible de faire des relations



Fichiers générés pour une simple entité

.jhipster/Entity.json : Les réponses de l'assistant au format JSON

Code source Java

- ***<rootPackage>/domain/Entity.java*** : La classe entité
- ***<rootPackage>/repository/EntityRepository.java*** : Le repository
SpringData
- ***<rootPackage>/web/rest/EntityResource.java*** : L'API Rest

Code de test

- ***<rootPackage>/web/rest/EntityResourceIntTest.java*** : Test
d'intégration

Liquibase

- Répertoire ***changelog*** : Un fichier XML décrivant la création de la table associée à l'entité
- ***master.xml*** : Ajout de l'inclusion du fichier précédent



Côté Front-end

Côté Angular, les fichiers générés comportent :

- Plusieurs composants (liste, vue détail, dialogue pour les erreurs de validation, dialogue pour confirmer une suppression)
- Un routeur
- Un service injectable faisant le lien avec le backend

TP : Génération de l'entité Topic (title et description)



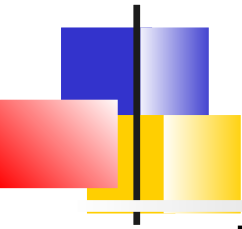
Application monolithique

Création de projet
Génération d'entité

Gestions des relations

Customisations de l'application CRUD

Générations de DTO
Génération de service



Générateur des relations

Lors de la génération d'une relation, il est possible d'indiquer

- la cardinalité
- Le nommage d'une relation
- Est-ce une relation requise ?
- Dans certains cas, le *owner* de la relation et le nom du champ dans l'autre entité



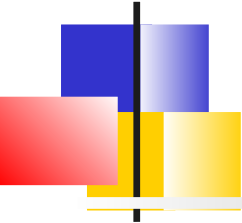
Entité avec relations

Types de relations

- **Cardinalité**

- *one-to-many*
- *many-to-one*
- *one-to-one*

- **Bidirectionnelle** ou **unidirectionnelle**



Relation unidirectionnelle *one-to-many*

Ce type de relation n'est pas supportée pour le moment par JHipster

2 solutions pour contourner le problème :

- Utiliser un mapping bidirectionnel, mais comment faire un graphe d'éléments ?
- Spécifier un mapping bidirectionnel à la génération et la modifier en un mapping unidirectionnel :
 - Supprimer l'attribut "mappedBy" sur l'annotation *@OneToMany*
 - Générer la table d'association nécessaire via `mvn liquibase:diff`



Fichiers générés

JHipster génère le code pour :

- Gérer la relation avec JPA dans les entités générées
- Créer le changelog dans *Liquibase* afin de créer les tables ou clé étrangères nécessaires dans la base
- Générer le front-end afin que la relation soit prise en compte dans l'interface



JDL

JDL (Jhipster Domain Language) décrit
l'application et tout le modèle de données dans
un seul fichier

Des éditeurs graphiques permette la
représentation visuelle du modèle

Génération des entités sur un nouveau projet ou
projet existant avec le sous-générateur *import-jdl*

Egalement possible de générer des entités et les
exporter dans un fichier JDL en utilisant JHipster
UML



Syntaxe Structure

```
application {  
  config {  
    <application option name> <application option value>  
  }  
}
```

```
[entities <application entity list>  
[<options>  
}
```



Syntaxe

```
[<entity javadoc>]
[<entity annotation>*]
entity <entity name> [( <table name> )] {
    [<field javadoc>]
    [<field annotation>*]
    <field name> <field type> [<validation>*]
}
relationship (OneToMany | ManyToOne | OneToOne | ManyToMany) {
    <from entity>[<{<relationship name>[(<display field>)]>}] to <to entity>[<{<relationship
    name>[(<display field>)]>}] +
}
enum Language {
    FRENCH, ENGLISH, SPANISH
}
```

dto A, B with mapstruct

paginate A with (infinite-scroll, pagination)

service A with (serviceClass, serviceImpl)



Example *.jdl*

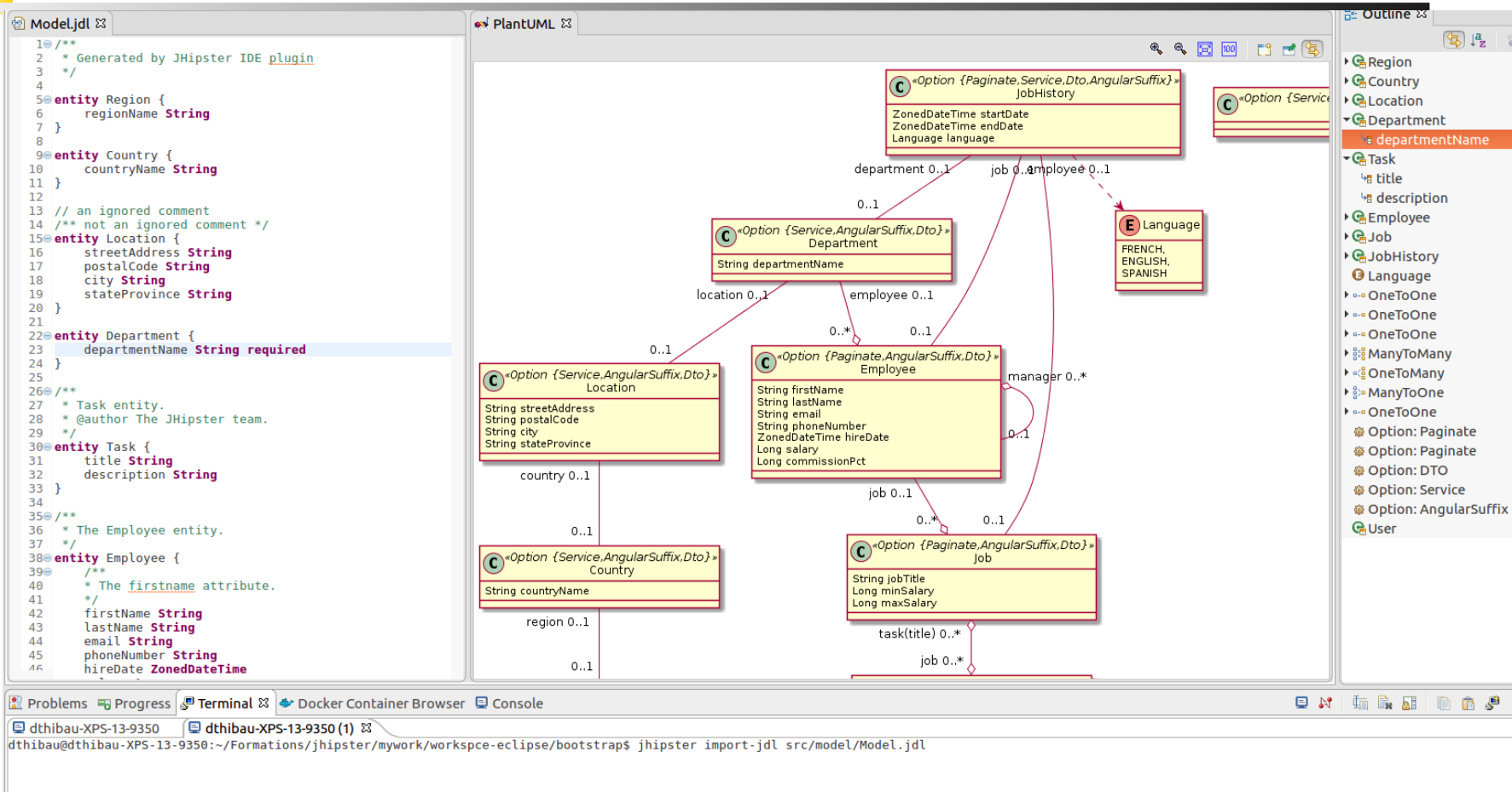
```
entity Topic {  
  title String maxlength(80),  
  description String  
}  
  
entity Message {  
  subject String maxlength(80),  
  content String  
}  
  
relationship ManyToOne {  
  Message{author} to User,  
  Message{topic} to Topic  
}  
  
relationship OneToMany {  
  Message{responses} to Message  
}  
  
// Set pagination options  
paginate Message with infinite-scroll  
paginate Topic with pagination
```



