Applications orientées données

Version 2021.1

Raphaël Fournier-S'niehotta et Philippe Rigaux

sept. 29, 2021

Table des matières

Contents :

CHAPITRE 1

Introduction

Supports complémentaires :

- Diapos pour la session « Introduction au cours »
- Vidéo : https://mediaserver.cnam.fr/permalink/v125f3576b428k5j7wyd/

Ce document constitue le support du cours NSY135, intitulé « Applications orientées données », proposé par le département d'informatique du Conservatoire National des Arts et Métiers (Cnam).

- Il est rédigé et maintenu par Philippe Rigaux (philippe.rigaux@cnam.fr), Professeur au Cnam et Raphaël Fournier-S'niehotta, Maître de conférences au Cnam (fournier@cnam.fr)
- Le cours est proposé au Cnam sous diverses modalités, en présentiel et en formation à distance (selon années scolaires et semestres).
- Les supports de cours sont tous disponibles à partir du site http://www.bdpedia.fr/applications/, au format HTML
- Une version PDF complète du support est téléchargeable ici
- Informations sur les enseignements : http://deptinfo.cnam.fr/new/spip.php?rubrique547

1.1 Objectifs du cours

Le cours a pour objectif de montrer *par la pratique* comment intégrer les accès aux bases de données relationnelles dans des applications orientées-objet d'envergure développées selon les meilleures pratiques. Il est centré sur les outils dits *Object-Relational Mapping* (ORM) qui établissent automatiquement une conversion entre les données stockées dans une base relationnelle et les composants (objets) manipulés par les applications informatiques. L'un des plus connus de ces outils est Hibernate, qui sert de base à une grande partie des exemples fournis. Cependant, l'objectif est, au-delà d'un outil particulier, d'expliquer le pourquoi et le comment des outils de type ORM, et de comprendre comment ils s'intègrent à l'architecture générale d'une application complexe. En particulier, les aspects suivant sont explorés en détail et indépendamment des particularités d'un outil particulier :

- problèmes posés par la cohabitation d'une base relationnelle et d'une application orientée-objet ;
- méthodologie générique pour automatiser les conversions rendues nécessaires par cette cohabitation ;
- opérations de création, mise à jour et interrogation de données ;
- sensibilisation aux problèmes de performance et aux méthodes d'optimisation,
- problèmes spécifiques aux transactions ;
- et, enfin, questions architecturales relatives à l'intégration avec d'autres composants logiciels.

Pour illustrer ce dernier aspect, les outils ORM ne sont pas étudiés en isolation mais en combinaison avec la construction d'application. Le contexte choisi est celui des applications web dynamiques, implantées selon des motifs de conception comme le *Model-View-Controler* (MVC). Ces motifs ne sont abordés que de manière très simplifiée, mais permettent de bien montrer la mise en œuvre de solutions ORM dans une approche globale.

1.2 Pré-requis

Le cours suppose au préalable une connaissance solide des sujets suivants :

1. Conception et programmation objet, de préférence avec Java.
2. Concepts essentiels des bases de données relationnelles : conception, SQL, notion de transaction.
3. Une maîtrise de base des outils et langages du Web, notamment HTML, et des principes des applications Web dynamiques.

De plus le contenu du cours dans l'ensemble suppose un intérêt pour la conception d'application en général, dans leur aspect architectural en particulier. Des connaissances pointues en développement ne sont pas indispensables, mais il doit être clair que la mise en pratique des concepts étudiés implique des manipulations pour reproduire les exemples proposés et la réalisation pratique d'une application très simplifiée. Le cours suit une démarche pas à pas, consistant à mettre en œuvre des exemples fournis et étendus dans le cadre de l'enseignement. Celà étant, une ignorance complète des bases ci-dessus constituera un obstacle réel pour l'assimilation du contenu.

Le langage de programmation étudié par défaut est Java. Des exemples équivalents pourront être donnés en PHP occasionnellement.

1.3 Organisation du cours

Le cours est donné sous plusieurs modes :

1. en *présentiel*, une soirée par semaine, au Cnam ;
2. en *formation ouverte et à distance* (FOD), avec enregistrement des cours, support par les enseignants et réunions occasionnelles de consolidation.

L'orientation est pratique, avec notamment une mise en œuvre rapide des concepts présentés. Une *séance en présentiel* consiste donc typiquement en une alternance de *sessions* comprenant tout d'abord une présentation des concepts et outils, puis une mise en pratique par des exercices. Pour chaque session, la partie présentation est courte, au maximum 1/2 heure, la mise en pratique dépend de vos capacités mais est conçue pour couvrir un temps équivalent. Un cours au Cnam en présentiel consiste donc typiquement en 2 à 3 sessions.

Dans le cas d'un suivi du cours en FOD, il est recommandé de suivre le support dans l'ordre des chapitres, chaque chapitre comprenant quelques sessions (numérotées S1, S2, erc.) regroupées autour d'une thématique commune. L'unité de travail est la session. Pour chaque session, vous devriez suivre le cours enregistré, lire le contenu correspondant et effectuer les exercices proposés. Comptez 1h à 1h30 de travail pour chaque session, sachant que le cours enregistré et la lecture peuvent s'effectuer sans utiliser un ordinateur. Bien entendu vous devez disposer d'un ordinateur personnel. Tous les logiciels nécessaires sont libres de droit et fonctionnent sous tous les environnements.

Le découpage est le suivant.

1. Mise en place d'un environnement de développement : installation et configuration des différents outils.
2. Rappel sur la construction d'application Web, et mise en pratique d'une architecture MVC simplifiée basée sur l'environnement J2EE.
3. Mise en place d'une base relationnelle et accès à cette base avec l'interface de programmation standard (JDBC pour Java).
4. Remplacement de la couche d'accès aux données par une abstraction objet construite avec le composant ORM.
5. Exploration des fonctionnalités du composant ORM : modélisation, accès, performances, transactions.

À l'issue du cours, vous devriez maîtriser le pourquoi et le comment de la construction d'une application orientée données, être sensibilisés aux problématiques (performances notamment) de cette approche, en être capable de prendre rapidement en main les solutions professionnelles éprouvées pour de tels systèmes.

1.4 Évaluation

Le cours est évalué par :

- un examen sur table, de 2h sans document. Voici un exemple de sujet donné il y a quelques années.
- un projet (à choisir parmi deux sujets détaillés dans la page *Projets*)

Le projet et l'examen sont notés sur 20, la note finale de l'UE est la moyenne de ces deux notes.

CHAPITRE 2

Applications Web dynamiques

Commençons par une courte introduction au Web et à la programmation Web, limitée aux pré-requis indespensables pour comprendre la suite du cours. Vous trouverez facilement sur le Web beaucoup de documentation plus étendue sur ce sujet, avec notamment une perspective plus large sur les autres langages (CSS, JavaScript) intervenant dans la réalisation d'une application web. Les lecteurs déjà familiers avec ces bases peuvent sauter ce chapitre sans dommage.

2.1 S1 : Bases sur le fonctionnement du Web

Supports complémentaires :

- Présentation
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3576b502fakcbbi/

Le *World-Wide Web* (ou WWW, ou Web) est un très grand système d'information réparti sur un ensemble de *machines* connectés par le réseau Internet. Ce système est essentiellement constitué de *documents hypertextes* et de documents *multimedia* : textes, images, sons, vidéos, etc. Le Web propose aussi des services ou des modes de communication entre machines permettant d'effectuer des calculs répartis ou des échanges d'information sans faire intervenir d'utilisateur : le cours n'aborde pas ces aspects.

Chaque machine propose un ensemble plus ou moins important de documents qui sont transmis sur le réseau par l'intermédiaire d'un *programme serveur*. Le plus connu de ces programmes est peut-être Apache, un serveur *Open source* très couramment utilisé pour des sites simples (Blogs, sites statiques). D'autres programmes, plus sophistiqués, sont conçus pour héberger des applications Web complexes et sont justement désignés comme des *serveurs d'application* (par exemple, Glassfish, ou des logiciels propriétaires comme WebSphere).

Dans tous les cas, ce programme serveur dialogue avec un *programme client* qui peut être situé n'importe où sur le réseau. Le programme client prend le plus souvent la forme d'un *navigateur*, grâce auquel un utilisateur du Web peut demander et consulter très simplement des documents. Les navigateurs les plus connus sont Firefox, Internet Explorer, Safari, Opera, etc.

2.1.1 Fonctionnement

Le dialogue entre un programme serveur et un programme client s'effectue selon des règles précises qui constituent un *protocole*. Le protocole du Web est HTTP.

Tout site web est constitué, matériellement, d'une machine connecté à l'Internet équipée du programme serveur tournant en permanence sur cet ordinateur. Le programme serveur est en attente de requêtes transmises à son attention sur le réseau par un programme client. Quand une requête est reçue, le programme serveur l'analyse afin de déterminer quel est le document demandé, recherche ce document et le transmet au programme client. Un autre type important d'interaction consiste pour le programme client à demander au programme serveur d'exécuter un programme, en fonction de paramètres, et de lui transmettre le résultat.

La figure *Architecture Web* illustre les aspects essentiels d'une communication web pour l'accès à un document. Elle s'effectue entre deux programmes. La requête envoyée par le programme client est reçue par le programme serveur. Ce programme se charge de rechercher le document demandé parmi l'ensemble des fichiers auxquels il a accès, et transmet ce document.

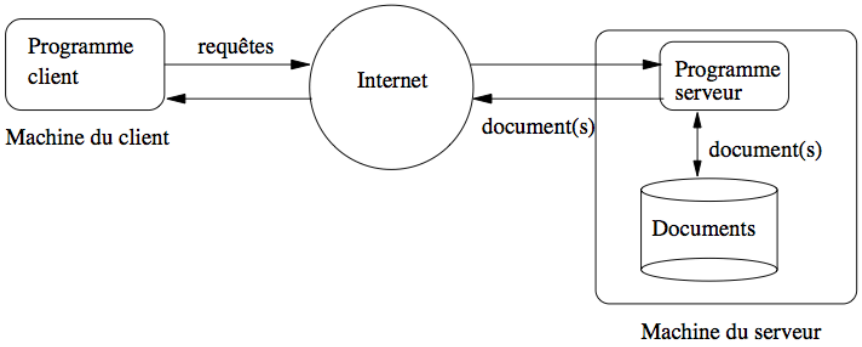

FIG. 1 – Architecture Web

Nous serons amenés à utiliser un serveur web pour programmer notre application.

1. Apache est peut-être le serveur web le plus répandu ; il est notamment très bien adapté à l'association avec le langage PHP.
2. Tomcat est un serveur d'application simple dédié à l'exécution d'application Web java ; c'est lui que nous utiliserons pour notre développement.

Dans tout ce qui suit, le programme serveur sera simplement désigné par le terme *serveur* ou par le nom du programme particulier que nous utilisons, Apache ou Tomcat. Les termes *navigateur* et *client* désigneront tous deux le programme client. Enfin le terme *utilisateur* sera réservé à la personne physique qui utilise un programme client.

2.1.2 Le protocole HTTP

HTTP, pour *HyperText Transfer Protocol*, est un protocole extrêmement simple, basé sur TCP/IP, initialement conçu pour échanger des documents hypertextes. HTTP définit le format des requêtes et des réponses. Voici par exemple une requête envoyée à un serveur Web :

```
GET /myResource HTTP/1.1
Host: www.example.com
```

Elle demande une ressource nommée `myResource` au serveur `www.example.com`. Et voici une possible réponse à cette requête :

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html>
  <head><title>myResource</title></head>
  <body><p>Hello world!</p></body>
</html>
```

Un message HTTP est constitué de deux parties : l'entête et le corps, séparées par une ligne blanche. La réponse montre que l'entête contient des informations qualifiant le message. La première ligne par exemple indique qu'il s'agit d'un message codé selon la norme 1.1 de HTTP, et que le serveur a pu correctement répondre à la requête (code de retour 200). La seconde ligne de l'entête indique que le corps du message est un document HTML encodé en UTF-8.

Le programme client qui reçoit cette requête affiche le corps du message en fonction des informations contenues dans l'entête. Si le code HTTP est 200 par exemple, il procède à l'affichage. Un code 404 indique une ressource manquante, une code 500 indique une erreur sévère au niveau du serveur, etc.

D'autres aspects du protocole HTTP incluent la protection par mot de passe, la négociation de contenu (le client indique quel type de document il préfère recevoir), des *cookies*, informations déposées côté client pour mémoriser les séquences d'échanges (sessions), etc. Le contenu du cours n'étant pas centré sur la programmation web, nous n'aurons à utiliser que des notions simples.

2.1.3 Les URLs

Les documents, et plus généralement les *ressources* sur le Web, sont identifiés par une *URL* (*Uniform Resource Locator*) qui encode toute l'information nécessaire pour trouver la ressource et aller la chercher. Cet encodage prend la forme d'une chaîne de caractères formée selon des règles précises illustrées par l'URL fictive suivante :

```
https://www.example.com:443/chemin/vers/doc?nom=orm&type=latex#fragment
```

Ici, `https` est le *protocole* qui indique la méthode d'accès à la ressource. Le seul protocole que nous verrons est HTTP (le `s` indique une variante de HTTP comprenant un encryptage des échanges). *L'hôte* (*hostname*) est `www.example.com`. Un des services du Web (le DNS) va convertir ce nom d'hôte en adresse IP, ce qui permettra d'identifier la machine serveur qui héberge la ressource.

Note : Quand on développe une application, on la teste souvent localement en utilisant sa propre machine de développement comme serveur. Le nom de l'hôte est alors `localhost`, qui correspond à l'IP `127.0.0.1`.

La machine serveur communique avec le réseau sur un ensemble de *ports*, chacun correspondant à l'un des services gérés par le serveur. Pour le service HTTP, le port est par défaut 80, mais on peut le préciser, comme sur l'exemple précédent, où il vaut 443.

Note : Par défaut, le serveur Tomcat est configuré pour écouter sur le port 8080.

On trouve ensuite le *chemin d'accès à la ressource*, qui suit la syntaxe d'un chemin d'accès à un fichier dans un système de fichiers. Dans les sites simples, « statiques », ce chemin correspond de fait à un emplacement physique vers le fichier contenant la ressource. Dans des applications dynamiques, les chemins sont virtuels et conçus pour refléter l'organisation logique des ressources offertes par l'application.

Après le point d'interrogation, on trouve la liste des paramètres (*query string*) éventuellement transmis à la ressource. Enfin, le fragment désigne une sous-partie du contenu de la ressource. Ces éléments sont optionnels.

Le chemin est lui aussi optionnel, auquel cas l'URL désigne la *racine* du site web. Les URL peuvent également être *relatives* (par opposition aux URLs absolues décrites jusqu'ici), quand le protocole et l'hôte sont omis. Une URL relative est interprétée par rapport au *contexte* (par exemple l'URL du document courant).

2.1.4 Le langage HTML

Les documents échangés sur le Web peuvent être de types très divers. Le principal type est le *document hypertexte*, un texte dans lequel certains mots, ou groupes de mots, sont des *liens*, ou *ancres*, référençant d'autres documents. Le langage qui permet de spécifier des documents hypertextes, et donc de fait le principal langage du Web, est HTML.

La présentation de HTML dépasse le cadre de ce cours. Il existe de très nombreux tutoriaux sur le Web qui décrivent le langage (y compris XHTML, la variante utilisée ici). Il faut noter que HTML est dédié à la *présentation* des documents d'un site, et ne constitue pas un langage de programmation. Son apprentissage (au moins pour un usage simple) est relativement facile. Par ailleurs, il existe de nombreux éditeurs de documents HTML qui facilitent le travail de saisie des balises et fournissent une aide au positionnement (ou plus exactement au pré-positionnement puisque c'est le navigateur qui sera en charge de la mise en forme finale) des différentes parties du document (images, menus, textes, etc). Notre objectif dans ce cours n'étant pas d'aborder les problèmes de mise en page et de conception graphique de sites web, nous nous limiterons à des documents HTML relativement simples.

Par ailleurs on trouve gratuitement des *templates* HTML/CSS définissant le « style » d'un site. Nous verrons comment intégrer de tels *templates* à notre application dynamique.

Voici un exemple simple du type de document HTML que nous allons créer. Notez les indications en début de fichier (`DOCTYPE`) qui déclarent le contenu comme devant être conforme à la norme XHTML :

```
<?xml version="1.0" encoding="utf8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Gestion de la persistance dans les applications</title>
<link rel='stylesheet' href="template.css" type="text/css"/>
</head>
<body>

<h1>Cours Frameworks/ORM</h1>

<hr/>

Ce cours aborde les ORM comme
<a href="http://hibernate.org">Hibernate</a>,
dans le cadre du développement d'aplications modernes
construites avec des <i>frameworks</i>.

</body>
</html>
```

La variante utilisée dans nos exemples est XHTML, une déclinaison de HTML conforme aux règles de constitution des documents XML. XHTML impose des contraintes rigoureuses, ce qui est un atout pour créer des sites portables et multi-navigateurs. Je vous conseille d'ailleurs d'équiper votre navigateur Firefox d'un validateur HTML qui vous indiquera si vos pages sont bien formées.

Important : Tous nos documents doivent être encodés en UTF 8, un codage adapté aux langues européennes. Il faut

utiliser un éditeur qui permet de sauvegarder les documents dans ce codage : dans notre cas ce sera Eclipse.

Exercice : quelques manipulations utiles avec Firefox.

Le navigateur que nous allons utiliser par défaut est Firefox qui a entre autres l'avantage d'être disponible (gratuitement) sur toutes les plateformes. Vous pouvez le télécharger et l'installer à partir de http://mozilla.org. Firefox peut être « étendu » avec des composants de toute sorte (*plugins*). Nous vous recommandons l'installation du *plugin Web Developer* : cette extension vous aide à analyser l'environnement et le statut de votre navigateur. Vous pouvez le récupérer à l'adresse https://addons.mozilla.org/fr/firefox/addon/web-developer/.

Une fois installé, *Web developer* se présente sous la forme d'une barre d'outils dans votre navigateur, ou dans un menu contextuel en cliquant avec le bouton droit. Voici maintenant quelques manipulations utiles pour aller plus loin que l'affichage de pages web (à vous de fouiller dans les menus Firefox et *Web developer* pour trouver comment effectuer chaque manipulation).

1. *Affichage du code source de la page*. Allez sur n'importe quel site et consultez le code source HTML. Avec Web Developer, l'option « Voir le code source généré » vous montre le code généré *localement* par le navigateur, par exemple par exécution de Javascript.
2. *Consultations des cookies*. Les *cookies* sont des informations (permettant notamment de vous identifier) déposées par les sites dans votre navigateurs. Il est instructif de les consulter…
3. *Entêtes HTTP*. Analysez les informations transmises dans l'entête HTTP du document que vous affichez.
4. *CSS*. Jetez un oeil au code CSS qui sert à la mise en page du document affiché, et effectuez quelques modifications avec *Web explorer*.

Il ne s'agit que d'une liste indicative. Nous vous invitons à consacrer 20 mns à explorer les autres outils proposés par *Web explorer* : cela peut s'avérer un gain de temps quand vous aurez à résoudre un problème pour votre application web plus tard.

Exercice : installation et utilisation de cURL.

`cURL` est un outil permettant de soumettre directement des requêtes HTTP à partir d'une console. Vous pouvez le récupérer sur le site http://curl.haxx.se/. Une fois installé, la commande `curl` sert à transmettre des requêtes HTTP. Voici par exemple comment effectuer un `GET` de la page d'accueil d'un site :

```
curl -X GET http://www.cnam.fr
```

L'option `-v` donne le détail des messages HTTP transmis et reçus. Voici quelques suggestions :

1. Effectuez la requête ci-dessus avec l'option `-v` et analysez les entêtes HTTP de la requête et de la réponse.
2. Utilisez cURL pour accéder à des services qui fournissent des données dans un format structuré en réponse à des requêtes HTTP. Par exemple :
   1. Les services de `freegeoip/net`. Exemple : `curl http://freegeoip.net/json/62.147.199.138`
   2. Un service de prévision météo, comme par exemple `OpenWeatherMap`. Exemple : `curl api.openweathermap.org/data/2.5/weather?q=London,uk`

   Etudiez les paramètres passés dans l'URL et tentez de faire des changements. Notez que le format de réponse n'est plus HTML, qui peut être vu comme un cas particulier (mais le plus courant) d'utilisation des protocoles du Web.

2.2 S2 : Applications Web et *frameworks*

Supports complémentaires :

— S2 : Présentation
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3576b55625t4r6a/

La programmation web permet de dépasser les limites étroites des pages HTML statiques dont le contenu est stocké dans des fichiers, et donc fixé à l'avance. Le principe consiste à produire les documents HTML par une *application* intégrée au serveur web. Cette application reçoit des requêtes venant d'un programme client, sous la forme d'URL associées à des paramètres et autres informations transmises dans le corps du message (cf. les services Web que vous avez dû étudier ci-dessus). Le contenu des pages est donc construit à la demande, *dynamiquement*.

2.2.1 Architecture

La figure *Architecture d'une application Web* illustre les composants de base d'une application web. Le navigateur (client) envoie une requête (souvent à partir d'un *formulaire* HTML) qui est plus complexe que la simple demande de transmission d'un document. Cette requête déclenche une *action* dans l'application hébergée par le serveur référencé par son URL. L'exécution de l'action se déroule en trois phases :

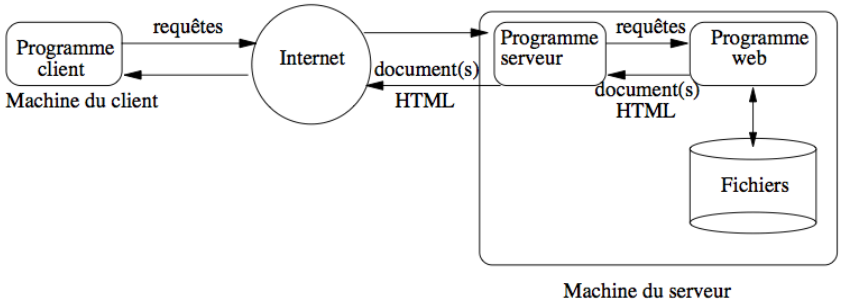

FIG. 2 – Architecture d'une application Web

1. *Constitution de la requête par le client* : le navigateur construit une URL contenant le nom du programme à exécuter, accompagné, le plus souvent, de paramètres saisis par l'internaute dans un formulaire ;
2. *Réception de la requête par le serveur* : le programme serveur récupère les informations transmises par le navigateur, et déclenche l'exécution du programme, en lui fournissant les paramètres reçus ;
3. *Transmission de la réponse* : le programme renvoie le résultat de son exécution au serveur sous la forme d'un document HTML, le serveur se contentant alors de faire suivre au client.

Le programme web est soit écrit dans un langage spécialisé (comme PHP) qui s'intègre étroitement au programme serveur et facilite le mode de programmation particulier aux applications web, soit écrit dans un langage généraliste comme Java, mais accompagné dans ce cas d'un ensemble de composants facilitant considérablement la réalisation des applications Web (comme JEE, *Java Enterprise Edition*). En particulier l'environnement de programmation doit offrir une méthode simple pour récupérer les paramètres de la requête, et autres informations (entêtes) transmis par HTTP. Il est alors libre de faire toutes les opérations nécessaires pour satisfaire la demande (dans la limite de ses droits d'accès bien sûr). Il peut notamment rechercher et transmettre des fichiers ou des images, effectuer des contrôles, des calculs, créer des rapports, etc. Il peut aussi accéder à une base de données pour insérer ou rechercher des informations. C'est ce dernier type d'utilisation (extrêmement courant) que nous étudions dans ce cours.

2.2.2 Et la base de données ?

La plupart des applications Web ont besoin de s'appuyer sur une base de données pour stocker des données persistantes : comptes clients, catalogues, etc. L'application Web, qu'elle soit en Java, PHP ou n'importe quel autre langage, fait donc intervenir un autre acteur, le *serveur de données* (le plus souvent un système relationnel comme MySQL), et communique avec lui par des requêtes SQL.

Dans le cas de PHP par exemple, il est possible à partir d'un script de se connecter à un serveur `mysqld` pour récupérer des données dans la base que l'on va ensuite afficher dans des documents HTM. D'une certaine manière, PHP permet de faire d'Apache un client MySQL, ce qui aboutit à l'architecture de la figure *Architecture d'une application Web avec base de données*.

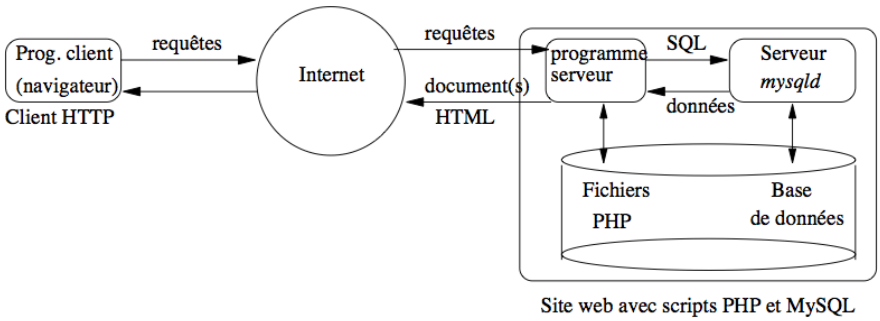

FIG. 3 – Architecture d'une application Web avec base de données

Il s'agit d'une architecture à trois composantes, chacune réalisant une des trois tâches fondamentales d'une application.

- le *navigateur* constitue *l'interface graphique* dont le rôle est de permettre à l'utilisateur de visualiser et d'interagir avec l'information ;
- MySQL est *le serveur de données* ;
- enfin l'ensemble des fichiers PHP contenant le code d'extraction, traitement et mise en forme des données est le *serveur d'application*, associé à Apache qui se charge de transférer les documents produits sur l'Internet.

2.2.3 Architecture et développement d'applications Web

La présentation qui précède n'est qu'une introduction extrêmement sommaire au monde de la conception et de la réalisation d'applications Web. À chaque niveau, on trouve une multitude de détails à gérer, relatifs aux protocoles d'échange, aux langages utilisés, à l'interaction des différents composants, etc. Pour faire simple à ce stade, et sur la base de ce qui est décrit ci-dessus, on peut noter des particularités essentielles qui caractérisent les applications Web :

- Ce sont des applications *interactives* et distribuées, à base d'échanges de documents hypertextes avec un protocole spécial, HTTP ;
- Plusieurs langages sont impliqués : au moins HTML, SQL, et le langage de programmation de l'application (plus éventuellement bien d'autres comme Javascript, les CSS, etc.) ;
- une application Web est ouverte sur le monde, ce qui soulève de redoutables problèmes de protection contre les messages nocifs, qu'ils soient volontaires ou non.

Ces particularités soulèvent des questions délicates relatives à *l'architecture* de l'application. Même si cette notion reste abstraite à ce stade, on peut dire qu'elle revient à trouver la meilleure manière de coordonner des composants chargés de rôles très différents (recevoir des requêtes HTTP, transmettre des réponses HTTP, accéder à la base de données), sans aboutir à un mélange inextricable de différents langages et fonctionnalités dans une même action.

Heureusement, cette problématique de l'architecture d'une application a été étudiée depuis bien longtemps, et un ensemble de *bonnes pratiques*, communément admises, s'est dégagé. De plus, ces bonnes pratiques sont encouragées, voire forcées, par des environnements de programmation dédiés aux applications Web, les fameux *frameworks*.

La partie qui suit est une courte introduction à la notion de *framework*. Elle est nécessairement abstraite et peut sembler extrêmement obscure quand on la rencontre hors de tout contexte. Pas d'affolement : essayez de vous imprégner des

idées principales et de saisir l'essentiel. Les choses se clarifieront considérablement quand nous aurons l'occasion de mettre les idées en pratique.

2.2.4 Les *frameworks* de développement

Un *framework* (littéralement, un *cadre de travail*), c'est un ensemble d'outils logiciels qui normalisent la manière de structurer et coder une application. Le terme regroupe trois notions distinctes et complémentaires

1. *Une conception générique* : une architecture, un modèle, une approche *générique* pour répondre à un besoin ou problème (générique) donné.

   La généricité garantit la portée universelle (dans le cadre du problème étudié, par exemple le développement d'applications web) de la conception. C'est une condition essentielle, sinon le *framework* n'est pas utilisable.

2. ***Des bonnes pratiques* : il s'agit, de manière moins formelle que le conception, d'un** ensemble de *règles*, de *conventions*, de *recommandations*, de *patrons* (*design patterns*), élaborés et affinés par l'expérience d'une large communauté, pour mettre en oeuvre la conception.

   Les bonnes pratiques capitalisent sur l'expérience acquise. Leur adoption permet d'éviter des erreurs commises dans le passé, et d'uniformiser la production du logiciel.

3. Et enfin (surtout ?), un *environnement logiciel* qui aide le développeur – et parfois le force – à adopter l'architecture/modèle et respecter les règles.

   La partie logicielle d'un *framework* fournit un support pour éviter des tâches lourdes et répétitives (par exemple le codage/décodage des messages HTTP), ce qui mène à un gain considérable en productivité.

L'adoption d'un *framework*, parce qu'il correspond à une forme de normalisation du développement, facilite considérablement le travail en équipe.

Note : On parle à tort et à travers de *frameworks*, notamment pour désigner des choses qui n'en sont pas (librairies, composants logiciels). Nous allons essayer d'être précis !

2.2.5 Caractérisation d'un framework

Un *framework* se distingue d'autres formes de composants logiciels (bibliothèques de fonction par exemple) par une caractéristique fondamentale, *l'inversion de contrôle*. Dans une application « régulière », le contrôle (c'est-à-dire le pilotage du flux de traitement en fonction des événements rencontrés) est géré par l'application elle-même. Elle décide par exemple d'afficher un questionnaire, puis de récupérer les réponses sur déclenchement de l'utilisateur, de les traiter, et enfin de produire un résultat. En chemin, des fonctionnalités externes, prêtes à l'emploi, peuvent être appelées, mais cela se fait sur décision de l'application elle-même. C'est le scénario (classique) illustré par la partie gauche de la figure *L'inversion de contrôle*.

Contrairement à une bibliothèque, un *framework* est un composant *actif* qui contrôle lui-même le déroulement de l'application, selon un modèle générique pré-établi. L'application qui utilise le framework se contente *d'injecter* des composants qui vont être utilisés le moment venu par la framework en fonction du déroulement des interactions.

Cela paraît abstrait ? Ça l'est, et cela explique deux aspects des *frameworks*, l'un négatif et l'autre positif :

1. *Côté négatif* : l'apprentissage (surtout initial) peut être long car le *framework* est une sorte de boîte noire à laquelle on confie des composants sans savoir ce qu'il va en faire. C'est - entre autres - parce que le contrôle échappe au développeur de l'application qu'il est difficile de s'y retrouver (au début).
2. *Côté positif* : l'inversion de contrôle signifie que toute une partie du comportement de l'application n'a plus *jamais* à être implantée, ce qui constitue un gain de productivité considérable. Entre autres avantages, un framework nous décharge de tâches lourdes, répétitives, et sources d'erreur de programmation ou de conception.

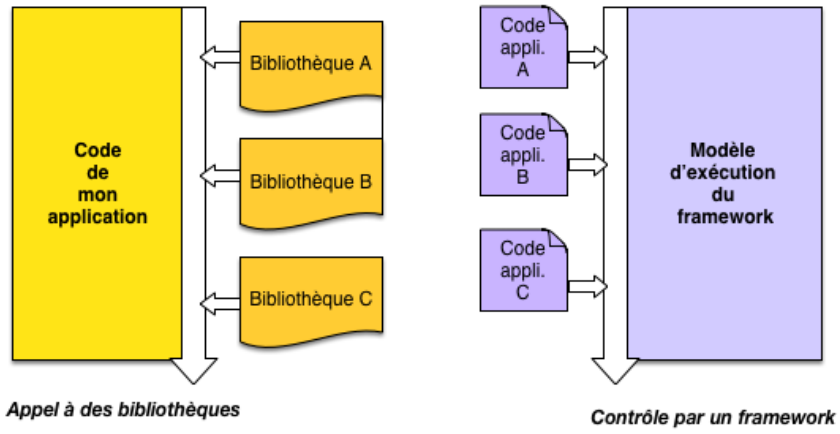

FIG. 4 – L'inversion de contrôle

L'inversion de contrôle n'est possible que pour des classes d'application obéissant à des comportements particuliers. C'est le cas par exemple des applications Web, dont le comportement est un cycle effectuant répétitivement la réception d'une requête HTTP, le traitement de cette requête, et la réponse (en HTTP) le plus souvent sous la forme d'une page HTML. Les frameworks web automatisent la gestion de ces cycles.

Pour conclure cette brève présentation sur un exemple, supposons que l'on veuille construire des questionnaires en ligne, avec fourniture d'un rapport chaque fois que le formulaire est rempli par un internaute. Une approche basique consiste à construire chaque formulaire par programmation avec des instructions du type :

