Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr

Plan de l'ECUE

1. Système de modules et services Java9+
2. (Rappels sur le) Développement d'applications Web avec Java
3. Modulariser les applications Java avec Spring
4. Bien structurer une application Web avec Spring MVC
5. Auto-configurer une application Web avec Spring Boot
6. Sécuriser une application Web avec Spring Security
7. Gérer des données massives avec Apache Kafka et Spring
8. Tester une application Web Spring
9. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan du cours

Pourquoi Spring Boot ?

- Pour vous éviter de passer du temps à :
  - définir les dépendances avec le framework Spring (context, web et webmvc) et les bibliothèques associées (Mapping JSON –Jackson, hibernate-validator, Thymeleaf, JUnit, ...)
  - gérer les versions de ces dépendances (parfois, elles ne sont pas compatibles)
  - écrire le code de configuration et de démarrage

  L'équipe Spring Boot a déjà fait les vérifications qu'il faut en ce qui concerne la compatibilité des versions des dépendances
- Pour vous permettre de vous concentrer sur le code *business* de votre application
- Pour vous permettre d'avoir une app *standalone* (avec un serveur embarqué) pour un déploiement facile sur le Cloud

spring initializr

- *spring initializr* est accessible ici : `https://start.spring.io/`
- Il permet de créer toute la structure d'un projet Spring Boot en quelques clics
- Il permet d'intégrer les dépendances nécessaires à votre application

* Aller sur le site et explorer les dépendances possibles (bouton "*Add Dependencies*")

- Dans le projet généré, il y a une classe annotée par cet outil qui permet de configurer et démarrer l'application

Tester *spring initializr*

- Tester l'outil en ligne, en choisissant :
  - Gradle, comme type de projet
  - Java, comme langage de prog
  - La dernière version stable de Spring Boot (2.6.4 ?)
  - des méta-données pour votre projet (donner des noms qui correspondent à votre projet)
  - Le packaging JAR
  - Java : sélectionner la version qui correspond à votre JDK
  - comme dépendances : ajouter Spring Web et PostgreSQL
- Télécharger le projet (un fichier ZIP)
- Décompresser le ZIP et importer le projet dans IntelliJ IDEA

Tester *spring initializr*

- Explorer la structure du projet :
  - ouvrir le script de build `build.gradle`. Noter les plugins et les dépendances
  - Dans le dossier "*External Libraries*", on voit le nombre assez important de bibliothèques téléchargés par Gradle pour vous (un nombre bien plus important que les dépendances dans le script de build)
  - ouvrir la classe dans `src/main/java/...`. C'est une classe exécutable (app *standalone*)
- Pour démarrer l'application, on peut exécuter la classe précédente ou bien, dans les *tasks* Gradle, exécuter `application/bootRun`

Préparer le terrain pour votre projet IWA

- Créer les sous-packages suivants dans le package de votre app : `controllers`, `models`, `services` et `repositories`
- Chaque sous-package contiendra un certain nombre de classes correspondant à la couche en question de l'app
- On reviendra sur ce projet plus tard

Autres outils

Autres possibilités pour travailler avec Spring Boot :

- Créer un nouveau projet sur IntelliJ IDEA :
  `File > New > Project... > Spring Initializr`
- Installer Spring Boot CLI (Command Line Interface) et tout faire en ligne de commande :
  `https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-cli.html`

Plan du cours

1. Introduction à Spring Boot
2. Spring Boot en action
3. Personnaliser une app Spring Boot
4. Déployer une app Spring Boot
5. Conclusion

Notion de starter

- Vous l'avez peut-être remarqué. Dans les dépendances déclarés dans le script de build, il y a des starters, sans numéros de versions
- Un starter : un moyen de regrouper des dépendances compatibles et traitant un aspect particulier de l'app (Web, persistance, sécurité, ...)
- Ceci vous permet de ne pas devoir gérer les versions des dépendances (et vérifier leur compatibilité)
- Pour voir les dépendances définies pour chaque starter, ouvrez la vue (Gradle) sur IntelliJ et ensuite *Dependencies*

