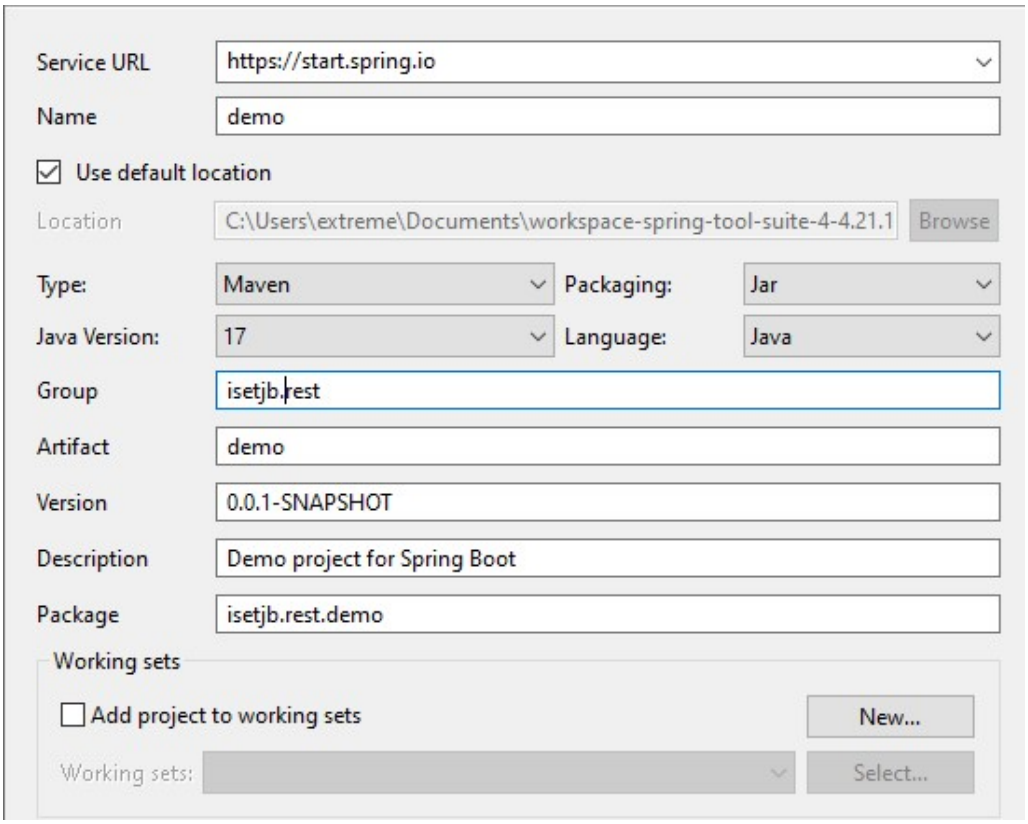
Atelier SOA : TP 1 – Fournisseur Service REST - Gestion des Utilisateurs

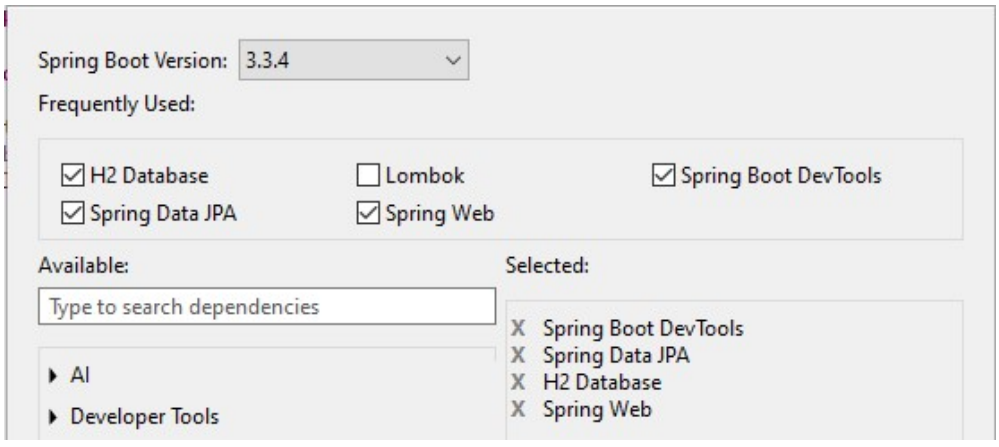
1/ Création du projet

Créer sous Spring Tools Suite un projet Spring Boot selon les spécifications suivantes :

