Spring Boot

Olivier Capuozzo, Frédéric Varni

Version 1.2, 2021-01-12

Table des matières

Introduction

Spring

Spring rend la programmation Java plus rapide, plus facile et plus sûre pour tout le monde. L'orientation de Spring sur la vitesse, la simplicité et la productivité en a fait l'un des plus populaires framework Java.

On retrouve Spring dans tous les contextes : depuis les sites de commerce en ligne jusqu'aux voitures connectées, en passant par les applications de streaming TV et bien d'autres applications.

Spring est un ensemble d'extensions et de bibliothèques tierces intégrées qui permet de bâtir à peu près n'importe quel type d'applicaton. Au coeur de Spring on retrouve l'Inversion de Contôle (Inversion of Control, *IoC*) et l'Injection de dépendances (Dependency Injection, *DI*), ces deux caractéristiques sont à la base d'un grand nombre d'autres caractéristiques et fonctionnalités.

Une application Spring *classique* est relativement difficile à mettre en oeuvre et demande une expertise certaine de la part du chef de projet pour en construire l'ossature et définir les fichiers de configuration. Spring Boot est une verion de Spring plus facile à mettre en oeuvre tout en offrant les mêmes avantages que Spring. Spring Boot évite notamment les fichiers de configuration XML complexes.

Spring Boot

Spring Boot est une version de Spring qui privilégie les conventions plutôt que les configurations (*convention over configuration*), à savoir que si le développeur respecte un certain nombre de normes comme le nommage, le placement des fichiers de code, les annotations, ... il peut s'affranchir d'un grand nombre de fichiers de configuration. Spring fournit même un site dédié qui permet de générer un squelette d'application, https://start.spring.io , connu également sous le nom de Spring Initializr.

Spring Boot c'est :

- Un outil de construction rapide de projets Spring
  - L'intégration des nombreux sous-projets Spring
  - La possibilité de choisir entre plusieurs technologies pour la persistance, la logique applicative ou la gestion des vues. Plusieurs technologies peuvent cohabiter
  - L'assurance de toujours avoir les bonnes dépendances
  - Etre prêt à commencer à programmer en quelques minutes
- Une vision convention plutôt que configuration
  - L'application est prévue pour fonctionner avec un ensemble cohérents de paramètres par défaut
  - La configuration et la dépendance entre beans se fait principalement par annotations
  - Utilisation de fichiers properties simples pour surcharger les paramètres par défaut

- On peut quand même utiliser des fichiers de configuration XML en faisant un effort

Spring Boot n'est pas :

- Une baguette magique
- Un outil de scaffolding
- La solution pour ne pas apprendre les mécanismes internes de Spring
- Un outil pour les allergiques à *Maven* et autre *Gradle*

