



# Exploiter des applications Spring Boot

---

David THIBAU – 2020

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- **Introduction**

- Rappels Spring Core
- L'offre Spring Boot
- Les applications SpringBoot
- Alternatives pour le déploiement
- Plugins Maven ou Gradle

- **SpringBoot**

- L'auto-configuration
- L'offre des starters, le starter actuator
- Surcharge de la configuration par défaut
- Profils
- Traces d'une application

- **Déploiement d'applications**

- Déploiement immuable et approche DevOps
- Alternatives pour l'infrastructure
- Mise en service, support et configuration
- Spring Boot et Docker : support
- Pipeline DevOps typique

- **Exploitation**

- Configuration de production
- Métriques à surveiller, Configuration de actuator
- Spécificité Kubernetes



# Introduction

---

## **Rappels Spring**

L'offre Spring Boot

Les applications Spring Boot

Alternatives pour le déploiement

Build avec Maven ou Gradle, plugins



# Historique

---

- ❖ *Spring* est un projet **OpenSource** avec un modèle économique basé sur le service (Support, Conseil, Formation, Partenariat et certifications)
- ❖ La société **SpringSource** fondée par les créateurs de Spring (Rod Johnson et Juergen Hoeller) a été rachetée par **VmWare**, puis intégrée dans la joint-venture **Pivotal Software**
- ❖ Succès de la solution : Perçu comme une alternative aux serveurs Java EE



# Projets Spring

---

*Spring* est en fait un ensemble de projets adaptés à toutes les problématiques actuelles.

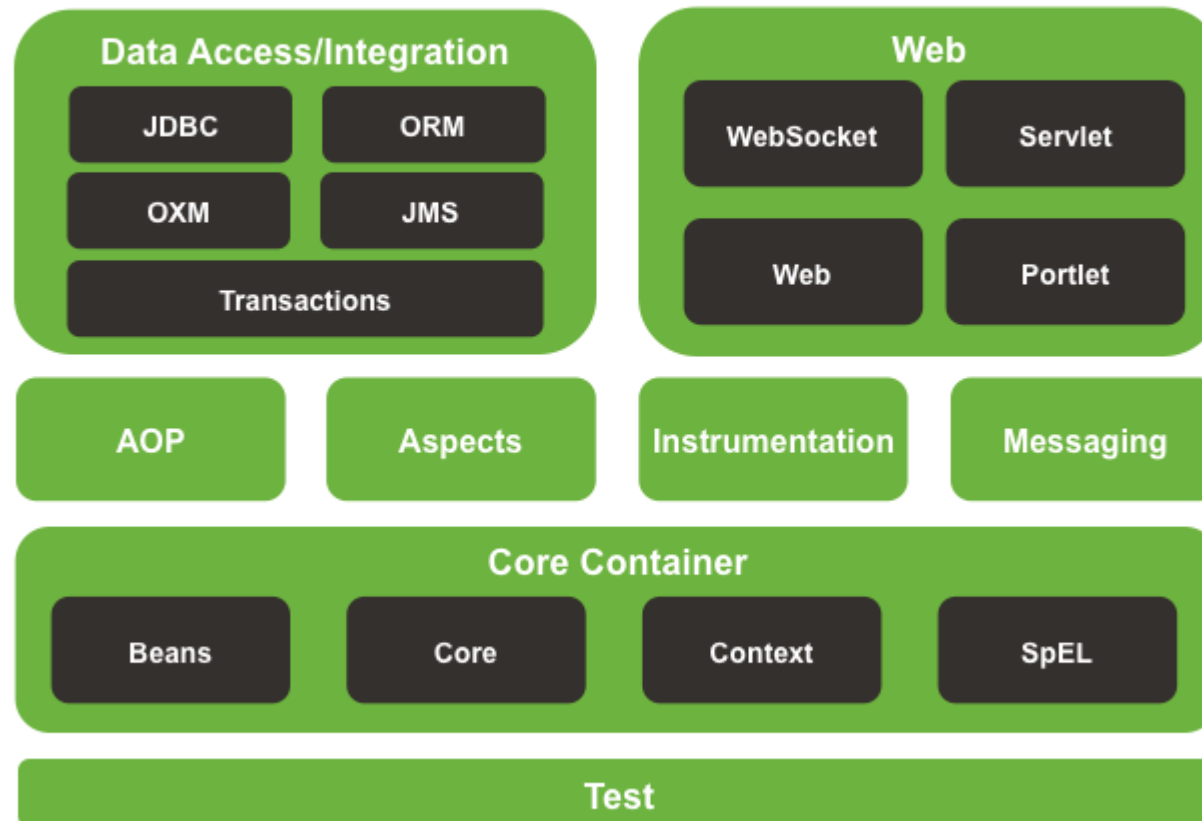
Tous ces projets ont comme objectifs :

- ✓ Permettre d'écrire du code propre, modulaire et testable
- ✓ Éviter d'avoir à coder les aspects techniques (plomberie)
- ✓ Être portable : déployer une application stand-alone, sur un serveur en Paas du moment qu'il y ait une JVM
- ✓ Favoriser le *stateless* : Client/serveur sans conservation d'états

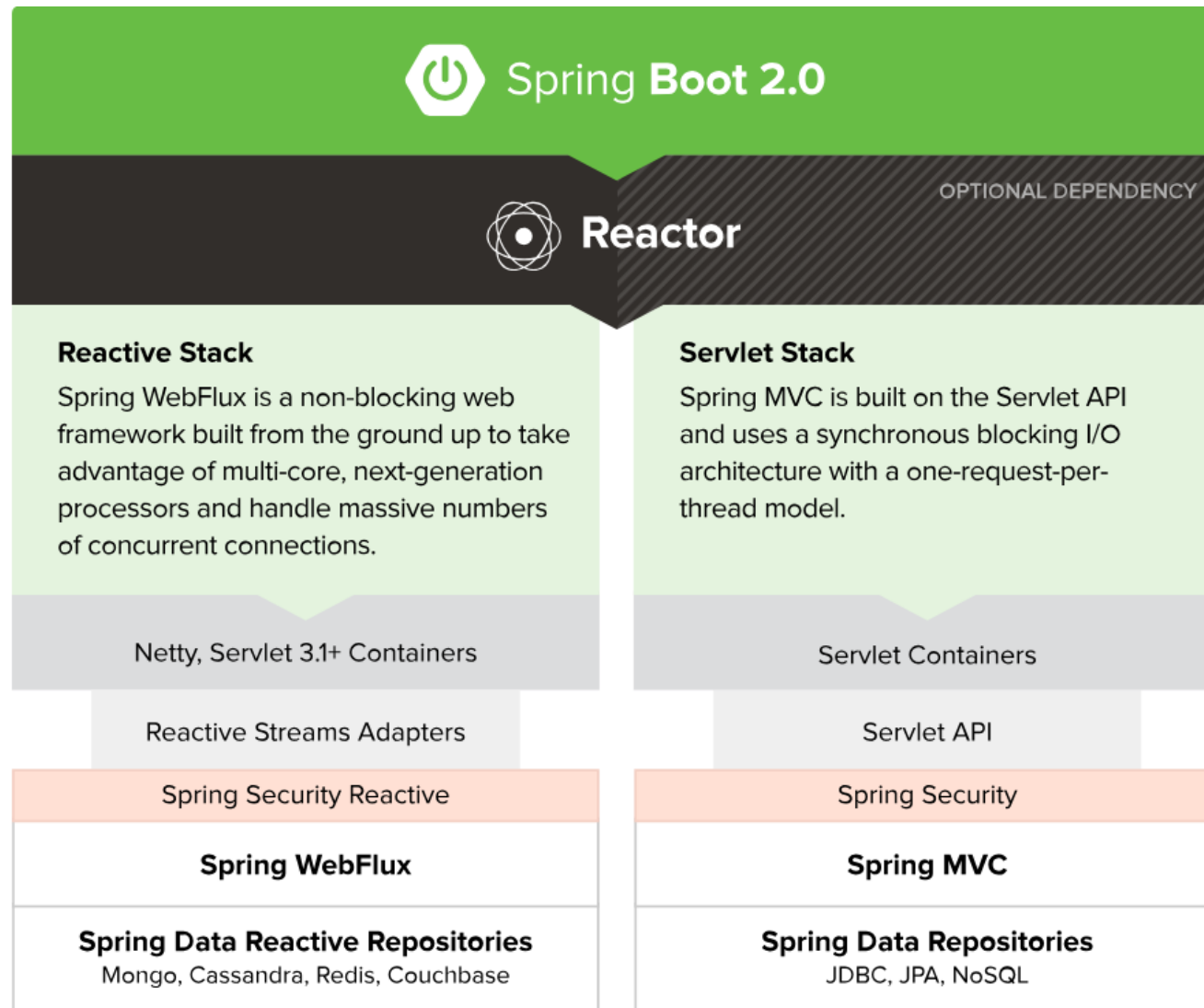
# Spring framework 4



## Spring Framework Runtime



# Spring 5.x et SB 2.0





# Pattern IoC

---

Pour enlever la dépendance, on délègue l'instanciation de l'implémentation à un framework (Spring, Serveur JavaEE).

Le framework utilise des données de configuration pour trouver l'implémentation à instancier

=> C'est le framework qui prend le contrôle des instanciations

=> Il **injecte** l'implémentation dans la classe contrôleur





# Types de configuration

---

- ❖ Différents choix sont possibles pour configurer le container :
  - **Fichier de configuration XML :**  
Changement sans recompilation
  - **Annotations Java :**
    - Définition de beans
    - Classe de configuration contenant des méthodes instanciant les beans
    - Spécification des injections



# Configuration XML

---

```
<beans>
```

```
  <bean id="MovieLister" class="spring.MovieLister">
```

```
    <property name="finder">
```

```
      <ref local="MovieFinder"/>
```

```
    </property>
```

```
  </bean>
```

```
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
```

```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```



# Configuration via annotations

---

**@Controller**

```
public class MovieLister {  
    MovieFinder finder ;
```

**@Autowired**

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder ;  
}
```

```
public List<Movie> moviesDirectedBy(String arg) {  
    List<Movie> allMovies = finder.findAll();  
    List<Movie> ret = new ArrayList<Movie>() ;  
    for (Movie movie : allMovies ) {  
        if (!movie.getDirector().equals(arg))  
            ret.add(movie);  
    }  
    return ret;  
}
```



# Classe de Configuration

---

## @Configuration

```
public class SimpleConfiguration {  
    @Autowired  
    Connection connection;  
  
    @Bean  
    Database getDatabaseConnection(){  
        return connection.getDBConnection();  
    }  
    // Mode code here....  
}
```



# Avantages de l'injection de dépendances

---

- ❖ L'injection de dépendance apporte d'importants bénéfices :
  - Les composants applicatifs sont **plus faciles à écrire**
  - Ils sont **plus faciles à tester**.  
Les implémentations mock peuvent être injectés lors des tests.
  - Le **typage** des objets est **préservé**.
  - Les **dépendances sont explicites** (à la différence d'une initialisation à partir d'un fichier *properties* ou d'une base de données)
  - La plupart des classes applicatives **ne dépendent pas de l'API du conteneur** et peuvent donc être utilisées avec ou sans le container.
  - Le framework peut injecter des implémentations ayant des **services techniques transverses** (Transaction, sécurité, Trace, Profiling)



# Exemples, services transverses

---

Avec un framework *IoC* comme Spring, un développeur peut :

- Écrire une méthode s'exécutant dans une transaction base de données sans utiliser l'API de transaction
- Rendre une méthode accessible à distance sans utiliser une API remote
- Définir une méthode de gestion applicative sans utiliser JMX
- Définir une méthode gestionnaire de message sans utiliser JMS



# Cycles de vie des objets gérés

- ❖ Les objets instanciés et gérés par Spring peuvent suivre 3 types de cycle de vie :
  - **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des services « stateless »  
90 % des beans développés
  - **Prototype** : A chaque fois que l'objet est demandé via son nom, une instance est créée.  
Très peu utilisé
  - **Custom object “scopes”** : Durée de vie liée à une autre objet (Exemple une requête HTTP, une session, une connexion BD, etc.=)  
En général, fourni par Spring pour s'intégrer à un service externe au conteneur



# Introduction

---

Rappels Spring

## **L'offre Spring Boot**

Les applications Spring Boot

Alternatives pour le déploiement

Build avec Maven ou Gradle, plugins





# Introduction

---

Spring Boot a été conçu pour **simplifier le démarrage** et le développement de nouvelles applications Spring

- ne nécessite aucune configuration XML
- Dès la première ligne de code, on a une application fonctionnelle

=> Offrir une expérience de développement simplifiant à l'extrême l'utilisation des technologies existantes



# Essence

---

Spring Boot est un ensemble de bibliothèques qui sont exploitées par un système de build et de gestion de dépendances ( **Maven** ou **Gradle** )

Les bibliothèques sont groupées en des *starter modules*

Toutes les librairies sont ensuite packagées avec le code applicatif dans un **fat-jar** : l'exécutable



# Auto-configuration

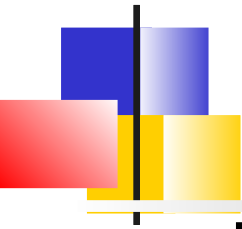
---

Le concept principal de *SpringBoot* est l'**auto-configuration**

*SpringBoot* est capable de détecter automatiquement la nature de l'application et de configurer les beans Spring nécessaires

- Cela permet de démarrer rapidement et de graduellement surcharger la configuration par défaut pour les besoins de l'application

Les mécanismes sont différents en fonction du langage : Groovy, Java ou Kotlin



# Java

## Gestion des dépendances

---

Dans un environnement Java, Spring Boot simplifie la gestion de dépendances et de leurs versions :

- Il organise les fonctionnalités de Spring en modules.  
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des "**starter**" **modules**.
- Il fournit un mécanisme permettant de gérer facilement les versions des dépendances.
- Il propose l'interface web "**Spring Initializr**", qui peut être utilisée pour générer des configurations Maven ou Gradle



# Auto-configuration (Java)

---

En fonction des librairies présentes dans l'exécutable Java, Spring Boot crée tous les beans techniques nécessaires avec une configuration par défaut.