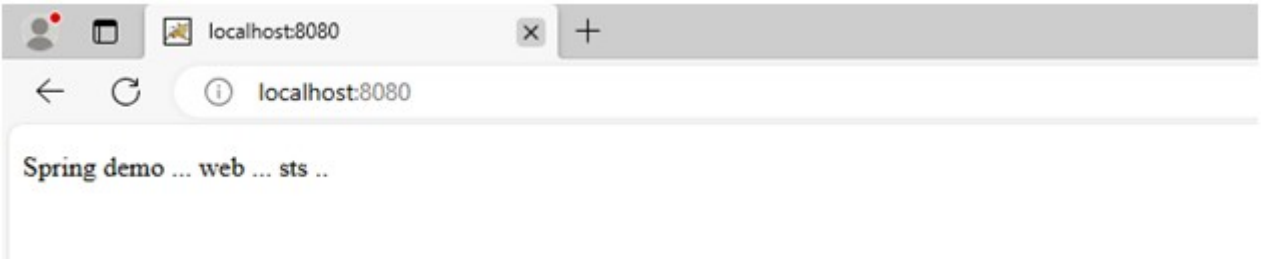
- Nom : demo
- Type : Maven
- Java version : 17
- Packaging : Jar
- Groupe : isetjb.rest
- Package : isetjb.rest.demo



Utiliser Spring Boot Version 3.3.4. Les starters requis pour notre application web sont :



Le port par défaut du serveur tomcat intégré est 8080



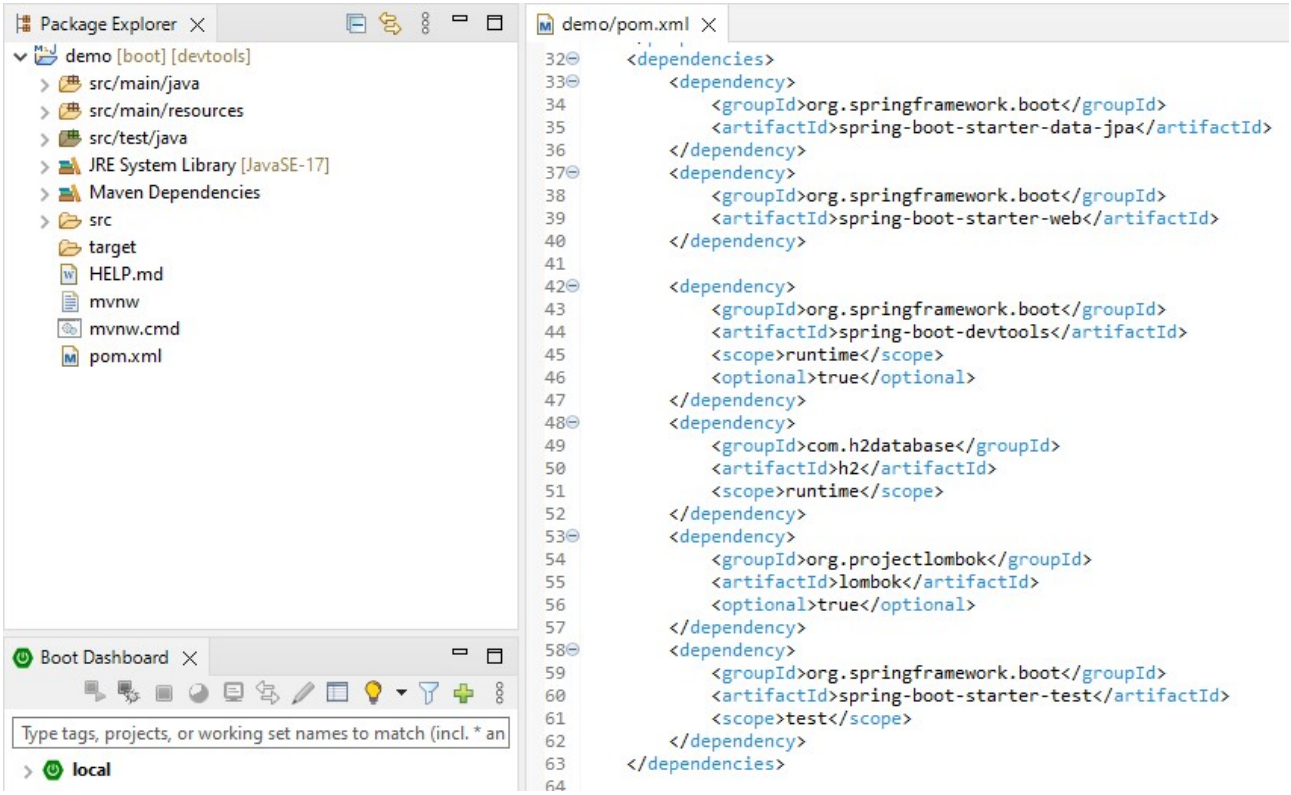
On peut modifier le numéro de port à l'aide du fichier de configuration « applications.properties » du dossier src/main/ressources. Ouvrer ce fichier et ajoutez les deux lignes suivantes :

