



Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Rue Royale, 67. 1000 Bruxelles
02/219.15.46 – esi@he2b.be

Applications d'Entreprise (Java Spring)

2020

Bachelor en Informatique
WEBG5

M. Codutti (MCD)

Document produit avec L^AT_EX.

Version du 25 mars 2021.

A participé à de précédentes versions du Syllabus : J. Lechien (JLC)



Ce document est distribué sous licence
Creative Commons Paternité - Partage à l'Identique 2.0 Belgique (<http://creativecommons.org/licenses/by-sa/2.0/be/>).
Les autorisations au-delà du champ de cette licence peuvent être demandées à esi-webg5-list@he2b.be.

Table des matières

Préface	7
Conventions	9
I Rappels et mise à niveau	11
1 Rappels	13
1.1 Requête web classique	13
1.2 Requête Ajax et service web	14
2 Les moteurs de production	15
2.1 Ant	16
2.2 Maven	18
3 Visual Studio Code	23
3.1 Développer en Java	23
4 Concepts avancés de Java	25
4.1 Générer des traces	25
4.2 Le projet Lombok	27
II Introduction à Spring	29
5 Java Spring vu du ciel	31
5.1 Un peu de vocabulaire	31
5.2 Les outils et leur installation	32
5.3 L'écosystème Spring	32
6 Première application Spring	35
6.1 Une base de projet	35
6.2 Une page d'accueil statique	37
6.3 Une page d'accueil dynamique	37
6.4 DevTools	38
6.5 Un accueil personnalisé	39
6.6 Tester le site	39
6.7 Déployer le site	40
7 Présentation de l'étude de cas	41
7.1 Le programme annuel de l'étudiant	41
III Une application web (Spring MVC)	43
8 Thymeleaf	45
8.1 Incorporer des ressources	45
8.2 Passer une donnée à la vue	46
8.3 Itérer sur une collection	47

TABLE DES MATIÈRES

8.4	Affichage conditionnel	48
8.5	Liens	48
8.6	Formulaire	49
8.7	Objets prédéfinis	50
8.8	Les fragments	50
8.9	Un layout	51
9	Le contrôleur	53
9.1	L'annotation @Controller	53
9.2	L'annotation @Autowired	53
9.3	L'annotation @Bean	54
9.4	Les durées de vie (portée, <i>scope</i>)	55
9.5	L'annotation @ModelAttribute	57
9.6	L'annotation @SessionAttributes	57
9.7	Redirection	57
9.8	La conversion des données	58
9.9	La validation des données	58
10	La validation des beans	59
10.1	Installation	59
10.2	Définir les contraintes	59
10.3	Demander une validation	60
10.4	Afficher les erreurs dans la vue	60
IV	Interopérabilité	61
11	Les services webs de type REST	63
11.1	Présentation	63
11.2	Définir un service	63
11.3	Passer des paramètres	64
11.4	Statut de la réponse	65
11.5	Rappel JSON	65
11.6	Format de la réponse	66
11.7	Utiliser un service	66
11.8	Exercices	67
V	Application mono page	69
12	Ajax et Thymeleaf	71
12.1	Rappel	71
12.2	Incorporer un fragment Thymeleaf	71
13	Un front-end indépendant	75
13.1	Un client JavaScript	75
13.2	Vue.js	77
VI	Persister les données (Spring JPA)	79
14	Introduction	81
14.1	Rappel JDBC	81
14.2	H2, un SGBD embarqué	82
15	La persistance avec JPA	83
15.1	Une entité	83
15.2	Spring JPA	83

16 Les entités	87
16.1 Définir la base ou les entités ?	87
16.2 Traces	87
16.3 Génération automatique des clés	88
16.4 Les clés composées	88
16.5 Persistance des types des données	88
16.6 Les associations	89
16.7 Paresseux ou gourmand ?	92
16.8 Cascade	93
16.9 Et le reste...	93
17 Requêtes avancées	95
17.1 Les méthodes requêtes	95
17.2 L'annotation @Query	95
17.3 Notions choisies de JPQL	96
18 Configurer le SGBD	99
18.1 MariaDB	99
18.2 Les profils	99
VII Compléments	101
19 Les tests	103
19.1 Tester une entité	104
19.2 Tester un repository	105
19.3 Tester un code métier	106
19.4 Tester un service REST	107
19.5 Tester un site web	108
19.6 Exercice	108
20 L'internationalisation	109
20.1 Internationalisation des messages	109
20.2 Localisation des données	111
21 La sécurité	113
21.1 Introduction	113
21.2 La sécurité par défaut	114
21.3 Utilisateurs codés en dur	115
21.4 Restreindre l'accès	115
21.5 Personnaliser les pages liées à l'identification	116
21.6 Savoir qui est connecté	117
21.7 Utilisateurs stockés dans une BD	118
21.8 Chiffrer les mots de passe	119
22 Les websockets	121
VIII Récapitulatif	123
23 Gestion de projet Scrum	125
24 Un quiz	129
25 Le programme des étudiants	133
IX Annexes	137
A Deployer une application Spring	139

TABLE DES MATIÈRES

A.1 Heroku	139
A.2 Docker	140
B Liste des dépendances utiles	141

Préface

Ce syllabus a été écrit à destination des étudiants de l'ÉSI comme support pour le cours APPLICATIONS D'ENTREPRISE de l'UE WEBG5. Il suppose que l'étudiant a réussi les UE WEBG2 et WEBG4 ainsi que les cours de Java de 1^{re} année. Par conséquent, il suppose que l'étudiant possède :

- ▷ une bonne connaissance de JAVA SE ;
- ▷ une certaine familiarité avec HTML, CSS, JAVASCRIPT, AJAX, XML et JSON ;
- ▷ une compréhension de certains concepts vus avec PHP qui resserviront ici : la notion de client-serveur, la production des pages au niveau du serveur via des templates, le SQL embarqué, notamment.

Nous renvoyons l'étudiant sur la plateforme d'apprentissage de l'école pour obtenir toutes les informations complémentaires :

- ▷ modalités d'évaluations ;
- ▷ exemples d'anciennes évaluations ;
- ▷ versions électroniques des exemples ;
- ▷ documents de support pour les exercices.

Attention ! Ces notes de cours sont récentes et contiennent probablement quelques erreurs. Nous vous invitons à signaler tout problème rencontré : typo, explication peu claire...

Conventions

Conventions typographiques

Dans ce document, nous adopterons les conventions suivantes.

- ▷ Les commandes, les noms de fichiers et les bouts de code à interpréter tels quels seront écrits dans une police de type machine à écrire, éventuellement avec un surlignement.
- ▷ Les sigles, acronymes et noms de produits sont écrits en PETITES CAPITALES (ex : JAVA).

Exemple 1

Un exemple

Ceci est un exemple pour illustrer la théorie.

Tutoriel 1

Un tutoriel

Les tutoriels vous permettent de coder un exemple concret en vous fournissant un pas-à-pas détaillé.

☒ Ceci est un point qui demande une action de votre part.

Exercice 1

Un exercice

Ceci est un exercice où vous devez mettre en pratique la théorie en vous inspirant des exemples et des tutoriaux.

```
1 /*  
2  * Les extraits de code avec un nom en haut à droite sont disponibles sur le site.  
3  * Si vous téléchargez de poÉSI le zip comprenant le syllabus et les codes,  
4  * les liens seront fonctionnels.  
5 */
```

convention.txt

Conventions de nommage

Dans nos codes, nous allons respecter certaines conventions de noms afin de s'y retrouver plus facilement. Il s'agit essentiellement de règles sur les noms des packages. Tout votre code sera réparti en sous-packages comme suit :

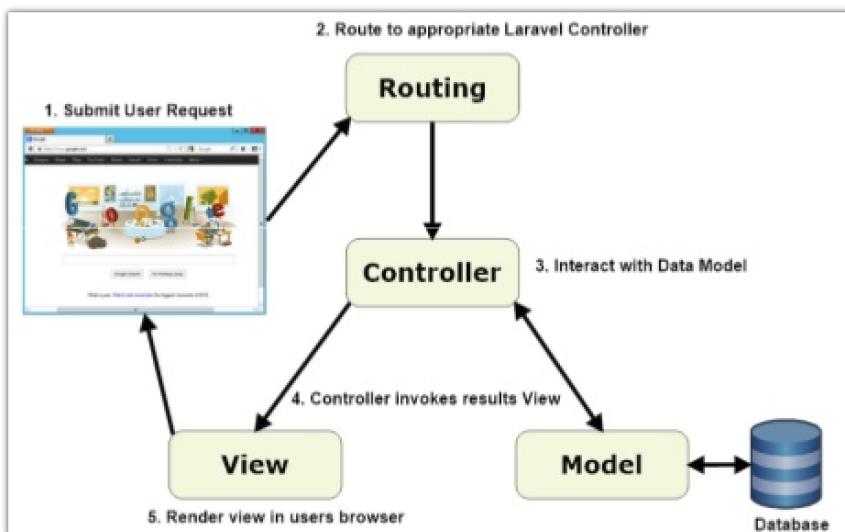
model	Les données du modèle (entités, DTO...).
db	Le code assurant la persistance des données.
business	Le code métier (façade...).
web	Les contrôleurs pour les pages web.
rest	Les services web de type REST.

Première partie

Rappels et mise à niveau

Commençons par nous rappeler ce que vous avez fait au cours de WEBG4.

1.1 Requête web classique



1. Le navigateur du client envoie une requête au serveur (protocole HTTP).
2. Le serveur web est configuré pour transférer la requête à LARAVEL.
3. En fonction du fichier de configuration de routage, la requête est passée au code PHP dédié (le contrôleur). Par exemple : `Route::get('/user', 'UserController@index');`
4. Le contrôleur (dans notre exemple la méthode `index` de la classe `UserController`) va interagir avec le modèle, accéder aux bases de données ¹ via du *SQL embarqué*. Certains d'entre-vous ont également vus ELOQUENT.
5. En fonction des résultats, le contrôleur va passer le relais la vue appropriée en ayant pris soin de placer dans une zone commune les données dont elle aura besoin.
6. La vue va générer du HTML au travers d'un *moteur de template* (BLADE).
7. Le serveur retourne le HTML produit au client.
8. Le navigateur du client interprète l'HTML reçu et s'occupe du rendu. Il devra éventuellement demander des ressources complémentaires au serveur : CSS, JAVASCRIPT, IMAGES...

Du point de vue du client, il ne sait pas quel langage tourne sur le serveur : il envoie une requête et reçoit du HTML.

¹ Si on veut suivre les bonnes pratiques jusqu'au bout, le contrôleur n'accède jamais directement à la BD. C'est la responsabilité de la couche *business* qui offre une façade aux contrôleurs.

1.2 Requête Ajax et service web

Vous avez également vu comment utiliser les technologies AJAX et les services webs REST pour développer des applications web mono-pages.

- ▷ En fonction d'une action de l'utilisateur sur la page, un code JAVASCRIPT va envoyer une requête REST au serveur.
- ▷ Au niveau du serveur, on va passer essentiellement par les mêmes phases que dans le cas précédent mais on ne va retourner que les données (souvent au format JSON) plutôt que toute une page HTML.
- ▷ Lorsque ces données arrivent au client, elles sont traitées par un code JAVASCRIPT qui peut, par exemple, les incorporer à la page.

Un programme doit passer par toute une série d'étapes avant d'obtenir une application aboutie : compilation, exécution des tests, création de la javadoc, déploiement...

En première année, vous avez utilisé NETBEANS et il suffisait alors d'appuyer sur des boutons. Au *laboratoire environnement* vous avez appris à taper des commandes explicites (`javac`, `javadoc`, `java`).

Dans une situation concrète, il n'est pas rare que plusieurs équipes développent ensemble une même application en utilisant des outils différents. Régulièrement, il faut mettre toutes les parties ensemble et les *intégrer* afin d'obtenir le produit final. Évidemment, tout ça doit se faire facilement et rapidement, d'où le besoin d'outils qui automatisent ces tâches.

Plusieurs outils — le terme consacré est « moteur de production » — sont utilisés actuellement. Citons :

Un script

contenant les instructions à enchaîner. Rarement utilisé car tout est reconstruit même si c'est inutile (par ex. on va compiler un fichier qui n'a pas été modifié).

Make

est l'outil historique sur les systèmes Unix. Il permet de ne refaire que ce qui est nécessaire. Il n'est pas lié à Java (<http://fr.wikipedia.org/wiki/Make>).

Ant

est la solution historique proposée par la société Apache pour *Java*. Elle est encore parfois utilisée pour des projets simples.

Maven

est un autre produit de la société Apache. Pensé pour les grosses applications JAVA, il est plus récent et plus adapté à de gros projets décentralisés. C'est la solution la plus utilisée actuellement en entreprise.

Gradle

Un produit récent qui tente d'allier les atouts de Ant et de Maven. Les fichiers de configuration sont écrits dans le langage Groovy. (<http://www.gradle.org/>)

Avec NETBEANS, il suffit de cliquer sur un bouton pour construire votre projet. Mais comment fait-il ? Pour plus de souplesse, ce qu'il doit faire n'est pas *en dur* dans son code mais est délégué à un moteur de production¹.

Dans ce cours, nous verrons ANT car c'est l'outil que vous avez utilisé en 1^{re} (certes, sans le savoir) et surtout MAVEN car c'est l'outil le plus utilisé actuellement en entreprise et que c'est celui que nous allons utiliser. Notre objectif n'est pas que vous soyez des pros de ces outils mais que vous en ayez la connaissance de base suffisante pour être capable de lire les fichiers de configuration et d'effectuer des modifications de base.

1. Les anciennes versions de NETBEANS utilisaient ANT par défaut. Les versions récentes mettent sur un même pied ces trois moteurs.

2.1 Ant

Ant (*Another Neat Tool*) [1] est un outil développé par James Duncan Davidson qui permet d'automatiser les tâches courantes du développeur. L'article sur WIKIPEDIA nous dit :

« ANT est un logiciel créé par la fondation APACHE qui vise à automatiser les opérations répétitives du développement de logiciel telles que la compilation, la génération de documents (JAVADOC) ou l'archivage au format JAR, à l'instar des logiciels MAKE. »

Il se base sur un fichier de configuration (`build.xml`) qui permet d'indiquer ce qu'on veut et comment on le veut.

Lorsque vous créez un projet ANT sous NETBEANS² il sera utilisé pour gérer votre code. Tout ça est transparent pour vous et vous n'avez pas à vous en soucier... tant que tout va bien. Parfois, la configuration n'est pas la bonne ou les fichiers sont corrompus et il est alors bon de savoir les lire pour les corriger.

Les bases

Basons-nous sur quelques exemples de difficultés croissantes pour apprêhender ce qu'on peut mettre dans le fichier de configuration et comment l'utiliser.

Exemple 1

Hello, world !

Respectons la tradition et écrivons un fichier ant qui affiche le message « *Hello, world!* » à l'écran. Soit le fichier `build.xml` contenant le texte suivant :

```

1 <project>
2   <target name="hello">
3     <echo message="Hello, world !"/>
4   </target>
5 </project>
```

moteur/ant-ex1.xml

Pour l'exécuter, il suffit de taper la commande `ant hello` et on obtient

```

> ant hello
Buildfile: C:\TEMP\build.xml

hello:
    [echo] Hello, world !

BUILD SUCCESSFUL
Total time: 0 seconds
```

Quelques remarques s'imposent.

- ▷ Le fichier de configuration contient un ensemble de cibles (*target*).
- ▷ Chaque **cible** porte un nom et indique les tâches à effectuer pour réaliser la cible. Dans notre exemple, afficher un message sur la console.
- ▷ Lorsqu'on lance l'outil, on indique la cible à atteindre. Il est également possible de spécifier une cible par défaut.

Exemple 2

Hello, world ! en Java

Abordons un exemple (un peu) plus réaliste où on utilise ANT pour automatiser les tâches dans le cadre d'un petit projet JAVA constitué d'un seul fichier source (`Hello.java`) sans *package*.

2. Dans ce cours, nous n'allons pas utiliser NETBEANS pour développer mais VISUAL CODE STUDIO. Mais, pour ne pas brûler les étapes, nous commencerons par voir comment fonctionnent ces moteurs de production avec NETBEANS.

```

1 <project>
2
3   <target name="compile">
4     <javac srcdir=". " includes="Hello.java"/>
5   </target>
6
7   <target name="run" depends="compile">
8     <java classname="Hello" classpath=". "/>
9   </target>
10
11  <target name="doc">
12    <mkdir dir="doc"/>
13    <javadoc sourcefiles="Hello.java" destdir="doc"/>
14  </target>
15
16 </project>

```

Pour l'exécuter, il suffit de taper `ant compile`, `ant run` ou `ant doc`.

Remarques :

- ▷ Les tâches JAVA les plus courantes sont prédéfinies dans ANT.
- ▷ Chaque tâche est paramétrable.
- ▷ Une cible peut avoir des dépendances. Ici, avant d'exécuter le programme on vérifiera qu'il est bien compilé.
- ▷ Plusieurs tâches peuvent être associées à la même cible.
- ▷ Une tâche n'est exécutée que si nécessaire. Ici,
 - ▷ si le fichier `Hello.class` est plus récent que le fichier `Hello.java` la compilation n'a pas lieu ;
 - ▷ si le dossier `doc` existe déjà, il n'est pas créé.
- ▷ Si une tâche rate (*fail*), tout s'arrête et la cible n'est pas atteinte (configurable).

Exemple 3

Un projet Netbeans

Si on crée un projet NETBEANS en utilisant ANT comme moteur de production, il crée une configuration composée de plusieurs fichiers.

`build.xml`

contient très peu de texte. Il importe le fichier `nbproject/build-impl.xml`.

`nbproject/build-impl.xml`

contient concrètement la configuration de ant. Si un élément de configuration ne vous convient pas, vous pouvez le modifier ici ou, mieux, le redéfinir dans le fichier `build.xml`.

`nbproject/project.properties`

contient des propriétés importées par `nbproject/build-impl.xml`.

D'autres fichiers dans le dossier `nbproject` ne sont pas liés à ANT. Ils contiennent la configuration de NETBEANS : préférences de l'utilisateur, fichiers ouverts...

Installer l'outil

Vous pouvez installer ANT indépendamment de NETBEANS. Il est aussi possible d'utiliser la version de ANT fournie avec NETBEANS en configurant correctement son environnement. ANT se base sur 3 variables d'environnement :

- ▷ `ANT_HOME` indique le dossier où se trouve ANT ;
- ▷ `JAVA_HOME` indique le dossier où se trouve le JDK de JAVA (ANT est écrit en JAVA) ;
- ▷ `PATH` doit contenir `%ANT_HOME%/bin`.

Sur les PC de l'école, ANT peut être trouvé dans le dossier `extide/ant` de NETBEANS.

Exercices

Exercice 1

Utiliser Ant

Pour cet exercice, je vous propose de définir votre environnement ANT et de reproduire les exemples 1 et 2.

Exercice 2

Hello Ant

Je vous propose de créer une petite application « Hello Ant ! » dans NETBEANS et de l'exécuter en dehors de NETBEANS via un accès direct à ANT.

Exercice 3

Manipuler la config de Netbeans

Repartons de l'application de l'exercice précédent. Apportez-y quelques modifications.

1. Modifiez le dossier où est placé la javadoc.
2. Arrangez-vous pour que NETBEANS affiche dans ses traces le message "Allez, on y croit!" avant de compiler.

2.2 Maven

Maven [2][3], plus récent que Ant, le remplace de plus en plus dans les entreprises. L'article sur Wikipedia nous dit :

« Apache Maven est un outil pour la gestion et l'automatisation de production des projets logiciels Java en général et Java EE en particulier. »

Le site consacré à Maven le présente ainsi :

« Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. »

De nos jours, c'est l'outil le plus utilisé, même en dehors de JAVA. Il commence à être concurrencé par GRADLE.

Comparaison avec Ant

Quelles sont les grosses différences avec Ant ?

- ▷ Il se base sur le principe « *Convention over configuration* ». Lorsque vous créez un certain type de projet (on parle d'ici d'*« archétype »*) il crée une arborescence par défaut pour votre projet. Si vous la suivez, il n'y a rien d'autre à préciser.
- ▷ On indique quelles sont les dépendances du projet via des identifiants uniques et des versions minimales mais pas où trouver ces dépendances.
- ▷ Les dépendances sont trouvées via un système de dépôts (local, de site ou global), un peu comme les dépôts dans une distribution Linux ou le *Play Store Android*. Si quelque chose manque, il va automatiquement le chercher dans les dépôts et le ramène.
- ▷ Un projet peut lui même être ajouté (installé) à un dépôt pour être utilisable par d'autres projet (via une dépendance).

Netbeans peut utiliser tout aussi bien MAVEN que ANT comme moteur de production. C'est cet outil que nous allons utiliser cette année puisque c'est celui que vous rencontrerez le plus probablement en entreprise.

Un peu de vocabulaire

Maven ayant un vocabulaire particulier, il est bon de s'y attarder un moment.

artifact

Un projet ou, plus précisément, ce qui va être produit (un fichier jar par exemple). Il est caractérisé par : un *artifactId* (nom unique), un *groupId* (permet de regrouper des artifacts liés) et une version. Ces éléments vont permettre de le distinguer dans les dépôts.

archéotype

Un prototype d'artifact. Il peut être utilisé pour créer une structure de projet adaptée à ce qu'on développe (application JAVA classique, SPRING, PHP...)

Une première approche

Le fichier de configuration s'appelle `pom.xml` pour *project object model*. Plus court qu'un fichier Ant, il *décrit* le projet et ses dépendances.

Exemple 4

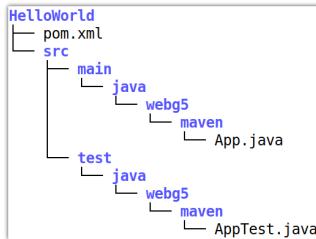
Un projet Java SE de base

La première étape est de créer la structure du projet. On le fait à partir d'un archéotype.

```
mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart
```

La commande pose quelques questions. Entrez `webg5.maven` comme id de groupe et `HelloWorld` comme id d'artifact. Vous pouvez garder les valeurs par défaut pour tout le reste.

Au final, MAVEN crée une structure de projet JAVA simple avec une classe de base et une classe de test.



Le fichier de configuration (`pom.xml`) contient (extraits)

```

1 <groupId>adi3g.maven</groupId>      // Le groupe auquel appartient le projet
2 <artifactId>HelloWorld</artifactId> // L'identifiant du projet (unique au sein du groupe)
3 <packaging>jar</packaging>          // Le format du produit final
4 <version>1.0-SNAPSHOT</version>     // La version du projet
5
6 <properties>
7   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
8   <maven.compiler.source>1.7</maven.compiler.source> // Utilisera Java 7
9   <maven.compiler.target>1.7</maven.compiler.target>
10 </properties>
11
12 <dependencies>
13   <dependency>                // Dépend de ...
14     <groupId>junit</groupId>
15     <artifactId>junit</artifactId> // ... JUnit ...
16     <version>4.11</version>       // ... 4.11 ...
17     <scope>test</scope>         // ... pour les tests.
18   </dependency>
19 </dependencies>
20 </project>

```

Remarquez qu'il est plus descriptif qu'un fichier de ANT. On peut à présent indiquer la *phase* qu'on veut atteindre, par exemple la création du package final, via

```
mvn package
```

Le projet passe alors par différentes phases : compilation, test... définies par défaut et produit, au final, le *jar* (dans `target`).

Patience !

Ce processus peut prendre du temps la première fois car Maven va automatiquement télécharger ce dont il a besoin, JUnit par exemple, à partir d'un dépôt central. Ce ne sera plus le cas les fois suivantes. Tout ce qu'il téléchargé est stocké dans le dépôt local, un dossier caché nommé `.m2`.

Erreur de version

S'il se plaint d'une incompatibilité avec la version 4 de JAVA, c'est qu'il a utilisé une vieille version de l'archetype. Vous pouvez forcer la dernière version en ajoutant l'option `-DarchetypeVersion=RELEASE` lors de la création du projet.

Le `jar` peut être exécuté de façon classique :

```
java -cp target/HelloWorld-1.0-SNAPSHOT.jar webg5.maven.App
```

Structure d'un projet

Maven a standardisé la structure d'un projet. Dans l'exemple, on peut constater que :

- ▷ **src** contient toutes les sources au sens large (ce que le développeur fournit) ;
- ▷ **src/main/java** contient les classes Java de l'application ;
- ▷ **src/test/java** contient les classes Java des tests unitaires ;
- ▷ **target** contient tout ce qui est produit et notamment le fichier jar ;
 - ▷ **target/classes** contient les versions compilées des classes ;
 - ▷ **target/test-classes** contient les versions compilées des classes de test (qui ne seront pas dans le jar produit).

Les étapes du développement

Maven standardise également les étapes par lesquelles passe un projet. Voici les principales :

clean	Nettoie le projet (détruit <code>target</code>)
compile	Compile les sources
test	Lance les tests unitaires
package	Crée l'artifact
install	Copie l'artifact dans le dépôt local

Toutes ces phases (à part `clean`) se suivent et s'enchaînent. Ainsi, si vous lancez l'étape `package`, il lancera les tests unitaires avant.

Les plugins

L'exécution n'est pas une phase standard du processus³. Cette phase est souvent introduite via un **plug-in**. Un plugin peut ajouter de nouvelles phases ou servir à modifier le comportement d'une phase standard.

Par exemple, notre code JAVA pourrait aussi s'exécuter via la commande

```
mvn exec:java -Dexec.mainClass="webg5.maven.App"
```

La différence étant qu'on passe ici par MAVEN qui se chargera de le construire avant de l'exécuter, si nécessaire.

3. MAVEN a été pensé avant tout pour *construire* un projet.

Les portées

Les dépendances permettent d'indiquer ce dont le projet a besoin et quand il en a besoin (via l'attribut *scope*). Voici quelques *scopes* fréquemment rencontrés :

compile

Nécessaire à la compilation et par la suite (scope par défaut). La dépendance est incorporée à l'artifact produit.

runtime

Nécessaire uniquement pour l'exécution. La dépendance est incorporée à l'artifact produit.

provided

Nécessaire à la compilation et par la suite (comme **compile**) mais la dépendance est supposée exister dans l'environnement d'exécution et n'est donc pas incorporée à l'artifact produit.

test

Nécessaire à la compilation et l'exécution des tests. La dépendance n'est *pas* incorporée à l'artifact produit.

Maven et Netbeans

Comment se passe l'intégration entre ces deux produits ? La bonne nouvelle, c'est que Netbeans reconnaît MAVEN directement sans devoir rien installer.

Exemple 5

Maven et Netbeans

Lorsqu'on crée un projet avec Netbeans, on peut remarquer qu'il existe une catégorie **Maven with Maven** avec un ensemble d'archétypes connus. Pour l'exemple, choisissons « Java Application ». On obtient un projet avec un fichier **pom.xml**.

Bien évidemment, les boutons de Netbeans sont associés à des *phases* de Maven. En fait, tant que tout va bien, c'est totalement transparent pour l'utilisateur. Si vous voulez plus de détails, vous pouvez consulter la partie **Actions** des propriétés du projet.

Installer l'outil

À nouveau, il est possible d'installer MAVEN à part ou d'utiliser la version fournie avec NETBEANS en configurant correctement l'environnement. Concrètement, il suffit d'ajouter au PATH le dossier (**bin**) contenant les binaires de Maven⁴.

Exercices

Exercice 4

Utiliser Maven

Pour cet exercice, je vous propose de définir votre environnement Maven et de reproduire l'exemple 4 page 19.

Exercice 5

Maven et Netbeans

Reproduisez l'exemple 5. Ajoutez une petite méthode principale. Compilez et exécutez à partir de Netbeans puis en ligne de commande.

4. Sur les PC de l'école, ils sont dans le dossier **java/maven** de Netbeans

Chapitre

3

Visual Studio Code

Vous avez l'habitude d'utiliser NETBEANS pour vos développements JAVA. Ce serait tout-à-fait possible d'utiliser cet IDE dans ce cours mais nous avons plutôt choisi d'utiliser VISUAL STUDIO CODE (VS CODE) avec lequel vous avez déjà fait connaissance dans les autres cours de développement web.

