Prise en main du projet de gestion des TER de M1

L'objectif de cette séance est de prendre en main l'application qui nous servira de fil conducteur tout au long du semestre. Le volet fonctionnel de l'application a été vu en cours, nous allons aujourd'hui étudier la mise en oeuvre de l'application, dont une partie vous est fournie.

L'application est écrite en Java tant pour que le front-end (avec utilisation d'un moteur de templates) que pour le back-end (basé sur Spring boot).

- Première partie de la séance (1h30) : étude dirigée de l'existant. Ecoutez bien les explications fournies par votre enseignant.
- Deuxième partie de la séance (1h30) : en se basant sur l'existant, ajouter des fonctionnalités.

Récupération du projet et première exécution (20 minutes)

Vous trouverez sur moodle une archive avec le projet en cours de développement.

- Téléchargez cette archive pour l'enregistrer dans votre répertoire personnel (où vous le souhaitez, selon votre organisation habituelle de données)
- Extrayez les données de l'archive
- Ouvrez le répertoire obtenu dans un IDE Java, de préférence IntelliJ ultimate qui est installé sur les postes FDS (il vous faudra simplement une licence, voir document explicatif sur moodle).
- Pour compiler :
    - avec IntelliJ, se placer dans la vue Maven (View -> Tool windows -> Maven) :
        - déplier Lifecycles
        - double-cliquer sur Compile
    - Ligne de commande :
        - `mvn compile`
- Pour exécuter :
    - avec IntelliJ : run (flèche verte)
    - En ligne de commande : `mvn package` puis `java -jar xxx.jar` (où `xxx.jar` est le nom du jar construit)
- Accès au front-end : aller dans un navigateur à l'adresse `localhost:8080`

Le projet qui vous est fourni contient les éléments suivants :

- mécanisme de connexion et de déconnexion à l'application : le gestionnaire de TER, les enseignants et les étudiants peuvent se connecter.
- CRUD Enseignant
    - C (Create) : création d'un nouvel enseignant, seul le gestionnaire de TER peut réaliser cette action
    - R (Read) : accès en lecture aux enseignants
    - U (Update) : mise à jour des enseignants, seul le gestionnaire de TER peut réaliser cette action
    - D (Delete) : suppresion des enseignants, seul le gestionnaire de TER peut réaliser cette action
- CRUD Etudiant
    - C (Create) : création d'un nouvel étudiant, seul le gestionnaire de TER peut réaliser cette action

    - R (Read) : accès en lecture aux étudiants
    - U (Update) : mise à jour des étudiants, seul le gestionnaire de TER peut réaliser cette action
    - D (Delete) : suppression des étudiants, seul le gestionnaire de TER peut réaliser cette action

Organisation générale du projet (5 minutes)

A la racine vous trouverez les éléments suivants :

- répertoire `src` : contient les sources du projet
- un fichier `.gitignore` : pour l'instant ne sert à rien, mais sera utile quand le projet sera placé sur git. Ce fichier indique globalement ce qui ne doit pas être historicisé.
- deux fichiers `mvnw` et `mvnw.cmd`, dont vous n'avez pas à vous occuper. Ces fichiers proviennent du wrapper Maven. Ils permettent d'exécuter le projet Maven même si Maven n'est pas installé : il télécharge Maven si Maven n'est pas trouvé. Le fichier `mvnw` est pour Linux (bash) et le `mvnw.cmd` est pour Windows.
- un fichier `pom.xml` : fichier qui configure le comportement de maven, n'y touchez pas, nous l'étudierons la semaine prochaine.
- un fichier `system.properties`, utilisé notamment par maven

le répertoire src

Le répertoire `src` se compose de 2 répertoires :

- `main` : contient les sources du projet
- `test` : contient les tests des sources du projet