Outils

- ***JHipster IDE*** et ***JDL-studio*** permettent d'éditer les fichiers JDL
 - Éditeur de fichier *.jpdL* (coloration, complétion, correction syntaxique, ...)
 - Synchronisation du *.jpdL* avec une vue UML
 - Terminal pour lancer les commandes de génération depuis l'IDE

Vues IDE





Usage

```
jhipster import-jdl my_file.jdl or jhipster-uml my_file.jdl.
```

```
jhipster import-jdl my_file1.jh my_file2.jh
```

#Ne pas régénérer le code

```
jhipster import-jdl ./my-jdl-file.jdl --json-only
```

Forcer la régénération (même des entités inchangées)

```
jhipster import-jdl ./my-jdl-file.jdl --force
```

Installation dans le projet :

```
npm install jhipster-core --save
```



Application monolithique

Création de projet
Génération d'entité
Gestions des relations

Customisations de l'application CRUD

Générations de DTO
Génération de service



Magique *JHipster* ?

Générateur de code → customisation

Que se passe t-il si j'ai besoin de régénérer ?

Et bien écraser son code spécifique !

Quelques workaround :

- Changement du modèle : Liquibase
- Branches Git et merging
- Ajout d'une fonctionnalité oubliée : Génération d'un nouveau projet et importation de notre code
- Développer le code spécifique en dehors de l'arborescence de génération



Structure front-end

webapp

- ├─ app
 - │ └─ account
 - │ └─ admin
 - │ └─ blocks
 - │ └─ entities
 - │ └─ home
 - │ └─ layouts
 - │ └─ shared
 - │ └─ app.main.ts
 - │ └─ app.module.js
 - │ └─ app.route.js
 - ├─ content
 - │ └─ css
 - │ └─ images
 - ├─ i18n
 - ├─ scss
 - ├─ swagger-ui
 - ├─ 404.html
 - ├─ favicon.ico
 - ├─ index.html
 - └─ robots.txt
- Racine du code source applicatif
 - Interface de gestion des utilisateurs
 - Interface d'administration
 - Blocks communs : configuration et intercepteurs
 - Les entités
 - La page d'accueil
 - Les layouts
 - Services communs (authentification, internationalisation)
 - Classe principale
 - Configuration des modules applicatifs
 - Routeur principal
 - Contenu statique
 - CSS
 - Images
 - Internationalisation
 - Sass
 - Swagger
 - 404 page
 - Fav icon
 - Index page
 - Configuration pour les web crawlers

Fichiers entités

```
webapp
├── app
│   ├── entities
│   │   ├── foo
│   │   │   ├── foo.component.html      - Un répertoire par entité
│   │   │   │                               - Template HTML pour la page liste
│   │   │   ├── foo.component.ts        - Contrôleur pour la page liste
│   │   │   ├── foo.model.ts            - Classe Modèle représentant l'entité
│   │   │   ├── foo.module.ts           - Module Angular module pour l'entité
│   │   │   ├── foo.route.ts            - Configuration routeur Angular
│   │   │   ├── foo.service.ts          - Service accédants ax ressources REST de l'entité
│   │   │   ├── foo-delete-dialog.component.html - Vue HTML pour la suppression
│   │   │   │                               - Contrôleur pour la suppression
│   │   │   ├── foo-delete-dialog.component.ts
│   │   │   ├── foo-detail.component.html - HTML pour Vue détail
│   │   │   │                               - Contrôleur
│   │   │   ├── foo-detail.component.ts
│   │   │   ├── foo-dialog.component.html - HTML pour l'édition
│   │   │   │                               - Contrôleur pour l'édition
│   │   │   ├── foo-dialog.component.ts
│   │   │   ├── foo-popup.service.ts    - Service pour la gestion de la création et de la mise à jour
│   │   │   └── index.ts                -Exports
│   └── i18n                            - Fichiers de traductions
│       ├── en                          -
│       │   ├── foo.json                - Traductions anglaises ...
│       │   └── fr
│       │       ├── foo.json            - Traductions françaises ...
```




Layouts

Main layout :

3 router outlets :

- Barre de navigation
- Corps
- Popup



Développement

Lors du développement démarrer

- le back-end avec
`./mvnw -P-webapp`
- Le front-end avec
`npm start` ou `yarn start`

Cela permet de ne pas recharger le navigateur

Si l'on veut utiliser de vrais BD, Jhipster fournit des fichiers *`docker-compose`* pour les démarrer