VSCodium

À l'école, vous trouverez plutôt VSCODIUM (<https://vscode.com/>), une version open source de VS CODE. C'est exactement les mêmes sources avec une grosse différence : il n'envoie pas de statistiques à MICROSOFT.

Certaines versions de CODIUM^a sont également configurées pour ne pas prendre les extensions dans le *Market Store* géré par MICROSOFT mais dans une alternative libre : *Open VSX Registry* (<https://open-vsx.org/>).

Certaines extensions ne s'y trouvent pas (pour des raisons de licence) mais vous pouvez les télécharger et les installer facilement à partir du *VSixHub* (<https://www.vsixhub.com/>).

^a. Ce n'est apparemment pas le cas de la version installée à l'école.

3.1 Développer en Java

Voyons donc comment l'utiliser pour du développement en JAVA [4]. Plus spécifiquement, avec MAVEN puisque c'est le moteur que nous utiliserons tout au long de ce cours.

Tutoriel 1

Premier projet Java avec VSCode

Ne dérogeons pas à la règle et écrivons un *Hello World*.

Étape 1 : Configurer VSCode.

Vous savez que de nombreuses extensions existent dans VS CODE pour étendre les capacités de l'éditeur. Le Java Extension Pack [5] installe les extensions essentielles pour le travail en JAVA.

- ☒ Dans VS CODE, allez dans l'onglet des extensions.
- ☒ Recherchez et installez le Java Extension Pack.

VS CODE redémarre tout seul et les extensions sont à présent disponibles. Cette étape ne doit bien sûr être faite qu'une seule fois (par machine).

Étape 2 : Créer le projet.

Créons un projet de base géré par MAVEN.

- ☒ Entrez dans le mode commande : CTRL-SHIFT-P.
- ☒ Entrez `Create Maven Project`. Il vous pose quelques questions sur le projet à créer.

- ☒ L'archetype à utiliser : choisissez `maven-archetype-quickstart`.
- ☒ La version de l'archetype : choisissez la plus récente.
- ☒ Le dossier où le créer sur le disque : choisissez ce que vous voulez.

Il lance alors la création d'un projet en mode interactif dans le terminal. Quelques questions vous sont encore posées.

- ☒ Le nom du groupe : `webg5.vscode`
- ☒ Le nom de l'artifact (du projet) : `Hello`
- ☒ La version et le package : acceptez les valeurs par défaut.
- ☒ Confirmez.

Le projet, au format MAVEN, est à présent créé dans le dossier indiqué.

Étape 3 : Ouvrir le projet.

Le projet créé n'est pas automatiquement ouvert.

- ☒ Ouvrez le dossier qui contient le projet nouvellement créé.
- Vous remarquerez qu'il est composé d'une classe de code `App` et d'une classe de test `AppTest`.

Étape 4 : Exécuter le projet.

- ☒ Ouvrez le fichier `src/main/java/.../App.java`.
- ☒ Cliquez sur le `Run` qui apparaît au-dessus de la méthode principale¹.

Le résultat apparaît dans la console en bas.

Étape 5 : Tester le projet.

- ☒ Ouvrez le fichier `src/test/java/.../AppTest.java`.
- ☒ Cliquez sur le `Run Test` qui apparaît au-dessus de la classe.

Des icônes apparaissent sur chaque méthode et sur la classe pour indiquer le résultat des tests.

Étape 6 : Modifier le projet.

L'extension installée offre de nombreuses facilités lors de l'édition du programme [6]. Voyons quelques exemples.

- ☒ Créez une ligne vide en fin de méthode `main` dans l'application. Entrez `sys` et choisissez `sysout` dans la liste. Un appel à `println` est généré.
- ☒ Placez-vous à présent en dessous de la méthode `main`. Entrez `to` et choisissez `toString` dans la liste. Une méthode `toString` de base est générée.
- ☒ Créez un nouveau fichier `Hello.java` à côté de l'existant. Il crée automatiquement un contenu minimal d'une classe JAVA.

Il en existe beaucoup d'autres (CTRL-ESPACE) ; je vous laisse les découvrir au fur et à mesure.

1. VSCode appelle **CodeLens** ces informations/actions contextuelles qui sont affichées dans le code. Elles ne se trouvent pas physiquement dans le fichier !

Vos connaissances de JAVA sont déjà très bonnes et il n'est pas nécessaire de connaitre tous les aspects avancés du langage pour pouvoir suivre ce cours. Il y a toutefois quelques concepts que vous ne connaissez peut-être pas (ou mal) sur lesquels nous aimerions insister.

4.1 Générer des traces

La journalisation (*logging* en anglais) est une façon de suivre les événements qui ont lieu durant le fonctionnement d'un logiciel. Le développeur ajoute des appels à l'outil de journalisation dans son code pour indiquer que certains événements ont eu lieu. Cela peut être très utile lors du développement pour simuler des parties non existantes, lors du débogage pour comprendre ce qui s'est passé ou encore pour garder un historique du déroulement du programme.

Il existe en JAVA de nombreux systèmes de logging différents mais ils fonctionnent tous plus ou moins de la même façon [7].

1. Chaque message est associé à un **niveau** d'importance (du moins important ou plus important) : TRACE, DEBUG, INFO, WARN, ERROR.
2. Chaque message est envoyé sur un **logger** portant un nom. Souvent, on prend le nom qualifié de la classe comme logger. La notation pointée créant une hiérarchie, on pourra configurer tous les loggers d'un même package d'un coup.
3. Chaque logger est associé à une destination physique (on parle de **appender**) : la console, un fichier, une base de données...
4. Un **fichier de configuration** détermine pour chaque logger quels niveaux de messages vont passer et sur quel *appender* l'envoyer. On peut également déterminer le format exact des messages. Notez que, par défaut, il est fort probable que les messages de faible importance ne soient pas visibles.

Celui que nous allons voir est **Logback** [8] [9] via la façade **SLF4J** [10].

Les concurrents

Passons rapidement en revue les choix possibles.

- ▷ **Java Util Logging (jul)** : la librairie par défaut. Peu utilisée.
- ▷ **Log4J** : une librairie tierce qui a longtemps été une référence. Projet abandonné en 2015.
- ▷ **Logback** : un successeur de Log4J écrit par le même auteur. Utilisé dans de nombreux gros projets. C'est le système de log par défaut dans SPRING.
- ▷ **Log4J2** : un autre successeur de Log4J développé par APACHE. Rencontre beaucoup de succès.

SLF4J est une façade offrant une interface commune à tous les systèmes. L'utiliser permet de facilement passer de l'un à l'autre.

4.1. GÉNÉRER DES TRACES

Tutoriel 1

Hello world en log

Écrivons une petite application qui affiche une trace sur la console.

- ☒ Créez un nouveau projet de base.
- ☒ Ajoutez la dépendence à LOGBACK :

```
1 <dependency>
2   <groupId>ch.qos.logback</groupId>
3   <artifactId>logback-classic</artifactId>
4   <version>1.2.3</version>
5 </dependency>
```

java/logback/pom.xml

Par transitivité, ça inclut également SLF4J.

- ☒ Voici la classe principale :

```
1 package webg5.logback;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5
6 public final class App {
7
8     private static final Logger log = LoggerFactory.getLogger("webg5.App");
9
10    public static void main(String[] args) {
11        log.info("Hello, ");
12        log.error("world !");
13    }
14}
```

java/logback/src/main/java/webg5/logback/App.java

- ☒ Lancez l'application !

Tutoriel 2

Rediriger les traces dans un fichier

Sans toucher au code, redirigeons les traces dans fichier. Faisons également en sorte que seuls les traces de niveau « warn » ou plus y apparaissent.

- ☒ Ajoutez le fichier `logback.xml` dans le dossier `src/main/resources` (à créer).

```
1 <configuration>
2   <appender name="FICHIER" class="ch.qos.logback.core.FileAppender">
3     <file>webg5.log</file>
4     <encoder>
5       <pattern>%d %5p [%c] - %m%n</pattern>
6     </encoder>
7   </appender>
8   <logger name="webg5" level="warn">
9     <appender-ref ref="FICHIER" />
10  </logger>
11 </configuration>
```

java/logback/src/main/resources/logback.xml

À la ligne 8, le paramètre `name` indique qu'on configure tous les loggers qui **commencent** par `webg5` !

- ☒ Relancez l'application !

4.2 Le projet Lombok

Le langage JAVA est parfois fort verbeux et répétitif. Pensez au constructeur, aux accesseurs, aux méthodes `equals`, `hashCode`...

Afin de vous soulager, la plupart des IDE se proposent de les écrire pour vous. C'est bien, mais ce code accroît la taille des fichiers et rend leur lecture plus difficile car le code original est noyé dans ce code répétitif. Or un code est lu plus souvent qu'il n'est écrit.

Afin de résoudre ce problème, le package LOMBOK [11] propose une autre approche : l'utilisation de notations qui remplacent tout le code répétitif. Voyons comment ça fonctionne [12].

Exemple 1

Un getter

Le code suivant définit un attribut et lui fournit automatiquement un getter standard.

```
@Getter  
private String name;
```

Il existe beaucoup d'annotations. Voici celles que nous pourrons utiliser :

@Getter/@Setter

fournit un getter/setter.

@ToString

génère un `toString` de base.

@EqualsAndHashCode

génère un `equals` et un `hashCode` de base.

@NoArgsConstructor/@AllArgsConstructor/@RequiredArgsConstructor

pour un constructeur sans/avec tous/uniquelement les finaux attributs.

@Data

un raccourci pour `@Getter`, `@Setter`, `@ToString` et `@EqualsAndHashCode`.

@Value

idem que le précédent mais sans les setters et tous les attributs sont finaux.

@Slf4j

introduit un attribut statique `log` pour les traces.

Tutoriel 3

Hello Lombok

Testons tout cela en pratique.

- ☒ Créez un nouveau projet JAVA.

Ajoutons la dépendance :

```
1 <dependency>  
2   <groupId>org.projectlombok</groupId>  
3   <artifactId>lombok</artifactId>  
4   <version>1.18.8</version>  
5   <scope>provided</scope>  
6 </dependency>
```

java/lombok/pom.xml

Pour ça :

- ☒ Dans la palette de commandes (CTRL-SHIFT-P), choisissez : Maven: Add a dependency.
- ☒ Cherchez et ajoutez : `org.projectlombok`.
- ☒ Dans le code généré dans le pom, ajoutez : `<scope>provided</scope>` car cette dépendance ne doit pas être ajoutée dans l'artefact.

Créons une classe simple.

- Créez la classe suivante :

```
1 import lombok.Value;
2
3 @Value
4 public class Position {
5     private final int row;
6     private final int column;
7 }
```

java/lombok/src/main/java/mcd/lombok/Position.java

Erreurs apparentes de compilation

Il est probable que l'éditeur souligne certaines erreurs. À priori, il ne sait rien de LOMBOK et croit donc que les constructeurs et les getters sont manquants. Pour résoudre ce problème, il suffit d'installer l'extension Lombok [13] à l'éditeur. Tout rentrera dans l'ordre.

Utilisons-la.

- Ajoutez ceci à la méthode principale :

```
1 System.out.println( "Test de Lombok" );
2 Position pos1 = new Position(1,2);
3 Position pos2 = new Position(1,2);
4 System.out.println(pos1);
5 System.out.println(pos2);
6 System.out.println(pos1.equals(pos2));
7 System.out.println(pos1.getRow());
```

java/lombok/src/main/java/mcd/lombok/App.java

- Testez !

Tutoriel 4

Les traces avec Lombok

Voyons comment utiliser `@Slf4j`.

- Reprenez le projet du tutoriel précédent.
- Ajoutez la dépendance à LOGBACK.
- Créez la classe suivante :

```
1 @Slf4j
2 public class LogHello
3 {
4     public static void main( String[] args ) {
5         log.info("Hello, world !");
6     }
7 }
```

java/lombok/src/main/java/mcd/lombok/LogHello.java

- Testez !

Deuxième partie

Introduction à Spring

Commençons par présenter brièvement le cadre dans lequel nous allons travailler cette année et les technologies que nous verrons.

5.1 Un peu de vocabulaire

Afin de bien nous comprendre, fixons le vocabulaire que nous allons utiliser tout au long de l'année.

Serveur web

Serveur qui a un navigateur web comme client privilégié et qui dialogue avec lui au travers du protocole HTTP (par exemple APACHE).

Serveur web Java

Un serveur web sachant utiliser les *servlets* et les *JSP* qui sont les technologies Java pour traiter les requêtes web. Citons par exemple TOMCAT.

Serveur d'application

Un serveur capable de répondre à des clients divers (web, lourd, autre serveur) et connaissant d'autres protocoles que HTTP. Citons par exemple JBOSS.

Serveur embarqué

Classiquement, un serveur web (ou d'application) tourne sur la machine et notre application y est déposée. Un *serveur embarqué* est intégré à l'application et lancé lorsqu'on exécute l'application. C'est beaucoup utilisé en situation de développement car on a une application qui tourne dans un environnement autonome et contrôlé qui facilite le développement et les tests.

Application d'entreprise

Par ce terme, Java désigne une grosse application comme on les trouve dans les entreprises. Elle est notamment distribuée (découpée en plusieurs parties, physiquement séparées) et sécurisée (contrôle fin sur qui peut faire quoi).

Java EE

Un sur-ensemble du Java standard (JAVA SE) qui ajoute des spécifications (et leurs API) permettant de développer plus facilement des applications d'entreprises. Par exemple, c'est ici que sont définis les *servlets* et les *JSP*. Attention ! C'est juste une norme, pas un produit. Comprenez bien que le langage est strictement identique, c'est l'API standard qui est plus riche. Cette norme a été maintenue par SUN puis ORACLE. Depuis 2018, c'est géré par la *fondation Apache* qui l'a renommé en JAKARTA EE.

Spring

Un *framework* JAVA développé par PIVOTAL. Il est une alternative à la norme JAVA EE. Lors de sa création, il était beaucoup plus innovant ce qui lui a permis de s'imposer comme un incontournable du développement web en JAVA. En interne, il utilise de nombreuses technologies JAVA EE (les *servlets* par exemple). On en est à la version 5.

5.2 Les outils et leur installation

Tous les outils nécessaires sont déjà installés à l'école et probablement chez vous. Nous vous conseillons, pour votre facilité, d'avoir **les mêmes versions** chez vous. Pratiquement, nous allons utiliser :

Java SE

La version actuelle de Java est la 14 (11 pour la LTS). À l'école c'est la 13 qui est installée et vous avez probablement été formé à la 8. Pas de grand changement dans la pratique mais nous insisterons sur certains points qui vous auront peut-être échappés.

Visual Studio Code

Nous pourrions développer avec n'importe quel IDE : NETBEANS, ECLIPSE, INTELLIJ, SPRING TOOL SUITE, VISUAL STUDIO CODE... Vous pouvez utiliser celui que vous voulez mais nous privilégions VS CODE cette année.

Spring

Spring en est à la version 5(.2.8) et nous utiliserons beaucoup de nouveautés de cette version. Vous ne devez rien installer ! Comme nous l'avons vu, les parties du framework utilisées seront chargées automatiquement par MAVEN.

Maven

Les projets que nous allons créer peuvent utiliser une version de MAVEN embarquée mais vous pouvez aussi (c'est recommandé) installer une version indépendante.

git et GitLab

Vous connaissez ces outils qui permettent de faire de la gestion de versions. Vous l'avez probablement déjà installé chez vous. Nous vous invitons à les utiliser.

5.3 L'écosystème Spring

SPRING [14] [15] [16] est un énorme eco-système composé de nombreuses parties. Afin de mieux comprendre ce qui va suivre et ce que vous pourriez trouver lors de vos recherches sur le net, il est bon d'en connaître les principaux composants.

Spring Core

C'est l'élément de base utilisé par tous les autres. Ce qu'il apporte de novateur et qui a fait son succès c'est l'**injection de dépendance**. Lorsqu'une classe a besoin d'une autre — par exemple, si la classe gérant les réservations de livres a besoin de la classe s'occupant des accès à la BD de la bibliothèque — elle ne va pas instancier directement la dépendance mais seulement **déclarer** qu'elle en a besoin. SPRING CORE maintient un ensemble de composants (on parle de **bean**) et va automatiquement les **injecter** là où c'est demandé.

Par exemple, soient les classes A et B suivantes :

```
@Bean
public class B {...}

@Bean
public class A {
    @Autowired B b;
}
```

Grâce à **@Bean**, SPRING sait que c'est à lui à s'en occuper. Le **@Autowired** permet d'indiquer que A a besoin d'un B. Il sera automatiquement initialisé par SPRING et de manière intelligente. Par exemple, il récupérera une instance déjà créée si c'est possible et pertinent.

Spring Boot [17] [18] [19] [20]

Il y a quelques années encore, une application utilisant SPRING devait être accompagnée de fichiers de configurations en XML. Il fallait notamment indiquer les beans et leurs dépendances. Une alternative a été de configurer avec du code JAVA.

SPRING BOOT, introduit il y a quelques années, va plus loin. Il se charge d'analyser votre code et d'en déduire un maximum d'éléments de configuration. Ainsi, plus besoin de déclarer les beans. Autre exemple, il suffit d'ajouter une dépendance à MARIABD pour qu'il comprenne qu'on veut utiliser ce SGBD pour sauver nos objets et il s'occupe de tout ! De nos jours, plus question de programmer en SPRING sans BOOT !

Spring MVC

La partie qui permet de développer des applications web en respectant le patron de conception MVC¹.

Spring Data

Contient tout ce qui traite de l'accès aux bases de données (relationnelles ou non) et de la persistance des objets.

Spring JPA

La partie de SPRING DATA qui s'occupe de la persistance des objets dans une base de donnée relationnelle.

Ce syllabus s'est beaucoup inspiré du livre de Craig Walls, « Spring in action » [21] ainsi que du site de BAELDUNG [22] riche de nombreux tutoriels très bien faits.

1. À l'instar de ce que propose LARAVEL en PHP.

Chapitre

6

Première application Spring

Commençons en douceur en développant un petit site web de bienvenue.

6.1 Une base de projet

Commençons par produire et analyser un projet vide de base.

Il existe plusieurs façons de créer le projet :

- ▷ Via un archetype dédié comme `spring-boot-blank-archetype` (difficile à configurer).
- ▷ Via un l'outil SPRING INITIALZR de PIVOTAL qui propose notamment un site web.
- ▷ Via un outil fournit par l'IDE qui n'est généralement qu'une interface vers l'outil susmentionné.
- ▷ Via un accès en ligne de commande à l'outil, par exemple avec `curl`.

Tutoriel 1

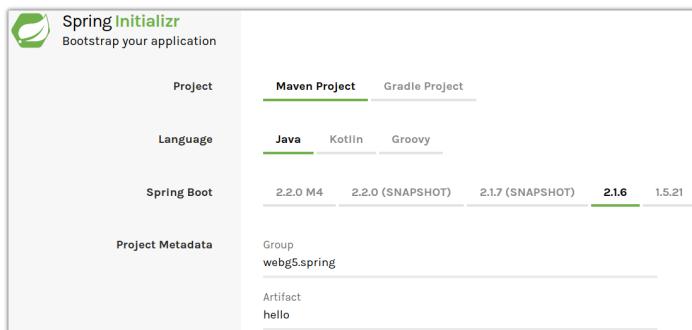
Un projet Spring de base

Dans le cadre de ce cours, nous créerons nos projets via le site web dédié.

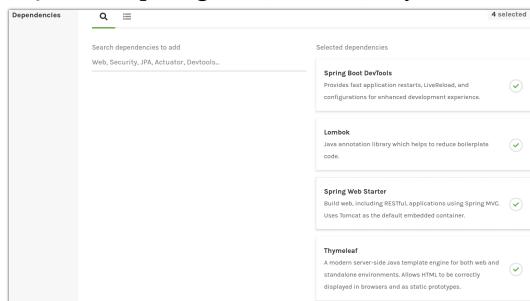
Création du projet

☒ Ouvrez un navigateur et allez sur le site : start.spring.io.

☒ Donnez la configuration de base du projet¹ :



☒ Dans les dépendances, ajoutez : Spring Web, Lombok, Thymeleaf et Spring Boot DevTools.



1. Le style de la page a changé depuis cette capture d'écran mais les informations demandées sont toujours les mêmes.

À quoi servent ces dépendances ?

- ▷ **Spring Web** : La base pour une application web standard.
 - ▷ **Thymeleaf** : Le moteur de template pour les pages web ^a.
 - ▷ **Spring Boot DevTools** : Outils utiles lors du développement (auto refresh...). Ne se retrouveront pas dans le site en production.
 - ▷ **Lombok** : Maintenant, vous savez ;)
-
- a.* Équivalent à BLADE que vous avez utilisé avec PHP.

☒ Cliquez sur le bouton pour générer le projet. Le projet est téléchargé.

Analyser ce qui a été produit

- ☒ Décompressez le projet.
- ☒ Ouvrez-le avec VS CODE et examinez-le.

Les fichiers produits

Qu'y-a-t'il de neuf par rapport à la structure générale d'un projet ?

- ▷ **mvmw (mvm wrapper), mvmw.cmd et .mvm** permettent d'utiliser une version **embarquée** de MAVEN. Utile pour travailler dans un environnement non maîtrisé.
- ▷ **src/main/java** contient une classe principale que nous prendrons le temps de décrire plus tard.
- ▷ **src/main/resources/static** contiendra le contenu statique de notre site (HTML, CSS, JAVASCRIPT, images...).
- ▷ **src/main/resources/templates** contiendra le code de génération des pages web sur le serveur.
- ▷ **src/test/java** contient une classe de test basique.

Exécuter le projet

La classe **HelloApplication** contient la méthode principale

```
1 @SpringBootApplication
2 public class HelloApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(HelloApplication.class, args);
5     }
6 }
```

☒ Exécutez-la !

L'extension Spring

Vous pouvez exécuter l'application :

- ▷ Via le *Codelens Run*.
- ▷ Via MAVEN : `mvn spring-boot:run`.
- ▷ En installant l'extension **Spring Boot Extension Pack** qui offre (notamment) un panneau permettant de lancer/arrêter l'application.

Vous voyez des traces indiquant que Spring est lancé. Et c'est tout ? Oui ! C'est une application web !

- Ouvrez-votre navigateur préféré et entrez l'URL : `localhost:8080`.
Pourquoi ce message d'erreur ? C'est bon signe. Un serveur web a répondu mais comme nous n'avons pas encore écrit de document `index.html`, il ne sait pas quoi nous répondre.

Nous n'avons pas lancé de serveur web ?

En effet ! Il s'agit d'un serveur web **embarqué**. Il est lancé par SPRING juste pour cette application. C'est commode en phase de développement car on contrôle ainsi complètement l'environnement de travail.

- Examinez les traces de l'exécution ; vous y trouverez le nom du serveur et le port sur lequel il écoute.

Tomcat vs Apache

Vous avez constaté que le serveur est TOMCAT. Ce serveur, développé aussi par la fondation APACHE, comprend le protocole HTTP mais aussi des protocoles liés à JAVA (les *servlets* et les *JSP*). En tant que serveur web pur, il est un peu moins perfectionné qu'APACHE.

Tester le projet

Le projet de base contient également une classe de test

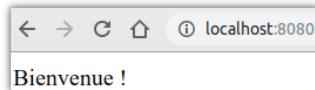
```
1 @SpringBootTest
2 public class HelloApplicationTests {
3     @Test
4     public void contextLoads() {
5     }
6 }
```

- Lancez le test ! `mvn test`

Ce test quasi-vide sert juste à vérifier que toute la machinerie a pu se lancer sans encombre. Nous apprendrons à écrire des tests plus pertinents.

6.2 Une page d'accueil statique

Commençons simplement en créant une page statique pour le site.



Tutoriel 2

Une page statique

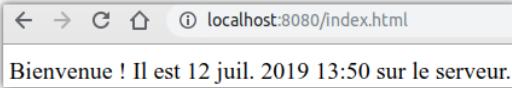
Pour obtenir ce qui est présenté ci-dessus :

- Créez un fichier `index.html` dans le dossier `src/main/resources/static`.
- Écrivez un document HTML de base avec un message de bienvenue.
- Si ce n'est pas le cas, lancez l'application
- Revenez à votre navigateur web et rechargez l'url : `localhost:8080`.

6.3 Une page d'accueil dynamique

Pas besoin de toute cette machinerie si c'est pour servir une simple HTML. Produisons une page qui donnera la date sur le serveur.

6.4. DEVTOOLS



Tutoriel 3

Une page dynamique

Pour obtenir ce qui est présenté ci-dessus :

- ☒ Créez le fichier `HomeController.java` suivant

```
1 @Controller  
2 public class HomeController {  
3  
4     @GetMapping("/")  
5     public String home() {  
6         return "home";  
7     }  
8 }
```

- ☒ Créez un fichier `home.html` dans `src/main/resources/templates` contenant le texte suivant :

```
1 Bienvenue ! Il est  
2 <span th:text="${#dates.format(#dates.createNow(), 'dd MMM yyyy  
3     ↵ HH:mm')}"></span>  
sur le serveur.
```

- ☒ Rechargez la page d'accueil et constatez la différence.

Analysons ce code

- ▷ Le code JAVA a associé une méthode du contrôleur à l'url « / » (via le `@GetMapping`).
- ▷ Ce contrôleur pourrait interagir avec un modèle, préparer des données à afficher sur une page...
- ▷ Dans notre cas, très simple, il se contente de retourner directement le nom de la vue à utiliser : `home`.
- ▷ La vue est trouvée dans le fichier `home.html` du dossier `resources`. Elle contient du code HTML classique et du code THYMELEAF. Ici, c'est juste un appel à la date sur le serveur.
- ▷ Si on n'avait pas donné de contrôleur, l'URL « / » serait mappée par défaut sur le fichier `index.html`.

6.4 DevTools

Rappelez-vous, nous avons inclus cette dépendance dans notre projet. Mettons en évidence ce qu'elle apporte.

Tutoriel 4

DevTools – Mise à jour automatique du projet

Voyons ce qui se passe quand on modifie un peu le projet...

- ☒ Modifiez un mot dans le texte du fichier `home.html`.
- ☒ Sans rien faire d'autre, sauvez le fichier.
- ☒ Retournez dans votre navigateur et rafraîchissez la page, automatiquement mise à jour.

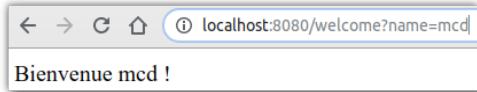
Devtools

DEVTOOLS scrute votre projet. Lorsqu'il voit qu'un fichier est modifié, il va efficacement redéployer ce qui est nécessaire ^a. Cela fonctionne également si la modification a lieu dans un code JAVA : il recompile et redéploie. Le seul fichier qui demandera un redémarrage de l'application est le `pom`.

^a. Ça peut vous paraître évident si vous venez de PHP mais ça ne l'est pas! En PHP, vous avez travaillé directement sur les fichiers présents sur le serveur. Ici, un fichier JAR est produit et c'est lui qui est traité par le serveur. Toute modification au projet doit être répercutée dans le JAR.

6.5 Un accueil personnalisé

Afin de mettre en évidence d'autres aspects de la technologie, imaginons que le nom de la personne soit passé en argument et utilisé dans la page.



Tutoriel 5 Un accueil personnalisé

Pour obtenir ce qui est présenté ci-dessus :

- Créez le fichier `WelcomeController.java` suivant

```
1 @Controller
2 public class WelcomeController {
3
4     @GetMapping("/welcome")
5     public String greeting( @RequestParam String name, Model model) {
6         model.addAttribute("name", name);
7         return "welcome";
8     }
9 }
```

- Créez un fichier `welcome.html` dans `src/main/resources/templates` contenant le texte suivant :

```
1 Bienvenue <span th:text="${name}"></span> !
```

- Chargez la page `localhost:8080/welcome?name=votreNom`.

6.6 Tester le site

Depuis la 1^{re} année, on vous parle de l'importance de tester votre code. Un site web peut aussi se tester !