#Tomcat Configuration

Server.port=88

Un projet Maven possède toujours un fichier pom.xml à la racine du projet. Ce fichier XML est le descripteur du projet et contient toutes les informations nécessaires à Maven pour gérer le cycle de vie du projet ; en particulier les propriétés et les dépendances du projet.

Le nom du fichier pom.xml vient de POM (Project Object Model).



Section « dependencies » du fichier pom.xml de notre projet

Comment structurer et configurer notre projet ?

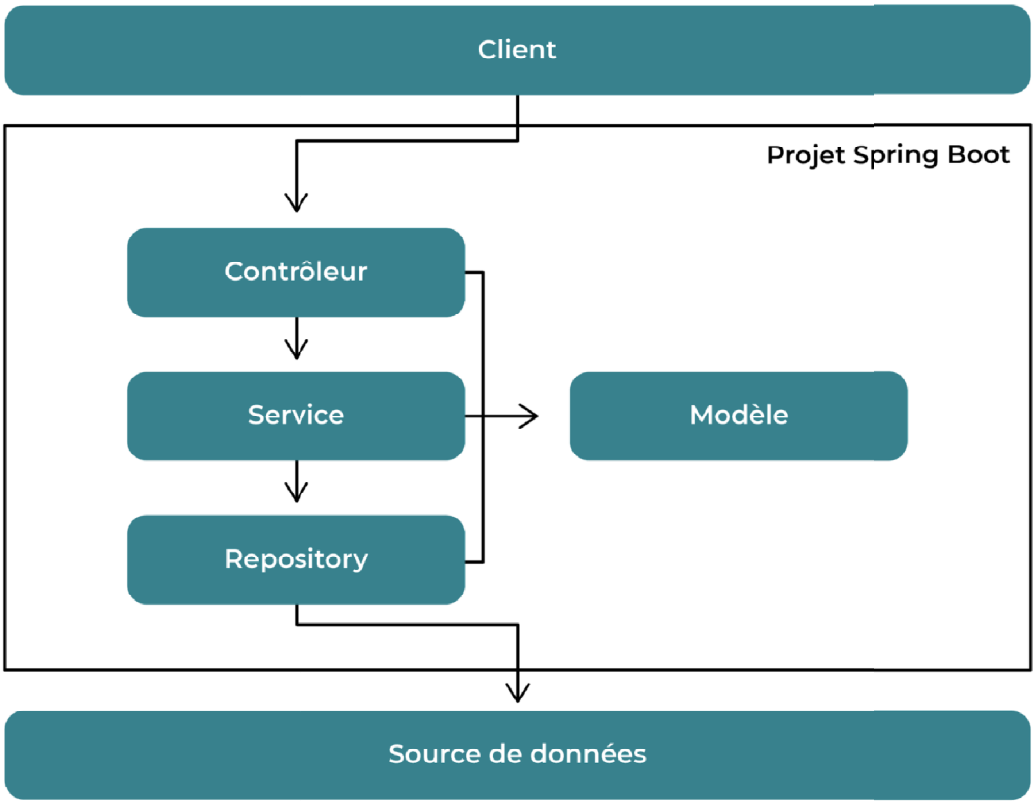
Malgré qu'on ne connaît pas encore les classes qui seront produites, on peut définir nos packages et leurs dépendances en recourant aux bonnes pratiques.

