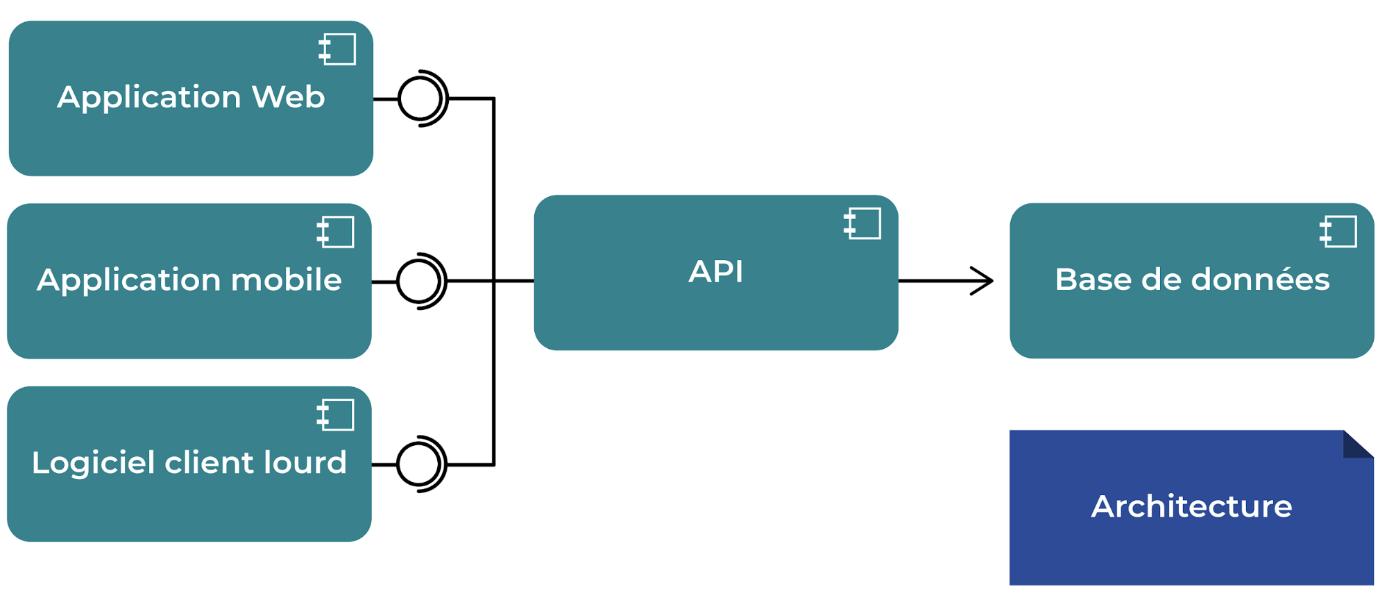
**Plongez-vous dans votre nouvelle mission**

Votre mission, si vous l’acceptez, est d’aider HR Association dans la création de sa suite d’outils de gestion des employés à destination des petites entreprises.

Voici l’architecture imaginée :

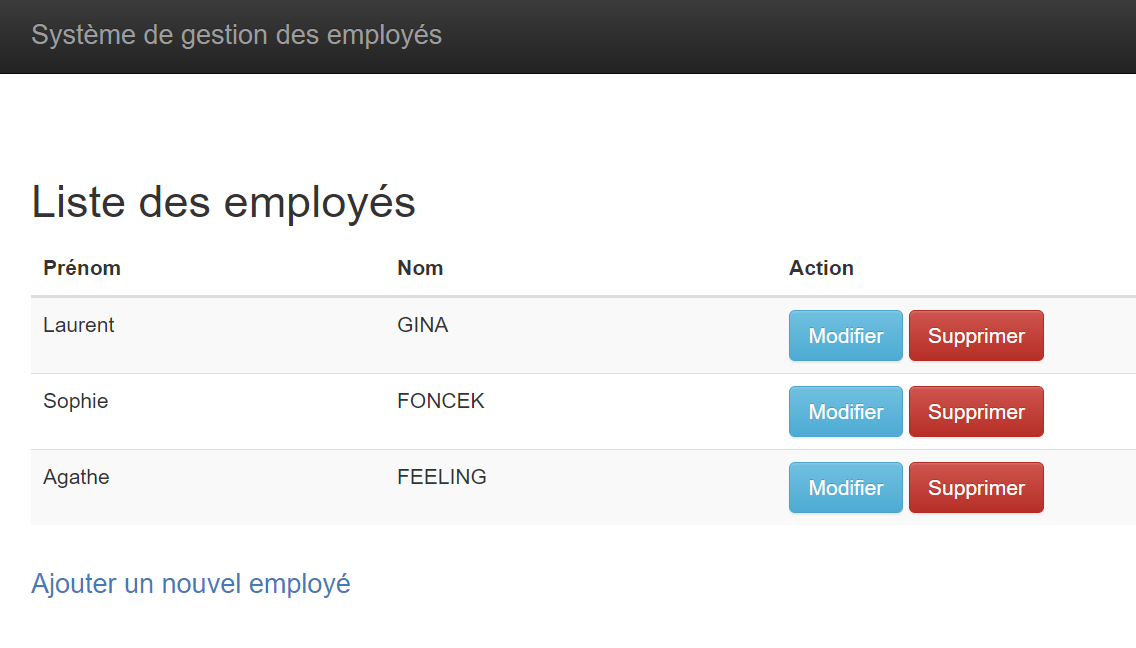
Architecture des composants logiciels de HR Association.