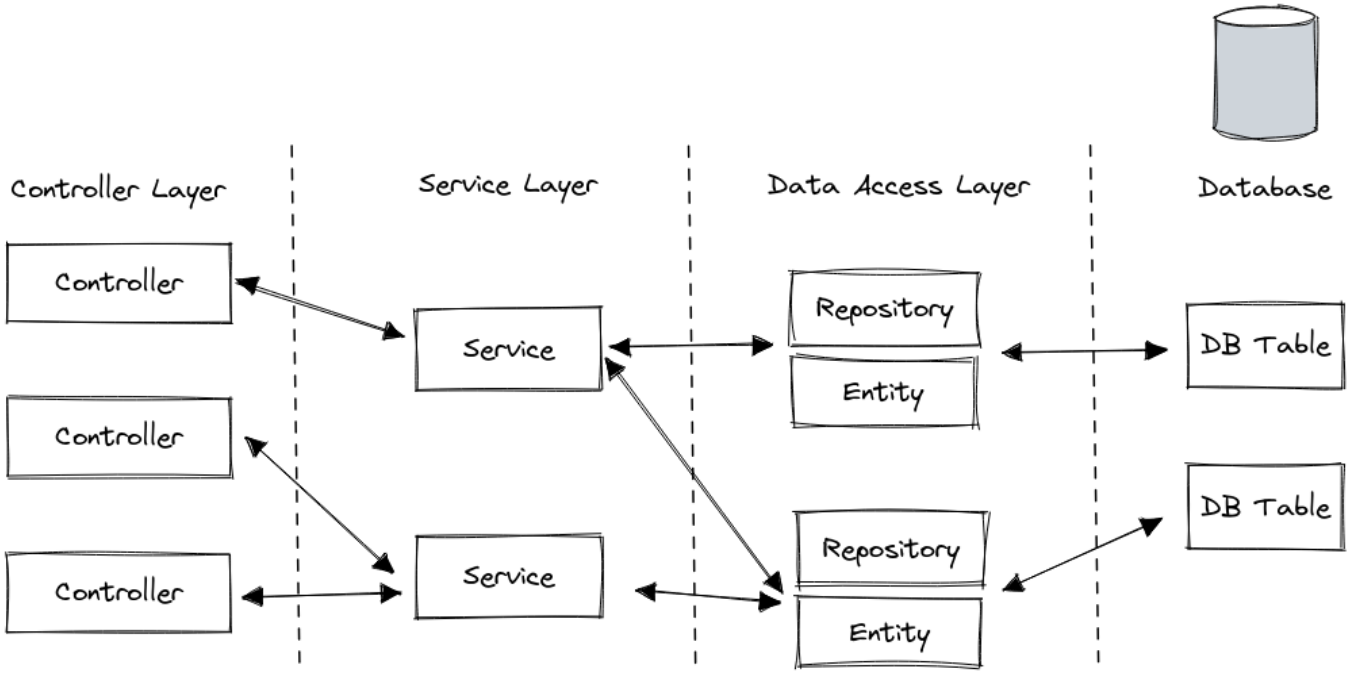
Cette séparation est classique. Ces deux répertoires devraient suivre la même architecture. Pour l'instant le répertoire `test` est assez vide.

le répertoire main

C'est le coeur du projet ! Il contient :

- un répertoire `java` qui contient les sources java, placées dans un package `um.fds.agl.ter23`
- un répertoire `resources` qui contient les éléments pour le front-end.

Différentes couches logicielles pour le back-end (5 minutes)



On met en place un back-end avec 4 couches logicielles.

- La couche base de donnée. C'est ici que seront les tables de la base de donnée. Nous utiliserons le SGBD H2, qui est écrit en Java, et pour lequel les tables peuvent être stockées en mémoire (vive) ou dans un fichier (sur disque dur). Nous choisissons de stocker les tables en mémoire, ce qui implique qu'elles ne seront pas persistantes : elles ne dureront que le temps de l'exécution de l'application. Ce n'est pas très réaliste mais cela nous simplifiera la vie, vous aurez suffisamment de choses à gérer.
- La couche d'accès aux données (data access layer). C'est la couche logicielle qui accède aux données stockées en base. Nous trouverons dans cette couche logicielle deux types d'éléments :
    - les entités (entities). Chaque entité est matérialisée par une classe Java, annotée avec l'annotation `@Entity`, et respectant un certain nombre de règles que nous verrons plus tard. Ces entités définissent les données à stocker, et donnent des indications sur comment elles doivent être stockées.
    - les dépôts (repositories). Chaque repository va gérer l'accès à la base de données pour un type d'entité, et met en place de quoi ajouter une entité, rechercher une entité à partir de son identifiant, supprimer une entité, modifier une entité, etc. Nous verrons que Spring nous facilite grandement le travail pour la mise en place des repositories, qui se matérialisent en général par des interfaces Java.
- La couche service (service layer) : elle définit les traitements métiers, et interroge donc la couche d'accès aux données. Dans notre application, nous avons assez peu de traitements métiers, l'application consistant majoritairement à accéder aux données de manière très simple. Nous aurions pu faire disparaître cette couche logicielle et la fusionner avec la couche controleur pour les quelques services à implémenter, toutefois pour vous habituer à la présence de cette couche logicielle et pour respecter les bonnes pratiques, cette couche est présente dans l'application. Les services se matérialisent par des classes Java.
- La couche controleur (controller layer). Les controleurs ont dans notre cas un double rôle. Ils prévoient les routes auxquelles le front-end va pouvoir s'adresser, et ils placent dans un modèle les différents éléments dont la vue va avoir besoin. Les controleurs interrogent les services.