Premièrement, gardons à l'esprit que notre projet est un projet web (Spring Boot est particulièrement utilisé dans le contexte d'application web, mais ça ne se limite pas à cela).

Deuxièmement, la majorité des applications ont la nécessité d'interagir avec des données externes (par exemple une base de données, un autre programme, ou même le système de fichiers).

De ces différents besoins, une architecture en couches a émergé, avec un rôle pour chaque couche :

- couche Control : gestion des interactions entre l'utilisateur de l'application et l'application ;
- couche Service : implémentation des traitements métiers spécifiques à l'application ;
- couche Repository : interaction avec les sources de données externes ;
- couche Model : implémentation des objets métiers qui seront manipulés par les autres couches.



Représentation visuelle de l'architecture en couches

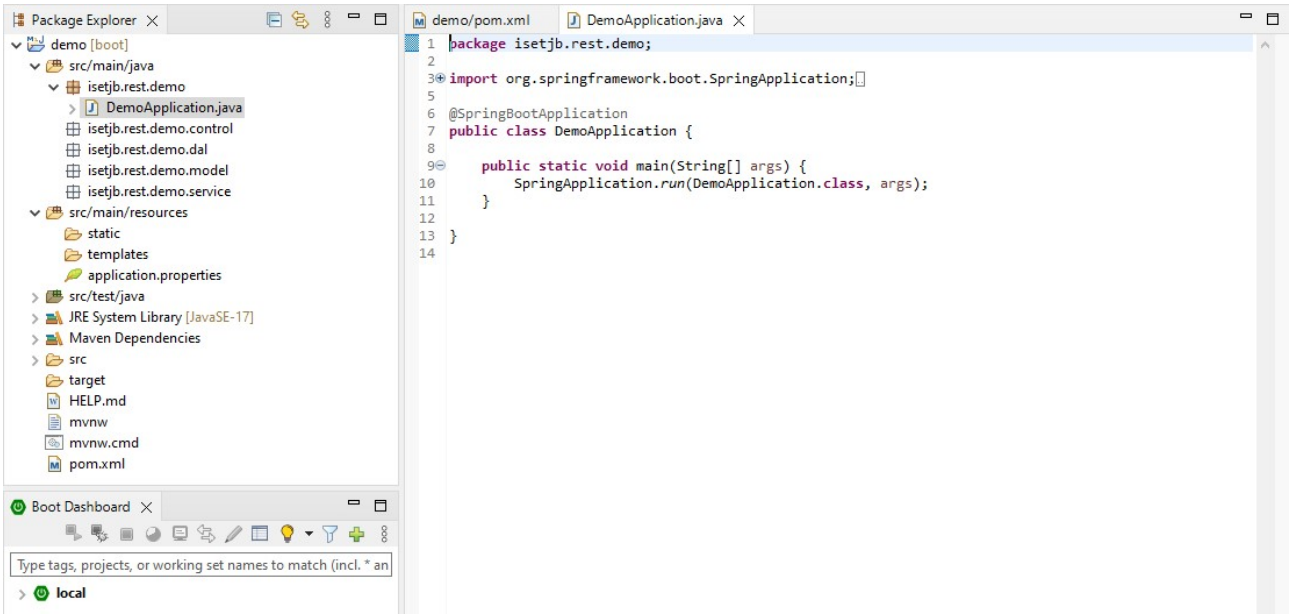
Le sujet ici étant de gérer des utilisateurs.

Notre application Web est minimale. On se contentera d'offrir un CRUD (Create, Read, Update, Delete) pour les données des utilisateurs.

Il nous faut maintenant structurer avec des packages, comme nous l'avons illustré dessus.