Ajouter un nouveau starter

- Dans la section dependencies du script de build, ajouter : `implementation 'org.springframework.boot:spring-boot-starter-data-jpa'`
- Ceci permet ensuite de se connecter en utilisant JPA (Java Persistence API) à une source de données
- On va utiliser une source de données Postgres. Ajouter donc une autre dépendance : `implementation 'org.postgresql:postgresql'`

Créer une source de données

- On va utiliser Docker sur vos machines personnelles (voir explications ci-dessous)
- Sinon, il faudrait trouver un serveur de BdD accessible depuis les machines des salles de TP (et passer à la diapo 14)
- Pour rendre les données de la base de données Postgres persistantes, au delà de la vie du container Docker, il faudrait créer un volume : `docker volume create pgdata`
- Créer une source de données :

```
docker run --rm -P -p 127.0.0.1:5432:5432
       -v pgdata:/var/lib/postgresql/data
       -e POSTGRES_PASSWORD="postgres"
       -e POSTGRES_USER="postgres"
       --name postgres-covid postgres:alpine
```

Créer une source de données -suite-

Créer une source de données -suite- :

- Créer une base de données Postgres nommée `covid_alert_db` : utiliser le client Postgres que vous voulez. `psql` ou `pgAdmin`
  - Avec psql :
    - Soit via Docker :

      ```
      docker exec -it postgres-covid psql -U postgres
      ```

      Pour quitter le Shell psql : $\backslash q$
    - ou bien sur votre OS (avec Postgres déjà installé) :

      ```
      psql postgresql://postgres:postgres@localhost:5432/
      ```

Alimenter la source de données

- Ensuite sur le Shell psql :

  `create database covid_alert_db;`

- Y ajouter quelques tables (récupérer le script `create_tables.sql` sur le dépôt Git) :

  - Sur le Shell OS (et non psql) :

```
docker cp create_tables_covid.sql
   postgres-covid:/create_tables.sql
docker exec -it postgres-covid
   psql -d covid_alert_db -f create_tables.sql
   -U postgres
```

  Nous avons créé dans cette BdD trois tables, une pour les utilisateurs, la seconde pour les localisations géographiques et la dernière pour relier les deux

Se connecter à la source de données

- Le fichier `application.properties` qui se trouve dans `src/main/resources` permet de personnaliser l'app et son runtime (changer le numéro de port par exemple)
- Dans ce fichier, il faudra ajouter les propriétés suivantes :

```
spring.datasource.url=jdbc:postgresql://localhost:5432/covid_alert_db
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=none
spring.jackson.serialization.fail-on-empty-beans=false
```

Ajouter des objets Entity au modèle

- Créer dans le package models une classe User, annotée :
  `@Entity(name="users")`
  `@Access(AccessType.FIELD)`

  `users`, dans l'annotation ci-dessus, est le nom de la table
- Dans cette classe, ajouter des attributs privés, qui correspondent aux colonnes de la table (soit leur donner le même nom que dans la table, soit leur donner un nom différent (conventions de nommage Java, *Camel Case*), mais dans ce cas, il faudra les annoter
- Annoter l'attribut `user_id` :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long user_id;
```

JPA et Spring

- Générer des getters et des setters aux attributs
- Ajouter une deuxième classe `Location` pour les objets Entity correspondants aux enregistrements de la table `locations`
- Nous allons ajouter un attribut `locations` dans la classe `User` et l'annoter :

```
@ManyToMany
@JoinTable(name="user_locations",
joinColumns = @JoinColumn(name="user_id"),
inverseJoinColumns = @JoinColumn(name="location_id"))
private List<Location> locations;
```

- Ajouter un attribut `users` dans la classe `Location` et l'annoter :

```
@ManyToMany(mappedBy = "locations")
@JsonIgnore // Pour ne pas produire des cycles
private List<User> users;
```

Ajouter un getter et un setter à chaque attribut

Définir les *repositories*

- Dans le package, ajouter une interface par classe d'entity :

```
public interface UserRepository
        extends JpaRepository<User,Long> { }
```