Entity, repository, service et controller par l'exemple : zoom sur les enseignants (30 minutes)

Entités

Les gestionnaires de TER, les enseignants et les étudiants partagent des propriétés communes : ils ont un nom, un prénom, un mot de passe, et un identifiant. L'entité `UserTER` permet de factoriser ces propriétés communes. Elle se matérialise par la classe abstraite UserTER,

```
@Entity
@Inheritance
public abstract class UserTER {
    public static final PasswordEncoder PASSWORD_ENCODER = new
BCryptPasswordEncoder();

    private @Id    @GeneratedValue Long id;
    private String firstName;
    private String lastName;
    private @JsonIgnore String password;
    private @JsonIgnore String[] roles;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = PASSWORD_ENCODER.encode(password);
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

```
    public String[] getRoles() {
        return roles;
    }

    public void setRoles(String[] roles) {
        this.roles = roles;
    }

    public UserTER(){}
    public UserTER(String firstName, String lastName, String password,
String[] roles) {
        this.firstName = firstName;
        this.lastName = lastName;
        setPassword(password);
        this.roles = roles;
    }

    public UserTER(String firstName, String lastName) {
        //default : the password is the name, no role ...
        this(firstName, lastName, lastName, new String[0]);
    }

    public UserTER(long id, String firstName, String lastName) {
        //default : the password is the name, no role ...
        this(firstName, lastName, lastName, new String[0]);
        this.id = id;
    }



    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof UserTER)) return false;

        UserTER user = (UserTER) o;

        if (!getId().equals(user.getId())) return false;
        if (!getFirstName().equals(user.getFirstName())) return false;
        if (!getLastName().equals(user.getLastName())) return false;
        if (!password.equals(user.password)) return false;
        return Arrays.equals(roles, user.roles);
    }

    @Override
    public int hashCode() {
        int result = getId().hashCode();
        result = 31 * result + getFirstName().hashCode();
        result = 31 * result + getLastName().hashCode();
        result = 31 * result + password.hashCode();
        result = 31 * result + Arrays.hashCode(roles);
        return result;
    }
```

```
}
```

- L'annotation `@Inheritance` permet de donner la stratégie de traduction du modèle d'héritage en table. Ici on ne précise rien, toutes les entités filles seront donc stockées dans la même table.
- Dans l'attribut `PasswordEncoder PASSWORD_ENCODER` on a un encodeur pour crypter le mot de passe.
- Chaque entité est munie d'un identifiant, ici nommé `id`. L'attribut représentant l'identifiant est stigmatisé par l'annotation `@Id`. Dans notre cas, les identifiants sont générés automatiquement, on place donc l'annotation `@GeneratedValue`.
- Chaque utilisateur a un mot de passe, mais ce mot de passe ne doit pas circuler, notamment si d'une manière ou d'une autre on "envoie" un utilisateur vers la vue, le mot de passe ne doit pas être envoyé. C'est le rôle de l'annotation `@JsonIgnore`, puisque dans notre cas les données entre le front et le back seront encodées en json.
- L'attribut `roles` sert à stocker les rôles de l'utilisateur. On représente ici les rôles sous forme de tableaux de chaînes de caractères, ce qui n'est pas le plus élégant, mais c'est la solution la plus simple. Les rôles n'ont pas à être transmis (c'est inutile dans notre cas). Dans notre application, il y aura 3 rôles : celui de gestionnaire de TER, celui d'enseignant et celui d'étudiant.
- La suite du fichier contient des accesseurs en lecture et en écriture pour chacun des attributs. C'est une obligation pour une entité. On note que l'accesseur en écriture du mot de passe ne stocke pas le mot de passe en clair mais de manière cryptée.
- Nous trouvons ensuite des constructeurs. Il est impératif d'avoir au minimum un constructeur par défaut dans une entité. Ici plusieurs constructeurs paramétrés ont été ajoutés. On note que par défaut, le mot de passe est le nom.
- Une méthode equals et hashCode terminent notre classe. Elles sont obligatoires pour une entité. Elles ont ici été générées par IntelliJ de manière assez classique.

Regardons maintenant la classe `Teacher`.