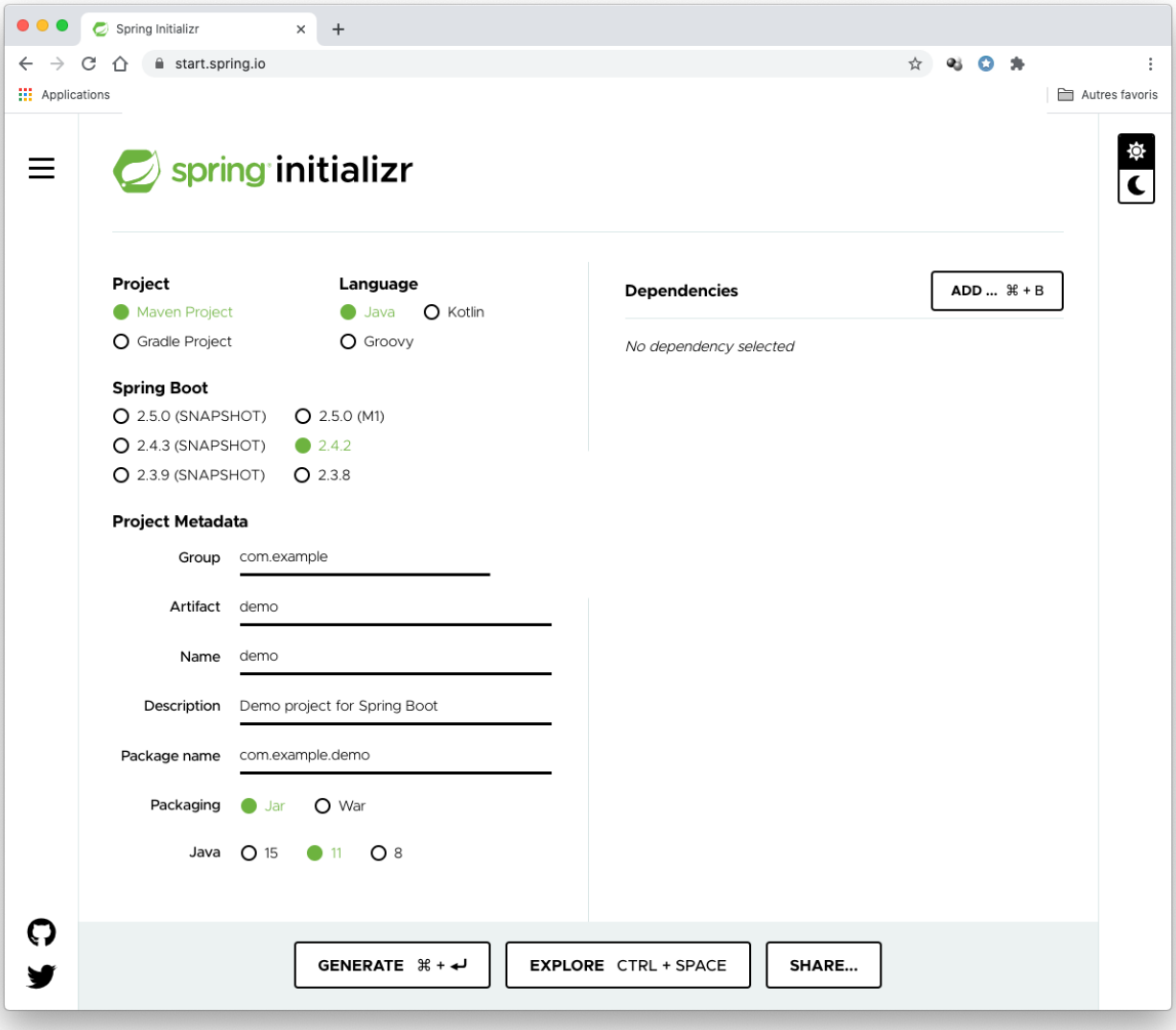
Les dépendances d'une application Spring Boot

Spring Initializr permet de choisir les dépendances/frameworks à utiliser pour la conception d'une application certaines dépendances sont presque indispensables à la création d'une application Web, d'autres sont optionnelles et peuvent être choisies plus tard.

- Les dépendances indispensables :
  - `Spring Boot DevTools` : permet des redémarrages rapides des applications, le rechargement temps réel des configurations pour une expérience de programmation plus fluide. Mais consomme un peu plus de ressources.
  - `Spring Web` : pour construire des applications Web MVC, y compris RESTFUL. Utilise Apache Tomcat comme conteneur de servlets par défaut.
  - `Apache Freemarker` ou `Thymeleaf` : un des deux moteurs de templates pour gérer les pages HTML, pas les deux en même temps autant que possible.
- Les dépendances optionnelles :
  - `Spring Data JPA` : pour gérer la persistance des données dans une base SQL
  - `Apache Derby Database` ou `H2 Database` ou `PostgreSQL Driver` ou `MySQL Driver` : pour gérer les accès à une base de données SQL. Nécessaire pour `Spring Data JPA`.
  - `Spring Security` : framework d'authentification et d'autorisation pour créer des applications sécurisées. Attention dès que ce composant est installé l'application demande une authentification.
  - `Lombok` : un framework qui permet de s'affranchir, entre autres, des getters et des setters. Attention pour l'utiliser il vaut mieux avoir installé le plugins IntelliJ correspondant.

Générer une application Spring Boot

Le moyen le plus simple pour générer une application Spring Boot est de passer par le site https://start.spring.io qui permet positionner un certain nombre de paramètres et de choisir les dépendances à utiliser.