```
out.println "<form action='...'>";
out.println "<input type='text' name='email'>Votre email</input>";
out.println "<input type='text' name='nom'>Votre nom</input>";
out.println "</form>";
```

Puis, il faut implanter du code pour récupérer les réponses et les traiter. Quelque chose comme :

```
for each (reponse in reponses) {
  // Il faut comprendre à quelle question correspond la réponse
  // Il faut ajouter la question et la réponse dans un joli rapport
}
```

Il est évident (?) qu'en repartant de zéro pour chaque questionnaire, on va produire *beaucoup* de code répétitif, ce qui est le signe que la production de questionnaires obéit en fait à une approche *générique* :

- construire des formulaires sous forme de question/réponses, en distinguant les différents types de réponse (libre/choix simple/choix multiple), leur caractère obligatoire ou non, etc.
- soumettre les formulaires à des internautes, avec toutes le options souhaitables (retour, correction, validation, etc.)
- enregistrer les réponses, produire un rapport (avec des mises en forme adaptées au besoin).

On en arrive donc rapidement à l'idée de construire un *framework* qui prendrait en charge tous les aspects génériques. Un tel framework devrait, s'il est bien conçu, permettre la construction d'un questionnaire en *injectant* simplement les questions à poser.

```
q = new Questionnaire(/* des options ...*/);

q->addChoixSimple ("Comment trouvez-vous ce cours?", ["Nul", "Moyen", "Parfait"]);
q->addQuestionLibre ("Avez-vous des commentaires ?");
```

(suite sur la page suivante)

(suite de la page précédente)

```
q->render();
```

Résultat : un code minimal, tout en bénéficiant d'une implantation efficace et puissante des fonctionnalités génériques pour ce type d'application. Détaillons maintenant les deux type de *framework* auxquels nous allons nous consacrer pour, respectivement, les applications web et les applications persistantes.

2.2.6 *Frameworks* pour applications Web

Le problème *générique* de base peut s'exprimer concisément ainsi : je veux créer une application Web *dynamique* qui produit des pages HTML créées à la volée en fonction de requêtes HTTP et alimentées par des informations provenant d'une base de données. Cette version basique se raffine considérablement si on prend en compte des besoins très courants (et donc formulables de manière générique également) : authentification, intégration avec un modèle graphique, gestion des messages HTTP, multi-langages, etc.

Une conception *générique* courante pour répondre à ce problème est l'architecture Modèle-Vue-Contrôleur (MVC) : mes pages HTML sont des *vues*, les interactions HTTP sont gérées par des *contrôleurs*, et la logique de mon application (persistante ou pas) est mon *modèle*. Le modèle MVC est sans doute le plus courant (mais pas le seul) pour le développement d'applications Web.

Les *frameworks* qui s'appuient sur ce modèle sont par exemple Symphony (PHP), Zend (PHP), Spring (Java), Grails (Groovy/Java), Django (Python), .NET (C#), Ruby on rails, etc. Outre le support des concepts de base (modèles, vues, contrôleurs), ils proposent tout un ensemble de bonnes pratiques et de conventions : le nommage des fichiers, des variables, organisation des répertoires, jusqu'à l'indentation, les commentaires, etc.

L'inversion de contrôle dans les *frameworks* web consiste à prendre entièrement en charge le cycle des échanges HTTP entre l'utilisateur (le client web) et l'application. En fonction de chaque requête, le *framework* détermine quel est le contrôleur à appliquer ; ce contrôleur applique des traitements implantés dans la couche « modèle », et renvoie au framework une « vue », ce dernier se chargeant de la transmettre au client.

2.2.7 *Frameworks* de persistance

Le problème (générique) peut se formuler comme suit : dans le cadre d'une application *orientée-objet*, je veux accéder à une base de données *relationnelle*. Les modes de représentation des données sont très différents : en objet, mes données forment un graphe d'entités dynamiques dotés d'un comportement (les méthodes), alors qu'en relationnel mes données sont organisées en tables indépendantes, et se manipulent avec un langage très particulier (SQL).

Cette différence de représentation soulève toutes sortes de problèmes quand on écrit une application qui doit « charger » des données de la base sous forme d'objet, et réciproquement stocker des objets dans une base. L'effet le plus concret est la lourdeur et la répétitivité du code pour effectuer ces conversions.

Le besoin (fort) est donc de rendre persistantes mes données objets, et « objectiser » mes données relationnelles. Les *frameworks* dits « de persistance » fournissent une méthode *générique* de transformation (*mapping*) des données relationnelles vers les données objet et vice-versa. Ils reposent là encore sur un ensemble de bonnes pratiques mises au point depuis des décennies de tâtonnement et d'erreurs. Ces bonnes pratiques s'appliquent à la représentation des associations, de l'héritage, à la traduction des *navigations* dans le graphe et *requêtes* sur les données, etc.

Ces *frameworks* sont dits ORM (*Object-Relational Mapping*). Les plus connus (courants) sont JPA/Hibernate (Java), JPA/EclipseLink (Java), Doctrine (PHP), CakePHP (PHP), ActiveRecords (Ruby), etc. Ils constituent le sujet principal du cours.

L'inversion de contrôle dans le cas des *frameworks* de persistance est la prise en charge de la conversion bi-directionnelle entre les objets et les tables, selon des directives de configuration / modélisation qui constituent les composants « injectés » dans le *framework*.

Etude : les différents types de *framework*

Cherchez sur le Web les principaux frameworks pour les classes d'application suivantes :

1. applications MVC (web)
2. applications persistantes (frameworks ORM)
3. application javascript (Ajax)
4. applications mobiles

Faites une synthèse présentée dans une page HTML, avec liens vers les sites que vous avez trouvées.

Etude : frameworks Web non-MVC

(Difficile). Existe-t-il d'autre modèles que le MVC pour des applications Web ? Etudiez la question et trouvez quelques exemples de *frameworks* correspondant.

2.3 Résumé : savoir et retenir

À ce stade, vous devez avoir acquis les notions suivantes :

1. Principes de base de HTTP : requête, réponse, entête, corps.
2. Forme des URL, signification des composants d'une URL.
3. Connaissance minimale de HTML ; compréhension du rôle de CSS.

Vous devez également avoir compris comment fonctionne une application Web, dans le principe. Le développement de telles applications implique beaucoup de langages et protocoles. Pour organiser ce développement, des motifs de conceptions (*design patterns*) et des bonnes pratiques ont été mises au point. Des logiciels, dits *frameworks*, sont destinés à nous guider dans l'application des ces motifs et pratiques.

Vous devez être à l'aise dans l'utilisation du navigateur Firefox, et être capable d'inspecter le document HTML affiché, les CSS utilisés, etc.

Lisez et relisez à l'avenir la section consacrée aux *frameworks* : elle contient des concepts abstraits d'abord difficile. Vous devez vous fixer pour objectif de les comprendre progressivement. La notion de généricité est essentielle. La notion d'inversion de contrôle est utile pour comprendre ce qui distingue un framework d'un autre type de composant logicielle (mais on peut utiliser un *framework* sans la connaître). Le cours est essentiellement consacré aux frameworks de persistance, mais les intègre dans une approche MVC simplifiée. Ces concepts particuliers seront revus en détail ensuite.

Tout cela est abstrait ? C'est justement le but du cours que de rendre ces avantages concrets. Le principal *framework* étudié sera celui assurant la connexion entre une application et une base de données (fonction dite « de persistance »), mais nous le placerons dans un contexte plus large, celui des *frameworks* de développement d'applications Web.

CHAPITRE 3

Environnement Java

Ce chapitre est particulier : il explique comment installer et configurer tous les outils de notre environnement de travail pour le développement d'une application Web, à l'exception de ce qui concerne la base de données : nous y reviendrons plus tard.

Comme nous allons travailler en Java, la présence d'un environnement de développement Java sur votre poste de travail est un pré-requis. En principe, ce devrait déjà être le cas, sinon allez sur la plateforme Java maintenue par Oracle et récupérez une version récente (supérieure ou égale à 6) du *Java Development Kit* (JDK). L'installation se fait en quelques clics.

La figure *Les outils de notre environnement* résume l'environnement que nous utilisons. La logique de notre processus de développement est la suivante :

1. Une application est gérée par un *projet* de l'environnement de développement intégré (IDE) Eclipse ; les fichiers correspondant à ce projet sont stockés, localement, sur votre disque dans un répertoire *workspace* affecté à Eclipse.
2. Quand nous sommes satisfaits d'un stade du développement, nous pouvons tester notre application en la *déployant* vers un serveur d'application Web ; dans notre cas, ce serveur est Tomcat.
3. Comme il s'agit d'une application Web, nous devons visualiser des pages HTML avec un client web ; il en existe un intégré à Eclipse, mais il est plus agréable à la longue d'utiliser un navigateur complet comme Firefox ou Chrome.

Eclipse est donc notre outil de contrôle général : il permet d'éditer le code, de contrôler son exécution sur le serveur, et même de visualiser un client Web. L'apprentissage des commandes Eclipse se fera tout au long des sessions. Comme tout apprentissage, il peut sembler fastidieux et frustrant au départ (les menus à parcourir pour trouver la bonne commande…), mais on se trouve rapidement récompensé.

Note : Si vous préferez utiliser un autre IDE, par exemple NetBeans, cela ne pose pas de problème. À vous bien entendu d'assumer dans ce cas cet apprentissage particulier.

Les sessions de ce chapitre consistent à installer les outils décrits ci-dessus, et à tester un premier projet, minimal.

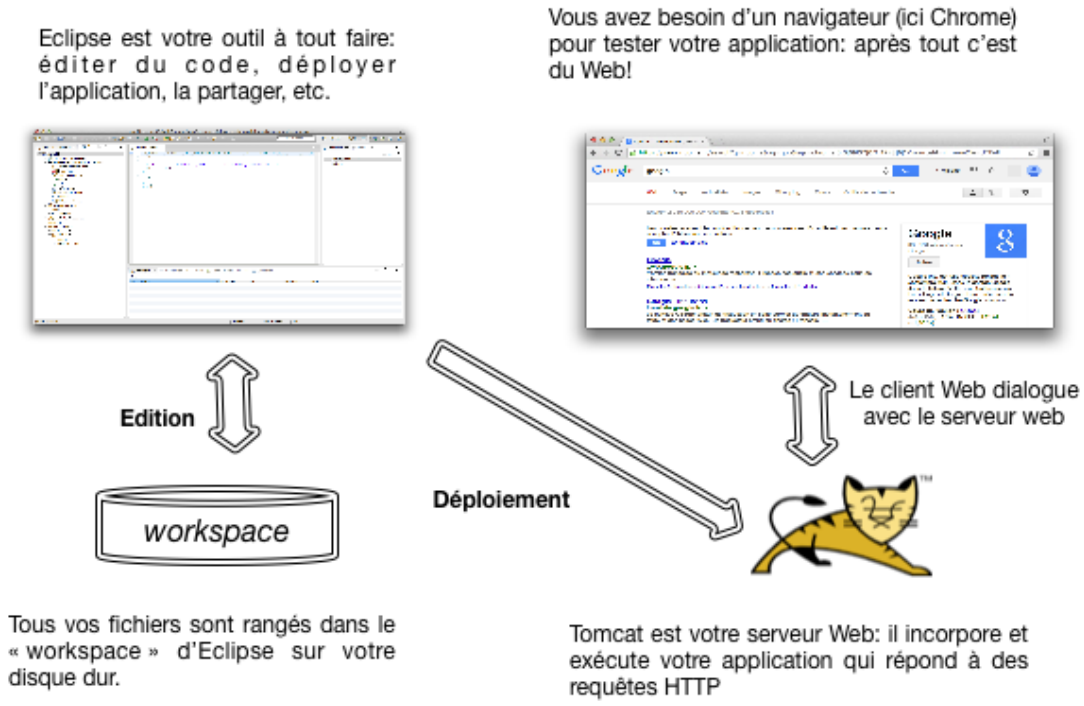

FIG. 1 – Les outils de notre environnement

3.1 S1 : Serveur d'application : Tomcat

Supports complémentaires :

- Diapos pour la session « S1 : Serveur d'application: Tomcat »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3576b4a1wj6eg0n/

Notre application web écrite en Java doit être intégrée à un serveur Web apte à interagir avec elle correctement. Nous allons utiliser Tomcat, un logiciel libre de droit proposé par la Fondation Apache.

Note : Tomcat n'est pas un serveur d'application JEE complet à proprement parler, mais il suffit largement à nos besoins. Pour un déploiement professionnel, un serveur comme Glassfish, WebSphere ou autre est préférable.

Comme d'habitude, l'installation consiste à se rendre sur le site de Tomcat. Allez à l'espace de téléchargement et choisissez la dernière version stable de Tomcat (au moins la 7) pour votre système.

On obtient une archive qui, une fois décompressée, donne un répertoire racine nommé `apache-tomcat-xxx` où `xxx` est un numéro de version.

Simplifions : renommez ce répertoire en `tomcat` et placez-le dans votre répertoire *Home*. Bien entendu, si vous pensez pouvoir vous en tirer avec une approche plus générale, faites !

Nous n'avons pas pour ambition de devenir des administrateurs émérites de Tomcat. Notre objectif est d'être capable d'installer et de de tester notre application sur le serveur local. Il peut quand même être intéressant (et surtout utile) de jeter un oeil à l'organisation des répertoires de Tomcat : voir la figure *Les répertoires de Tomcat (ici, version 7)*.

- le répertoire `bin` contient des programmes exécutables et notamment des scripts de démarrage et d'arrêt de Tomcat (`startup.sh` et `shutdown.sh`) ;
- `conf` contient les fichiers de configuration ;
- `log` contient les fichiers *log* ;

— enfin, `webapps` est le répertoire contenant les applications gérées par le serveur Tomcat.

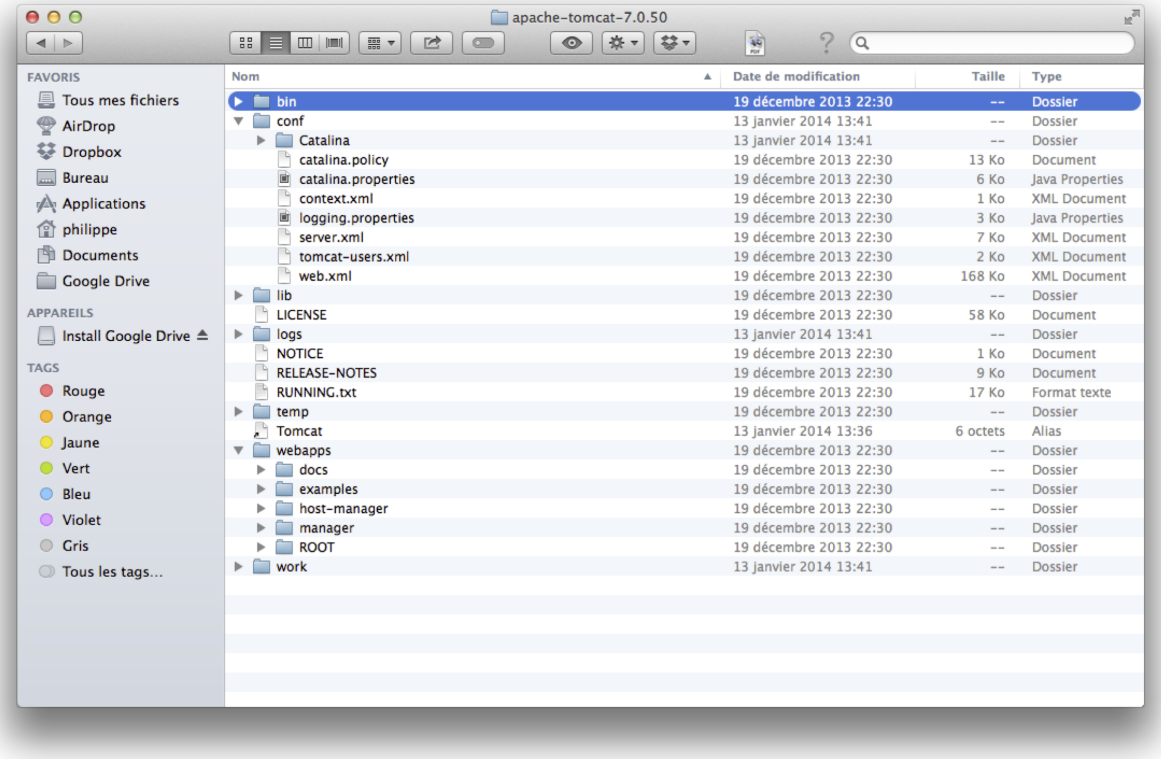

FIG. 2 – Les répertoires de Tomcat (ici, version 7)

Pour pouvoir démarrer et stopper Tomcat sans avoir à se déplacer dans le répertoire `bin` à chaque fois, il peut être utile de placer le chemin d'accès à ce dernier dans la variable d'environnement `PATH`. Par exemple, sous Linux ou Mac OS X :