- Par exemple, si il s'aperçoit que des librairies Web sont présentes, il démarre un serveur Tomcat embarqué sur le port 8080 et configure tous les beans techniques permettant de développer une application Web ou API Rest
- Si il s'aperçoit que le driver Postgres est dans le classpath, il crée automatiquement un pool de connexions vers la base



# Personnalisation de la configuration

---

La configuration par défaut peut être surchargée par différents moyens

- Des **variables d'environnement**
- Des **arguments de la ligne de commande**
- Des **fichiers de configuration externe** (*.properties* ou *.yml*) présent dans l'archive ou accessible via http.  
On peut fournir différents fichiers qui seront activés en fonction de profils
- Du code en utilisant des **classes spécifiques du framework** (exemple *AuthenticationManagerBuilder* pour gérer l'authentification)



# Introduction

---

Rappels Spring

L'offre Spring Boot

**Les applications Spring Boot**

Alternatives pour le déploiement

Build avec Maven ou Gradle, pluginsS



# Introduction

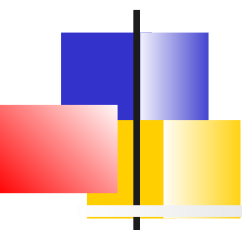
---

La majorité des applications actuellement développées sont des API REST s'appuyant sur un système de persistance :

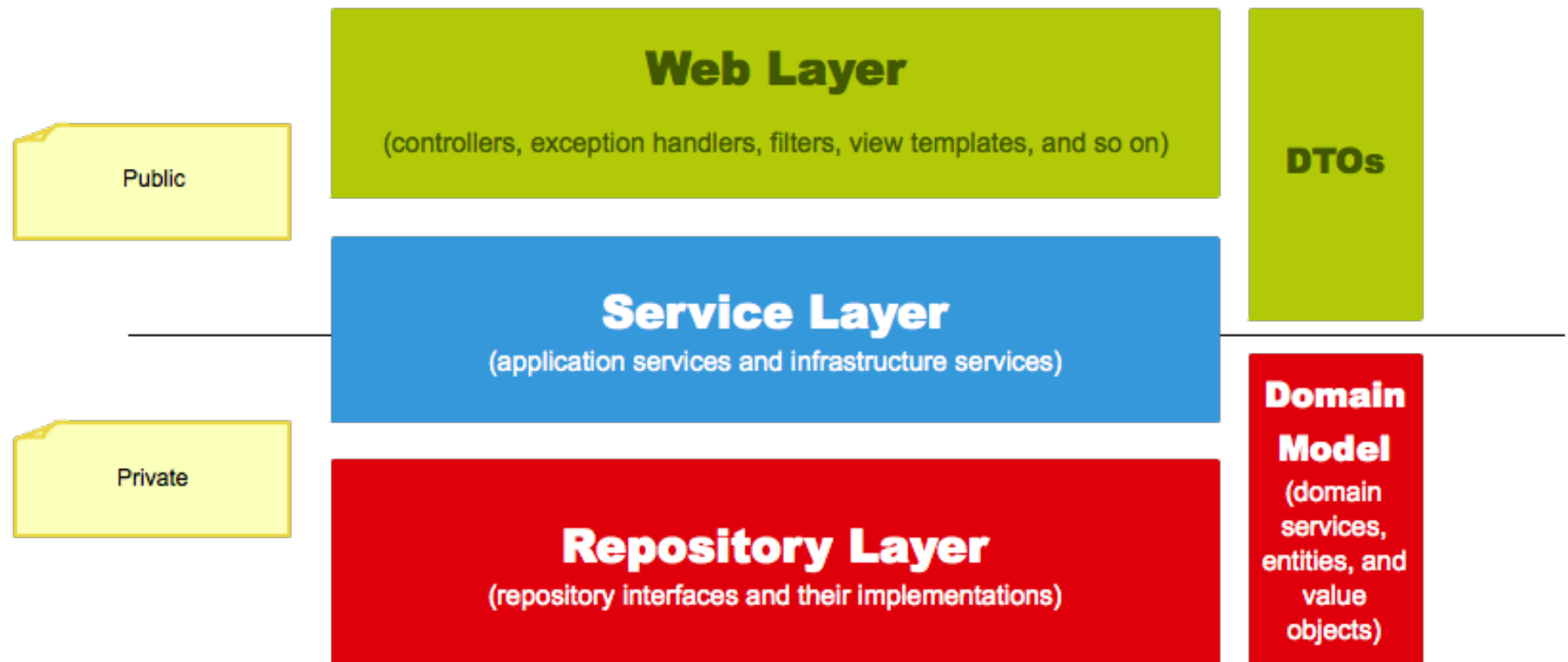
- Base de données relationnelles
- NoSQL
- Messagerie (Kafka, RabbitMQ)

Spring Boot est également adapté à des applications batch (tâches planifiées, traitement par lots, etc.)

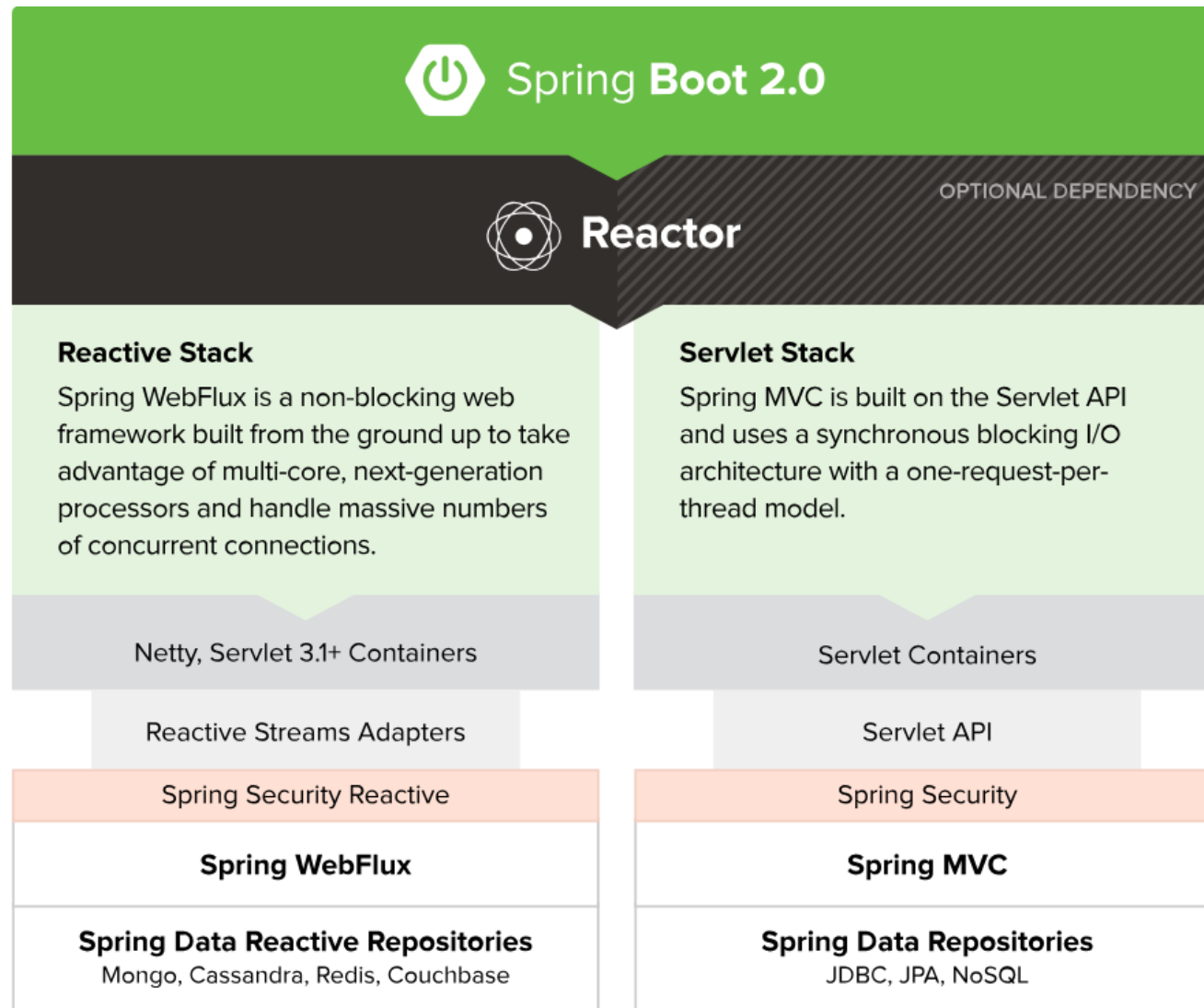




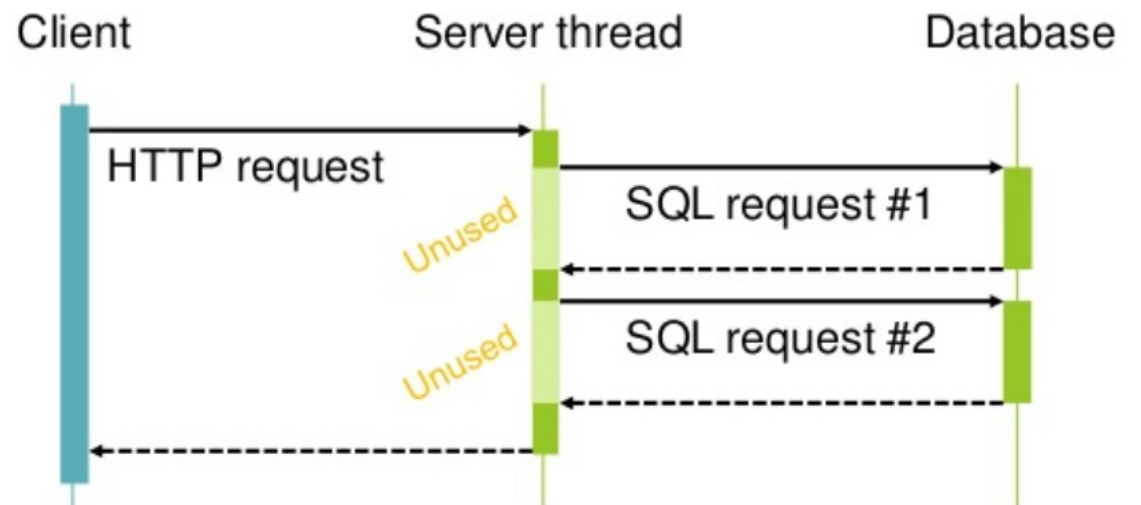
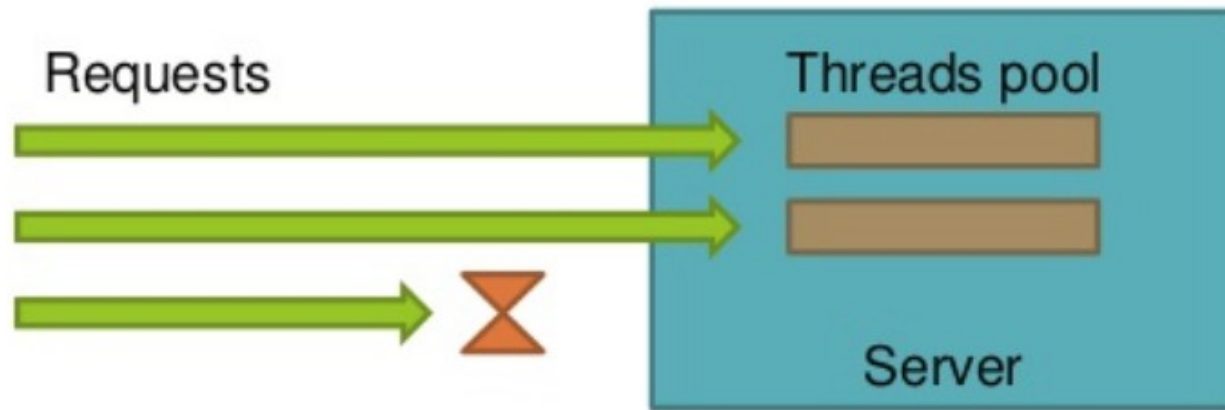
# Architecture classique projet



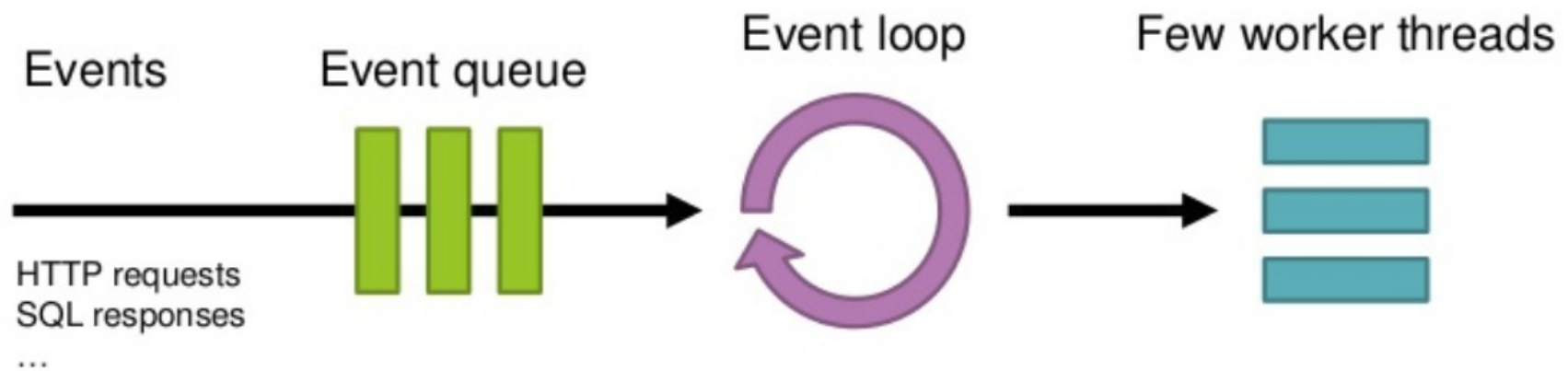
# Les 2 stacks



# Modèle bloquant



# Modèle réactif





# Motivation pour le modèle réactif

---

2 principales motivations pour Spring Webflux (modèle réactif) :

- Le besoin d'un stack non-bloquante permettant de gérer la concurrence avec peu de threads et de scaler avec moins de ressources CPU/mémoire
- La programmation fonctionnelle



# Publication de l'API via Swagger

En général, une API Rest est documentée via un outil de type Swagger. L'intégration de Swagger avec *SpringBoot* est assez directe.