Les choix possibles :

- Les paramètres de l'application :
  - l'outil utilisé pour gérer le projet : `Maven` ou `Gradle`
  - le langage de programmation utilisés : `Java`, `Groovy` ou `Kotlin`
  - La version de Spring Boot
  - Les métadonnées de l'application :
    - Nom du groupe : nom dns de l'organisation
    - Nom de l'artefact : nom du projet
    - Nom d'affichage du projet
    - Description du projet
    - Le type de packaging :
      - `jar` : application indépendante avec serveur tomcat intégré
      - `war` : application prête à être déployée dans un conteneur de servlets

La version de `Java` : 8 ou 11 recommandées, pas 15

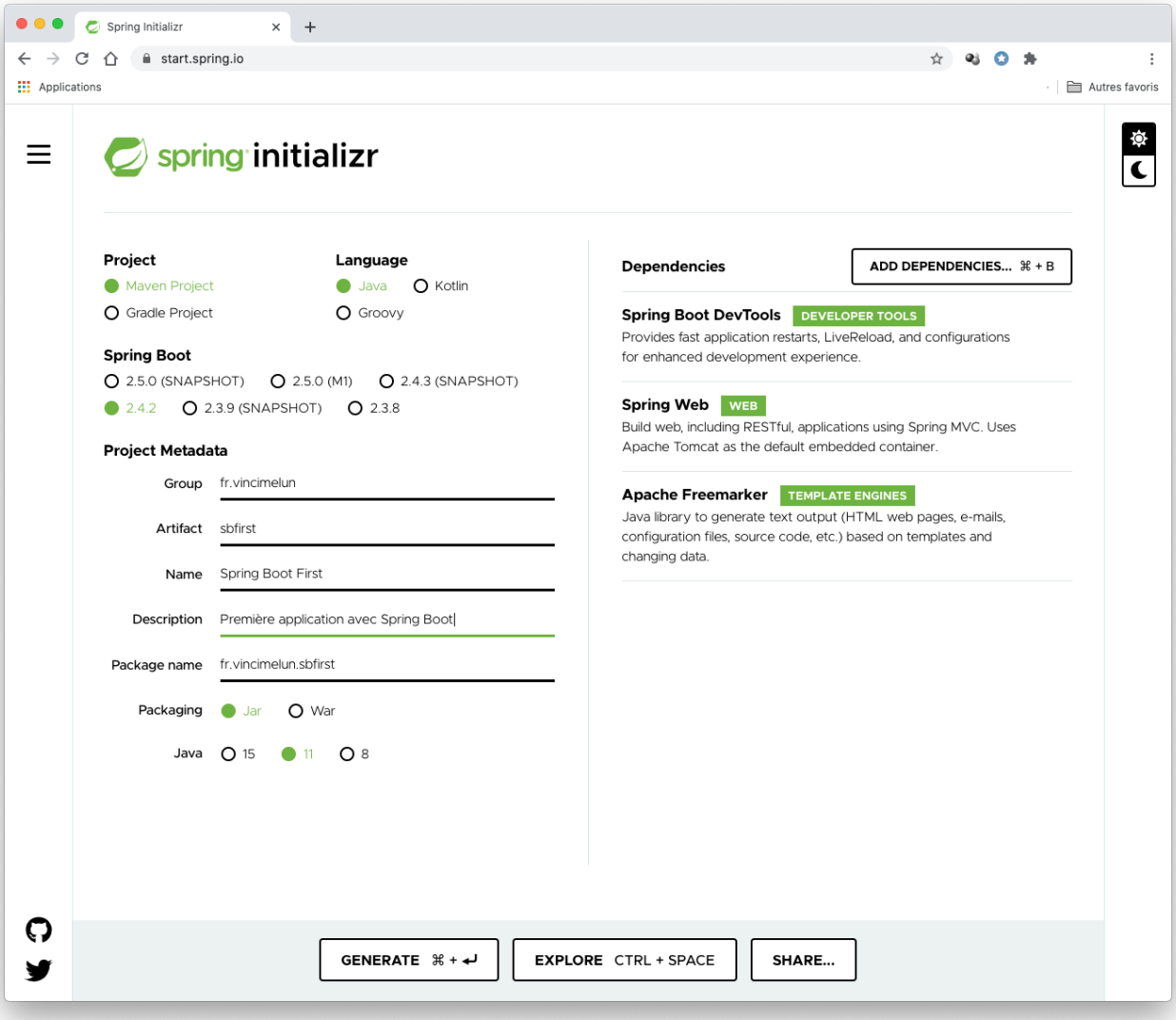
- Les dépendances de l'application

Première application

Une première application Spring Boot

Créer le projet

Dans Spring Initializr effectuez les choix suivants :