CSS

Le moyen le plus simple de personnaliser l'application est de surcharger le CSS

- Dans *src/main/webapp/content/css/global.css*
- Ou *src/main/webapp/scss/global.scss* si SaaS



Sass

Sass (*“syntactically awesome style sheets”*) utilise le symbole \$ pour définir une variable qui peut être référencer ensuite

Exemple

```
$font-stack:
Helvetica, sans-serif;
$primary-color: #333;
body {
  font: 100% $font-stack;

  color: $primary-color;
}
```

TP : Modification styles, fichier de libellés, page statique (suppression des colonnes id)



Développement Angular

Les applications *JHipster* nécessitent de la personnalisation du code généré

Elles peuvent être effectuées en dehors de l'environnement Eclipse et par des équipes distinctes. Voir :

- **Angular Augury** : Extension chrome pour le debugging d'application Angular4
- **John Papa Angular 2 style guide** : Document sur les bonnes pratiques de dev. Angular 4
- **Angular cli** : Outils d'aide à la création de composants Angular4



ng-jhipster library

JHipster propose ses propres composants Angular2+ à travers le projet ***ng-jhipster***

Les composants sont reconnaissables via le préfixe ***jhi***. Citons :

- Des directives de validation
- Directive pour l'internationalisation
- Des pipes utilitaires (Mise en majuscule, tri, césure de mot)
- Services stateless de pagination, ..
- Un système de notification (Push vers les composants *jhiAlertComponent*) via *AlertService*



Autorisations

Les autorisations sont spécifiées dans les routeurs Angular (*.route.ts)

Les rôles définis côté serveur (*AuthoritiesConstants.java*) et requis pour une route sont listées dans le champ ***data***

Si la liste est vide, les routes sont accessibles en anonyme
Exemple :

```
export const adminState: Routes = [{  
  path: '',  
  data: {  
    authorities: ['ROLE_ADMIN']  
  },  
  canActivate: [UserRouteAccessService],  
  children: ADMIN_ROUTES  
},  
  ...userDialogRoute  
];
```



Vérification

Une fois les permissions définies dans le router, elles peuvent être utilisées avec la directive ***jhiHasAnyAuthority*** :

Ex :

```
<h1 *jhiHasAnyAuthority="[ 'ROLE_ADMIN',  
'ROLE_USER' ]">Hello, dear user</h1>
```




Internationalisation

Si l'internationalisation a été choisie dès le début du projet

La librairie ***ngx-translate*** (ramenée par la dépendance *ng-jhipster*) peut être utilisée via la directive Angular ***jhiTranslate***.



Fichiers libellés

```
<h1 class="display-4" jhiTranslate="home.title">Welcome,  
  Java Hipster!</h1>
```

La clé référence un document JSON contenant le libellé

Ex : *home.json*

```
{  
  "home": {  
    "title": "Bienvenue, Java Hipster !",  
    "subtitle": "Ceci est votre page d'accueil",  
    "logged": {  
      "message": "Vous êtes connecté en tant que \"{{username}}\"." ,  
    },  
    ...  
  }  
}
```



Angular CLI

JHipster génère les fichiers de configuration utilisés par Angular CLI : un fichier **.angular-cli.json** est généré et la dépendance est déclarée dans *package.json*

=> L'utilisation d'Angular CLI n'est donc pas incompatible avec JHipster.

A priori pas de *ng build* mais :

- **ng generate** pour générer les composants

TP : Création nouvelle page, nouvelle route



Customisation Back-end

L'application Jhipster générée est un Application Spring Boot.

Pour rester dans le workflow de développement Jhipster, On peut :

- Générer des Bean Service et utiliser les annotations Spring
- Générer des classes DTO en utilisant MapStruct



Application monolithique

Création de projet
Génération d'entité
Gestions des relations
Customisations de l'application CRUD
Générations de DTO
Génération de service



Introduction

Pour des cas d'utilisation autres que le CRUD, les classes entités ne suffisent plus et il est souvent nécessaire de mettre en place des ***Data Transfer Objects*** (ou ***DTOs***) qui seront exposés par REST.

Ces objets en général agrègent des attributs provenant de plusieurs entités champ.



Couche Service

Lors de la génération d'une entité il est possible d'ajouter une couche de service.

L'option DTO n'est possible que si on a la couche Service.



DTOs simples

Lors de la génération d'une entité, il est possible de générer un DTO associé :

Le DTO contient alors :

- Les attributs de l'entité
- L'ID et les attributs sont ramenés d'une relation *many-to-one*
- Pour une relation *many-to-many*, les DTOs de l'autre entité (si ils existent) sont ramenés sous forme de *Set*.

Le DTO ignore les relations *one-to-many* et *many-to-many* si l'entité n'est pas le owner (cohérent avec l'annotation *@JsonIgnore* de ces champs).



DTOs complexes

JHipster pour l'instant utilise un outil tierce pour explicitement mapper les entités vers des DTO

MapStruct traite des annotations pendant la phase de compilation qui génère les classes DTOs

Pour configurer l'IDE, il faut activer le profil Maven *IDE*



Mapper Object

Les objets mappers de *MapStruct* sont des beans Spring et peuvent s'injecter un objet *Repository* pour exécuter des requêtes supplémentaires afin de récupérer les données supplémentaires de l'entité



Application monolithique

Création de projet
Génération d'entité
Gestions des relations
Customisations de l'application CRUD
Générations de DTO
– **Génération de service**



Introduction

C'est le sous-générateur le plus simple.

Son but est de générer un service Spring
ou l'on peut coder la logique métier

L'échange avec la couche REST
s'effectue

- Soit avec les objets entités
- Soit avec des objets DTO



Options de l'assistant

jhipster service ServiceName

L'assistant demande si le Service doit définir une interface

La recommandation est **non**

- Dans ce cas Spring AOP utilisera CGLIB pour ajouter des aspects au Service

Les aspects disponibles sont :

- transactionnels
- La sécurité
- Le monitoring



Fichiers générés

Juste un Bean *@Service*

Éventuellement les interfaces



Tests, CI, déploiement

Tests
CI
Production



Tests

JHipster génère un ensemble de tests :

- Tests unitaire et d'intégration avec *JUnit5 / Spring Boot Test*
- Tests d'UI tests avec *Jtest*
- Tests d'architecture avec *ArchUnit* (Contraintes sur l'architecture Java)

Optionnellement :

- Tests de performance avec *Gatling*.
- Tests d'acceptation avec *Cucumber*
- Tests d'intégration avec *Cypress* ou *Protractor*.

