TP - Conception d'un Contrôleur Spring MVC avec Spring Boot

Table of Contents

Objectifs

- Savoir créer un projet Spring Boot minimal sous IntelliJ avec Spring Initializr.
- Comprendre le rôle et la conception d'un contrôleur Spring MVC.
- Manipuler le mécanisme de transmission de données à la vue via l'objet `Model`.
- Savoir faire la différence entre un `redirect`, un `forward` et un retour classique dans un contrôleur.

Prérequis

- Java 17+
- IntelliJ IDEA avec plugin Spring Boot (Ultimate)
- Maven

1. Préparation du projet Spring Boot avec Spring Initializr

Étapes

1. Ouvrir IntelliJ IDEA et créer un nouveau projet Spring Boot à l'aide de Spring Initializr.
2. Choisir :
   - Project SDK : Java 17+
   - Maven (et non pas Gradle)
   - Group : `fr.info.orleans.pnt.springmvc`
   - Packaging : jar
   - Dependencies : `Spring Web`, `Thymeleaf`
3. Nom du projet : `tp-springmvc-evenements`
4. Une fois le projet généré, vérifier la structure suivante :

```
src/
├──── main/
│     ├──── java/
│     │     └──── fr.info.orleans.pnt.springmvc.tpspringmvcevenements/
│     │           └──── TpSpringmvcEvenementsApplication.java
│     └──── resources/
│           ├──── templates/
│           └──── application.properties
```

5. Démarrez l'application (classe `TpSpringmvcEvenementsApplication`) pour s'assurer que la configuration fonctionne (accès à http://localhost:8080). Vous devriez avoir une page d'erreur `Whitelabel Error Page` ce qui est normal car aucune route n'est définie.

2. Enregistrement d'Événements – Contrôleur Spring MVC

2.1. Contexte métier

On souhaite créer une petite application permettant d'enregistrer des événements.

Voici la classe métier de base :

```
public class Evenement {
    private final int identifiant;
    private String nom;
    private String lieu;
```

```
    private String date;

    public Evenement(int identifiant, String nom, String lieu, String date) {
        this.nom = nom;
        this.lieu = lieu;
        this.date = date;
        this.identifiant = identifiant;
    }
    // getters et setters
}
```

Et une façade métier, initialement simple, stockant les événements en mémoire :

```
@Component
public class GestionEvenements {
    private List<Evenement> evenements = new ArrayList<>();
    private int compteur = 0;
    public int enregistrer(String nom, String lieu, String date) {
        Evenement e = new Evenement(compteur,nom, lieu, date);
        compteur++;
        evenements.add(e);
        return e.getIdentifiant();
    }
    public Evenement trouverParId(String id) throws EvenementInconnuException {
        return evenements.stream()
                .filter(e -> e.getIdentifiant().equals(id))
                .findFirst()
                .orElseThrow(() -> new EvenementInconnuException("Événement inconnu :
" +
                        id));
    }
    public List<Evenement> lister() {
        return evenements;
    }
}
```

Ajoutez ces deux classes ainsi que la classe (`extends Exception`) `EvenementInconnuException` dans un package `modele` qui doit être créé au même niveau que la classe principale `TpSpringmvcEvenementsApplication`.

Un objet métier ne doit pas être créé en dehors du périmètre métier (façade et classes métiers).

- A quoi sert à votre avis l'annotation `@Component` ?

2.2. Premier contrôleur : paramètres primitifs

Créez un contrôleur `EvenementControleur` dans le package `controleur` (au même niveau que `modele`).

```java
@Controller
@RequestMapping("/mesevenements")
public class EvenementControleur {
    private final GestionEvenements gestion;
    public EvenementControleur(GestionEvenements gestion) {
        this.gestion = gestion;
    }
    @GetMapping("/ajouter")
    public String ajouter(
            @RequestParam String nom,
            @RequestParam String lieu,
            @RequestParam String date,
            Model model
    ) {
        int identifiant = gestion.enregistrer(nom, lieu, date);
        model.addAttribute("message", "Événement ajouté avec succès !");
        model.addAttribute("identifiant", identifiant);
        return "resume";
    }
}
```

Créez la vue `resume.html` avec un simple message dans `src/main/resources/templates` :

```html
<html xmlns:th="http://www.thymeleaf.org">
<body>
<h2 th:text="${message}"></h2>
L'identifiant de l'évènement créé est <span th:text="${identifiant}"></span>.

</body>
</html>
```

Testez ce code en lançant l'application et en accédant à l'URL :

```
http://localhost:8080/mesevenements/ajouter?nom=Festival&lieu=Marseille&date=2025-08-01
```

- A votre avis à quoi sert l'annotation `@RequestParam` ?
- A quoi sert l'objet `Model` passé en paramètre de la méthode `ajouter()` d'où vient-il ?
- A quoi sert la chaine de caractères "resume" retournée par la méthode `ajouter()` ?
- Dans la vue, comment est récupéré le message passé par le contrôleur ?
- Relancez une seconde requête avec d'autres paramètres, que concluez vous sur la facade `GestionEvenements` ?

  ```
  http://localhost:8080/mesevenements/ajouter?nom=Noel&lieu=Groenland&date=2025-12-25
  ```

- Plus important, comment est créé l'instance de `GestionEvenements` dans le contrôleur ? D'où vient-elle ?
- Que se passe t'il si on supprime l'annotation `@Component` dans la classe `GestionEvenements` ?
- Remettez le code en état après vos tests.