Tutoriel 6 Tester la page welcome

Pour tester que la page `welcome` affiche bien le nom donné en paramètre :

- Créez la classe de test `WelcomeControllerTest`

```
1 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
2 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
3
4 @SpringBootTest
5 @AutoConfigureMockMvc
6 public class WelcomeControllerTest {
7
8     @Autowired
9     private MockMvc mockMvc; // Permet de simuler le navigateur
10
11    @Test
12    public void testWelcomePage() throws Exception {
13        mockMvc.perform( get("/welcome?name=mcd")) // L'url à tester
14            .andExpect(status().isOk()) // La page est retournée
15            .andExpect(view().name("welcome")) // Générée à partir du template welcome
16            .andExpect(content().string(Matchers.containsString("mcd")))
17    }
18 }
```

- Lancez le test !

6.7 Déployer le site

Avoir une application qui tourne en local sur la machine, c'est bien. La rendre disponible sur un serveur, c'est mieux ! Vous trouverez plusieurs possibilités expliquées en annexe. Je vous propose de déployer votre application sur HEROKU.

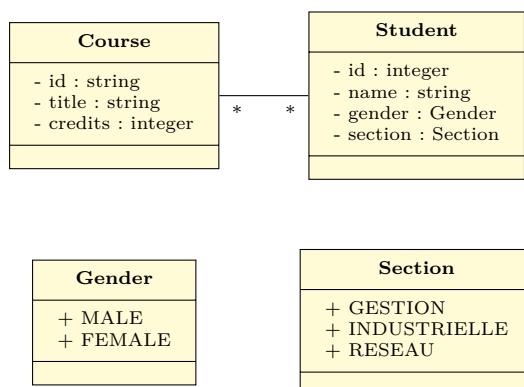
Tout au long de ce cours, nous allons illustrer nos propos au travers d'une application que nous allons construire couche après couche en appliquant les concepts vus.

7.1 Le programme annuel de l'étudiant

Cette application va permettre de visualiser et gérer les cours et les étudiants qui les suivent :

- ▷ Liste des cours ;
- ▷ Liste des étudiants ;
- ▷ Étudiants inscrits à un cours ;
- ▷ Cours faisant partie du programme d'un étudiant.

Voici le schéma des données :



Et voici quelques captures d'écran de notre version.

7.1. LE PROGRAMME ANNUEL DE L'ÉTUDIANT

Programme annuel des étudiants 

HE2B-ESI / WEBG5 / MCD / 2019-2020

[Home](#) Cours Étudiants

Liste des cours

Sigle	Titre	ECTS
INT1	Introductions	10
MAT1	Mathématiques II	3
CAI1	Anglais I	2
DEV1	Développement I	10
DEV2	Développement II	10
WEBG2	Développement web I	5

Ajouter un cours

Sigle

Titre

ECTS

[Ajouter](#)

Programme annuel des étudiants 

HE2B-ESI / WEBG5 / MCD / 2019-2020

[Home](#) Cours Étudiants

Liste des étudiants (7 au total)

Numéro	Nom	Genre	Section
1	Bulbizarre	M	Gestion
16	Roucool	F	Gestion
25	Pikachu	F	Gestion
52	Miaouss	M	Réseau
114	Saquedeneu	M	Réseau
128	Tauros	M	Industrielle
143	Ronflex	M	Gestion

Ajouter un étudiant

Nom

Genre Masculin Féminin

Section

[Ajouter](#)

Programme annuel des étudiants 

HE2B-ESI / WEBG5 / MCD / 2019-2020

[Home](#) Cours Étudiants

DEV1 - Développement I - 10 ECTS

Étudiants inscrits (2 au total)

Numéro	Nom	Genre	Section
1	Bulbizarre	M	Gestion
16	Roucool	F	Gestion

Inscrire un étudiant

id étudiant

[Ajouter](#)

Troisième partie

Une application web (Spring MVC)

Chapitre

8

Thymeleaf

Comme nous l'avons vu dans l'introduction, nous allons utiliser THYMELEAF [23] [24] [25] pour générer les pages webs à renvoyer au navigateur. Ce n'est pas la seule solution envisageable mais c'est souvent celle-là qui est choisie.

8.1 Incorporer des ressources

Les pages que nous créons doivent incorporer des ressources statiques : images, CSS, JAVASCRIPT... Avec THYMELEAF :

- ▷ Les ressources sont placées dans le dossier `src/main/resources/static`
- ▷ `@{/image/logo.jpg}` fait référence à `src/main/resources/static/image/logo.jpg`¹.

Exemple 1

Include une feuille de style

Pour inclure la feuille de style `src/main/resources/static/css/style.css`

```
<link rel="stylesheet" th:href="@{/css/style.css} />
```

Remarquez l'utilisation de l'attribut `th:href` au lieu de `href` ! Il y a aussi `th:src` pour les images et le JAVASCRIPT.

Exercice 1

PAE – Page d'accueil

Construisons une page d'accueil pour le site de gestion du programme des étudiants que nous allons développer tout au long du cours. Le résultat ressemblera à ceci :



1. Créez un projet spring comme vu en introduction : `groupId=<votreLogin>.webg5, artifactId=pae`.
2. Créez, dans le sous-package `web`, un contrôleur pour la page d'accueil.
3. Développez une page d'accueil faisant référence à un logo et une page de style.
Respectez les bonnes pratiques apprises en 1^{re} : utilisation des balises structurantes (`header`, `footer`, `main...`), utilisation des layouts `grid` et `flex` pour la mise en page.

Gestion via git

Ce que vous venez de coder va être enrichi tout au long du cours. Nous vous conseillons vivement de le placer sur le *gitlab* de l'école.

1. Concrètement, il y aura une copie des ressources dans le produit final (un jar). C'est « *comme si* » on les trouvait là.

8.2 Passer une donnée à la vue

Pour que le contrôleur passe une donnée à la vue :

- ▷ La méthode du contrôleur doit posséder un paramètre `Model model`.
- ▷ Dans la méthode, on enregistre la donnée via `model.addAttribute("key", "value")`
- ▷ La vue injecte la valeur via l'attribut `th:text="${key}"` disponible dans la plupart des balises.

Exemple 2

Passer le nom de l'utilisateur

Supposons que le contrôleur sache le nom de l'utilisateur et le passe à la vue pour qu'il apparaisse sur la page. Le code du contrôleur ressemble à

```
@GetMapping("/")
public String showIndex(Model model) {
    model.addAttribute("username", "MCD");
    return "home";
}
```

Et la vue contient

```
Bienvenue <span th:text="${username}">Inconnu</span> !
```

Le texte est remplacé par la valeur.

Natural Templating

On peut regretter que l'écriture soit verbeuse. Après tout beaucoup de moteurs de template proposent une écriture compacte comme `${name}` (appelée notation *moustache*). L'argument avancé est que ça permet d'avoir des pages qui donnent un résultat même s'ils sont ouverts directement dans le navigateur (sans passer par THYMELEAF) ce qui facilite grandement le prototypage. C'est ce qu'on appelle le *natural templating*. On peut même aller plus loin dans ce sens en faisant de même avec les ressources (ex : `<link rel="stylesheet" th:href="@{/css/style.css}" href="../static/css/style.css"/>`)

Expression Thymeleaf

Ce qu'on peut mettre dans le `th:text` est plus généralement une expression standard THYMELEAF. Exemple : `th:text="|Bonjour, ${nom}|"`. Pour les détails, voir <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#standard-expression-syntax>.

Spring EL

Les expressions `${...}` peuvent contenir plus qu'un nom d'attribut du modèle. Par exemple : `${user.name}` . Elles respectent un langage qu'on appelle le **Spring EL** (*Spring Expression Language*), une variante de OGNL, le « *Object Graph Navigation Library* ».

Voir par exemple : <https://www.baeldung.com/spring-expression-language>.

Il y a de nombreux points communs avec le langage de THYMELEAF. Par exemple, ceci `"${user.isAdmin() == false}"` est équivalent à `"${user.isAdmin()} == false"`.

8.3 Itérer sur une collection

Souvent la donnée à afficher est une collection dont il faut traiter chaque élément.

```
<li th:each="itemname : ${collectionname}">
    <span th:text="${itemname}">Nom</span>
</li>
```

L'attribut `th:each` va entraîner une répétition de la balise pour chaque valeur de la liste ou, plus généralement, de l'objet itérable.

Exemple 3

Afficher une liste de noms

Supposons que le contrôleur ait créé la variable `usernames` contenant une liste de noms. On peut l'afficher via

```
<ul>
<li th:each="username : ${usernames}">
    <span th:text="${username}">nom</span>
</li>
</ul>
```

On peut également définir une variable qui va contenir des informations sur l'étape en cours : indice, pair/impair, premier ?, dernier ?...²

Exemple 4

Numérotter les lignes

Partons de la même liste que l'exercice précédent et affichons les noms dans un tableau avec un numéro d'ordre en 1^{re} colonne et le nom en seconde colonne.

```
<table>
<tr><th>Numéro</th><th>Nom</th></tr>
<tr th:each="username,iterStat : ${usernames}">
    <td th:text="${iterStat.count}">1</td>
    <td th:text="${username}">nom</td>
</tr>
</table>
```

Exercice 2

PAE - La liste des cours

Le but est de créer une page qui affiche tous les cours sous forme d'une table. Le résultat ressemblera à ceci :

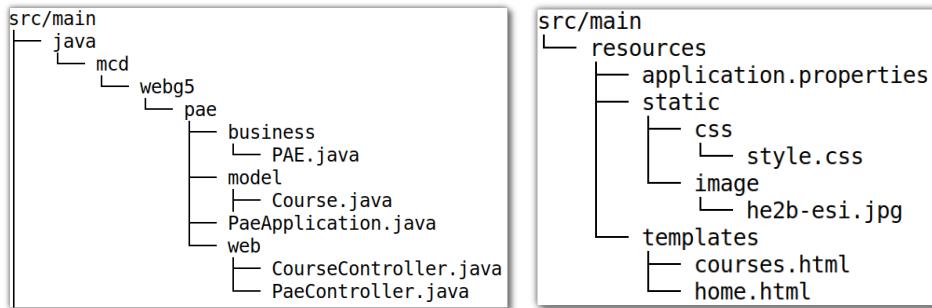
Liste des cours		
Sigle	Titre	ECTS
INT1	Introductions	10
MAT1	Mathématiques II	3
CAI1	Anglais I	2
DEV1	Développement I	10
DEV2	Développement II	10
WEBG2	Développement web I	5

- Créez, dans le sous-package `model`, une classe `Course` en vous aidant de LOMBOK. Un cours est représenté par son id, son libellé et le nombre d'ECTS. Revoyez le schéma présenté dans le chapitre « *Étude de cas* ».
- Créez, dans le sous-package `business`, une classe PAE qui propose une méthode `getCourses()` retournant une liste de cours. Comme nous n'avons pas encore de base de données, cette liste sera codée en dur.
- Créez un contrôleur pour l'url `.../courses`. Elle envoie à la vue la liste des cours.
- Générez une vue qui affiche les cours dans un tableau.

2. cf. [24], « Keeping iteration status » pour les détails.

8.4. AFFICHAGE CONDITIONNEL

Pour info, voici les fichiers présents dans notre solution :



8.4 Affichage conditionnel

Parfois une partie de la page ne doit s'afficher que sous certaines conditions. Par exemple, le bouton de login ne doit s'afficher que si l'utilisateur n'est pas encore connecté. Parfois encore le contenu doit être différent en fonction de la valeur d'une variable. Par exemple, on affichera un message spécial si une liste est vide. Que nous propose THYMELEAF ?

Attributs liés aux conditions

- ▷ `th:if="condition"` n'affiche la balise que si le test est **vrai**.
- ▷ `th:unless="condition"` n'affiche la balise que si le test est **faux**.
- ▷ Pour un selon que :

```
<span th:switch="${student.section.name()}">
<td th:case="GESTION">Gestion</td>
<td th:case="INDUSTRIELLE">Industrielle</td>
<td th:case="RESEAU">Réseau</td>
</span>
```

Écrire une condition. Vous pouvez écrire vos tests de façon assez intuitive. Notons toutefois :

- ▷ `conditon?casVrai:casFaux` pour l'opérateur ternaire de choix.
- ▷ `#lists.isEmpty(var)` permet de savoir si la collection `var` est vide.
- ▷ `valeur?:défaut` prend une valeur par défaut si la valeur est vide (opérateur *Elvis*).

Exercice 3

PAE - Tenir compte d'une liste vide

Modifiez l'exercice précédent afin qu'un message spécial (`Aucun cours`) soit affiché à la place du tableau si la liste est vide.

8.5 Liens

Si on veut utiliser un `<a>` pour créer un lien symbolique, on a vu qu'on peut utiliser le `@` pour construire l'URL. Voici quelques cas plus complexes :

- ▷ Pour obtenir `/order?orderId=3` : `th:href="@{/order(orderId=${o.id})}"`
- ▷ Pour obtenir `/order/3/details` : `th:href="@{/order/{orderId}/details(orderId=${o.id})}"`

Exercice 4

Informations sur un cours

Faites en sorte que le sigle et le titre du cours soient cliquables et amènent vers une page spécifique à ce cours. Pour l'instant, cette page ne contiendra qu'un lien pour revenir sur la page principale.

8.6 Formulaire

Voyons comment écrire un formulaire et envoyer l'information au serveur pour traitement.

La base

Commençons par un petit formulaire avec juste un champ de saisie.

Exemple 5

Un simple formulaire

Au niveau de la page :

```
<form th:action="@{/urlCtrl}" th:object="${user}" method="post">
<input type="text" th:field="*{firstname}"/>
<input type="text" th:field="*{lastname}"/>
<button type="submit">Envoyer</button>
</form>
```

Au niveau du contrôleur :

```
@PostMapping("/urlCtrl")
public String doSomething(User user) {...}
```

Remarques :

- ▷ **th:action** contient une URL. On peut donc utiliser la notation `@{}`.
- ▷ **th:object** définit l'objet qui va recevoir tous les champs du formulaire dans ses attributs. Ce peut être un objet du modèle ou un DAO (*Data Access Object*) créé pour l'occasion. Dans les 2 cas, il doit avoir été précédemment ajouté dans le modèle. Ex : `model.add("user",new User())`.
- ▷ **th:field** indique le champ (attribut de l'objet défini via **th:object**) qui va accueillir la valeur de cet élément du formulaire.
- ▷ Au niveau du contrôleur, on déclare l'objet comme paramètre.

Exercice 5

Ajouter un cours

La page listant les cours va proposer un formulaire pour en ajouter un.

- ▷ Lors de la création de la page, n'oubliez pas d'ajouter un cours vide au modèle.
- ▷ Nous n'avons pas encore de BD où sauver le nouveau cours. Contentez-vous pour l'instant d'un message de trace avec SLF4J.

Les listes et cases à cocher

Le principe est sensiblement le même pour les autres éléments de formulaire comme les listes et cases à cocher.

Exemple 6

Cases à cocher

```
<input type="checkbox" th:field="*{nomAttribut}" th:value="valeur1" />choix1
<input type="checkbox" th:field="*{nomAttribut}" th:value="valeur2" />choix2
```

Exemple 7**Boutons radios**

```
<input type="radio" th:field="*{nomAttribut}" th:value="valeur1" />choix1
<input type="radio" th:field="*{nomAttribut}" th:value="valeur2" />choix2
```

Exemple 8**Liste**

```
<select th:field="*{nomAttribut}" >
<option th:value="valeur1">libellé1</option>
<option th:value="valeur2">libellé2</option>
<option th:value="valeur3">libellé3</option>
</select>
```

8.7 Objets prédefinis

THYMELEAF met à disposition toute une série d'objets prédefinis qui peuvent se révéler fort utiles. Ils sont tous utilisés via le symbole « # » [26]. Voici quelques exemples :

- ▷ **#dates** : fournit des méthodes pour manipuler les dates.
Exemple : \${#dates.format(date, 'dd/MMM/yyyy HH:mm')}.
- ▷ **#strings** : fournit des méthodes pour manipuler les chaînes.
Exemple : \${#strings.toLowerCase(name)}.
- ▷ **#arrays** : fournit des méthodes utiles pour les tableaux.
Exemple : \${#arrays.length(array)}.
- ▷ **#lists** : fournit des méthodes utiles pour les listes.
Exemple : \${#lists.size(uneListe)}.
- ▷ **#id** : permet de générer des *id* uniques.
Exemple : \${#ids.seq('someId')}.

Il y en a d'autres ! Vous pouvez les trouver ici : <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#appendix-b-expression-utility-objects>.

8.8 Les fragments

Il est possible de définir un bout de page, un *fragment*. C'est utile lorsqu'une même information se retrouve dans plusieurs pages. C'est aussi la base pour créer un site où toutes les pages se ressemblent.

Le fragment se définit ainsi :

```
<header th:fragment="header">
<h1>Programme annuel des étudiants</h1>
<h2>HE2B-ESI / WEBG5 / MCD / 2019-2020</h2>
</header>
```

Il est placé dans une page THYMELEAF classique, en général au sein d'une page complète qui peut même contenir plusieurs fragments.

Il est utilisé ainsi :

```
<header th:replace="~{fragments/main.html::header}"></header>
```

où `fragments/main.html` est le fichier dans lequel se trouve le fragment. Le fragment vient remplacer la balise qui y fait appel.

On peut même utiliser des paramètres :

```
<head th:fragment="head(titre)">
<title th:text="${titre}">titre par défaut</title>
</head>
```

L'utilisation se fait alors ainsi :

```
<head th:replace="~{fragments/main.html::head(titre='Le vrai titre')}"></head>
```

8.9 Un layout

Souvent, toutes les pages d'un site ont des éléments en commun. Pour éviter de répéter du code, on peut utiliser la technique des fragments qu'on vient de voir. Elle souffre toutefois de deux défauts :

- ▷ À chaque nouvelle page, il faut inclure toute une série de fragments sans rien oublier.
- ▷ Si on ajoute un élément commun, par exemple un menu, il faut aller l'inclure dans **toutes** les pages, ce qui est lourd et source d'erreurs.

L'approche *layout* (ou *décorateur*) [27] est différente.

- ▷ Une page *layout* définit complètement une page type.
- ▷ Chaque page va faire référence au layout en indiquant uniquement le contenu original.

Tutoriel 1 Utilisation d'un décorateur

Écrivons un exemple minimal pour montrer comment ça fonctionne.

☒ Créez un projet de base.

Ce mécanisme de décorateur n'est pas standard, il faut l'installer.

☒ Ajoutez dans le pom la dépendance suivante :
`nz.net.ultraq.thymeleaf/thymeleaf-layout-dialect`.

Créons le layout et la page qui l'utilise.

☒ Le layout (`layout.html`) est :

```
1 <!DOCTYPE html>                                         thymeleaf/demodecorator/src/main/resources/templates/layout.html
2 <html xmlns:th="http://www.thymeleaf.org"
3     xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
4 <head>
5     <title>Layout</title>
6 </head>
7 <body>
8     <header>
9         Ici vient un entête complet
10    </header>
11    <main layout:fragment="content">
12        Ce contenu sera remplacé par le vrai.
13    </main>
14 </body>
15 </html>
```

Remarquez l'utilisation du *namespace layout* introduit par la dépendance qu'on a ajoutée avant. Le `layout:fragment` indique que cette partie sera remplacée.

☒ La page (`index.html`) est :

8.9. UN LAYOUT

```
1 <!DOCTYPE html>                                         thymeleaf/demodecorator/src/main/resources/templates/index.html
2 <html xmlns:th="http://www.thymeleaf.org"
3   xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
4   layout:decorate=~{layout}">
5 <head>
6   <title>PAE</title>
7 </head>
8 <body>
9   <main layout:fragment="content">
10    Voici le vrai contenu.
11  </main>
12 </body>
13 </html>
```

Remarquez le `layout:decorate` qui indique quel est la layout. La partie `layout:fragment` indique ce qui va remplacer la partie équivalente dans le layout.

- ☞ Vous pouvez lancer l'application et vous connecter sur `localhost:8080/`³. Remarquez que le titre dans l'onglet a aussi été changé.

Exercice 6

Un layout pour nos pages

Créez et utilisez un layout pour votre application PAE.

Alternatives

Quelles sont les alternatives ?

- ▷ **JSP** : La technologie historique pour créer des pages web.
- ▷ **Mustache** : Un moteur de template à la syntaxe plus compacte que THYMELEAF.
- ▷ **FreeMarker** : Une autre alternative, gérée par APACHE.

Vous pouvez trouver une liste comparative sur wikipedia :

https://en.wikipedia.org/wiki/Comparison_of_web_template_engines

3. Pas besoin de contrôleur, par défaut, spring boot va chercher une page `index.html` (dans `static` puis `templates`).

Bean = objet au sein de java instancié par SPRING

Chapitre

9

Le contrôleur

Vous avez déjà écrit quelques contrôleurs et vous en savez l'essentiel. Développons quelques aspects qui vous auront peut-être échappés ou que nous n'avons pas encore du tout vus mais qui peuvent vous faciliter la vie [28] [29] [30].

9.1 L'annotation @Controller

L'annotation `@Controller` indique à SPRING qu'il doit **gérer** cette classe (on parle de **bean génér**é). C'est lui qui va créer et détruire les objets en fonction des besoins. D'ailleurs, vous aurez remarqué que vous n'avez écrit aucun code qui instancie explicitement un contrôleur.

Il existe d'autres annotations similaires : `@Component`, `@Service`, `@Repository`, `@Configuration`. La seule différence est sémantique ; elle indique au lecteur le but de la classe.

9.2 L'annotation @Autowired

Vous n'avez pas encore rencontré cette annotation. Si un attribut est annoté `@Autowired` [31]¹, vous demandez à SPRING de l'initialiser automatiquement. Il faut que la classe où se trouve l'attribut et la classe de l'attribut soient toutes les deux des **beans** — c'est-à-dire que les classes soient annotées par `@Controller` ou équivalent —. De plus, il faut que la classe soit instanciée par SPRING — vous aurez un `null` si vous l'instanciez explicitement —.

Exemple 1

Annoter la façade business

Pour le moment, votre contrôleur fait appel à une méthode dans une classe business pour obtenir la liste des cours. Pour ce faire, il instancie explicitement la classe. Avec l'annotation `@Autowired`, ce ne serait pas nécessaire.

En pratique, il existe plusieurs façons d'utiliser cette annotation.

1. Directement sur l'attribut.

```
public class Controller {  
    @Autowired  
    private final BusinessFacade business;  
}
```

2. Sur le constructeur.

```
public class Controller {  
    private final BusinessFacade business;  
    @Autowired  
    public Controller( BusinessFacade business ) {  
        this.business = business;  
    }  
}
```

1. Vous pourrez aussi rencontrer `@Inject` qui est quasi équivalent.

9.3. L'ANNOTATION @BEAN

Le constructeur sera appelé automatiquement par SPRING avec les bons paramètres. Cette notation est plus lourde que la précédente mais elle est plus rapide à l'exécution.

3. Sans l'annotation.

```
public class Controller {  
    private final BusinessFacade business;  
    public Controller( BusinessFacade business ) {  
        this.business = business;  
    }  
}
```

Fait la même chose que de mettre `@Autowired` au-dessus de l'attribut

Depuis la dernière version de SPRING, l'annotation sur le constructeur n'est plus obligatoire !

4. Via une annotation de LOMBOK.

```
@RequiredArgsConstructor  
public class Controller {  
    private final BusinessFacade business;  
}
```

L'annotation implique l'existence du constructeur initialisant tous les attributs finaux. On obtient ainsi la concision de la première solution et la rapidité de la seconde.

Exercice 1

Un façade gérée par Spring

Modifiez votre programme affichant la liste des cours en annotant `@Service` la façade du modèle (la classe contenant la méthode donnant la liste des cours).

9.3 L'annotation @Bean

Dans vos recherches sur Internet, vous verrez parfois cette annotation sur une méthode. Qu'est-ce qu'elle signifie ? Placée sur une méthode qui retourne un objet, dans une classe gérée de SPRING (annotée par `@Controller` ou équivalent), il va automatiquement l'exécuter pour créer un objet de ce type. On pourra dès lors y faire référence avec un `@Autowired`.

C'est utile pour une classe qu'on ne peut pas annoter parce ce qu'on ne peut pas la changer ou pour une classe qu'on voudrait construire avec un autre constructeur que celui par défaut.

Exemple 2

Injection du clavier

Par exemple le code suivant pose problème parce que SPRING ne peut pas créer `kbd` (`Scanner` n'est pas géré par lui) :

```
@Service  
public class A {  
    @Autowired  
    private Scanner kdb;  
}
```

En fait, vu que nous n'avons pas écrit à classe 'Scanner', il faut indiquer à Spring que `Scanner` est un Bean, mais la classe dans laquelle se trouve la variable de type `Scanner` doit être un Bean également. Solution :

Il suffit d'ajouter une méthode de création dans une classe gérée :

```
@Configuration  
public class WebConfig {  
    @Bean  
    public Scanner initKbd() {  
        return new Scanner(System.in);  
    }  
}
```

9.4 Les durées de vie (portée, scope)

Quand un contrôleur possède un attribut, quelle va être sa durée de vie ? Par exemple, dans le code suivant :

```
@Controller
public class WebController {

    @Autowired
    ABean abean;

    @GetMapping("/")
    public String test(Model model) {
        model.addAttribute("aBean", aBean);
        return "index.html";
    }
}
```

Lorsqu'on fait deux requêtes, aurons-nous la même version de `aBean` ou deux différentes ? On peut aller plus loin dans le questionnement : si deux classes injectent un `ABean`, sera-ce le même ou pas ?

En fait, ça dépend du *scope* du bean – défini via l'annotation `@Scope` –. Il existe notamment les scopes suivants :

singleton

(le défaut) Il n'existe qu'**une seule version** du bean dans le conteneur et c'est toujours cette version qui est injectée partout.

session

Il n'existe qu'une seule version du bean **par session**.

request

Une nouvelle instance est créée à chaque requête.

Exemple 3

Mise en évidence des scopes

Faisons une petite expérience pour illustrer les portées de vies. Partons d'une application web de base. Commençons par définir une classe qui va attribuer un numéro unique à chaque instance :

```
1  @Data
2  public class IDGenerator {
3
4      private static int count = 0;
5      private int id;
6
7      public IDGenerator() {
8          this.id = ++count;
9      }
10 }
```

Définissons à présent plusieurs instances avec des portées différentes² :

2. Ne vous préoccupez pas de l'attribut `proxyMode`.

9.4. LES DURÉES DE VIE (PORTÉE, SCOPE)

```
bean/scope/src/main/java/webg5/scope/WebConfig.java
1 import static org.springframework.beans.factory.config.ConfigurableBeanFactory.*;
2 import static org.springframework.web.context.WebApplicationContext.*;
3 @Configuration
4 public class WebConfig {
5
6     @Bean
7     @Scope(value = SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
8     public IDGenerator requestBean() {
9         return new IDGenerator();
10    }
11
12    @Bean
13    @Scope(value = SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
14    public IDGenerator sessionBean() {
15        return new IDGenerator();
16    }
17
18    @Bean
19    @Scope(SCOPE_SINGLETON)
20    public IDGenerator singletonBean() {
21        return new IDGenerator();
22    }
23
24    @Bean
25    public IDGenerator defaultBean() {
26        return new IDGenerator();
27    }
28
29 }
```

Introduisons ensuite un contrôleur pour la page d'index :

```
bean/scope/src/main/java/webg5/scope/WebController.java
1 @Controller
2 public class WebController {
3
4     @Autowired
5     IDGenerator requestBean;
6
7     @Autowired
8     IDGenerator sessionBean;
9
10    @Autowired
11    IDGenerator singletonBean;
12
13    @Autowired
14    IDGenerator defaultBean;
15
16    @GetMapping("/")
17    public String test(Model model) {
18        model.addAttribute("requestBean", requestBean.getId());
19        model.addAttribute("sessionBean", sessionBean.getId());
20        model.addAttribute("singletonBean", singletonBean.getId());
21        model.addAttribute("defaultBean", defaultBean.getId());
22        return "index.html";
23    }
24
25 }
```

Et, enfin, la page web :

```
bean/scope/src/main/resources/templates/index.html
1 <p th:text="${requestBean}" | Request: ${requestBean} | "></p>
2 <p th:text="${sessionBean}" | Session: ${sessionBean} | "></p>
3 <p th:text="${singletonBean}" | Singleton: ${singletonBean} | "></p>
4 <p th:text="${defaultBean}" | Default: ${defaultBean} | "></p>
```

Si vous lancez l'application et rafraîchissez la page plusieurs fois, vous constaterez que seuls le premier nombre change. Si vous lancez l'application sur un autre navigateur (ou en mode

privé), le 2^e sera fixe mais différent de celui de la 1^{re} fenêtre. Les deux derniers, eux, ne changent jamais.