Lors de la partie 3 de ce cours, nous avons réalisé l’API qui sera exploitée par les différents clients.

Aujourd’hui, l’objectif est de développer une application web qui permettra de :

1. Visualiser les employés.
2. Ajouter un nouvel employé.
3. Modifier un employé.
4. Supprimer un employé.

Voici un visuel :

Page d’accueil du système de gestion des employés

Techniquement, nous allons évidemment utiliser **Spring Boot**, et je vous apprendrai comment renvoyer une page HTML et comment communiquer avec une API. Ce sera passionnant !

Allons-y ! 

**À vous de jouer !**



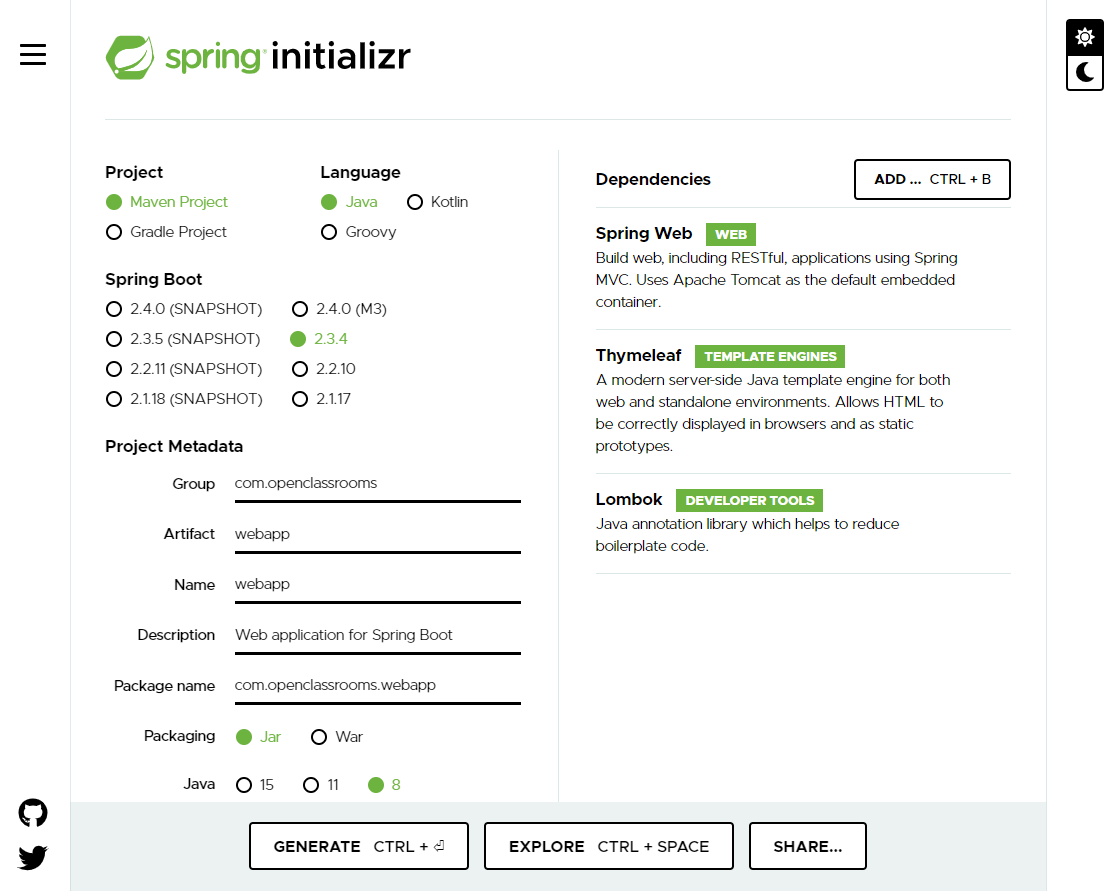
Je vous laisse réaliser la première étape, à savoir **créer la structure minimale** de ce projet. Utilisez **Spring Initializr** ou bien **STS**, le résultat sera le même !