```
public interface LocationRepository
        extends JpaRepository<Location,Long> {}
```

- Les interfaces sont vides. Elles héritent de plusieurs méthodes CRUD dans les super-interfaces

Définir les contrôleurs d'API

- Dans le package `controllers`, nous allons définir des classes annotées `@RestController` (`UsersController` et `LocationsController`)
  - l'annotation `@RestController` est l'équivalent de `@Controller` couplée avec `@ResponseBody` utilisée pour annoter les méthodes de la classe
- Modifier les classes écrites lors du dernier cours sur Spring MVC
- Cette annotation permet de traiter ces contrôleurs comme des *endpoints* Rest, qui par défaut recevront et retourneront des données au format JSON
- Nous allons annoter ces classes avec `@RequestMapping("/api/v1/users")` (ex pour la classe `UsersController`)

Définir les contrôleurs d'API -suite-

- Déclarer un attribut par classe pour référencer le repository. Ex :

```
@Autowired
private UserRepository userRepository;
```

- Ajouter les méthodes de l'API dans ces classes. Ex :

```
@GetMapping
public List<User> list() {
  return userRepository.findAll();
}
@GetMapping
@RequestMapping("{id}")
public User get(@PathVariable Long id) {
  return userRepository.getById(id);
}
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public User create(@RequestBody final User user) {
  return  userRepository.saveAndFlush(user);
}
```

Les méthodes de l'API

- Les paramètres et les valeurs de retour des méthodes précédentes sont gérés par Spring MVC et la bibliothèque de sérialisation (Jackson, par défaut)
- Ils sont automatiquement sérialisés et désérialisés au format JSON
- Exemple 1 : la méthode `get(...)` retourne un objet de type `User`. Cet objet est automatiquement transformé en donnée JSON avant d'être renvoyée dans la réponse HTTP
- Exemple 2 : le paramètre de la méthode `create(...)` est injecté par Spring MVC avec un objet construit à partir de la donnée JSON reçue dans le body de la requête HTTP Post
- Tout cela est transparent pour le développeur

Méthodes *Delete* et *Update*

- Exemple d'une méthode *Delete* :

```
@RequestMapping(value = "{id}",method = RequestMethod.
    DELETE)
public void delete(@PathVariable Long id) {
  // Toujours verifier s'il faut supprimer aussi
  // les enregistrements enfants
  userRepository.deleteById(id);
}
```

Méthodes *Delete* et *Update* -suite-

- Exemple d'une méthode *Update* :

```
@RequestMapping(value="{id}",method = RequestMethod.PUT)
public User update(@PathVariable Long id, @RequestBody User
    user) {
  // TODO: Ajouter ici une validation si tous
  // les champs ont ete passes
  // Sinon, retourner une erreur 400 (Bad Payload)
  User existingUser = userRepository.getById(id);
  BeanUtils.copyProperties(user,existingUser,"user_id");
  return userRepository.saveAndFlush(existingUser);
}
```

Possible d'utiliser `RequestMethod.PUT` pour une mise à jour de tous les champs et `RequestMethod.PATCH` pour une MAJ partielle

- Dernier paramètre de copyProperties - très important. Pourquoi ?

Tester vos services Rest

- Pour éliminer certaines erreurs liées à la sérialisation des objets du modèle, annoter les classes Entity avec :
  `@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})`
- Démarrer l'application : exécuter la classe avec le main ou la tâche Gradle bootRun
- Pour les méthodes GET, utiliser le navigateur pour envoyer vos requêtes et voir les réponses (vous pouvez utiliser les outils de développement de votre navigateur pour inspecter les requêtes/réponses HTTP) :
  `http://localhost:8080/api/v1/users/`
  `http://localhost:8080/api/v1/users/1`
- Pour les autres méthodes, utiliser curl en ligne de commande ou Postman

Tester vos services Rest -suite-

- Exemple avec curl pour la méthode POST
  - Créer un fichier JSON (user.json) avec les données de la requête :