Et c'est utile de savoir ça ? Oui ! Supposez qu'une classe métier a pour but d'aller chercher dans la BD, la liste des catégories de produits sur un site de vente. Cette liste change peu. Vous pouvez, à la 1^{re} lecture, sauver le résultat dans un attribut et le retourner les fois suivantes. Par défaut, c'est toujours la même copie qui est utilisée et les attributs sont donc préservés. Autre exemple, si vous devez stocker le nom de l'utilisateur qui est connecté, vous devrez le faire dans une classe qui a la portée *session*.

9.5 L'annotation @ModelAttribute

Lorsqu'une méthode du contrôleur a un paramètre dont le type n'est pas une classe de SPRING, il essaie de trouver cet objet dans les attributs du **Model**. Par exemple, vous avez du écrire un code comme celui-ci :

```
@GetMapping("...")  
public String add(Cours cours, Model model) {
```

Spring a tout un algorithme pour déterminer l'objet que vous voulez en fonction du nom et de la classe du paramètre. Vous pouvez aussi être explicite :

```
@GetMapping("...")  
public String add(@ModelAttribute("newCours") Cours cours, Model model) {  
    ↑ Dans l'HTML, il faudra utiliser 'th:object=<> newCours </>'
```

Où on indique explicitement le nom de l'attribut dans **Model**.

Cette annotation s'utilise également dans d'autres situations. En voici une. Parfois, toutes les méthodes d'un contrôleur doivent mettre le même attribut en modèle. On peut éviter de répéter le code grâce à une méthode annotée.

```
@ModelAttribute(name = "attributeName")  
public AttributType aName(Model model) {  
    return <attributeValue>;  
}
```

À chaque intervention du contrôleur, il va vérifier si un attribut de ce nom-là existe. Si ce n'est pas le cas, il va le créer avec la valeur donnée par la méthode associée.

9.6 L'annotation @SessionAttributes

L'annotation `@SessionAttributes("attributeName")` placée sur un contrôleur indique que l'attribut indiqué, mis dans le modèle passé à la vue aura une durée de vie plus longue (toute la session et pas seulement la requête).

On peut par exemple l'utiliser pour retenir dans les attributs donnés à la vue l'utilisateur qui est connecté, sans devoir l'y replacer à chaque fois.

9.7 Redirection

Lorsque le contrôleur a fini, il indique la vue qui doit s'afficher. Une autre possibilité est d'indiquer `redirect:/url` pour demander une **redirection**. Rappelez-vous ! Ce mécanisme va demander au navigateur de refaire une demande à l'url indiquée.

Une application pratique est après avoir traité la formulaire d'ajout d'un cours. On voudrait renvoyer vers la même page donnant la liste des cours. C'est facile avec le *redirect*.

9.8 La conversion des données

Vous savez qu'en HTML, les données sont toutes représentées par des chaînes. Ainsi, si on saisit un nombre dans un champ de saisie et qu'on soumet le formulaire, c'est une chaîne (contenant le nombre) qui arrive sur le serveur. Pourtant, on a pu déclarer le paramètre comme un entier au niveau du contrôleur. C'est parce que SPRING fait la conversion pour nous !

Il connaît beaucoup de conversions par défaut et ça suffira pour nos exemples. Pour des conversions plus poussées, consultez la documentation.

9.9 La validation des données

Ce n'est pas parce qu'une donnée a le bon type qu'elle est correcte. Par exemple, le nombre peut ne pas être dans le bon intervalle. On peut bien sûr ajouter du code dans le contrôleur pour vérifier la validité de la donnée mais il y a mieux !

On peut simplement **déclarer** les contraintes et demander à ce qu'elles soient vérifiées. C'est un sujet vaste, traité dans le chapitre suivant.

Il existe de nombreux endroits dans une application où il est nécessaire de valider des données : les champs de saisie d'une interface utilisateur, les paramètres donnés à une méthode du modèle, des données qui vont être persistées dans une base de données...

Il existe une API standard spécialisée pour ça : la *Java's Bean Validation API* (JSR-303). Et la bonne nouvelle, c'est que SPRING l'intègre [32].

Grâce à elle, on peut se contenter de **déclarer** une contrainte (ex : tel attribut est obligatoire, telle chaîne doit comporter entre 5 et 10 caractères...) et la vérification se fera automatiquement sur simple demande.

Nous allons donc voir 3 aspects :

- ▷ Comment spécifier les contraintes au niveau des données.
- ▷ Comment demander qu'une validation doit être effectuée au niveau du contrôleur.
- ▷ Comment afficher les erreurs dans les vues.

10.1 Installation

Depuis la version 2.3 de SPRING BOOT, il faut explicitement ajouter la dépendance à la validation :

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-validation</artifactId>
```

10.2 Définir les contraintes

Une série d'**annotations** prédéfinies standard permettent d'indiquer des contraintes de base sur les données. En voici quelques unes :

NotNull

La donnée ne peut pas être nulle.

NotBlank

La donnée ne peut pas être nulle ni une chaîne vide.

Min/Max

La donnée numérique ne peut pas être plus petite/grande qu'une valeur donnée.

Positive

La donnée numérique est Positive.

Future/Past

Une date doit être dans le futur/passé.

Size

Pour contraindre la taille d'un tableau, une collection ou une chaîne.

10.3. DEMANDER UNE VALIDATION

Digits

Permet de contraindre le nombre de chiffres (partie entière et décimale) d'un nombre.

Pattern

Permet d'utiliser un pattern pour restreindre les valeurs acceptées.

Voyez [33] pour une liste complète des annotations standards. Il est également possible d'en trouver d'autres, comme celles proposées par JBoss [34].

Toutes ces annotations, possèdent un attribut `message` pour indiquer le message d'erreur en cas de non respect de la contrainte.

Voici quelques exemples :

```
@NotNull @Size(min=4, max=50)
private String name;
@Min(0) @Max(20)
private int score;
@Pattern(regexp = "[A-Za-z][A-Za-z0-9]*",
message="Le login est composé de chiffres et de lettres et doit commencer par une lettre.")
private String login;
```

10.3 Demander une validation

Pour demander une validation des données du formulaire, il suffit d'annoter l'objet les contenant et d'ajouter un paramètre pour les erreurs.

```
@PostMapping("/urlCtrl")
public String doSomething(@Valid User user, Errors errors) {
    if(errors.hasErrors()) {
        return ...
    }
    ... // traitement sans erreur.
}
```

10.4 Afficher les erreurs dans la vue

Pour afficher les erreurs dans la vue, on peut utiliser un code comme celui-ci :

```
<input type="text" th:field="*{name}" th:errorclass="errorField" />
<span th:errors="*{name}" class="errorMsg">erreur</span>
```

- ▷ `th:errorclass` indique la classe CSS à donner à l'élément du formulaire s'il est erroné (par exemple, on peut l'encadrer en rouge);
- ▷ `th:errors` indique que si une erreur existe pour le champ donné (ici `name`), le message associé doit apparaître ici. Le span n'apparaîtra pas s'il n'y a pas d'erreur associée.

Exercice 1

PAE - Validation du formulaire

Vous avez déviné ce qu'on va vous demander : ajoutez de la validation au formulaire d'ajout d'un étudiant.

Ajouter un cours

Sigle	Entrez le sigle du cours..	ne peut pas être vide
Titre	Entrez le titre du cours...	ne peut pas être vide
ECTS	0	doit être strictement positif
<input type="button" value="Ajouter"/>		

Quatrième partie

Interopérabilité

Les *services web* permettent à un client et un serveur de communiquer au travers du web (via le protocole HTTP). Un service web est accessible quel que soit le système et le langage du client. Il s'agit d'un élément essentiel dans le B2B (*Business to business*). Il existe deux grandes technologies concurrentes. Nous ne verrons pas SOAP, historiquement la première, qui commence à être abandonnée. Nous allons nous concentrer sur REST (*Representational State Transfert*), une technologie plus récente qui a pris le parti de se baser uniquement sur la simplicité du protocole HTTP.

Les services web prennent également beaucoup d'importance avec le développement des applications mobiles et du cloud. La plupart des applications (FACEBOOK, GOOGLE, TWITTER...) proposent des API accessibles via le protocole REST.

11.1 Présentation

REST est un protocole client-serveur, **sans état** basé sur HTTP. Avec REST, un service est l'accès à une information élémentaire (on parle de ressources). Cette information est accessible via une URI (ex : <http://www.monsite/catalogue/cover/isbn/123-12345-75/> pour obtenir une image de la couverture de l'ouvrage dont on donne l'ISBN).

Rappelons que le protocole HTTP propose les ordres GET, POST, PUT et DELETE. Avec REST, ces ordres sont repris pour former la base d'un accès CRUD aux données (PUT=C, GET=R, POST=U et DELETE=D)

Comme il est basé sur les mécanismes de base, un service web REST peut être utilisé en indiquant simplement son URL dans la barre d'adresse du navigateur.

Un service web REST peut proposer l'information sous plusieurs formats (texte, HTML, XML, JSON...). Le format effectivement envoyé va dépendre des préférences du client.

11.2 Définir un service

Il suffit de quelques annotations pour définir un service REST [35].

```
1  @RestController
2  @CrossOrigin(origins="*")
3  @RequestMapping("/api")
4  public class HelloRest {
5      @GetMapping("/hello")
6      public String hello() {
7          return "Hello, world !";
8      }
9 }
```

rest/tuto/src/main/java/webg5/rest/tuto/HelloRest.java

où on indique qu'un appel (GET) à la ressource : <http://localhost:8080/api/hello> retournera la chaîne « Hello ! ».

11.3. PASSER DES PARAMÈTRES

L'annotation `@RestController` est proche de `@Controller`. La seule différence est qu'on indique que les méthodes vont retourner directement les réponses sans passer par une vue. C'est pourquoi le code doit vous paraître familier.

L'annotation `@CrossOrigin` permet de faire appel au service à partir d'une machine d'un autre domaine.

Tester un service web est très facile puisqu'il s'agit juste d'entrer une URL dans son navigateur.

Tutoriel 1

Bonjour le monde !

Commençons par un classique *Hello, world*.

- ☒ Créez un nouveau projet. Vous n'aurez pas besoin de la dépendance à THYMELEAF.
- ☒ Ajoutez une classe JAVA avec le code donné dans l'exemple ci-dessus.
- ☒ Lancez le projet.
- ☒ Testez dans un navigateur en entrant l'URL : <http://localhost:8080/api/hello>
- ☒ Testez aussi via un site dédié comme : <https://resttesttest.com>.
- ☒ Testez également en ligne de commande [36] :

```
[~] curl "http://localhost:8080/api/hello"
Hello, world !
```

11.3 Passer des paramètres

Les méthodes fournissant le service peuvent accepter des paramètres qui proviennent de diverses sources :

- ▷ Les paramètres de l'URL.
Exemple : L'URL pourrait être `.../cover?isbn=123-12345-75`.
- ▷ Un paramètre *embarqué* dans l'URL. Cette technique est préférée à la précédente pour les services webs.
Exemple : Une méthode peut récupérer l'ISBN dans `...cover/isbn/123-12345-75`.
- ▷ Le contenu des champs de formulaire.

Tutoriel 2

Paramètre lié à un paramètre de l'URL

Partons du tutoriel précédent et développons une ressource d'URL `.../hello?name=votre_nom` qui retournerait la chaîne : `Bonjour, username !`.

- ☒ Ajoutez un paramètre à la méthode `hello` :

```
@RequestParam(defaultValue = "Inconnu") String name
```

- ☒ Modifiez la méthode pour qu'elle utilise le paramètre.
- ☒ Testez !

Tutoriel 3

Paramètre embarqué dans le chemin

L'idée est à présent d'accéder à la ressource via `.../hello2/un_nom`.

- ☒ Créez une copie (`hello2`) de la méthode `hello`.
- ☒ Ajoutez un paramètre à l'annotation `@GetMapping` pour qu'elle intègre le nom de l'utilisateur :

```
("/hello2/{name}")
```

- ☒ Modifiez le paramètre de la méthode `hello2` :
- ☒ Testez !

Tutoriel 4**Paramètre lié à un champ d'un formulaire**

La valeur du paramètre est à trouver dans un champ de formulaire.

- ☒ Ajoutez la méthode `post` suivante :

```
@PostMapping("/hello")
public String helloPost(String name) {
    return "Hello, " + name + " !";
}
```

- ☒ Testez avec la commande :

```
[~] curl --request POST "http://localhost:8080/api/hello" --data "name=MCD"
Hello, MCD !
```

11.4 Statut de la réponse

Il arrive qu'il soit impossible de satisfaire à la requête. Par exemple, si on demande des informations sur un cours qui n'existe pas. La bonne idée est d'utiliser le mécanisme de statut de réponse prévu dans le protocole HTTP lui-même. C'est facile grâce à la classe `ResponseEntity`. Voici un code qui renvoie un statut en plus de la réponse.

```
1   public ResponseEntity<String> hello(@RequestParam String name) {
2       if ("mcd".equals(name)) {
3           return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
4       } else {
5           return new ResponseEntity<>("Hello, " + name + " !", HttpStatus.OK);
6       }
7   }
```

On peut également définir un code qui répond à toute URL invalide grâce à une syntaxe spéciale pour l'URL : `@GetMapping("/**")`.

11.5 Rappel JSON

Pour l'instant, nos services renvoient du simple texte. Dans un cas concret, on doit souvent envoyer une information complexe qu'il s'agit de structurer.

Pour rappel, JSON (Java Script Object Notation) est un format qui permet de représenter des données sous forme d'un texte simple et lisible. Concurrent de XML, il est de plus en plus utilisé pour échanger des données, notamment lorsqu'il s'agit de sérialiser des objets dans le cadre d'un service web.

Ci-contre, un exemple de fichier JSON : il est composé d'objets (délimités par `{}`) et de tableaux (délimités par `[]`). Ces deux structures sont composées d'éléments de la forme « clé-valeur », séparés par une virgule, où une clé est une chaîne et une valeur peut-être une entier, une chaîne, `true`, `false`, `null` ou, récursivement, un objet ou un tableau.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": [
    {
      "streetAddress": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postalCode": "10021"
    },
    "phoneNumber": [
      {
        "type": "home",
        "number": "212 555-1234"
      },
      {
        "type": "fax",
        "number": "646 555-4567"
      }
    ]
}
```

11.6 Format de la réponse

Une méthode d'un service web peut indiquer, via une annotation, le type de réponse fournie : texte, XML, JSON...

```
@Produces("application/json")
```

Il peut y avoir plusieurs méthodes répondant à la même requête mais fournissant leur réponse sous un format différent. Lors de l'utilisation, la méthode choisie dépendra des préférences du client (exprimées via le paramètre `Accept` du protocole HTTP).

Avec SPRING, il suffit d'ajouter une dépendance à JACKSON pour que les réponses soient converties par défaut en JSON — sans avoir besoin d'écrire explicitement le `@Produces`.

Tutoriel 5 Réponse JSON

On va donner un message et une date au format JSON.

- ☒ Ajoutez la dépendance à JACKSON

```
1 <dependency>
2   <groupId>com.fasterxml.jackson.core</groupId>
3   <artifactId>jackson-databind</artifactId>
4 </dependency>
```

rest/tuto/pom.xml

- ☒ Créez la classe `Info` suivante :

```
1 @Data
2 @AllArgsConstructor
3 public class Info {
4     private String message;
5     private Date date;
6 }
```

rest/tuto/src/main/java/webg5/rest/tuto/Info.java

- ☒ Créez le service web suivant :

```
1 @RestController
2 @RequestMapping(path="/api/hellojson")
3 public class HelloRestJSON {
4     @GetMapping
5     public Info hello() {
6         return new Info("Hello, world !", new Date());
7     }
8 }
```

rest/tuto/src/main/java/webg5/rest/tuto/HelloRestJSON.java

- ☒ Testez !

Récursion

Si vos objets font référence l'un à l'autre de façon circulaire (par exemple, un cours contient la liste des étudiants inscrits et un étudiant contient la liste des cours auxquels il est inscrit) alors la conversion en JSON va se planter. Pour résoudre ce problème vous pouvez utiliser les annotations `@JsonManagedReference` et `@JsonBackReference` [37].

11.7 Utiliser un service

Pour consommer un service, SPRING fournit la classe `RestTemplate` qui rend le code vraiment simple. Par exemple :

```
RestTemplate restTemplate = new RestTemplate();
Info info = restTemplate.getForObject("http://localhost:8080/api/hello", Info.class);
```

11.8 Exercices

Exercice 1 Liste des cours

Développez un service web pour l'application PAE permettant d'obtenir la liste des cours.

Exercice 2 Un Client

Écrivez un client Java qui fait appel au service web pour obtenir la liste des cours et l'affiche.

- ▷ Concrètement, vous pouvez créer une classe avec une méthode principale qui ne contiendra que quelques lignes.
- ▷ Par facilité, ne parcourez pas la liste reçue ; affichez-la directement.
- ▷ Modifiez votre client pour utiliser le service web d'un autre étudiant.

Il faut, pour cela, travailler sur le même réseau que l'autre élève

Lancer le client :

1. Lancer le serveur à accéder par le client
2. Exécuter : mvn compile exec:java -Dexec.mainClass="webg5.App"

Cinquième partie

Application mono page

12.1 Rappel

Les applications web telles que développées jusqu'à présent dans ce cours, sont appellées **Applications multi pages** (*MPA, Multi Pages Application*). Lorsque l'utilisateur effectue une action sur une page, une requête est envoyée au serveur qui retourne une **page complète**.

Bien souvent, il y a de nombreux éléments communs entre les pages. Afin de rendre le rendu plus fluide, on trouve souvent des applications web qui sont développées suivant le principe d'**Application mono page** (*SPA, Single Page Application*). Dans ce cas, le serveur retourne un **fragment de page** qui est **incorporé** à la page courante.

Dans ce chapitre, nous allons voir une application qui retourne un fragment de page créée via THYMELEAF et qui est incorporée via AJAX.

Dans le chapitre suivant, nous reprendrons le même exemple mais le serveur retournera des données en JSON qui seront incorporées toujours via AJAX.

12.2 Incorporer un fragment Thymeleaf

Dans cette configuration, les pages sont créées via THYMELEAF. Du côté client, une action de l'utilisateur lance un appel AJAX qui va incorporer tout le fragment retourné par le serveur.

Tutoriel 1

Calcul du BMI – le service

Illustrons notre propos en développant un petit service web qui permet de calculer le BMI (*Body Mass Index*) et la corpulence d'un individu.

Calcul IMC

Entrez vos informations.

Homme Femme

Votre taille (en cm) :

Votre poids (en kg) :

Calcul IMC

Entrez vos informations.

Homme Femme

Votre taille (en cm) : 177

Votre poids (en kg) : 69

Votre IMC = 22.0

Votre corpulence = corpulence normale

- ▷ Le service web répondra à l'URL : `/api/bmi?height=...&weight=...&gender=....`
- ▷ Une page principale, écrite avec THYMELEAF propose un formulaire.
- ▷ Le bouton initie un traitement AJAX qui va faire appel au service et incorporer la réponse.

Voyons comment le faire étape par étape.

12.2. INCORPORER UN FRAGMENT THYMELEAF

Créez une nouvelle application SPRING.

Ajoutez une classe permettant de calculer le *bmi* :

```
1  @Service
2  public class BMIService {
3
4      public double computeBMI(int tailleCM, int poids) {
5          double tailleM = tailleCM / 100.0;
6          double bmi = poids / (tailleM * tailleM);
7          return Math.round(bmi*10)/10.0;
8      }
9
10     public String computeCategory(double bmi, String gender) {
11         if ("woman".equals(gender)) {
12             bmi++;
13         }
14         if (bmi < 16.5) {
15             return "famine";
16         }
17         if (bmi < 18.5) {
18             return "maigreux";
19         }
20         if (bmi < 25) {
21             return "corpulence normale";
22         }
23         if (bmi < 30) {
24             return "surpoids";
25         }
26         if (bmi < 35) {
27             return "obésité modérée";
28         }
29         if (bmi < 40) {
30             return "obésité sévère";
31         }
32         return "obésité morbide";
33     }
34
35 }
```

Ajoutez le contrôleur REST suivant :

```
1  import lombok.extern.slf4j.Slf4j;
2
3  /**
4   * BMIRest
5   */
6  @RestController
7  @Slf4j
8  public class BMIRest {
9
10     @Autowired
11     BMIService bmiService;
12
13     @GetMapping(value = "/api/bmi", produces = "text/html; charset=UTF-8")
14     public ModelAndView bmi(Model model,
15         @RequestParam int height,
16         @RequestParam int weight,
17         @RequestParam String gender) {
18         log.info("Rest: calcul du BMI");
19         double bmi = bmiService.computeBMI(height, weight);
20         String corpulence = bmiService.computeCategory(bmi, gender);
21         model.addAttribute("bmi", bmi);
22         model.addAttribute("corpulence", corpulence);
23         return new ModelAndView("report::content");
24     }
25
26 }
```

Si nous avions simplement retourné le nom de la vue comme on le fait d'habitude, le contrôleur aurait cru qu'il s'agit du texte à retourner. Ici, nous devons explicitement faire appel à

THYMELEAF via la classe ModelAndView.

- Vous pouvez déjà tester avec votre navigateur que le service fonctionne correctement : <http://localhost:8080/api/bmi?height=170&weight=70&gender=male>

Au niveau des pages, il faut donner la page principale et le fragment à incorporer.

- Voici la page principale :

```

1 <!doctype html>
2
3 <head>
4   <meta charset="utf-8">
5   <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.js"></script>
6   <script type="text/javascript">
7     $(document).ready(function () {
8       $("#compute").click(
9         function () {
10           let poids = $("#poids").val();
11           let taille = $("#taille").val();
12           let genre = $("input[name=genre]:checked").val();
13           let url = "http://localhost:8080/api/bmi?"
14             + "weight=" + poids
15             + "&height=" + taille
16             + "&gender=" + genre;
17           $("#report").load(url);
18         }
19       );
20     });
21   </script>
22 </head>
23
24 <body>
25   <h1>Calcul IMC</h1>
26   <p>Entrez vos informations.</p>
27   <input type="radio" name="genre" value="man" checked> Homme
28   <input type="radio" name="genre" value="woman"> Femme<br/>
29   Votre taille (en cm) : <input id="taille" value="177" size="10" /><br/>
30   Votre poids (en kg) : <input id="poids" value="69" size="10" /><br/>
31   <button id="compute">Calculer</button><br/>
32   <br/>
33   <div id="report">
34   </div>
35 </body>
36 </html>
```

La fonction JAVASCRIPT qui est lancée lorsqu'on clique sur le bouton va : lire les valeurs dans les champs de saisie, construire l'url et charger¹ la page retournée par l'appel AJAX.

- Et voici le fragment qui sera incorporé.

```

1 <!DOCTYPE html>
2 <head>
3 </head>
4 <body>
5   <div th:fragment="content">
6     Votre IMC = <span th:text="${bmi}">??</span>
7     <br/>
8     Votre corpulence = <span th:text="${corpulence}">??</span>
9   </div>
10 </body>
11 </html>
```

1. On simplifie le code par un appel à `load`. On pourrait écrire explicitement l'appel AJAX si on devait gérer les erreurs.

Dans le chapitre précédent, nous avons utilisé AJAX tout en continuant à produire les (fragments de) pages sur le serveur.

Une tendance actuelle est d'aller plus loin encore dans l'utilisation des services REST : le serveur retourne les données en JSON et c'est le client qui se charge de les incorporer dans la page (et donc, vraisemblablement, de produire du HTML à la volée). C'est plus lourd au niveau du JAVASCRIPT mais on obtient une meilleure indépendance entre le *front-end* et le *back-end*.

Voyons comment cela fonctionne : d'une part avec un code JAVASCRIPT/JQUERY pur ; d'autre part en utilisant le framework VUE.JS.

13.1 Un client JavaScript

Avec cette approche, on n'utilise pas de framework dédié¹ mais on code tout en JAVASCRIPT (aidé de JQUERY). Le service web qu'on vient de développer peut être utilisé via une application client écrite purement en JAVASCRIPT.

Tutoriel 1

Calcul du BMI – client JavaScript pur

Reprenons la même application que dans le chapitre précédent. On pourrait écrire le *front-end* dans un autre projet mais nous allons tout regrouper par facilité.

- Prenez une copie de l'application BMI utilisant THYMELEAF.
- Supprimez la dépendance à THYMELEAF Ajoutez la dépendance à JACKSON

```
1 <dependency>
2   <groupId>com.fasterxml.jackson.core</groupId>
3   <artifactId>jackson-databind</artifactId>
4 </dependency>
```

[js/bmi/pom.xml](#)

- Comme on ne retourne plus une vue mais des données, il faut les stocker dans un objet.

```
1 @NoArgsConstructor
2 @AllArgsConstructor
3 public class BMIResponse {
4   private double bmi;
5   private String corpulence;
6 }
```

[js/bmi/src/main/java/webg5/js/bmi/BMIResponse.java](#)

- Le contrôleur REST change un peu puisqu'on met la réponse dans l'objet dédié. Comme on a ajouté une dépendance à JACKSON, SPRING BOOT comprend qu'il doit transformer la réponse en JSON.

1. Ce qui n'est probablement pas une bonne idée. Les frameworks facilitent grandement la tâche de développement et permettent d'aboutir à des produits plus robustes.

13.1. UN CLIENT JAVASCRIPT

@Gett

```

1  @RestController
2  @RequestMapping("/api/bmi")
3  public class BMIRest {
4
5      @Autowired
6      private BMIService bmiService;
7
8      @GetMapping
9      public BMIResponse bmi(
10          @RequestParam int height,
11          @RequestParam int weight,
12          @RequestParam String gender) {
13          double bmi = bmiService.computeBMI(height, weight);
14          String corpulence = bmiService.computeCategory(bmi, gender);
15          return new BMIResponse(bmi, corpulence);
16      }
17  }

```

js/bmi/src/main/java/webg5/js/bmi/BMIRest.java

Vous pouvez déjà tester avec votre navigateur que le service fonctionne toujours (vous recevrez les données JSON) : <http://localhost:8080/api/bmi?height=170&weight=70&gender=male>

La page principale est légèrement modifiée pour incorporer les données reçues à la page.

```

1  <!doctype html>
2  <head>
3      <meta charset="utf-8">
4      <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.js"></script>
5      <script type="text/javascript">
6          $(document).ready(function () {
7              $("#compute").click(
8                  function () {
9                      var poids = $("#poids").val();
10                     var taille = $("#taille").val();
11                     var genre = $("input[name=genre]:checked").val();
12                     $.ajax({
13                         url: "http://localhost:8080/api/bmi?"
14                             + "weight=" + poids
15                             + "&height=" + taille
16                             + "&gender=" + genre
17                     }).then(function (data) {
18                         var bmi = data.bmi;
19                         var corpulence = data.corpulence;
20                         $("#imc").text(bmi);
21                         $("#corpulence").text(corpulence);
22                     });
23                 });
24             });
25         </script>
26     </head>
27     <body>
28         <h1>Calcul IMC</h1>
29         <p>Entrez votre poids et votre taille.</p>
30         <input type="radio" name="genre" value="man" checked> Homme
31         <input type="radio" name="genre" value="woman"> Femme<br/>
32         Votre taille (en cm) : <input id="taille" value="177" size="10"/><br/>
33         Votre poids (en kg) : <input id="poids" value="69" size="10"/><br/>
34         <input id="compute" type="button" value="Calculer"/><br/><br/>
35         Votre IMC = <span id="imc"></span><br/>
36         Votre corpulence = <span id="corpulence"></span>
37     </body>
38 </html>

```

js/bmi/src/main/resources/static/index.html

Testez : <http://localhost:8080/>

13.2 Vue.js

VUE.JS [38] est un framework JAVASCRIPT développé par EVAN YOU très à la mode pour le moment² pour développer la partie cliente des applications web. Il est open-source.

Tutoriel 2 Calcul du BMI via Vue.js

Pour le mettre en pratique dans notre application de BMI, ce n'est pas compliqué.