## Api Documentation <sup>1.0</sup>

[ Base URL: plbsi-rec.plb.fr:9443/ ]

<https://plbsi-rec.plb.fr:9443/v2/api-docs?group=B-cms-account-offre>

Api Documentation

[Terms of service](#)

[Apache 2.0](#)

### account-resource Account Resource

GET

/api/backoffice/accounts getAll

POST

/api/backoffice/accounts createAccount

DELETE

/api/backoffice/accounts/{id} deleteAccount

PATCH

/api/backoffice/accounts/{id} updateAccount

### gabarit-resource Gabarit Resource

### logged-user-resource Logged User Resource



# Architecture micro-services

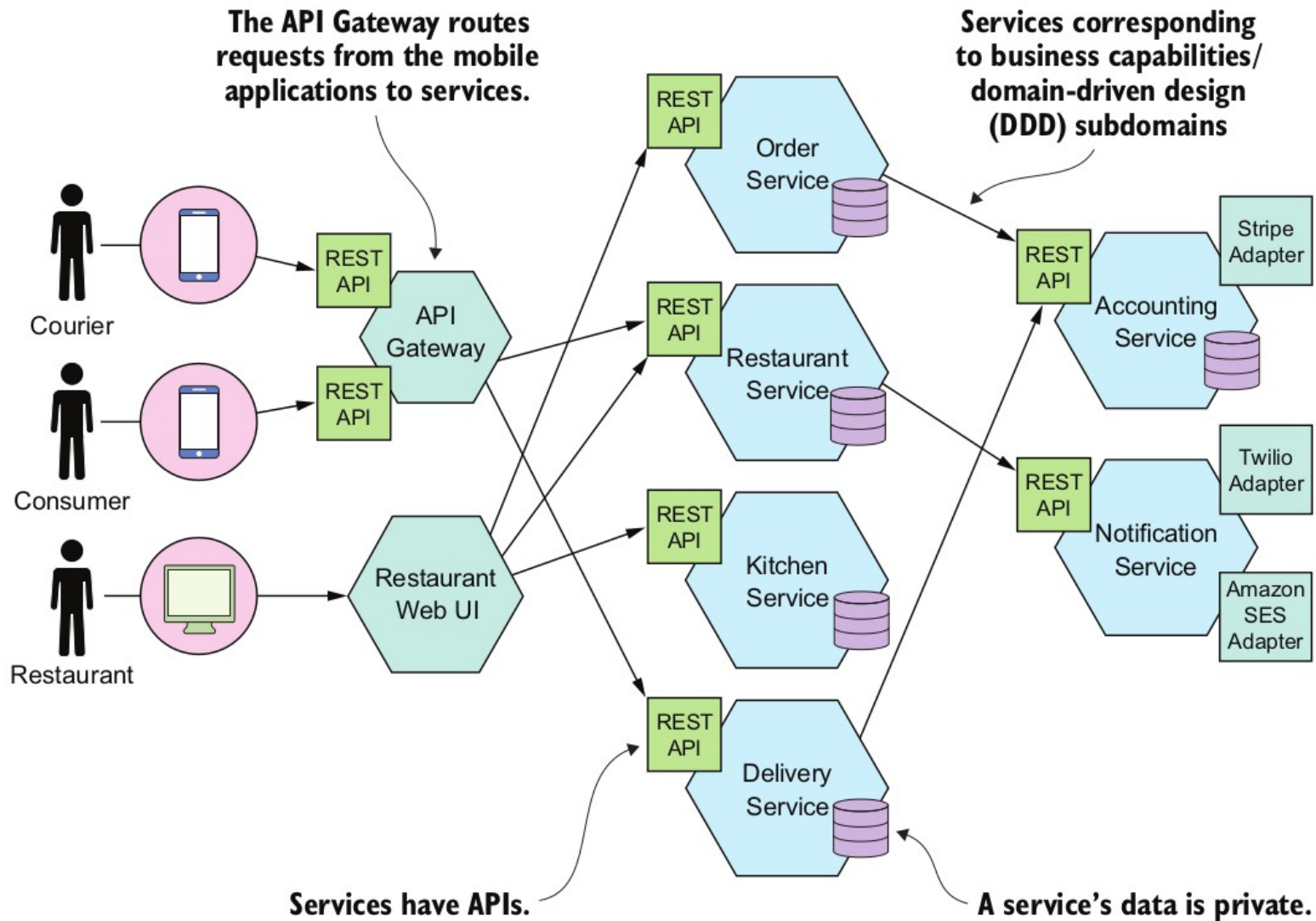
---

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « **micro-services** »

L'application métier est alors décomposée en de nombreux petits services exposant leur API Rest

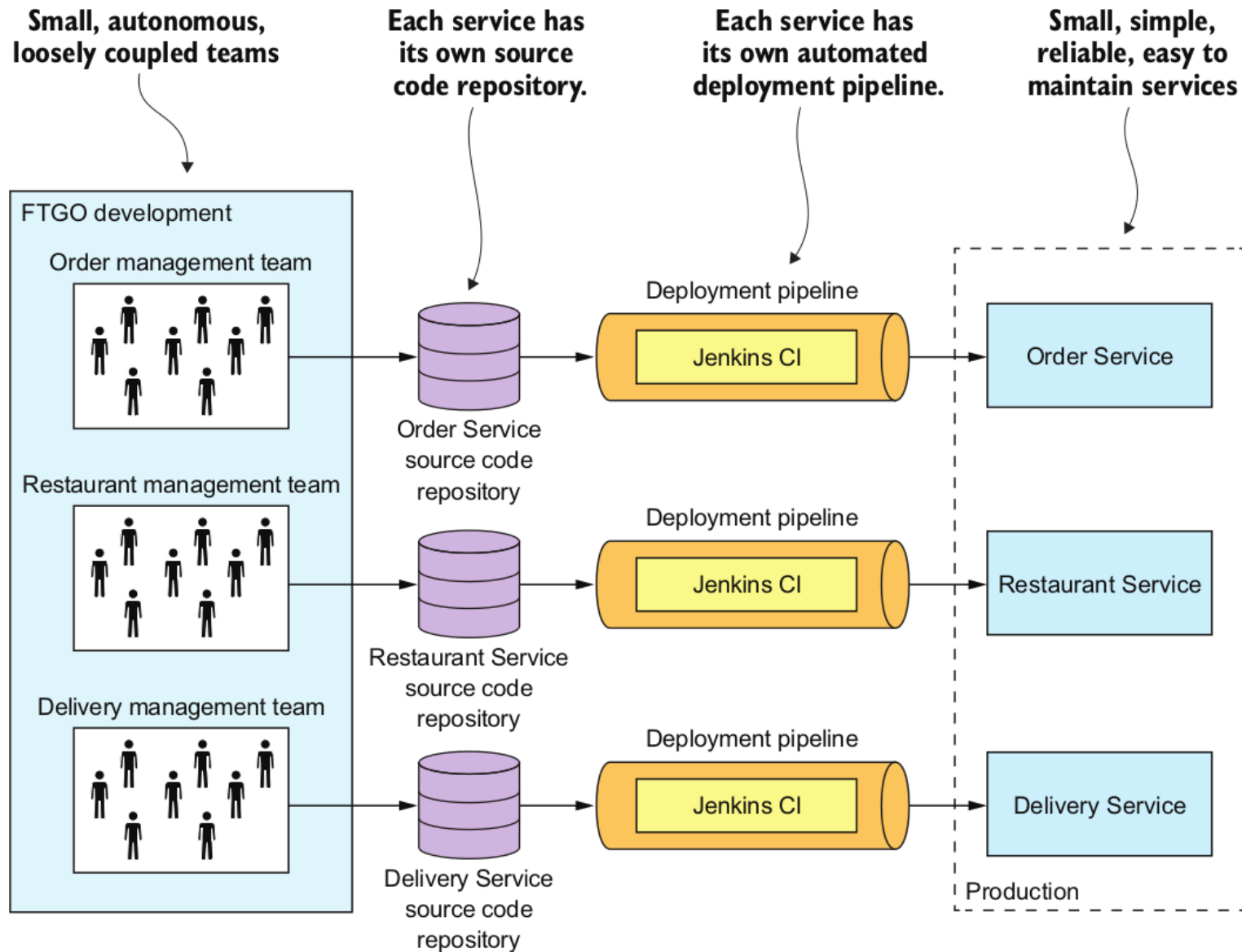
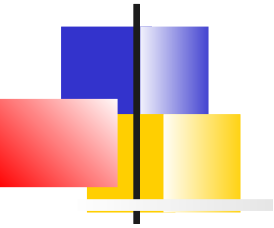
- Chaque service expose une API REST  
=> Couplage faible
- Chaque service implémente une seule fonctionnalité métier  
=> Meilleur évolutivité
- Chaque service est développés par une équipe DevOps indépendantes  
=> Meilleur maîtrise de l'application

# Une architecture micro-service





# Organisation DevOps





# Sécurité des applications Web et des APIs Rest

---

Les application web (stateful) et les APIs REST (stateless) n'ont pas la même stratégie pour la gestion de la sécurité.

- Dans une application stateful, les informations liées à l'authentification sont stockées dans la session utilisateur (cookie).
- Dans une application stateless, les droits de l'utilisateur sont transmis à chaque requête via un jeton
  - Le jeton est fourni par l'application
  - Ou par un service externe. Exemple *oAuth2*



# Processus d'authentification appli web back-end

---

1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier :
  1. En redirigeant vers une page de login
  2. En fournissant les entêtes pour une authentification basique du navigateur .
3. Le navigateur renvoie une réponse au serveur :
  1. Soit le POST de la page de login
  2. Soit les entêtes HTTP d'authentification.
4. Le serveur décide si les crédits sont valides :
  1. si oui. L'authentification est stockée dans la session, la requête originelle est réessayée, si les droits sont suffisants la page est retournée sinon un code 403
  2. Si non, le serveur redemande une authentification.
5. L'objet *Authentication* contenant l'utilisateur et ses rôles est présent dans la session. Il est récupérable à tout moment par `SecurityContextHolder.getContext().getAuthentication()`