Les objectifs sont double :

- Favoriser le développement des tests pour le code spécifique
- Valider que le code généré est correct



Tests Java

Emplacement : *src/test/java*

Exécution : *./mvnw test*

JHipster génère les tests suivants :

- Tests de la configuration du ServletContext via *WebConfigurerTest*
- Tests pour la solution de sécurité choisie
- Tests d'intégration des services *JHipster* (Couche service + repository)
- Tests d'intégration Rest des entités



Exemple test entité

@Test

@Transactional

```
public void createTopic() throws Exception {
    int databaseSizeBeforeCreate = topicRepository.findAll().size();

    // Create the Topic
    restTopicMockMvc.perform(post("/api/topics")
        .contentType(TestUtil.APPLICATION_JSON_UTF8)
        .content(TestUtil.convertObjectToJsonBytes(topic)))
        .andExpect(status().isCreated());

    // Validate the Topic in the database
    List<Topic> topicList = topicRepository.findAll();
    assertThat(topicList).hasSize(databaseSizeBeforeCreate + 1);
    Topic testTopic = topicList.get(topicList.size() - 1);
    assertThat(testTopic.getTitle()).isEqualTo(DEFAULT_TITLE);
    assertThat(testTopic.getDescription()).isEqualTo(DEFAULT_DESCRIPTION);
}
```



Base de test

Une base de données spécifiques est utilisée pour exécuter les tests.

Dans le cas d'une base SQL, une base H2 est créé et mise en place par Liquibase

Attention : JHipster n'est pas capable de générer des valeurs correctes pour les entités ayant des contraintes de validation trop complexe (*regexp* par exemple)



Tests UI : JTest

Emplacement : *src/test/javascript/spec*

Exécution : *npm test*.

Ces tests « *mock* » l'accès au back-end, et s'exécutent sans démarrage du back-end.



Test e2e Cypress/Protractor

Emplacement : *src/test/javascript/e2e*

Exécution : *npm run e2e*

Les tests lancent un navigateur Web et utilisent l'application comme le ferait un véritable utilisateur.

L'application réelle est en cours d'exécution, avec sa configuration de base de données.



Qualité et Sonar

JHipster propose des images Docker d'une installation Sonar :

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

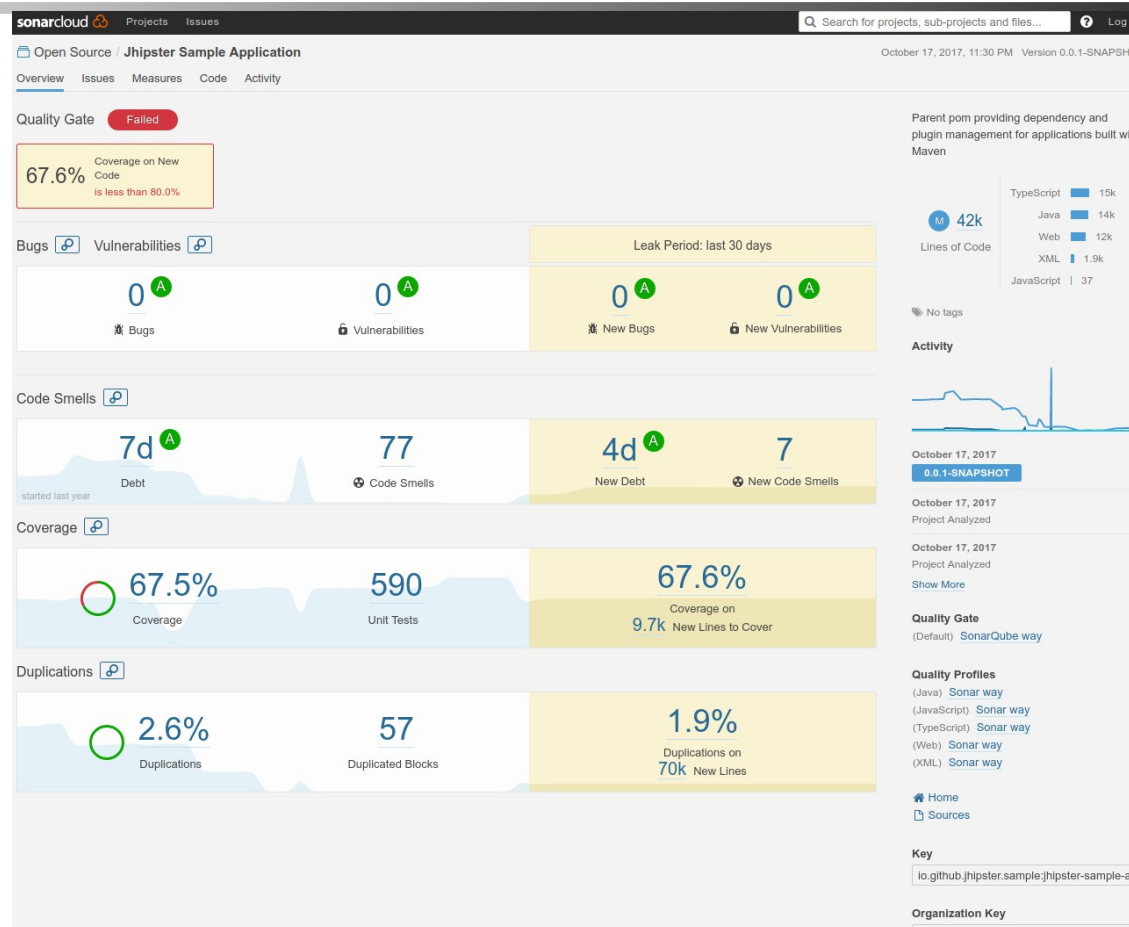
L'analyse peut être démarrée par :

```
./mvnw -Pprod clean verify sonar:sonar -  
Dsonar.host.url=http://localhost:9001
```

Les résultats sont alors disponibles sur le serveur Sonar :

```
http://127.0.0.1:9001/
```

Exemple : Sample project



<https://sonarcloud.io/dashboard?id=io.github.jhipster.sample%3Ajhipster-sample-application>

TP : Exécution Sonar et génération de métriques



Tests, intégration continue, déploiement

Tests
CI
Production



Introduction

JHipster tente de supporter les solutions de CI suivantes :