- Ajoutez une alternative à la page d'accueil, `indexvue.html` :

```

1 <!doctype html>                                         js/bmi/src/main/resources/static/indexvue.html
2
3 <head>
4   <script src="http://code.jquery.com/jquery-latest.min.js"></script>
5   <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
6   <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
7 </head>
8
9 <body>
10  <h1>Calcul IMC</h1>
11  <p>Entrez votre poids et votre taille.</p>
12
13 <div id="requestData">
14   <input type="radio" id="genreM" v-model="genre" value="man">
15   <label for="genreM">Homme</label>
16   <br>
17
18   <input type="radio" id="genreW" v-model="genre" value="woman">
19   <label for="genreW">Femme</label>
20   <br>
21
22   Votre taille (en cm) :
23   <input type="number" v-model="taille" />
24   <br>
25
26   Votre poids (en kg) :
27   <input type="number" v-model="poids" />
28   <br>
29
30   <button id="compute" v-on:click="computeBMI">Calculer</button>
31   <br>
32 </div>
33
34 <div id="responseData" v-if="imc > 0">
35   Votre IMC = {{ imc }}
36   <br>
37   Votre corpulence = {{ corpulence }}
38 </div>
39
40   <script src="bmi.js"></script>
41 </body>
42
43 </html>
```

La route n'est pas bonne. Il faut le mettre dans .../templates/indexvue.html
ET créer un controller qui pointera vers ce template

- La page fait appel au code JAVASCRIPT suivant :

2. REACT et ANGULAR sont également très appréciés.

13.2. VUE.JS

```
1 var requestData = new Vue({
2   el: '#requestData',
3   data: {
4     genre: "man",
5     taille: 177,
6     poids: 69
7   },
8   methods: {
9     computeBMI: function () {
10       url = "http://localhost:8080/api/bmi?"
11         + "weight=" + requestData.poids
12         + "&height=" + requestData.taille
13         + "&gender=" + requestData.genre;
14       console.log("Appel Rest: " + url);
15       axios.get(url)
16         .then(function (response) {
17           data = response.data;
18           responseData.imc = data.bmi;
19           responseData.corpuulence = data.corpuulence;
20         })
21         .catch(function (error) {
22           alert("Erreur appel REST");
23         });
24     }
25   }
26 });
27
28 var responseData = new Vue({
29   el: '#responseData',
30   data: {
31     imc: 0,
32     corpuulence: ""
33   }
34 });
```

Où on utilise l'AXIOS pour l'appel AJAX.

☒ Et c'est tout ! Testez : <http://localhost:8080/indexvue.html>

Exercice 1 Un PAE mono page

À coté de la version actuelle, faites une version mono-page de l'application. Je vous laisse le choix de la technologie (THYMELEAF, JAVASCRIPT sans framework ou VUE) mais je vous conseille VUE.

Au début, la page ne présente que la liste déroulante des cours, construite au lancement de la page via du AJAX.

The screenshot shows the main page of the "Programme annuel des étudiants". At the top left is the HE2B ESI logo. The title "Programme annuel des étudiants" is centered at the top. Below the title, a message says "Bienvenue dans le site de gestion du programme annuel des étudiants.". A dropdown menu labeled "Choisissez un cours..." is visible. At the bottom right, there is a footer note: "HE2B-ESI / WEBG5 / MCD / 2020".

Lorsque l'utilisateur choisit un cours, le détail apparaît en-dessous (via un appel AJAX à un service web que vous allez devoir écrire pour l'occasion).

The screenshot shows the same page after a course has been selected. The dropdown menu now shows "DEV1 - Développement I". Below it, the course details are displayed: "Sigle : DEV1", "Intitulé : Développement I", and "Nb de crédits : 10". The footer note "HE2B-ESI / WEBG5 / MCD / 2020" is also present.

Sixième partie

Persistir les données (Spring JPA)

Vos programmes créent des objets. Une fois le programme terminé, ces objets sont détruits. La **persistence des données** est la propriété d'exister au-delà de la durée de vie du programme. Cela demande évidemment de sauver l'objet pour une relecture ultérieure. Quelles solutions peut-on envisager ?

Sérialisation

Vous avez probablement pratiqué cette technique au laboratoire Java en 1^{re} année. Ce mécanisme permet de transformer un objet en une suite d'octets. Il peut ainsi être sauvé dans un fichier. Plus tard, il pourra être relu et désérialisé. Cela ne peut être envisagé que pour une petite quantité de données ou des données faiblement partagées.

JDBC

Cette API permet d'accéder à des BD via des ordres SQL. C'est la solution que vous avez utilisée en deuxième année. Elle est assez lourde car il faut ajouter, pour chaque classe persistante, du code pour assurer le dialogue avec la BD. Ce code est long et répétitif.

Mapping objet-relationnel

Le code JDBC étant assez répétitif on peut automatiser sa production. Un système de **ORM** (*Objet-Relational Mapping*) automatise les tâches de synchronisation que vous devez coder manuellement avec la technique précédente.

Ce chapitre propose un rappel JDBC puis introduit les fondations qui vont permettre, dans les autres chapitres, de voir ce que SPRING offre comme API de plus haut niveau pour simplifier la tâche de persistance.

14.1 Rappel JDBC

Commençons en douceur par un rappel de ce que vous avez déjà codé pour persister des données.

```
public class FirstExample {  
    static final String DB_URL = "jdbc:mysql://localhost/EMP";  
    static final String USER = "username";  
    static final String PASS = "password";  
  
    public static void main(String[] args) {  
        Connection conn = null;  
        Statement stmt = null;  
        try{  
            //Open a connection  
            conn = DriverManager.getConnection(DB_URL,USER,PASS);  
  
            //Execute a query  
            stmt = conn.createStatement();  
            String sql;  
            sql = "SELECT id, first, last, age FROM Employees";  
            ResultSet rs = stmt.executeQuery(sql);  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            if (stmt != null)  
                try {  
                    stmt.close();  
                } catch (SQLException se) {}  
            if (conn != null)  
                try {  
                    conn.close();  
                } catch (SQLException se) {}  
        }  
    }  
}
```

```

//Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
}

//Clean-up environment
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}
}

```

Beaucoup de code pour juste obtenir une liste des employés !

14.2 H2, un SGBD embarqué

Nous pourrions persister nos données avec n'importe quel SGBD. Nous allons voir H2 [39] qui est un **SGBD**

- ▷ **embarqué** : tout comme notre serveur il est démarré par l'application ;
- ▷ **en mémoire** : il peut fonctionner dans un mode où tout est stocké en mémoire vive. En phase de développement, c'est rapide et suffisant.

Plus tard, nous verrons comment sauver sur disque ou utiliser un SGBD relationnel plus classique.

La bonne dépendance

H2 n'étant pas dans les dépendances par défaut, il faut l'ajouter dans le POM.

```

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

```



```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

```

Le gérer

Comme il est embarqué, il n'est exécuté que lorsqu'on lance l'application ? Comment le gérer ? Facile ! Lorsque *Spring DevTools* est activé (et il l'est !), il propose une console de gestion à l'URL <http://localhost:8080/h2-console>.

Exercice 1

Installer H2

Ajoutez la dépendance à H2 et vérifier que vous pouvez visualiser la console dans un navigateur. La valeur du paramètre JDBC-URL peut être lu dans les traces (il commencera par JDBC-URL=jdbc:h2:mem:) et User-Name=sa. À ce stade, vous ne pourrez pas encore aller plus loin.

Fixer le nom

À chaque lancement, il choisit un nouveau nom pour la bd. On peut fixer ce nom dans le fichier de configuration `application.properties` grâce à la ligne

`spring.datasource.url=jdbc:h2:mem:testdb`

C'est un fichier que java interprétera comme une hashmap. Donc, c'est une logique de clef-valeur

JPA (*Java Persistence API*) est une API standard (JSR-338) pour la persistance des données. Elle permet de ne plus (ou presque) devoir écrire d'ordre SQL pour accéder à la base de données. Cette API est le fruit du travail conjoint de plusieurs sociétés qui proposaient déjà des produits équivalents : HIBERNATE, TOPLINK, ECLIPSELINK. En fait, il s'agit d'une norme implémentée par plusieurs *providers*. Nous utiliserons HIBERNATE dans ce cours car c'est le choix par défaut avec SPRING.

Spring JPA [40] une couche pour encore simplifier les opérations en fournissant bon nombres de méthodes toutes faites pour les opérations classiques sur la BD.

15.1 Une entité

Dans le vocabulaire de JPA, une **entité** est un objet qui est persisté comme ligne dans une table d'une base de données relationnelle.

Une entité se présente comme une classe Java classique à laquelle on a ajouté des annotations pour préciser certains points. Par exemple :

```
@Entity           Ne pas appeler la classe 'User' car ce nom est réservé au SGBD
public class User {                               Solution : spring.datasource.url=jdbc:h2:mem:testdb;NON_KEYWORDS=USER
    @Id
    private String login;
    private String name;
    // Constructeurs, accesseurs (explicites ou via Lombok)
}
```

Ainsi, on précise qu'il s'agit bien d'une entité (`@Entity`) et quelle est la clé primaire (`@Id`). Certaines valeurs sont déduites par défaut du code¹ (ex : nom de la table = nom de la classe) mais peuvent être configurées via d'autres annotations. Nous consacrerons tout le prochain chapitre à ce sujet.

On obtient ainsi une véritable vue OO de la BD. Ainsi, si deux tables sont liées de telle sorte qu'un utilisateur est associé à plusieurs factures, alors la classe `User` contiendra un attribut avec la liste des factures. Ou, pour voir les choses dans l'autre sens, on programme directement au niveau des objets (du MCD) et les tables (le modèle physique) sont déduites automatiquement.

15.2 Spring JPA

SPRING-JPA fournit par exemple une méthode `findAll()` qui va automatiquement :

1. Générer un ordre SQL pour aller chercher tous les utilisateurs dans la table ;
2. Parcourir le résultat, transformer chaque ligne en un `User` ;
3. Mettre le tout dans un itérable et le renvoyer.

1. On parle de « programmation par convention ». (ou encore de « configuration par exception »)

Tutoriel 1**Démo**

Dans ce tutoriel, vous allez consulter une table d'utilisateurs et sauver un utilisateur dans la table.

Créez un nouveau projet (THYMELEAF ne sera pas nécessaire ce coup-ci).

Ajoutez les dépendances à H2 et SPRING-JPA :

```
jpa/demo/pom.xml
1 <dependency>
2   <groupId>com.h2database</groupId>
3   <artifactId>h2</artifactId>
4   <scope>runtime</scope>
5 </dependency>
6 <dependency>
7   <groupId>org.springframework.boot</groupId>
8   <artifactId>spring-boot-starter-data-jpa</artifactId>
9 </dependency>
```

A la compilation (et non) à l'exécution, il ne téléchargera pas cette dépendance mais le fera à l'exécution

Créez une entité définissant un utilisateur avec juste un login et un nom.

```
jpa/demo/src/main/java/webg5/jpa/demo/User.java
1 @Entity
2 @Data
3 @NoArgsConstructor
4 @AllArgsConstructor
5 public class User {
6
7   @Id
8   private String login;
9   private String name;
10 }
```

Le constructeur sans argument est obligatoire pour une entité. Celui avec tous les arguments est là pour notre confort.

Créez une interface pour gérer l'entité.

```
jpa/demo/src/main/java/webg5/jpa/demo/UserDB.java
1 public interface UserDB extends CrudRepository<User, String> {
2 }
```

L'interface `CrudRepository` doit être paramétrée avec la classe des entités à gérer et le type de la clé primaire. Elle offre alors de nombreuses méthodes. Et le mieux dans tout ça ? Il n'est **pas nécessaire d'écrire l'implémentation**, SPRING-JPA le fait pour nous !

Afin de nous concentrer sur JPA, nous n'allons pas passer par une application web mais écrire un code qui va s'exécuter directement sur le serveur.

Créez la classe `Hello` suivante

```
jpa/demo/src/main/java/webg5/jpa/demo/Hello.java
1 @Slf4j
2 @Component
3 public class Hello implements CommandLineRunner {
4
5   @Override
6   public void run(String... args) {
7     log.info("Hello, world !");
8   }
9 }
```

SPRING exécute automatiquement la méthode `run()` de toute classe qui implémente `CommandLineRunner`.

Lancez le projet ; vous devriez voir le message dans la console.

Complétons la classe. Elle devient :

```
jpa/demo/src/main/java/webg5/jpa/demo/Demo.java
1  @Autowired
2  private UserDB userDB;
3
4  @Override
5  public void run(String... args) {
6      log.info(userDB.findAll().toString());
7  }
8 }
```

Pour rappel, l'annotation `@Autowired` indique à SPRING qu'il doit instancier cet attribut pour nous.

Le projet se relance automatiquement et vous verrez la modification dans la console. Bon ok, la liste est vide ! Ce qui est normal. Si SPRING trouve un fichier `data.sql` à la base du dossier des ressources, il l'exécute au démarrage. Idéal pour remplir les tables en phase de développement.

- Dans le dossier `src/main/resources`, créez le fichier `data.sql` que voici :

```
jpa/demo/src/main/resources/data.sql
1 insert into User (login, name)
2   values ('JLC', 'Lechien'),
3          ('MCD', 'Codutti');
```

L'application se relance automatiquement et vous pouvez constater le résultat.

Pour terminer ce tutorial, tentons de sauver un nouvel utilisateur.

- Modifiez le code de la méthode `run` :

```
jpa/demo/src/main/java/webg5/jpa/demo/Demo.java
1 log.info(userDB.findAll().toString());
2 userDB.save( new User("PBT", "Bettens") );
3 log.info(userDB.findAll().toString());
```

Constatez le résultat !

Fin Tutoriel 1

Exercice 1

Persistez les cours

Reprenez l'application du programme annuel des étudiants qui affiche la liste des cours et modifiez-le afin que :

1. Les données soient dans une base de données et plus codées en dur dans le code ;
2. La base de données contienne quelques valeurs au départ ;
3. L'ajout d'un cours se fasse réellement dans la BD.

Approfondissons nos connaissances sur les entités.

16.1 Définir la base ou les entités ?

On a vu que SPRING peut déduire et même créer la BD à partir des entités. À l'inverse, certains outils¹ peuvent créer les entités à partir des tables. Que faire en pratique ?

Dans le cadre d'un petit projet, avec peu de données, il est probablement plus judicieux de définir les entités et de laisser le serveur créer les tables. On peut arrêter l'analyse au MCD sans se préoccuper de l'implémentation physique.

Dans le cadre d'un projet plus conséquent, il est probable que les tables soient préexistantes. Et même dans le cas contraire, les gestionnaires de BD voudront garder le contrôle sur les tables créées. Dans ce cas, à vous d'écrire les entités à la main ou via un outil de génération automatique.

Attention ! L'utilisation d'un outil automatique de génération du code nécessite quand même de bien comprendre ce qui est généré car il y a souvent des petites modifications à y apporter. C'est ce à quoi nous allons nous attacher dans les sections suivantes.

16.2 Traces

JPA fait beaucoup de choses pour nous. Quand ça ne va pas comme on le pense, ou simplement pour comprendre, on peut lui demander d'afficher des traces afin d'avoir une idée plus précise de ce qu'il fait réellement (par exemple pour visualiser les requêtes SQL qui sont envoyées au SGBD). Pour ce faire, il suffit d'ajouter ceci dans le fichier de configuration `application.properties`.

```
logging.level.org.hibernate.SQL=debug  
logging.level.org.hibernate.type.descriptor.sql=trace
```

On peut aussi, comme alternative, utiliser le fichier `application.yml` au format YAML, plus hiérarchique. Ce qui, dans ce cas-ci, donnerait :

```
logging.level.org.hibernate  
SQL: debug  
type.descriptor.sql: trace
```

Cf. <https://fr.wikipedia.org/wiki/YAML> pour les détails de cette syntaxe.

1. Par exemple, NETBEANS en propose un. Il y a aussi le projet *JPA entity generator* : <https://github.com/smartnews/jpa-entity-generator>.

16.3 Génération automatique des clés

L'annotation `@Id` permet d'indiquer quelle est la clé primaire. Sans autre renseignement, il revient au programmeur d'attribuer une valeur unique lors de la création d'une nouvelle entité.

Dans le cas d'une identité numérique, il serait confortable de pouvoir automatiser l'attribution d'un numéro unique. C'est possible !

```
entite/sequencegeneratedvalue.java
1 @Id
2 @GeneratedValue(generator = "my_gen", strategy = GenerationType.SEQUENCE)
3 @SequenceGenerator(name = "my_gen", sequenceName = "my_seq", allocationSize = 50)
4 private long id;
```

Ici, on demande d'utiliser une *séquence* nommée `my_seq` pour générer les clés en demandant les numéros de clés par paquets de 50.

Les autres possibilités sont : `TABLE`, `IDENTITY` et `AUTO` (par défaut) qui choisit la stratégie la plus adaptée (`SEQUENCE` si possible).

Sans option, HIBERNATE utilise la stratégie `SEQUENCE` nommée `hibernate_sequence`.

```
entite/generatedvalue.java
1 @Id @GeneratedValue
2 private long id;
```

Une fois cette annotation définie, vous pouvez créer une entité avec un `id` à 0 ; il sera ajusté quand l'entité sera persistée, par exemple lors de l'appel à `save()`.

Pour la lisibilité et afin d'éviter des erreurs d'inattention, c'est probablement une bonne idée de modifier le constructeur pour que l'`id` ne soit plus passé en paramètre.

16.4 Les clés composées

`@Embeddable`

```
public class EmployeeId {
    @Id
    private String firstName;
    private String lastName;
    private String idNumber;
}
```

L'annotation `@Id` ne peut être mise que sur un seul attribut. Comment faire quand une clé primaire est composée ?

JPA propose deux solutions. Voyons-en une. Il faut regrouper les attributs composants la clé primaire dans une classe à part annotée `@Embeddable` et de remplacer dans l'entité l'annotation `@Id` par `@EmbeddedId`.

`@Entity`

```
@Entity
public class Employee {
```

16.5 Persistance des types des données

Chaque attribut d'une entité doit être synchronisé avec une colonne de la table associée, mais les types possibles pour les colonnes ne correspondent pas toujours facilement à un type Java et vice-versa. Pour des types simples (numérique, chaîne) la correspondance est plutôt facile mais quid d'attributs aux types plus complexes ?

booléen

Par défaut, JPA sauve un booléen comme un `boolean`².

Si on a quelque chose de différent dans la table (ex : un `char`), on pourrait se contenter de modifier le getter/setter pour qu'il retourne/reçoive un booléen et laisser l'attribut du type de base. Exemple :

```
public boolean isEnabled() {
    return enabled=='Y';
}
public void setEnabled(boolean enabled) {
    this.enabled = enabled ? 'Y' : 'N';
}
```

2. Pour autant que ce type existe dans le SGBD utilisé. C'est le cas pour H2.

Une meilleure solution est de demander à JPA de faire lui-même les conversions via un `@Converter` que je vous laisse découvrir [41].

énumération

Par défaut, les énumérations sont sauveées comme un entier (en commençant par 0). On peut sauver les énumérations comme des chaînes via une option de l'annotation `@Enumerated` [42].

objet

Un objet qui n'est pas une entité est sauvé comme un *blob* via la sérialisation. Idem pour les tableaux (cf. `@Lob`).

date

En SQL, il existe les types DATE, TIME et TIMESTAMP.

Si l'entité utilise une date classique (`Date` et `Calendar`), il est obligatoire d'utiliser l'annotation `@Temporal` afin de préciser ce qu'il faut sauver de la date : juste le jour (DATE), juste l'heure (TIME) ou les deux (TIMESTAMP).

Si vous utilisez la `DATETIME` API de JAVA 8, c'est plus simple : `LocalDate`, `LocalTime` et `LocalDateTime` sont mappés respectivement en DATE, TIME et TIMESTAMP.

Pour comprendre le schéma de la DB, aller la voir. Cela permettra aussi de voir pourquoi une enum ou un objet n'est pas inséré (exemple : travailler avec un type DATE en Europe n'est pas la même chose qu'un type DATE en Amérique)

Exercice 1

La liste des étudiants

Bien ! Il est grand temps de mettre en pratique tout ce qu'on vient de voir. Écrivez la page qui affiche la liste des étudiants et permet d'en ajouter un.

Matricule	Nom	Genre	Section
6	Dracaufeu	M	GESTION
19	Rattata	M	GESTION
25	Pikachu	F	GESTION
39	Rououdou	F	INDUSTRIELLE
309	Dynavolt	M	INDUSTRIELLE
378	Regice	M	RESEAU
391	fddfs	F	INDUSTRIELLE

- ▷ Par facilité, codez en multi-page.
- ▷ Créez une entité pour les étudiants mais ne faites pas encore de lien avec les cours.
- ▷ Le numéro d'étudiant est créé automatiquement. `@GeneratedValue(strategy = GenerationType.IDENTITY)` Ceci permet de générer des ids avec une valeur plus grande que les ids des tuples existants déjà dans la DB
- ▷ Attention ! Le genre et la section sont des énumérations pas des chaînes.

16.6 Les associations

Les relations entre les tables/classes présentent des différences selon le niveau auquel on se place.

Au niveau du **code Java**, toutes les relations sont dans un seul sens. Une entité possède un lien vers une autre entité via un attribut ou une collection d'attributs.

Dans un **schéma UML**, les classes peuvent être associées. de façon uni-directionnelles ou bi-directionnelles. De plus, elles se distinguent par leur multiplicité (1 – 1, 1 – N et N – M).

Au niveau du **schéma relationnel** qui décrit les tables physiques, il n'y a pas de direction. Le lien entre les entités peut être réalisé via une clé étrangère ou une table de jointure.

Cette section s'attache à vous faire comprendre comment JPA fait le lien entre les entités et les tables relationnelles.

Paramétriser les liens

Au niveau des entités, les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany` permettent d'indiquer la nature du lien et, éventuellement, de le paramétrer (chargement différé³ par exemple).

Table de jointure ou clé étrangère ?

Dans le schéma de la BD, il y a deux possibilités pour représenter une association : la table de jointure et la clé étrangère.

- ▷ Pour un lien 1 – 1, la clé étrangère est privilégiée mais on peut envisager une table de jointure.
- ▷ Pour un lien 1 – N, les deux sont possibles (table de jointure par défaut).
- ▷ Pour un lien N – M, la table de jointure est la seule possibilité.

Les annotations `@JoinTable` et `@JoinColumn` permettent d'imposer une technique et de la paramétrer.

Exemple : `@JoinColumn(nullable=false)` indique que la clé étrangère ne peut pas être nulle. On a donc bien un lien 1 – 1 et pas un lien 1 – 0..1

Lien bi-directionnel

Si le lien est bi-directionnel, un des deux côtés est le *propriétaire* de la relation. C'est au propriétaire qu'il revient de paramétrer la relation. L'autre côté va juste y faire référence via le paramètre `mappedBy`, valable pour toutes les annotations de type ...To... sauf `@ManyToOne`.

Exemple :

```
public class SaleOrder {
    @OneToMany(mappedBy="order")
    private Collection<OrderItem> items;
}

public class OrderItem {
    @ManyToOne
    private SaleOrder order;
}
```

Le paramètre de l'annotation `@OneToMany` indique que c'est l'attribut `order` de la classe `OrderItem` qui est l'autre côté de la relation bidirectionnelle de l'association.

Si on ne met pas le `mappedBy`, JPA considère qu'il s'agit de **deux** liens uni-directionnels ce qui n'implique pas la même traduction en tables. On aura, par exemple, deux tables de jointures pour une relation $N - M$.

Cohérence d'un lien bi-directionnel

JPA n'assure pas la cohérence d'un lien bi-directionnel. Reprennons l'exemple d'une commande qui contient une liste de lignes de commandes et d'une ligne de commande qui connaît la commande à laquelle elle appartient.

Au niveau du code JAVA, on peut très bien avoir une ligne de commande qui appartient à une collection `items` d'une commande mais a pour attribut `order` une autre commande. Une bonne idée est probablement d'ajouter une méthode qui ajoute une ligne de commande tout en assurant la cohérence.

```
public class SaleOrder {
    public void addOrderItem(OrderItem item) {
        this.items.add(item);
        item.setSaleOrder(this);
    }
}
```

3. Ce point sera expliqué plus loin.

Exercice 2 Compréhension des associations

Envisageons 2 cardinalités différentes ($1 - 1$ et $1 - N$) et considérons les cas uni- et bi-directionnels. On a donc 4 cas en tout. Pour chaque situation :

1. Écrivez un exemple Java minimal pour l'illustrer.
2. Quelles vont être les tables associées ?
3. Lancez l'application et vérifiez avec la console H2.
4. Si le choix par défaut est d'utiliser une table de jointure, serait-il possible d'utiliser une clé étrangère ? Pourquoi et comment ?

Exercice 3 Lien entre cours et étudiants

Dans votre application PAE, ajoutez le lien entre les étudiants et leurs cours ainsi qu'entre les cours et leurs étudiants.

Liste des cours

Sigle	Titre	ECTS
INT1	Introductions	10
MAT1	Mathématiques II	3
CAI1	Anglais I	2
DEV1	Développement I	10
DEV2	Développement II	10
WEBG2	Développement web I	5

Ajouter un cours

Sigle	Entrez le sigle du cours
Titre	Entrez le titre du cours..
ECTS	0
<input type="button" value="Ajouter"/>	

Liste des étudiants

Matricule	Nom	Genre	Section
6	Dracaufeu	M	GESTION
19	Rattata	M	GESTION
25	Pikachu	F	GESTION
39	Roudoudou	F	INDUSTRIELLE
309	Dynavolt	M	INDUSTRIELLE
378	Regice	M	RESEAU

Ajouter un étudiant

Nom	Entrez le nom de l'étudiant...
Genre :	<input checked="" type="radio"/> Homme <input type="radio"/> Femme
Section :	<input checked="" type="radio"/> Gestion <input type="radio"/> Industrielle <input type="radio"/> Réseau
<input type="button" value="Ajouter"/>	

Lorsqu'on clique sur un cours, on arrive à une page détaillée qui reprend la liste des étudiants inscrits à ce cours. Lorsqu'on clique sur un étudiant, on arrive à une page détaillée avec la liste des cours où il est inscrit.

INT1 - Introductions

Matricule	Nom	Genre	Section
6	Dracaufeu	M	GESTION
19	Rattata	M	GESTION

19 - Rattata

Sigle	Titre	ECTS
INT1	Introductions	10
DEV1	Développement I	10

Une liste d'étudiants apparaît sur des pages différentes : celle les donnant tous et celles donnant ceux inscrit à un cours. Idem pour les listes de cours. Utilisez les fragments pour éviter la duplication de code dans les pages HTML.

Exercice 4 Réparer ce qui est cassé

Vérifiez votre service web donnant la liste des cours ; il est fort probable qu'il ne fonctionne plus. Réparez-le !

Exercice 5 Inscription au cours

Dans la page qui reprend le détail d'un cours, ajoutez un formulaire pour y inscrire un étudiant (en donnant son numéro). Idem dans la page qui reprend le détail d'un étudiant (en donnant le matricule du cours).

16.7 Paresseux ou gourmand ?

Les entités étant liées entre elles, il y a un problème potentiel de performances lorsqu'on va les chercher dans la BD.

Si on récupère un cours (`findById`), faut-il récupérer, dans le même accès, tous ses étudiants ? C'est un travail inutile si on n'en a pas besoin. Par contre, si on en a besoin et qu'on ne l'a pas fait, SPRING devra, au moment où on accède à la collection, générer un accès supplémentaire à la BD.

Il existe deux modes possibles :

EAGER (gourmand)

les étudiants seront chargées dès que le cours est chargé. Un tel comportement peut être très dangereux s'il est activé en cascade. En effet, si ce comportement est mis sur les étudiants d'un cours mais aussi sur les cours d'un étudiant, le chargement d'un cours entraîne le chargement des étudiants inscrits à ce cours et donc de tous les cours suivis par ses étudiants et donc ... Au final, toute la BD est chargée en mémoire !

LAZY (paresseux ou différé)

lorsque le cours sera chargé, la collection des étudiants ne sera chargée que lorsqu'elle sera utilisée.