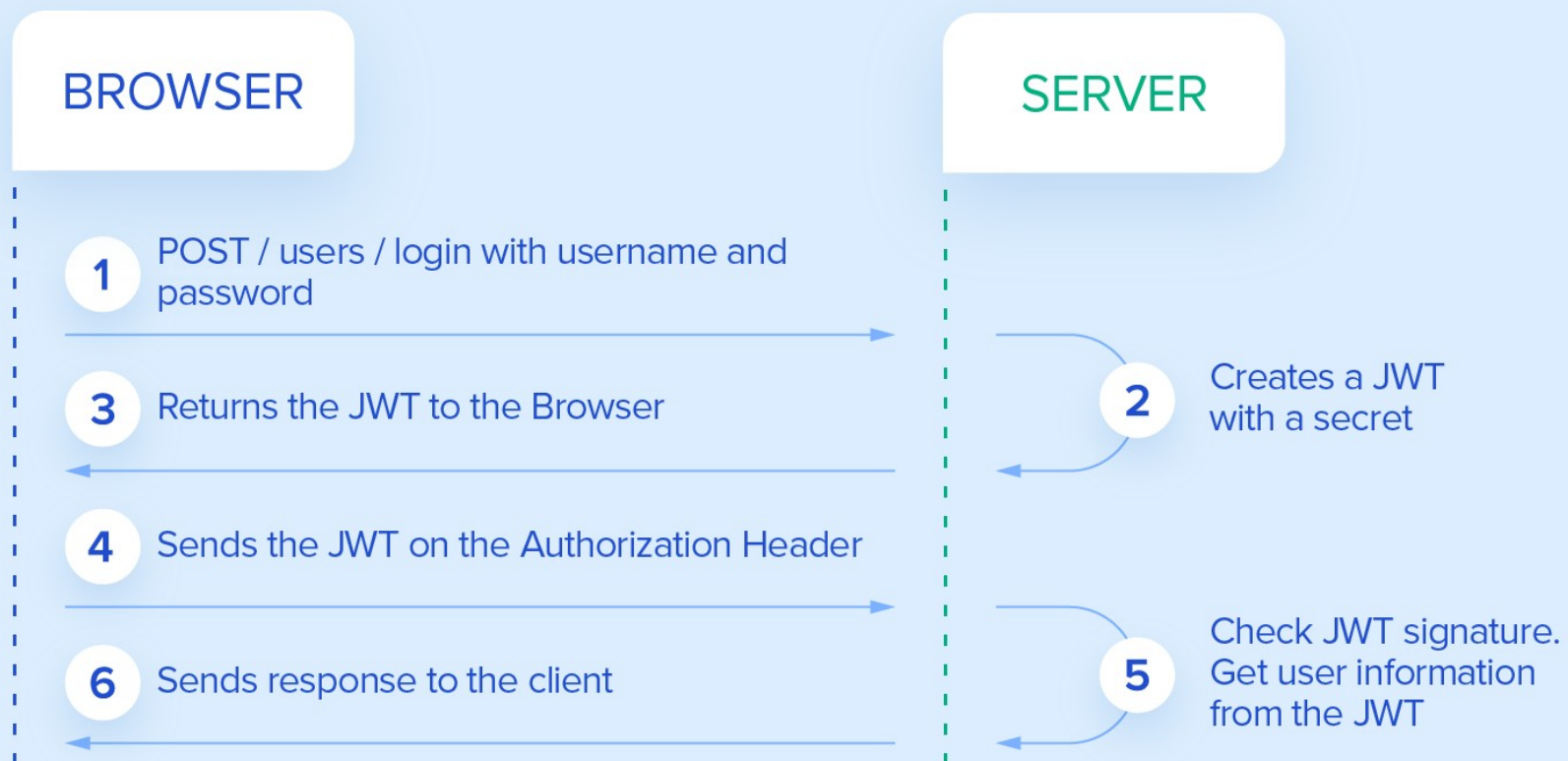


# Processus d'authentification appli REST

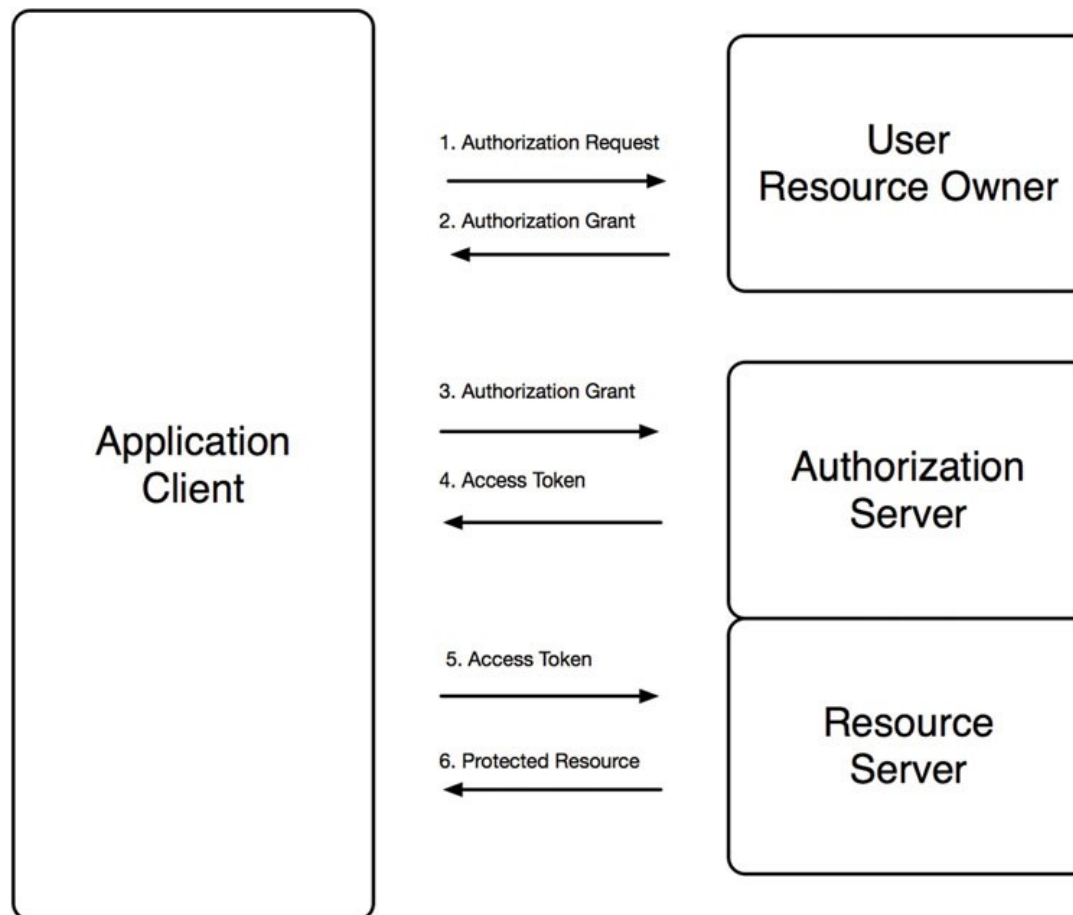
---

1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier en envoyant une réponse 403.
3. Le navigateur propose un formulaire de login puis envoie le formulaire sur un serveur d'authentification (peut être différent que le serveur d'API)
4. Le serveur d'authentification décide si les créden-tiels sont valides :
  1. si oui. Il génère un token avec un délai de validité
  2. Si non, le serveur redemande une authentification .
5. Le client récupère le jeton et l'associe à toutes les requêtes vers l'API
6. Le serveur de ressources décrypte le jeton et déduit les droits de l'utilisateur. Il autorise ou interdit l'accès à la ressource

# Authentication Rest



# Protocole *oAuth2*





# Support *oAuth* de SpringBoot

---

3 starters sont désormais fourni :

- ***OAuth2 Client*** : Intégration pour utiliser un login *oAuth2* fournit par Google, Github, Facebook, ...
- ***OAuth2 Resource server*** : Application permettant de définir des ACLs par rapport aux scopes client et aux rôles contenu dans des jetons *oAuth*
- ***Okta*** : Pour travailler avec le fournisseur *oAuth* Okta



# Solutions pour un serveur d'autorisation

---

- Utiliser un produit autonome
- Un projet Spring pour un serveur d'autorisation est en cours :  
<https://github.com/spring-projects-experimental/spring-authorization-server>
- Une autre alternative est d'embarquer une solution *oAuth* comme *KeyCloak* dans un application SpringBoot  
Voir par exemple :  
<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app>





# Introduction

---

Rappels Spring  
L'offre Spring Boot  
Les applications Spring Boot  
**Alternatives pour le déploiement**  
Build avec Maven ou Gradle, plugins



# Pré-requis

---

La seule contrainte pour exécuter une application Spring est de disposer d'une **JRE (Java Runtime Environment)**

Plusieurs alternatives pour le déploiement :

- Application stand-alone démarré par un script
- Archive war à déployer sur serveur applicatif
- Service Linux ou Windows
- Exécution d'une image Docker sur une infrastructure Kubernetes par exemple
- Déploiement sur le cloud à partir des sources et les buildpack (CloudFoundry, Heroku)



# Application stand-alone

---

Les plugins Maven ou Gradle permettent de générer l'application stand-alone :

`mvnw package`

`gradlew bootJar`

L'archive exécutable contient alors les classes applicatives et les dépendances

Pour l'exécuter :

`java -jar artifactId-version.jar`



# Fichier Manifest

---

Manifest-Version: 1.0

Implementation-Title: documentService

Implementation-Version: 0.0.1-SNAPSHOT

Archiver-Version: Plexus Archiver

Built-By: dthibau

**Start-Class: org.formation.microservice.documentService.DocumentsServer**

Implementation-Vendor-Id: org.formation.microservice

Spring-Boot-Version: 1.3.5.RELEASE

Created-By: Apache Maven 3.3.9

Build-Jdk: 1.8.0\_121

Implementation-Vendor: Pivotal Software, Inc.

**Main-Class: org.springframework.boot.loader.JarLauncher**



# Format des artefacts

---

Le format des artefacts fournis par les développeurs dépend donc de l'infrastructure de production :

- Machines matérielles ou virtuelles provisionnées avec Java: Fat Jar
- Cluster Kubernetes : Images OCI (Docker)
- CloudFoundry : Les sources



# Exécution dans les différents environnements

---

Plusieurs moyens sont disponibles pour adapter l'exécution à l'environnement (intégration, staging, production, etc.)

- Variables d'environnement.
- Arguments de la commande de démarrage
- Configuration externe.  
Exemple les ConfigMap de Kubernetes



# Rappels, paramètres JVM

---

En dehors de la configuration applicative, la commande de démarrage passe des options à la JVM

- Dimensionnement mémoire -Xmx, -Xms
- Garbage collector
- Propriétés système -D
- ...



# Choix du conteneur

---

Les APIs REST démarrent un serveur web embarqué par défaut Tomcat ou Netty en mode réactif.

Il est possible d'en utiliser d'autres

Par exemple, pour utiliser *undertow* à la place de Tomcat

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions><exclusion>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
  </exclusion></exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```





# Configuration du conteneur

---

Les conteneurs peuvent être configurés par de nombreuses propriétés

- Réseau : *server.port, server.address*
- Session : *server.session.persistence, server.session.timeout, server.session.store-dir, server.session.cookie.\**.
- Gestion des erreurs : *server.error.path*
- SSL
- Compression HTTP



# Démarrage d'une application Spring Boot

---

Le démarrage d'une application SpringBoot consiste en les étapes suivantes :

- Chargement des classes de SpringBoot
- Chargement de la classe principale applicative
- Détection des auto-configurations
- Création de beans pour chaque auto-configuration. (En général, démarrage d'un serveur web ou autre)
- Scan à la recherche des classes de Configuration et beans applicatifs



# Introduction

---

Rappels Spring

L'offre Spring Boot

Les applications Spring Boot

Alternatives pour le déploiement

**Build avec Maven ou Gradle, plugins**



# Introduction

---

Les plugins Maven/Gradle permettent de :

- créer un package d'archives jar ou war exécutables,
- d'exécuter des applications Spring Boot,
- de générer des informations de build
- de démarrer une application Spring Boot avant d'exécuter des tests d'intégration.



# Mise en place Maven

---

```
<!-- Héritage de Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
<properties>
  <java.version>1.8</java.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



# Objectifs Maven

---

***spring-boot:repackage*** : Repackage le jar applicatif en un fat jar

***spring-boot:build-image*** : Package une application en image OCI (docker) en utilisant buildpack.

***spring-boot:build-info*** : Génère un fichier *build-info.properties* contenant les informations de version du pom.xml

Et aussi :

- *spring-boot:help* : Aide
- *spring-boot:run* : Exécute l'application
- *spring-boot:start* : Exécute l'application en background, permet l'exécution de tests d'intégration
- *spring-boot:stop* : Arrête l'application en background



# Mise en place Gradle

---

```
plugins {  
    id 'org.springframework.boot' version '2.3.1.RELEASE'  
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'  
    id 'java'  
}  
group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '1.8'  
  
repositories {  
    mavenCentral()  
}  
springBoot {  
    BuildInfo() // Nécessaire pour la tâche bootBuildInfo  
}  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter'  
    testImplementation('org.springframework.boot:spring-boot-starter-test') {  
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'  
    }  
}  
test {  
    useJUnitPlatform()  
}
```



# Tâches Gradle

---

***bootJar*** permet de construire le jar

***bootWar*** permet de construire un war

***bootBuildInfo*** permet de construire le  
fichier *build-info.properties*

Et aussi :

- *bootRun* : Permet de démarrer l'application





# Spring Boot

---

## **L'auto-configuration**

Starters Modules

Propriétés de configuration

Profils, activation de profils

Configuration des traces



# Dépendances

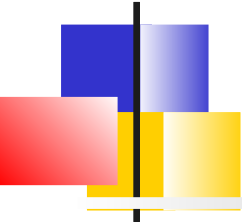
---

Chaque version de Spring Boot fournit sa liste de dépendances.

- 1 seule numéro de version fixe l'ensemble des versions des dépendances utilisées .
- Lors d'une mise à jour de SpringBoot, toutes les dépendances sont mises à niveau de manière cohérente.

La liste des dépendances est importée dans le projet via

- le *pom* parent (Maven)
- le plugin *io.spring.dependency-management* (Gradle)



# Exemple Maven

---

```
<!-- Héritage de Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
<properties>
  <java.version>1.8</java.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



# Exemple Gradle

---

```
plugins {  
    id 'org.springframework.boot' version '2.3.1.RELEASE'  
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'  
    id 'java'  
}  
  
group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '1.8'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter'  
    testImplementation('org.springframework.boot:spring-boot-starter-test') {  
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'  
    }  
}  
  
test {  
    useJUnitPlatform()  
}
```



# Auto-configuration

---

Le mécanisme d'auto-configuration de Spring Boot est implémenté à base :

- De classe **@Configuration** classiques qui définissent des beans
- Et des annotations **@Conditional** qui précisent les conditions pour que la configuration s'active

Ainsi au démarrage de Spring Boot, les conditions sont évaluées et si elles sont respectées, les beans d'intégration correspondant sont instanciés et configurés.