```
@Entity
public class Teacher extends UserTER {

    private @ManyToOne TERManager terManager;



    public Teacher(){}
    public Teacher(String firstName, String lastName, String password,
TERManager manager, String... roles) {
        super(firstName, lastName, password, roles);
        this.terManager=manager;
    }

    public Teacher(String firstName, String lastName, TERManager manager) {
        super(firstName, lastName);
        String[] roles={"ROLE_TEACHER"};
        this.setRoles(roles);
        this.terManager=manager;
```

```
    }


    public TERManager getTerManager() {
        return terManager;
    }

    public void setTerManager(TERManager terManager) {
        this.terManager = terManager;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "id=" + getId() +
                ", firstName='" + getFirstName() + '\'' +
                ", lastName='" + getLastName() + '\'' +
                ", manager='"+ getTerManager() + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Teacher)) return false;
        if (!super.equals(o)) return false;

        Teacher teacher = (Teacher) o;

        return getTerManager() != null ?
getTerManager().equals(teacher.getTerManager()) : teacher.getTerManager()
== null;
    }

    @Override
    public int hashCode() {
        int result = super.hashCode();
        result = 31 * result + (getTerManager() != null ?
getTerManager().hashCode() : 0);
        return result;
    }
}
```

- Cette classe hérite de UserTER comme expliqué précédemment.
- On introduit un attribut référençant le gestionnaire de TER associé à l'enseignant. A vrai dire, celà n'est pas nécessaire, puisqu'il n'y a qu'un seul gestionnaire de TER. Nous l'avons placé ici comme s'il y avait plusieurs gestionnaires de TER afin d'illustrer certains points qui vous seront utiles pas la suite.
- On note l'annotation `@ManyToOne` qui sert pour le mapping vers le modèle relationnel de données. On indique ainsi que plusieurs enseignants peuvent avoir le même gestionnaire de TER.
- On trouve ensuite comme dans UserTER des constructeurs, et les méthodes equals et hashCode.

Repository

Pour mettre en place le dépôt pour les enseignants `TeacherRepository`, on va se reposer sur les dépôts fournis par Spring. On se base ici, en passant par l'extension de l'interface UserBaseRepository, sur le dépot `CrudRepository`.

```
@NoRepositoryBean
public interface UserBaseRepository<T extends UserTER>
        extends CrudRepository<T, Long> {

    public T findByLastName(String lastName);

}
```

L'annotation `@NoRepositoryBean` sert à indiquer à Spring de ne pas instancier ce dépôt, ce sont en effet les dépôts implémentant `UserBaseRepository` qui seront à implémenter.

Vous trouverez plus d'explications sur les repositories Spring ici.

Globalement, quand on écrit :

```
public interface UserBaseRepository<T extends UserTER>
        extends CrudRepository<T, Long>
```

on étend `CrudRepository` qui est une interface marqueur, qui sert à indiquer le type avec lequel le repository travaille (ici `UserTER`), et à définir les opérations qui seront présentes dans le repository. On note au passage l'usage de la généricité contrainte, que vous avez vue en HAI401I.

Un `CrudRepository` fournit des fonctionnalités CRUD, comme indiqué dans l'extrait ci dessous :

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    <S extends T> S save(S entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
    // ...
}
```

On y trouve des fonctionnalités CRUD (save, delete, findAll par exemple) mais aussi utilitaires comme le count et le exists.

On note dans `UserBaseRepository` la présence de la méthode `findByLastName(String lastName)`. Cette méthode sera présente pour notre repository, bien que nous n'en fournissions pas d'implémentation,

et que cette méthode ne soit pas présente dans `CrudRepository`. En effet, Spring permet de construire "gratuitement" des méthodes d'accès, pour peu que l'on respecte des conventions de nommage. Ici, le nom de la méthode commence par `findBy`, et est suivi par le nom d'un attribut de `UserBase`. `findByLastName(String lastName)` permettra donc d'obtenir l'utilisateur portant le nom passé en paramètre. Vous trouverez plus d'explications sur la construction de ces noms de méthodes ici.

Le repository expose également les méthodes de `CrudRepository` sur des routes codifiées par des conventions de nommage. Nous ne nous servirons pas ici de ces routes, qui pourraient être désactivées.

Regardons à présent `TeacherRepository`.

```
public interface TeacherRepository extends UserBaseRepository<Teacher> {
    @Override
    @PreAuthorize("hasRole('ROLE_MANAGER') and (#teacher?.terManager ==
null or #teacher?.terManager?.lastName == authentication?.name)")
    Teacher save(@Param("teacher") Teacher teacher);

    @Override

@PreAuthorize("@teacherRepository.findById(#id).get()?.terManager?.lastName
== authentication?.name")
    void deleteById(@Param("id") Long id);

    @Override
    @PreAuthorize("#teacher?.terManager?.lastName == authentication?.name")
    void delete(@Param("teacher") Teacher teacher);
}
```

Certaines méthodes CRUD sont ici "redéfinies". Celà est fait pour en altérer le comportement via des annotations permettant de faire notamment des contrôles de droits d'accès.