```
export PATH=$PATH:$HOME/tomcat/bin
```

Placez cette commande dans un fichier d'initialisation de votre environnement (par exemple `.bashrc` si vous utilisez un environnement BASH shell).

Pas besoin d'en faire plus, au moins à ce stade (pour un déploiement réel il faudrait bien sûr se pencher sérieusement sur la question de la configuration de Tomcat). En fait nous allons piloter le serveur à partir d'Eclipse qui sera vraiment notre tour de contrôle.

Vérification : lancement et accès à Tomcat

Lancez Tomcat avec la commande `startup.sh`. Les messages d'information ou d'erreur (éventuelle) sont placés dans des fichiers dans le répertoire `tomcat/logs`. Si tout se passe bien, Tomcat se lance et se met en écoute sur le port 8080. Vous devriez avec votre navigateur accéder à l'URL `http://localhost:8080` et obtenir une page d'accueil donnant accès à la documentation. Rien ne vous empêche de lire cette dernière si vous avez un peu de temps.

Arrêtez Tomcat avec la commande `shutdown.sh` quand vous avez terminé.

3.2 S2 : L'environnement de développement : Eclipse

Supports complémentaires :

— Diapos pour la session « S2 : L'environnement de développement Eclipse »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3585f428qwbgftq/

Tout développement peut être vu de manière simpliste comme une séquence de tâches élémentaires : édition du code, compilation, test, correction, etc. On peut effectuer ces tâches avec des outils rudimentaires comme le Bloc Note Windows, `vi` sous Unix, et un terminal pour entrer des lignes de commande… Heureusement, il existe des environnements de développement dédiés qui allègent considérablement toutes ces tâches. Inutile d'énumérer tous leurs avantages : vous serez rapidement convaincus à l'usage.

Nous allons utiliser l'un des plus connus, Eclipse (un autre candidat sérieux pour du développement Java est Netbeans).

3.2.1 Installation

Si Eclipse n'est pas déjà présent sur votre poste, rendez-vous sur le site https://www.eclipse.org/downloads/packages/ (section : Eclipse IDE for Enterprise Java and Web Developers) et choisissez l'installeur correspondant à votre système (Windows, MacOS, Linux).

Renommez le répertoire racine obtenu en décompressant l'archive `eclipse` (par simplicité) et placez-le dans votre répertoire racine (par simplicité encore).

Et voilà. Dans le répertoire `eclipse` vous trouverez un exécutable que vous pouvez lancer pour bénéficier de l'affichage de la page d'accueil. Maintenant, il faut faire un peu de configuration.

3.2.2 Configuration

Nous allons pré-régler quelques options utiles pour éviter des problèmes ultérieurs.

Encodage UTF-8

Tous nos documents vont être encodés en UTF-8. C'est une bonne habitude à prendre pour éviter des problèmes d'affichage (et aussi éviter de se poser des questions).

— accédez au menu `Préférences` d'Eclipse, en principe sous le menu `Windows` (sauf sous Mac OS…)
— entrez `encoding` dans la fonction de recherche ; une liste d'options s'affiche ;
— pour chaque option, repérer le formulaire de réglage de l'encodage, et choisissez UTF-8.

La figure *Choix de l'encodage par défaut* montre une des fenêtres de configuration en cours de paramétrage.

Réglages de l'éditeur de code

Vous pouvez (toujours dans les `Préférences`), configurer l'éditeur de code, par exemple pour afficher les numéros de ligne. Accédez à la fenêtre `General -> Editors -> Text Editor` (figure *Réglage de l'éditeur de code*). Notez que le sous-menu `Spelling` permet d'activer/désactiver la correction orthographique, à votre convenance.

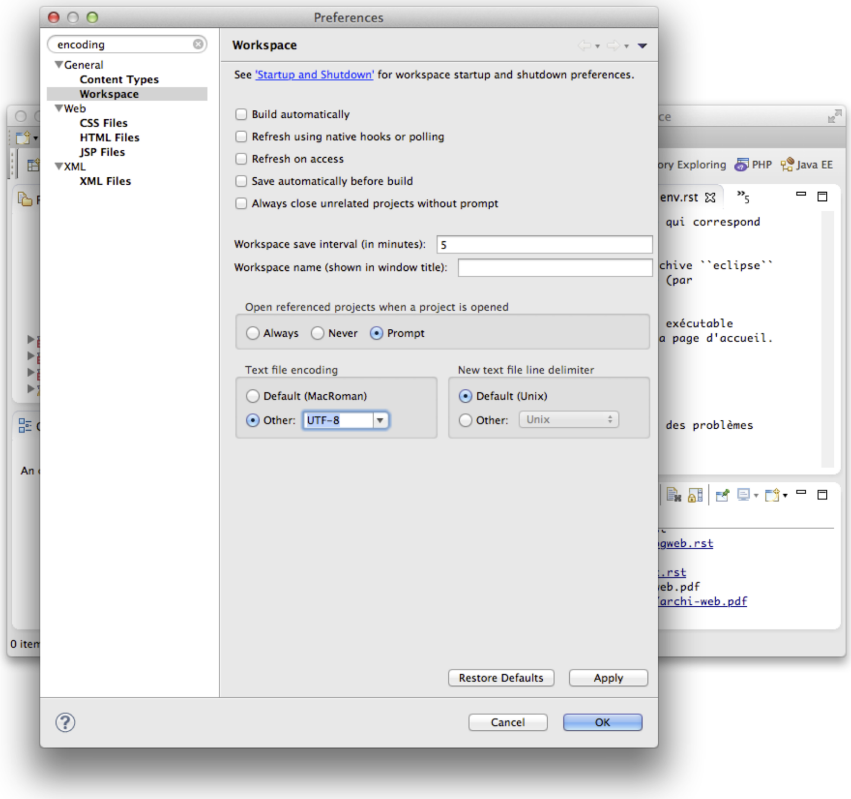

FIG. 3 – Choix de l'encodage par défaut

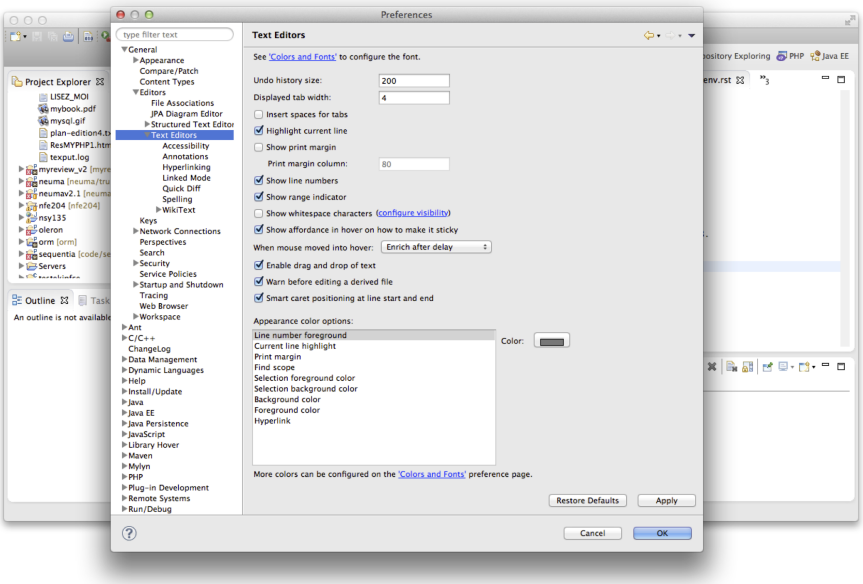

FIG. 4 – Réglage de l'éditeur de code

Association Eclipse/Tomcat

Pendant nos développements, nous testerons chaque étape en déployant l'application sur le serveur Tomcat. Eclipse est capable de faire cela automatiquement, et même de prendre en compte en continu les modifications du code pour les installer dans le serveur.

Il faut au préalable configurer un environnement d'exécution (*Runtime environment*) dans Eclipse. pour cela, toujours à partir du menu `Préférences`, accéder à `Servers -> Runtime environments`. On obtient la fenêtre de la figure *Configuration d'un serveur dans Eclipse*.

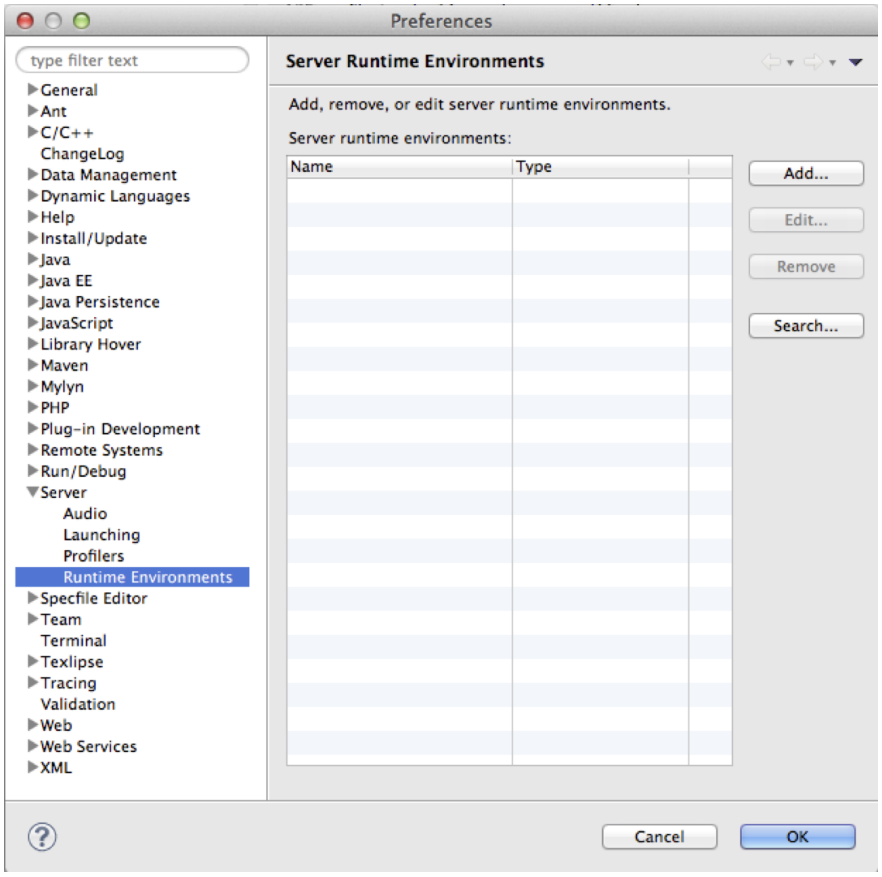

FIG. 5 – Configuration d'un serveur dans Eclipse

Suivre le bouton `Add`, ce qui mène à la fenêtre de la figure *Choix du serveur dans Eclipse*. Sélectionner Apache Tomcat dans la liste de tous les conteneurs JEE connus par Eclipse. La fenêtre suivante vous demandera le répertoire d'installation de Tomcat, autrement dit votre répertoire *Home* si vous avez scrupuleusement suivi les instructions ci-dessus.

Note : Cocher l'option `Create a local server` n'est utile que si vous partagez l'installation de Tomcat avec d'autres utilisateurs ou d'autres applications. Dans ce cas il peut être gênant de démarrer/stopper le serveur pour vos tests, en perturbant les autres. Si vous utilisez une machine personnelle, pas de problème.

Et voilà, la beauté de la chose est que vous pouvez maintenant démarrer ou stopper votre instance de Tomcat depuis Eclipse. Ajouter une vue `Server` dans la fenêtre en bas à droite, avec le menu `Window -> Show view -> Servers` comme montré sur la figure *Démarrage/arrêt de Tomcat depuis Eclipse*. Un onglet *Servers* apparaît avec la liste des serveurs configurés à l'étape précédente. Cliquez sur le nom du serveur. La flèche verte dans la barre des onglets sert à démarrer un serveur, un rectangle rouge à le stopper.

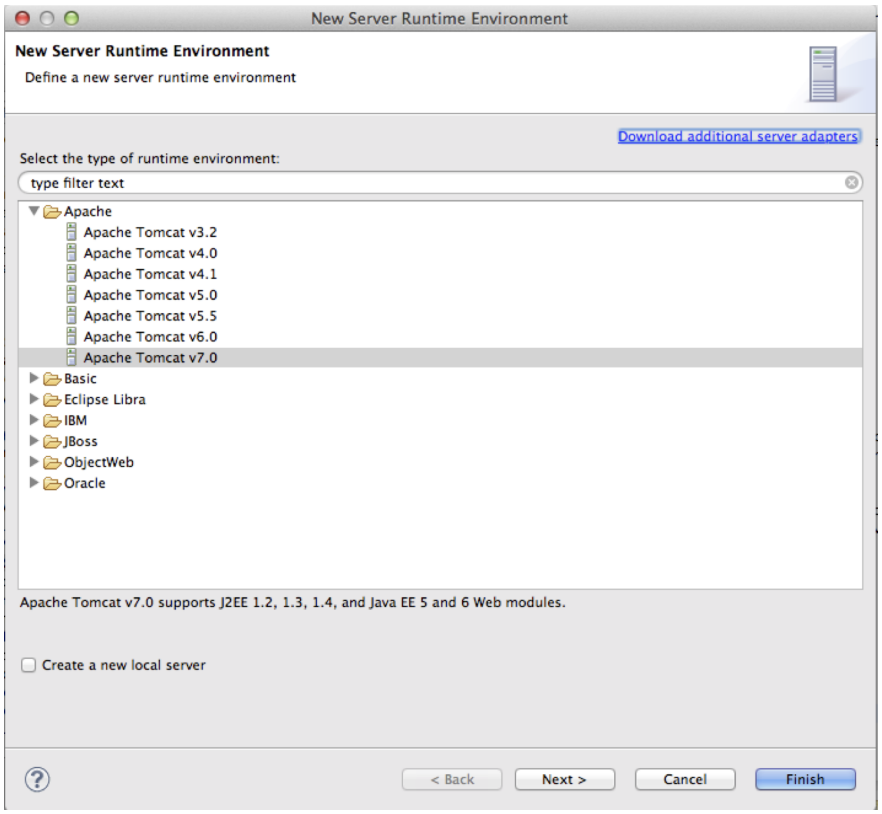

FIG. 6 – Choix du serveur dans Eclipse

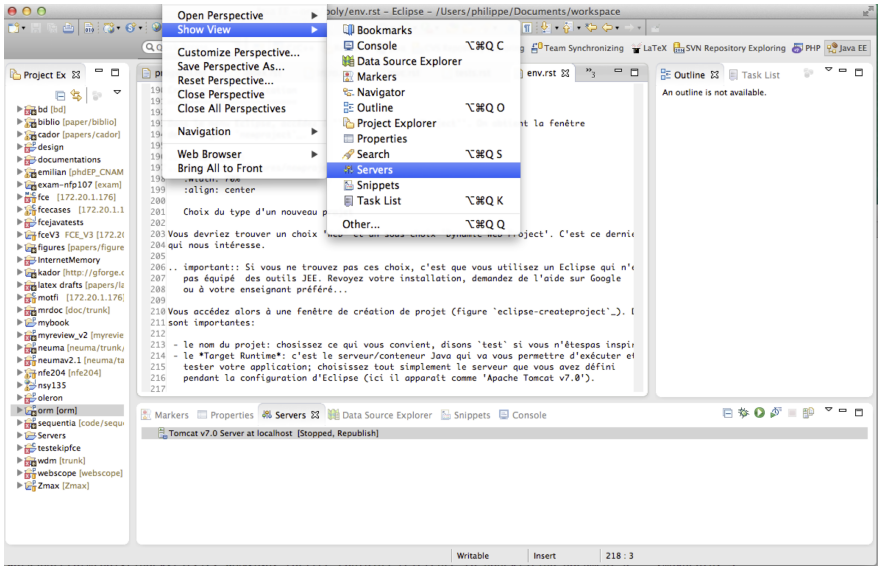

FIG. 7 – Démarrage/arrêt de Tomcat depuis Eclipse

Nous voici dotés d'un environnement d'exécution prêt à l'emploi. Il ne reste plus qu'à lui fournir une application.

Vérification : pilotage de Tomcat par Eclipse.

Vérifiez que vous pouvez lancer Tomcat avec Eclipse, accéder à l'URL http://localhost:8080 avec votre navigateur préféré, puis stopper Tomcat (avec Eclipse).

3.3 S3 : Création d'une application JEE

Supports complémentaires :

- Diapos pour la session « S3 : Création d'une application JEE »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3585f4a8jlzrhns/

Le moment est enfin venu de créer notre première application JEE avec notre Eclipse tout frais. Nous allons créer une application par défaut et faire quelques essais pour vérifier que tout fonctionne.

3.3.1 Création de l'application

Dans le menu Eclipse, accédez à `File -> New -> Project`. On obtient la fenêtre de la figure *Choix du type d'un nouveau projet*.

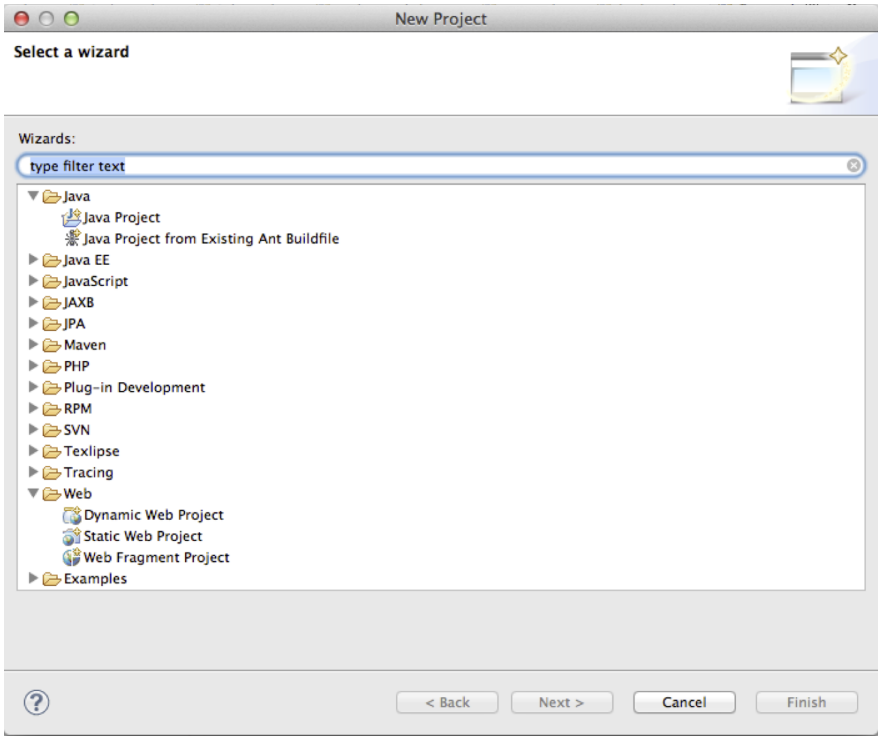

FIG. 8 – Choix du type d'un nouveau projet

Vous devriez trouver un choix "*Web*" et un sous-choix "*Dynamic Web Project*". C'est ce dernier qui nous intéresse.

Important : Si vous ne trouvez pas ces choix, c'est que vous utilisez un *package* Eclipse qui n'est pas équipé des outils JEE. Revoyez votre installation, demandez de l'aide sur Google ou à votre enseignant préféré…

Vous accédez alors à une fenêtre de création de projet (figure *La fenêtre de création de projet*). Deux options sont importantes :

— le nom du projet : chosissez ce qui vous convient, disons *test* si vous n'êtes pas inspiré(e) ;
— le *Target Runtime* : c'est le serveur/conteneur Java qui va vous permettre d'exécuter et de tester votre application ; choisissez tout simplement le serveur que vous avez défini pendant la configuration d'Eclipse (ici il apparaît comme « Apache Tomcat v7.0 »).

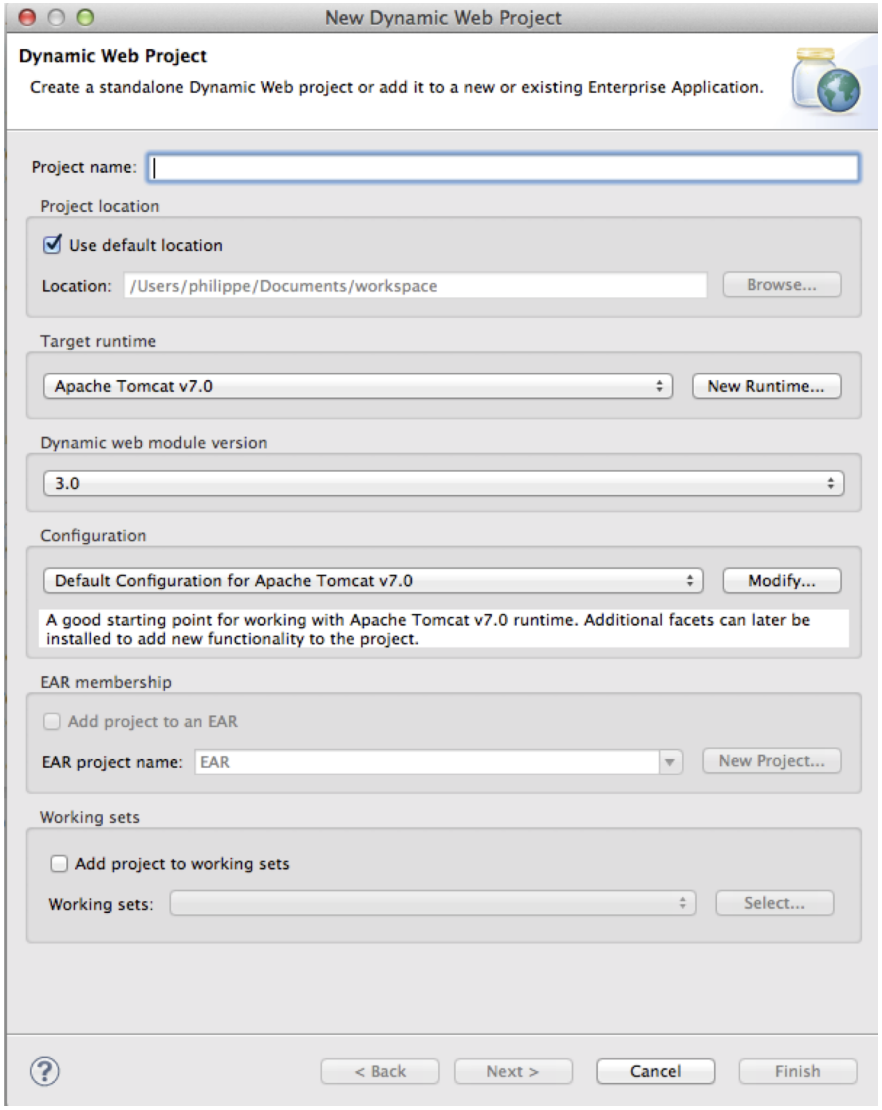

FIG. 9 – La fenêtre de création de projet

Pour le reste, gardez les choix par défaut et validez. Le projet doit alors apparaître dans la liste de gauche d'Eclipse. Cliquez-droit sur le nom du projet et choissez `Run As -> Run on Server` dans la liste des actions qui s'affiche. Vous devriez obtenir la fenêtre de la figure *Exécuter une application sur un serveur*.

Est-ce assez clair ? Eclipse vous propose d'exécuter votre application (vide pour l'instant) sur un des serveurs prédéfinis. Vous pouvez même indiquer que vous souhaitez toujours utiliser le même serveur, ce qui simplifie encore la

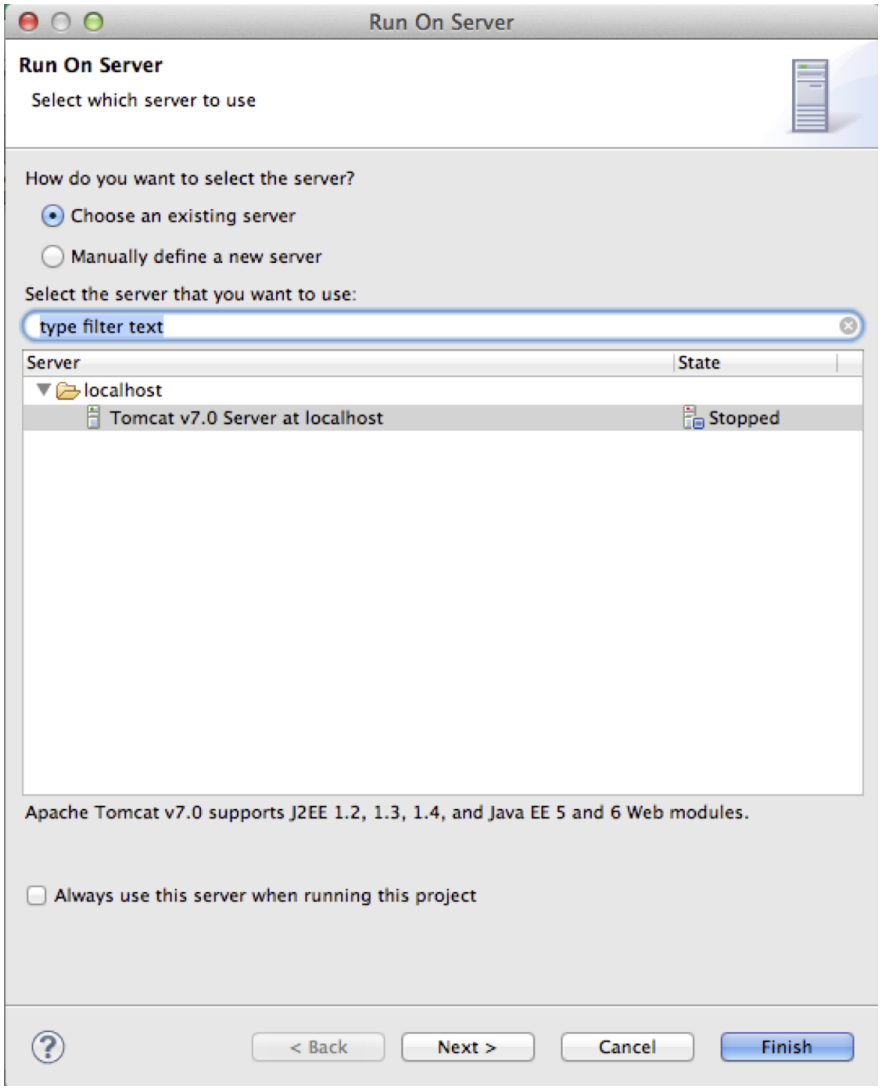

FIG. 10 – Exécuter une application sur un serveur

tâche.

Choisissez votre serveur et continuez. La première fois, une fenêtre intermédiaire affiche la liste des applications configurées pour ce serveur (vérifiez que la vôtre y figure). Puis le serveur Tomcat est lancé, et votre application s'exécute, produisant une belle erreur HTTP 404. Pas d'inquiétude, c'est normal puisqu'à ce stade il n'existe aucune ressource à afficher.

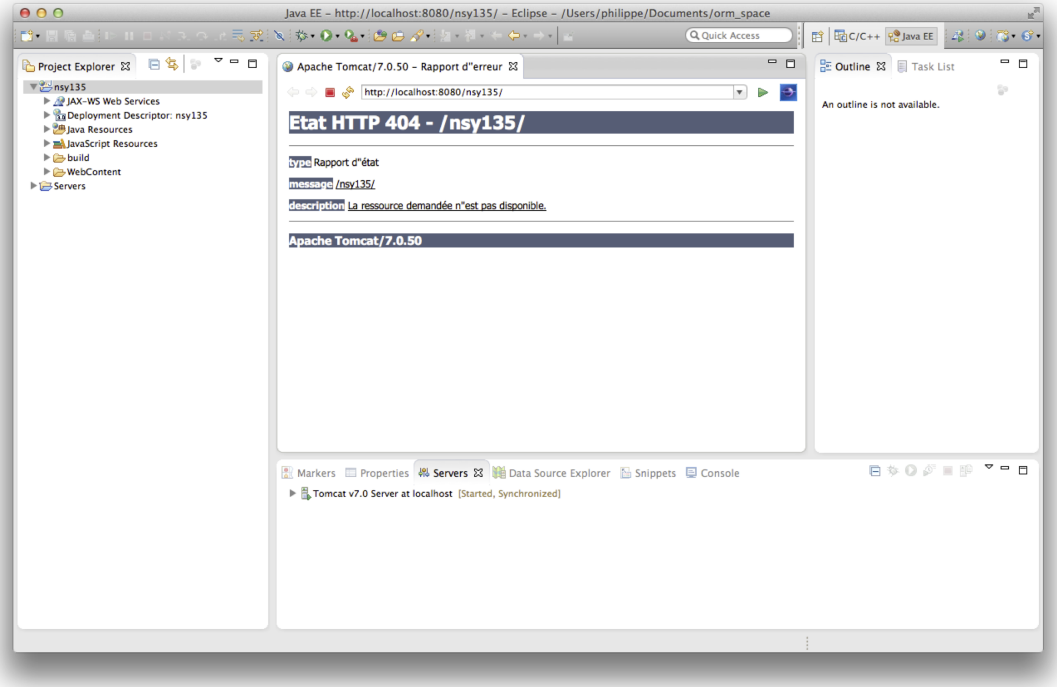

FIG. 11 – Exécution d'une application (vide)

Eclipse devrait afficher à peu près le contenu de la figure *Exécution d'une application (vide)*. Nous avons

— dans la liste de gauche (les projets), votre nouvelle application, ici nommée `nsy135` ; notez un début d'arborescence, détaillé plus loin ;
— dans la fenêtre centrale, l'affichage par un navigateur embarqué dans Eclipse de la page d'accueil de votre application (soit une erreur 404 pour l'instant) ; vous pouvez aussi utiliser Firefox ou un autre navigateur, à l'adresse http://localhost:8080/nsy135 (évidemment, remplacez `nsy135` par le nom de votre application) ;
— enfin, la fenêtre au centre en bas montre, dans l'onglet `Servers`, votre instance de Tomcat en cours d'exécution.

Bravo, vous venez de mettre en place un environnement de développement et de test pour une application JEE. L'étape suivante consiste à lui donner du contenu.

3.3.2 Qu'est-ce qu'une application JEE ?

La figure suivante montre les répertoires constituant l'arborescence standard d'une application JEE telle qu'elle doit être fournie à un serveur d'application comme Tomcat. Le répertoire racine porte le nom de votre application (c'est par exemple `nsy135` dans les manipulations précédentes). Vous avez ensuite un sous-répertoire `WEB-INF` qui contient, pour le dire brièvement, tout ce qui constitue la configuration et la partie dynamique de votre application. *Ce répertoire n'est pas accessible par HTTP*, et vous êtes donc assurés que le client Web (ou toute application externe en général) ne pourra pas récupérer les ressources qui s'y trouvent.

Notez par exemple que `WEB-INF` a un sous-dossier `lib` qui doit contenir toutes les librairies (`jar` ou autres) dont votre application dépend, et un sous-dossier `classes` contenant vos propres classes compilées. Le fichier `web.xml` est un

fichier de configuration que nous n'avons pas besoin de modifier pour ce cours.

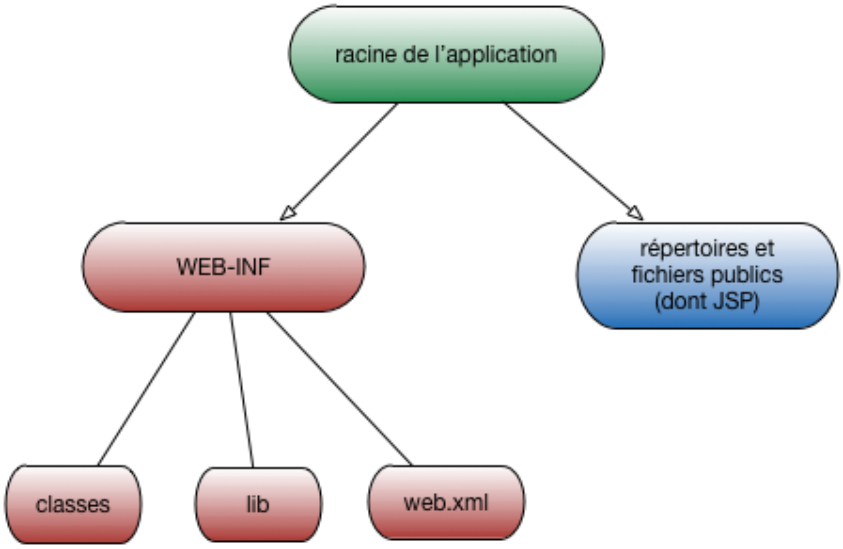

FIG. 12 – Structure d'une application JEE

Tout ce qui n'est pas dans `WEB-INF` contient la partie statique/publique de votre application Web, autrement dit les ressources directement accessibles par une URL avec le navigateur : fichiers de style (CSS), code javascript, images, fichiers à télécharger éventuels, voire des pages HTML si votre application en contient.

3.3.3 Et dans Eclipse ?

Dans Eclipse, nous retrouvons cette structure, plus, bien entendu, le code de l'application lui-même, ce qui donne la hiérarchie de répertoires de la figure *Structure d'une application JEE dans Eclipse*. Tout ce qui est sous `src/main/webapp` correspond donc au contenu qui doit être « *packagé* » et transmis au serveur d'application (Tomcat). Ne vous inquiétez pas : c'est Eclipse qui s'en charge.

Quand on exécute une application, le code nouveau ou modifié est compilé, transféré sous `src/main/webapp/WEB-INF/classes`, puis l'ensemble du contenu de `src/main/webapp/` est mis à disposition de Tomcat qui peut alors exécuter l'application. Tout cela se fait en un clic, sans avoir à se soucier des détails.

3.3.4 Premier pas : création d'un contenu statique

Pour vérifier que tout fonctionne nous allons créer quelques fichiers statiques et les déployer. Avec le menu contextuel (bouton droit) de votre projet, choisissez la création d'un nouveau fichier HTML (figure *Création d'un nouveau document HTML*).

Eclipse vous propose gentiment une structuration de base pour votre document. Complétez-là comme dans l'exemple ci-dessous, et sauvegardez le fichier dans `src/main/webapp/maPage.html`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
     Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Ma première page HTML</title>
```

(suite sur la page suivante)

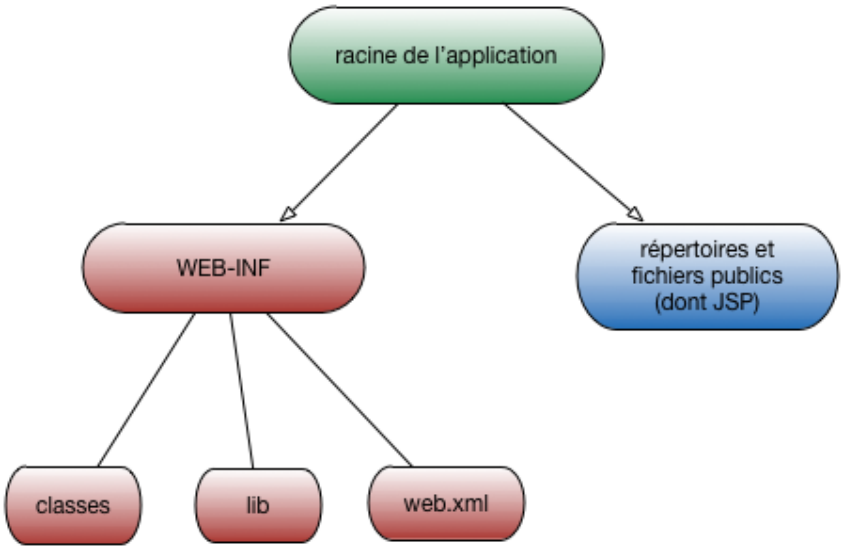

FIG. 13 – Structure d'une application JEE dans Eclipse

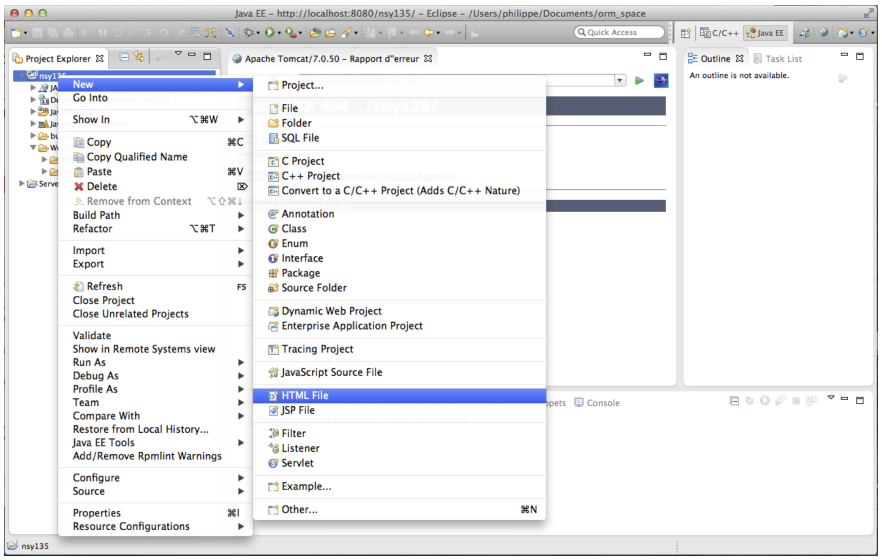

FIG. 14 – Création d'un nouveau document HTML

(suite de la page précédente)

```
 </head>
 <body>
  Ma première page HTML pour mon application JEE.
 </body>
</html>
```

What else ? Et bien rien du tout. En accédant avec un navigateur à l'adresse http://localhost:8080/nsy135/maPage.html, vous devriez simplement voir s'afficher votre document HTML. Au passage, cela nous montre qu'Eclipse déploie automatiquement toute modification du code de l'application sur le serveur Tomcat associé, ce qui permet de vérifier tout de suite l'impact du développement.

