Pré-requis :

- JDK 8+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)

TP1 : Première configuration XML avec Spring

Le premier atelier reprend l'exemple du support.

Nous voulons développer un objet de type service capable de lister tous les films d'un réalisateur. Les films sont stockés dans un support persistant (un simple fichier texte pour le moment).

La classe service fournit une méthode :

```
public List<Movie> moviesDirectedBy(String director)
```

Une interface Java définit les méthodes que doit impémenter par la couche de persistence .

```
public interface MovieDAO {
	public List<Movie> findAll();
}
```

Une implémentation basée sur un fichier CSV est fournie (***org.fomation.dao.FileDAO***).

Une classe de test est également fournie pour valider votre implémentation .

Etapes

1. Mise en place du projet
Importer le projet Maven fourni
Visualiser le POM

2. Implementer org.formation.service.MovieLister
Implémentez la méthode métier et utilisez le constructeur ou le setter pour injecter des dépendances.

5. Configurer Spring
Dans le fichier *org.formation.service.test.xml*, déclarez les beans requis et définissez leurs attributs.

6. Executer la classe de test
Executer le test *org.formation.test.MovieListerTest.*

TP2 : Cycle de vie des beans

Dans *MovieLister*,

- Implémenter une méthode d'initialisation : Afficher le timestamp
- implémenter ApplicationContextAware et afficher toutes les définitions de Beans disponibles

Ré exécuter le test

Changer le scope du bean MovieLister pour le passer en mode prototype.

Dupliquer le code de la méthode de test pour faire appel 2 fois à la méthode getBean(«MovieLister »)

Exécuter le test et visualiser les affichages

TP3 : Mise en place des annotations

Nous voulons transformer la configuration du précédent projet avec des annotations.

- Dans l'arborescence test crée une classe TestConfiguration qui déclare un bean de type FileDao
- Utiliser les annotations nécessaires dans MovieLister
- Modifier le test afin que Spring boote avec notre classe de configuration

TP3-Bis : Multiple Configurations

Nous voulons utiliser *MovieLister* avec une implémentation différente de *MovieDao* : ***MockDao***

- Créer une classe *MockDao* implémentant *MovieDao* et renvoyant une liste d'objet *Movie* en « dur »
- Compléter le test afin que Spring celui exécute avec la classe Mockée

TP4 : Propriété de configuration et SpEL

Créer un fichier *properties* et le placer dans le classpath du projet

Y définir une propriété inidiquant le fichier ou sont stocker les films

Annotez correctement la classe de configuration et injectez la propriété externe avec @Value

Utiliser @Component pour le Bean *MovieDao*

TP5 : Mise en route SpringBoot

- Création d'un projet Java Spring Boot
  - Dépendance sur Web + code exemple
  - Exécution, (Run As → Spring Boot App)
  - Accéder à *localhost:8080*
  - Aller dans les Run → Configurations, surcharger la propriété *server.port*
    - Accéder à la nouvelle URL

- Créer une classe contenant le code suivant :

```
@Component
@ConfigurationProperties("hello")
public class HelloProperties {

	/**
	 * Greeting message returned by the Hello Rest service.
	 */
	private String greeting = "Welcome ";

	public String getGreeting() {
		return greeting;
	}

	public void setGreeting(String greeting) {
		this.greeting = greeting;
	}
}
```

- Une autre classe implémentant un service REST :

```
@RestController
public class HelloController {

	@Autowired
	HelloProperties props;

	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
		return props.getGreeting()+name;
	}
}
```

Ajouter dans le *pom.xml* une dépendance permettant de traiter l'annotation *@ConfigurationProperties* pendant la phase de build et créer les fichiers nécessaires afin que Spring Tool prennent en compte les nouvelles propriétés :

```
		<dependency>
```

```
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-configuration-
processor</artifactId>
            </dependency>
```

- Tester ensuite la complétion dans l'éditeur de propriétés
- Tester l'application

TP6 : Développement avec SpringBoot

Auto configuration et Configuration de Debug

Créer un nouveau projet SpringBoot avec un simple classe Main et sans ajout de modules

Dans la classe Main, utiliser la valeur de retour de `SpringApplication.run(Application.class, args);` pour afficher les beans Spring configurés.

Démarrer l'application en ligne de commande après avoir fait un build (*mvn package* ou *gradlew bootJar*)

Ajouter le module starter-web et réexécuter le programme

Ajouter les classes du TP précédent, la dépendance sur `spring-boot-configuration-processor`

et l'activation de ***devtools***

Tester le redémarrage automatique lors du changement du code Java

Mode DEBUG et Configuration des traces

Activer l'option ***-debug*** au démarrage

Modifier la configuration afin de générer un fichier de trace

Modifier le niveau de trace du logger *org.springframework.boot* à DEBUG sans l'option ***-debug***

TP7 : Propriétés de configuration

Configuration Externe

Renommer le fichier *application.properties* en *application.yml*

Définir les propriétés suivantes :

- Valeur aléatoire dans le fichier *yml* affectée à une variable du contrôleur
  Afficher la valeur dans un point d'accès du contrôleur
- Dans *HelloProperties* (Ajouter des validations):
  - *hello.greeting* : Non vide
  - *hello.styleCase* : Upper, Lower ou Camel
  - *hello.position* : 0 ou 1

TP8 : Profils

Profil

Définir un port différent pour le profil *prod*

Activer le profil :

- Via votre IDE
- Via la ligne de commande après avoir généré le *fat jar*