Prenons par exemple la méthode `save`. Elle porte l'annotation :

```
@PreAuthorize("hasRole('ROLE_MANAGER') and (#teacher?.terManager == null or
#teacher?.terManager?.lastName == authentication?.name)")
```

Cette annotation indique qu'un pré-contrôle de droits sera réalisé, en suivant la règle donnée en paramètre de l'annotation. La règle qui doit être satisfaite se lit comme suit :

- la requête doit avoir été émise par un utilisateur ayant le rôle `ROLE_MANAGER`
- ET :
    - le gestionnaire de TER de l'enseignant en paramètre (`#teacher?.terManager`) doit être nul
    - OU alors sinon le gestionnaire de TER de l'enseignant en paramètre doit avoir pour nom de famille le nom stocké dans le jeton d'authentification. Il se cache beaucoup de choses derrière cette annotation, nous n'expliciterons que certains points.
- Entre les guillemets, se place une expression écrite dans un dialecte nommé SpEL, fourni par Spring (plus d'informations ici).

- Nous utilisons ici le fait que les utilisateurs sont authentifiés pour utiliser l'application, et qu'il y a donc un jeton d'authentification. Nous reviendrons brièvement plus tard sur cette authentification.

Suite à la définition de nos interfaces de dépôt, nous disposons d'une couche d'accès aux données, sans aucune implémentation de ces accès, juste en paramétrant des accès pré-conçus dans Spring.

Service

```
@Service
public class TeacherService {

    @Autowired
    private TeacherRepository teacherRepository;

    public Optional<Teacher> getTeacher(final Long id) {
        return teacherRepository.findById(id);
    }

    public Iterable<Teacher> getTeachers() {
        return teacherRepository.findAll();
    }

    public void deleteTeacher(final Long id) {
        teacherRepository.deleteById(id);
    }

    public Teacher saveTeacher(Teacher teacher) {
        Teacher savedTeacher = teacherRepository.save(teacher);
        return savedTeacher;
    }

    public Optional<Teacher> findById(long id) {
        return teacherRepository.findById(id);
    }

}
```

La classe `TeacherService` porte l'annotation `@Service` : c'est un service. Ce service va utiliser le dépôt d'enseignants `teacherRepository`. Cet attribut porte l'annotation `@Autowired` : il n'y a pas à initialiser cet attribut, c'est Spring qui se chargera d'injecter sa valeur. Le mécanisme sous-jacent est l'injection de dépendance, vous le verrez plus finement quand vous serez plus grands, pour l'instant on va se servir de cette annotation comme d'une baguette magique. Attention à ne pas utiliser cette baguette magique à tort et à travers ! Elle ne servira que pour les composants gérés par Spring, dans notre cas les services et les repositories.

Comme nous l'avons déjà expliqué, la classe service n'est ici pas vraiment utile : elle se contente de faire des requêtes au repository, sans valeur métier ajoutée.

Nous notons au passage l'utilisation d'Optional déjà vue en HAI401I.

Controller

La classe `TeacherController` utilise le `ServiceTeacher`, définit les primitives qui seront utiles à la vue, et se charge d'alimenter le modèle qui sera utile à la vue.

```
@Controller
public class TeacherController {

    @Autowired
    private TeacherService teacherService;
    @Autowired
    private TERManagerService terManagerService;

    @GetMapping("/listTeachers")
    public Iterable<Teacher> getTeachers(Model model) {
        model.addAttribute("teachers", teacherService.getTeachers());
        return teacherService.getTeachers();
    }
    @PreAuthorize("hasRole('ROLE_MANAGER')")
    @GetMapping(value = { "/addTeacher" })
    public String showAddTeacherPage(Model model) {

        TeacherForm teacherForm = new TeacherForm();
        model.addAttribute("teacherForm", teacherForm);

        return "addTeacher";
    }

    @PostMapping(value = { "/addTeacher"})
    public String addTeacher(Model model, @ModelAttribute("TeacherForm")
TeacherForm teacherForm) {
        Teacher t;
        if(teacherService.findById(teacherForm.getId()).isPresent()){
            // teacher already existing : update
            t = teacherService.findById(teacherForm.getId()).get();
            t.setFirstName(teacherForm.getFirstName());
            t.setLastName(teacherForm.getLastName());
        } else {
            // teacher not existing : create
            t=new Teacher(teacherForm.getFirstName(),
teacherForm.getLastName(), terManagerService.getTERManager());
        }
        teacherService.saveTeacher(t);
        return "redirect:/listTeachers";

    }

    @GetMapping(value = {"/showTeacherUpdateForm/{id}"})
    public String showTeacherUpdateForm(Model model, @PathVariable(value =
"id") long id){

        TeacherForm teacherForm = new TeacherForm(id,
```

```
teacherService.findById(id).get().getFirstName(),
teacherService.findById(id).get().getLastName());
        model.addAttribute("teacherForm", teacherForm);
        return "updateTeacher";
    }

    @GetMapping(value = {"/deleteTeacher/{id}"})
    public String deleteTeacher(Model model, @PathVariable(value = "id")
long id){
        teacherService.deleteTeacher(id);
        return "redirect:/listTeachers";
    }

}
```