# Annotations conditionnelles

---

Les conditions peuvent se basées sur :

- La présence ou l'absence d'une classe :  
**@ConditionalOnClass** et **@ConditionalOnMissingClass**
- La présence ou l'absence d'un bean :  
**@ConditionalOnBean** ou **@ConditionalOnMissingBean**
- Une propriété :  
**@ConditionalOnProperty**
- La présence d'une ressource :  
**@ConditionalOnResource**
- Le fait que l'application est une application Web ou pas :  
**@ConditionalOnWebApplication** ou  
**@ConditionalOnNotWebApplication**
- Une expression SpEL



# Exemple Apache SolR

---

```
@Configuration
@ConditionalOnClass({ HttpSolrClient.class, CloudSolrClient.class })
@EnableConfigurationProperties(SolrProperties.class)
public class SolrAutoConfiguration {

    private final SolrProperties properties;

    private SolrClient solrClient;

    public SolrAutoConfiguration(SolrProperties properties) {
        this.properties = properties;
    }

    @Bean
    @ConditionalOnMissingBean
    public SolrClient solrClient() {
        this.solrClient = createSolrClient();
        return this.solrClient;
    }

    private SolrClient createSolrClient() {
        if (StringUtils.hasText(this.properties.getZkHost())) {
            return new CloudSolrClient(this.properties.getZkHost());
        }
        return new HttpSolrClient(this.properties.getHost());
    }
}
```



# Conséquences

---

Le starter *Solar* tire

- Les classes de Configuration conditionnelles
- Les librairies Sonar

Les beans d'intégration de SolR sont créés et peuvent être injectés dans le code applicatif.

```
@Component
public class MyBean {

    private SolrClient solrClient;

    public MyBean(SolrClient solrClient) {
        this.solrClient = solrClient;
    }
}
```





# Spring Boot

---

L'auto-configuration

**Starters Modules**

Propriétés de configuration

Profils, activation de profils

Configuration des traces



# Starters

---

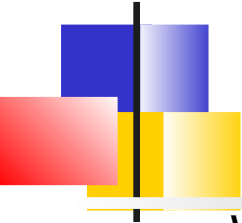
Les développeurs utilisent des starters-modules qui fournissent :

- Une ensemble de librairies dédiés à un type d'application.
- Des beans techniques qui fournissent des services d'intégration configurables

Spring fournit des starter-module pour tout type de problématique

Un assistant liste tous les starters-disponibles :

**<https://start.spring.io/>**



# Starters les + importants

---

## Web

- \*-**web** : Application web ou API REST
- \*-**reactive-web** : Application web ou API REST en mode réactif
- \*-**web-services** : Services Web SOAP

## Cœurs :

- \*-**logging** : Utilisation de logback (Starter par défaut)
- \*-**test** : Test avec Junit, Hamcrest et Mockito
- \*-**devtools** : Fonctionnalités pour le développement
- \*-**lombok** : Simplification du code Java
- \*-**configuration-processor** : Complétion des propriétés de configuration dans l'IDE



# Sécurité

---

- \*-**security** : Spring Security, sécurisation des URLs et des services métier
- \*-**oauth2-client** : Pour obtenir un jeton *oAuth* d'un serveur d'autorisation
- \*-**oauth2-resource-server** : Sécurisation des URLs via *oAuth*
- \*-**ldap** : Intégration LDAP
- \*-**okta** : Intégration avec le serveur d'autorisation Okta



# Starters de persistance

---

## Accès aux données en utilisant Spring Data

- \*-**jdbc** : JDBC avec pool de connexions Tomcat
- \*-**jpa** : Accès avec Hibernate et JPA
- \*-**<drivers>** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic)
- \*-**data-cassandra**, \*-**data-reactive-cassandra** : Base distribuée Cassandra
- \*-**data-neo4j** : Base de données orienté graphe de Neo4j
- \*-**data-couchbase** \*-**data-reactive-couchbase** : Base NoSQL CouchBase
- \*-**data-redis** \*-**data-reactive-redis** : Base NoSQL Redis
- \*-**data-geode** : Stockage de données via Geode
- \*-**data-elasticsearch** : Base documentaire indexée ElasticSearch
- \*-**data-solr** : Base indexée SolR
- \*-**data-mongodb** \*-**data-reactive-mongodb** : Base NoSQL MongoDB
- \*-**data-rest** : Utilisation de Spring Data Rest



# Messaging

---

- \*-**integration**: Spring Integration (Couche de + haut niveau)
- \*-**kafka**: Intégration avec Apache Kafka
- \*-**kafka-stream**: Intégration avec Stream Kafka
- \*-**amqp**: Spring AMQP et Rabbit MQ
- \*-**activemq** : JMS avec Apache ActiveMQ
- \*-**artemis** : JMS messaging avec Apache Artemis
- \*-**websocket** : Intégration avec STOMP et SockJS
- \*-**camel** : Intégration avec Apache Camel



# Starters UI Web

---

## Interfaces Web, Mobile REST

- \*-**thymeleaf** : Création d'applications web MVC avec des vues *Thymeleaf*
- \*-**mobile** : Spring Mobile
- \*-**hateoas** : Application RESTFul avec Spring Hateoas
- \*-**jersey** : API Restful avec JAX-RS et Jersey
- \*-**websocket**: Spring WebSocket
- \*-**mustache** : Spring MVC avec Mustache
- \*-**groovy-templates** : MVC avec gabarits Groovy
- \*-**freemarker**: MVC avec freemarker



# Autres Starters

---

## I/O

- \*-**batch** : Gestion de batchs
- \*-**mail** : Envois de mails
- \*-**cache** : Support pour un cache
- \*-**quartz** : Intégration avec Scheduler

## Ops

- \*-**actuator** : Points de surveillance REST ou JMX
- \*-**spring-boot-admin** : UI au dessus d'actuator





# Spring Cloud

---

## Services cloud

*Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba*

## Micro-services, SpringCloud

Services de discovery, de configuration externalisée, de répartition de charge, de proxy, de monitoring, de tracing, de messagerie distribuée, de circuit breaker, etc ...



# Spring Boot

---

L'auto-configuration

Starters Modules

**Propriétés de configuration**

Profils, activation de profils

Configuration des traces



# Propriétés de configuration

---

Spring Boot permet d'externaliser la configuration afin de s'adapter à différents environnements

On peut utiliser des fichiers ***properties*** ou ***YAML***, des variables d'environnement, des arguments de commande en ligne ou même des serveurs de configuration indépendant.

- Les développeurs injectent facilement les valeurs de configuration via les annotations ***@Value*** ou ***@ConfigurationProperties***



# Priorités

De nombreux niveaux de propriétés différents mais en résumé l'ordre des propriétés est :

1. *spring-boot-devtools.properties* si *devtools* est activé
2. Les propriétés de test
3. **La ligne de commande. Ex : *--server.port=9000***
4. Environnement REST, Servlet, JNDI, JVM
5. **Variables d'environnement de l'OS**
6. Propriétés avec des valeurs aléatoires
7. **Propriétés spécifiques à un profil**
8. ***application.properties* , *yaml***
9. Annotation *@PropertySource* dans la configuration
10. Les propriétés par défaut spécifiées par *SpringApplication.setDefaultProperties*



# *application.properties (.yml)*

---

Spring recherche un fichier ***application.properties (.yml)*** dans les emplacements suivants :

- Un sous-répertoire *config*
- Le répertoire courant
- Un package *config* dans le classpath
- A la racine du classpath



# Particularités

---

Les valeurs d'une propriété peuvent faire référence à une propriété déjà définie.

```
app.name=MyApp  
app.description=${app.name} is a Boot app.
```

Il est possible d'injecter des valeurs aléatoires

```
my.secret=${random.value}  
my.number=${random.int}  
my.bignumber=${random.long}  
my.uuid=${random.uuid}  
my.number.less.than.ten=${random.int(10)}  
my.number.in.range=${random.int[1024,65536]}
```



# Format YAML

---

***YAML*** (*Yet Another Markup Language*)  
est une extension de JSON, il est très  
pratique et très compact pour spécifier  
des données de configuration  
hiérarchique.

... mais également très sensible, à  
l'indentation par exemple



# Exemple *.yml*

---

```
environments:  
  dev:  
    url: http://dev.bar.com  
    name: Developer Setup  
  prod:  
    url: http://foo.bar.com  
    name: My Cool App
```

## Produit :

```
environments.dev.url=http://dev.bar.com  
environments.dev.name=Developer Setup  
environments.prod.url=http://foo.bar.com  
environments.prod.name=My Cool App
```





# Listes

---

Les listes YAML sont représentées par des propriétés avec un index.

```
my:
  servers:
    - dev.bar.com
    - foo.bar.com
```

Devient :

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```



# Spring Boot

---

L'auto-configuration  
Starters Modules  
Propriétés de configuration  
**Profils, activation de profils**  
Configuration des traces



# Introduction

---

Les **profils** fournissent un moyen de séparer des parties de la configuration et de les rendre disponible seulement dans certains environnements :  
Production, Test, Debug, etc.



# Annotations utilisant les profils

---

Tout *@Component* ou *@Configuration* peut être marqué avec **@Profile** pour limiter son chargement

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```



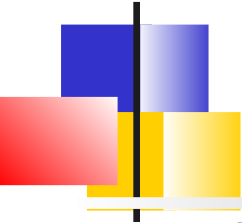
# Activation des profils

---

Les profils sont activés généralement par la propriété ***spring.profiles.active*** qui peut être positionnée :

- Dans un fichier de configuration
- En commande en ligne via :  
***--spring.profiles.active=dev,hsqldb***
- Programmatically, via :  
***SpringApplication.setAdditionalProfiles(...)***

Plusieurs profils peuvent être activés simultanément



# Propriétés spécifiques via *\*.properties*

---

Les propriétés spécifiques à un profil  
(ex : intégration, production) peuvent  
être dans des fichiers nommés :

***application-{profile}.properties***



# Propriété spécifiques à un profil via fichier *.yml*

---

Il est possible de définir plusieurs profils dans le même document.

```
server:  
  address: 192.168.1.100
```

---

```
spring:  
  profiles: development
```

```
server:  
  address: 127.0.0.1
```

---

```
spring:  
  profiles: production
```

```
server:  
  address: 192.168.1.120
```



# Inclusion de profils

---

La directive ***spring.profiles.include*** permet d'inclure des profils.

Par exemple :

```
---  
my.property: fromyamlfile  
---  
spring.profiles: prod  
spring.profiles.include:  
- proddb  
- prodm
```





# Spring Boot

---

L'auto-configuration  
Starters Modules  
Propriétés de configuration  
Profils, activation de profils  
**Configuration des traces**



# Introduction

---

Spring utilise *Common Logging* en interne mais permet de choisir son implémentation

Des configurations sont fournies pour :

- Java Util Logging
- Log4j2
- Logback (défaut)



# Format des traces

---

Une ligne contient les informations suivantes :

- Timestamp à la ms
- Niveau de trace : ERROR, WARN, INFO, DEBUG or TRACE.
- Process ID
- Un séparateur --- .
- Nom de la thread entouré de [].
- Le nom du Logger <=> Nom de la classe .
- Un message
- Une note entouré par des []



# Configurer les traces via Spring

---

Par défaut, Spring affiche les messages de niveau ERROR, WARN et INFO sur la console

- *java -jar myapp.jar -debug* : Active les messages de debug
- Propriétés ***logging.file.name*** et ***logging.file.path*** afin que les messages soient stockés dans un fichier
- Changer les niveaux de logs au niveau des logger via *application.properties*  
logging.level.root=WARN  
logging.level.org.springframework.web=DEBUG  
logging.level.org.hibernate=ERROR



# Configuration personnalisée via l'implémentation

---

Il est possible de personnaliser la configuration en utilisant les fichiers propres à chaque implémentation :

- Logback : *logback-spring.xml, logback-spring.groovy, logback.xml ou logback.groovy*
- Log4j2 : *log4j2-spring.xml ou log4j2.xml*
- JDK (Java Util Logging) : *logging.properties*



# Extensions Logback

---

Spring propose des extensions à Logback permettant des configurations avancées.

Il faut dans ce cas utiliser le fichier ***logback-spring.xml***

- Multi-configuration en fonction de profil
- Utilisation de propriétés dans la configuration



# Déploiement

---

## **Alternatives pour l'infrastructure**

Déploiement immuable et approche DevOps

Anatomie du jar

Mise en service, configuration

Spring Boot et Docker

Pipeline CI,CD



# Infrastructure

---

L'application produite nécessite une infrastructure d'exécution constituée de :

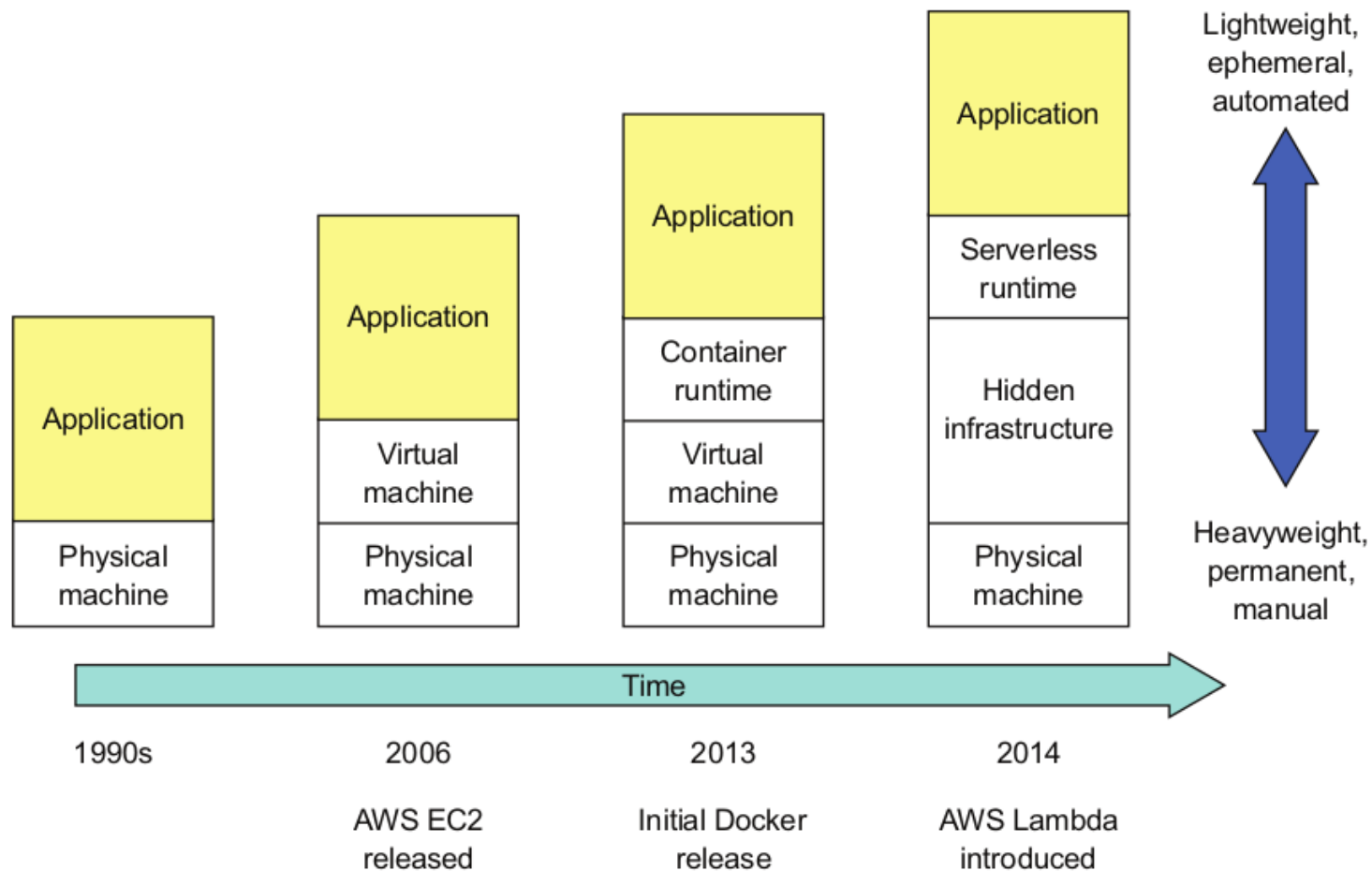
- **Matérielles** : Combien de CPU, RAM, Disque sont nécessaires pour l'application
- **Système d'exploitation** : Quelle est le système d'exploitation Cible
- **Middleware, produits, stack** : Quelles sont le middleware et les produits à installer ? Serveur applicatif, Base de données, ...