Note : Cela suppose bien entendu que vous ayez au préalable exécuté votre application avec l'option *Run on server* comme indiqué précédemment.

Vous voici prêts à faire un premier exercice par vous-mêmes.

Exercice : installation d'un contenu statique complet.

Le but est de mettre en place un contenu statique complet, avec images, CSS, et peut-être Javascript. Pour cela, cherchez sur le Web un *template* de site Web prêt à l'emploi et gratuit. Vous en trouverez, par exemple, sur http://www.freecsstemplates.org. Récupérez le code et installez le *template* dans votre application JEE. Au final, http://localhost:8080/nsy135/ pourrait ressembler à la figure *Notre application avec un modèle de site prêt à l'emploi.*, avec bien entendu le modèle que vous avez choisi (astuce : si le nom du fichier HTML n'est pas donné dans l'URL, Tomcat prend le fichier `index.html` si ce dernier existe). Profitez-en pour effectuer quelques modifications du code HTML ou CSS, et vous familiariser avec les commandes d'Eclipse.

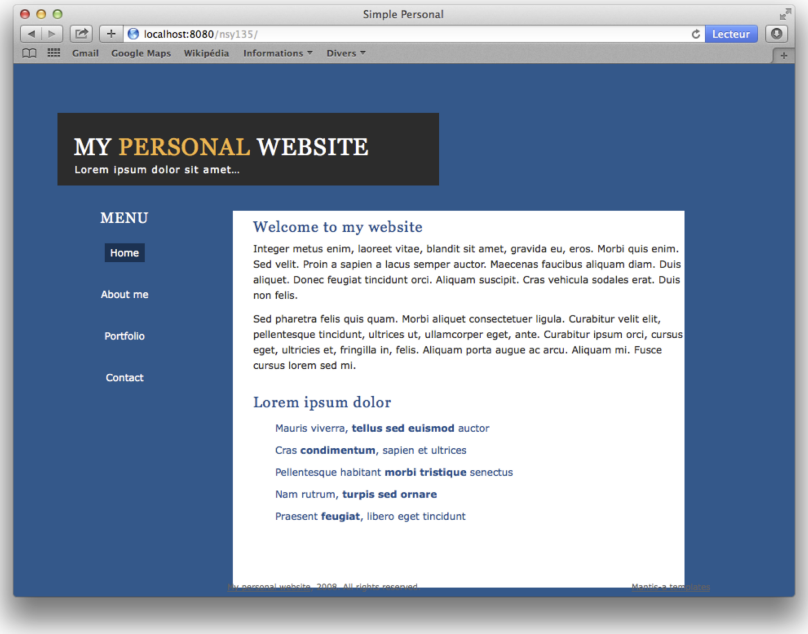

FIG. 15 – Notre application avec un modèle de site prêt à l'emploi.

3.3.5 Premier pas, bis : création d'un contenu dynamique

Bien entendu, notre but est de développer une application *dynamique* qui va produire du contenu HTML à la volée en réponse aux actions utilisateurs. Cela demande une certaine méthodologie qui va être exposée par la suite, mais pour commencer nous allons créer un premier composant qui servira de base à des discussions initiales.

Ce premier composant est une *servlet*. Qu'est-ce qu'une servlet ? Très concrètement, c'est un objet (Java) qui parle le HTTP. Il est pré-équipé d'une interface qui lui permet de recevoir des requêtes transmises par un client web, et de répondre à ce client, en général sous la forme d'un document HTML.

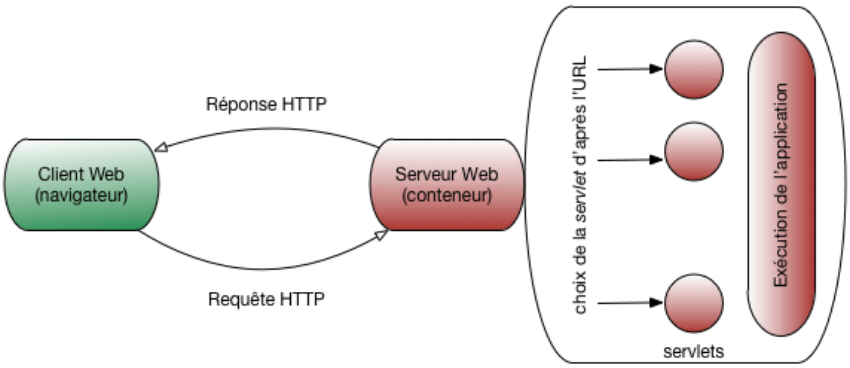

FIG. 16 – Un conteneur d'application et ses *servlets*

Pendant l'exécution d'une application, une servlet est un objet contenu dans le serveur (d'où le nom de *conteneur* parfois donné à ce dernier), et la configuration de l'application indique à quelle URL cette *servlet* sait répondre. Quand une requête est transmise, le serveur (disons Tomcat), analyse l'URL, détermine quelle est la *servlet* appropriée, et transmet la requête à cette dernière (figure *Un conteneur d'application et ses servlets*).

Soyons concrets et créons une classe pour notre première *servlet*. Comme d'habitude, Eclipse est notre ami et nous facilite le travail. Choisissez l'option *New* dans le menu contextuel de votre application, et cherchez *Servlet*. Vous obtenez la fenêtre de la figure *Fenêtre de création d'une servlet*.

Il ne reste qu'à choisir le nom du *package* et celui de la classe. Disons que le premier est `tests` et le second `Basique`. On arrive alors à une seconde fenêtre, ci-dessous.

Ce qui est important ici c'est le champ *URL mapping*. Il indique le chemin qui va déterminer le déclenchement de la *servlet* par le conteneur d'application. Le choix par défaut est d'utiliser le nom de la classe (ici `Basique`). Cliquez directement sur `Finish`, et Eclipse crée la classe Java suivante, que nous copions intégralement pour pouvoir l'examiner.

```
1  package tests;
2
3  import java.io.IOException;
4  import javax.servlet.ServletException;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.HttpServlet;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 /**
11  * Servlet implementation class Basique
12 */
13 @WebServlet(description = "Une première servlet pour voir ce qui se passe",
14    urlPatterns = { "/Basique" })
```

(suite sur la page suivante)

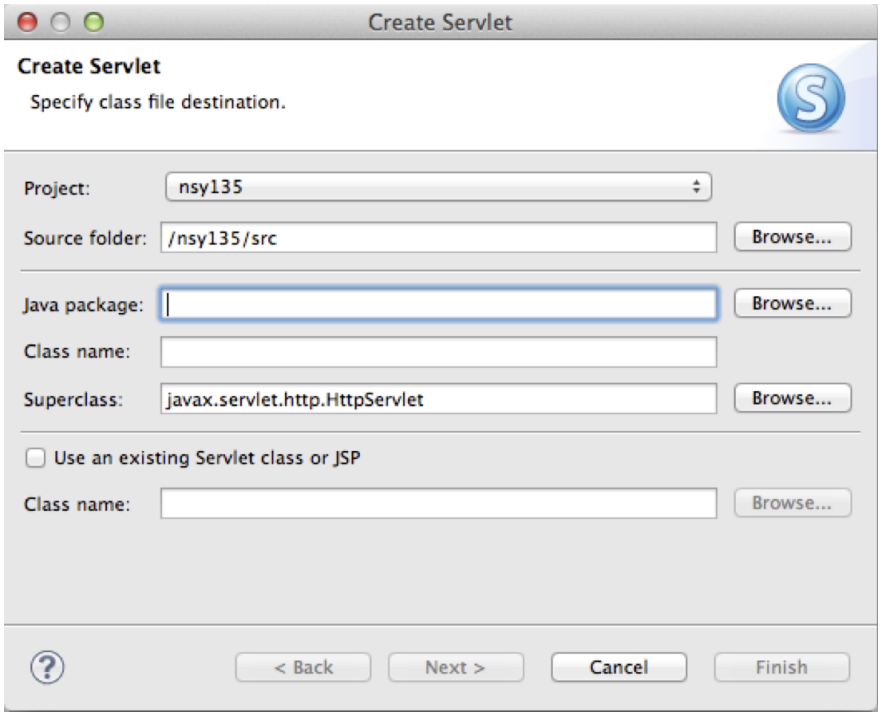

FIG. 17 – Fenêtre de création d'une *servlet*

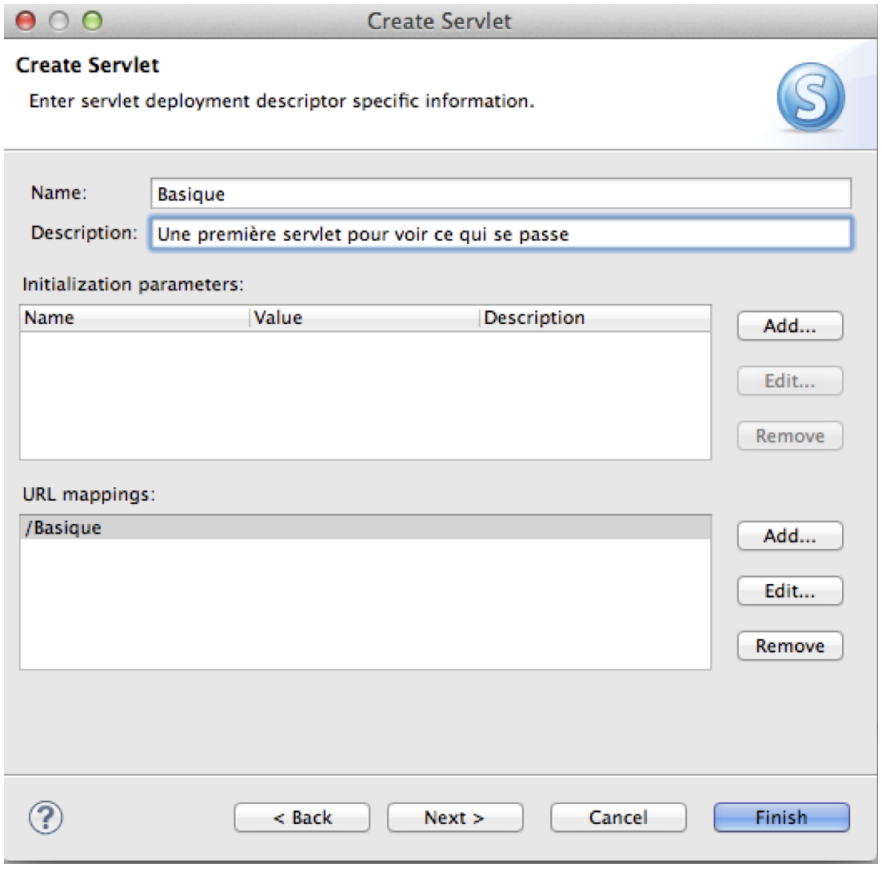

FIG. 18 – Seconde fenêtre de création d'une *servlet*

(suite de la page précédente)

```
15 public class Basique extends HttpServlet {
16     private static final long serialVersionUID = 1L;
17
18  /**
19   * @see HttpServlet#HttpServlet()
20   */
21  public Basique() {
22     super();
23     // TODO Auto-generated constructor stub
24  }
25
26     /**
27      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
28      */
29     protected void doGet(HttpServletRequest request, HttpServletResponse response)
30        throws ServletException, IOException {
31             // TODO Auto-generated method stub
32     }
33
34     /**
35      * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
36      */
37     protected void doPost(HttpServletRequest request, HttpServletResponse response)
38       throws ServletException, IOException {
39             // TODO Auto-generated method stub
40     }
41 }
```

Donc, les lignes 2 à 8 importent les *packages* Java correspondant aux super-classes et interfaces des servlets. En particulier, la classe `HttpServlet` doit être la super-classe de nos servlets spécifiques :

```
public class Basique extends HttpServlet
```

En l'occurrence, `Basique` hérite du comportement général (dialogue via HTTP) défini par `HtppServlet` et doit implanter quelques méthodes abstraites sur lesquelles nous revenons plus loin.

L'annotation `WebServlet` définit l'association (*mapping*) entre une URL relative (à l'URL de l'application) et la servlet :

```
@WebServlet(description = "Une première servlet pour voir ce qui se passe",
 urlPatterns = { "/Basique" })
```

Il est possible d'associer plusieurs motifs d'URL (*url patterns*) à une servlet. Cette annotation peut également contenir des paramètres d'initialisation de la *servlet*. Nous n'allons pas détailler tout cela, au moins pour l'instant.

Important : Cette annotation n'existe que depuis la mise en place de la version 3 des servlets. Auparavant, tout devait être indiqué dans le fichier de configuration `web.xml`. Voici d'aileurs le contenu de ce fichier, pour une configuration équivalente. Les annotations évitent dans une large mesure d'avoir à éditer le fichier de configuration en parallèle au code java.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
```

(suite sur la page suivante)

(suite de la page précédente)

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
↪javaee/web-app_3_0.xsd"
      version="3.0">
          <servlet>
                  <servlet-name>Basique</servlet-name>
                  <servlet-class>tests.Basique</servlet-class>
          </servlet>

          <servlet-mapping>
                  <servlet-name>Basique</servlet-name>
                  <url-pattern>/Basique</url-pattern>
          </servlet-mapping>
     </web-app>
```

Continuons l'exploration du code. Deux méthodes ont été automatiquement produites, avec un contenu vide : `doGet` et `doPost`. Comme leur nom le suggère, ces méthodes correspondent respectivement au code à exécuter en réponse à une requête HTTP `GET` ou `POST`. Nous allons compléter la méthode `doGet` de la manière suivante.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
        response.setContentType("text/html");
        response.setCharacterEncoding( "UTF-8" );
        PrintWriter out = response.getWriter();
        out.println("Ma première <i>servlet</i> s'exécute");
}
```

Notons tout d'abord que cette méthode reçoit deux paramètres représentant respectivement la requête HTTP reçues (`request`) et la réponse HTTP à transmettre (`response`). Le contenu des messages HTTP a été automatiquement analysé, structuré, et placé dans l'objet `request`. Réciproquement, le message HTTP à envoyer en réponse au programme client sera automatiquement mis en forme en fonction des informations que nous plaçons dans l'objet `response`. En d'autres termes, on n'a plus à se soucier du protocole HTTP, ou même du caractère distribué de l'application que nous réalisons.

Il nous reste à tester cette servlet. Pour cela, redémarrez Tomcat pour qu'il enregistre son existence (dans l'onglet *Servers*, utilisez la flèche verte). L'URL http://localhost:8080/nsy135/Basique devrait afficher le résultat de la figure *Exécution de notre première servlet.*.

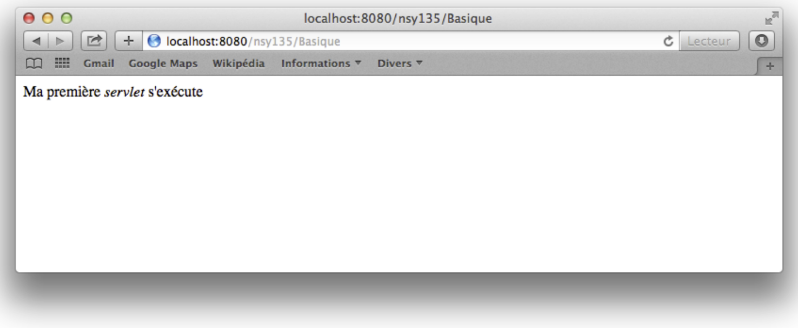

FIG. 19 – Exécution de notre première servlet.

3.4 Résumé : savoir et retenir

À ce stade, vous diposez déjà des bases pour réaliser une application Web : vous savez organiser une application, la tester avec Eclipse, Tomcat et un navigateur, et écrire du code Java traitant les requêtes et réponses HTTP. Avec beaucoup d'efforts, de tatonnements et d'erreurs, vous pourriez réaliser des fonctionnalités sophistiquées impliquant des accès à une base de données, la mise en forme de documents HTML complexes, etc.

Vous pourriez le faire, mais vous ne le ferez pas, car ce serait infiniment pénible et long. Comme le montrent les quelques commandes qui précèdent, l'écriture d'applications Web avec Java est maintenant très balisée, et des outils comme Eclipse apportent un gain de temps considérable pour appliquer automatiquement les bonnes pratiques mises au point pendant les dernières décennies. Imaginez juste à quel point il serait épouvantable de produire un long document HTML avec des séries d'appels `ou.println`…

Sans chercher à refaire l'histoire pas à pas, nous allons directement mettre en œuvre les meilleures pratiques et leurs techniques associées, en commençant par le motif de conception MVC pour notre application Web. La bonne nouvelle est qu'il s'agit également de la manière la plus simple et la plus économique de produire le code de notre application.

Pour la suite, vous devez avoir intégré les choses suivantes :

— Eclipse est notre tour de contrôle pour *tout* faire dans la phase de développement ; vous devriez avoir configuré Eclipse correctement, et être familier des commandes de création de fichiers divers, d'interaction avec Tomcat, de test de votre application ;
— Une application JEE obéit à une certaine organisation, est composée de servlets, et s'exécute dans un conteneur comme Tomcat ;
— La programmation Java, sans recourir à un environnement et à des outils spécialisés dans la production d'application Web, serait tout simplement un cauchemard (que certains ont vécu dans le passé, mais qu'il serait maintenant inexcusable de revivre).

CHAPITRE 4

Modèle-Vue-Contrôleur (MVC)

Avant de nous lancer dans un développement, clarifions ce motif de conception (*pattern design*) dit *Modèle-Vue-Contrôleur* (MVC). Il est maintenant très répandu et accepté sans contestation comme un de ceux menant, notamment pour la réalisation de sites Web dynamiques, à une organisation satisfaisant le but recherché d'une organisation rigoureuse et logique du code.

Un des objectifs est la séparation des différentes *couches* constituant une application interactive, de manière à simplifier la gestion de chacune, par exemple en permettant la remise en cause indépendante de l'un des composants de l'architecture globale. Il devrait par exemple toujours être possible de revoir complètement la *présentation* d'un site sans toucher au code applicatif (ou *métier*), et, réciproquement, le code *métier* devrait être réalisé avec le minimum de présupposés sur la présentation. La question de l'évolutivité est elle aussi essentielle. Un logiciel doit être modifiable facilement et sans dégradation des fonctions existantes (régression). Enfin, dans tous les cas, l'organisation du code doit être suffisamment claire pour qu'il soit possible de retrouver très rapidement la partie de l'application à modifier, sans devoir ouvrir des dizaines de fichiers. C'est notamment très utile sur de gros projets impliquant plusieurs personnes : le partage des mêmes principes de développement et de structuration représente à terme un gain de temps considérable.

Ce chapitre présente le MVC dans un contexte pratique, en allant aussi vite que possible à une illustration (simplifiée à l'essentiel) de son application. Pour des raisons de clarté et d'introduction à des concepts parfois complexes, le MVC présenté ici vise à la simplicité et à la légèreté plus qu'à la richesse. L'apprentissage de solutions plus complètes destinées à des développements à grande échelle devrait en être facilité.

4.1 S1 : Principe général

Supports complémentaires :

- Diapos pour la session « S1 : Principe général »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3585f56d5y921cu/

Le MVC est un motif de conception (*design pattern*) qui propose une solution générale au problème de la structuration d'une application. Le MVC définit des règles qui déterminent dans quelle couche de l'architecture, et dans quelle classe (orientée-objet) de cette couche, doit être intégrée une fonctionnalité spécifique. Une application conforme à ces règles est plus facile à comprendre, à gérer et à modifier. Ces règles sont issues d'un processus d'expérimentation et de mise au point de bonnes pratiques qui a aboutit à une architecture standard.

Cette petite introduction au MVC est volontairement courte afin de dire l'essentiel sans vous surcharger avec toutes les subtilités conceptuelles qui accompagnent le sujet. Pour en savoir plus, vous pouvez partir du site de Wikipedia : le modèle MVC (Modèle-Vue-Contrôleur).

4.1.1 Vue d'ensemble

L'objectif global du MVC est de séparer les aspects *traitement*, *données* et *présentation*, et de définir les interactions entre ces trois aspects. En simplifiant, les données sont gérées par le *modèle*, la présentation par la *vue*, les traitements par des *actions* et l'ensemble est coordonné par les *contrôleurs*. La figure *Architecture MVC* donne un aperçu de l'architecture obtenue, en nous plaçant d'emblée dans le cadre spécifique d'une application Web.

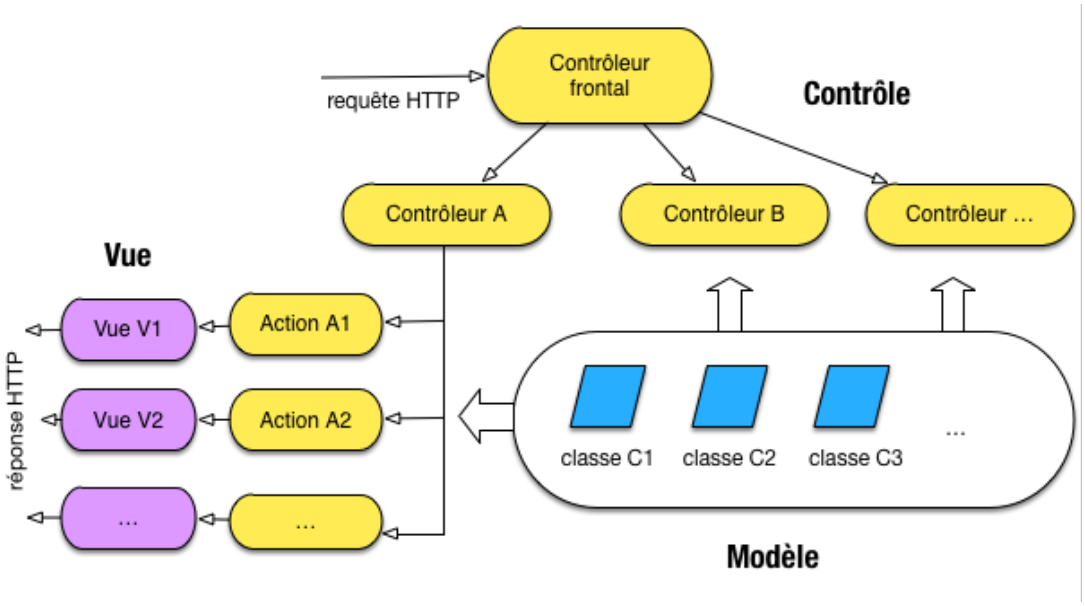

FIG. 1 – Architecture MVC

La figure montre une application constituée de plusieurs contrôleurs, chaque contrôleur étant lui-même constitué d'un ensemble d'actions. La première caractéristique de cette organisation est donc de structurer hiérarchiquement une application. Dans les cas simples, un seul contrôleur suffit, contenant l'ensemble des actions qui constituent l'application Web.

Chaque requête HTTP est analysée par le *framework* qui détermine alors quel sont le contrôleur et l'action concernés. Il existe un *contrôleur frontal* (intégré au *framework* et donc transparent pour le programmeur), chargé de recevoir les requêtes HTTP, qui exécute l'action en lui passant les paramètres HTTP. Il se base notamment sur les *mappings* qui associent des URLs à des *servlets*, comme nous l'avons vu dans le chapitre *Environnement Java*.

Note : Souvenez-vous du concept *d'inversion de contrôle* qui définit les *frameworks*. Les contrôleurs (avec leurs actions) sont les composants injectés par l'application, et c'est le *framework* qui décide du composant à appliquer en fonction du contexte. C'est lui aussi qui se charge de transmettre la vue au client web.

Au niveau du déroulement d'une action, les deux autres composants, la *vue* et le *modèle*, entrent en jeu. Dans le schéma de la figure *Architecture MVC*, l'action A_1 s'adresse au modèle pour récupérer des données et éventuellement déclencher des traitements spécifiques à ces données. L'action passe ensuite les informations à afficher à la vue qui se charge de créer la présentation. Concrètement, cette présentation est le plus souvent dans notre cas le document HTML qui constitue la réponse HTTP.

Il s'agit d'un schéma général qui peut se raffiner de plusieurs manières, et donne lieu à plusieurs variantes, notamment sur les rôles respectifs des composants. Sans entrer dans des discussions qui dépassent le cadre de ce document, voici quelques détails sur le modèle, la vue et le contrôleur.

4.1.2 Le modèle

Le modèle implante les fonctionnalités de l'application, indépendamment des aspects interactifs. Le modèle est également responsable de la préservation de *l'état* d'une application entre deux requêtes HTTP, ainsi que des fonctionnalités qui s'appliquent à cet état. Toute donnée persistante doit être gérée par la couche *modèle*, mais des objets métiers non persistant implantant une fonctionnalité particulière (un calcul, un service) sont également dans cette couche. Le modèle gère les données de session (le panier dans un site de commerce électronique par exemple) ou les informations contenues dans la base de données (le catalogue des produits en vente, pour rester dans le même exemple). Cela comprend également les règles, contraintes et traitements qui s'appliquent à ces données, souvent désignées collectivement par l'expression *logique métier de l'application*.

Important : Les composants de la couche modèle doivent pouvoir être utilisés dans des contextes applicatifs différents : application web, *back office*, application bureau, etc. Ils doivent donc impérativement être *totalement indépendants* de la gestion des interactions avec les utililsateurs, ou d'un environnement d'excution particulier.

La *persistance* est définie comme la propriété de l'état d'un objet à être préservé au-delà de la durée d'exécution de l'application en général, et d'une action en particulier. Concrètement, la persistance est obtenue le plus souvent par stockage dans une base de données. La persistance est aspect d'un objet métier ; elle ne doit pas (ou très peu) impacter la logique applicative implantée par cet objet. Pour le dire autrement, on devrait pouvoir ajouter/modifier/supprimer le comportement de persistance d'un objet indépendamment des services qu'il fournit. Nous reviendrons bien sûr longuement sur ce critère.

4.1.3 La vue

La vue est responsable de l'interface, ce qui recouvre essentiellement dans notre cas les fragments HTML qui sont assemblés pour constituer les pages du site. La vue est également responsable de la mise en forme des données (pour formater une date par exemple) et doit d'ailleurs se limiter à cette tâche. Il faut prendre garde à éviter d'y introduire des traitements complexes qui relève de la *logique métier*, car ces traitements ne seraient alors pas réutilisables dans une autre contexte. En principe la vue ne devrait pas accéder au modèle et obtenir ses données uniquement de l'action (mais il s'agit d'une variante possible du MVC).

La vue est souvent implantée par un moteur de *templates*, dont les caractéristiques, avantages et inconvénients donnent lieu à de nombreux débats. Une des difficultés est qu'il est souvent nécessaire de prendre des décisions concernant l'affichage dans la vue elle-même (par exemple, si telle donnée a telle valeur, alors on affiche tel fragment). Ces décisions s'implantent avec des instructions impératives comme les tests ou les boucles, ce qui soulève deux problèmes majeurs :

- la vue devient lourde et ressemble à de la programmation, ce qu'on voudrait éviter ;
- et surtout si on permet de coder dans la vue, qu'est-ce qui empêche un programmeur d'y mettre de la logique métier ? On risque de perdre la clarté du modèle MVC.

Dans le cas de Java, les outils pour gérer les vues ont beaucoup évolué, depuis les *Java Server Pages* (JSP) jusqu'à la *Java Standard Tag Library* (JSTL) que nous présenterons plus loin. Les moteurs de *template*, dans l'environnement Java, s'appuient sur ces outils.

4.1.4 Contrôleurs et actions

Le rôle des contrôleurs est de récupérer les données utilisateurs, de les filtrer et de les contrôler, de déclencher le traitement approprié (via le modèle), et finalement de déléguer la production du document de sortie à la vue. Comme nous l'avons indiqué précédemment, l'utilisation de contrôleurs a également pour effet de donner une structure hiérarchique à l'application, ce qui facilite la compréhension du code et l'accès rapide aux parties à modifier. Indirectement, la structuration *logique* d'une application MVC en contrôleurs et actions induit donc une organisation normalisée.

Les contrôleurs, comme leur nom l'indique, ont pour rôle essentiel de coordonner les séquences d'actions/réactions d'une application web. Ils reçoivent des requêtes, déclenchent une logique à base d'objets métiers, et choisissent la vue qui constituent la réponse à la requête.

Important : Attention à ne pas placer de logique applicative dans un contrôleur, car (comme dans le cas des vues) le code *n'est pas réutilisable dans un autre contexte*. Il est impératif de placer cette logique, systématiquement, dans la couche métier (le modèle).

Tout cela est un peu abstrait sans doute pour l'instant, mais l'essentiel est dit et nous vous invitons à revenir aux quelques paragraphes qui précèdent chaque fois que c'est nécessaire pour bien vous imprégner des principes fondamentaux du MVC.

4.2 S2. En pratique

Supports complémentaires :

- Diapos pour la session « S2 : En pratique »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594b8a7494zi2u/

Allons-y pour une première découverte de la structuration d'une application selon les principes du MVC. Nous allons faire très simple et construire un petit convertisseur de température de l'unité Celsius vers l'unité Fahrenheit. Nous utiliserons un seul contrôleur, un seul objet métier, et deux vues :

- la première affiche un formulaire permettant de saisir une valeur en Celsius et de déclencher une action ;
- la seconde vue affiche le résultat de la conversion de la valeur entrée en Fahrenheit.

Le but est de faire tourner ce premier exemple, et de l'approfondir ensuite.

Note : L'exemple qui suit est volontairement simpliste, pour aller à l'essentiel et fournir la base d'une première discussion. De plus, certaines choses vous paraîtront magiques à première vue. Les explications suivent !

4.2.1 Notre première vue : une page JSP

Pour construire nos composants « vues » nous allons nous baser sur les *Java Server Pages* ou JSP. Il s'agit d'une technologie maintenant assez ancienne (à l'échelle de l'informatique en tout cas) et qui a subi de profondes évolutions depuis sa première apparition. Essentiellement, l'idée et de créer des pages HTML dans lesquelles on peut insérer du code java de manière à obtenir des fragments produits dynamiquement par du code. À l'usage cette idée s'est révélée utile mais peu élégante, avec un mélange de différents langages conduisant à beaucoup de lourdeurs et de répétitivité.

Les JSP ont progressivement mué pour éliminer ces inconvénients, selon deux directions principales :

- les actions les plus fréquentes sont exprimées par des balises (tags) spéciaux qui s'intègrent beaucoup mieux au code HTML,
- une forme de programmation *par convention* est apparue qui revient à considérer implicitement que le programmeur suit toujours des règles standard de codage (notamment pour les noms), la connaissance de ces règles menant à de très grandes simplifications.

Les deux exemples que nous donnons dans cette première réalisation constituent une illustration basique de ces deux tendances. Il vaut la peine de souligner que l'étape ultime de cette évolution est la création de langages entièrement dédiés à l'intégration d'instructions programmatiques dans du HTML, ces langages faisant partie de fameux *frameworks* que nous allons explorer ensuite. Et il est également bon de se souvenir que quel que soit l'aspect convivial de ces langages, ils sont ultimement transformés en une page JSP, qui elle-même n'est rien d'autre qu'une *servlet* déguisée. Soyez donc attentifs à ce qui suit, car c'est vraiment le socle sur lequel se sont construits tous les outils de plus haut niveau utilisés couramment.

Voici donc notre page d'accueil pour ce convertisseur, montrant un formulaire. C'est du HTML *presque* pur, la première différence (ici) état une instruction `<%@` donnant des directives sur la production du message HTTP. Cette directive en l'occurrence indique que l'encodage du message est en UTF-8 et que le contenu (`ContentType`) est du HTML.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>Convertisseur de température</title>
   </head>
   <body>
      Vous pouvez convertir une température exprimée en
      <b>Celsius</b> en une valeur exprimée en
      <b>Fahrenheit</b>.

      <hr />
      <form method="post"
      action="${pageContext.request.contextPath}/convertisseur">

      Valeur en Celsius: <input type="text" size="20" name="celsius" /> <br />

      <input type="submit" value="Convertir" />

      </form>
      <hr />
   </body>
</html>
```