- Les services nécessaires sont auto-injectés par Spring.
- La méthode `getTeachers` sera appelée lors d'un `GET` sur la route `/listTeachers`. Elle prend un `Model` en paramètre et retourne des `Teacher` (sous forme d'un `Iterable<Teacher>`). Les enseignants, obtenus depuis le service, sont placés dans le modèle avec l'étiquette `teachers`.

```
 @PreAuthorize("hasRole('ROLE_MANAGER')")
    @GetMapping(value = { "/addTeacher" })
    public String showAddTeacherPage(Model model) {

        TeacherForm teacherForm = new TeacherForm();
        model.addAttribute("teacherForm", teacherForm);

        return "addTeacher";
    }
```

- La méthode `showAddTeacherPage` sera appelée lorsque l'on souhaitera depuis la vue ajouter un enseignant.
    - Elle contrôle que l'utilisateur est un gestionnaire de TER
    - Elle est associée à un `GET` sur la route `addTeacher`
    - Elle crée une instance de TeacherForm (qui représente les données qui seront à échanger lors de la création d'un enseignant : son prénom et son nom).
    - Elle ajoute cette instance au modèle
    - Elle retourne la chaîne `addTeacher`, qui correspond à la page à afficher.

```
   @PostMapping(value = { "/addTeacher"})
    public String addTeacher(Model model, @ModelAttribute("TeacherForm")
TeacherForm teacherForm) {
        Teacher t;
        if(teacherService.findById(teacherForm.getId()).isPresent()){
            // teacher already existing : update
            t = teacherService.findById(teacherForm.getId()).get();
            t.setFirstName(teacherForm.getFirstName());
```

```
            t.setLastName(teacherForm.getLastName());
        } else {
            // teacher not existing : create
            t=new Teacher(teacherForm.getFirstName(),
teacherForm.getLastName(), terManagerService.getTERManager());
        }
        teacherService.saveTeacher(t);
        return "redirect:/listTeachers";

    }
```

- La méthode `addTeacher` sera appelée quand les données du nouvel enseignant auront été saisies.
    - Elle est associée à un `POST` sur la route `addTeacher`.
    - Elle prend en paramètre un `Model`, et un `TeacherForm` qui est issu du `Model` (du fait de l'annotation `@ModelAttribute`)
    - Elle se sépare en 2 cas
        - cas où il s'agit d'un enseignant existant (mise à jour) : dans ce cas on obtient l'enseignant (via le service), on récupère de la TeacherForm les nom et prénom mis à jour, et on les place dans l'enseignant.
        - cas où il s'agit d'un nouvel enseignant (création) : dans ce cas on crée un nouvel enseignant en utilisant les données de la `TeacherForm`
    - Dans les 2 cas, l'enseignant est ensuite sauvegardé (via le service) et on retourne une instruction de redirection vers la page permettant de lister les enseignants.

```
 @GetMapping(value = {"/showTeacherUpdateForm/{id}"})
    public String showTeacherUpdateForm(Model model, @PathVariable(value =
"id") long id){

        TeacherForm teacherForm = new TeacherForm(id,
teacherService.findById(id).get().getFirstName(),
teacherService.findById(id).get().getLastName());
        model.addAttribute("teacherForm", teacherForm);
        return "updateTeacher";
    }
```

- La méthode `showTeacherUpdateForm` sera appelée pour mettre à jour l'enseignant.
    - Elle est associée à un GET sur la route `/showTeacherUpdateForm/{id}`, où `{id}` est une variable de la route, qui est associé le paramètre id de la méthode. En d'autres termes, si la route est `/showTeacherUpdateForm/42`, alors le paramètre `id` vaudra 42.

HomeController

```
@Controller
public class HomeController {
    @RequestMapping(value = "/") // support of the "/" route
    public String index() {
```

```
        return "index"; // index is the name of the template, the view
resolver will map it into src/main/resources/templates/index.html.
    }

}
```

Ce controleur définit la page index de l'application (ici index.html).