- Paramètres :
  - Project : Maven Project
  - Language : Java
  - Spring Boot version : 2.4.2
  - Métadonnées du projet :

    - Group : fr.vincimelun
    - Artifact : sbfirst
    - Name : Spring Boot First
    - Description : Première application Spring Boot
    - Package name : fr.vincimelun.sbfirst
    - Packaging : jar
    - Java (JVM) version : 11

- Dependencies :
    - Spring Boot DevTools
    - Spring Web
    - Apache Freemarker

Spring Initilizr génére un fichier `.zip`, `sbfirst.zip`, qui contient le projet prêt à l'emploi.

Arborescence du projet `sbfirst`

```
sbfirst
├─── .gitignore
├─── HELP.md
├─── mvnw
├─── mvnw.cmd
├─── pom.xml ①
└─── src
    ├─── main
    │    ├─── java
    │    │    └─── fr
    │    │        └─── vincimelun
    │    │            └─── sbfirst
    │    │                └─── SpringBootFirstApplication.java ②
    │    └─── resources
    │        ├─── application.properties ③
    │        ├─── static ④
    │        └─── templates ⑤
    └─── test
        └─── java
            └─── fr
                └─── vincimelun
                    └─── sbfirst
                        └─── SpringBootFirstApplicationTests.java
```

① Fichier `Maven` définissant le projet

② L'application principale (fonction main)

③ Le fichier de configuration principal de l'application

④

Le dossier pour stocker les éléments fixes comme les images ou les feuilles de style css

⑤ Le dossier de stockage des templates Freemarker ou Thymeleaf

La classe principale de l'application ressemble à celles qu'on peut écrire pour n'importe quelle application en mode ligne de commande : une fonction `main` statique qui permet d'instancier et de lancer l'application.

Listing 1. La classe principale de l'application

```
package fr.vincimelun.sbfirst;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootFirstApplication {

  public static void main(String[] args) { ①
   SpringApplication.run(SpringBootFirstApplication.class, args); ②
  }

}
```

① Fonction `main` de l'application

② Construction et lancement de l'application

Le fichier `application.properties` est vide, il ne contiendra que les éléments particuliers à la configuration de l'application comme les informations de connexion aux bases de données, ou la configuration du moteur de templates. Par défaut tout doit fonctionner sur les `conventions`.

Configurer le projet

```
spring.freemarker.template-loader-path= classpath:/templates
spring.freemarker.suffix= .ftl
```

Un premier contrôleur

Un contrôleur est une classe qui est annotée avec `@Controller` et qui contient des méthodes associées à des URI, ces methodes peuvent être associée à des annotations comme `@GetMapping`, `@PostMapping`, … ou `@RequestMapping` plus générique. Les paramètres passés à ces méthodes sont variables et interprêtés par le moteur de Spring MVC.

Par convention les contrôleurs sont créés dans le package `controller` de l'application

Listing 2. Un contrôleur simple

```
package fr.vincimelun.sbfirst.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class MainController {

    @GetMapping("/") ①
    public String index(){
        return "index"; ②
    }
}
```

① Route correspondant pour la méthode GET et l'URI `/`

② Nom du template associé, sans l'extention `.ftl`

Listing 3. Un template simple sans modèle associé

```
<html>
<body>
<h1>Bienvenue dans votre première application Spring Boot</h1>

</body>
</html>
```

Pour passer des données au template associé à l'URI, il suffit de déclarer un objet de type `Model` dans les paramètres de la méthode. Attention cet objet est instancié par Spring, il n'y a qu'à l'utiliser. On peut considérer un objet de type `Model` comme un `Map`.