Pour ce qui concerne le HTML proprement dit, notez que la méthode HTTP choisie quand on soumet le formulaire est `post`. Et notez également qu'un paramètre HTTP nommé `celsius` est alors transmis à l'URL :

```
${pageContext.request.contextPath}/convertisseur
```

Cette expression `${pageContext.request.contextPath}` représente l'URL contextuelle de notre servlet `convertisseur` (que nous allons bientôt créer), autrement dit le domaine et chemin menant à notre application, dans le contexte particulier où elle est déployée. Dans notre environnement de développement, cette expression prend la valeur `http://localhost:8080/nsy135`, mais lors d'un déploiement éventuel, l'URL contextuelle sera probablement quelque chose de la forme `http://mondomaine.com/convertisseur`. Il serait bien dommage (en fait, inacceptable) de coder « en dur » ce genre d'information, avec la sanction d'avoir à changer le code à chaque changement d'environnement. Les JSP nous permettent d'éviter ce genre de désagrément en utilisant des expressions dynamiques évaluées au moment de l'exécution.

Pour créer cette JSP, dans Eclipse, utilisez l'option `New` et le choix `JSP page`. Par défaut Eclipse place les JSP dans la racine du site, à savoir `src/main/webapp`. Vous pouvez garder ce choix pour l'instant. Copiez/collez le code ci-dessus dans une page que vous pouvez appeler (par exemple) `convinput.jsp`.

Ce n'est pas plus compliqué que cela, et de fait cette page est directement accessible à l'adresse :

```
http://localhost:8080/nsy135/convinput.jsp
```

Vous devriez obtenir un affichage semblable à celui de la figure *Affichage du formulaire du convertisseur.*.

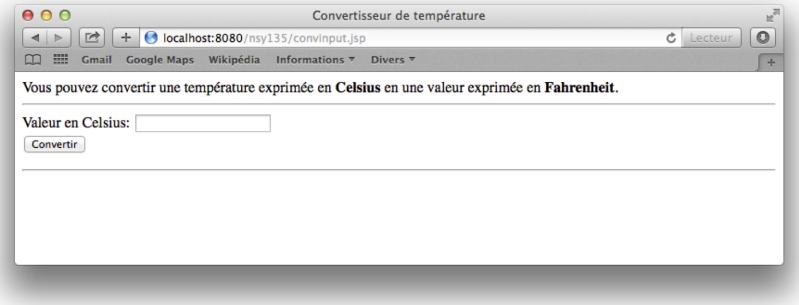

FIG. 2 – Affichage du formulaire du convertisseur.

Rien de particulier à ce stade, l'affichage est celui d'un formulaire HTML.

Note : Ceux parmi vous qui ont déjà intégré fortement les principes du modèle MVC peuvent froncer les sourcils en se disant : « est-il normal d'accéder à une vue sans passer par un contrôleur ? » Et ils ont raison, mais patience, tout sera dévoilé en temps et en heure.

4.2.2 Le contrôleur : une *servlet*

Notre contrôleur est une servlet, que nous créons selon la procédure habituelle avec notre Eclipse (faut-il vous la ré-expliquer ?). Appelons-la `Convertisseur.java` et associons-la à l'URL `/convertisseur`. Si vous ne comprenez pas la phrase qui précède, relisez les explications pour la création de *servlet* que nous avons détaillées à la fin du chapitre *Environnement Java*.

Nous allons définir les deux méthodes `doGet` et `doPost` de manière à ce que d'une part la première affiche notre formulaire précédemment créé, et que d'autre part la seconde reçoive le paramètre de ce formulaire et effectue la conversion de la température saisie. Il s'agit d'une implantation *extrêmement basique* (et insatisfaisante) de la notion d'*action* présentée au début de ce chapitre, mais elle nous suffira pour l'instant.

Voici donc la méthode `doGet`, déclenchée quand on envoie un message `GET` à l'URL `/convertisseur`.

```
/**
 * Méthode Get: on affiche le formulaire
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
        String maVue = "/convinput.jsp";
        RequestDispatcher dispatcher =
                        getServletContext().getRequestDispatcher(maVue);
        dispatcher.forward(request,response);
}
```

Cet exemple est très simple et nous permet d'apprendre une nouvelle technique : la *délégation* par le contrôleur (la servlet) de tout ce qui concerne l'affichage à la *vue* (la JSP). Le code consiste donc à indiquer quelle JSP on veut

déclencher (ici c'est `convinput.jsp`, que l'on prend dans la racine du site, soit `src/main/webapp`) et à transmettre, en fin de méthode, le flot de l'exécution avec la méthode `forward` à cette JSP.

C'est donc la mise en application du principe MVC qui veut que le contrôleur reçoive les requêtes (ici, pour l'instant, on n'en fait rien), applique les fonctionnalités métiers implantés par les modèles (ici, pour l'instant, il n'y en a pas) et finalement transmette à la vue les éléments nécessaires à l'affichage.

On pourrait aller un peu plus loin en adoptant la convention suivante : la vue (JSP) porte le même nom que la servlet, et on lui transfère systématiquement l'exécution. Il n'y aurait plus le moindre code à produire ! C'est en fait le comportement par défaut appliqué par certains *frameworks*, d'où leur aspect parfois *magique* quand on les découvre.

Important : nous avons maintenant deux manières d'appeler notre page JSP : directement, ou en passant par la *servlet*. *Dans une approche MVC, seule la seconde est bonne*. Pour interdire la première, il faudrait placer la JSP dans un répertoire inaccessible par une URL. Un bon candidat pour cela est le répertoire `WEB-INF`.

Cette approche (interdiction d'exécuter une vue) est en fait forcée par les *frameworks* donc nous n'aurons plus à nous poser ce genre de question quand nous y aurons recours.

4.2.3 Le modèle : une classe `Temperature`

Notre modèle est on ne peut plus simple : il consiste en une classe représentant une température, et capable de fournir la valeur de cette température en degrés Celsius ou Fahrenheit.

La classe `Temperature` donnée ci-dessous va suffire à nos besoins.

```
package modeles;

    /**
     * Une classe permettant d'obtenir une température en Celsius ou␣
↪Fahrenheit.
     *
     */
    public class Temperature {

      /**
       * La valeur, exprimée en degrés Celsius
       */
      private double celsius;

      /**
       * Le constructeur, prend des Celsius en paramètres
       */
      public Temperature(double valeurCelsius)
      {
            celsius = valeurCelsius;
      }

      /**
       * Pour obtenir la valeur en Celsius
       *
       * @return Valeur de la température en Celsius
      */
      public double getCelsius() {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                return celsius;
        }

        /**
         * Pour obtenir la valeur en Fahrenheit
         *
         * @return Valeur de la température en Fahrenheit
         */
        public double getFahrenheit() {
                return (celsius * 9/5) + 32;
      }
}
```

Remarquez que nous suivons les bonnes pratiques de base de la programmation objet, entre autres :

— aucune propriété n'est publique ;
— des méthodes `set` et `get`, au nommage normalisé, servent à lire/écrire les propriétés ;
— les commentaires sont abondants et permettront de produire une `javadoc` pour documenter le projet.

De plus, conformément aux principes du MVC, cette classe est totalement indépendante du contexte Web. Elle peut en fait être utilisée dans n'importe quelle application Java. Il serait évidemment souhaitable de l'enrichir avec quelques méthodes, par exemple pour l'initialiser à partir d'une valeur en Fahrenheit et pas en Celsius, pour ajouter des températures, etc. Je vous laisse ce plaisir.

Pour créer cette classe, vous devriez pouvoir trouver la démarche tout seul maintenant. Utilisez à nouveau l'option `New` du menu contextuel du projet, et prenez le choix `Class` dans le menu déroulant. Nous avons placé notre classe dans le *package* `modeles`, le fichier sera donc sauvegardé dans `src/modeles`.

4.2.4 La seconde action : conversion

Équipés de ce modèle, nous sommes prêts à recevoir une température (en Celsius) entrée par l'utilisateur, à la convertir et à l'afficher. La conversion correspond à une seconde action que nous implantons dans la méthode `doPost` de notre servlet. La voici :

```
1   /**
2    * Méthode Post: on affiche la conversion
3    */
4   protected void doPost(HttpServletRequest request, HttpServletResponse response)
5                   throws ServletException, IOException {
6           // On devrait récuperer la valeur saisie par l'utilisateur
7           String valCelsius = request.getParameter("celsius");
8
9           if (valCelsius.isEmpty()) {
10                  // Pas de valeur: il faudrait afficher un message, etc.
11                  valCelsius = "20";
12          }
13
14          // Action: appliquons le convertisseur. Espérons que valCelsius représente
15          // bien un nombre, sinon...
16          Temperature temp = new Temperature(Double.valueOf(valCelsius));
17          // Enregistrons l'objet dans la requête
18          request.setAttribute("temperature", temp);
19
20          // Transfert à la vue
```

(suite sur la page suivante)

(suite de la page précédente)

```
            String maVue = "/convoutput.jsp";
            RequestDispatcher dispatcher =
                            getServletContext().getRequestDispatcher(maVue);
            dispatcher.forward(request,response);
    }
```

Décomposons ce code. La première étape consiste à récupérer le paramètre saisi par l'utilisateur. Souvenez-vous de notre formulaire et de son champ `celsius`. Ce champ, quand l'utilisateur valide le formulaire, devient un paramètre `celsius` de la *requête* HTTP. C'est donc logiquement par l'objet `request` que nous pouvons récupérer la valeur de ce paramètre :

```
String valCelsius = request.getParameter("celsius");
```

Deux remarques s'imposent : (i) on récupère des chaînes de caractères, car les paramètres HTTP ne sont pas typés, (ii) rien ne garantit que l'utilisateur a vraiment saisi une valeur avant d'appuyer sur le bouton, et encore moins que cette valeur est convertible en numérique. Il y aurait toute une phase de contrôle à intégrer dans le code, que nous nous épargnons ici.

Passons donc directement à la suite : on instancie un objet de notre classe `Temperature` avec la valeur du paramètre :

```
Temperature temp = new Temperature(Double.valueOf(valCelsius));
```

C'est ici que le logique « métier » de l'application intervient sous la forme d'un objet du modèle. Pas question de mettre *directement* dans le code du contrôleur la conversion de Celsius en Fahrenheit, car elle ne serait pas *réutilisable* dans un autre contexte. Le contrôleur se charge des interactions, de la coordination des composants, et c'est tout.

Nous en arrivons à une partie importante et pas forcément évidente à comprendre dans un premier temps : comment passer à la *vue*, qui va se charger de l'affichage, l'information à afficher, à savoir la température ? Le principe est que la servlet et la vue *partagent* un certain nombre d'objets formant le *contexte* de l'exécution d'une requête côté serveur. Parmi ces objets, on trouve `request` et quelques autres (`session` par exemple) que nous découvrirons ensuite.

En conséquence : toute information disponible dans `request` sera accessible par la vue JSP. Par ailleurs, l'objet `request` a la capacité d'enregistrer des *attributs* dont la valeur peut être n'importe quel objet java. Ici, nous créons donc un nouvel attribut de `request`, nommé `temperature`, et nous lui affectons notre objet-modèle :

```
request.setAttribute("temperature", temp);
```

C'est le mécanisme général : la servlet place dans `request` les informations dont la vue a besoin, et déclenche ensuite la vue (ici, `convoutput.jsp`) avec la méthode `forward` que nous avons déjà rencontrée. Il reste à voir comment la vue gère cet affichage.

4.2.5 L'autre vue JSP

La voici.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Résultat de la conversion</title>
</head>
```

(suite sur la page suivante)

(suite de la page précédente)

```
 <body>

      <p>Vous avez demandé la conversion en Fahrenheit de la valeur en
             Celsius ${requestScope.temperature.celsius}</p>
      <p>
             <b>Et le résultat est: ${requestScope.temperature.fahrenheit}
                    degrés Fahrenheit </b>!
      </p>
 </body>
</html>
```

Toute la nouveauté (et la « magie » de la chose) tient à ces expressions intégrées au HTML :

```
${requestScope.temperature.celsius}
```

Elles offrent une syntaxe extrêmement simple pour accéder aux objets placés dans le contexte de la vue par la servlet. Le seul de ces objets que nous avons utilisés pour l'instant est la requête, désignée ici par `requestScope`. L'expression `requestScope.temperature` fait donc référence à *l'attribut* `temperature` que nous avons ajouté à la requête avant l'appel au JSP.

Et finalement que signifie `temperature.celsius` ? C'est ici qu'un peu de magie basée sur la convention de codage intervient : le contexte d'exécution « devine » que `temperature.celsius` est une valeur obtenue par la méthode `getCelsius()` appliquée à l'objet `temperature`. La convention de nommage (celle dite des *java beans*, à base de *getters* et *setters*) permet cette substitution.

Nous avons fini ce premier tour, minimaliste mais complet, d'une application MVC écrite avec JEE. Créez cette page JSP avec Eclipse, ajoutez à la servlet les deux méthodes `doGet` et `doPost` vues ci-dessus, et exécutez votre application qui devrait afficher un résultat de conversion semblable à celui de la figure *Affichage de la seconde JSP*.

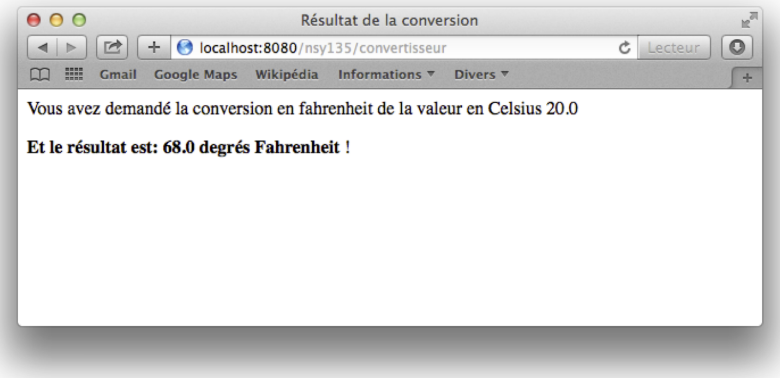

FIG. 3 – Affichage de la seconde JSP

Il est maintenant probablement profitable pour vous de faire quelques modifications au code, de bien réfléchir aux mécanismes d'échange, au rôle des différents composants, etc. Vous êtes donc invités à effectuer l'exercice suivant.

Exercice : compléter notre convertisseur

Il s'agit de compléter naturellement les classes, controleurs et JSP donnés en exemple précédemment pour permettre la saisie d'une valeur en Celsius OU ou Fahrenheit. L'application doit effectuer la conversion dans l'autre unité et produire

l'affichage approprié.

Passez ensuite à la dernière session qui va récapituler ce que nous avons appris et en tirer un bilan.

4.3 S3 : un embryon MVC

Supports complémentaires :

- Diapos pour la session « S3 : un embryon MVC »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594b9567r24pby/

Ce chapitre avait pour objectif une introduction la plus concise possible aux principes du MVC. Récapitulons les acquis, les manques, et ce qui reste à faire pour aller jusqu'au bout.

4.3.1 Les principes

Si on reprend le modèle tel qu'il est décrit dans la section *S1 : Principe général* pour le comparer à notre implantation, un certain nombre de principes ont bien été respectés :

- le modèle, sous forme de classes *Plain Old Java*, est bien indépendant du contexte (une application Web) et se charge de l'implantation de la logique « métier » ; les classes du modèle pourraient très bien être développées par un programmeur totalement ignorant de la programmation Web ;
- le contrôleur est implanté par des servlets qui font parfaitement l'affaire car elles sont justement conçus pour être au centre des échanges HTTP gérés par le serveur ; nous avons fait attention à ne pas introduire le moindre texte « en dur », la moindre balise HTML : le contrôleur, c'est de la coordination de composants, point.
- enfin, parlons des vues ; c'est d'une certaine manière le point le plus délicat car rien dans le langage Java de base ne correspond au besoin d'intégrer des éléments dynamiques dans des fragments HTML ; nous avons intégré directement les *tag libraries* dans les JSP pour obtenir une syntaxe moderne, assez propre, qui n'inclut aucune programmation java.

Tout cela est donc relativement satisfaisant, mais il est bien entendu nécessaire d'aller plus loin. Voici une liste de ce que nous allons découvrir dans la suite pour enrichir cette première approche.

4.3.2 Contrôleurs, actions, sessions

Rappelez-vous, dans la section *S1 : Principe général*, nous avions indiqué qu'un contrôleur se décomposait en *actions*, ce qui permet de donner une structure hiérarchique à notre application MVC. Un contrôleur dans cette optique se charge d'une partie bien identifiée des fonctionnalités (par exemple la gestion des utilisateurs), et est lui-même constitué d'action correspondant aux tâches élémentaires, par exemple le formulaire d'identification, le formulaire de création d'un compte utilisateur, l'édition d'un compte, etc.

Les servlets ne sont pas nativement équipées pour une décomposition en actions, et cela constitue une sévère limite pour les considérer comme des candidats totalement appropriés à l'implantation des contrôleurs. Nous avons contourné cette limitation dans notre application basique en utilisant `doGet` et `doPost` pour nos deux actions, mais cela ne se généralise évidemment pas à un nombre quelconque d'actions.

Par ailleurs, nous n'avons que superficiellement abordé la gestion des échanges HTTP : paramétrage de la requête, de la réponse, et surtout gestion des sessions.

4.3.3 Améliorez votre vue

Les premiers pas avec les JSTP (JSP étendues aux *tag libraries*) sont vraiment élémentaires. Nous ne savons pas traiter les éléments d'un tableau par exemple, ou effectuer un test (« j'affiche ce fragment HTML si et seulement si cette variable vaut *xxx* »). Il serait bon également de pouvoir combiner des fragments HTML, par exemple pour utiliser un *template* général pour notre site, que nous appliquerions à chaque contrôleur.

4.3.4 Modèle : et nos bases de données ?

Nous ne savons pas rendre les instances d'un modèle *persistantes* dans une base de données. La persistance est la capacité d'une information à survivre à l'interruption de l'application ou à l'arrêt de la machine. En pratique, elle est conditionnée par le stockage sur un support non volatile comme un disque magnétique ou SSD. Ce stockage est assuré par une base de données.

Notre convertisseur n'a pas besoin de persistance car la règle de conversion est immuable. Dans la majorité des applications, le recours à une base de données est indispensable et constitue un des points les plus délicats de la conception et de la réalisation. Nous allons y venir, et de manière très détaillée.

La persistance est une des propriétés des objets du modèle. Elle est optionnelle ; l'état de certains objets doit être persistant, pour d'autres ce n'est pas nécessaire. Selon le principe de séparation des tâches du MVC, les fonctions liées à la persistance doivent apparaître comme un des services fournis par les objets et utilisés par les contrôleurs. Mais en aucun cas ces derniers ne doivent avoir à se soucier des détails de la fonction de persistance. Pour les vues c'est encore plus radical : elles ne doivent rien savoir de la persistance.

4.3.5 L'arme ultime : un *framework*

Comment répondre aux besoins mentionnés ci-dessus à partir des bases technologiques que nous venons d'étudier ? La réponse est : *frameworks*. Un *framework* est une boîte à outil, *construite sur des technologies standard*, qui offre aux programmeurs un moyen d'appliquer facilement les bonnes pratiques (par exemple la décomposition d'un contrôleur en actions) et même force le recours à ces pratiques, avec divers avantages déjà cités (entre autres une forme de normalisation du développement favorable au travail collaboratif).

Les *frameworks* sont construits sur des outils existant, et il est donc toujours intéressant de connaître ces derniers. C'est pourquoi il est utile de consacrer un peu de temps à étudier l'environnement JEE pour les applications Web, car tous les *frameworks* sont des couches (parfois empilées) au-dessus de cet environnement. Sans ces connaissances de base, un *framework* apparaît comme un monstre abstrait qu'il est bien difficile de maîtriser et dont la courbe d'apprentissage est longue et pénible.

4.3.6 Notre framework MVC minimal

Nous n'irons pas beaucoup plus loin dans ce cours dans l'apprentissage des frameworks MVC car nous avons presque toutes les connaissances nécessaires pour étudier comment combiner une application Web et un framework de persistance (qui, rappelons-le, est notre sujet principal). À ce stade vous avez plusieurs choix :

- vous connaissez déjà un framework MVC comme Spring (ou, d'ailleurs, un autre type de framework comme Java Server Faces) et vous l'utiliserez par la suite ;
- vous avez décidé d'apprendre un framework MVC, à vous de jouer,
- vous vous en tenez là pour l'instant (recommandé) et nous allons nous contenter pour la suite de régler quelques-uns des problèmes mentionnés ci-dessus en adoptant quelques conventions de développement.

Voici donc quelques règles que nous allons suivre à partir de maintenant pour organiser notre code.

Répertoires

Nous créons deux *packages* :
- `controleurs` contiendrait tous les contrôleurs, chacun implanté par une *servlet* ;
- `modeles` contiendra toutes les classes du modèle.

Nous créons également un répertoire `vues` dans `src/main/webapp`. Dans ce répertoire nous créerons un répertoire par contrôleur, contenant toutes les vues du contrôleur. Par convention, le nom du sous-répertoire sera celui du contrôleur, sans majuscules.

Le répertoire des vues d'un contrôleur contiendra au moins une vue, nommée `index.jsp`, qui présentera le contrôleur et ses actions.

Contrôleurs et actions

Nous passerons un paramètre HTTP `action`, en mode `GET`, au contrôleur. Ce dernier, si ce paramètre est absent ou s'il ne correspond pas à une action connue, affichera la vue `index.jsp`. Sinon, le contrôleur déclenchera l'action dont le nom est donné par le paramètre.

Voici donc le squelette d'un contrôleur codé en respectant ces règles et convention. Nous prenons l'exemple de notre convertisseur.