```
{"first_name":"Mila",
 "last_name":"No",
 "email":"mila.no@github.io",
 "phone_number":"06 62 34 00 99",
 "password":"123456789"}
```

  - Exécuter la commande :

```
curl -d @user.json -H "Content-Type: application/json" -X POST
http://localhost:8080/api/v1/users/
```

Gestion des erreurs

- Essayer d'envoyer une requête GET pour récupérer un *User* ou une localisation (*Location*) inexistante
- Suprise : la réponse est 500 (erreur interne du serveur) et non 404 (ressource non trouvée)
- Pour améliorer votre code :

```
@GetMapping @RequestMapping("{id}")
public User get(@PathVariable Long id) {
  if(userRepository.findById(id).isEmpty()) {
    throw new ResponseStatusException(HttpStatus.
    NOT_FOUND, "User with ID "+id+" not found");
  }
  return userRepository.getById(id);
}
```

Plan du cours

1. Introduction à Spring Boot
2. Spring Boot en action
3. Personnaliser une app Spring Boot
4. Déployer une app Spring Boot
5. Conclusion

Sources possibles de configuration

External Sources

Command line parameters, JNDI,
OS environment variables

Internal Sources

Servlet parameters, property files,
java configuration

Ordre de précédence dans les sources de configuration

Order of precedence:

1. Command line args
2. SPRING_APPLICATION_JSON args
3. Servlet parameters
4. JNDI
5. Java System Properties
6. OS environment variable
7. Profile properties
8. Application properties
9. @PropertySource annotations
10. Default properties

Recommandations pour les sources de configuration

- Ne pas multiplier le nombre de sources de configuration (utiliser 2 ou 3 max)
- Idéalement, utiliser une source interne (fichier de propriétés de l'application, `application.properties` vu avant, par ex.) et une source externe (variables d'environnement OS, par ex.)
- Utiliser la source externe pour les paramètres de config sensibles (mots de passe de connexion, ...) parce que, souvent, les sources internes font partie du code/config de l'application, et sont placées dans des dépôts de code (Git, ...)

Propriétés Spring Boot communes

- Propriétés Spring Boot communes sont listées ici :
  `https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html`
  A googler (*Spring Boot Common Application properties*) si l'URL ne marche plus
- Exemple : `server.port` (valeur par défaut 8080)
- Tester le changement de port et redémarrer l'app

Source externe

- Les paramètres de connexion à la base de données sont au mauvais endroit (`application.properties`), qui fait partie du code (si c'est sur un dépôt Git, ces paramètres sont rendus accessibles à plus de personnes que ce qu'il faut)
- Nous allons maintenant les sortir de là pour les mettre comme variables d'environnement :

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}
```

Variables d'environnement en Dév

- Sur IntelliJ IDEA, il faudrait ouvrir le menu *Run* > *Edit Configurations* puis *Environment variables*
- Ajouter les 3 variables d'environnement précédentes
- Redémarrer votre app pour tester

Paramètres de config selon le profil

- Selon si l'on est en "Dev", "Recette", "Prod", ... on est amené à utiliser différentes valeurs pour les paramètres de configuration
- Dans Spring Boot, il est possible de définir des fichiers selon le "**profil**", `application-{profil}.properties`
- Exemples : `application-dev.properties`, `application-prod.properties`, ...
- Ensuite, il faudra indiquer à Spring Boot quel environnement utiliser en paramétrant :
  -Dspring.profiles.active=dev
  Ici, c'est une option de la VM qui a été utilisée (sur IntelliJ IDEA, Run > Edit Configuration)

Exercice

- Ajouter un fichier `application-prod.properties` dans `src/main/resources`
- Ajouter dans ce fichier le paramètre :
  `logging.level.fr=WARN`
  Ainsi, tous les logs INFO provenant des packages commençant par fr ne seront plus affichés (on est dans un environnement de prod, on veut avoir les logs les plus utiles)