Par défaut, toutes les relations de type ...`toMany` se font en mode **LAZY** et toutes les relations de type ...`toOne` se font en mode **EAGER**. Pour le dire autrement : une entité seule est chargée directement (**EAGER**), une collection d'entités est chargée à la demande (**LAZY**)

Ce comportement est configurable via l'option `fetch` des annotations `@...To....`

```
@ManyToMany(mappedBy="courses", fetch=FetchType.EAGER)
```

Tutoriel 1

Vérifier le mode

Faisons une petite expérience pour vérifier ce que nous venons d'expliquer

- Reprenez votre application PAE.
- Assurez-vous d'avoir demandé à afficher les ordres SQL générés dans les logs (cf. `applications.properties`).
- Dans le contrôleur, retrouvez la méthode qui gère la demande des détails d'un cours. Ajoutez une trace au début et à la fin.
- Lancez votre application et demandez les détails d'un cours.
- Vérifiez dans le log les ordres SQL générés.

Vous constatez que les détails des étudiants ont bien été demandés après la fin du contrôleur. Concrètement, lors de l'exécution du code THYMELEAF de génération de la page.

- Modifiez à présent le mode en **EAGER** pour la liste des étudiants d'un cours.
- Rafraîchissez la page de détail d'un cours et constatez la différence dans les logs.
- Revenez au mode **LAZY** par défaut et modifiez le contrôleur pour qu'il affiche le nombre d'étudiants dans les logs. Constatez la différence.

Un petit truc

Si une relation est en mode **LAZY** mais qu'on veut charger quand même la collection liée, il suffit de demander sa taille.

Ce n'est pas qu'une question de performance

Vous pourriez vous dire qu'il s'agit ici d'un détail qui n'influence que la performance mais pas le résultat obtenu. Détrompez-vous ! Ici, le détail des étudiants est demandé par THYMELEAF ce qui fait qu'on est encore sur le serveur et que la demande peut être satisfaite. Mais ce ne sera plus le cas si on est déjà du côté du client. Dans ce cas, tenter de parcourir la collection provoquera une exception.

16.8 Cascade

Reprendons l'exemple de la commande et de son détail.

```
public class SaleOrder {
    @OneToMany(mappedBy="order")
    private Collection<OrderItem> items;
}

public class OrderItem {
    @ManyToOne
    private SaleOrder order;
}
```

Supposons qu'on crée une nouvelle commande (`SaleOrder`) contenant donc une liste de `OrderItem`. Que doit-il se passer si on la sauve ? La commande `save` doit-elle sauver aussi les éléments de la commande ou est-ce que ça doit être fait à part ?

Cet aspect est contrôlé par le paramètre `cascade` des annotations `@...ToMany`. Parmi les valeurs possibles, on retrouve :

- ▷ `CascadeType.PERSIST` : une sauvegarde d'une entité dans la BD entraîne la sauvegarde des entités liées.
- ▷ `CascadeType.MERGE` : une mise à jour d'une entité dans la BD entraîne la mise à jour des entités liées.
- ▷ `CascadeType.REMOVE` : une suppression d'une entité dans la BD entraîne la suppression des entités liées.
- ▷ `CascadeType.ALL` : toutes les valeurs ci-dessus.

Par exemple, on pourrait écrire :

```
public class SaleOrder {
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private Collection<OrderItem> items;
}

public class OrderItem {
    @ManyToOne
    private SaleOrder order;
}
```

16.9 Et le reste...

Plusieurs notions liées aux entités ne seront pas abordées dans ce cours mais je voulais au moins que vous connaissiez leur existence.

Héritage

Il peut y avoir de l'héritage entre les entités. Il existe deux façons de le traduire au niveau des tables.

Version

L'annotation `@Version` permet d'éviter les problèmes d'accès concurrents.

Transient

On peut indiquer qu'un attribut ne doit pas être persisté via l'annotation `@Transient`.

Plusieurs tables

Une entité peut être persistée sur plusieurs tables via l'annotation `@SecondaryTables`.

Collection d'objets

Une collection d'objets (qui ne sont pas des entités) peut être persistée comme un BLOB ou, mieux, dans une table associée via l'annotation `@ElementCollections` [43].

Nous avons vu qu'il est très facile d'avoir des méthodes pour les opérations de bases CRUD de gestion d'une entité — il suffit d'étendre l'interface `CrudRepository` —.

Parfois, ces méthodes ne suffisent pas. Comment, par exemple, chercher un étudiant à partir de son nom et pas son numéro ?

SPRING propose deux solutions : les « méthodes requêtes » et l'annotation `@Query`. Voyons cela en détail.

17.1 Les méthodes requêtes

On peut ajouter des méthodes supplémentaires à l'interface étendant le `CrudRepository`. Si on respecte une certaine convention de nom ce code là aussi sera généré pour nous. En anglais, on parle de *derived query* [44]. Exemples :

- ▷ `findByNom(String nom)` pour chercher un étudiant à partir de son nom.
- ▷ `readOrdersByDeliveryZipAndPlacedAtBetween(zip, start, end)` pour lire les commandes faites pour un code postal donné entre 2 dates données.

Je vous renvoie à la documentation pour les détails.

Exercice 1

Requêtes dérivées pour PAE

Créez et testez des requêtes dérivées pour obtenir la liste des cours :

1. qui ont un nombre de crédits supérieur ou égal à une valeur donnée ;
2. dont le nom contient un texte donné.

Pour les tester, vous pouvez vous contenter d'écrire une méthode, lancée au démarrage, qui les appelle et affiche le résultat.

17.2 L'annotation `@Query`

Si les méthodes requêtes ne suffisent pas, on peut donner l'ordre SQL à exécuter via l'annotation `@Query` [45]. Par exemple :

```
@Query("SELECT * FROM USER u WHERE u.status = 1", nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

À nouveau, le code sera généré automatiquement et vous pouvez utiliser cette méthode pour obtenir tous les utilisateurs qui ont un 1 dans leur colonne `status`.

Vous aurez remarqué l'attribut `nativeQuery` précisant qu'il s'agit d'une requête native (en SQL pur, envoyée directement au SGBD). C'est donc qu'il existe une autre possibilité ? Oui, JPQL !

Les requêtes natives ont deux problèmes : le SQL accepté présente de légères différences entre les SGBD et ça demande de connaître précisément les tables utilisées, ce qui n'est pas forcément le cas lorsqu'elles sont générées automatiquement à partir des entités.

17.3. NOTIONS CHOISIES DE JPQL

JPQL^[46] est une sorte de *SQL orienté objet* qui permet de se placer à un plus haut niveau d'abstraction. La requête équivalente à la précédente en JPQL est :

```
@Query("SELECT u FROM User u WHERE u.status = 1")  
Collection<User> findAllActiveUsersNative();
```

Nom de la classe que l'on a écrite en java,
d'où le fait qu'elle ait une majuscule

SQL ≠ JPQL

Les deux notations sont très proches dans des cas très simples mais elles n'en sont pas moins différentes. En SQL, **User** est le nom d'une table et **status** le nom d'une colonne de cette table. En JPQL, **User** est le nom d'une **classe** et **status** le nom d'un **attribut** de cette classe.

La section suivante s'attache à vous apprendre ce JPQL.

17.3 Notions choisies de JPQL

Une requête JPQL (*Java Persistence Query Language*) permet toute la puissance de SQL tout en restant en OO. Elle est assez intuitive quand on connaît le langage SQL. Notons quelques éléments particuliers.

La clause SELECT

On peut y trouver des entités mais également des champs. Par exemple, pour obtenir la liste des noms des étudiants dans une **List<String>**, on peut écrire :

```
SELECT s.name FROM Student s
```

Si on demande plusieurs éléments, on obtient une **List<Object[]>**. Par exemple :

```
SELECT s.id, s.name FROM Student s
```

On peut également utiliser les opérations d'agrégation. Par exemple, pour calculer le nombre total de crédits de tous les cours :

```
SELECT SUM(c.credits) FROM Course c
```

La clause WHERE

On peut utiliser les opérateurs :

BETWEEN	pour un intervalle
LIKE	pour une comparaison de chaîne (_ et % utilisables)
IS NULL	pour tester si une valeur est nulle
MEMBER OF	pour tester l'appartenance à une collection
IS EMPTY	pour tester si une collection est vide

Exemples

```
SELECT c FROM Course c WHERE c.CREDITS BETWEEN 5 AND 10  
SELECT c FROM Course c WHERE c.students IS EMPTY
```

Paramètres

Les ordres JPQL peuvent contenir des paramètres. Par exemple :

```
@Query("SELECT c FROM Course c WHERE c.CREDITS BETWEEN :low AND :high")  
List<Course> findCourseByCreditsBetween(long low, long high);
```

La clause FROM

Pour traverser un lien de type n-1 ou 1-n, il suffit d'utiliser la notation pointée. Par exemple, si un **Customer** a une liste de **Command** et qu'une commande n'appartient qu'à un seul client,

```
SELECT com.customer.name FROM Command com WHERE com.date = :date
```

donne les noms des clients qui ont passé commande un jour donné.

Pour traverser un lien de type 1-n ou n-n, il existe l'équivalent des jointures. Par exemple, l'exemple précédent aurait pu s'écrire :

```
SELECT cust.name FROM Customer cust JOIN (cust.commands) com WHERE com.date =
    ↪ :date
```

Vous voyez, tout dépend du point de départ.

En vrac

Groupement

Les clauses GROUP BY et HAVING sont disponibles.

Subqueries

Comme avec SQL, on peut utiliser des sous-requêtes.

Requêtes JPQL pour PAE

Créez et testez des requêtes JPQL pour obtenir la liste :

1. des noms des étudiants ;
2. des id et des noms des étudiants ;
3. des noms des étudiants et du nombre total d'ects dans leur programme ;
4. des étudiants qui ont plus d'ects dans leur programme qu'une valeur donnée en paramètre.

Pour les tester, vous pouvez vous contenter d'écrire une méthode, lancée au démarrage, qui les appelle et affiche le résultat.

Exercice 3

Filtrer sur le nom

Ajoutez à la page présentant la liste de tous les étudiants, un champ de saisie permettant de filtrer sur le nom.

Liste des étudiants

Recherche par le nom

Matricule	Nom	Genre	Section
6	Dracaufeu	M	GESTION
19	Rattata	M	GESTION
25	Pikachu	F	GESTION
309	Dynavolt	M	INDUSTRIELLE

Pour le moment, le SGBD utilisé est un H2 embarqué. C'est certainement pratique lors du développement mais probablement pas la solution à retenir en production. Dans ce chapitre nous allons voir comment se connecter à d'autres SGBD.

18.1 MariaDB

Dans le cours de WEB II, vous avez appris à utiliser MARIADB, le SGBD inclus dans la suite XAMPP. Il est parfaitement possible de l'utiliser ici, et c'est facile !

Tutoriel 1

Utiliser MariaDB

Nous allons modifier votre application PAE de sorte que les données soient persistées dans MARIADB.

- ☒ Premièrement, lancez MARIADB. Comment ?! Vous ne souvenez plus comment faire ? ;)
- ☒ Créez une BD pour accueillir les tables. Par exemple : paedb.
- ☒ Dans le pom.xml de votre application, enlevez la dépendance à h2.
- ☒ Ajoutez ensuite une dépendance à MARIADB.

```
1 <dependency>
2   <groupId>org.mariadb.jdbc</groupId>
3   <artifactId>mariadb-java-client</artifactId>
4 </dependency>
```

mariadb/pom.xml

- ☒ Ajoutez enfin la configuration suivante dans le fichier application.properties en adaptant, si nécessaire le nom de la bd, le user ou le mot de passe.

```
1 spring.datasource.url=jdbc:mariadb://localhost:3306/paedb
2 spring.datasource.username=root
3 spring.datasource.password=
4 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
5 spring.jpa.hibernate.ddl-auto=create-drop
6 spring.datasource.initialization-mode=always
```

mariadb/application.properties

- ☒ Et c'est tout ! Vous pouvez tester !

18.2 Les profils

Il est probable que la configuration doit être différente selon que vous soyez en développement (un H2 embarqué par exemple) ou en production (disons MARIADB). Comment passer facilement de l'un à l'autre ? Grâce aux profils.

Voici comment ça fonctionne :

- ▷ Un fichier de configuration application-nomDuProfil.properties ne sera appliqué que si on est dans le profile donné.
- ▷ Le profil peut se donner comme paramètre de configuration lors du lancement du JAR.

Le but est de pouvoir, par exemple, utiliser 2 bases de données différentes entre le moment du développement et le moment de la mise en production

18.2. LES PROFILS

```
java -jar nomDuJar.jar --spring.profiles.active=nomDuProfil
```

- ▷ Il peut aussi se définir via une variable d'environnement

```
export SPRING_PROFILES_ACTIVE=nomDuProfil
```

- ▷ Une classe possédant l'annotation `@Profile("nomDuProfil")` ne sera exécutée que dans ce profil là.

Ce mécanisme n'est bien sûr pas limité à la configuration du SGBD. On le retrouvera notamment lorsqu'il s'agira de configurer la sécurité.

Septième partie

Compléments

Chapitre

19

Les tests

On vous l'a beaucoup répété, il est important d'écrire des tests automatiques pour vérifier votre code. Comme notre application est développée en couches, il est important de différencier les types de tests.

Test unitaire

Un **test unitaire** teste un élément indépendamment de tous les autres. Par exemple, lorsqu'on teste une méthode métier, on cherche à être indépendant de la base de données. Ceci est en général réalisé grâce à l'utilisation de *mocks*.

Mock

Un **mock** est un élément qui prend la place d'un autre dans un logiciel. Par exemple, un mock d'une base de donnée va hardcoder quelques valeurs sans utiliser de base de donnée. L'utilisation d'un mock permet de tester chaque couche indépendamment et rend en général les tests plus rapides.

Test d'intégration

Dans un **test d'intégration**, aucun élément n'est simulé. C'est généralement plus lent mais ça permet de vérifier que les différents éléments fonctionnent correctement ensemble.

SPRING fournit tout un ensemble d'outils pour faciliter l'écriture de ces tests [47].

Concrètement, on va voir comment écrire des tests unitaires pour :

- ▷ les entités ;
- ▷ un repository ;
- ▷ le code métier ;
- ▷ les services Rest ;
- ▷ les pages web.

19.1 Tester une entité

Commençons par voir comment tester une entité. Une entité étant une classe relativement simple, les seuls éléments qu'on peut avoir envie de tester sont les annotations introduisant des contraintes de validation.

Comme c'est la couche la plus basse, il n'y a pas de différence à faire entre test d'intégration et test unitaire.

Tutoriel 1 Tester une entité

Nous allons créer une entité basique et vérifier que les contraintes de validations fonctionnent bien comme prévu.

- ✍ Créez une nouvelle application de base. Les dépendances nécessaires sont : devtools, Lombok, Spring Web et Spring Data JPA.
- ✍ Vous devez également ajouter à la main les dépendances à H2 et aux validations.
- ✍ Ajoutez l'entité suivante :

```
tests/src/main/java/webg5/test/User.java
1 @Entity
2 @Data
3 @AllArgsConstructor
4 @NoArgsConstructor
5 public class User {
6     @Id
7     @Size(min = 6)
8     private String login;
9     @NotBlank
10    private String name;
11 }
```

- ✍ Le test peut ressembler à ceci :

```
tests/src/test/java/webg5/test/UserTest.java
1 @SpringBootTest
2 public class UserTest {
3
4     @Autowired
5     private BeanValidationUtil<User> validator;
6
7     @Test
8     public void loginValid() {
9         User user = new User("ValidLogin", "Name");
10        validator.assertIsValid(user);
11    }
12
13     @Test
14     public void loginSizeLessThan6Error() {
15         User user = new User("Login", "Name");
16         validator.assertHasError(user, "login", Size.class);
17     }
18
19     @Test
20     public void nameValid() {
21         User user = new User("ValidLogin", "Name");
22         validator.assertIsValid(user);
23     }
24
25     @Test
26     public void nameBlankError() {
27         User user = new User("ValidLogin", "");
28         validator.assertHasError(user, "name", NotBlank.class);
29     }
30 }
}
```

qui fait référence à la classe utilitaire `BeanValidationUtil` :

```

1  /**
2  * Méthodes utilitaires pour tester les annotations de validations posées sur les entités.
3  * @author MCD
4  */
5 @Component
6 public class BeanValidationUtil<T> {
7
8     @Autowired
9     private Validator validator;
10
11    /**
12     * Teste qu'une entité est bien valide.
13     * Le test qui appelle cette méthode échoue si l'entité n'est pas valide.
14     * @param entity l'entité à tester.
15     */
16    public void assertIsValid(T entity) {
17        assertTrue(validator.validate(entity).isEmpty());
18    }
19
20    /**
21     * Teste qu'une entité possède une violation de contrainte.
22     * Le test qui appelle cette méthode réussit si l'entité viole
23     * la contrainte indiquée par les paramètres et uniquement celle-là.
24     * @param entity l'entité à tester.
25     * @param invalidField l'attribut qui n'est pas valide.
26     * @param violatedConstraint l'annotation de l'attribut qui n'est pas respectée.
27     */
28    public void assertHasError(T entity, String invalidField, Class<? extends Annotation>
29                               ↪ violatedConstraint) {
30        Set<ConstraintViolation<T>> violations = validator.validate(entity);
31        assertEquals(1, violations.size());
32        ConstraintViolation<T> violation = violations.iterator().next();
33        assertEquals(invalidField, violation.getPropertyPath().toString());
34        assertEquals(violatedConstraint,
35                     ↪ violation.getConstraintDescriptor().getAnnotation().annotationType());
36    }
}

```

Packages cohérents

Il doit y avoir une cohérence dans les noms de packages. Par exemple, si la classe principale est dans le package `webg5.pae`, alors les classes de tests doivent avoir un package qui commence par la même valeur.

19.2 Tester un repository

Lorsqu'on écrit un repository, on peut être amené à ajouter quelques méthodes : basées sur la convention de nom et/ou accompagnées d'un JPQL (ou SQL). Il faudrait vérifier qu'elles fonctionnent comme prévu.

SPRING fournit une annotation qui permet facilement d'exécuter les tests en mémoire même si une vraie base de données est utilisée en production.

Tutoriel 2

Tester un repository

Nous allons reprendre l'application précédente, lui ajouter un repository et vérifier qu'il fonctionne.

Partons de l'application précédente.

Ajoutez le repository suivant :

```

1  public interface UserRepository extends JpaRepository<User, String> {
2      public User findByName(String name);
3      @Query("select u FROM User u WHERE length(u.login)>12")
4      public List<User> findByLongLogin();
5  }

```

Le test peut s'écrire :

19.3. TESTER UN CODE MÉTIER

```
tests/src/test/java/webg5/test/UserRepositoryTest.java
1  @DataJpaTest
2  public class UserRepositoryTest {
3
4      @Autowired
5      private UserRepository userRepository;
6
7      @Test
8      public void findByName() {
9          User user = new User("ValidLogin", "MCD");
10         userRepository.save(user);
11         User found = userRepository.findByName(user.getName());
12         assertEquals(user, found);
13     }
14
15     @Test
16     public void findByLongLogin() {
17         User user1 = new User("ValidLogin", "MCD");
18         User user2 = new User("VeryLongLogin", "MCD");
19         userRepository.save(user1);
20         userRepository.save(user2);
21         List<User> found = userRepository.findByLongLogin();
22         assertEquals(1, found.size());
23         assertEquals(user2, found.get(0));
24     }
}
```

L'annotation `@DataJpaTest` configure le système pour utiliser une base de données H2 en mémoire (même si on en a configuré une autre pour l'application). De plus, chaque test est suivi d'un *rollback* afin de les rendre indépendants.

19.3 Tester un code métier

Pour tester unitairement le code métier, il est bon de simuler l'accès à la BD via un *mock*.

Tutoriel 3

Tester un code métier

Nous allons reprendre l'application précédente, lui ajouter une méthode métier et vérifier qu'elle fonctionne correctement.

☒ Prenez une copie de l'application précédente.

☒ Ajoutez le service suivant :

```
tests/src/main/java/webg5/test/UserService.java
1  @Service
2  public class UserService {
3
4      @Autowired
5      private UserRepository userRepository;
6
7      public User getUserByName(String name) {
8          return userRepository.findByName(name);
9      }
10 }
```

☒ Le test peut s'écrire :

```

1  @SpringBootTest
2  public class UserServiceTest {
3
4      @Autowired
5      private UserService userService;
6
7      @MockBean
8      private UserRepository userRepository;
9
10     @Test
11     public void getUserByName() {
12         String name = "MCD";
13         User user = new User("ValidLogin", name);
14         Mockito.when(userRepository.findByName(name)).thenReturn(user);
15         User found = userService.getUserByName(name);
16         assertEquals(found.getName(), name);
17     }
18 }
```

tests/src/test/java/webg5/test/UserServiceTest.java
Contrairement à @DataJpaTest (qui crée une DB rien que pour les test), cette annotation est pour dire que nous allons faire des tests unitaires

Remarquez l'annotation `@MockBean` qui va faire que **partout** (et donc même dans la classe `UserService`) le `UserRepository` ne sera pas celui qu'on a défini mais un *faux simulant* le vrai.

La classe `Mockito` permet de configurer le mock : que doit-il répondre en fonction de l'appel qu'on fait ?

19.4 Tester un service Rest

Attaquons à présent le test des services REST. Nous allons :

- ▷ Simuler le code métier via un mock.
- ▷ Simuler les appels HTTP via un mock dédié fourni par SPRING.

Tutoriel 4

Tester un service Rest

Nous allons reprendre l'application précédente, lui ajouter un service web rentrant du JSON et vérifier qu'il fonctionne correctement.

- ☒ Prenez une copie de l'application précédente.
- ☒ Ajoutez une dépendance à JACKSON.
- ☒ Ajoutez le service web suivant :

```

1  @RestController
2  @RequestMapping("/api")
3  public class UserRest {
4
5      @Autowired
6      private UserService userService;
7
8      @GetMapping("/user/{name}")
9      public User getUserByName(@PathVariable("name") String name) {
10          return userService.getUserByName(name);
11      }
12 }
```

tests/src/main/java/webg5/test/UserRest.java

- ☒ Le test peut s'écrire :

19.5. TESTER UN SITE WEB

```
tests/src/test/java/webg5/test/UserRestTest.java
1  @WebMvcTest
2  public class UserRestTest {
3
4      @Autowired
5      private MockMvc mvc;
6
7      @MockBean
8      private UserService userService;
9
10     @Test
11     public void getUserByName() throws Exception {
12         String name = "MCD";
13         User user = new User("ValidLogin", name);
14         Mockito.when(userService.getUserByName(name)).thenReturn(user);
15         mvc.perform(get("/api/user/MCD"))
16             .andExpect(status().isOk())
17             .andExpect(jsonPath("$.name").value(name));
18     }
19 }
```

Vérifie qu'une balise contienne 'name' et que cette balise contienne la valeur en question

Par rapport à `@SpringBootTest`, l'annotation `@WebMvcTest` indique que toute la couche métier sera simulée via des mocks (vrai test unitaire). Il peut alors démarrer plus vite.

Remarquez également `mockMVC` pour simuler les appels webs et `userService` qui est un mock de notre vrai code métier. Il est configuré dans la méthode de test.

19.5 Tester un site web

Tester un site web peut se faire de la même façon. Simplement on devra tester le contenu HTML plutôt que le contenu JSON. Voici l'exemple qui avait été donné au début du cours :

```
hello/hello/src/test/java/webg5/spring/hello/WelcomeControllerTest.java
1 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
2 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
3
4 @SpringBootTest
5 @AutoConfigureMockMvc
6 public class WelcomeControllerTest {
7
8     @Autowired
9     private MockMvc mockMvc; // Permet de simuler le navigateur
10
11    @Test
12    public void testWelcomePage() throws Exception {
13        mockMvc.perform( get("/welcome?name=mcd")) // L'url à tester
14            .andExpect(status().isOk()) // La page est retournée
15            .andExpect(view().name("welcome")) // Générée à partir du template welcome
16                // Elle contient le texte attendu
17            .andExpect(content().string( Matchers.containsString("mcd")));
18    }
19 }
```

L'annotation `@AutoConfigureMockMvc` permet d'utiliser le `MockMVC`. Elle est induite avec `@WebMvcTest` mais doit être mise explicitement ici.

19.6 Exercice

Exercice 1

Tester votre application

Vous aurez compris ce qu'on attend de vous. Ajoutez des tests à votre application pour : a) les entités, b) les repositories, c) la couche métier, d) les services webs et e) la couche web.

La plupart des applications doivent être multilingues. Cela couvre (au moins) deux aspects : la traduction des messages et la présentation de certaines données (dates, nombres...) selon les habitudes de l'utilisateur. Ces aspects sont souvent regroupés sous le sigle `i18n`¹. Voyons ce que SPRING propose [48].

20.1 Internationalisation des messages

Les fichiers de propriétés

Quel que soit le type d'application à internationaliser (JAVA SE, client web), la première étape est d'enlever les textes des programmes et de les mettre dans des fichiers (appelés *fichiers de propriétés*, d'extension `.properties`), un par langue².

<code>messages.properties</code>	version par défaut
<code>messages_fr.properties</code>	version francophone
<code>messages_en.properties</code>	version anglaise

Il s'agit ici d'un mécanisme qui n'est pas propre SPRING mais est utilisé dans toutes les solutions JAVA. Chaque texte est identifié par une clé commune.

Version anglaise

```
greetings=Hello
farewell=Goodbye
inquiry=How are you?
```

Version française

```
greetings=Bonjour
farewell=Au revoir
inquiry=Comment allez-vous ?
```

Les messages peuvent aussi contenir des paramètres dont la valeur sera donnée par le programme au moment de leur utilisation.

Exemples :

```
nb.to.guess=Le nombre à trouver est entre {0} et {1}.
basket.size={0,choice,0#pas d'article|1#1 article|2#{0} articles}
```

Utilisation dans Spring

Par défaut, SPRING va chercher des fichiers de messages qui s'appellent précisément `messages`, placés directement dans le dossier `resources`.

Dans THYMELEAF, il suffit d'utiliser l'expression `#{key}` qui va chercher dans le bon fichier le texte associé au message dont on donne la clé.

1. Il y a 18 lettres entre le « i » et le « n » du mot « internationalization ».

2. Plus précisément, on ne désigne pas une langue mais un **locale**. Cela désigne une **langue** et une **région**. Par exemple : `fr_BE` désigne le français de Belgique.

Tutoriel 1**Localiser une page web**

Dans ce 1^{er} tutoriel, nous allons créer une page qui s'adaptera au *locale* demandé par le navigateur.

Démarrez un nouveau projet avec les extensions *web* et *thymeleaf* (ce sera suffisant).

Créez les fichiers de propriétés

`resources/messages_en.properties resources/messages_en.properties`

`greetings=Hello`

`greetings=Bonjour`

`resources/messages.properties`

`greetings=DefaultGreeting`

Créez une page `index.html` de base avec le message suivant :

`<h1 th:text="#{greetings}">Welcome</h1>`

Vous pouvez tester ; vous aurez probablement la version française.

Pour tester une autre langue, vous pouvez modifier les préférences du navigateur ou utiliser un outil comme CURL

`curl -H "Accept-Language: en" "http://localhost:8080/"`

Tutoriel 2**Choix de la langue via un paramètre**

Modifions le tutoriel pour accepter la langue dans un paramètre (`?lang=en`) et s'adapter en conséquence.

Il suffit d'ajouter le code JAVA suivant :

```
i18n/hello/src/main/java/webg5/i18n/LocaleConfig.java
1 @Configuration
2 public class LocaleConfig implements WebMvcConfigurer {
3
4     @Bean
5     public LocaleResolver localeResolver() {
6         return new SessionLocaleResolver();
7     }
8
9     @Override
10    public void addInterceptors(InterceptorRegistry registry) {
11        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
12        lci.setParamName("lang");
13        registry.addInterceptor(lci);
14    }
15 }
```

Vous pouvez tester ainsi

`curl "http://localhost:8080"`
`curl "http://localhost:8080?lang=en"`
`curl "http://localhost:8080?lang=fr"`

Tutoriel 3**Choix de la langue via un menu**

Modifions le tutoriel pour offrir des liens pour choisir la langue.

Ajoutez à la page dans le corps du document :

```
i18n/hello2/src/main/resources/templates/welcome.html
1 <p th:text="${#locale.language}"></p>
2 <a th:href="@{/(lang=fr)}>Français</a>
3 <a th:href="@{/(lang=en)}>English</a>
```

Testez !

Remarque : Ce tutoriel renvoie systématiquement sur la page d'accueil en cas de changement de langue. Si vous voulez rester sur la même page, vous pouvez écrire :

`th:href="#">th:T(org.springframework.web.servlet.support.ServletUriComponentsBuilder).fromCurrentRequest().replaceQueryParam('lang', 'fr').toUriString()#`

Exercice 1 PAE bilingue

Modifiez votre application du programme étudiant afin qu'un menu soit présent en permanence dans le haut de la page. Il doit permettre de changer de langue à tout moment. Prévoyez au moins 2 langues.

Exemple de résultat :



20.2 Localisation des données

JAVA et SPRING proposent des outils pour faciliter la localisation de certaines données : les dates, les nombres et les valeur monétaires.

La première étape est d'ajouter une extension de THYMELEAF qui gère la nouvelle API `time` offerte par JAVA 8

```

1 <dependency>
2   <groupId>org.thymeleaf.extras</groupId>
3   <artifactId>thymeleaf-extras-java8time</artifactId>
4 </dependency>
```

i18n/hello/pom.xml

On dispose alors de l'objet `#temporals`. Par exemple :

```
th:text="#{#temporals.format(#temporals.createNow(), 'SHORT')}
```


Chapitre

21

La sécurité

Voilà un vaste sujet qui couvre de nombreux aspects ; nous n'en aborderons que quelques uns.

21.1 Introduction

Dans la plupart des cas réels, on doit pouvoir identifier la personne qui visite un site mais aussi lui présenter un site différent en fonction de son rôle (client, fournisseur, employé, gestionnaire du site...). Voyons comment mettre cela en œuvre.

Un peu de terminologie

On utilise les termes suivants :

User principal

identifiant de la personne (login).

Authority

identifie la catégorie à laquelle appartient le visiteur. Un « User principal » peut appartenir à plusieurs *autorités* et une *autorité* peut, bien sûr, être partagée par plusieurs « User principal ».

Les différentes possibilités

Il existe de nombreux scénarios possibles. Voici ceux que nous allons aborder.

- ▷ L'utilisateur peut indiquer qui il est via :
 - ▷ Une page de login par défaut.
 - ▷ Une page de login définie par le développeur.
- ▷ L'ensemble des login et des mots de passe peuvent se trouver :
 - ▷ En dur dans la configuration de l'application.
 - ▷ Dans une BD.
 - ▷ Sur un serveur LDAP.
- ▷ On peut ou non chiffrer les mots de passe.
- ▷ On peut indiquer quelles pages sont publiques et quelles sont celles qui demandent une authentification.
- ▷ On peut imposer des rôles pour accéder à certaines pages.

Voilà un vaste programme. Abordons les points un à un.

21.2 La sécurité par défaut

Vous avez pu le constater, par défaut, il n'y a aucune sécurité dans le site. Il suffit pourtant d'ajouter la dépendance à `spring-boot-starter-security` pour obtenir une sécurité de base.

- ▷ Toutes les pages nécessitent d'être identifié.
- ▷ L'identification se fera via une page par défaut.
- ▷ Il y a un seul utilisateur : `user`.
- ▷ Le mot de passe, différent à chaque fois, est affiché dans les logs.
- ▷ Aucun rôle n'est créé.

Tutoriel 1

Une sécurité de base

Construisons une petite application de base pour notre apprentissage de la sécurité. La page d'accueil sera — à terme — publique et possédera un lien vers une page privée nécessitant d'être authentifié.

Commencez une nouvelle application de base.

Introduisez les deux vues que voici :

```
1 <h1>Bienvenue sur le site.</h1>                                     securite/demo/src/main/resources/templates/home.html
2 <a th:href="@{/private}">Espace privé</a>

1 <h1>Espace privé</h1>                                              securite/demo/src/main/resources/templates/private.html
2 <a th:href="@{/}">Retour à l'espace public</a>
```

Et un contrôleur pour gérer le tout.