- Jenkins
- Travis
- GitHub Actions
- Azure Pipelines
- CircleCI
- GitlabCI

Les fichiers de configurations nécessaires sont générés via un générateur yeoman :

jhipster ci-cd



Générateur

Le générateur demande quelle solution de CI est choisie

Pour Jenkins, il propose les options :

- Effectuer le build dans un conteneur Docker
- Analyser le code avec Sonar
- Envoyer le statut de build vers Gitlab
- Construire et publier une image Docker

Le générateur génère :

- Une pipeline décrite par le fichier *JenkinsFile*
 - Une image docker de Jenkins permettant de tester localement la pipeline
- ```
docker-compose -f src/main/docker/jenkins.yml up -d
```



# Tests, intégration continue, déploiement

---

Tests  
CI  
**Production**



# Profil *prod*

---

Le profil ***prod*** compile, test et package l'application avec la configuration de production

***./mvnw -Pprod***

Le build produit 2 fichiers .war :

- target/<appli>-0.0.1-SNAPSHOT.war : Fat jar executable
- target/<appli>-0.0.1-SNAPSHOT.war.original : .war à déployer sur un serveur



# Image docker

---

Une alternative peut être de générer une image Docker permettant d'exécuter l'application dans une configuration de production :

```
./mvnw package -Pprod dockerfile:build
```

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



# Front-end optimisé

---

Le profile *prod* déclenche l'optimisation du front-end avec webpack

- *webpack* traite alors toutes les ressources statiques (CSS, TypeScript, HTML, JavaScript, images...)
- Il compile le TypeScript en JavaScript et génère les source maps permettant de debugger l'application cliente.

Si l'on utilise le jar exécutable, gzip est utilisé pour compresser les échanges entre le client et le serveur

- Des entêtes de cache pour les ressources statiques sont ajoutées par un filtre SpringBoot



# Sécurité

---

Lors d'un déploiement en production, ne pas oublier de changer les mots de passe des utilisateurs par défaut :

*“admin”* , *“system”* et *“user”*

Soit par l'interface ou mieux par Liquibase :

*/src/main/resources/config/liquibase/users.csv*

Voir *BcryptPasswordEncoder* pour le hashage de mot de passe



# Monitoring

---

*JHipster* propose un support pour le monitoring via la librairie OpenSource Metrics.

Les données de Metrics sont accessibles via jmx  
(Clients *jmc* ou *jconsole*)

En production, l'application envoie les données vers un ELK, la console *JHipster* ou un serveur Graphite, en fonction des propriétés de ***application-prod.yml***



# Console jmx

JVM Browser [1.8.0\_131] org.dthibau.ForumApp (17835)

MBean Browser

MBean Tree

Filter:

- Threading
  - java.nio
  - java.util.logging
  - javax.cache
  - metrics
    - HikariPool-1.pool.ActiveConnections
    - HikariPool-1.pool.ConnectionCreation
    - HikariPool-1.pool.ConnectionTimeoutRate
    - HikariPool-1.pool.IdleConnections
    - HikariPool-1.pool.PendingConnections
    - HikariPool-1.pool.TotalConnections
    - HikariPool-1.pool.Usage
    - HikariPool-1.pool.Wait
    - jcache.statistics.org.dthibau.domain.Authority.average-get-time
    - jcache.statistics.org.dthibau.domain.Authority.average-put-time
    - jcache.statistics.org.dthibau.domain.Authority.average-remove-time
    - jcache.statistics.org.dthibau.domain.Authority.cache-evictions
    - jcache.statistics.org.dthibau.domain.Authority.cache-gets
    - jcache.statistics.org.dthibau.domain.Authority.cache-hit-percentage
    - jcache.statistics.org.dthibau.domain.Authority.cache-hits

MBean Features

| Name              | Value         | Update Interval |
|-------------------|---------------|-----------------|
| 50thPercentile    | 0.0           | Default         |
| 75thPercentile    | 0.0           | Default         |
| 95thPercentile    | 0.0           | Default         |
| 98thPercentile    | 0.0           | Default         |
| 999thPercentile   | 0.0           | Default         |
| 99thPercentile    | 0.0           | Default         |
| Count             | 0             | Default         |
| DurationUnit      | milliseconds  | Default         |
| FifteenMinuteRate | 0.0           | Default         |
| FiveMinuteRate    | 0.0           | Default         |
| Max               | 0.0           | Default         |
| Mean              | 0.0           | Default         |
| MeanRate          | 0.0           | Default         |
| Min               | 0.0           | Default         |
| OneMinuteRate     | 0.0           | Default         |
| RateUnit          | events/second | Default         |
| StdDev            | 0.0           | Default         |



# Tableaux de bord

---

*JHipster* propose plusieurs tableaux de bord accessibles avec un rôle **ADMIN** (endpoints Actuator de SpringBoot)

- Métriques JVM, Requête HTTP, Interface REST, Cache, BD
- Health : Statut des différents composants (espace disque, base de données, ELS, ...)
- Configuration SpringBoot
- Audits sécurité
- Niveaux des Loggers
- API via Swagger
- Console BD



# *JHipster registry*

---

***Jhipster Registry*** est un processus séparé de l'application (fat jar téléchargeable) qui a 3 objectifs :

- **Annuaire** des applications (discovery serveur) permettant de localiser les instances d'une application
- Serveur de **configuration** : Les propriétés de configuration des différentes application SpringBoot y sont stockées et gérées
- Serveur **d'administration** : Fourniture des dashboards de monitoring (Architecture recommandée pour le monitoring)

Application centrale d'une architecture micro-services



# Jhipster Console

---

***JHipster Console*** permet d'historiser les métriques et donc d'afficher des graphiques d'évolution en fonction du temps

Il est basé sur ELK et prêt à l'emploi

2 types d'information sont alors envoyés vers un cluster ELK :

- Les traces des applications
- Les métriques



# Configuration

---

```
jhipster:
 logging:
 logstash:
 enabled: true
 host: localhost
 port: 5000
 QueueSize: 512
 metrics:
 logs:
 enabled: true
 reportFrequency: 60 # seconds
```



# Mise en place

---

Le plus simple est d'utiliser docker

Télécharger le Dockerfile

```
curl -O
```

```
https://raw.githubusercontent.com/jhipster/jhipster-console/master/bootstrap/docker-compose.yml
```