L’important est que vous choisissiez bien vos **starters** en fonction de ce que vous souhaitez réaliser. Cela ne sera pas facile pour vous, mais en parcourant attentivement la liste des starters et en lisant leur description, vous devriez pouvoir vous approcher du bon résultat.

Dès que vous avez fini, venez voir la correction que je vous propose.

**Correction : identifiez les starters permettant de créer l’API**

Voici ma solution :

Formulaire du site Spring Initializr complété pour le projet

Pour les **“Project Metadata**”, je suis resté simple, vous avez toute latitude pour saisir les données de votre choix.

Pour les “**Dependencies**”, c’est ici que ça se complique !

Je pense que **Spring Web** a été une évidence pour vous ! Je vous félicite d’avoir mis le doigt dessus.  Sachez que cela ne nous permettra pas uniquement de fournir une page HTML à afficher. Ce starter contient également le code pour communiquer avec une API.

Lorsque vous avez parcouru les “**Template Engines**”, vous avez certainement noté les mentions du HTML, et cela vous a peut-être donné envie de choisir un de ces starters. Si c’est le cas, vous avez eu raison !

En plus de Spring Web, utilisons **Thymeleaf**qui est l’un des**moteurs de template** (template engine) les plus couramment utilisés.

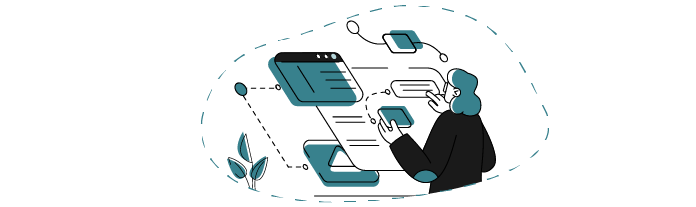
En deux mots, un moteur de template HTML va nous**aider à formater la page HTML** que nous voulons renvoyer.

Mon dernier starter, Lombok, est une librairie qui permet d’optimiser certaines classes. Par exemple, grâce à Lombok, en ajoutant l’annotation @Data à une classe, je n’ai plus besoin d’écrire les getters et setters, magique !

**En résumé**

* Pour implémenter une application web qui communique avec une API, 2 éléments sont essentiels :
  + Le starter **Spring Web** qui permettra de renvoyer des pages HTML et communiquer avec l’API.
  + Un **moteur de template** comme Thymeleaf qui permet de formater le HTML.

## Configurez et structurez votre projet avec des packages



Notre structure minimale étant prête, il nous faut désormais :

1. Structurer avec des packages.
2. Configurer notre application.

### À vous de jouer !



Pour la structure de packages, je vous encourage à reprendre ce que je vous ai montré dans la [partie 2](https://openclassrooms.com/fr/courses/6900101-creez-une-application-java-avec-spring-boot/7077993-structurez-et-configurez-votre-projet). Comme indiqué, c’est une organisation standard qui s’applique très bien aux projets web.

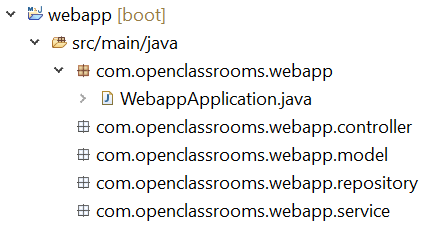
Pour la configuration, vous pouvez également reprendre ce qu’on avait vu ensemble précédemment concernant le fichier application.properties. On reste sur une configuration standard.

Je vous laisse la main, essayez de votre côté, on se retrouve dans quelques minutes pour débriefer ces étapes.

### Correction

#### Créez les packages

Certainement le plus facile de cet exercice, voici tout simplement la capture d’écran du résultat :

Arborescence des packages du projet

Cette structure en couches est utilisable**quel que soit l’objectif fonctionnel**. Nous avons pu l’utiliser pour le HelloWorld de la [**partie 2**](https://openclassrooms.com/fr/courses/6900101-creez-une-application-java-avec-spring-boot/7077993-structurez-et-configurez-votre-projet), l’API de la [**partie 3**](https://openclassrooms.com/fr/courses/6900101-creez-une-application-java-avec-spring-boot/7078013-configurez-et-structurez-votre-api-avec-des-packages), et désormais pour cette application web.