Autres fichiers java présents dans le projet (5 minutes)

- `Ter22Application` : la classe principale de l'application

```
@SpringBootApplication
public class Ter22Application {

    public static void main(String[] args) {
        SpringApplication.run(Ter22Application.class, args);
    }

}
```

- SecurityConfiguration

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private SpringDataJpaUserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth
                .userDetailsService(this.userDetailsService)
                .passwordEncoder(UserTER.PASSWORD_ENCODER);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .antMatchers("/built/**", "/main.css").permitAll()
                .anyRequest().authenticated()
                .and()
                .formLogin()
                .defaultSuccessUrl("/", true)
```

```
                .permitAll()
                .and()
                .httpBasic()
                .and()
                .csrf().disable()
                .logout()
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login");
    }

}
```

Nous ne rentrerons pas dans les détails de cette classe, elle paramètre l'authentification. Elle permet notamment d'indiquer le besoin d'authentification pour chaque requête, l'url accédée en cas de login/logout correct, etc. Plus d'information peut être trouvée ici.

- `DataBaseLoader` peuple la base de données avec quelques éléments (jetez-y un oeil, elle contient en dur des données peuplant l'application au démarrage).

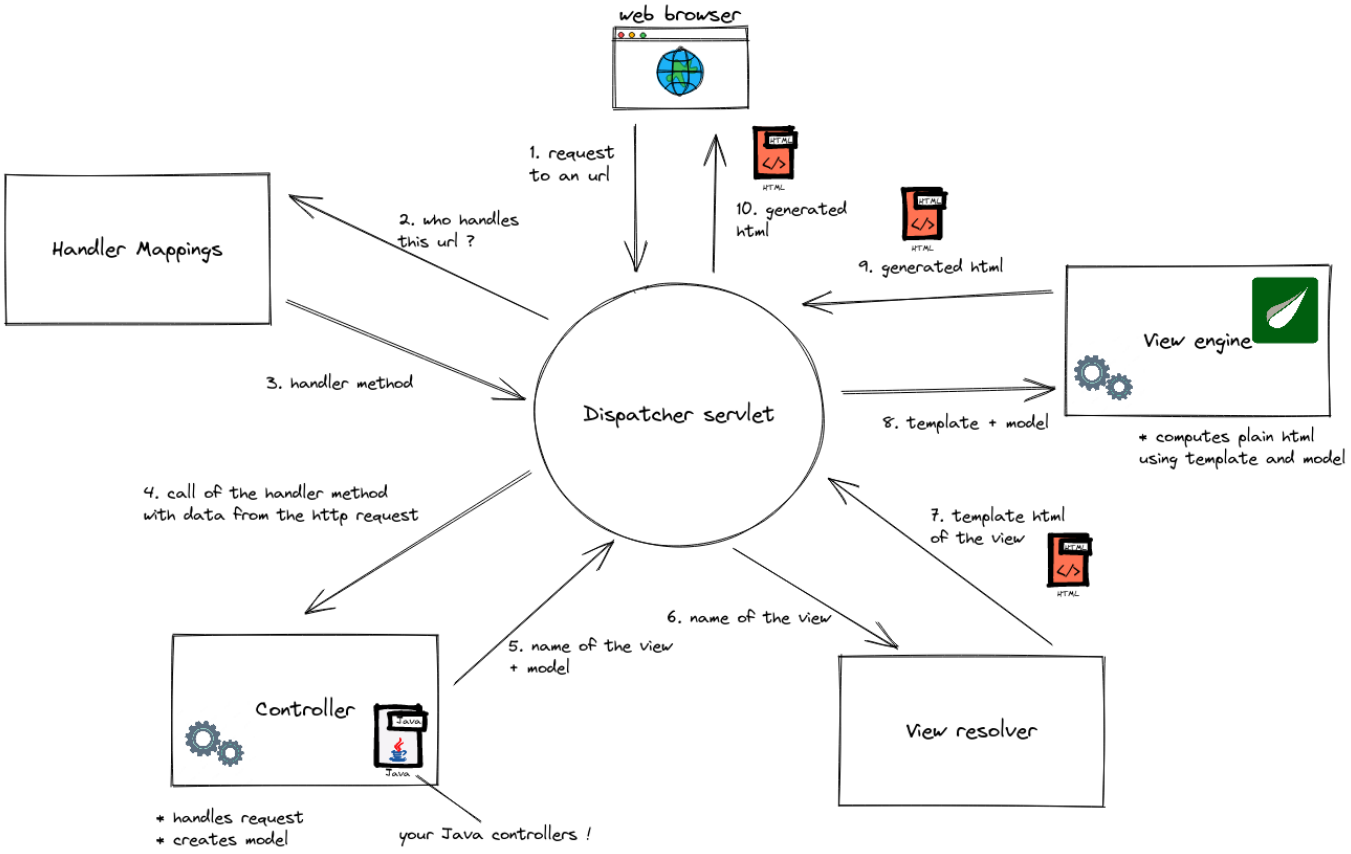
Front-end avec Spring MVC et ThymeLeaf (10 minutes)

Nous utilisons le moteur de template `ThymeLeaf`. Le principe général est le suivant :

- On développe des templates html.
- Ces templates html sont des fichiers html classiques, augmentés d'éléments thymeleaf utilisant les données du `model`.
- ThymeLeaf utilise le template et un `model` pour générer de l'html pur.

Plus précisément, dans une application MVC Spring, les classes contrôleur préparent un modèle (un genre de dictionnaire avec des données indexées par des clefs/des étiquettes) et sélectionnent la vue à retourner. Ce modèle découple le contrôleur de la vue. Concrêtement, il permet ensuite au moteur de vue (pour nous ThymeLeaf) de générer l'html à partir du template et des données du model. Le model est transformé en objets de contexte Thymeleaf et en fait des variables disponibles pour écrire les expressions ThymeLeaf des templates.

Le schéma ci-dessous résume les principales étapes mises en oeuvre depuis la requête http jusqu'à la réponse html.



Regardons un fichier de template, par exemple `listTeachers.html`.

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Teacher List</title>
    <link rel="stylesheet" href="/main.css" />
</head>
<body>
<h1>Teacher List</h1>
<a href="/">Accueil</a>
<a href="addTeacher">Add Teachers</a>
<br/><br/>
<div>
    <table border="1">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
        </tr>
        <tr th:each ="teacher : ${teachers}">
            <td th:utext="${teacher.firstName}">...</td>
            <td th:utext="${teacher.lastName}">...</td>
            <td><a th:href="@{/showTeacherUpdateForm/{id}
(id=${teacher.id})}">Update</a>
                <a th:href="@{/deleteTeacher/{id}
(id=${teacher.id})}">Delete</a></td>
        </tr>
    </table>
```

```
</div>
</body>
</html>
```