Exécuter l'ensemble des services

```
docker-compose up -d
```

Se connecter à l'interface :

```
http://localhost:5601
```



# Third-party

---

*JHipster* permet d'exporter Metrics vers Graphite et Prometheus.

```
jhipster:
 metrics:
 jmx.enabled: true
 graphite:
 enabled: true
 host: localhost
 port: 2003
 prefix: jhipster
 prometheus:
 enabled: true
 endpoint: /prometheusMetrics
```



# *Elastalert*

---

*JHipster Console* intègre ***Elastalert*** et peut générer des alertes en fonctions des données stockées dans *ElasticSearch*

*Elastalert* permet de définir des règles d'alerte pour détecter des erreurs, des pics de charge, ou autre

Il est basé sur des requêtes *Elasticsearch*





# Mise en place

---

Dans ***docker-compose.yml***

```
jhipster-alerter:
 image: jhipster/jhipster-alerter
 #volumes:
 # - ../jhipster-alerter/rules:/opt/elastalert/rules/
 # - ../alerts/config.yaml:/opt/elastalert/config.yaml
```

Dans ***alerts/config.yaml***

```
run_every:
 minutes: 1
buffer_time:
 Minutes: 5
```

Dans ***alerts/rules***

- Définir de nouvelles alertes



# Micro-services

---

## **Architecture micro-services**

Spring Cloud, et ses micro-services  
techniques

Création avec Jhipster, Choix de JHipster



# Introduction

---

Le terme « ***micro-services*** » décrit un nouveau pattern de développement visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel

Les méthodes agiles, la culture *DevOps*, le *PaaS*, les containers d'application et les environnement d'injection de dépendances permettent d'envisager de construire de grand systèmes orientés services de façon modulaire



# Architecture

---

Une architecture micro-services implique la décomposition des applications en très petit services

- faiblement couplés
- ayant une seule responsabilité
- Développés par des équipes full-stack indépendantes.

Le but étant de livrer et maintenir des systèmes complexes avec la rapidité et la qualité demandées par le business digital actuel

On l'appelle également *SOA 2.0* ou *SOA For Hipsters*



# Caractéristiques

---

**Design piloté par le métier** : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

**Principe de la responsabilité unique** : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

**Une interface explicitement publiée** : Un producteur de service publie une interface qui peut être consommée

**DURS (Deploy, Update, Replace, Scale) indépendants** : Chaque service peut être indépendamment déployé, mis à jour, remplacé, scalé

**Communication légère** : REST sur HTTP, STOMP sur WebSocket, ....



# Bénéfices

---

**Scaling indépendant** : les services les plus sollicités (cadence de requête, mutualisation d'application) peuvent être scalés indépendamment (CPU/mémoire ou sharding),

**Mise à jour indépendantes** : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes

**Maintenance facilitée** : Le code d'un micro-service est limité à une seule fonctionnalité

**Hétérogénéité des langages** : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

**Isolation des fautes** : Un service dysfonctionnant ne pénalise pas obligatoirement le système complet.

**Communication inter-équipe renforcée** : Full-stack team



# Contraintes

---

**Réplication** : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

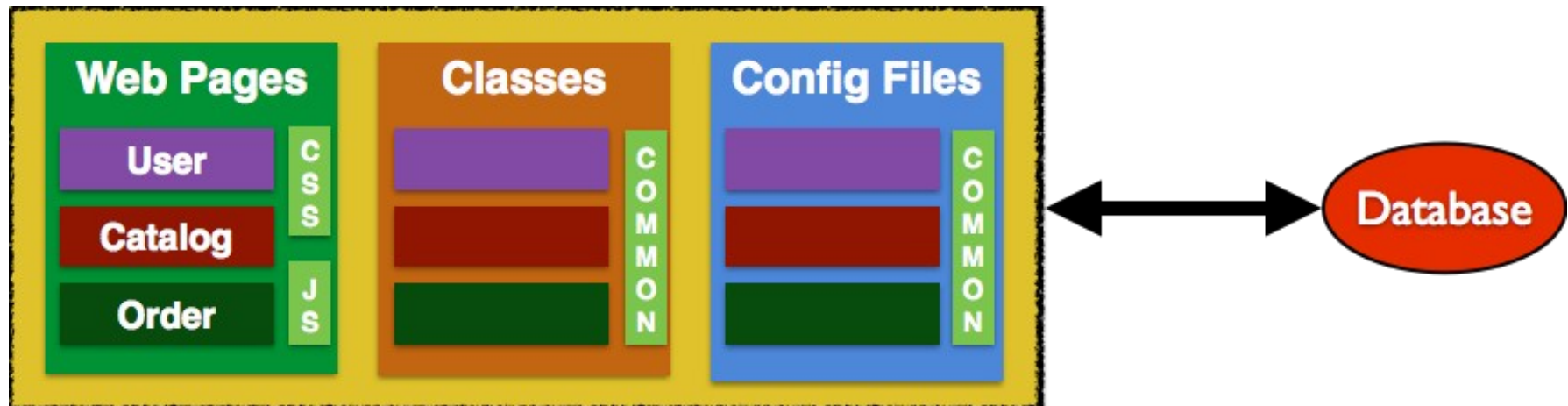
**Découverte automatique** : Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certaines métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