```
1 @Controller                                         securite/demo/src/main/java/webg5/security/demo/DemoController.java
2 public class DemoController {
3
4     @GetMapping("/")
5     public String home() {
6         return "home";
7     }
8
9     @GetMapping("/privé")
10    public String privé() {
11        return "privé";
12    }
13
14 }
```

Lancez l'application. Sans surprise, pour l'instant, tout est accessible.

Ajoutez à présent la dépendance

```
1 <dependency>                                         securite/demo/pom.xml
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-security</artifactId>
4 </dependency>
```

Sauvez et constatez qu'il faut à présent s'identifier. Le nom est `user` et le mot de passe est affiché dans le log.

Le mode incognito

Si vous vous identifiez correctement, le navigateur retient l'information et ne vous la demandera plus, même si vous le quittez puis le relancez. Difficile, dès lors, de correctement tester les modifications de sécurité que nous allons apporter. La solution ? Le mode incognito où tout est oublié lorsque la fenêtre est fermée ;)

Bon ! Il reste du travail.

21.3 Utilisateurs codés en dur

Voyons comment créer des utilisateurs autres que celui par défaut, mais toujours en dur dans le code.

Tutoriel 2

Des utilisateurs en dur

Pour créer les utilisateurs `prof` et `etudiant`.

- Ajoutez simplement la classe suivante à votre projet :

```
1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfig extends WebSecurityConfigurerAdapter {
4      @Override
5      protected void configure(AuthenticationManagerBuilder auth) throws Exception {
6          auth.inMemoryAuthentication() Les User sont définis dans la mémoire
7              .withUser("prof")
8                  .password("{noop}prof") // noop = non chiffré
9                  .authorities("PROF") //
10             .and()
11                 .withUser("etudiant")
12                     .password("{noop}etudiant")
13                     .authorities("USER");
14     }
15 }
```

où les `authorities` représentent les rôles.

- Testez avec les deux utilisateurs créés (l'utilisateur par défaut `user` n'existe plus).

Configuration XML ou Java

De nombreux paramétrages SPRING peuvent se faire soit dans des fichiers XML soit directement en JAVA. C'est l'option que nous avons prise ici.

Le chainage de méthode

Le code de la méthode utilise un patron de conception qui s'appelle l'**enchaînement de méthodes** (*method chaining*). Chaque méthode de la classe retourne l'objet sur lequel il est appelé : `return this`. On peut ainsi *enchainer* les appels et obtenir un code plus simple et plus lisible.

21.4 Restreindre l'accès

Pour indiquer que seule la page `private` doit avoir un accès restreint, il suffit de redéfinir une méthode dans la classe de configuration :

Tutoriel 3

Restreindre l'accès à la partie privée uniquement

- Ajoutez ceci à la classe `SecurityConfig`

```
1  @Override
2  protected void configure(HttpSecurity http) throws Exception {
3      http.authorizeRequests()
4          .antMatchers("/private").authenticated() // Nécessite d'être identifié
5          .antMatchers("/**").permitAll() // Toutes les autres sont publiques
6      .and()
7          .formLogin() // Identification via la page de login par défaut
8      ;
9 }
```

- Testez.

- Modifiez le `authenticated()` par `hasAuthority("PROF")` et testez

21.5 Personnaliser les pages liées à l'identification

Dans le tutoriel précédent, nous avons continué de nous baser sur la page de login mais nous pouvons le configurer.

Tutoriel 4

Une page de login personnalisée

Nous allons donner notre propre page de login. Il n'y a que 3 règles à respecter :

1. Le formulaire doit être envoyé à `/login`;
2. Le champ contenant le login doit avoir `name="username"`
3. Le champ contenant le mot de passe doit avoir `name="password"`

Dans le détail :

- Dans la configuration, donnez l'url de la page de login :

```
1 .formLogin().loginPage("/login")
```

Ce sont des noms de fonction par défaut de Spring

- Créez un contrôleur pour lier l'URL et la vue :

```
1 @Controller
2 public class SecurityCtrl {
3     @GetMapping("/login")
4     public String login() {
5         return "login";
6     }
7 }
```

- Créez une vue de base :

```
1 <h1>Veuillez vous identifier</h1>
2 <form method="POST" th:action="@{/login}">
3     <input type="text" placeholder="Login..." name="username">
4     <input type="password" placeholder="Mot de passe..." name="password">
5     <button type="submit">Login</button>
6 </form>
```

Tutoriel 5

La page en cas d'accès refusé

Jusqu'à présent, si le login ou le mot de passe est mauvais, on reste sur la page de login. Par contre, si les informations sont valides mais que cette personne n'a pas les droits requis, une erreur se produit. On peut indiquer sur quelle page aller dans ce cas.

- Dans la configuration, ajoutez ceci :

```
1 .and()
2     .exceptionHandling().accessDeniedPage("/')
```

ce qui aura pour effet de revenir à la page d'accueil en cas d'accès refusé.

Tutoriel 6

Permettre la déconnexion

L'utilisateur doit pouvoir se déconnecter. Ce n'est pas compliqué.

- Dans la configuration, ajoutez ceci :

```
1 .and()
2     .logout().logoutSuccessUrl("/")
```

qui indique que le logout est permis et renvoie vers l'accueil du site.

- Vous pouvez par exemple ajouter ce bouton dans la partie privée

```
1 <form method="POST" th:action="@{/logout}">
2     <button type="submit">Logout</button>
3 </form>
```

21.6 Savoir qui est connecté

Il nous reste encore à savoir qui est connecté, dans le code JAVA ou dans les vues afin de s'adapter à la situation.

Tutoriel 7

Adapter la page si la personne est connectée

Il est possible dans THYMELEAF d'adapter le contenu en fonction du statut du visiteur (connecté ou non) et de bien d'autres paramètres encore.

- ☒ La première étape est d'ajouter une dépendance à une extension de THYMELEAF introduisant la sécurité.

```
1 <dependency>
2   <groupId>org.thymeleaf.extras</groupId>
3   <artifactId>thymeleaf-extras-springsecurity5</artifactId>
4 </dependency>
```

securite/form/pom.xml

qui indique que le logout est permis et renvoie vers l'accueil du site.

- ☒ Ajoutez le code suivant pour afficher le login de la personne connectée :

```
<p sec:authentication="name">Bob</p>
```

Par défaut

- ☒ Vous pouvez maintenant, ajouter le bouton de logout dans la page d'accueil uniquement si le visiteur est connecté.

```
1 <form method="POST" th:action="@{/logout}" sec:authorize="isAuthenticated()>
2   <button type="submit">Logout</button>
3 </form>
```

securite/form/home.html

Il y a d'autres méthodes que `isAuthenticated()`.

Sécurité dans le code Java

Dans le code JAVA, il suffit que le contrôleur ajoute un paramètre `Principal principal` pour obtenir des informations sur la personne connectée (ou pas).

Exercice 1

PAE et sécurité

Ajoutez de la sécurité dans l'application web PAE.

- ▷ Il y a 3 catégories d'utilisateurs : les étudiants, les enseignants et le secrétariat.
- ▷ La page d'accueil est accessible sans identification et c'est la seule.
- ▷ Tous les formulaires (ajout d'un cours, d'un étudiant et inscription) ne sont accessibles qu'aux membres du secrétariat.
- ▷ La barre de menu affiche le nom de la personne connectée.
- ▷ Si personne n'est connecté, la barre de menu offre un lien pour se connecter.
- ▷ Si une personne est connectée, la barre de menu offre un lien pour se déconnecter.



21.7 Utilisateurs stockés dans une BD

Voyons comment procéder pour stocker les utilisateurs dans la BD par défaut (H2 embarqué) [49].

Tutoriel 8

Utilisateurs dans une BD

Voici les étapes pour utiliser des utilisateurs qui sont stockés dans une BD.

- Si ce n'est pas encore le cas, ajoutez à votre projet les dépendances à SPRING JPA et H2 :

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-jpa</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>com.h2database</groupId>
7   <artifactId>h2</artifactId>
8   <scope>runtime</scope>
9 </dependency>
10 </dependencies>
```

- Modifiez le code de la classe `SecurityConfig` :

```
1 @Autowired
2 private DataSource dataSource;
3
4 @Override
5 protected void configure(AuthenticationManagerBuilder auth) throws Exception {
6   auth.jdbcAuthentication()
7     .dataSource(dataSource) Pour la DB. Il connaît le lieu de la DB grâce à ce qu'il y a dans application.properties
8     .withDefaultSchema()
9     .withUser(
10       User.withUsername("user")
11         .password("{noop}user")
12         .authorities("USER")
13     )
14   ;
15 }
```

Et c'est tout !

Paramétrier les tables

Par défaut, SPRING choisit les noms des tables et des colonnes pour stocker les informations de sécurité. Si ça ne convient pas, on peut modifier ces choix par défaut. C'est également utile pour entrer des valeurs via le SQL.

Tutoriel 9

Utilisateurs dans une BD initialisée en sql

Dans cette variante, nous allons imposer les tables pour stocker les utilisateurs et en ajouter quelques uns via du SQL comme nous l'avons fait pour les entités.

- Créez une entité pour les comptes et une pour les permissions.

```
1 @Data
2 @Entity
3 public class User {
4   @Id
5   private String username;
6   private String password;
7   private boolean enabled;
8 }
```

1. org.springframework.security.core.userdetails.User

```
1  @Entity
2  @Data
3  public class Authority {
4      @Id
5      private long id;
6      private String username;
7      private String authority;
8  }
```

- Modifiez la classe **SecurityConfig** comme suit :

```
1  @Override
2  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
3      auth.jdbcAuthentication()
4          .dataSource(dataSource)
5          .usersByUsernameQuery(
6              "select username, password, enabled from user where username=?")
7          .authoritiesByUsernameQuery(
8              "select username, authority" + " from authority where username=?");
9  }
```

On n'indique plus qu'on utilise le schéma par défaut et on donne les ordres pour accéder aux informations.

- Dans le fichier **data.sql** vous pouvez entrer les informations ainsi :

```
1  INSERT INTO User (username,password(enabled)) values ('40001','{noop}40001',true);
2  INSERT INTO Authority (id,username,authority) values (2,'40001','STUDENT');
```

21.8 Chiffrer les mots de passe

Pour l'instant, les mots de passes sont stockés en clair dans la BD ce qui n'est évidemment pas une bonne idée.

Si le compte est créé via du code JAVA, vous pouvez utiliser ceci :

```
1  @Override
2  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
3      PasswordEncoder pwdEncoder = new BCryptPasswordEncoder();
4      auth.jdbcAuthentication()
5          .dataSource(dataSource).withDefaultSchema()
6          .passwordEncoder(pwdEncoder)
7          .withUser(
8              User.withUsername("user")
9                  .password(pwdEncoder.encode("passwd"))
10                 .authority("USER"))
11         )
12     ;
13 }
```

Dans le fichier DATA.SQL, vous pouvez utiliser :

```
1  INSERT INTO User (username,password(enabled)) values
2      ('mcd','{bcrypt}$2a$10$YqviTSsx3..moyikdVrhm.tr8woUqFBqf4egwuMRCj/qVkkFlNyp6',true);
```

Pour générer la version chiffrée du mot de passe, vous pouvez utiliser un site comme celui-ci : <https://www.browserling.com/tools/bcrypt>.

Exercice 2 PAE - utilisateurs dans la BD

Modifiez votre application PAE afin que les logins/passwords ne soient plus codés en dur dans le code mais stockés dans la BD.

Chapitre

22

Les websockets

Dans un modèle traditionnel, un client HTTP envoie une requête et le serveur retourne une réponse. Le dialogue est toujours à l'initiative du client. Les **websockets** permettent de créer un canal **bi-directionnel** et **asynchrone**, entre un serveur et un (ou plusieurs) client(s). Le serveur peut prendre l'initiative d'envoyer des informations au client.

On peut imaginer une application de chat. Lorsqu'un client envoie un message, le retour lui revient directement ; il peut donc continuer à travailler. Le serveur, pendant ce temps, répercute ce message chez tous les clients connectés au même chat.

Le protocole WEBSOCKET n'est pas propre à SPRING, ni même à JAVA d'ailleurs. Avec SPRING, il est pris en charge par SPRING WEB SOCKET.

Tutoriel 1

Un Chat

Plutôt que de tout expliquer ici, je vous renvoie vers un tutoriel bien fait qui va vous montrer comment ça marche.

« Intro to WebSockets with Spring » par BAELDUNG (modifié le 20 octobre 2020). C'est ici : <https://www.baeldung.com/websockets-spring>.

Remarques

- ▷ Partez d'une application de base avec `devtools`, `Lombok` et `Spring web`.
- ▷ Lorsque vous ajoutez des éléments dans le `pom` vous pouvez omettre les numéros de versions.
- ▷ La classe `AbstractWebSocketMessageBrokerConfigurer` est maintenant dépréciée. Arrangez-vous !
- ▷ La classe `OutputMessage` est à créer.
- ▷ Le code du contrôleur est à placer dans un contrôleur web.
- ▷ Placez le code HTML dans un `index.html`.
- ▷ Ce code HTML fait appel à 2 ressources que vous pourrez trouver dans le GIT du tutoriel ou sur le réseau.

Huitième partie

Récapitulatif

Chapitre

23

Gestion de projet Scrum

En guise d'exercice récapitulatif, vous allez coder une petite application permettant de gérer les histoires de projets gérés via la méthodologie Scrum.

On vous demande d'écrire une petite application web permettant de consulter et d'encoder les histoires utilisateurs de différents projets. Ce développement doit respecter l'architecture vue au cours.

Détail du projet Stratego

Liste des projets

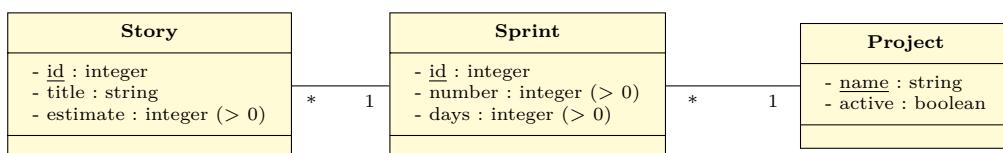
Nom	Nb sprints	Actif
Othello	3	Terminé
Stratego	2	En cours
EsiProgramme	4	En cours

Titre	Sprint	Points
Ajout des pièces	1	3
Déplacements des pièces	1	9
Aide utilisateurs	1	12
Validation des coups	1	5
Enregistrement des parties	2	10
Reprendre une partie	2	8
Pièces bonus	2	7

Ajouter histoire

Le Modèle Conceptuel des Données

Le schéma des données est le suivant :



Story

Représente une histoire utilisateur : son identifiant, son titre et son nombre de points estimés. Une histoire est liée à un unique sprint.

Sprint

Représente un sprint : son identifiant, son numéro au sein du projet et le nombre de jour de travail qu'il comprend. Les numéros de sprint commencent à 1 pour chaque projet. Un sprint est lié à un unique projet. Un sprint peut contenir plusieurs histoires utilisateurs.

Project

Représente un projet : son nom et son statut qui précise si il est terminé ou non. Un projet peut contenir plusieurs sprints.

Les identifiants numériques sont générés automatiquement.

Remplir la BD

Nous vous fournissons des ordres SQL pour remplir la BD.

```
recap/scrum/data.sql
1 INSERT INTO Project (name, active) VALUES ('Othello', false);
2 INSERT INTO Project (name, active) VALUES ('Stratego', true);
3 INSERT INTO Project (name, active) VALUES ('EsiProgramme', true);
4
5 INSERT INTO Sprint (id, number, days, project_name) VALUES (1,1, 10,'Othello');
6 INSERT INTO Sprint (id, number, days, project_name) VALUES (2,2, 7,'Othello');
7 INSERT INTO Sprint (id, number, days, project_name) VALUES (3,3, 10,'Othello');
8 INSERT INTO Sprint (id, number, days, project_name) VALUES (4,1, 15,'Stratego');
9 INSERT INTO Sprint (id, number, days, project_name) VALUES (5,2, 15,'Stratego');
10 INSERT INTO Sprint (id, number, days, project_name) VALUES (6,1, 10,'EsiProgramme');
11 INSERT INTO Sprint (id, number, days, project_name) VALUES (7,2, 7,'EsiProgramme');
12 INSERT INTO Sprint (id, number, days, project_name) VALUES (8,3, 9,'EsiProgramme');
13 INSERT INTO Sprint (id, number, days, project_name) VALUES (9,4, 10,'EsiProgramme');
14
15 ALTER SEQUENCE sprint_seq restart 100;
16
17 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (1,'Plateau de jeu', 5, 1);
18 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (2,'Algorithme de victoire', 3, 1);
19 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (3,'Tests unitaires', 3, 1);
20 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (4,'Inscription des joueurs', 6, 2);
21 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (5,'Création IA', 7, 3);
22 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (6,'Mise à jour de la vue', 5, 3);
23 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (7,'Ajout des pièces', 3, 4);
24 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (8,'Déplacements des pièces', 9, 4);
25 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (9,'Aide utilisateurs', 12, 4);
26 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (10,'Validation des coups', 5, 4);
27 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (11,'Enregistrement des parties', 10, 5);
28 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (12,'Reprendre une partie', 8, 5);
29 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (13,'Pièces bonus', 7, 5);
30 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (14,'Consultation étudiant', 8, 6);
31 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (15,'Consultation programme', 7, 6);
32 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (16,'Encodage des données', 5, 7);
33 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (17,'Authentification', 4, 8);
34 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (18,'Internationalisation', 5, 8);
35 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (19,'Tests acceptances', 6, 8);
36 INSERT INTO Story (id, title, estimate, sprint_id) VALUES (20,'Mises à jours programmes', 5, 9);
37
38 ALTER SEQUENCE story_seq restart 100;
```

Un conseil : attendez d'avoir défini vos entités avant de l'intégrer à votre projet. Vous pourrez ajuster ces ordres si c'est nécessaire pour être en accord avec la définition de vos entités ou – et c'est probablement le plus facile – en tenir compte lorsque vous allez définir vos entités.

Création des entités

Créez un nouveau projet et commencez par y ajouter les entités dont vous avez besoin pour réaliser ce développement. N'oubliez pas d'y inclure les **validations** nécessaires, accompagnées des **tests unitaires** les vérifiant. Nous vous demandons de persister vos données en mémoire vive via le SGBD **H2**.

Gabarit

Mettez en place un gabarit THYMELEAF qui sera utilisé dans **toutes les pages** de votre application.

Liste des projets

Développez une première page qui propose un tableau affichant la liste des projets. On y trouve 3 colonnes : le nom du projet, le nombre de sprints du projet et la mention *Terminé* si le projet n'est plus actif et *En cours* dans le cas contraire.

Cliquer sur le nom d'un projet permet de consulter une nouvelle page présentant les détails du projet.

Détails d'un projet

La page de détails d'un projet affiche le titre du projet accompagné d'un tableau contenant la liste des histoires de ce projet. On trouve dans ce tableau 3 colonnes : le titre de l'histoire, le numéro de l'itération associée à l'histoire et le nombre de points estimé de l'histoire. Indiquez à votre utilisateur via un message adéquat si le projet ne contient aucune histoire.

N'oubliez pas d'ajouter sur cette page de détail, un bouton permettant de revenir à la liste des projets.

Tester le site web

Validez votre développement en ajoutant un test qui vérifie le contenu HTML des détails d'un projet. Ce test doit consulter un projet présent dans le fichier *data.sql* et vérifier la présence du titre de ce projet ainsi que l'existence de sa première histoire.

Ajouter une histoire

Ajoutez à la page de détails d'un projet un bouton qui renvoie vers un formulaire permettant d'ajouter une histoire. Attention une nouvelle histoire ne peut être ajoutée qu'au dernier sprint du projet. Si un projet est clôturé, aucune histoire ne peut y être ajoutée. Lorsqu'une histoire est ajoutée par le formulaire, l'utilisateur est redirigé vers la page de détails du projet. Cette page affichant la liste des histoires mise à jour.

Sécurité

Ajoutez de la sécurité. On va considérer que les utilisateurs sont enregistrés dans une base de données. Et on va dire que tout le monde peut visualiser tout le site (pas de login nécessaire) mais que seules les personnes enregistrées peuvent modifier les données (ajouter une histoire).

Internationalisation

Faites en sorte que l'application puisse être visualisée en français ou en anglais via un menu. Les noms des projets et les titres des histoires ne doivent pas être traduits.

Service Web

Développez un service web qui permet d'obtenir la liste des projets actifs dont le nom commence par les lettres données en paramètres. Le service retourne une liste de projets avec les données suivantes pour chaque projet : son nom, le nombre de sprints qu'il contient, le nombre d'histoires qu'il contient.

La réponse est au format JSON.

On vous demande d'écrire un ordre JPQL dédié pour répondre à la demande.

Chapitre

24

Un quiz

On vous propose de coder une application web permettant aux utilisateurs de répondre aux questions et de consulter toutes les réponses données. Ce développement doit respecter l'architecture vue au cours.

Sondage

Nom	Nb de réponses	Nb Oui	Nb Non
Traversant une dizaine de pays, le Danube est le plus long fleuve d'Europe.	3	2	1
La superficie des États-Unis est plus grande que celle de l'Antarctique.	5	5	0
Le 4 juillet 1959, Neil Armstrong devient le premier homme à marcher sur la Lune.	1	1	0
David Bowie est mort deux jours après le lancement de son dernier album, Blackstar.	1	1	0
Le 16 janvier 2016, la NASA a annoncé avoir réussi à faire éclore la première fleur dans l'espace.	0	0	0
Dans le conte de Charles Perrault, la belle au bois dormant a deux enfants avec le prince.	0	0	0
Brian May, le guitariste de Queen, est détenteur d'un PhD en astrophysique.	0	0	0
Le père de BigFlo et Oli est un chanteur de salsa argentine.	0	0	0
Once Upon a Time in... Hollywood signe la troisième collaboration entre Leonardo DiCaprio et Quentin Tarantino.	0	0	0
En 1983, le premier trophée de la Coupe du monde a été volé et n'a jamais été retrouvé.	0	0	0

Survey

Le père de BigFlo et Oli est un chanteur de salsa argentine.

Aucune réponse pour l'instant

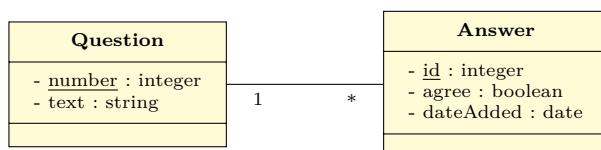
Je suis d'accord

[Enregistrer votre réponse](#)

[Retour à la liste des questions](#)

Le Modèle Conceptuel des Données

Le schéma des données est le suivant :



Question

Représente une question à poser. Une question est décrite par un identifiant et un texte. Le texte de cette question **ne peut pas être vide**. Une question est liée à plusieurs réponses. Chaque question attend une réponse simple : Oui ou Non.

Answer

Représente une réponse à une question. Une réponse est décrite par un identifiant, une valeur (oui ou non) et une date d'ajout. La date d'ajout **ne peut pas être null**. Chaque réponse est liée à une unique question.

Les identifiants numériques sont générés automatiquement.

Remplir la BD

Nous vous fournissons des ordres SQL pour remplir la BD.

```
recap/quiz/data.sql
1 insert into Question (number,text) values
2 (1,'Traversant une dizaine de pays, le Danube est le plus long fleuve d''Europe.'),
3 (3,'Le 4 juillet 1959, Neil Armstrong devient le premier homme à marcher sur la Lune.'),
4 (2,'La superficie des États-Unis est plus grande que celle de l''Antarctique.'),
5 (4,'David Bowie est mort deux jours après le lancement de son dernier album, Blackstar.'),
6 (5,'Le 16 janvier 2016, la NASA a annoncé avoir réussi à faire éclore la première fleur dans l''espace.'),
7 (6,'Dans le conte de Charles Perrault, la belle au bois dormant a deux enfants avec le prince.'),
8 (7,'Brian May, le guitariste de Queen, est détenteur d''un PhD en astrophysique.'),
9 (8,'Le père de BigFlo et Oli est un chanteur de salsa argentine.'),
10 (9,'Once Upon a Time in... Hollywood signe la troisième collaboration entre Leonardo DiCaprio et Quentin
    → Tarantino.'),
11 (10,'En 1983, le premier trophée de la Coupe du monde a été volé et n''a jamais été retrouvé.');
12
13 ALTER SEQUENCE question_seq restart 20;
14
15 INSERT INTO Answer (id,question_number,agree,date_added) VALUES
16 (1,1,true,'2020-08-12'),
17 (2,1,false,'2020-08-11'),
18 (3,1,true,'2020-08-11'),
19 (4,2,true,'2020-05-30'),
20 (5,2,true,'2020-05-30'),
21 (6,2,true,'2020-05-25'),
22 (7,2,true,'2020-05-22'),
23 (8,2,true,'2020-06-01'),
24 (9,3,true,'2020-08-19'),
25 (10,4,true,'2020-08-17');
26
27 ALTER SEQUENCE answer_seq restart 20;
```

Un conseil : attendez d'avoir défini vos entités avant de l'intégrer à votre projet. Vous pourrez ajuster ces ordres si c'est nécessaire pour être en accord avec la définition de vos entités ou – et c'est probablement le plus facile – en tenir compte lorsque vous allez définir vos entités.