- A l'exception des éléments `th`, c'est un fichier html normal, qui peut être ouvert normalement par un navigateur.
- En haut du fichier on trouve un lien vers `addTeacher`, qui est un autre template (à utiliser pour ajouter un nouvel enseignant) et un lien vers l'index (`/`) our revenir facilement à l'accueil.
- Ensuite on trouve un tableau html simple.
- Suite à l'entête du tableau, on trouve une table row (`<tr>`) et c'est là que les expressions thymeleaf apparaissent.

```
<tr th:each ="teacher : ${teachers}">
```

Ici, `${teachers}`sélectionne la variable appelée `teachers` dans le contexte (issu du `model`), et l'évalue comme un itérable, pouvant donc être utilisé par une boucle for `th:each`. Cette boule foreach a une syntaxe proche de celle de Java, on peut lire notre boucle ainsi : pour chaque élément trouvé dans `${teachers}` et que j'appelle `teacher`, faire ... Ainsi, lors de l'exécution du moteur, on aura une ligne par enseignant présent dans `teachers`.

```
<td th:utext="${teacher.firstName}">...</td>
```

- La première cellule de la ligne doit contenir le prénom de l'enseignant : `${teacher.firstName}`. `teacher` est ici notre variable de boucle foreach.
- utext signifie unescapedText : les tags html ne doivent pas être "échappés" (ici cela ne sert à rien).
- La seconde cellule de la ligne suit le même schéma
- La troisème cellule de la ligne contient 2 liens, étudions le premier.

```
<a th:href="@{/showTeacherUpdateForm/{id}(id=${teacher.id})}">Update</a>
```

- Le label du lien est update.
- On utilise une Link URL Expression (expression de lien URL) dénotée par le `@`.
- l'URL est : /showTeacherUpdateForm/{id}(id=${teacher.id})
    - l'url est de la forme `/showTeacherUpdateForm/{id}`
    - avec `id` qui vaut `${teacher.id}`
- En d'autres termes, quand on clique sur le lien update, on enverra la requête `/showTeacherUpdateForm/x` où `x` sera l'id de l'enseignant à mettre à jour (celui qui est affiché sur la même ligne)

Travail à réaliser

Maintenant que vous avez tout compris, vous pouvez commencer à travailler ! Et dans le cas probable où tout celà reste encore un peu nébuleux, vous allez calquer ce que nous avons vu pour mieux le comprendre.

- Mettez en oeuvre, côté back et côté front, le CRUD pour les sujets de TER. Un sujet de TER se compose pour l'instant de l'enseignant ayant posé le sujet, d'un titre pour le sujet
    - Create : les enseignants et le gestionnaire de TER peuvent créer des sujets. Pour simplifier, dans les deux cas, on fournira le nom de l'enseignant responsable.
    - Read : permet de lister tous les sujets, de même que ce qui a été proposé pour les enseignants ou les étudiants.
    - Update : chaque sujet pourra être mis à jour, soit par le gestionnaire de TER, soit par l'enseignant ayant posé le sujet.
    - Delete : chaque sujet pourra être supprimé, soit par le gestionnaire de TER, soit par l'enseignant ayant posé le sujet.
- liens utiles :
    - dialecte ThymeLeaf en 5 minutes
    - Pour mettre en place une relation bi-directionnelle entre Teacher et l'entité des sujets de TER