```
package controleurs;

    /**
    * Servlet implementation class Convertisseur
    */
    @WebServlet("/convertisseur")
    public class Convertisseur extends HttpServlet {

      private static final String  VUES="/vues/convertisseur/", DEFVUE="index.jsp";

      protected void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws ServletException, IOException {

            // On devrait récupérer l'action requise par l'utilisateur
            String action = request.getParameter("action");

            // La vue par défaut
            String maVue = VUES + DEFVUE;

            try {
                    if (action == null) {
                            // Rien à faire, la vue est celle par défaut
                    } else if (action.equals("formulaire")) {
                            // Action + vue formulaire de saisie
                            maVue = VUES + "formulaire.jsp";
                    } else if (action.equals("conversion")) {
                            maVue = VUES + "conversion.jsp";
                    } else if (action.equals(...)) {
                      // Etc.
                    }
            } catch (Exception e) {
                    maVue = VUES + "exception.jsp";
                    request.setAttribute("message", e.getMessage());
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }

            // On transmet à la vue
            RequestDispatcher dispatcher = getServletContext()
                            .getRequestDispatcher(maVue);
            dispatcher.forward(request, response);
        }
}
```

On pourrait faire mieux encore, mais il faut savoir s'arrêter (si vous avez des idées d'amélioration n'hésitez pas à les appliquer). Avec ces règles nous avons déjà gagné un organisation du code plus claire, et nous gérons la notion d'action d'une manière un peu simpliste et laborieuse (il faut passer un paramètre à chaque requête). Notez également que nous gérons les exceptions au niveau de chaque contrôleur par une page JSP spéciale.

À vous de mettre en pratique ces règles pour une premier contrôleur complet.

Exercice : le contrôleur de conversion

Implantez un contrôleur de conversion de températures qui propose, dans la page d'accueil (`index.jsp`) de convertir de Celsius en Fahrenheit ou l'inverse, donne accès aux formulaires de conversion correspondants à ces choix, et affiche le ensuite résultat. Décomposez en actions, et suivez les règles énoncées ci-dessus.

Correction

Vous trouverez dans cette archive le code correspondant à une correction possible de cet exercice.

4.4 Résumé : savoir et retenir

Vous devriez à l'issue de ce chapitre avoir acquis une compréhension générale des principes des *frameworks* MVC, par combinaison des concepts et de la mise en pratique que nous proposons. Il est important que vous soyez à l'aise à partir de maintenant pour créer des contrôleurs, des actions et des vues.

Les points suivants devraient être clairs :

- les contrôleurs sont les composants qui réagissent aux requêtes HTTP, ils ne contiennent ni code métier, ni aucun élément de présentation ;
- le modèle implante les fonctionnalités métiers ; une classe du modèle doit être totalement indépendante d'un contexte d'exécution (application web ou autre) ou de critères de présentation ;
- les vues se chargent de l'affichage, et c'est tout.

Pour ces dernières, le chapitre qui suit donnera des éléments complémentaires pour bien contrôler la présentation des données. Nos aborderons ensuite, pour le restant du cours, la gestion des accès aux bases de données pour finir de régler la liste des problèmes évoqués dans la dernière session.

CHAPITRE 5

Vues : JSP et *tag libraries*

Dans ce chapitre nous complétons l'introduction de Java EE avec quelques compléments sur les *vues* JSP/JSTL, toujours dans l'optique du développement d'une application MVC.

Important : Ne considérez surtout pas ce qui suit comme une couverture complète du sujet, le but est vraiment de connaître l'essentiel, pour en *comprendre* les mécanismes. Pour devenir un expert il faut aller plus loin (mais les bases acquises devraient rendre la tâche facile), en recourant par exemple aux tutoriels disponibles sur le Web.

5.1 S1 : JSP, en bref

Supports complémentaires :

— Diapos pour la session « S1 : JSP, en bref »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594ba27iy06mx3/

Les JSP sont une approche ancienne qui a beaucoup bénéficié de l'introduction de constructions syntaxiques plus élégantes que l'introduction de code Java, et d'un codage basé sur les conventions qui simplifie considérablement l'écriture. Les langages de *vues* proposés par les *frameworks* sont pour l'essentiel assez proches des JSP. Nous allons donc passer en revue calmement les concepts généraux à connaître, qu'il vous suffira ensuite de décrypter dans votre *framework* préféré.

5.1.1 Balises et directives

L'idée initiale des JSP était d'étendre HTML avec des « balises » spéciales interprétées comme des instructions Java, avec l'espoir d'aboutir à une syntaxe à peu près uniforme intégrant le HTML « statique » et les parties dynamiques. De nombreuses mises au point ont été nécessaires pour aboutir à un résultat satisfaisant. Quelques exemples de balises proposées initialements (et toujours valides) :

— <%-- --%> est un commentaire JSP ;
— <%! %> est une balise de déclaration d'une variable ;
— <% %> est une balise d'inclusion de code Java quelqconque. . . . ;
— <%= %> est une balise d'affichage.

Cette approche fondée sur l'inclusion vaguement déguisée de code est maintenant dépréciée et doit être évitée. Elle repose clairement sur l'idée de simplifier l'écriture d'une *servlet* en évitant de placer des `out.println` devant chaque ligne HTML, une ambition louable mais limitée, ouvrant surtout la porte à des dérives vers du code illisible et non maintenable.

Note : Le conteneur compile une page JSP sous la forme d'une *servlet* dans laquelle les éléments HTML sont transcrits en instructions `out.println()`. Il s'agit donc vraiment, au moins initialement, d'un moyen simplifié de créer des servlets produisant du HTML.

La norme JSP comprend également des *directives* qui se présentent sous la forme de balises `<%@ %>`. Toujours dans une optique de programmation, on peut par exemple importer des *packages* Java :

```
<%@ page import="java.util.Date"  %>
```

Remarquez le mot-clé `page` qui indique le type de la directive. Reprenez les JSP créées précédemment : elles contiennent une directive initiale indiquant l'encodage de la page.

Une des améliorations des JSP a consisté à les rendre *extensibles* en ajoutant des *bibliothèques de balises* (*tag libraries*) associées à des fonctionnalités externes. Ces bilbiothèques doivent être déclarées avec une directive :

```
<%@ taglib uri="NsyTagLib.tld" prefix="nsy" %>
```

Le `prefix` désigne un espace de nommage (*namespace*) qui permet d'identifier une balise comme appartenant à cette bibliothèque. Par exemple, une balise `<nsy:conv />` sera identifiée comme appartenant à la bibliothèque `NsyTagLib.tld`, et le code associé sera exécuté.

Enfin le dernier type de directive est `include` qui permet de composer des pages HTML à partir de plusieurs fragments :

```
<%@ include file="header.jsp" %>
```

Comme les balises de code, les directives sont anciennes et peuvent dans la grande majorité des cas être remplacées par une approche plus moderne et plus élégante.

5.1.2 Portée des objets

Nous avons vu qu'un objet instancié dans la servlet-contrôleur pouvait être mis à disposition de la vue en l'affectant à l'objet-requête `request`. Il existe plus généralement quatre *portées* de visibilité des objets dans votre application, `request` étant l'une d'entre elles. Un objet placé dans une portée est accessible par tous les composants de l'application (par exemple les JSP) qui ont accès à cette portée. Nous avons donc :

- la portée *page* : tout objet instancié dans une JSP est accessible dans cette même JSP ; c'est la portée la plus limitée ;
- la portée *request* : les objets placés dans cette portée restent accessibles durant toute l'exécution d'une requête HTTP par le serveur, même si l'exécution de cette requête implique plusieurs contrôleurs (par des mécanismes de `forward`) ou plusieurs JSP ;
- la portée *session*, une session étant constituée par un ensemble d'interactions client/serveur, un objet placé dans la session (par exemple le fameux panier des achats dans un site de commerce) reste accessible sur la période couverte par ces interactions ;
- enfin la portée *application* est la plus générale de toute : un objet placé dans cette portée reste accessible tant que l'application est active dans le conteneur de servlets.

La figure *Les portées des variables en JEE.* illustre les portées des variables.

Note : Le protocole HTTP ne mémorise pas les échanges entre un client et un serveur. Pour effectuer un suivi, on doit

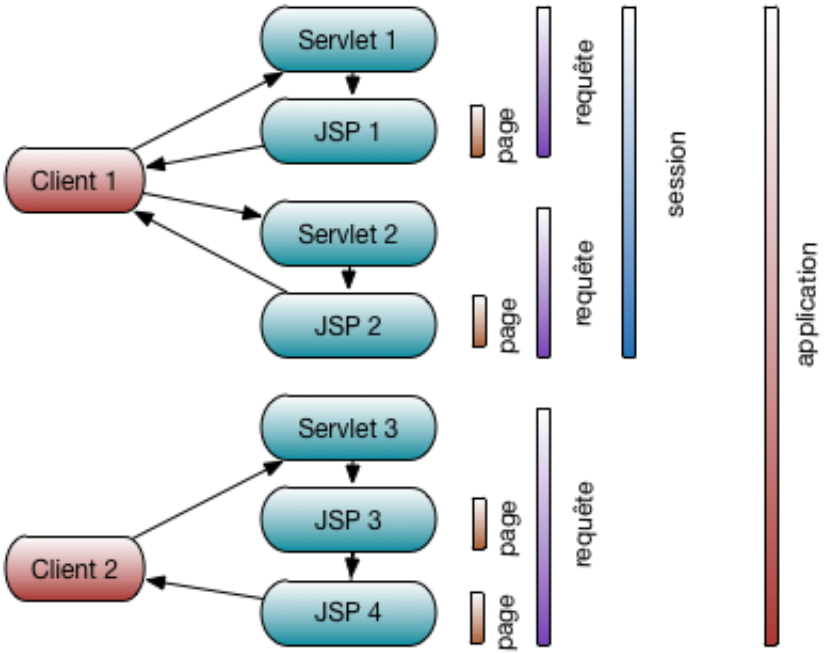

FIG. 1 – Les portées des variables en JEE.

donc simuler des *sessions*, le plus souvent en se basant sur les *cookies*.

Au sein d'une JSP, on peut accéder à une variable en indiquant la portée dans laquelle elle a été placée : `requestScope.monbean` par exemple est l'objet `monBean` placé au préalable comme attribut dans l'objet `request`.

Avertissement : Si la portée n'est pas indiquée, le conteneur va explorer les portées pour chercher un objet portant le nom indiqué. Cette pratique est potentiellement source d" ambiguité et d'erreurs difficiles à détecter : je vous conseille de toujours indiquer la portée.

5.1.3 Les expressions

Que faire d'un objet accessible dans un JSP ? Le plus courant est d'accéder à leurs propriétés (pour les afficher dans la vue) avec une syntaxe minimaliste basée sur un *langage d'expressions*. Une *expression* désigne, en programmation, toute instruction ou ensemble d'instructions renvoyant une valeur (fonction, opération arithmétique, etc.). En JSP on garde la même définition. Les expressions sont présentées sous la forme :

```
${expression}
```

Quand la JSP est exécutée, la valeur de l'expression (après évaluation par le moteur de servlet) est insérée dans la page HTML produite, en substitution de l'expression. C'est une simplification syntaxique de la balise ancienne des JSP <%= %>. Par exemple :

```
${ 12 + 8 }
```

sera remplacé par `20` à l'exécution. Intéressant ? Peut-être mais la principale utilisation des expressions est l'accès aux propriétés des objets. Pour reprendre l'exemple plus haut :

```
${requestScope.monbean.maprop}
```

a pour effet d'afficher la propriété `maprop` de l'objet `monbean` placé dans la portée `request`. Si cette propriété n'est pas publique (ce qui est toujours recommandé), le conteneur de servlet va chercher une méthode `getMaprop()` (un « *getter* » selon les conventions JavaBeans).

Plus généralement le conteneur va chercher à « deviner » quelle est l'interprétation naturelle de l'expression. Si votre objet `mamap` par exemple est une `HashMap` et contient une entrée indexée par `bill`, alors :

```
${requestScope.mamap.bill}
```

renverra la valeur de l'entrée `mamap['bill']`. Encore une fois il s'agit d'une tentative radicale de simplifier l'inclusion dans des pages HTML de valeurs représentées dans des structures Java. Les expressions sont tolérantes aux objets manquants. Par exemple :

```
${requestScope.mampa.bill}
```

ne renverra rien, (notez que `mampa` est mal écrit ?) alors que l'objet n'existe pas dans la portée et qu'une exception `null value` devrait être levée.

5.1.4 Objets implicites

L'objet `requestScope` que nous utilisons dans les exemples qui précèdent est un *objet implicite* accessible par les expressions. Il en existe d'autres, l'ensemble constituant un contexte de communication entre le contrôleur (la servlet) et la vue (la page JSP). Le moment est venu de les récapituler (Tableau *Objets implicites dans une page JSP*). Notez que la plupart sont des tables d'association (`Map`).

TABLEAU 1 – Objets implicites dans une page JSP

Nom	Type	Description
`pageContext`		Informations sur l'environnement du serveur.
`pageScope`	`Map`	Attributs de portée *page*.
`requestScope`	`Map`	Attributs de portée `request`.
`sessionScope`	`Map`	Attributs de portée `session`.
`applicationScope`	`Map`	Attributs de portée `application`.
`param`	`Map`	Les paramètres de la requête HTTP, pour les paramètres monovalués.
`paramValues`	`Map`	Les paramètres de la requête HTTP, pour les paramètres multivalués (chaque valeur d'un attribut est un tableau de `String`).
`header`	`Map`	Les paramètres de l'entête HTTP, pour les paramètres monovalués.
`headerValues`	`Map`	Les paramètres de l'entête HTTP, pour les paramètres multivalués (chaque valeur d'un attribut est un tableau de `String`).
`cookie`	`Map`	Les `cookies`.
`initParam`	`Map`	Les paramètres du fichier `web.xml`.

Prudence : Il est évidemment tout à fait déconseillé de créer un objet portant le même nom que ceux du tableau *Objets implicites dans une page JSP*.

Voici maintenant quelques exercices pour mettre en pratique les notions couvertes par cette session.

Exercice : affichage des paramètres d'une requête

Créez une page JSP `jspparam.jsp` qui accepte trois paramètres, nommés `p1`, `p2` et `p3`. Créez également un contrôleur `JspParam` associé à une URL `jspparam`. On passe donc les paramètres avec une URL de la forme `/jspparam?p1=xx&p2=yy&p3=zz`. Affichez les valeurs de ces paramètres dans la page.

5.2 S2 : Les *tag libraries* : JSTL

Supports complémentaires :

- Diapos pour la session « S2 : Les tag libraries : JSTL »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594baf4rzxa75k/

Comme indiqué au début de ce chapitre, une évolution majeure des JSP a consisté à les rendre extensibles en permettant la définition de balises fonctionnelles associées à une librairie Java. Le placement d'une de ces balises dans une page JSP correspond à l'exécution d'une fonctionnalité dans la librairie. C'est un moyen élégant de faire appel à des éléments dynamiques (programmés) dans une vue, sans avoir à intégrer du code Java.

Ces bibliothèques de balise peuvent être définies par tout programmeur, mais certaines sont destinées à résoudre des problèmes récurrents, et à ce titre elles sont fournies sous la forme d'une librairie standard, la *Java Standard Tag Library* ou JSTL. Nous allons brièvement l'explorer dans cette section, en nous basant sur la version 1.2.

Installation sous Tomcat.

Pour que les balises de la JSTL soit reconnues par Tomcat, il faut installer les deux fichiers suivants :

- jsp-api-2.1-6.0.2.jar
- jstl-1.2.jar

Récupérez ces fichiers à partir des liens ci-dessus et copiez-les sous `WEB-INF/lib`. Il peut être nécessaire de redémarrer Tomcat pour que les librairies soient prises en compte.

Une fois les librairies mises en place, on peut les utiliser dans des JSP. Voici un exemple simplissime.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <title>Ma première JSTL</title>
</head>

<body>
  <c:out value="test" />
 </body>
</html>
```

Deux choses sont à noter. Tout d'abord la librairie est incluse avec la directive `taglib` :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

L'URI (*Universal Resource Identifier*) identifie la librairie, que Tomcat va donc chercher dans son environnement. Si vous avez bien installé le `jar` comme indiqué ci-dessus, il devrait le trouver dans `WEB-INF/lib`.

Le préfixe `c` var servir, lui, à reconnaître les balises de la librairie. C'est par exemple :

```
<c:out value="test" />
```

Le préfixe agit comme un *espace de nommage* au sens de XML. Toutes les balises dont le nom est préfixé par `c:` sont considérées comme appartenant à la JSTL, et cette dernière leur associe une fonctionnalité particulière. Dans notre exemple, il n'est pas trop difficile d'imaginer que `c:out` permet d'afficher une valeur dans le document HTML produit. Nous allons explorer les principales balises de la JSTL. Je vous invite à prendre les fragments de code et à les inclure dans notre petite JSP ci-dessus pour les tester.

5.2.1 Variables et expressions

La librairie JSTL est étroitement associée aux expressions. On peut donc combiner une expression, et l'affichage de son résultat, comme dans :

```
<c:out value="${2+2}" />
```

Vous me direz : pourquoi ne pas placer directement l'expression au lieu de l'inclure dans une balise. La réponse est que JSTL effectue un pré-traitement de la chaîne de caractères à afficher, pour la rendre compatible avec XHTML. Les caractères réservés comme <, >, &, sont remplacé par des références à des *entités*, comme `<`, `&gt`, `&`, etc. Cela vous évite de le faire vous-mêmes. Si, par exemple, vous voulez afficher *littéralement* des fragments de code (comme on en trouve dans ce document), comme :

```
<langageC>if (x > 2 && x < 5) {}</langageC>
```

La bonne option est d'utiliser `c:out` :

```
<c:out value="<langageC>if (x > 2 && x < 5) {}"/>
```

Je vous invite à tester la différence.

Important : La protection des caractères interdits est particulièrement importante quand on veut afficher du texte provenant d'une source externe (formulaire, base de données) pour éviter les attaques d'injection (de code javascript par exemple) ;

La balise `c:out` accepte une valeur par défaut si l'expression échoue (par exemple parce qu'un objet n'existe pas) :

```
<c:out value="${bean}" default="Où est mon bean??" />
```

La balise `c:set` sert à *définir* ou *modifier* une variable et à lui affecter une valeur :

```
<c:set var="uneVariable" value="${17 * 23}" />
```

On peut ensuite l'afficher avec `${uneVariable}`. Attention cependant à ne pas vous lancer dans de la programmation compliquée dans la vue. Gardez l'objectif .... en vue : nous sommes là pour afficher des informations produites par le modèle et le contrôleur, pas pour développer une algorithmique liée à l'application.

5.2.2 Conditions et boucles

Supposons que vous souhaitiez afficher un message ou un fragment HTML en fonction d'une certaine condition. Par exemple, vous voulez afficher une message de bienvenue si l'utilisateur est connecté, un formulaire de connexion sinon. Vous avez besoin de tester la valeur d'une variable. Avec les JSTL, c'est la balise `<c:if>`, avec un attribut `test` contenant une expression Booléenne. Voici un exemple en supposant que vous avez défini une variable `uneVariable` comme ci-dessus :

```
<c:if test="${uneVariable > 200 }">
   Ma variable vaut plus que 200.
</c:if>
```

Pour des tests un peu plus étendus, on peut recourir à une variante du `case/switch` :

```
<c:choose>
   <c:when test="${uneVariable == 0}">Variable null</c:when>
   <c:when test="${uneVariable > 0 && uneVariable < 200}">Valeur modérée</c:when>
  ...
  <c:otherwise>Sinon...</c:otherwise>
</c:choose>
```

La possibilité d'exprimer des boucles est également indispensable dans une vue pour traiter des variables de type liste ou ensemble. Premier cas, le plus simple (mais pas le plus courant) : on dispose d'une variable de type tableau indicé. Nous avons donc besoin d'un moyen d'énumérer toutes les valeurs de l'indice. Le voici :

```
<table>
        <c:forEach var="i" begin="0" end="5" step="1">
                <tr>
                        <td><c:out value="${i}" /></td>
                        </tr>
        </c:forEach>
</table>
```

Je pense que l'exemple se passe de longues explications : il s'agit ni plus ni moins d'une mise à la sauce « balises HTML » de la boucle `for` d'un langage de programmation classique. Plus intéressant : dans le cas ou la variable correspond à une *collection* dont on ne connaît pas à priori la taille (*HashMap*, *ArrayMap*, *Collection*, typiquement), la boucle JSTL prend une forme légèrement différente. Supposons que la collection soit dans une variable `personnes`. On la parcourt de la manière suivante :

```
<c:forEach items="${personnes}" var="personne">
   <c:out value="${personne.nom}" />
</c:forEach>
```

Une variable locale à la boucle, de nom `personne`, est ici définie et successivement associée à toutes les valeurs de la collection. Dans la boucle, on se retrouve donc en situation d'avoir une variable simple, objet, à laquelle on peut appliquer les opérations déjà vues d'accès aux propriétés et d'affichage.

L'objet implicite `pageContext` est un peu différent. Ce n'est pas une `Map` mais un objet donnant accès au contexte de l'exécution d'une JSP par l'ensemble des attributs suivants :

Tableau 2 – Attributs de l'objet `pageContext`

Nom	Type	Description
`request`	`ServletRequest`	L'objet `request`
`response`	`ServletResponse`	L'objet `response`
`servletConfig`	`ServletConfig`	Configuration de la servlet
`servletContext`	`ServletContext`	Contexte de la servlet
`session`	`HttpSession`	La session

Je vous laisse recourir au Web pour des exemples d'utilisation de ces informations. Retenez qu'elles existent et sont accessibles dans la vue.

5.2.3 Liens

À priori un lien peut être directement inclus comme du HTML, sous la forme standard d'une ancre <a/>. Par exemple :

```
http://orm.bdpedia.fr
```

Dans de nombreux cas les choses sont plus compliquées, et des éléments dynamiques interviennent dans la création de l'URL du lien. Voici les principaux cas :

- le lien mène à une autre page du site : faut-il indiquer une adresse relative ou une adresse absolue, et dans ce dernier cas laquelle ?
- le lien comprend des paramètres : leur valeur doit être encodée selon la norme HTTP pour survivre au transfert réseau ;
- enfin citons une fonctionnalité sur laquelle nous revenons plus loin : la gestion des sessions.

La balise JSTL <c:url> fournit de manière transparente un support pour gérer les besoins ci-dessus. Voici une brève présentation, en commençant par la gestion du *contexte*. Souvenez-vous de notre première page JSP. Nous y avions introduit une URL de la forme :

```
${pageContext.request.contextPath}/convertisseur
```

L'URL interne est `/convertisseur`. Pour ne pas dépendre, dans l'URL complète, du contexte qui peut varier (`dev` pour le développeur, `testapp` pour la machine d'intégration, `appname`, etc.), nous avons introduit un contexte *dynamique* représenté par `${pageContext.request.contextPath}`. Nous savons maintenant qu'il s'agit d'une expression, mais nous n'avions pas encore la possibilité d'utiliser les extensions JSTL. Avec ces dernières, le lien devient simplement :

```
<c:url value="/convertisseur"/>
```

Notez bien que cette adresse est *absolue*. Elle est automatiquement étendue par le conteneur de servlet en ajoutant le chemin de l'application. Pour nous, cela donne :

```
/nsy135/convertisseur
```

Faut-il préciser qu'il est extrêmement important de ne pas coder « en dur » des valeurs dépendants de la configuration dans du code ? Le *tag* `c:url` est intégrable tel-quel dans une balise HTML `a`, ce qui conne une syntaxe un peu étrange :

```
<a href="<c:url value="/convertisseur"/>">Lien vers mon convertisseur</a>
```

Le second avantage de `c:url` est la prise en charge automatique de l'encodage des valeurs de paramètre. La syntaxe pour les paramètres est la suivante :

```
<c:url value="/convertisseur">
  <c:param name="param1" value="${personne.nom}"/>
  <c:param name="param2" value="${societe.id}"/>
</cr:url>
```

Si un paramètre contient des caractères accentués, des espaces, ou autres caractères interdits, la balise <c:param> sera encodée selon les règles décrites ici : http://www.w3schools.com/TAGs/ref_urlencode.asp.

Note : D'autres balises non décrites ici sont <c:redirect>``(envoie une directive de redirection au navigateur) et ``<c:import> pour importer des fragments HTML. Je vous laisse recourir à un tutorial JSTL *récent* si vous souhaitez comprendre ces balises. Notez également que la partie du JSTL présentée ici est la librairie `core`. Il existe quelques autres librairies d'un intérêt moins central : `xml` (traitement de documents XML), `fmt` (formatage, par exemple des dates), `fn` (chaînes de caractères).

Exercice : affichage des objets implicites

Reprenez les objets implicites du tableau *Objets implicites dans une page JSP* et affichez leur contenu. Par exemple, affichez le contenu du tableau `header`. Vous devez exprimer une boucle qui récupère un à un les élements, chacun comprenant une clé `key` et une valeur `value`.

Exercice : affichage des paramètres saisis dans un formulaire

Nous allons faire un premier essai d'association formulaire HTML/page de traitement. Voici un formulaire simple (mais notez qu'il comprend un champ `select` à choix multiple.

```
<form method="get" action='jstlparam.jsp'>
     <table>
          <tr>
                  <td>Prénom:</td>
                  <td><input type='text' name='prenom' /></td>
          </tr>
          <tr>
                  <td>Nom:</td>
                  <td><input type='text' name='nom' /></td>
          </tr>
          <tr>
                  <td>Langages que vous connaissez:</td>
                  <td><select name='langages' size='7' multiple='true
↪'>
                                  <option value='C'>C</option>
                                  <option value='C++'>C++</option>
                                  <option value='Objective-C'>
↪Objective-C</option>
                                  <option value='Java'>Java</option>
                  </select></td>
          </tr>
      </table>
      <p>
          <input type='submit' value='Afficher!' />
 </form>
```

Ecrivez la page `jstlparam.jsp` qui affiche les paramètres transmis par ce formulaire. Attention : les attributs à choix multiples sont représentés dans l'objet implicite `paramValue` (donc, ici, on aura une collection `paramValues.langage`). Par curiosité, vous pouvez également afficher tout le contenu de `paramValues`.

Exercice : intégrer un *template* à nos JSP.

Cet exercice consiste à intégrer automatiquement nos fragments HTML produits par nos JSP dans un cadre de présentation du site agréable, doté de menus, d'une jolie image, etc. La méthode pour utiliser des *templates* est indiquée sur ce site : http://mobiarch.wordpress.com/2013/01/04/page-templating-using-jsp-custom-tag/

Voici un peu d'aide. Supposons que notre graphiste décide que toutes les pages seront affichées d'après le modèle suivant :

```
<html>
 <head>
  <title>Un titre</title>
 </head>
 <body>
  <!-- Partie dynamique: le contenu de la page -->
 </body>
</html>
```

Il ne s'est pas beaucoup fatigué, mais ce n'est pas notre problème. Nous allons créer un *template* à partir de ce modèle de page. Le voici :

```
 <%@tag description="Simple Template" pageEncoding="UTF-8"%>
 <%@attribute name="body_area" fragment="true" %>

 <html>
  <head>
  <title>Un titre</title>
 </head>
 <body>
   <jsp:invoke fragment="body_area"/>
 </body>
</html>
```

On comprend intuitivement que nous définissons des fragments HTML avec un nom (ici, `body_area`) et que nous indiquons avec `<jsp:invoke>` l'emplacement dans le template où doit être inséré le fragment. Sauvegardons ce *template* dans `WEB-INF/tags/layout.tag`.