L'infrastructure est déclinée dans les différents environnements requis (intégration, QA, production)

Préparer l'infrastructure et les logiciels nécessaires s'appelle le **provisionnement**. Dans un contexte de CI/CD, il doit être également automatisé.



# Evolution des infrastructures





# Application Spring Boot

---

Même si les applications Spring Boot sont théoriquement éligibles pour toutes des infrastructures.

2 modes sont généralement pratiqués actuellement

- Machine matérielle/virtuelle

- Provisionnement de la machine : Installation du JDK au minimum
- Mise en service du jar applicatif

- Kubernetes/Cloud

- Fourniture de l'image par les développeurs
- Descripteurs de déploiements .yaml pour configuration, niveau de scalabilité, mots de passe, connectivité entre images\*

Pour l'instant, Spring Boot n'est généralement pas assez rapide pour les infrastructure *Serverless*



# Déploiement

---

Alternatives pour l'infrastructure

**Déploiement immuable et approche DevOps**

Anatomie du jar

Mise en service, configuration

Spring Boot et Docker

Pipeline CI,CD



# Modèle classique

## *Le serveur monstre mutable*

---

Un serveur Web qui contient toute l'application et mis à jour à chaque déploiement.

- Fichiers de configuration, Artefacts, schémas base de données.

=> il mute au fur et à mesure des déploiements.

=> On n'est plus sûr que les environnements de dev, de QA ou même les instances en production soient identiques

=> Difficulté de revenir à une version précédente



# Modèle DevOps

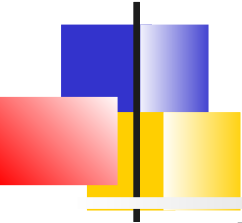
*Serveur immuable et Reverse Proxy*

---

L'approche DevOps s'appuie sur les déploiements immuables qui garantissent que chaque instance déployée est exactement identique.

Un package immuable contient tout :  
serveur d'applications, configurations  
et artefacts, i.e. Container OCI (docker)

C'est ce tout qui est déployé



# Apports du déploiement immuable

---

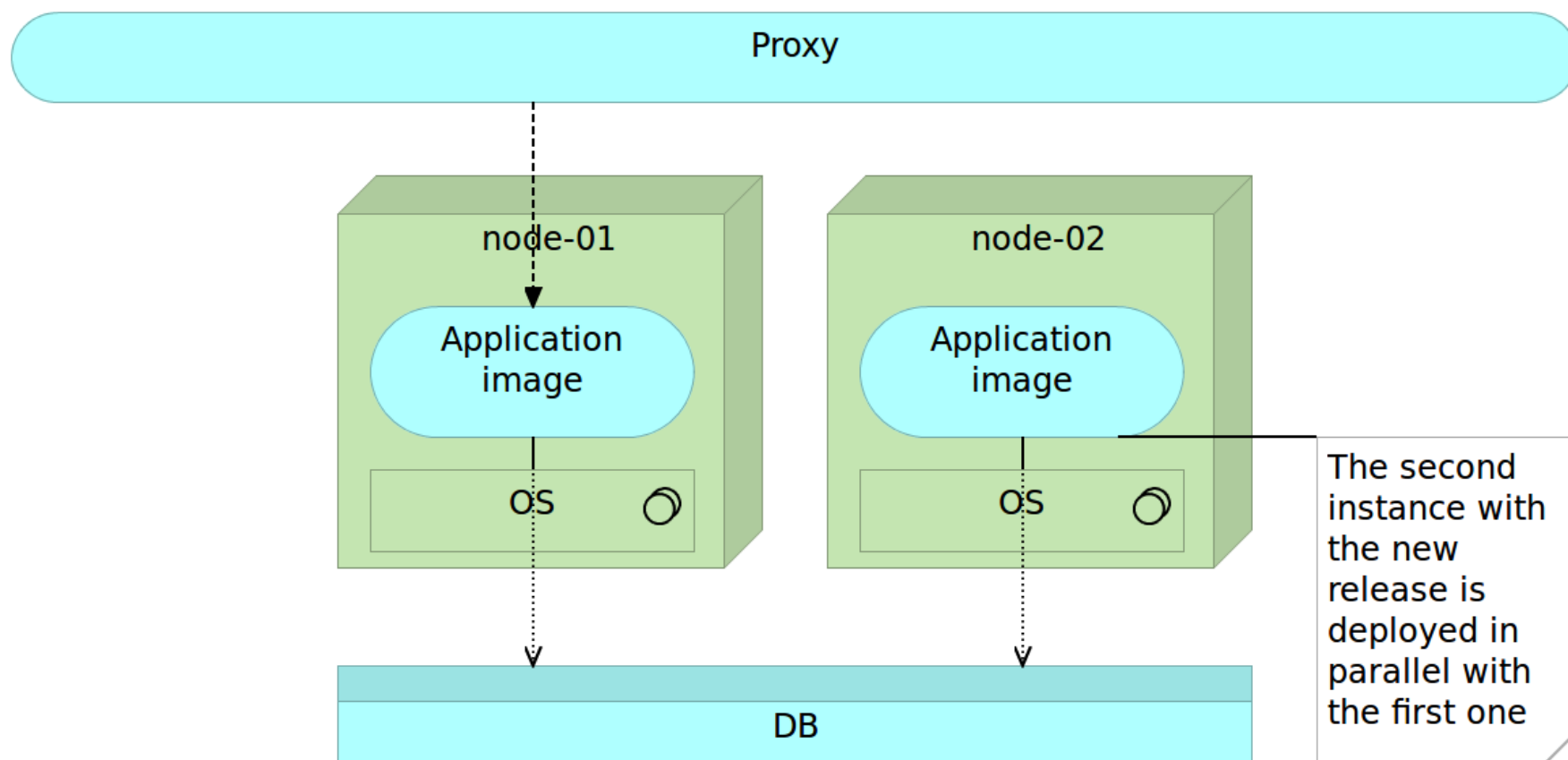
Le modèle de déploiement immuable a de nombreux avantages :

- Scalabilité avec plusieurs instances
- Déploiement de différentes versions en même temps  
=> Blue-green deployment, Canary testing, Review apps, ...
- Dev/prod parity qui fiabilise les déploiements



# Modèle DevOps

## *Serveur immuable et Reverse Proxy*





# Spring Boot et Docker

---

Spring Boot et en particulier Spring Cloud encourage l'utilisation d'images OCI pour le déploiement immuable.

- Il fournit des plugins Maven/Gradle permettant de créer des images
- Il fournit du support pour le déploiement de ces images dans des infrastructure Kubernetes
  - Maison. (Spring Cloud Kubernetes)
  - Cloud (Amazon, Google, Cloud Foundry, Heroku) en général sur des build packs





# Déploiement

---

Déploiement immuable et approche DevOps

Alternatives pour l'infrastructure

**Mise en service, configuration**

Spring Boot et Docker

Pipeline CI,CD



# Introduction

---

il est possible de créer des applications entièrement exécutables pour les systèmes Unix.

Un jar entièrement exécutable peut être exécuté comme n'importe quel autre binaire exécutable ou il peut être enregistré avec `init.d` ou `systemd`

Cela est implémenté en incorporant un script supplémentaire à l'avant du fichier.

Certains outils comme `jar -xf` n'acceptent pas ce format.

Le script par défaut prend en charge la plupart des distributions Linux et est testé sur CentOS et Ubuntu



# Création de service Linux

---

## Maven

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <executable>true</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

## Gradle

```
bootJar {
    launchScript()
}
```

=> target/artifactId.jar is executable !

=> ln -s target/artifactId.jar /etc/init.d/artifact  
service artifact start

Les logs sont présents dans : /var/log/<appname>.log



# Sécurisation du service init.d

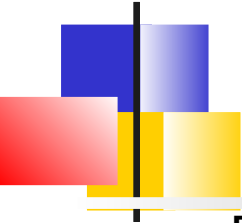
---

Le script par défaut exécute l'application en tant qu'utilisateur spécifié dans la variable d'environnement ***RUN\_AS\_USER***.

Si la variable n'est pas définie, l'utilisateur qui possède le fichier jar est utilisé à la place

=> Créer un utilisateur spécifique avec un shell à */usr/sbin/nologin* pour exécuter l'application et positionner la variable d'environnement *RUN\_AS\_USER* ou utiliser *chown*

=> Rendre également le jar immuable  
*sudo chattr +i your-app.jar*



# Installation comme service *systemd*

---

En supposant que vous ayez une application Spring Boot installée dans `/var/myapp`, créez un script nommé `myapp.service` et placez-le dans le répertoire `/etc/systemd/system` puis :

```
systemctl enable myapp.service
```

Exemple :

```
[Unit]
Description=myapp
After=syslog.target
```

```
[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143
```

```
[Install]
WantedBy=multi-user.target
```



# Personnalisation du script de démarrage

---

Le script peut être personnalisé

- au moment de la génération par des propriétés de build
- ou à l'exécution via des variables d'environnement ou un fichier de configuration. (Par défaut, un fichier avec le même nom que le jar dans le même emplacement)



# Variables ou propriétés de personnalisation

---

***RUN\_AS\_USER*** : L'utilisateur qui sera utilisé pour exécuter l'application.

***USE\_START\_STOP\_DAEMON*** : Si la commande start-stop-daemon, lorsqu'elle est disponible, doit être utilisée pour contrôler le processus. La valeur par défaut est true.

***PID\_FOLDER*** : Le nom racine du dossier pid (/ var/run par défaut).

***LOG\_FOLDER*** : Le nom du dossier dans lequel placer les fichiers journaux (/var/log par défaut).

***CONF\_FOLDER*** : Le nom du dossier à partir duquel lire les fichiers .conf (même dossier que jar-file par défaut).

***LOG\_FILENAME*** : Le nom du fichier journal dans LOG\_FOLDER (<appname> .log par défaut).

***APP\_NAME*** : Le nom de l'application.

***RUN\_ARGS*** : Les arguments à transmettre au programme (l'application Spring Boot).

***JAVA\_HOME*** : L'emplacement de l'exécutable java est découvert en utilisant le PATH par défaut, mais vous pouvez le définir explicitement s'il existe un fichier exécutable dans \$ JAVA\_HOME/bin/java.

***JAVA\_OPTS*** : Options transmises à la JVM lors de son lancement.

***JARFILE*** : L'emplacement explicite du fichier jar, au cas où le script n'incorpore pas le jar.

***DEBUG*** : l'indicateur -x sur le processus shell, vous permettant de voir la logique dans le script.

***STOP\_WAIT\_TIME*** : Temps d'attente en secondes lors de l'arrêt de l'application avant de forcer un arrêt (60 par défaut).



# Service Windows

---

Une application Spring Boot peut être démarrée en tant que service Windows à l'aide de ***winsw***.

Voir :

<https://github.com/snicoll-scratches/spring-boot-daemon>





# Déploiement

---

Déploiement immuable et approche DevOps  
Alternatives pour l'infrastructure  
Mise en service, configuration  
**Spring Boot et Docker**  
Pipeline CI,CD



# Dockerfile basique

---

Il est facile de construire un jar exécutable d'une application Spring Boot via les plugins Maven/Gradle

Un Dockerfile basique est alors :

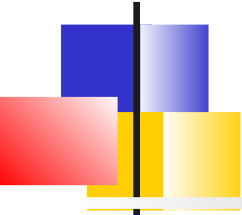
```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Pour le construire :

```
docker build --build-arg JAR_FILE=target/*.jar -t myorg/myapp .
```

Pour l'exécuter :

```
docker run -p 8080:8080 myorg/myapp
```



# Dockerfile + sophistiqué

---

Pour pouvoir passer des variables  
d'environnement et des propriétés  
SpringBoot au démarrage

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -jar /app.jar ${0} $@"]
```

On peut alors démarrer l'image via :

```
docker run -p 9000:9000 -e "JAVA_OPTS=-Ddebug -Xmx128m"
myorg/myapp --server.port=9000
```



# Taille des images

---

Les images alpine sont plus petites que les images standard openjdk de Dockerhub.

20 mB sont économisés en utilisant une jre plutôt qu'une jdk

On peut également essayer d'utiliser jlink (à partir de Java11) pour se créer une image sur mesure en ne sélectionnant que les modules Java utilisés

- Attention pas de cache, si tous les services Java utilisent des JRE personnalisés



# Couches Docker

---

Un jar Spring Boot a naturellement des "couches" Docker en raison de son packaging.

On peut isoler dans une couche Docker les dépendances externes ainsi tous les services utilisant les mêmes dépendances pourront se construire et se lancer plus vite.  
(Cache des container runtime)



# Support Maven/Gradle

---

Les plugins pour Maven et Gradle permettent de générer des jars en couches, séparant le code applicatif du framework.