Listing 4. Exemple de gestion d'un objet `Model`

```
    @GetMapping("/avecmodele")
    public String withModel(Model model){ ①
        model.addAttribute("nom", "Lagaffe"); ②
        model.addAttribute("profession", "Touche à tout"); ③
        return "avecmodele"; ④
    }
```

① Injection d'un objet de type Model dans la méthode

② Ajout de l'attribut `nom`

③ Ajout de l'attribut `profession`

④ Nom du template qui sera fusionné avec le `Model`

Le template associé peut utiliser les objets `nom` et `profession` du `Model`

Listing 5. Template associé au `Model`

```
<html>
<body>
<h1>Avec des données passées au template</h1>
Nom : ${nom} <br/>
Profession : ${profession} <br/>
</body>
</html>
```

Gérer les données renvoyées par un formulaire HTML

Pour récupérer les données d'un formulaire HTML ou utilise en paramètre un POJO annoté `@ModelAttribute` qui représente les données gérées par le formulaire. Par exemple si on souhaite récupérer des informations sur une personne représentée par son nom, son prénom et sa profession, il faut d'abord créer le POJO correspondant dans un package appelé `Model`

Listing 6. Le POJO `Personne`

```
package fr.vincimelun.sbfirst.model;

public class Person {
    private String name;
    private String givenName;
    private String job;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGivenName() {
        return givenName;
    }

    public void setGivenName(String givenName) {
        this.givenName = givenName;
    }

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }
}
```

Ensuite il faut créer deux méthodes dans la classe `MainController` une pour afficher le formulaire et une pour récupérer les données du formulaire et les traîter (ici il s'agira juste de les afficher).

```
@GetMapping("/personne") ①
public String personFormDisplay(){
  return "personform";
}

 @PostMapping("/personne") ②
 public String personFormProcess(
     @ModelAttribute(name="persondata") Person person, ③
     Model model) ④
 {
  model.addAttribute("persondisplay", person); ⑤
  return "persondisplay";
 }
```

① Méthode associée à l'URI `/personne` et à la méthode GET pour afficher le formulaire

② Méhtode associée à l'URI `/personne` et à la méthode POST pour traiter le formulaire

③ L'objet associé au formulaire pour récupérer les données (voir l'explication sur le formulaire), le nom est celui de la valise `FORM` du formulaire HTML `persondata`

④ L'objet `person` récupéré du formulaire est passé en paramètre au template d'affichage du résultat sous le nom `persondisplay`

Les templates associés sont les suivants

Listing 7. Formulaire de saisie : personform.ftl

```
<html>
<body>
<h1>Fiche d'information</h1>
<form action="/personne" method="post" name="persondata"> ①
    Nom : <input type="text" name="name"><br/> ②
    Prénom : <input type="text" name="givenName"><br/> ②
    Professtion : <input type="text" name="job"><br/> ②
    <input type="submit" value="Valider">
</form>
</body>
</html>
```

① Le nom associé au formulaire doit être le même que celui du `@ModelAttribute` dans la méthode `personFormProcess`

② Les noms des champs doivent être les mêmes que ceux de l'objet de type `Person` annoté par `@ModelAttribute`

Listing 8. Affichage des données

```
1 <html>
2 <body>
3 <h1>Fiche personnelle</h1>
4 <ul>
5     <li>Nom : ${persondisplay.name}</li>
6     <li>Prenom : ${persondisplay.givenName}</li>
7     <li>Profession : ${persondisplay.job}</li>
8 </ul>
9 </body>
10 </html>
```

`HttpServletRequest` et `HttpServletResponse`

Les objets `HttpServletRequest`, particulièrement utiles pour récupérer la session de l'utilisateur, et `HttpServletResponse` pour positionner les en-têtes HTTP sont toujours disponibles, il suffit de les déclarer en paramètre de la méthode contrôleur.

Exercices

ToDo

Reprendre l'exercice sur les ToDos du cours sur les servlets et l'adapter à Spring Boot. Pour cet exercice vous vous contenterez de créer une classe Controller nommée `ToDoController` que vous ajouterez au projet existant et qui gère l'URI `/todo`.

NagiosCfg

Reprendre l'exercice sur la génrération de fichiers de configuration Nagios et l'adapter à Spring Boot. Vous trouverez un corrigé version servlets en suivant le lien https://github.com/ocapuozzo/tomcat-switch-log . Pour cet exercice vous créerez un projet depuis `Spring Initializr`.