Fichiers de config YML

- Ce sont des fichiers de config mieux structurés que les .properties (on voit la hiérarchie des paramètres de config)
- Ils sont pris en charge par Spring Boot (même avec des profils)
- Créer un fichier `application.yml` dans le dossier `src/main/resources`
- Supprimer la ligne paramétrant le port du serveur du fichier .properties
- Ajouter dans le fichier .yml :

```
server:
  port: 5000
```

- Dans une vraie application, ne pas mélanger .yml et .properties

Nouvelles propriétés

- Dans le fichier de propriétés d'une application, on a la possibilité de définir nos propres paramètres de config
- Exemple : `app.version=1.0.0`
- Pour récupérer dans le code la valeur de ce paramètre, il suffit de déclarer dans l'un des beans Spring (un contrôleur par ex.) un attribut et demander son injection par la DI Spring :

```
@Value("${app.version}")
private String appVersion;
```

- Exercice : définir un contrôleur (`HomeController`) qui fournit un endpoint (méthode annotée par `@GetMapping` et `@RequestMapping("/")`) qui retourne un objet Map qui contient la version de l'application

Plan du cours

1. Introduction à Spring Boot
2. Spring Boot en action
3. Personnaliser une app Spring Boot
4. Déployer une app Spring Boot
5. Conclusion

Architectures `Containerless`

- Vous l'avez peut-être remarqué, avec Spring Boot, on n'avait pas à gérer un container de servlets ou un serveur d'app (le démarrer, l'arrêter, ...)
- Spring Boot embarque un serveur (Tomcat, par défaut) dans le framework, le démarre pour vous, et y déploie l'application
- C'est ce que l'on appelle une application avec une "*Containerless Architecture*"

Modifier le serveur par défaut

- Dans le script de build, ajouter la dépendance vers un autre serveur (ex. Jetty) :

```
implementation('org.springframework.boot:spring-boot-
    starter-jetty')
```

- exclure la dépendance vers `spring-boot-starter-tomcat`, qui est incluse dans `spring-boot-starter-web`, mais qui peut être incluse par d'autres dépendances (donc, autant le faire pour tout le projet) :

```
configurations.implementation {
  exclude module: "spring-boot-starter-tomcat"
}
```

A placer en dehors de la section *dependencies*

Déployer l'app comme un JAR exécutable

- Il est possible d'empaqueter l'application dans un JAR exécutable avec la task `bootJar` de Gradle
- Ceci permet de générer un fichier .jar dans le dossier `build/libs`
- Ce jar peut être exécuté ensuite par un simple : `java -jar votre-projet-version.jar`
- Cela démarre le serveur, déploie l'application, ...

Plateformes Cloud

- Voici quelques plateformes pouvant héberger des app Spring Boot :
  - Cloud Foundry (CF)
  - Heroku
  - Google Cloud
  - Amazon Web Services
  - Microsoft Azur
- Chacune a ses propres paramètres de config (DATABASE_URL sur Heroku ou CF par ex., au lieu de DB_URL en Dév)↘
- Il faut faire attention, dans le Cloud, la récupération des logs peut être complexe, l'intégration de certains services peut être différente de ce qui a été prévu en Dév, des firewalls peuvent empêcher certains services de communiquer, ...
- Le plus simple est de Dockeriser l'application d'abord

Déployer votre app Spring Boot

- Créer un compte gratuit sur Heroku :
  `https://dashboard.heroku.com/apps`
  Vous aurez un espace de 500Mo
- Placer votre application (Covid Alert) dans un dépôt :
  - Github
  - ou Heroku Git
    Pour ce dernier il faudra installer Heroku CLI : (faites-le, on en a besoin plus loin)
    `https://devcenter.heroku.com/articles/heroku-cli#download-and-install`
- Créer une application sur votre compte en ligne Heroku et la lier au dépôt Git
- Déployer l'application (onglet *Deploy*)