Créez des packages

Certainement le plus facile de cet exercice, voici tout simplement le résultat avec une capture d'écran :



Arborescence des packages du projet

3/ Configurer notre application. En particulier pour l'utilisation de la base de données H2

Configurer l'accès à la base de données

La méthode la plus facile pour configurer une base de données H2 est d'utiliser le fichier de configuration « applications.properties », comme le montre l'exemple suivant :

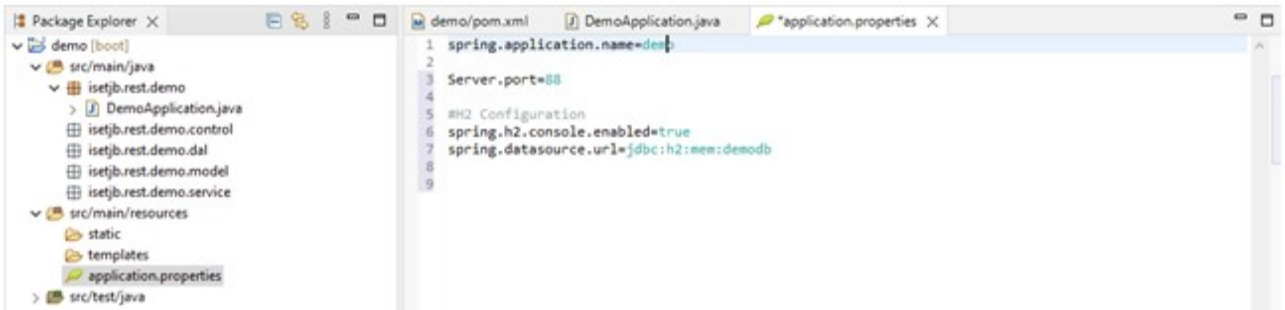
#Tomcat Configuration

Server.port=88

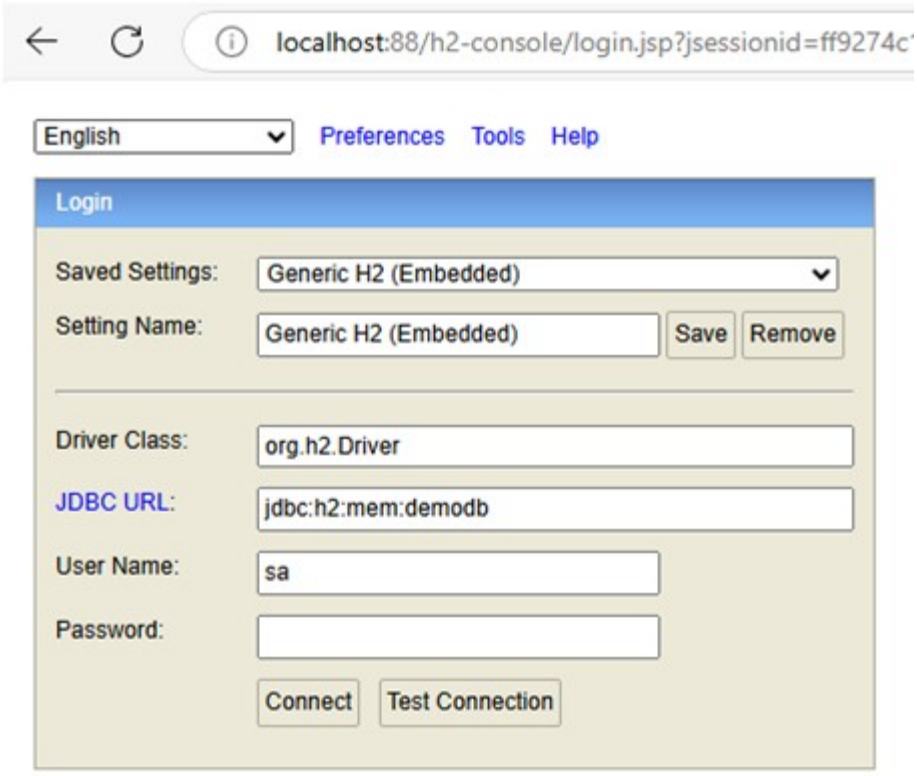
#H2 Configuration

spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:demodb



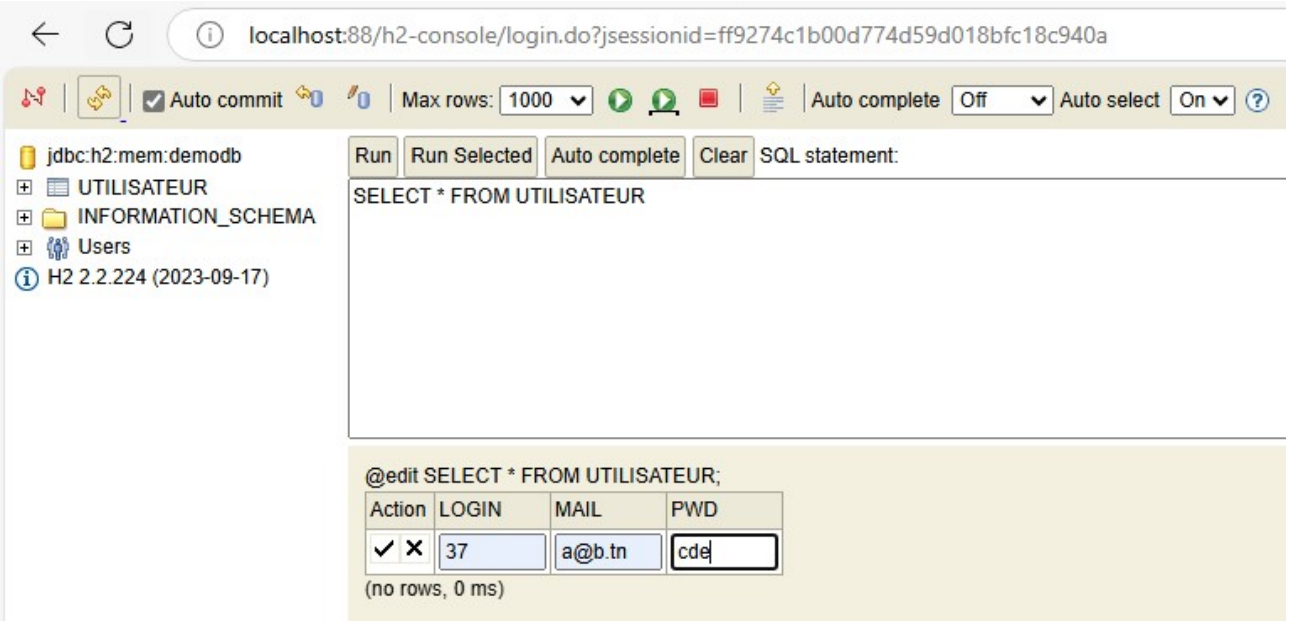
Concernant la console H2, une fois l'application démarrée, vous pouvez aller sur l'URL "http://localhost:88/h2-console". Une fenêtre de login s'ouvre, et il est nécessaire d'indiquer l'URL Jdbc.



Fenêtre de login de la console web H2

Le username par défaut est bien "sa", et le password par défaut est vide. Cliquer le bouton « Connect ».

Une fois connecté, vous pouvez consulter le contenu de votre table :



Affichage de la table Employees au sein de la console H2

Cependant, avec un comportement par défaut impliquant zéro configuration on doit pour activer la console de visualisation copier l'URL Jdbc qui change à chaque démarrage de l'application. (Dans votre console, vous aurez une ligne qui doit ressembler à la suivante : "H2 console availableat '/h2-console'. Database available at 'jdbc:h2:mem:b59feadd-5612-45fe-bd1c-3b62db66ea8a'".)

Récupérez l'URL JDBC (en l'occurrence jdbc:h2:mem:b59feadd-5612-45fe-bd1c-3b62db66ea8a'), puis le coller dans la zone de texte correspondantede la console.

4/ Création des classes

Au niveau de ce fournisseur de service, nous n'aurons pas de vues. Nous en aurons besoin au niveau du consommateur.

Les Models

Créer la classe Utilisateur avec les attributs login, pwd et mail, les trois de type String.

Ajouter à cette classe les annotations @Entity et @Table.

Ajouter aussi à l'attribut login l'annotation @Id

```
package isetjb.rest.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.persistence.Id;

@Entity
@Table
public class Utilisateur {

	@Id
	private String login;
	private String pwd;
	private String mail;

	public Utilisateur() {
	}

	public String getLogin() {
		return login;
	}

	public void setLogin(String login) {
		this.login = login;
	}

	public String getPwd() {
		return pwd;
	}

	public void setPwd(String pwd) {
		this.pwd = pwd;
	}

	public String getMail() {
		return mail;
	}
```

```
	public void setMail(String mail) {
		this.mail = mail;
	}

}
```

Les Repositories

Créer l'interface UtilisateurRepository qui étend l'interface JPARepository. Doter cette interface de l'annotation recommandée.