Création des entités

Créez un nouveau projet et commencez par y ajouter les entités dont vous avez besoin pour réaliser ce développement. N'oubliez pas d'y inclure les **validations** nécessaires, accompagnées des **tests unitaires** les vérifiant. Nous vous demandons de persister vos données en mémoire vive via le SGBD **H2**.

Gabarit

Mettez en place un gabarit THYMELEAF qui sera utilisé dans **toutes les pages** de votre application.

Liste des questions

Développez une première page qui affiche la liste des questions.

Cliquer sur le texte d'une question permet de consulter une nouvelle page présentant les détails de la question.

Détails d'une question

La page de détails d'une question affiche le texte de la question accompagné d'un tableau contenant la liste des réponses. On trouve dans ce tableau 2 colonnes : la date d'ajout de la réponse et la valeur de la réponse. Indiquez à votre utilisateur via un message adéquat si la question ne contient aucune réponse.

N'oubliez pas d'ajouter sur cette page de détail, un bouton permettant de revenir à la liste des questions.

Tester le site web

Validez votre développement en ajoutant un test qui vérifie le contenu HTML des détails d'une question. Ce test doit consulter une question présente dans le fichier *data.sql* et vérifier la présence du texte de cette question ainsi que l'existence de sa première réponse.

Ajouter une réponse

Ajoutez un formulaire qui permet à l'utilisateur d'encoder une réponse.

Via ce formulaire l'utilisateur peut uniquement choisir si il est d'accord avec la question (oui ou non), la date d'ajout étant la date du jour.

Plusieurs possibilités s'offrent à vous pour ajouter ce formulaire :

- ▷ il peut être placé directement sur la page de détails d'une question ;
- ▷ il peut être sur une nouvelle page accessible via un bouton à partir de la page de détails d'une question.

Lorsqu'une réponse est ajoutée par le formulaire, l'utilisateur doit voir cette nouvelle réponse sur la page de détails d'une question. Pensez à actualiser son contenu si nécessaire.

Internationalisation

Faites en sorte que l'application puisse être visualisée en français ou en anglais via un menu. Le texte des questions et des réponses ne doit pas être traduit.

Service Web

Développez un service web qui permet d'obtenir la liste des questions dont le nombre de réponses est plus grand qu'une certaine valeur donnée en paramètre.

Pour réaliser cette recherche vous pouvez par exemple créer une méthode qui :

- ▷ va obtenir la liste des questions de la base de données ;
- ▷ parcourir la liste des questions ;
- ▷ pour chaque question compter le nombre de réponse ;
- ▷ conserver dans une liste temporaire les questions qui ont le nombre de réponses suffisantes ;
- ▷ retourner la liste temporaire.

Vous pouvez également réaliser cette recherche en utilisant des expressions lambda ou des requêtes JPQL adaptées.

La réponse est au format JSON.

Par exemple, l'appel `http://localhost:8080/api/question/2` donnerait :

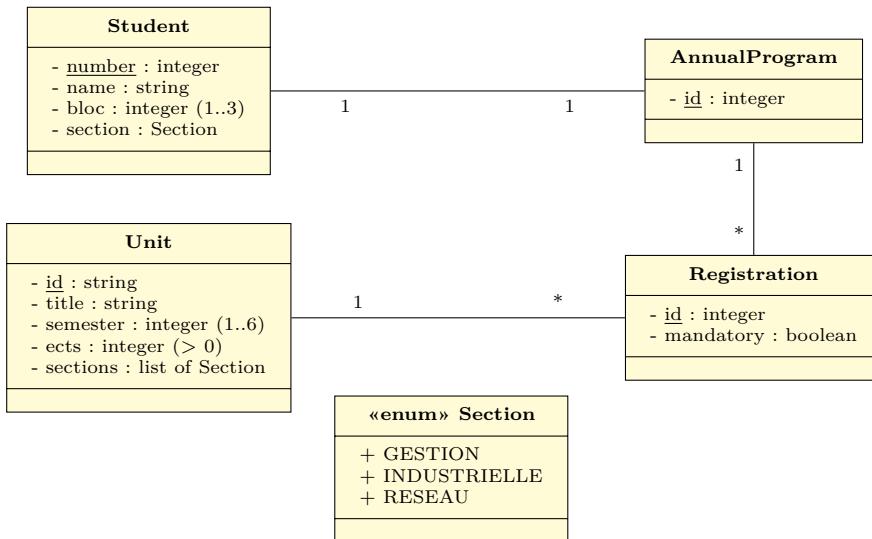
```
[{"text":"Traversant une dizaine de pays, le Danube est le plus long fleuve  
→ d'Europe.", "nbAnswers":3}, {"text":"La superficie des États-Unis est plus grande que  
→ celle de l'Antarctique.", "nbAnswers":5}]
```


Dans cet exemple récapitulatif, vous allez retravailler l'application PAE avec un MCD enrichi et plus de fonctionnalités.

- ▷ Un enseignant pourra :
 - ▷ Faire une recherche sur les étudiants en fonction de différents critères ;
 - ▷ Visualiser/modifier leur programme personnalisé.
- ▷ Un étudiant pourra uniquement visualiser/modifier son propre programme personnalisé.

Le Modèle Conceptuel des Données

Voici le schéma des données :



Quelques précisions :

- ▷ Les **id** numériques de **AnnualProgram** et **Registration** seront générés automatiquement.
- ▷ La liste des sections associées à un cours ne peut pas être vide.
- ▷ Vous trouverez peut-être certains choix bizarres. Nous avons fait en sorte que cet exemple vous permettre de vous exercer sur un maximum de concepts vus au cours.

Remplir la BD

Nous vous fournissons des ordres SQL pour remplir la BD. En voici un extrait (le fichier complet est disponible dans les codes sur poÉSI) :

```
1 insert into Unit (id,title,semester,ects) values ('INT1', 'Introduction à l''informatique',1,10);
2 insert into Unit (id,title,semester,ects) values ('MAT1', 'Mathématique contextualisée',1,6);
3 insert into Unit (id,title,semester,ects) values ('CAI1', 'Communication anglophone
   ↪ contextualisée 1',1,2);
4 insert into Unit_Sections (unit_id, sections) values ('INT1', 'G');
5 insert into Unit_Sections (unit_id, sections) values ('MAT1', 'G');
6 insert into Unit_Sections (unit_id, sections) values ('CAI1', 'G');
7 insert into Unit_Sections (unit_id, sections) values ('CPT1', 'G');
8 insert into Unit_Sections (unit_id, sections) values ('INT1', 'I');
9 insert into Unit_Sections (unit_id, sections) values ('MAT1', 'I');
10 insert into Unit_Sections (unit_id, sections) values ('CAI1', 'I');
11 insert into Unit_Sections (unit_id, sections) values ('INT1', 'R');
12 insert into Unit_Sections (unit_id, sections) values ('MAT1', 'R');
13 insert into Unit_Sections (unit_id, sections) values ('CAI1', 'R');
14 INSERT INTO Student (number,name,bloc,section) VALUES (40001,'Baker Gates',2,'G');
15 INSERT INTO Student (number,name,bloc,section) VALUES (40002,'Lillith Meyers',2,'G');
16 INSERT INTO Student (number,name,bloc,section) VALUES (40003,'Nevada Cline',3,'G');
17 ALTER SEQUENCE student_seq restart 40100;
18
19 INSERT INTO Annual_Program (id,student_number) VALUES (1,40001);
20 INSERT INTO Annual_Program (id,student_number) VALUES (2,40002);
21 INSERT INTO Annual_Program (id,student_number) VALUES (3,40003);
22 INSERT INTO Registration (id,mandatory,annual_program_id,unit_id) VALUES
   ↪ (101,true,1,'INT1');
23 INSERT INTO Registration (id,mandatory,annual_program_id,unit_id) VALUES
   ↪ (102,true,2,'INT1');
24 INSERT INTO Registration (id,mandatory,annual_program_id,unit_id) VALUES
   ↪ (103,true,3,'INT1');
25 ALTER SEQUENCE hibernate_sequence restart 1000;
```

Un conseil : attendez d'avoir défini vos entités avant de l'intégrer à votre projet. Vous pourrez ajuster ces ordres si c'est nécessaire pour être en accord avec la définition de vos entités ou – et c'est probablement le plus facile – en tenir compte lorsque vous allez définir vos entités.

Les entités

Comme première étape dans le développement nous vous proposons :

1. De démarrer un nouveau projet et de le suivre via GIT ?
2. De définir les entités.
 - ▷ Ne pas oublier les validations.
 - ▷ Pour la liste des sections d'une unité, vous pouvez vous pencher sur l'annotation `@ElementCollection`.
3. D'écrire des tests unitaires pour vérifier la validation.
4. D'ajouter le fichier `data.sql` et d'effectuer les éventuels ajustements nécessaires à sa compatibilité avec les entités.
 - ▷ Remarquez les ordres `ALTER SEQUENCE` dans le SQL fourni. Vous devrez probablement explorer les possibilités et les options des annotations `@GeneratedValue` et `@SequenceGenerator`.

La liste des étudiants

Vous pouvez maintenant commencer à travailler chaque fonctionnalité. Nous vous proposons de commencer par la liste des étudiants, sans introduire pour l'instant d'authentification de l'utilisateur.

1. Commencez par afficher la liste complète.

Numéro	Nom	Section	Bloc
40001	Baker Gates	G	2
40002	Lillith Meyers	G	2
40003	Nevada Cline	G	3
40004	Kaitlin Christensen	G	2
40005	Peter Cote	G	3
40006	Gannon Wade	G	1

2. Mettez en place un gabarit THYMELEAF qui vous servira pour chaque page.
3. Ajoutez la possibilité de données des critères pour filtrer la liste.

Numéro	Partie du nom	Section	Bloc
<input type="text"/>	<input type="text"/>	Toutes ▾	Tous ▾
<input type="button" value="Filtrer"/>			

- ▷ Vous pouvez filtrer en JAVA la liste complète – elle n'est pas bien grande.
- ▷ Voici un bon exercice pour vous entraîner à programmer en fonctionnel ;)

4. Prévoyez des tests et ne négligez pas la documentation.

Le détail d'un étudiant

Il doit être à présent possible de cliquer sur un étudiant pour arriver sur une page affichant les informations sur cet étudiant. Pour l'instant, vous pouvez vous contenter d'afficher son programme sans possibilité d'édition.

Fiche étudiant

40001 - Baker Gates - G - bloc 2

Programme annuel				
ID	Intitulé	ECTS	Quadri	Obligatoire
INT1	Introduction à l'informatique	10	1	✓
DEV1	Développement I	10	1	✓
CAI1	Communication anglophone contextualisée I	2	1	

Un site bilingue

Faites en sorte que l'application puisse être visualisée en français ou en anglais via un menu.



Authentification

Introduisez à présent l'obligation de s'identifier avant de pouvoir accéder aux pages. Faites en sorte qu'un étudiant soit envoyé directement sur sa fiche – la seule qu'il pourra consulter – alors que tout autre personne aura accès à la liste des étudiants ainsi qu'à toutes les fiches des étudiants.

Modifier le programme

Ajoutez à la fiche d'un étudiant un bouton permettant de modifier son programme. Attention, dans ce cas, il faudra afficher tous les cours (de cette section). Vous devrez aussi empêcher de ne pas prendre un cours obligatoire. Vous serez probablement amenés à définir un DTO adapté.

Programme annuel					
ID	Intitulé	ECTS	Quadri	Obligatoire	Inscrit
INT1	Introduction à l'informatique	10	1	✓	<input checked="" type="checkbox"/>
MAT1	Mathématique contextualisée	6	1		<input type="checkbox"/>
CAI1	Communication anglophone contextualisée I	2	1		<input checked="" type="checkbox"/>
CPT1	Comptabilité contextualisée I	2	1		<input type="checkbox"/>
DEV1	Développement I	10	1	✓	<input checked="" type="checkbox"/>
STA2	Statistique contextualisée	3	2		<input type="checkbox"/>
CAI2	Communication anglophone contextualisée II	2	2		<input type="checkbox"/>
DEV2	Développement II	10	2		<input type="checkbox"/>
DON2	Persistance des données I	5	2		<input type="checkbox"/>
SYS2	Systèmes d'exploitation I	5	2		<input type="checkbox"/>
WEBG2	Développement Web I	5	2		<input type="checkbox"/>
PHYIR2	Physique I	3	2		<input type="checkbox"/>
RESIR2	Réseau II	2	2		<input type="checkbox"/>

Enregistrer Annuler

Remarque : pour être au plus proche de la situation réelle, il faudrait empêcher (ne pas proposer) les cours déjà réussis dans la liste. Mais nous ne disposons pas de cette information dans le modèle actuel. Il faudrait ajouter quelques entités pour retenir le passé de l'étudiant.

Les services webs

Introduisez des services web pour obtenir les informations déjà codées :

- ▷ La liste complète des étudiants ;
- ▷ La liste en fonction de critères ;
- ▷ Le détail d'un étudiant.

Neuvième partie

Annexes

Deployer une application Spring

Avoir une application qui tourne en local sur la machine, c'est bien. La rendre disponible sur un serveur, c'est mieux ! Voyons quelques possibilités.

A.1 Heroku

Heroku est une application opensource qui offre un serveur cloud où les utilisateurs peuvent déployer et faire tourner gratuitement quelques applications. On parle de **platform as a service (PaaS)**.

Paas

Comme vous le verrez dans le tutoriel, nous allons vraiment l'utiliser comme un PAAS. Nous allons y déployer uniquement les sources et la plateforme va se charger de produire l'application, de la tester...

Tutoriel 1 Déployer une application Spring sur Heroku

Nous allons déployer sur HEROKU une petite application *Hello World!*.

Créer un compte

- ☒ Allez sur heroku.com et créez-vous un compte.

Installez le client

- ☒ Allez sur <https://devcenter.heroku.com/articles/heroku-cli>.
- ☒ Téléchargez et installez l'outil `heroku`.

Se connecter

- ☒ Dans un terminal

```
heroku login
```

Préparer l'application

HEROKU fonctionne en symbiose avec GIT. Déployer une application c'est juste l'envoyer sur un serveur git.

- ☒ Placez-vous dans votre application *Hello, World!* ou une copie.
- ☒ Si ce n'est pas encore fait, faites suivre ce projet par GIT.

```
git init
```

- ☒ Demandez à ne pas suivre le dossier `target`.

A.2. DOCKER

```
echo target >>.gitignore
```

- ☒ La commande ci-dessous crée un *remote* nommé heroku sur le serveur GIT de HEROKU.

```
heroku create he2b-votreLogin-hello
```

Déployer l'application

- ☒ Déployer, c'est juste commiter et pusher les modifications.

```
git add .  
git commit -m "test heroku"  
git push heroku master
```

Tester

- ☒ Vous pouvez simplement vous connecter sur
<https://he2b-votreLogin-hello.herokuapp.com/>.
- ☒ Vous pouvez aussi vérifier sur le tableau de bord
<https://dashboard.heroku.com/apps/>.

A.2 Docker

Docker est un logiciel libre permettant facilement de lancer des applications dans des conteneurs logiciels. Une image (on parle de **conteneur**) contient l'application et toutes ses dépendances. Elle pourra alors tourner sur toute machine disposant de DOCKER.

C'est un peu comme de la virtualisation mais en plus léger (on parle de *conteneurisation*) car on utilise encore beaucoup de fonctionnalités de l'hôte.

TODO

Faire (où trouver) un petit tuto Docker.

TODO

Montrer comment déployer un conteneur Docker (par exemple sur Kubernetes)

Annexe

B

Liste des dépendances utiles

Voici les dépendances que vous serez amenés à incorporer à votre pom.xml en fonction de vos besoins. On n'indique pas de numéro de version ce qui permet à Spring Boot de prendre celle qui est compatible avec le reste. Exception : quand on fait référence à un élément externe dont il ne peut deviner la version utilisée.

Devtools

Facilités pour le développement.

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-devtools</artifactId>
3 <scope>runtime</scope>
4 <optional>true</optional>
```

pom/pom.xml

Lombok

Permet d'utiliser les annotations Lombok.

```
1 <groupId>org.projectlombok</groupId>
2 <artifactId>lombok</artifactId>
3 <optional>true</optional>
```

pom/pom.xml

Starter-web

Permet de coder des contrôleurs web et rest.

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-web</artifactId>
```

pom/pom.xml

Starter-thymeleaf

Permet de coder les vues (les pages web) avec THYMELEAF

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-thymeleaf</artifactId>
```

pom/pom.xml

Thymeleaf-layout-dialect

Extension pour utiliser les layouts dans THYMELEAF

```
1 <groupId>nz.net.ultraq.thymeleaf</groupId>
2 <artifactId>thymeleaf-layout-dialect</artifactId>
```

pom/pom.xml

Validation

Permet d'intégrer la validation des données

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-validation</artifactId>
```

pom/pom.xml

Starter-data-jpa

Permet d'utiliser JPA pour la persistance (entités, validation...)

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-data-jpa</artifactId>
```

pom/pom.xml

H2

Permet d'utiliser H2 comme SGBD (embarqué).

```
1 <groupId>com.h2database</groupId>
2 <artifactId>h2</artifactId>
3 <scope>runtime</scope>
```

[pom/pom.xml](#)

MariaDB

Permet d'utiliser MARIADB comme SGBD.

```
1 <groupId>org.mariadb.jdbc</groupId>
2 <artifactId>mariadb-java-client</artifactId>
3 <version>2.4.3</version>
```

[pom/pom.xml](#)

Jackson

Permet d'avoir une conversion automatique des réponses REST en JSON.

```
1 <groupId>com.fasterxml.jackson.core</groupId>
2 <artifactId>jackson-databind</artifactId>
```

[pom/pom.xml](#)

JUnit

Permet d'écrire des tests JUNIT 5 avec une rétro-compatibilité avec la version 4.

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-test</artifactId>
3 <scope>test</scope>
```

[pom/pom.xml](#)

Pour enlever la retro-compatibilité JUNIT 4, ajoutez :

```
1 <exclusions>
2   <exclusion>
3     <groupId>org.junit.vintage</groupId>
4     <artifactId>junit-vintage-engine</artifactId>
5   </exclusion>
6 </exclusions>
```

[pom/pom.xml](#)

Starter-security

Pour introduire de la sécurité dans l'application.

```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-security</artifactId>
```

[pom/pom.xml](#)

Thymeleaf-extras-springsecurity5

Extension de THYMELEAF pour la sécurité.

```
1 <groupId>org.thymeleaf.extras</groupId>
2 <artifactId>thymeleaf-extras-springsecurity5</artifactId>
```

[pom/pom.xml](#)

Spring-security-test

Permet de tester la sécurité dans JUNIT.

```
1 <groupId>org.springframework.security</groupId>
2 <artifactId>spring-security-test</artifactId>
3 <scope>test</scope>
```

[pom/pom.xml](#)

Références

Les moteurs de production

- [1] *Ant website*. The Apache Software Foundation. URL : <http://ant.apache.org/> (cf. p. 16).
- [2] *Maven website*. The Apache Software Foundation. URL : <http://maven.apache.org/> (cf. p. 18).
- [3] *Maven Getting Started Guide*. The Apache Software Foundation. URL : <http://maven.apache.org/guides/getting-started/> (cf. p. 18).

Visual Studio Code

- [4] *Java in Visual Studio Code*. Microsoft. URL : <https://code.visualstudio.com/docs/languages/java> (cf. p. 23).
- [5] *Java Extension Pack*. URL : <https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack> (cf. p. 23).
- [6] *Editing Java in Visual Studio Code*. Microsoft. URL : <https://code.visualstudio.com/docs/java/java-editing> (cf. p. 24).
- [13] *Lombok Extension*. URL : <https://marketplace.visualstudio.com/items?itemName=GabrielBB.vscode-lombok> (cf. p. 28).

Les logs

- [7] BAELDUNG. *Introduction to Java Logging*. URL : <https://www.baeldung.com/java-logging-intro> (cf. p. 25).
- [8] Eric GOEBELBECKER. *A Guide to Logback*. URL : <https://www.baeldung.com/logback> (cf. p. 25).
- [9] *Logback website*. URL : <https://logback.qos.ch/> (cf. p. 25).
- [10] *SLF4J website*. URL : <https://www.slf4j.org/manual.html> (cf. p. 25).

Lombok

- [11] *Lombok website*. URL : <https://projectlombok.org/> (cf. p. 27).
- [12] BAELDUNG. *Introduction to Project Lombok*. URL : <https://www.baeldung.com/intro-to-project-lombok> (cf. p. 27).
- [13] *Lombok Extension*. URL : <https://marketplace.visualstudio.com/items?itemName=GabrielBB.vscode-lombok> (cf. p. 28).

Spring framework

- [14] *Spring website*. On y trouve de nombreux guides et tutoriels. URL : <https://spring.io/> (cf. p. 32).
- [15] *Spring Framework Documentation*. URL : <https://docs.spring.io/spring/docs/5.1.9.RELEASE/spring-framework-reference/index.html> (cf. p. 32).

- [16] *Spring Framework API javadoc*. URL : <https://docs.spring.io/spring-framework/docs/current/javadoc-api/> (cf. p. 32).
- [21] Craig WALLS. *Spring in action*. Fifth Edition. Manning, 2018, p. 520. ISBN : 9781617294945. URL : <https://spring.io/> (cf. p. 33).
- [22] *Baeldung website*. Propose sur son site beaucoup de tutoriaux bien faits à propos de SPRING et JAVA en général. URL : <https://www.baeldung.com/> (cf. p. 33).

Spring boot

- [17] *Spring Initializr*. URL : <https://start.spring.io/> (cf. p. 32).
- [18] *Spring Boot Getting started*. URL : <https://spring.io/guides/gs/spring-boot/> (cf. p. 32).
- [19] *Spring Boot Reference Guide*. URL : <https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/index.html> (cf. p. 32).
- [20] *Spring Boot API javadoc*. URL : <https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/api/> (cf. p. 32).

Thymeleaf

- [23] *Thymeleaf Website*. URL : <https://www.thymeleaf.org/index.html> (cf. p. 45).
- [24] *Using Thymeleaf Tutorial*. URL : <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html> (cf. p. 45, 47).
- [25] *Thymeleaf + Spring Tutorial*. URL : <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html> (cf. p. 45).
- [26] BAELDUNG. *Spring and Thymeleaf 3 : Expressions*. Explique l'utilisation des objets spéciaux comme `#lists`, `#dates...`. URL : <https://www.baeldung.com/spring-thymeleaf-3-expressions> (cf. p. 50).
- [27] BAELDUNG. *Thymeleaf : Custom Layout Dialect*. URL : <https://www.baeldung.com/thymeleaf-spring-layouts> (cf. p. 51).

Spring MVC

- [28] *Spring MVC officiel documentation*. URL : <https://docs.spring.io/spring/docs/5.1.9.RELEASE/spring-framework-reference/web.html#spring-web> (cf. p. 53).
- [29] BAELDUNG. *Spring MVC Guides*. URL : <https://www.baeldung.com/spring-mvc> (cf. p. 53).
- [30] BAELDUNG. *Spring Request Parameters with Thymeleaf*. URL : <https://www.baeldung.com/spring-thymeleaf-request-parameters> (cf. p. 53).
- [31] BAELDUNG. *Guide to Spring @Autowired*. URL : <https://www.baeldung.com/spring-autowire> (cf. p. 53).

Validation des beans

- [32] *All You Need To Know About Bean Validation With Spring Boot*. URL : <https://reflectoring.io/bean-validation-with-spring-boot/> (cf. p. 59).
- [33] *Liste des annotations standards*. URL : <https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html> (cf. p. 60).
- [34] *Liste des annotations proposées par JBoss*. URL : <https://docs.jboss.org/hibernate/stable/validator/api/> (cf. p. 60).

SGBD

- [39] *H2 website*. URL : <https://www.h2database.com/html/main.html> (cf. p. 82).

Spring JPA

- [40] Eugen PARASCHIV. *Introduction to Spring Data JPA*. URL : <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa> (cf. p. 83).
- [41] Thorben JANSSEN. *Hibernate Tips : How to map a Boolean to Y/N*. URL : <https://thoughts-on-java.org/hibernate-tips-how-to-map-a-boolean-to-y-n/> (cf. p. 89).
- [42] Krzysztof WOYKE. *Persisting Enums in JPA*. URL : <https://www.baeldung.com/jpa-persistingEnums-in-jpa> (cf. p. 89).
- [43] Rajeev SINGH. *JPA / Hibernate ElementCollection Example with Spring Boot*. URL : <https://www.callicoder.com/hibernate-spring-boot-jpa-element-collection-demo/> (cf. p. 93).
- [44] *JPA Repositories Official Documentation*. Explique la syntaxe pour les requêtes avancées. URL : <https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html> (cf. p. 95).
- [45] BAELDUNG. *Spring Data JPA @Query*. URL : <https://www.baeldung.com/spring-data-jpa-query> (cf. p. 95).
- [46] Thorben JANSSEN. *Ultimate Guide to JPQL Queries with JPA and Hibernate*. URL : <https://thoughts-on-java.org/jpql/> (cf. p. 96).

Rest

- [35] *Building REST services with Spring*. URL : <https://spring.io/guides/tutorials/rest/> (cf. p. 63).
- [36] BAELDUNG. *Test a REST API with curl*. URL : <https://www.baeldung.com/curl-rest> (cf. p. 64).
- [37] Eugen PARASCHIV. *Jackson – Bidirectional Relationships*. URL : <https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion> (cf. p. 66).

JavaScript et ses frameworks

- [38] *Vue.js website*. URL : <https://vuejs.org/> (cf. p. 77).

Tests

- [47] BAELDUNG. *Testing in Spring Boot*. URL : <https://www.baeldung.com/spring-boot-testing> (cf. p. 103).

i18n

- [48] BAELDUNG. *Guide to Internationalization in Spring Boot*. URL : <https://www.baeldung.com/spring-boot-internationalization> (cf. p. 109).

Sécurité

- [49] Ger ROZA. *Spring Security : Exploring JDBC Authentication*. URL : <https://www.baeldung.com/spring-security-jdbc-authentication> (cf. p. 118).

Index

@Autowired, 53
@Bean, 54
@Controller, 53
@Converter, 89
@DataJpaTest, 106
@ElementCollections, 93
@Embeddable, 88
@EmbeddedId, 88
@Entity, 83
@Enumerated, 89
@GeneratedValue, 88
@Id, 83
@IdClass, 88
@ManyToOne, 90
@Max, 59
@Min, 59
@MockBean, 107
@ModelAttribute, 57
@NotBlank, 59
@NotNull, 59
@OneToMany, 90
@Pattern, 60
@Positive, 59
@Query, 95
@Scope, 55
@SessionAttributes, 57
@SpringBootTest, 104
@Temporal, 89
@WebMvcTest, 108

ant, 16
application d'entreprise, 31
archéotype, 19
artifact, 18

DevTools, 38

eager, 92

fetch, 92

gradle, 15

java EE, 31
JPQL, 96
jpql, 96

lazy, 92
log, 25
lombok, 27

make, 15
mappedBy, 90
maven, 18
mockMVC, 108

orm, 81

serveur d'application, 31
serveur embarqué, 31
serveur web Java, 31
spring, 31

VSCode, 23
VSCodium, 23
Vue.js, 77

websocket, 121

yaml, 87