Voici maintenant une JSP qui applique ce *template*.

```
 <%@taglib prefix="t" tagdir="/WEB-INF/tags" %>
 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

 <t:layout>
  <jsp:attribute name="body_area">
   <div>
    Je suis le contenu: <c:out value="${1+3}"/>
  </div>
 </jsp:attribute>
</t:layout>
```

La première ligne indique le répertoire où se trouvent les *templates*. Ensuite on définit le contenu des fragments à insérer dans les parties dynamiques (ici, `body_area`).

Nous avons déjà au début du cours récupéré un *template* HTML pour l'intégrer à notre serveur Tomcat. Nous allons le reprendre, et identifier dans le fichier principal (`index.html`) la partie correspondant au contenu (par opposition aux menus, entêtes, pieds de page, etc.) Il faut remplacer cette partie statique par le contenu du fragment produit par notre JSP. À vous de jouer.

Correction

Vous trouverez dans cette archive le code correspondant à une correction possible de cet exercice, et des précédents de ce chapitre.

5.3 Résumé : savoir et retenir

CHAPITRE 6

Modèle : La base de données

Le moment est venu de mettre en place notre base de données pour compléter l'architecture de notre application. Tous ce qui suit utilise MySQL mais, à quelques détails près, on peut très bien le remplacer par PotsgreSQL ou n'importe quel autre système relationnel.

Comme point de départ nous prenons une base existante, *Films*, pour laquelle je fournis un jeu de données permettant de faire des premières expérimentations. La conception de cette base (notée en UML) est brièvement présentée. Nous verrons ensuite comment y accéder avec notre application Web grâce à l'interface d'accès à une base relationnelle en Java : JDBC. À l'issue de ce chapitre vous devriez pouvoir écrire un outil de soumission de requêtes SQL. Et vous devriez surtout être en mesure d'adopter une position critique sur l'utilisation d'une API d'assez bas niveau comme JDBC dans le cadre d'une application basée sur le motif MVC.

6.1 S1 : Installations et configurations

Supports complémentaires :

- Diapos pour la session « S1 : Installations et configurations »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594bbceb0wi9d7/

Nous avons besoin de quelques installations complémentaires :

- MySQL ;
- un outil d'administration et de gestion de nos bases, phpMyAdmin ;
- les librairies de connexion Java/MySQL ;
- et une première base pour commencer sans attendre.

6.1.1 MySQL et phpMyAdmin

Pour MySQL, s'il n'est pas déjà installé dans votre environnement, je vous laisse procéder sans instruction complémentaire : vous trouverez des dizaines de ressources sur le Web pour le faire. En ce qui concerne phpMyAdmin, il vous faut une suite Apache/PHP/MySQL, soit un des environnements de développement les plus courants à l'heure actuelle. Là encore je vous laisse trouver sur le Web les instructions pour vous aider. Pour faire bref, vous devriez disposer grâce à cette installation d'un serveur Web Apache en écoute sur le port 80 (il ne sera donc pas en conflit avec Tomcat), doté d'une intégration au moteur PHP et sur lequel est installée l'application phpMyAdmin.

6.1.2 Première base de données

Dernière étape préparatoire : créer une base et lui donner un contenu initial. Nous vous proposons application de gestion de films, avec notations et réservation, comme exemple. La base de données représente des films avec leurs acteurs et metteurs en scène, et des internautes qui donnent des notes à ces films. L'application permet entre autres d'effectuer des recommandations en fonction des notes données. Le schéma (conçu pour illustrer les principaux cas de figure des entités et associations) est donné en notation UML dans la figure *Schéma (partiel) de la base Films*.

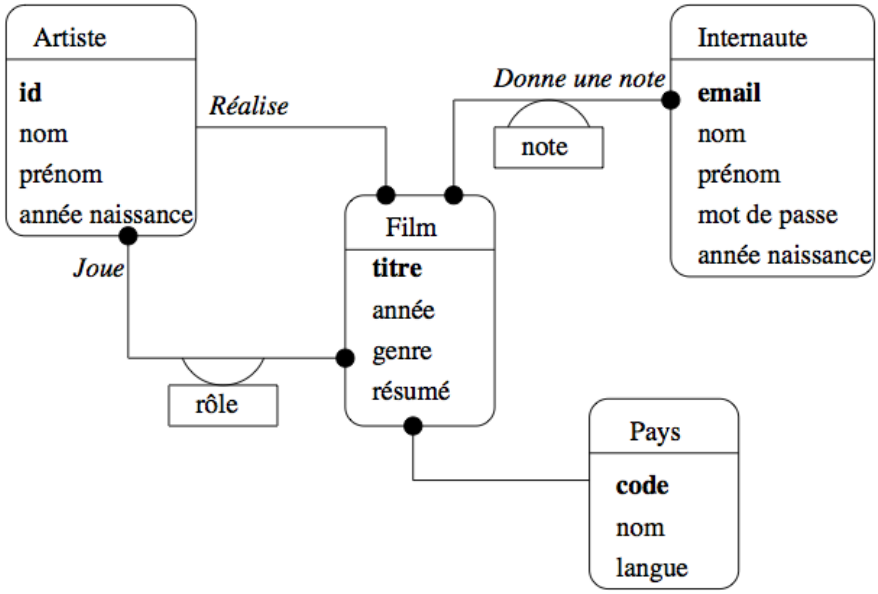

FIG. 1 – Schéma (partiel) de la base *Films*

Je suppose que vous avez les notions suffisantes pour interpréter ce schéma correctement. Quelques choix de simplification ont été effectués. On voit par exemple qu'un film n'a qu'un seul metteur en scène, qu'un acteur ne peut pas jouer deux rôles différents dans le même film. Les identifiants sont en général des séquences, à l'exception des internautes idéntifiés par leur *email* (ce qui n'est pas un bon choix mais va nous permettre d'étudier aussi cette situation).

Voici le schéma relationnel obtenu à partir de cette conception. Les clés primaires sont notées en **gras**, les clés étrangères en *italiques*.

- Film (**id**, titre, année, genre, résumé, *id_realisateur*, *code_pays*)
- Artiste (**id**, nom, prénom, année_naissance)
- Internaute (**email**, nom, prénom, mot_de_passe, année_naissance)
- Pays (**code**, nom, langue)
- Rôle (**id_film, id_acteur**, nom_rôle)
- Notation (**id_film**, **email**, note)

Pour créer la base, récupérez l'export SQL de la base Webscope ici : http://orm.bdpedia.fr/files/webscope.sql. Grâce à phpMyAdmin vous pouvez importer ce fichier SQL dans votre propre système. Voici les étapes :

— à partir de la page d'accueil de phpMyAdmin, créez une nouvelle base (appelez-la *webscope* par exemple) en indiquant bien un encodage en UTF-8 ;
— cliquez sur le nom de votre base, puis allez à l'onglet *Importer* ; vous pouvez alors charger le fichier SQL `webscope.sql` que vous avez placé sur votre disque : toutes les tables (et leur contenu) seront créées.

Il faut également créer un ou plusieurs utilisateurs. Dans la fenêtre *SQL* de phpMyAdmin, entrez la commande suivante (vous êtes libres de changer le nom et le mot de passe bien sûr).

```
GRANT ALL PRIVILEGES ON webscope.* TO orm@localhost IDENTIFIED BY 'orm';
```

Après installation, vous devriez disposer avec phpMyAdmin d'une interface d'inspection et de gestion de votre base, conforme à la copie d'écran *L'interface phpMyAdmin en action sur notre base webscope*.

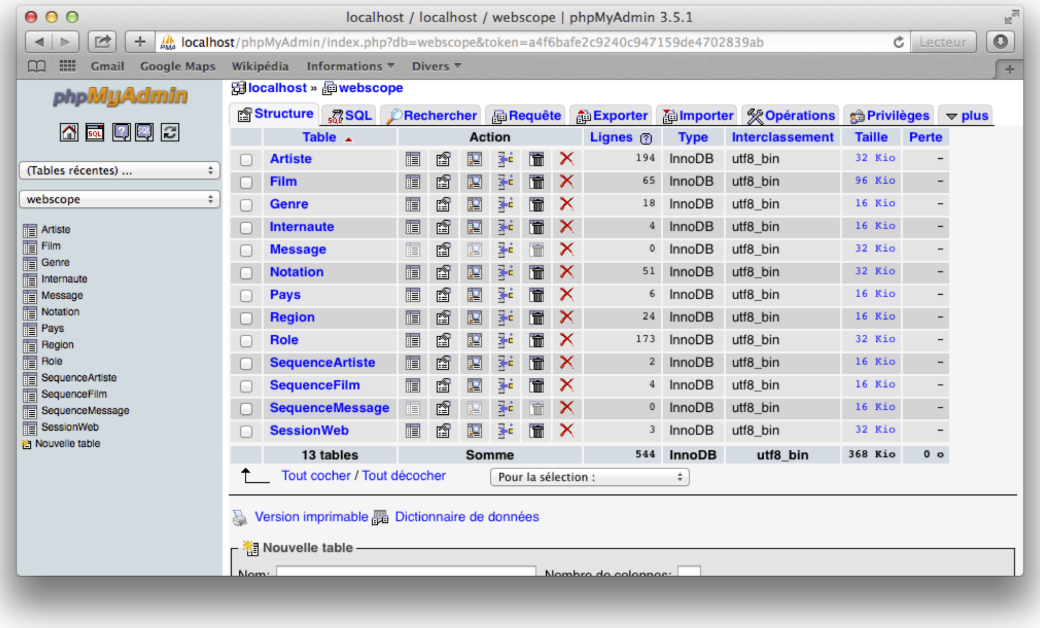

FIG. 2 – L'interface phpMyAdmin en action sur notre base *webscope*

Je vous invite à explorer avec phpMyAdmin le schéma et les tables de la base pour vous familiariser avec eux. Faites quelques requêtes SQL pour vous préparer à les intégrer dans l'application. Par exemple :

— trouver les titres des films dirigés par Hitchcock ;
— les films parus avant 2000, avec Clint Eastwood comme acteur ;
— les films qui ont obtenu une note de 5.

6.1.3 Connecteur JDBC

L'interface *Java Database Connectivity* ou JDBC est une API intégrée à la *Java Standard Edition* pour communiquer avec des bases relationnelles. Elle est censée normaliser cette communication : en principe une application s'appuyant sur JDBC peut de manière transparente passer d'une base MySQL à PostgreSQL ou à un autre système relationnel. En pratique cela suppose une certaine rigueur pour s'en tenir à la partie normalisée de SQL et éviter les extensions particulières de chaque système.

JDBC est implanté pour chaque système relationnel sous la forme d'un « pilote » (*driver*). Pour MySQL ce pilote est le *Connector/J* que l'on peut récupérer gratuitement, par exemple sur le site principal Oracle/MySQL http://dev.mysql.com/downloads/connector/j/. Je vous laisse procéder pour obtenir ce pilote, sous la forme typiquement d'un

fichier ZIP `mysql-connector-java-xx.yy.zz.zip`. En décompressant cette archive vous obtenez, entre autres, un fichier JAR `mysql-connector-java-xx.yy.zz-bin.jar`. Copiez ce fichier dans le répertoire `WEB-INF/lib` de votre application. L'API JDBC devient alors disponible.

Note : Pour des librairies si courantes qu'elles sont utilisées dans beaucoup d'applications, il peut être profitable de les associer directement à Tomcat en les plaçant dans *TOMCAT_HOME/lib*. Cela évite de les réinclure systématiquement dans le déploiement de chaque application.

Vous trouverez également dans le sous-répertoire *doc* une documentation complète sur le connecteur JDBC de MySQL (applicable dans une large mesure à un autre système relationnel). Mettez cette documentation de côté pour vous y reporter car nous n'allons pas tout couvrir dans ce qui suit.

6.2 S2 : introduction à JDBC

Supports complémentaires :

- Diapos pour la session « S2 : Introduction à JDBC »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594bc86bqkkywd/

Nous allons découvrir JDBC par l'intermédiaire d'un ensemble d'exemples accessibles par notre application Web dans un unique contrôleur, que nous appelons *Jdbc* pour faire simple. Ce contrôleur communiquera lui-même avec un objet-modèle `TestJdbc` qui se chargera des accès à la base. Nous restons toujours dans un cadre MVC, avec l'exigence d'une séparation claire entre les différentes couches.

6.2.1 Contrôleur, vues et modèle

Chaque exemple JDBC correspondra à une action du contrôleur, et en l'absence d'action nous afficherons une simple page HTML présentant le menu des actions possibles. Nous suivons les règles définies à la fin du chapitre *Modèle-Vue-Contrôleur (MVC)*. Voici l'essentiel du code de cette *servlet*.

```
package controleurs;

import java.sql.*;

    /**
    * Servlet implementation class Jdbc
    */
    @WebServlet("/jdbc")
    public class Jdbc extends HttpServlet {

      private static final String SERVER="localhost", BD="webscope",
                    LOGIN="orm", PASSWORD="orm", VUES="/vues/jdbc/";

      protected void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws ServletException, IOException {

            // On devrait récupérer l'action requise par l'utilisateur
            String action = request.getParameter("action");
            // Notre objet modèle: accès à MySQL
            TestsJdbc jdbc;
            // La vue par défaut
```

(suite sur la page suivante)

(suite de la page précédente)

```
            String maVue = VUES + "index.jsp";

            try {
                    jdbc = new TestsJdbc();

                    if (action == null) {
                            // Rien à faire
                    } else if (action.equals("connexion")) {
                            // Action + vue test de connexion
                            jdbc.connect(SERVER, BD, LOGIN, PASSWORD);
                            maVue = VUES + "connexion.jsp";
                    } else if (action.equals("requeteA")) {
                        // Etc...
                    }
            } catch (Exception e) {
                    maVue = VUES + "exception.jsp";
                    request.setAttribute("message", e.getMessage());
            }

            // On transmet à la vue
            RequestDispatcher dispatcher = getServletContext()
                            .getRequestDispatcher(maVue);
            dispatcher.forward(request, response);
      }
}
```

Notez l'import de `java.sql.*` : toutes les classes de JDBC. L'ensemble des instructions contenant du code JDBC (encapsulé dans `TestJdbc`) est inclus dans un bloc `try` pour capturer les exceptions éventuelles.

Nous avons également déclaré des variables statiques pour les paramètres de connexion à la base. Il faudra faire mieux (pour partager ces paramètres avec d'autres contrôleurs), mais pour l'instant cela suffira.

Voici maintenant la page d'accueil du contrôleur (conventionnellement nommée `index.jsp`).

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

     <html>
     <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>Menu JDBC</title>
     </head>
     <body>

       <h1>Actions JDBC</h1>

       <ul>
             <li><a
                     href="${pageContext.request.contextPath}/jdbc?action=connexion">
↪Connexion</a>
             <li><a
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    href="${pageContext.request.contextPath}/jdbc?action=requeteA">
↪RequêteA</a>
            </li>
            <li>...</li>
        </ul>

    </body>
</html>
```

Et pour finir, nous avons besoin d'un *modèle* qui se chargera en l'occurrence de se connecter à MySQL et d'implanter les différentes actions de test des fonctionnalités JDBC. Appelons-le `TestsJdbc`, et plaçons-le dans le *package* `modeles`. Voici le squelette de cette classe, que nous détaillerons ensuite.

```
package modeles;

import java.sql.*;

public class TestsJdbc {
  private static final Integer port = 3306;

  /**
   * Pour communiquer avec MySQL
  */
  private Connection connexion;

  /**
   * Constructeur sans connexion
  */
 public TestsJdbc() throws ClassNotFoundException {
        /* On commence par "charger" le pilote MySQL */
        Class.forName("com.mysql.cj.jdbc.Driver");
  }
  (...)
}
```

Nous avons un objet privé, `connexion`, instance de la classe JDBC `Connection`. L'interface JDBC est abstraite : pour communiquer avec un serveur de données, il faut charger le pilote approprié. C'est ce que fait l'instruction :

```
Class.forName("com.mysql.jdbc.Driver");
```

Si vous avez bien placé le `jar` de MySQL dans le répertoire `WEB-INF/lib`, le chargement devrait s'effectuer correctement, sinon vous aurez affaire à une exception `ClassNotFoundException`. Voyons maintenant, pas à pas, comment développer des accès JDBC. Je vous invite à intégrer chacune des méthodes qui suit dans votre code, et à les tester sur le champ.

6.2.2 Connexion à la base

La première chose à faire (après le chargement du pilote) est la connexion à la base avec un compte utilisateur. Voici notre méthode `connect`.

```
public void connect(String server, String bd, String u, String p)
                throws SQLException {
        String url = "jdbc:mysql://" + server + ":" + port + "/" + bd;
        connexion = DriverManager.getConnection(url, u, p);
}
```

Quatre paramètres sont nécessaires (cinq si on compte le port qui est fixé par défaut à 3306).

- le nom du serveur : c'est pour nous `localhost`, sinon c'est le nom de domaine ou l'IP de la machine hébergeant le serveur MySQL ;
- le nom de la base
- le nom d'utilisateur MySQL (ici variable `u`) et le mot de passe (ici variable `p`).

Les deux premiers paramètres sont intégrés à une chaîne de connexion dont le format est lisible dans le code ci-dessus. Cette chaîne comprend notamment le serveur, le port et le nom de la base de données. Dans notre cas ce sera par exemple :

```
jdbc:mysql://locahost:3306/webscope
```

À vous de jouer. En affichant la vue par défaut (`index.jsp`) et en cliquant sur le choix `Connexion`, vous devriez déclencher l'action qui exécute cette méthode `connect`. Deux résultats possibles :

- soit tout se passe bien (c'est rarement le cas à la première tentative…) et la vue est `connexion.jsp` ;
- soit une exception est levée, et c'est la vue `exception.jsp` qui est choisie par le contrôleur ; elle devrait afficher le message d'erreur.

Prenez votre temps pour bien assimiler, si ce n'est déjà fait, les mécanismes d'interaction HTTP mis en place ici. Il est indispensable que vous familiarisiez avec les séquences de vue-contrôleur-action-modèle typiques des applications Web. Les fragments de code donnés précédemment sont presque complets, à vous des les finaliser (par exemple en créant la page `exception.jsp` qui affiche les messages d'exception).

Quand la connexion s'effectue correctement nous pouvons passer à la suite.

6.2.3 Exécution d'une requête

Nous pouvons en venir à l'essentiel, à savoir accéder aux données de la base. C'est ici que nous allons être confronté au fameux *impedance mismatch*, terme établi (quoique peu compréhensible) pour désigner l'incompatibilité de représentation des données entre une base relationnelle et une application objet. Concrètement, cette incompatibilité nécessite des conversions répétitives, et leur intégration dans une architecture générale (type MVC) s'avère laborieuse. Démonstration par l'exemple dans ce qui suit.

Le mécanisme *d'interrogation* de JDBC est assez simple :

- on instancie un objet `statement` par lequel on exécute une requête SQL ;
- l'exécution renvoie un objet `ResultSet` par lequel on peut parcourir le résultat.

C'est la méthode basique, avec des requêtes fixes dites « non préparées ». En première approche, nous ajoutons une méthode `chercheFilmA` dans `TestJdbc.java`.

```
public ResultSet chercheFilmsA() throws SQLException
{
        Statement statement = connexion.createStatement();
        return statement.executeQuery( "SELECT * FROM Film");
}
```

Comme vous pouvez le voir cette méthode fait peu de choses (ce qui est mauvais signe car le travail est délégué aux autres composants dont ce n'est pas le rôle). Elle retourne l'objet `ResultSet` créé par l'exécution de `SELECT * FROM Film`. Dans le contrôleur cette méthode est appelée par l'action `requeteA` que voici.

```
if (action.equals("requeteA")) {
        jdbc.connect(SERVER, BD, LOGIN, PASSWORD);
        ResultSet resultat = jdbc.chercheFilmsA();
        request.setAttribute("films", resultat);
        maVue = "/vues/jdbc/filmsA.jsp";
}
```

Cela impose donc d'importer les classes JDBC `java.sql.*` dans le contrôleur. Si vous avez bien assimilé les principes du MVC vous devriez commencer à ressentir un certain inconfort : nous sommes en train de propager des dépendances (la gestion d'une base de données) dans des couches qui ne devraient pas être concernées.

Le pire reste à venir : voici le vue `filmsA.jsp`.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ page import="java.sql.*" %>

<html>
 <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Liste des films</title>
 </head>
 <body>

      <h1>Liste des films</h1>

      <ul>
      <%
       ResultSet films = (ResultSet) request.getAttribute("films");
        while (films.next()) {
                out.println ("<li>Titre du film: " + films.getString("titre") + "</li>");
        }
      %>
      </ul>

      <p>
              <a href="${pageContext.request.contextPath}/jdbc">Accueil</a>
      </p>

 </body>
</html>
```

Rien ne va plus ! Nous avons été obligés d'introduire de la programmation Java dans notre vue. J'espère que vous êtes conscients des forts désavantages à long terme de cette solution. Concrètement, la gestion des accès JDBC est maintenant répartie entre le modèle, la vue et le contrôleur. Tout changement dans une couche implique des changements dans les autres : nous avons des *dépendances* qui compliquent (ou compliqueront) très sérieusement la vie de tout le monde à court ou moyen terme.

Comment faire mieux ? En restreignant strictement l'utilisation de JDBC au modèle. Après avoir vérifié que le code ci-dessus fonctionne (malgré tout), passez à la seconde approche ci-dessous.

6.2.4 Requête : deuxième approche

Voici une seconde méthode d'exécution de requête :

```
public List<Map<String, String>> chercheFilmsB() throws SQLException
{
        List<Map<String, String>> resultat = new ArrayList<Map<String, String>>();

        Statement statement = connexion.createStatement();
        ResultSet films = statement.executeQuery( "SELECT * FROM Film");
        while (films.next()) {
                Map<String, String> film = new HashMap<String, String> ();
                film.put("titre", films.getString("titre"));
                resultat.add(film);
        }
        // Et on renvoie
        return resultat;
}
```

Cette fois on ne renvoie plus un objet `ResultSet` mais une structure Java (que je vous laisse décrypter) permettant de représenter le résultat de la requête sans dépendance à JDBC. Vous voyez maintenant plus clairement le mécanisme de parcours d'un résultat avec JDBC : on boucle sur l'appel à la méthode `next()` de l'objet `ResultSet` qui se comporte comme un itérateur pointant successivement sur les lignes du résultat. Des méthodes `getInt()`, `getString()`, `getDouble()`, etc., récupèrent les valeurs des champs en fonction de leur type.

Le code du contrôleur change peu (je vous laisse l'adapter du précédent, à titre d'exercice). Voici en revanche la nouvelle version de la vue (en se restreignant au `body`) :

```
<h1>Liste des films</h1>

<ul>
  <c:forEach items="${requestScope.films}" var="film">
    <li> <c:out value="${film.titre}"/> </li>
  </c:forEach>
</ul>

<p>
        <a href="${pageContext.request.contextPath}/jdbc">Accueil</a>
</p>
```

Nous nous appuyons cette fois sur les balises de la JSTL, ce qui est permis par l'utilisation de structures Java standard en lieu et place du `ResultSet`. Le code est infiniment plus simple (j'espère que vous êtes d'accord) et surtout complètement indépendant de toute notion d'accès à une base de données. Il est possible par exemple de le tester sans recourir à MySQL.

Exercice : afficher les autres champs de la table *Film*

Vous ne devriez pas avoir de difficulté à compléter le code ci-dessus pour afficher tous les champs de la table Film : année, code du pays, etc.

Une fois l'exercice accompli, je vous propose une dernière version de cette même fonction.

6.2.5 Requête : troisième approche

La méthode qui précède est acceptable, mais elle souffre d'une limitation évidente : nous manipulons des structures (listes et tables de hachage) pour réprésenter des *entités* de notre domaine applicatif. Il serait bien plus logique et satisfaisant de représenter ces entités comme des *objets*, avec tous les avantages associés, comme l'encapsulation des propriétés, la possibilité d'associer des méthodes, la navigation vers d'autres objets associés (les acteurs, le metteur en scène), etc.

Pour cela, nous allons passer à une dernière étape : introduire une classe `Film` dans notre modèle, et modifier `TestJdbc` pour renvoyer des instances de `Film`.

Une version réduite de la classe `Film` est donnée ci-dessous. C'est un *bean* très classique.

```
package modeles;

/**
* Encapsulation de la représentation d'un film
*
 */
public class Film {

  private String titre;
  private Integer annee;

  public Film() {}

  public void setTitre(String leTitre) { titre = leTitre;}
  public void setAnnee(Integer lAnnee) { annee = lAnnee;}

  public String getTitre() {return titre;}
  public Integer getAnnee() {return annee;}
}
```

Et voici donc la méthode de notre classe JDBC.

```
public List<Film> chercheFilmsC() throws SQLException
{
        List<Film> resultat = new ArrayList<Film>();

        Statement statement = connexion.createStatement();
        ResultSet films = statement.executeQuery( "SELECT * FROM Film");
        while (films.next()) {
                Film film = new Film ();
                film.setTitre(films.getString("titre"));
                film.setAnnee(films.getInt("annee"));
                resultat.add(film);
        }
        // Et on renvoie
        return resultat;
}
```

Nous renvoyons une liste de films, et tout ce qui concerne la base de données (à part les exceptions éventuellement levées) est confiné à notre objet-service JDBC. Je vous laisse compléter le contrôleur et la vue. Ils changent très peu par rapport à la version précédente, ce qui confirme que nous sommes arrivés à un niveau d'indépendance presque total.

Tout cela demande un peu plus de travail que la version « basique », et un développeur peu consciencieux pourrait utili-

ser cet argument pour justifier de recourir au « vite fait/mal fait ». Il est vrai qu'il faut définir une classe supplémentaire, et appliquer des fastidieux `get` et `set` pour chaque propriété (la fameuse *conversion* nécessitée par l'incompatibilité des représentations dont nous parlions au début de cette session). Le bénéfice n'apparaît qu'à long terme. *Sauf* que nous allons bientôt voir comment nous épargner ces tâches répétitives, et qu'il n'y aura absolument plus d'excuse à ne pas faire les choses proprement.

Exercice : comme avant, afficher les autres champs de la table *Film*

Compléter le code donné ci-dessus pour gérer complètement la table *Film*, y compris l'identifiant du réalisateur, `id_realisateur`.

Exercice : écrire une classe Artiste et des méthodes de lecture

Ecrivez une classe Artiste pour représenter les artistes. Dans la classe `testJdbc`, écrivez une méthode :

```
Artiste chercheArtiste (Integer idArtiste)
```

C'est presque comme la méthode `chercheFilmsC`, mais on renvoie un seul objet. Utilisez la connexion JDBC, construisez la requête qui recherche l'artiste dont l'id est `idArtiste`, instantiez un objet `Artiste` avec les valeurs obtenues par la requête, renvoyez-le.

Exercice : permettre d'accéder au metteur en scène du Film

Voici maintenant un exercice complémentaire très instructif : à partir d'un film, nous aimerions récupérer son metteur en scène. Concevez une extension de notre petite application qui respecte soigneusement les principes exposés jusqu'ici. Vous devriez pouvoir utiliser le code produit dans les exercices précédents, la question préalable étant : *comment représenter l'association entre un film et son réalisateur ?*.

Le dernier exercice devrait déboucher sur quelques questions importantes relatives à la représentation d'une base de données relationnelles sous forme de graphe d'objet. Cette transformation (*mapping*) objet-relationnel apparaît très séduisante en terme d'ingénierie logicielle, mais demande cependant la mise en place de bonnes pratiques, et une surveillance sourcilleuse des accès à la base effectués dans les couches profondes de notre application. À suivre !

6.3 Résumé : savoir et retenir

Dans toute application, l'accès à une base de données relationnelle se fait par l'interface fonctionnelle (API) du SGBD. Dans le cas des applications Java cette interface (JDBC) est normalisée. Mais les mécanismes restent les mêmes quel que soit le langage et le système de bases de données utilisé : on effectue une requête, on récupère un curseur sur la résultat, ou itère sur ce curseur en obtenant à chaque étape une structure (typiquement un tableau) représentant une ligne de la table.