Les Contrôleurs

Dans le modèle MVC, un contrôleur gère les interactions entre l'utilisateur et le système. Dans le cadre d'une application Web, Les contrôleurs Spring Web MVC permettent de gérer les requêtes HTTP entrantes (émises par le navigateur client).

Dans une application monolithique, un contrôleur est une classe Java portant l'annotation @Controller. Pour que le contrôleur soit appelé lors du traitement d'une requête, il suffit d'ajouter l'annotation @RequestMapping au dessus de l'une des méthodes publiques de la classe en précisant la méthode HTTP concernée (par défaut GET) et le champ « Action » de l'URI (la portion de l'URI supplémentaire au nom du site) pris en charge par la méthode. Une telle méthode renvoie le plus souvent une chaine de caractères qui désigne le nom de la page web (la vue) à renvoyer à l'utilisateur.

Pour un service web de type REST, le contrôleur côté fournisseur porte l'annotation particulière @RestController et une méthode d'un tel contrôleur mappée à un type de requête http renvoie le plus souvent des données et pas une page web.

Remarques

- nom du site = nom du domaine ou du serveur, ou adresse IP du serveur, suivi éventuellement du numéro du port (par exemple localhost :88.
  Par exemple dns l'URI localhost :8080***/utilisateurs*** le nom du site est localhost :8080 et l'action est ***/utilisateurs***.
- @GetMapping est équivalente à @RequestMapping(method = RequestMethod.GET)

Créer dans le package adéquat le contrôleur suivant :

```
package isetjb.rest.demo.control;

import java.util.Optional;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.CrossOrigin;

import isetjb.rest.demo.model.Utilisateur;
import isetjb.rest.demo.dal.UtilisateurDAL;

@CrossOrigin(origins = "*")
@RestController
public class UtilisateurController {

	@Autowired
	UtilisateurDAL utilisateurDAO;

	public UtilisateurController() {
		// TODO Auto-generated constructor stub
	}

	@GetMapping("/utilisateurs")
	public Iterable<Utilisateur> getListeUtilisateurs() {
		return utilisateurDAO.findAll();
	}

	@GetMapping("/utilisateur/{login}")
	public Utilisateur getDetailsUtilisateur(@PathVariable("login") final String login) {
		Optional<Utilisateur> utlBox = utilisateurDAO.findById(login);
		if (utlBox.isPresent())
			return utlBox.get();
		else
			return null;
	}

	@PostMapping("/utilisateur")
	public Utilisateur insererUtilisateur(@RequestBody Utilisateur utilisateur) {
		return utilisateurDAO.save(utilisateur);
	}

	@PutMapping("/utilisateur")
	public Utilisateur mettreAJourUtilisateur(@RequestBody Utilisateur utilisateur) {
		Optional<Utilisateur> utlBox = utilisateurDAO.findById(utilisateur.getLogin());
		if(utlBox.isPresent()) {
			Utilisateur currentUtilisateur = utlBox.get();
			currentUtilisateur = utilisateur;
			utilisateurDAO.save(currentUtilisateur);
			return currentUtilisateur;
		} else {
			return null;
		}
	}

	@DeleteMapping("/utilisateur/{login}")
	public void supprimerUtilisateur(@PathVariable("login") final String login) {
		utilisateurDAO.deleteById(login);
	}

}
```

Quelle est la signification de l'annotation @RequestBody ? Quelle est alors la différence entre @PathVariable et @RequestBody ?

Rappeler la signification de l'annotation @Autowired. Quel est alors l'effet de cette déclaration ?

Notes

- Il existe les annotations @GetMapping, @PutMapping, @PostMapping, @DeleteMapping, @PatchMapping qui fonctionnent comme l'annotation @RequestMapping sauf qu'il n'est pas nécessaire de préciser la méthode HTTP concernée puisqu'elle est mentionnée dans le nom de l'annotation :
- L'annotation @RequestMapping peut être utilisée directement sur la déclaration de classe pour donner des informations applicables par défaut pour l'ensemble des méthodes de cette classe. Si on donne un chemin, alors ce dernier s'ajoute avant celui déclaré par une méthode.

****

Insérer à l'aide du h2-console des enregistrements dans la table Utilisateur.

Tester votre serveur via un navigateur. Corriger les éventuelles erreurs.

****