#### Définissez les propriétés

Regardons maintenant le résultat de la configuration du fichier application.properties.

Et voici mon résultat en image :

#Global configuration

spring.application.name=webapp

#Tomcat configuration

server.port=9001

#Log level configuration

logging.level.root=ERROR

logging.level.com.openclassrooms=INFO

logging.level.org.springframework.boot.web.embedded.tomcat=INFO

* spring.application.name=webapp : pour définir un nom à l’application ;
* server.port=9001 : pour ne pas être sur le port par défaut 8080, ni sur le port 9000 qui est utilisé par l’API (Cf. partie 3 chapitre 2) ;
* logging.level :
  + root=ERROR : par défaut, seules les traces en ERROR s’affichent. L’idée est simple : réduire les affichages dans la console de toutes les "3rd party",
  + com.openclassrooms=INFO : pour ce qui est de notre code, on est en INFO pour avoir du détail,
  + org.springframework.boot.web.embedded.tomcat : permet de voir dans la console le port utilisé par Tomcat au démarrage.

### Allez plus loin dans la gestion des propriétés

Je souhaite saisir l’opportunité de ce chapitre pour vous apprendre à créer de la configuration **custom**. Jusqu’à présent, je vous ai montré comment configurer votre application en fonction de propriétés existantes.

Mais comment faire si on souhaite créer de nouvelles propriétés ?

Rassurez-vous, Spring Boot s’occupe de tout ! (Comme d’hab )

Dans la partie 2 du cours, chapitre 2, j’ai eu l’occasion de vous apprendre que Spring Boot lit **nos sources de propriétés**, et rend disponible les propriétés via des beans.

Le fichier application.properties correspond à l’une de ces sources de propriétés.

Jusqu’à présent, nous avons saisi des valeurs pour des propriétés existantes. Propriétés utiles à des classes de Spring qu’on ne manipule pas. Cependant, si je souhaite ajouter une nouvelle propriété, comment y accéder dans mon code ?

Un screencast, ça vous tente ? 

Récapitulons !

Etape 1 : Créez la nouvelle propriété.

J’ai commencé par ajouter ma nouvelle propriété dans mon fichier application.properties (pour rappel, ce fichier est ma source de propriétés).

Étape 2 : Créez le bean associé.

J’ai ensuite créé une nouvelle classe nommée CustomProperties. Je l’ai annotée avec :

* @Configuration : permet de déclarer la classe en tant que bean de configuration.
* @ConfigurationProperties(prefix = “com.openclassrooms.webapp”) : demande à Spring de charger les properties qui commencent par “com.openclassrooms.webapp” au sein des attributs de la classe.
* @Data : génère automatiquement getter/setter pour les attributs de classe.

J’ai conclu cette classe en ajoutant l’attribut apiUrl.

package com.openclassrooms.webapp.repository;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Configuration;

import lombok.Data;

@Configuration

@ConfigurationProperties(prefix = "com.openclassrooms.webapp")

@Data

public class CustomProperties {

private String apiUrl;

}

Étape 3 : Utilisez les propriétés dans le code.

Pour la démonstration, j’ai modifié la classe WebappApplication, afin qu’elle implémente un CommandLineRunner. Ainsi, dans la méthode run, j’ai voulu vérifier que j’accédais bien à la propriété, en affichant cette dernière dans la console.

Le point clé à retenir est l’injection du bean CustomProperties, comme vous pouvez le voir aux lignes 16/17 ci-dessous.

Il est ensuite facile d’accéder à la propriété, comme le montre la ligne 25.

package com.openclassrooms.webapp;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.openclassrooms.webapp.repository.CustomProperties;

import lombok.Data;

@Data

@SpringBootApplication

public class WebappApplication implements CommandLineRunner {

@Autowired

private CustomProperties properties;

public static void main(String[] args) {

SpringApplication.run(WebappApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

System.out.println(properties.getApiUrl());

}

}

Cette étape 3 servant uniquement à démontrer que les étapes précédentes sont valides, je vais revenir à la version initiale de la classe WebappApplication pour la suite du cours.

Il est également courant de créer un nouveau package nommé "configuration" pour mettre les classes associées. N’hésitez pas à le faire.

### En résumé

* La structure des packages reste le standard : **controller/service/repository/model**.
* Le fichier**application.properties** est ma source de propriétés.
* Je peux créer des propriétés et les manipuler dans mon code, notamment grâce à l’annotation **@ConfigurationProperties**.