Gradle :

```
bootJar {  
    layered()  
}
```

Maven :

```
<plugin>  
<artifactId>spring-boot-maven-plugin</artifactId>  
<configuration>  
<layers>  
<enabled>true</enabled>  
</layers>  
</configuration>  
</plugin>
```



# Dockerfile à couche

---

```
$ mkdir target/dependency
$ (cd target/dependency; jar -xf ../*.jar)
$ docker build -t myorg/myapp .
```

—

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "hello.Application"]
```



# Accélérateur de démarrage

---

Quelques astuces pour accélérer le démarrage du service :

- Améliorer le scan du classpath avec *spring-context-indexer*<sup>1</sup>
- Limiter *actuator* à ce qui est vraiment nécessaire
- SB 2.1+ et S 5.1+
- Spécifier le chemin vers le fichier de config :  
*spring.config.location*
- Désactiver JMX `spring.jmx.enabled=false`
- Exécuter la JVM avec `-noverify`
- Pour Java8, également `-XX:+UnlockExperimentalVMOptions`  
`-XX:+UseCGroupMemoryLimitForHeap`

1. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-scanning-index>





# Utilisateur dédié

---

Les services doivent s'exécuter avec un utilisateur non root

```
FROM openjdk:8-jdk-alpine
```

```
RUN addgroup -S demo && adduser -S demo -G demo  
USER demo
```



# Spring-Boot Plugin

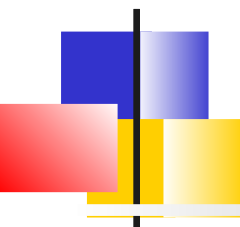
---

Depuis la 2.3, Spring boot fournit des plugins Maven/Gradle capables de construire les images

```
$ ./mvnw spring-boot:build-image
```

```
$ ./gradlew bootBuildImage
```

- Plus besoin de Dockerfile
- Nécessite que le démon Docker s'exécute
- Par défaut le nom de l'image est :  
    `docker.io/<group>/<artifact>:latest`
- Utilise les *Cloud Native Buildpacks*



# Autres plugins

---

***Spotify Maven Plugin*** nécessite un Dockerfile et ajoute des objectifs permettant d'automatiser la construction de l'image dans le build Maven

***Palantir Gradle*** est capable de générer un Dockerfile ou d'utiliser celui que l'on fournit

***Jib*** est également un projet Google qui permet de construire des images optimisées (Docker ou OCI) sans Docker installé



# BuildPacks

---

Cloud Foundry utilise les ***buildbacks*** qui transforment automatiquement le code source en conteneur<sup>1</sup>

- Les développeurs n'ont pas besoin de se soucier des détails sur la construction du conteneur.
- Les buildpacks ont de nombreuses fonctionnalités pour la mise en cache des résultats de construction et des dépendances  
=> Souvent un buildpack s'exécute beaucoup plus rapidement qu'un docker natif



# Sortie standard d'un buildpack

---

```
$ pack build myorg/myapp --builder=cloudfoundry/cnb:bionic --path=.
2018/11/07 09:54:48 Pulling builder image 'cloudfoundry/cnb:bionic' (use --no-pull flag to skip this step)
2018/11/07 09:54:49 Selected run image 'packs/run' from stack 'io.buildpacks.stacks.bionic'
2018/11/07 09:54:49 Pulling run image 'packs/run' (use --no-pull flag to skip this step)
*** DETECTING:
2018/11/07 09:54:52 Group: Cloud Foundry OpenJDK Buildpack: pass | Cloud Foundry Build System Buildpack: pass | Cloud Foundry JVM Application Buildpack: pass
*** ANALYZING: Reading information from previous image for possible re-use
*** BUILDING:
-----> Cloud Foundry OpenJDK Buildpack 1.0.0-BUILD-SNAPSHOT
-----> OpenJDK JDK 1.8.192: Reusing cached dependency
-----> OpenJDK JRE 1.8.192: Reusing cached launch layer

-----> Cloud Foundry Build System Buildpack 1.0.0-BUILD-SNAPSHOT
-----> Using Maven wrapper
      Linking Maven Cache to /home/pack/.m2
-----> Building application
      Running /workspace/app/mvnw -Dmaven.test.skip=true package
...
---> Running in e6c4a94240c2
---> 4f3a96a4f38c
---> 4f3a96a4f38c
Successfully built 4f3a96a4f38c
Successfully tagged myorg/myapp:latest
.
```



# *KNative*

---

***KNative*** est un projet initié par Google qui a pour but de fournir une infrastructure *ServerLess* au dessus d'un cluster Kubernetes

Comme les *buildpacks*, il est capable de construire les conteneurs à partir du code source.

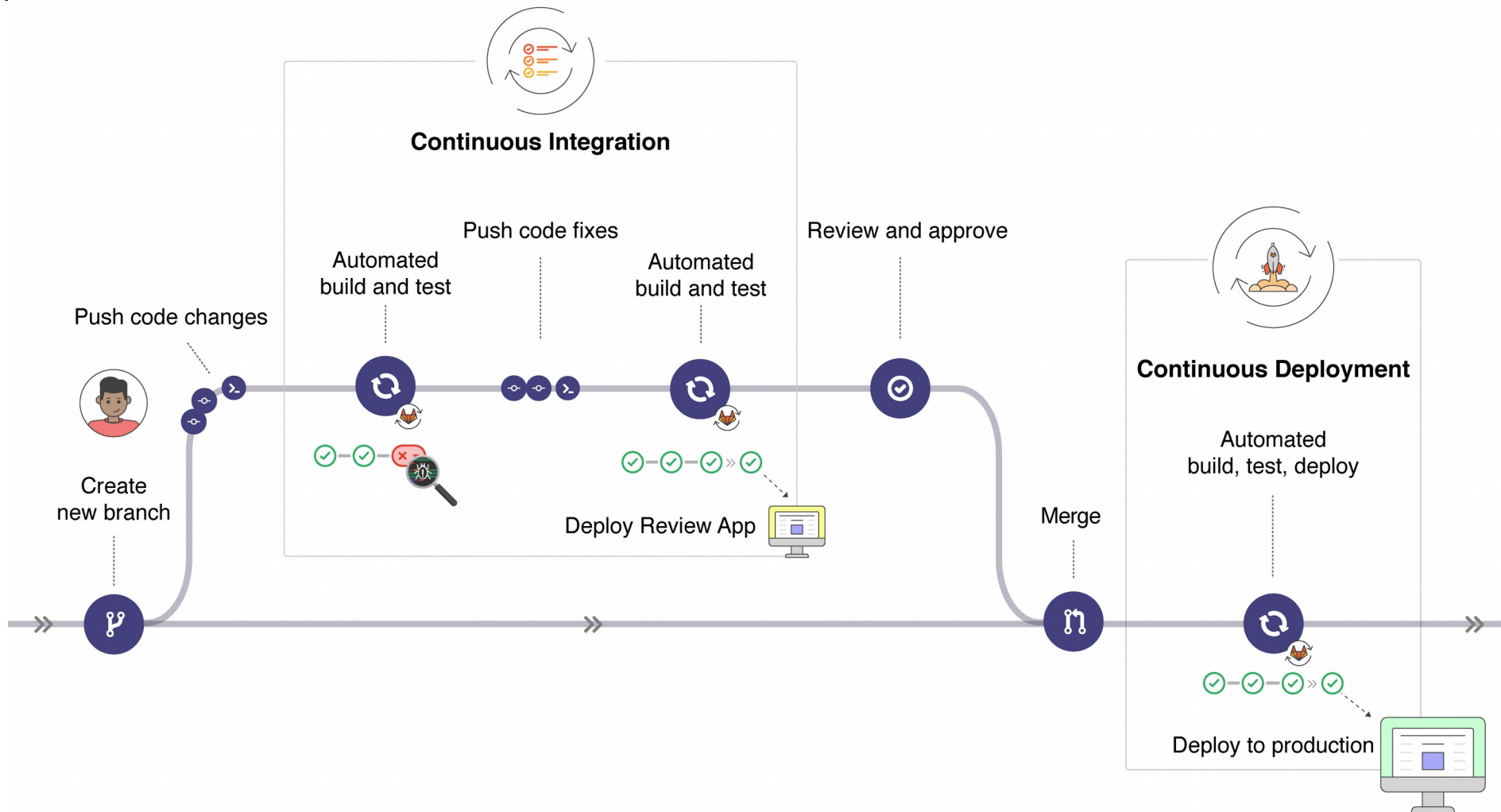


# Déploiement

---

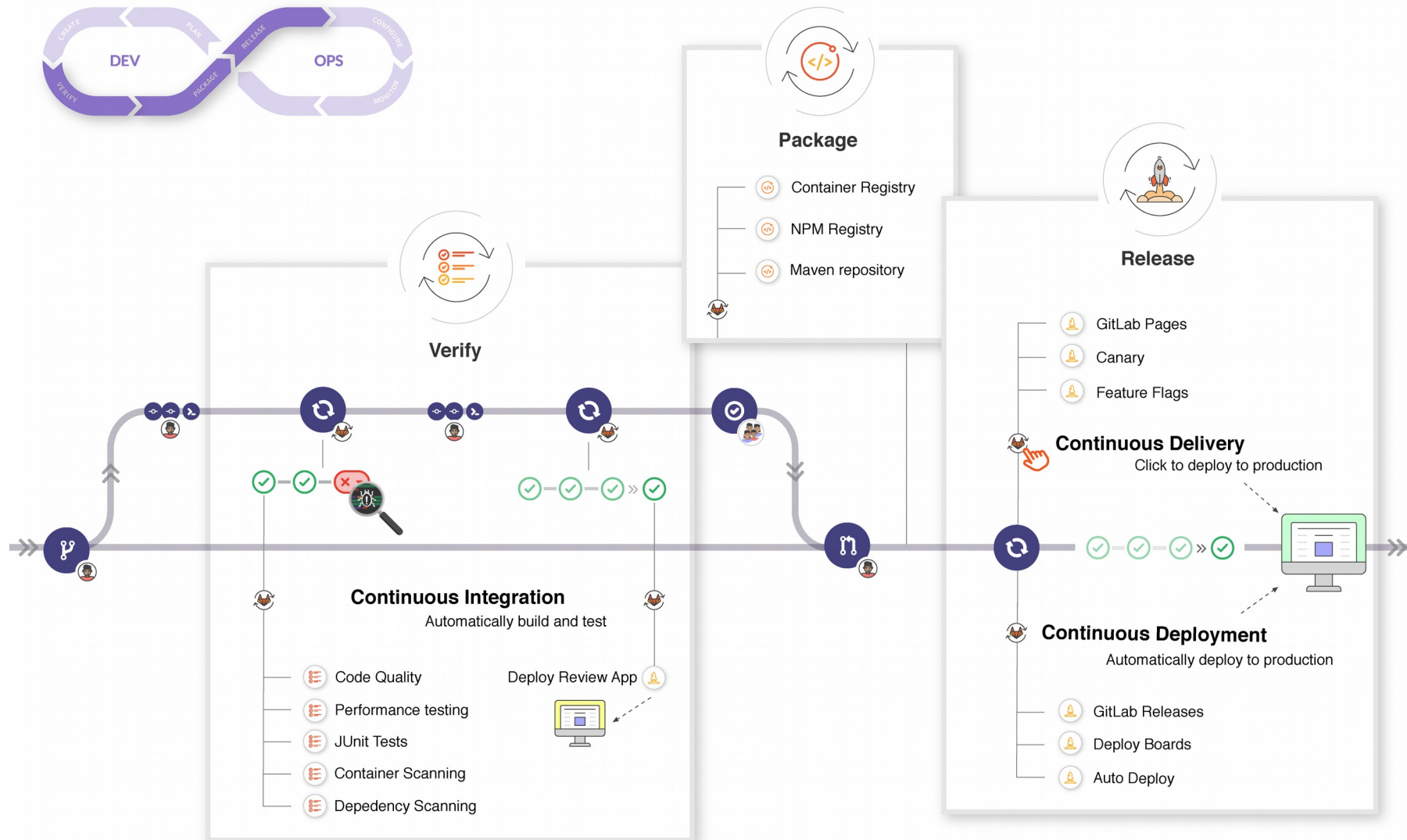
Déploiement immuable et approche DevOps  
Alternatives pour l'infrastructure  
Anatomie du jar  
Mise en service, configuration  
Spring Boot et Docker  
**Pipeline CI,CD**

# Exemple Gitlab CI





# En plus détaillé





# Exemple *.gitlab-ci.yml* (1)

---

stages:

- Build
- Test
- Publish-Artefact
- Deploy

build:

stage: Build

script: **'./mvnw \$MAVEN\_OPTS package'**

artifacts:

paths:- application/target/\*.jar

integration-test:

stage: Test

except:

refs:

- master

script: **'./mvnw integration-test'**

analysis:

stage: Test

except:

refs:

- master

script: **'./mvnw verify sonar:sonar'**



# Exemple *.gitlab-ci.yml* (2)

---

build-docker:

stage: Publish-Artefact

dependencies:

- build

script:

- echo "\$DOCKER\_LOGIN \$DOCKER\_PASSWORD"
- cp target/\*.jar .
- cp src/main/docker/Dockerfile .
- docker build -t dthibau/gitlab-multi-module .
- docker login -u \$DOCKER\_LOGIN -p \$DOCKER\_PASSWORD
- docker push dthibau/gitlab-multi-module



# Exemple *.gitlab-ci.yml*

---

## **deploy\_review:**

```
stage: Deploy
script:
  - sed -i "s/#NS#/$KUBE_NAMESPACE/g" multi-module-deployment.yml
  - kubectl apply -f multi-module-deployment.yml
environment:
  name: review/${CI_COMMIT_REF_NAME}
except:master
```

## **deploy\_staging:**

```
stage: Deploy
script:
  - sed -i "s/#NS#/$KUBE_NAMESPACE/g" multi-module-deployment.yml
  - kubectl apply -f multi-module-deployment.yml
environment:
  name: staging
only: master
```

## **deploy\_prod:**