2.3. Amélioration : encapsulation des paramètres

Créez un package `dto` au même niveau que `modele` et créez-y une classe `EvenementDTO` : La classe `EvenementDTO` représentera une copie d'un objet de la classe `Evenement`. Un *data transfert object* (DTO) est une classe qui sert à transporter des données entre différentes couches d'une application, souvent entre la couche de présentation et la couche métier. Un tel objet peut être créé dans le contrôleur pour encapsuler les paramètres de l'événement.

Modifier la façade pour qu'elle accepte directement un objet `EvenementDTO` (c'est l'étape 2 du raffinement).

```
public int enregistrer(EvenementDTO e) {
        Evenement evenement = new Evenement(compteur,e.getNom(), e.getLieu(), e
.getDate());
        compteur++;
        e.setIdentifiant(evenement.getIdentifiant());
        evenements.add(evenement);
        return evenement.getIdentifiant();
    }
```

Le contrôleur peut alors faire dans la fonction `ajouter()` :

```
EvenementDTO e = new EvenementDTO();
e.setNom(nom);
e.setLieu(lieu);
e.setDate(date);
gestion.enregistrer(e);
```

- Quelle différence y a t'il entre un EvenementDTO et un Evenement au delà du fait qu'ils n'ont pas le même périmètre dans lequel ils ont le droit de transiter ?
- Que se passe t'il après l'appel à `gestion.enregistrer(e)` du point de vue de `e` ?

Mais cela implique de gérer la création de l'objet dans le contrôleur. **On peut faire mieux.**

2.4. Variante avancée : `@ModelAttribute`

Voici une autre approche, plus idiomatique en Spring MVC :

```
    @GetMapping("/ajouter2")
    public String ajouterViaModel(@ModelAttribute EvenementDTO evenement, Model model)
```

```
{
        int identifiant = gestion.enregistrer(evenement);
        model.addAttribute("message", "Événement ajouté avec succès !");
        model.addAttribute("identifiant", identifiant);
        return "resume";
    }
```

Dans ce cas, Spring instancie un `EvenementDTO` et injecte ses propriétés depuis les paramètres de la requête.

Tester avec :

```
http://localhost:8080/mesevenements/ajouter2?nom=Festival&lieu=Marseille&date=2025-08-01
```

- Que comprenez vous de l'annotation `@ModelAttribute` associée à `EvenementDTO` ?
- Maintenant, dans `EvenementDTO`, supprimez le constructeur sans paramètre, générez un constructeur avec tous les paramètres nécessaires et relancez avec la même URL que précédemment. Une erreur doit être déclenchée. Observez la trace pour analyser le problème. Quelle est la source du problème ?

2.5. Bilan

Intérêt progressif des trois approches :

Approche	Avantage	Limite
Paramètres séparés + création dans la façade	Encapsulation métier respectée	Moins flexible pour la validation
Paramètres séparés + création d'un DTO dans le contrôleur	Flexible pour tests, logique conditionnelle	Brise l'encapsulation métier
`@ModelAttribute` avec passage d'objet DTO	Plus idiomatique Spring, prépare la validation	Nécessite des constructeurs, binding propre

3. Consultation d'un Événement par son identifiant – Utilisation de @PathVariable

3.1. Objectif

Apprendre à récupérer une ressource en fonction d'un identifiant transmis dans l'URL à l'aide de `@PathVariable`.

3.2. Nouvelle méthode dans le contrôleur

Ajoutez une méthode permettant d'afficher un événement à partir de son identifiant dans l'URL :

```
@GetMapping("/evenement/{id}")
public String afficherUn(@PathVariable int id, Model model) {
    try {
        Evenement evenement = gestion.trouverParId(id);
        EvenementDTO evenementDTO = new EvenementDTO();
        evenementDTO.setIdentifiant(evenement.getIdentifiant());
        evenementDTO.setNom(evenement.getNom());
        evenementDTO.setLieu(evenement.getLieu());
        evenementDTO.setDate(evenement.getDate());
        model.addAttribute("evenement", evenementDTO);
        model.addAttribute("id", id);
        return "detail";
    } catch (EvenementInconnuException e) {
        model.addAttribute("erreur", "Événement introuvable.");
        return "erreur";
    }
}
```