**Monitoring** : Les services sont surveillés en permanence. Des traces sont générées et éventuellement agrégées

**Résilience** : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

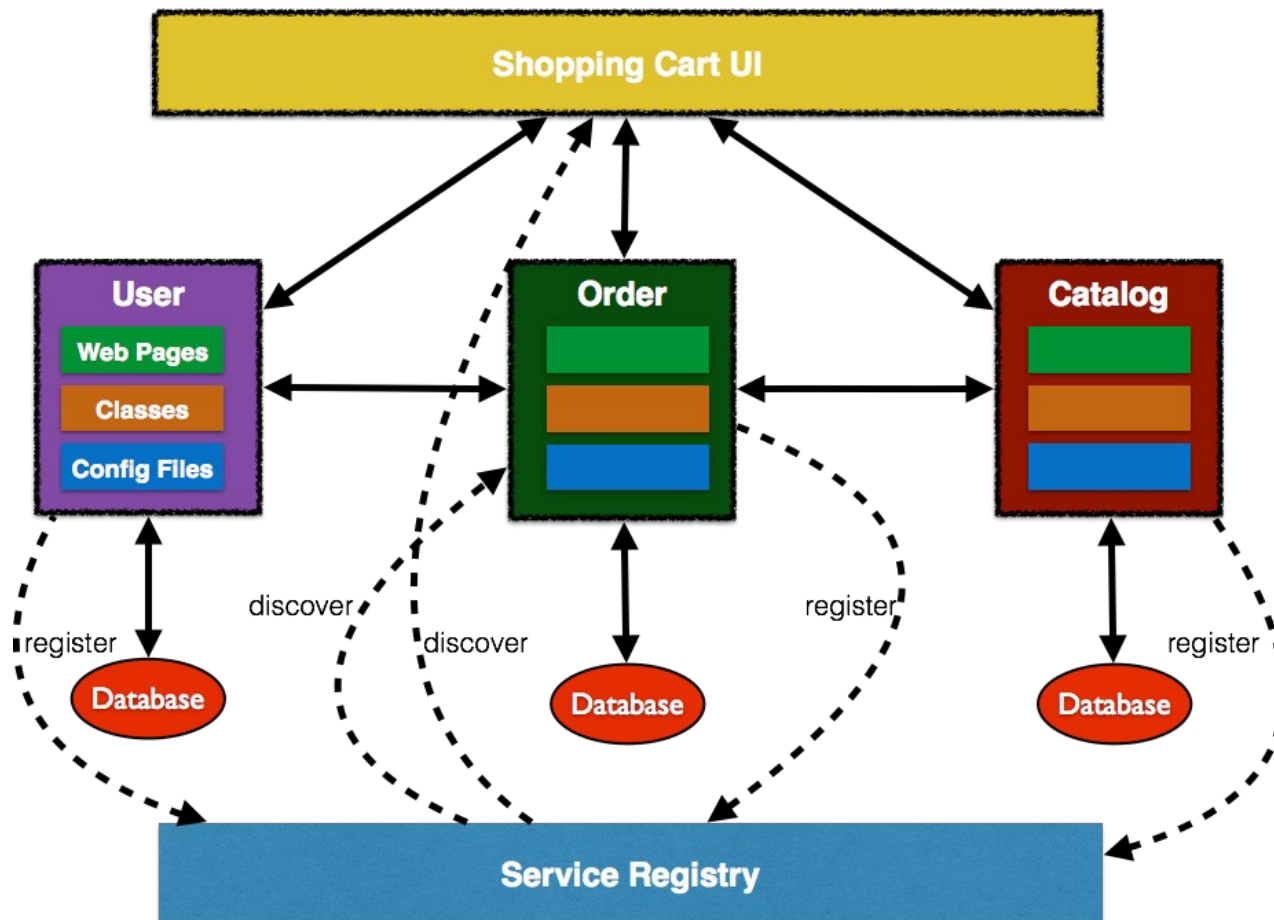
**DevOps** : L'intégration et le déploiement continu sont indispensables pour le succès.

# Architecture monolithique





# Version micro-service





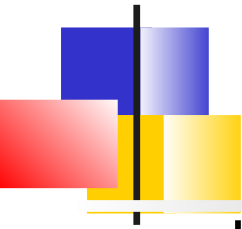
# Micro-services

---

Architecture micro-services

**Spring Cloud, et ses micro-services  
techniques**

Création avec Jhipster, Choix de JHipster

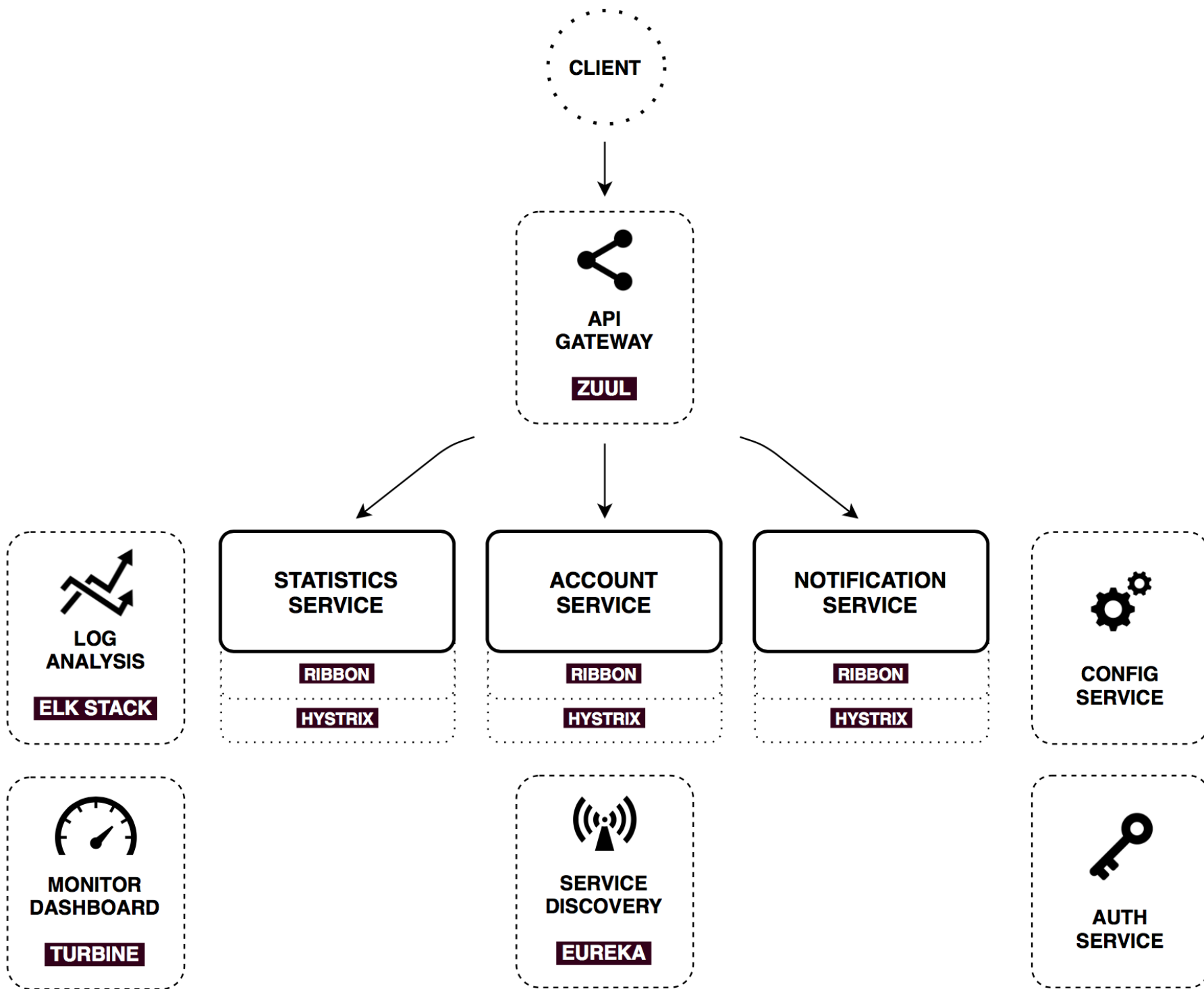
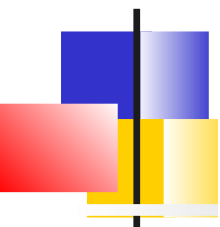