```
stage: Deploy
script:
  - kubectl apply -f multi-module-deployment.yml
environment:
  name: production
when: manual
only: master
```



# Exploitation

---

## **Mise en production**

Monitoring des applications et actuator  
Spécificités Kubernetes



# Introduction

---

Lors d'une mise en production, certaines configuration sont recommandées :

- Forcer l'usage de https
- Configurer les traces
- Supprimer les mots de passe de la configuration
- Autoriser la surveillance de l'application
- ...



# Mise en place SSL

---

SSL peut être configuré via les propriétés préfixées par ***server.ssl.\****

Par exemple :

```
server.port=8443
```

```
server.ssl.key-store=classpath:keystore.jks
```

```
server.ssl.key-store-password=secret
```

```
server.ssl.key-password=another-secret
```

Par défaut si SSL est configuré, le port 8080 disparaît.

Si l'on désire les 2, il faut configurer explicitement le connecteur réseau



# https

---

```
public class SslWebSecurityConfigurerAdapter extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // Forcer l'usage de https
        http.requiresChannel().anyRequest().requiresSecure();
    }
}
```

## Si derrière proxy

```
server.tomcat.remoteip.remote-ip-header=x-forwarded-for
server.tomcat.remoteip.protocol-header=x-forwarded-proto
```





# Configuration des traces

---

Une solution courante pour la gestion des traces en production est de pouvoir les agréger en un point central.

Le format JSON est alors souvent utilisé (outils comme *logstash*, *elastic search*, grafana)

SpringBoot, logback permettent de facilement mettre en place des traces au format JSON adapté à logstash



# Exemple

---

```
<!-- Dépendance logstash encoder -->
<dependency>
  <groupId>net.logstash.logback</groupId>
  <artifactId>logstash-logback-encoder</artifactId>
</dependency>

<!-- Configuration logback.xml -->
<configuration>
  <appender name="consoleAppender"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
  </appender>
  <logger name="jsonLogger" additivity="false" level="DEBUG">
    <appender-ref ref="consoleAppender"/>
  </logger>
  <root level="INFO">
    <appender-ref ref="consoleAppender"/>
  </root>
</configuration>
```



# Mots de passe en clair

---

Pour empêcher les mots de passe en clair dans les fichiers de configuration, plusieurs approches sont possibles :

- Encrypter les mots de passe et utiliser une secret key au démarrage
- Stocker les mots de passe sur un serveur externe (Vault) et donner les moyens d'y accéder



# Crypter les propriétés

---

Pour encrypter des mots de passe, on peut utiliser *Jasypt* qui nécessite un algorithme ... et un mot de passe

Le mot de passe peut être positionné en temps que variable d'environnement dans le script de démarrage

```
export JASYPT_ENCRYPTOR_PASSWORD=supersecretz  
Java -jar application.jar
```



# Configuration externe

---

Soit l'infrastructure (ex. Kubernetes) soit un service annexe tel que Hashicorp Vault permet d'externaliser les secrets.

Le projet Spring Cloud Vault permet de s'intégrer à Hashicorp Vault



# Hashicorp Vault

---

Un serveur Vault fourni des points d'accès permettant de récupérer des valeurs de configuration (clé/valeur). Typiquement :

```
/secret/{application}/{profile}  
/secret/{application}  
/secret/{default-context}/{profile}  
/secret/{default-context}
```

Il supporte différentes bases de données afin de pouvoir créer dynamiquement des crédeniels d'accès.



# Spring Cloud Vault

---

```
<!-- Dépendances coeur + librairies spécifiques pour BD -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-vault-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-vault-config-databases</artifactId>
</dependency>
-----
# Configuration pour l'accès au serveur et l'autorisation des 2 fonctionnalités de Vault
spring:
  cloud:
    vault:
      uri: https://localhost:8200
      ssl:
        trust-store: classpath:/vault.jks
        trust-store-password: changeit
      generic:
        enabled: true
      database:
        enabled: true
        role: fakebank-accounts-rw
```



# Désactiver JMX

---

Si JMX n'est pas utilisé, on peut le désactiver

- => gain en performance au boot
- => Sécurisation

Il suffit de positionner la propriété

`spring.datasource.jmx-enabled=false`





# Exploitation

---

Mise en production  
**Monitoring des applications et  
actuator**  
Spécificités Kubernetes



# Actuator

---

*Spring Boot Actuator* fournit un support pour la surveillance et la gestion des applications SpringBoot

Il peut s'appuyer

- Sur des points de terminaison HTTP (Si on a utilisé Spring MVC)
- Sur JMX

L'activation de Actuator nécessite  
***spring-boot-starter-actuator***



# Mise en production

---

Les fonctionnalités transverses offertes par *Actuator* concernent :

- Statut de santé de l'application
- Obtention de métriques
- Audit de sécurité
- Traces des requêtes HTTP
- Visualisation de la configuration
- ...

Elles sont accessibles via JMX ou REST



# Endpoints

---

Actuator fournit de nombreux endpoints :

- **auditevents** : Informations d'audit : login succès ou échoués
- **beans** : Une liste des beans Spring
- **caches** : Les caches disponibles
- **conditions** : Les conditions remplies ou non pour l'auto-configuration
- **env / configprops** : Liste des propriétés configurables
- **flyway,liquibase** : Les migrations BD par flyway, liquibase appliqués
- **health** : Etat de santé de l'appli
- **httptrace** : Les traces HTTP
- **info** : Informations arbitraires. En général, Commit, version
- **loggers** : Les niveaux de traces des différents loggers
- **metrics** : Mesures
- **mappings** : Liste des mappings configurés
- **scheduledtasks** : Les tâches planifiés
- **sessions** : Les sessions en cours.
- **threaddump** : Dump des threads
- ---



# Configuration

---

Les *endpoints* peuvent être configurés par des propriétés.

Chaque endpoint peut être

- Activé/désactivé
- Exposé via JMX ou Web
- Sécurisé par Spring Security
- Mappé sur une autre URL

Par défaut, seuls les endpoints */health* et */info* sont exposés en http, et tous les endpoints sont exposés via JMX

Pour activer les autres :

- *management.endpoints.web.exposure.include=\**
- Ou les lister un par un



# Exposition des endpoints

---

4 propriétés permettent de contrôler l'exposition des endpoints, on peut les lister ou utiliser le caractère \* :

management

    endpoints

        jmx

            exposure

                exclude :

                include :

        web

            exposure

                exclude :

                include :



# Sécurisation

---

La sécurisation des ressources actuator s'effectue de la même façon que les ressources applicatives via Spring Security

```
@Configuration(proxyBeanMethods = false)
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->
            requests.anyRequest().hasRole("ENDPOINT_ADMIN"));
        http.httpBasic();
    }
}
```



# CORS

---

Si les requêtes doivent être faites d'une page web issus d'un autre serveur (produit de monitoring par exemple), il est nécessaire de mettre en place le CORS.

```
management.endpoints.web.cors.allowed-origins=https://example.com  
management.endpoints.web.cors.allowed-methods=GET,POST
```





# Endpoint */health*

---

L'information fournie permet de déterminer le statut d'une application en production.

- Elle peut être utilisée par des outils de surveillance responsable d'alerter lorsque le système tombe (Kubernetes par exemple)

Par défaut, le endpoint affiche un statut global mais on peut configurer Spring pour que chaque sous-système (beans de type *HealthIndicator*) affiche son statut :

```
management.endpoint.health.show-details= always
```



# Indicateurs fournis

---

Spring fournit les indicateurs de santé suivants lorsqu'ils sont appropriés :

- ***CassandraHealthIndicator*** : Base Cassandra est up.
- ***DiskSpaceHealthIndicator*** : Vérifie l'espace disque disponible .
- ***DataSourceHealthIndicator*** : Connexion à une source de données
- ***ElasticsearchHealthIndicator*** : Cluster Elasticsearch up.
- ***JmsHealthIndicator*** : JMS broker up.
- ***MailHealthIndicator*** : Serveur de mail up.
- ***MongoHealthIndicator*** : BD Mongo up.
- ***RabbitHealthIndicator*** : Serveur Rabbit up
- ***RedisHealthIndicator*** : Serveur Redis up.
- ***SolrHealthIndicator*** : Serveur Solr up
- ...



# Information sur l'application

---

Le *endpoint* **/info** par défaut n'affiche rien.

On peut personnaliser par des propriétés info.\*

```
info.app.encoding=UTF-8
info.app.java.source=1.8
info.app.java.target=1.8
```

Si l'on veut les détails sur Git :

```
<dependency>
  <groupId>pl.project13.maven</groupId>
  <artifactId>git-commit-id-plugin</artifactId>
</dependency>
```

Si l'on veut les informations de build :

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>build-info</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



# Metriques

---

Le endpoint ***metrics*** donne accès à toute sorte de métriques. On retrouve :

- Système : Mémoire, Heap, Threads, GC
- Source de données : Connexions actives, état du pool
- Cache : Taille, Hit et Miss Ratios
- Tomcat Sessions



# Loggers

---

Spring Boot Actuator inclut la possibilité d'afficher et de configurer les niveaux de trace de l'application.

Pour visualiser un GET, pour mettre à jour, un POST vers *actuator/loggers/<logger>*

```
{  
    "configuredLevel": "DEBUG"  
}
```



# Tracing HTTP

---

Le traçage de requête HTTP peut être activé en fournissant un bean de type *HttpTraceRepository*.

Par défaut, Spring fournit ***InMemoryHttpTraceRepository*** qui stocke les traces des 100 derniers échanges.

Cette solution est limitée et il est recommandé d'utiliser des solutions comme Zipkin ou Spring Cloud Sleuth en production.



# Audit

---

Par défaut, Spring Boot Actuator publie des événements d'audit (par défaut, «authentification réussie», «échec» et «accès refusé»).

Ces événements sont stockés en mémoire mais on peut diriger ces événements vers un support persistant (Bean AuditEventRepository)

Cela peut être très utile pour la création de rapports et pour la mise en œuvre d'une politique de verrouillage basée sur les échecs d'authentification



# Exploitation

---

Mise en production  
Monitoring des applications et actuator  
**Spécificités Kubernetes**





# Sondes Kubernetes

---

Les applications déployées sur Kubernetes peuvent fournir des informations sur leur état interne avec les *Container Probes*

- ***livenessProbe***: Indique si le container s'exécute
- ***readinessProbe***: Indique si le container est prêt à répondre à des requêtes
- ***startupProbe***: Indique si l'application à l'intérieur du conteneur est démarré.

Actuator est capable d'exposer les informations "Liveness" et "Readiness" en http si il détecte un environnement Kubernetes ou via la propriété `management.endpoint.health.probes.enabled`



# Descripteur Kubernetes

---

livenessProbe:

httpGet:

path: /actuator/health/liveness

port: <actuator-port>

failureThreshold: ...

periodSeconds: ...

readinessProbe:

httpGet:

path: /actuator/health/readiness

port: <actuator-port>

failureThreshold: ...

periodSeconds: ..



# ConfigMap

---

Les ressources **ConfigMap** de Kubernetes peuvent être utilisées :

- Pour définir des paires clés valeurs de configuration
- Pour embarquer des fichiers complets de configuration (*application.properties* ou *application.yml*)

Les valeurs de configuration sont lues au bootstrap et peuvent être rechargées en cours d'exécution lors de la détection de changement



# Mécanisme

---

Starter : `spring-cloud-starter-kubernetes-config`

L'objet *ConfigMapPropertySource* est chargé à partir d'une ressource Kubernetes ConfigMap dont le ***metadata.name*** est identique

- à la propriété de bootstrap  
***spring.application.name***
- Ou à la propriété spécifique :  
***spring.cloud.kubernetes.config.name***

Des configurations plus avancées sont possibles comme l'utilisation de plusieurs *ConfigMap*



# Profils

---

La gestion des différents profils peut être effectuée :

- A l'intérieur du contenu *.yml* présent dans la *ConfigMap*
- En définissant plusieurs *ConfigMap* par profils.

Exemple :

```
monService-dev  
monService-prod
```



# Activation d'un profil

---

Pour activer un profil sur un pod, le plus simple est de positionner la variable d'environnement **SPRING\_PROFILES\_ACTIVE**

```
apiVersion: apps/v1
kind: Deployment
```

```
...
```

```
  spec:
    containers:
      - name: container-name
        image: your-image
        env:
          - name: SPRING_PROFILES_ACTIVE
            value: "development"
```



# Introduction

---

Kubernetes propose les ressources ***Secrets*** pour stocker les informations sensibles (mot de passe, etc.)

Les secrets peuvent être accédés par les pods *SpringCloud*

Cette fonctionnalité est activée avec **`spring.cloud.kubernetes.secrets.enabled`**



# Mécanisme

---

Si la fonctionnalité est activée, l'interface ***SecretsPropertySource*** recherche des secret *Kubernetes* à partir de :

- Lecture récursive des volumes secrets montés
- Secret nommé comme l'application (*spring.application.name*)
- Des secrets correspondant à des labels particuliers





# Annexes



# Création de war

---

Pour créer un war, il est nécessaire de :

- Fournir une sous-classe de **SpringBootServletInitializer** et surcharger la méthode *configure()*. Cela permet de configurer l'application (Spring Beans) lorsque le war est installé par le servlet container.
- De changer l'élément packaging du *pom.xml* en war  
<packaging>war</packaging>
- Puis exclure les librairies de tomcat  
Par exemple en précisant que la dépendance sur le starter Tomcat est fournie

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-tomcat</artifactId>  
  <scope>provided</scope>  
</dependency>
```



# Exemple

---

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```



# Cloud

---

Les jars exécutable de Spring Boot sont prêts à être déployés sur la plupart des plate-formes PaaS

La documentation de référence offre du support pour :

- Cloud Foundry
- Heroku
- OpenShift
- Amazon Web Services
- Google App Engine



# Exemple CloudFoundry/Heroku

---

## Cloud Foundry

```
cf login
```

```
cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

## Heroku

Mise à jour d'un fichier Procfile :

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

```
git push heroku master
```