3.3. Vue de détail

Créez une page `detail.html` dans `templates`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Détail</title></head>
<body>
<h1>Détail de l'événement n°<span th:text="${id}"></span></h1>
<p><strong>Identifiant :</strong> <span th:text="${evenement.identifiant}"></span></p>
<p><strong>Nom :</strong> <span th:text="${evenement.nom}"></span></p>
<p><strong>Lieu :</strong> <span th:text="${evenement.lieu}"></span></p>
<p><strong>Date :</strong> <span th:text="${evenement.date}"></span></p>
<a th:href="@{/mesevenements/evenements}">Retour à la liste</a>
</body>
</html>
```

Créez aussi une page simple `erreur.html` pour les cas où l'id est invalide :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Erreur</title></head>
<body>
    <h2 th:text="${erreur}"></h2>
    <a th:href="@{/mesevenements/evenements}">Retour à la liste</a>
```

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

Testez avec l'URL suivante :

```
http://localhost:8080/mesevenements/evenement/0
```

- Que se passe-t-il ?
- Créez une ressource avec l'URL ci-dessous :

  ```
  http://localhost:8080/mesevenements/ajouter2?nom=Festival&lieu=Marseille&date=2025-08-01
  ```

- Testez de nouveau l'URL :

  ```
  http://localhost:8080/mesevenements/evenement/0
  ```

- Qu'avez vous compris du rôle de la PathVariable ?
- En regardant de plus près le code de la vue `detail.html`, que remarquez-vous sur la manière dont les données sont injectées dans la vue ?
- Pour l'instant le retour à la liste n'existe pas et n'est donc pas fonctionnel.

3.4. Intégration d'une vue de la liste des évènements

Créez une page `liste.html` dans `templates` avec le code suivant :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>Liste</title></head>
<body>
    <h1>Liste des événements</h1>
    <ul>
        <li th:each="e : ${evenements}">
            <span th:text="${e.nom}">Nom</span> à
            <span th:text="${e.lieu}">Lieu</span> le
            <span th:text="${e.date}">Date</span>
        </li>
    </ul>
</body>
</html>
```

Ajoutez une méthode pour afficher tous les événements :

```
@GetMapping("/evenements")
public String lister(Model model) {
    model.addAttribute("evenements", gestion.lister());
    return "liste";
}
```

- Relancez l'application et accédez aux URL ci-dessous :
  - http://localhost:8080/mesevenements/ajouter?nom=Festival&lieu=Marseille&date=2025-08-01
  - http://localhost:8080/mesevenements/ajouter?nom=Noel&lieu=Groenland&date=2025-12-25
  - http://localhost:8080/mesevenements/evenements
- A quoi sert le `th:each` dans la vue `liste.html` ? Expliquez le rôle de `e` et de `${evenements}`.
- Analysez la phase de création des liens pour chaque évènement dans `liste.html` en particulier la façon dont sont créées les URL. Jetez un oeil à `detail.html` et `erreur.html` pour voir comment les liens étaient construits.

4. Retour de vue, redirect et forward

4.1. Objectif

Modifier le contrôleur précédent pour expérimenter :

- Retour classique (`return "vue";`)
- Redirection (`return "redirect:/chemin";`)
- Forward (`return "forward:/chemin";`)

4.2. Variante du contrôleur

Changer le retour de `ajouter()` :

```
// Avec redirection vers la liste
return "redirect:/mesevenements/evenements";

// Ou avec forward
return "forward:/mesevenements/evenements";
```

- Normalement, la liste ne s'affiche plus dans la vue après un redirect, mais elle s'affiche après un forward. Pourquoi ?
- Comment corriger le problème en préservant `redirect` ? Se renseigner sur l'utilisation d'un `RedirectAttributes`. Il vous faudra modifier la méthode `ajouter()` pour utiliser `RedirectAttributes` et y ajouter un message de succès.

4.3. Comparaison

Type de retour	Effet
`return "vue"`	Appelle directement la vue correspondant à `vue.html`.
`return "redirect:/chemin"`	Redirige le navigateur vers `/chemin`. Cela implique une nouvelle requête HTTP.
`return "forward:/chemin"`	Transfert en interne la requête à un autre handler, sans changer l'URL dans le navigateur.

5. Conclusion

Ce TP vous a permis d'expérimenter la conception de contrôleurs Spring MVC avec injection de composants métier et transmission de données à la vue. Vous avez également exploré les différentes stratégies de retour dans un contrôleur.

Ce qu'il faut retenir aussi dans ce TP, c'est que la création de ressources comme `Evenement` ne devrait pas se faire dans une fonction du contrôleur annotée par `@GetMapping`. La création est plutôt depuis une fonction annotée par `@PostMapping`. Ici nous avons fait ce choix pour éviter d'avoir un créer un formulaire HTML pour la création d'un événement (équipé de la méthode **POST**), mais dans une application réelle, il est préférable de suivre cette convention. L'objet du prochain TP sera donc l'utilisation de formulaires, leur traitement dans Spring MVC, ainsi que le procéssus de validation.