# Micro-services techniques

---

Les architectures micro-services basées sur une distribution massive nécessitent des micro-services d'infrastructure :

- Service **d'annuaire** permettant de localiser un micro-service disponible
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Services **d'authentification** offrant une fonctionnalité de SSO parmi l'ensemble des micro-services
- Service de **monitoring** surveillant la disponibilité, centralisant les fichiers journaux
- Service de répartition de charge, de gestion d'appartenance au cluster, d'élection de maître, ...
- Service de session applicative, horloge synchronisée,





# Spring Cloud

---

**@EnableConfigServer** : Gère les configurations des différents micro-services typiquement dans un repository Git

**@EnableDiscoveryServer** : Service de discovery Eureka, Consul ou Zookeeper

**@EnableZuulProxy** : Routage de requêtes HTTP

**@EnableFeignClients** : Client REST (communication entre micro-services)

**@RibbonClient** : Répartiteur de requêtes côté client

**@HystrixCommand** : Fall-back en cas de défaillance d'un micro-service

**Turbine et Hystrix** : Monitoring



# Micro-services

---

Architecture micro-services  
Spring Cloud, et ses micro-services  
techniques  
**Création avec Jhipster, Choix de  
JHipster**



# Architecture

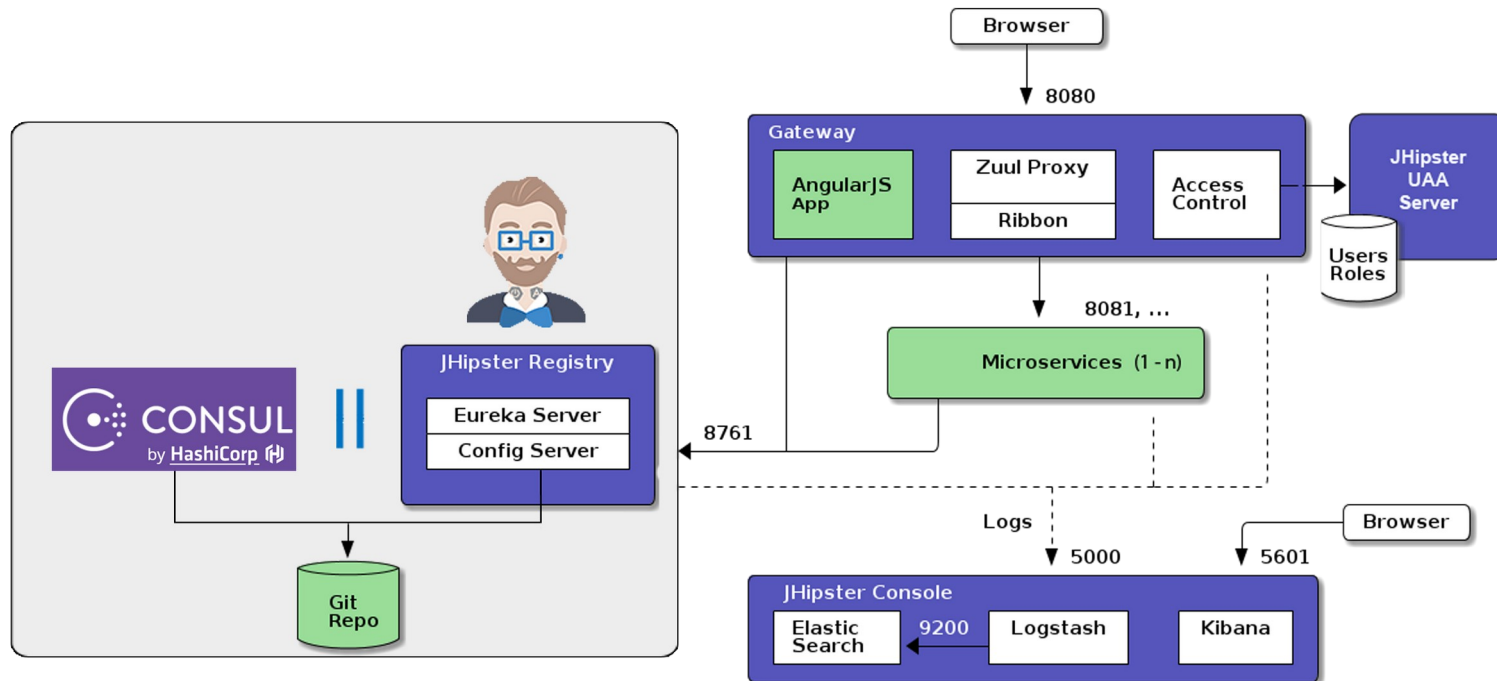
---

L'architecture micro-services de JHipster est composée de :

- Une **gateway** : application générée par Jhipster qui gère le trafic http et offre une application Angular.
- **Traefik** un reverse proxy HTTP et un répartiteur de charge pouvant fonctionner avec la gateway.
- **JHipster Registry** ou **Consul** : Discovery et monitoring
- **JHipster UAA** serveur d'authentification et autorisation OAuth2.
- Les **micro-services** applicatifs générés par Jhipster
- **JHipster Console** pour la surveillance et les alertes.

# Architecture

NETFLIX | OSS +  +  docker



 elastic +  logstash +  kibana





# Nature *JHipster*

---

Les micro-services sont des applications *JHipster* qui n'ont pas de front-end

Les front-end Angular doivent alors être générés à partir d'une gateway

Une *gateway* est une application JHipster qui est le point d'entrée à un ensemble de micro-services

Le workflow est alors :

- Générer les entités dans l'application micro-service
- Générer l'entité sur la *gateway* et indiquer le micro-service existant



# Merci !!

---

Pour votre attention !