Déployer votre app Spring Boot sur Heroku

- Heroku va détecter que c'est une application Gradle et Spring Boot. Il va installer un JDK et démarrer l'app
* Par défaut, Heroku va utiliser une ancienne version du JDK (8, à la date de création de ce cours). Dans ce JDK l'API de la classe `Optional<T>` est différente de celle utilisée dans les diapos précédentes. Pour éliminer l'erreur de compil., remplacer `isEmpty()` dans la classe `UsersController` par `"! ... isPresent()"`
- Si pas d'erreurs, cliquer sur le bouton Open app (en haut à droite)
- Si conflit de port remonté dans les logs de Heroku, définir dans le fichier `application.properties` : `server.port=${PORT:8080}`

Déployer votre app Spring Boot sur Heroku -suite-

- Vous pouvez visualiser les logs, en cliquant sur le bouton *More* puis *View logs*
- Pour suivre la progression du déploiement, aller dans l'onglet *Activity*
- Heroku a dû détecter l'utilisation d'une base de données Postgres (onglet *Resources*). Si ce n'est pas le cas (parfois, à cause d'erreurs lors du déploiement), ajouter dans l'onglet *Resources* un Add-on : Heroku Postgres

Déployer votre app Spring Boot sur Heroku

- Dans l'onglet *Settings*, cliquer sur *Reveal Config Vars* pour voir l'URL de la BdD à utiliser
- Ajouter une nouvelle variable (`DB_URL`) avec la valeur de `DATABASE_URL`
- Ajouter les tables dans la base. Pour cela, il faudra utiliser Heroku CLI :

  ```
  heroku pg:psql <votre-bdd> -app
  <votre-app-heroku>
  ```

  A partir de là, copier/coller les requêtes SQL pour créer les tables et y insérer quelques enregistrements

Déployer votre app Spring Boot sur Heroku

- Attention, avec l'add-on gratuit, le nombre de connexions est limité à 20 sur Postgres (une connexion par déploiement de l'app)
- L'objectif ici est juste de faire fonctionner une première version de votre app sur ce Cloud

Déploiement Containerful (WAR)

- On peut également prévoir un déploiement sous la forme d'un WAR (Web Archive) de l'application
- Contrairement au déploiement JAR avec une architecture containerless (serveur/container embarqué dans le JAR), ici le WAR contient juste l'application
- On suppose que le container sera fourni et que le WAR y sera installé

Déploiement Containerful (WAR) -suite-

Très simple à mettre en place :

- sur Gradle, ajouter une dépendance à un serveur (Tomcat, par ex.) et indiquer qu'il a une configuration/scope `compileOnly` Ceci permet de l'exclure du packaging WAR
- modifier la classe qui contient la méthode `main()` pour étendre la classe `SpringBootServletInitializer`
- remettre dans les properties les paramètres de config (DB_URL, ...)
- Exécuter la tâche Gradle de packaging
- Installer le WAR obtenu dans le dossier `webapps/` d'un Tomcat qui tourne

Déploiement avec Docker

- Suivre le tutoriel disponible ici : `https://spring.io/guides/gs/spring-boot-docker/`
- Adapter le tutoriel à l'application Covid Alert ! pour construire une image Docker
- Déposer sur Moodle le Dockerfile de votre image

- Déposer sur Moodle tout le code écrit dans ce cours

Plan du cours

1. Introduction à Spring Boot
2. Spring Boot en action
3. Personnaliser une app Spring Boot
4. Déployer une app Spring Boot
5. Conclusion

Wrap-up

- Spring Boot simplifie le développement d'application Web Spring MVC, grâce à ses *initializers* et *starters*
- Il évite de passer du temps dans la configuration d'une application
- Il gère de façon transparente toutes les dépendances compatibles
- Framework personnalisable facilement et pris en charge par les fournisseurs main-stream de Cloud

Pour aller plus loin

Organiser son application à micro-services selon le patron `API Gateway` :

- `https://medium.com/an-idea/spring-boot-microservices-api-gateway-e9dbcd4bb754`
- `https://spring.io/projects/spring-cloud-gateway`

Références biblio

- Site Web de Spring Boot :
  `https://spring.io/projects/spring-boot`
- Tutoriels sur Pluralsight et Baeldung
- Livre sur le sujet :
  Spring Boot in Action. Craig Walls. Manning Publications. Dec 2015.