Dans une application objet, nous avons modélisé notre domaine fonctionnel par des classes (en UML par exemple), et les instances de ces classes (des objets donc) doivent être rendus persistants par stockage dans la base relationnelle. On se retrouve donc dans la situation de convertir des objets en lignes, et réciproquement de convertir des lignes ramenées par des requêtes en objets. C'est ce que nous avons fait dans ce chapitre : à un moment donné, on se retrouve dans la situation d'avoir à copie un tableau de données issu du `ResultSet` vers un objet. Cette copie est fastidieuse à coder et très répétitive.

Vous devriez, à l'issue de ce chapitre, être sensibilisé au problème et convaincu de deux choses :

— dans une application objet, tout doit être objet ; la couche de persistance, celle qui accède à la base de données, doit donc renvoyer des objets : elle est en charge de la conversion, et c'est l'approche finale à laquelle nous sommes parvenus ;
— à l'échelle d'une application non triviale (avec des dizaines de tables et des modèles complexes), la méthode consistant à coder manuellement les conversions est très pénalisante pour le développement et la maintenance ; une méthode automatique est grandement souhaitable.

Réflechissez bien à ces deux aspects : quand ils sont intégrés vous êtes mûrs pour adopter une solution ORM, que nous abordons dans le prochain chapitre.

CHAPITRE 7

Introduction à JPA et Hibernate

Nous avons donc un premier aperçu de JDBC, et la démarche logique que nous avons suivie en cherchant à respecter les principes d'indépendance dictés par notre approche MVC nous a mené à « encapsuler » les lignes d'une table relationnelle sous forme d'objet avec d'obtenir une intégration naturelle à l'application. Cela revient à isoler les accès à la base dans la couche modèle, de manière à ce que ces accès deviennent transparents pour le reste de l'application. À ce stade les avantages devraient déjà vous être perceptibles :

- possibilité de faire évoluer la couche d'accès aux données indépendamment du reste de l'application ;
- possibilité de tester séparément chaque couche, et même la couche modèle sans avoir à se connecter à la base (par création d'entités non persistantes).

Si ce n'est pas encore clair, j'espère que cela le deviendra. Au passage, on peut souligner qu'il est tout à fait possible de développer une application valide sans recourir à ces règles de conception qui peuvent sembler lourdes et peu justifiées. C'est possible mais cela devient de moins en moins facile au fur et à mesure que l'application grossit ainsi que les équipes en charge de la développer.

7.1 S1 : Concepts et mise en route

Supports complémentaires :

- Diapos pour la session « S1 : Concepts et mise en route »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594bcf65g3xaes/

Il est temps de clarifier cette notion de représentations incompatibles entre la base de données et une application objet. Le constat de cette incompatibilité justifie le développement des outils ORM, et il est bien utile de la maîtriser pour comprendre le but de ces outils.

7.1.1 Relationnel et objet

Admettons donc que nous avons décidé d'adopter cette approche systématique de séparation en couches, avec tout en bas une couche d'accès aux données basée sur une représentation objet.

Rôle d'un système ORM

Le rôle d'un système ORM est de convertir *automatiquement*, *à la demande*, la base de données sous forme d'un *graphe d'objet*. L'ORM s'appuie pour cela sur une *configuration* associant les *classes* du modèle fonctionnel et le schéma de la base de donnés. L'ORM génère des requêtes SQL qui permettent de matérialiser ce graphe ou une partie de ce graphe en fonction des besoins.

Pour bien comprendre cette notion de graphe et en quoi elle diffère de la représentation dans la base relationnelle, voici une petit échantillon de cette dernière.

Tableau 1 – Table des films

id	titre	année	idRéalisateur
17	Pulp Fiction	1994	37
54	Le Parrain	1972	64
57	Jackie Brown	1997	37

Tableau 2 – Table des artistes

id	nom	prénom	année_naissance
1	Coppola	Sofia	1971
37	Tarantino	Quentin	1963
64	Coppola	Francis	1939

Ce petit exemple illustre bien comment on représente une association en relationnel. C'est un mécanisme de référence *par valeur* (et pas par adresse), où la clé primaire d'une entité (ligne) dans une table est utilisée comme attribut (dit *clé étrangère*) d'une autre entité (ligne).

Ici, le réalisateur d'un film est un artiste dans la table `Artiste`, identifié par une clé nommée `id`. Pour faire référence à cet artiste dans la table `Film`, on ajoute un attribut *clé étrangère* `idRealisateur` dont la valeur est l'identifiant de l'artiste. Notez dans l'exemple ci-dessus que cet identifiant est 37 pour *Pulp Fiction* et *Jackie Brown*, 64 pour *Le parrain*, avec la correspondance à une et une seule ligne dans la table `Artiste`.

Voyons maintenant la représentation équivalente dans un langage objet en général (et donc java en particulier). Nous avons des objets, et la capacité à référencer un objet depuis un autre objet (cette fois par un système d'adressage). Par exemple, en supposant que nous avons une classe `Artiste` et une classe `Film` en java, le fait qu'un film ait un réalisateur se représente par une référence à un objet `Artiste` sous forme d'une propriété de la classe `Film`.

```
class Film {

  (...)

  Artiste realisateur;

  (...)
}
```

Et du côté de la classe `Artiste`, nous pouvons représenter les films réalisés par un artiste par un ensemble.

```
class Artiste {

  (...)

  Set<Film> filmsDiriges;

  (...)
}
```

Important : On peut noter dès maintenant qu'une différence importante entre les associations en relationnel et en java est que les premières sont *bi-directionnelles*. Il est toujours possible par une requête SQL (une jointure) de trouver les films réalisés par un artiste, ou le réalisateur d'un film. En java, le lien peut être représenté de chaque côté de l'association, ou des deux. Par exemple, on pourrait mettre la propriété `réalisateur` dans la classe `Film`, mais pas `filmsDiriges` dans la classe `Artiste`, ou l'inverse, ou les deux. Cette subtilité est la source de quelques options plus ou moins obscures dans les systèmes ORM, nous y reviendrons.

La base de données, transformée en java, se présentera donc sous la forme d'un graphe d'objet comme celui montré par la figure *Le graphe d'objets Java*.

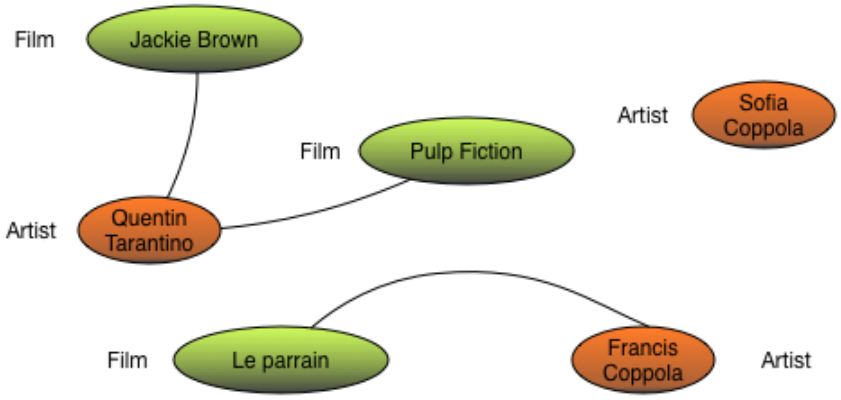

FIG. 1 – Le graphe d'objets Java

Lisez et relisez cette section jusqu'à ce qu'elle soit limpide (et sollicitez vos enseignants si elle ne le devient pas). Les notions présentées ci-dessous sont à la base de tout ce que nous allons voir dans les prochains chapitres.

7.1.2 La couche ORM

Pour bien faire, il nous faut une approche systématique. On peut par exemple décréter que :

- on crée une classe pour chaque entité ;
- on équipe cette classe avec des méthodes *create()*, *get()*, *delete()*, *search()*, . . . , que l'on implante en SQL/JDBC ;
- on implante la navigation d'une entité à une autre, à l'aide de requête SQL codées dans les classes et exécutées en JDBC.

Il ne serait pas trop difficile (mais pas trop agréable non plus) de faire quelques essais selon cette approche. Par exemple la méthode *get(id)* est clairement implantée par un `SELECT * From <table> WHERE id=:id`. Le *create()* est implanté par un `INSERT INTO ()`, etc. Nous ne le ferons pas pour au moins trois raisons :

- concevoir nous-mêmes un tel mécanisme est un moyen très sûr de commettre des erreurs de conception qui se payent très cher une fois que des milliers de lignes de code ont été produites et qu'il faut tout revoir ;

— la tâche répétitive de produire des requêtes toujours semblables nous inciterait rapidement à chercher un moyen automatisé ;
— et enfin - et surtout - ce mécanisme dit *d'association objet-relationnel* (ORM pour *Objet-relational mapping*) a été mis au point, implanté, et validé depuis de longues années, et nous disposons maintenant d'outils matures pour ne pas avoir à tout inventer/développer nous-mêmes.

Non seulement de tels outils existent, mais de plus ils sont normalisés dans la plate-forme java, sous le nom générique de *Java Persistence API* ou JPA. JPA est essentiellement une spécification intégrée au JEE qui vise à standardiser la couche d'association entre une base relationnelle et une application Java construite sur des objets (à partir de maintenant nous appelerons cette couche ORM). Le rôle de la couche ORM est illustré par la figure *Le mapping d'une base relationnelle en graphe d'objets java*.

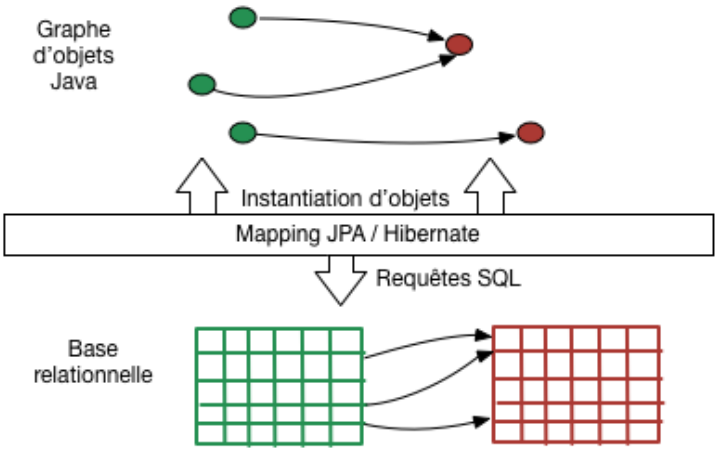

FIG. 2 – Le *mapping* d'une base relationnelle en graphe d'objets java

Il est important (ou du moins utile) de signaler que cette standardisation vient après l'émergence de plusieurs *frameworks* qui ont exploré les meilleures pratiques pour la réalisation d'un système ORM. Le plus connu est sans doute Hibernate, que nous allons utiliser dans ce qui suit (voir http://fr.wikipedia.org/wiki/Mapping_objet-relationnel pour une liste). Avec l'apparition de JPA, ces frameworks vont tendre à devenir des *implantations* particulières de la norme (rappelons que JPA est une spécification visant à la standardisation). Comme JPA est une sorte de facteur commun, il s'avère que chaque *framework* propose des extensions spécifiques, souvent très utiles. Il existe une implantation de référence de JPA, EclipseLink.

JPA définit une interface dans le package `javax.persistence.*`. La définition des associations avec la base de données s'appuie essentiellement sur les annotations Java, ce qui évite de produire de longs fichiers de configuration XML qui doivent être maintenus en parallèle aux fichiers de code. On obtient une manière assez légère de définir une sorte de base de données objet *virtuelle* qui est matérialisée au cours de l'exécution d'une application par des requêtes SQL produites par le *framework* sous-jacent.

Le choix que j'ai fait dans ce qui suit est d'utiliser Hibernate comme *framework* de persistance, et de respecter JPA autant que possible pour que vous puissiez passer à un autre framework sans problème (en particulier, les annotations sont utilisées systématiquement).

Dans ce chapitre nous allons effectuer nos premier pas avec JPA/Hibernate, avec pour objectif de construire une première couche ORM pour notre base *webscope*. L'approche utilisée est assez simple, elle nous servira de base de départ pour découvrir progressivement des concepts plus avancés.

Dernière chose : les ORM s'appuient sur JDBC et nous allons donc retrouver certains aspects déjà connus (et en découvrir d'autres).

Exercice : comprendre la représentation par graphe

Prenez les deux films suivants : *Impitoyable* et *Seven*, avec leurs acteurs et metteurs en scène, et construisez le graphe d'objets.

7.2 S2 : Premiers pas avec Hibernate

Supports complémentaires :

— Diapos pour la session « S2 : Premiers pas avec Hibernate »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594bdb82xlnti5/

7.2.1 Installation

Vous allez installer Hibernate dans votre environnement, en commençant par récupérer la dernière version stable sur le site Hibernate. À l'heure où j'écris, cette version est la 5.4, les exemples connées dans ce qui suit ne devraient pas dépendre, sauf sur des points mineurs, de la version.

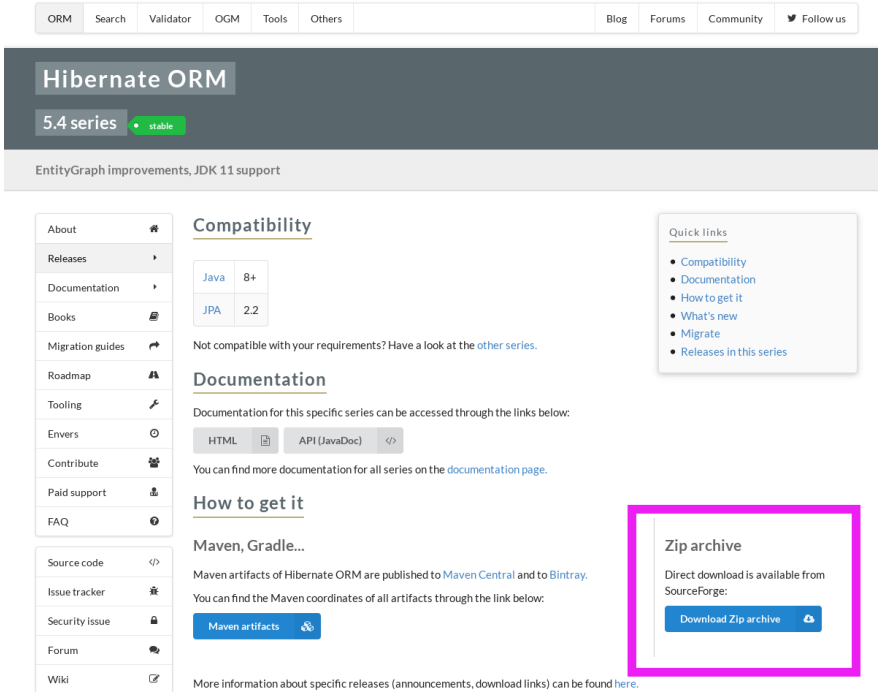

FIG. 3 – La page de téléchargement d'Hibernate (cliquez sur le bouton dans la zone violette)

Dans le répertoire `hibernate` obtenu après décompression, vous trouverez une documentation complète et les librairies, dans `lib`. Elles sont groupées en plusieurs catégories correspondant à autant de sous-répertoires (notez par exemple celui nommé `jpa`). Pour l'instant nous avons besoin de toutes les librairies dans `required` : copiez-les, comme d'habitude, dans le répertoire `WEB-INF/lib` de votre application Web. Le pilote MySQL doit bien entendu rester en place : il est utilisé par Hibernate. Utilisez l'option `refresh` sous Eclipse sur votre projet (clic bouton droit) pour que ces nouvelles librairies soient identifiées.

Un des objets essentiels dans une application Hibernate est l'objet `Session` qui est utilisé pour communiquer avec la base de données. Il peut être vu comme une généralisation/extension de l'objet `Connection` de JDBC. Une session est créée à partir d'un ensemble de paramètres (dont les identifiants de connexion JDBC) contenus dans un fichier de configuration XML nommé `hibernate.cfg.xml`.

Important : Pour que ce fichier (et toute modification affectant ce fichier) soit automatiquement déployé par Tomcat, il doit être placé dans un répertoire `WEB-INF/classes`. Créez ce répertoire s'il n'existe pas.

Voici le fichier de configuration minimal avec lequel nous débutons.

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
      <session-factory>
            <!-- local connection properties -->
            <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
↪webscope</property>
            <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</
↪property>
            <property name="hibernate.connection.username">orm</property>
            <property name="hibernate.connection.password">orm</property>
            <property name="hibernate.connection.pool_size">10</property>

            <!-- dialect for MySQL -->
            <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>

            <property name="hibernate.show_sql">true</property>
            <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</
↪property>
            <property name="cache.use_query_cache">false</property>

      </session-factory>
</hibernate-configuration>
```

On retrouve les paramètres qui vont permettre à Hibernate d'instancier une connexion à MySQL avec JDBC (mettez vos propres valeurs bien sûr). Par ailleurs chaque système relationnel propose quelques variantes du langage SQL qui vont être prises en compte par Hibernate : on indique le dialecte à utiliser. Finalement, les derniers paramètres ne sont pas indispendables mais vont faciliter notre découverte, notamment avec l'option `show_sql` qui affiche les requêtes SQL générées dans la console java.

Il peut y avoir plusieurs `<session-factory>` dans un même fichier de configuration, pour plusieurs bases de données, éventuellement dans plusieurs serveurs différents.

7.2.2 Test de la connexion

Comme pour JDBC, nous créons

- un contrôleur `Hibernate.java` associé à l'URL `/hibernate`. Il devrait avoir grosso modo la même structure que celui utilisé pour JDBC puisque nous allons reproduire les mêmes actions ;
- une liste d'actions implantés dans une classe de tests nommée `TestsHibernate.java` ;
- des vues placées dans `WebContent/vues/hibernate`.

Nous commençons par tester que notre paramétrage est correct. Voici le squelette de `TestsHibernate.java`.

```
package modeles;

import org.hibernate.Session;

public class TestsHibernate {

   /**
    * Objet Session de Hibernate
    */
   private Session session;

   /**
    * Constructeur établissant une connexion avec Hibernate
    */
   public TestsHibernate()  {
     Configuration configuration = new Configuration().configure("/hibernate.cfg.xml");
     ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().
↪applySettings(configuration.getProperties()).build();
     SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
     session = sessionFactory.openSession();
   }

}
```

Pour créer une connexion avec Hibernate nous passons par une `SessionFactory` qui repose sur le paramétrage de notre fichier de configuration (ce fichier lui-même est chargé dans un objet `Configuration`). Si tout se passe bien, on obtient un objet par lequel on peut ouvrir/fermer des sessions. Plusieurs raisons peuvent faire que ça ne marche pas (au moins du premier coup).

- le fichier de configuration n'est pas trouvé (il doit être dans le `CLASSPATH` de l'application) ;
- ce fichier contient des erreurs ;
- les paramètres de connexion sont faux ;
- et toutes sortes d'erreurs étranges qui peuvent nécessiter le redémarrage de Tomcat..

En cas de problème une exception sera levée avec un message plus ou moins explicite. Testons directement cette connexion. Voici l'action de connexion :

```
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws␣
↪ServletException, IOException {
  TestsHibernate tstHiber = new TestsHibernate();
  String maVue = VUES + "connexion.jsp";
  RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(maVue);
  dispatcher.forward(request, response);
}
```

Si cela fonctionne du premier coup, bravo, sinon cherchez l'erreur (avec notre aide bienveillante si nécessaire). On y arrive toujours.

7.2.3 Ma première entité

Dans JPA, on associe une table à une classe Java annotée par `@Entity`. Chaque ligne de la table devient une *entité*, instance de cette classe. La classe doit obéir à certaines contraintes qui sont à peu près comparables à celle d'un JavaBean. Les propriétés sont privées, et des accesseurs *set* et *get* sont définis pour chaque propriété. L'association d'une propriété à une colonne de la table est indiquée par des annotations.

L'exemple qui suit est plus parlant que de longs discours :

```
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Pays  {
      @Id
      String code;
      public void setCode(String c) {code = c;}
      public String getCode() {return code ;}

      @Column
      String nom;
      public void setNom(String n) {nom = n;}
      public String getNom() {return nom;}

      @Column
      String langue;
      public void setLangue(String l) {langue = l;}
      public String getLangue() {return langue;}
}
```

Remarquez l'import du *package* `javax.persistence`, et les annotations suivantes :

- `@Entity` : indique que les instances de la classe sont *persistantes* (stockées dans la base) ;
- `@Id` : indique que cette propriété est la clé primaire ;
- `@Column` : indique que la propriété est associée à une colonne de la table.

Et c'est tout. Bien entendu les annotations peuvent être beaucoup plus complexes. Ici, on s'appuie sur des choix par défaut : le nom de la table est le même que le nom de la classe ; idem pour les propriétés, et l'identifiant n'est pas auto-généré.

Cela nous suffit pour un premier essai. Il reste à déclarer dans le fichier de configuration que cette entité persistante est dans notre base :

```
<session-factory>
     (...)
            <mapping class="modeles.webscope.Pays"/>
    </session-factory>
```

Hibernate ne connaitra le *mapping* entre une classe et une table que si elle est déclarée dans la configuration.

Une autre solution pour déclarer une classe persistante est de la charger explicitement dans la configuration.

```
  public TestsHibernate() {
    Configuration configuration = new Configuration().configure();

    // ICI ON AJOUTE LES CLASSES JPA
```

(suite sur la page suivante)

(suite de la page précédente)

```
    configuration.addAnnotatedClass(Pays.class);
    // FIN DE L'AJOUT DES CLASSES JPA

    ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
            .applySettings(configuration.getProperties()).build();

    SessionFactory sessionFactory = configuration
            .buildSessionFactory(serviceRegistry);

    session = sessionFactory.openSession();
}
```

À vous de choisir. Notez que l'absence de déclaration d'un *mapping* entraîne une exception `Unknown Entity`. La déclaration de l'entité avec l'une des deux méthodes ci-dessus doit régler le problème.

Important : Depuis la version 5, il semble qu'Hibernate ne prenne plus en compte les instructions du document de configuration XML. *Il est donc impératif d'ajouter les classes persistantes au moment du chargement de la configuration, comme indiqué ci-dessus.*

7.2.4 Insertion d'une entité

Maintenant définissez une action `insertion` dans votre contrôleur, avec le code suivant.

```
TestsHibernate tstHiber = new TestsHibernate();
Pays monPays = new Pays();
monPays.setCode("is");
monPays.setNom("Islande");
monPays.setLangue("islandais");
tstHiber.insertPays(monPays);
maVue = VUES + "insertion.jsp";
```

Vous voyez que l'on instancie et utilise un objet `Pays` comme n'importe quel *bean*. La capacité de l'objet est devenir persistant n'apparaît pas du tout. On peut l'utiliser comme un objet « métier », *transient* (donc non sauvegardé dans la base). Pour le sauvergarder on appelle la méthode `insertPays` de notre classe utilitaire, que voici.

```
session.beginTransaction();
session.save(pays);
session.getTransaction().commit();
```

Il suffit donc de demander à la session de sauvegarder l'objet (dans le cadre d'une transaction) et le tour est joué. Hibernate se charge de tout : génération de l'ordre SQL correct, exécution de cet ordre, validation. En appliquant ce code vous devriez obtenir dans la console java l'affichage de la requête SQL. Vous pouvez vérifier avec phpMyAdmin que l'insertion s'est bien faite.

Exercice : associer un formulaire de saisie, et une vue

Il s'agit de compléter notre première fonction d'insertion en créant une formlaire pour saisir les paramètres d'un pays. La validation de ce formulaire doit déclencher l'insertion dans la base et afficher une vue donnant les valeurs affichées.

7.2.5 Lecture de données

Voyons maintenant comment lire des données de la base. Nous avons plusieurs possibilités.

— *Transmettre une requête SQL via Hibernate.* C'est la moins portable des solutions car on ne peut pas être sûr à 100% que la syntaxe est compatible d'un SGBD à un autre ; il est vrai qu'on ne change pas de SGBD tous les jours.

— *Transmettre une requête HQL.* Hibernate propose un langage d'interrogation proche de SQL qui est transcrit ensuite dans le dialecte du SGBD utilisé.

— *Utiliser l'API* `Criteria`. Plus de langage externe (SQL) passé plus ou moins comme une chaîne de caractères : on construit une requête comme un objet. Cette interface a également l'avantage d'être normalisée en JPA.

Note : JPA définit un langage de requête, JPQL (*Java Persistence Query Language*) qui est un sous-ensemble de HQL. Toute requête JPQL est une requête HQL, l'inverse n'est pas vrai.

Passons à la pratique. Voici la requête qui parcours la table des pays, avec l'API `Criteria`.

```
public List<Pays> lecturePays() {
   // Create CriteriaBuilder
   CriteriaBuilder builder = session.getCriteriaBuilder();

   // Create CriteriaQuery
   CriteriaQuery<Pays> criteria = builder.createQuery(Pays.class);
   criteria.from(Pays.class);
   // Execute query
   List<Pays> pays = session.createQuery(criteria).getResultList();

   return pays;
}
```

La requête était assez simple avec des versions antérieures de Criteria, mais elle reste assez lisible : on crée un constructeur de requêtes, on lui donne une requête avec une classe Java (ici Pays), et l'on récupère la liste des résultats dans une liste après exécution.

Bien sûr pour des requêtes plus complexes, la construction est plus longue. Voici la méthode équivalente en HQL, le langage associé à Hibernate, que nous explorerons en détail dans le chapitre *Le langage HQL*.

```
public List<Pays> lecturePaysHQL() {
        Query query = session.createQuery("from Pays");
        return query.list();
}
```

C'est presque la même chose, mais vous voyez ici que l'on introduit une chaîne de caractères contenant une expression syntaxique qui n'est pas du java (et qui n'est donc pas contrôlée à la compilation).

Exercice : afficher la liste des pays

À vous de jouer : créez l'action, la vue et le modèle pour afficher la liste des pays en testant les deux versions ci-dessus.

7.2.6 Gérer les associations

Et nous concluons cette introduction avec un des aspects les plus importants du *mapping* entre la base de données et les objets java : la représentation des associations. Pour l'instant nous nous contentons de l'association (plusieurs à un) entre les films et les pays (»*plusieurs* films peuvent avoir été tournés dans *un seul* pays »). En relationnel, nous avons donc dans la table `Film` un attribut `code_pays` qui sert de clé étrangère. Voici comment on représente cette association avec Hibernate.

```
 package modeles.webscope;

 import javax.persistence.*;

@Entity
 public class Film {

       @Id
       private Integer id;
       public void setId(Integer i) {id = i;}

       @Column
       String titre;
       public void setTitre(String t) {titre= t;}
       public String getTitre() {return titre;}

       @Column
       Integer annee;
       public void setAnnee(Integer a) {annee = a;}
       public Integer getAnnee() {return annee;}

       @ManyToOne
       @JoinColumn (name="code_pays")
       Pays pays;
       public void setPays(Pays p) {pays = p;}
       public Pays getPays() {return pays;}
}
```

Cette classe est incomplète : il manque le genre, le réalisateur, etc. Ce qui nous intéresse c'est le lien avec `Pays` qui est représenté ici :

```
@ManyToOne
@JoinColumn (name="code_pays")
Pays pays;
public void setPays(Pays p) {pays = p;}
public Pays getPays() {return pays;}
```

On découvre une nouvelle annotation, `@ManyToOne`, qui indique à Hibernate que la propriété `pays`, instance de la classe `Pays`, encode un des côtés d'une association plusieurs-à-un entre les films et les pays. Pour instancier le pays dans lequel un film a été tourné, Hibernate devra donc exécuter la requête SQL qui, connaissant un film, permet de trouver le pays associé. Cette requête est :

```
select * from Pays where code = :film.code-pays
```

La *clé étrangère* qui permet de trouver le pays est `code_pays` dans la table `Film`. C'est ce qu'indique la seconde annotation, `@JoinColumn`.

Note : N'oubliez pas de modifier votre fichier de configuration pour indiquer que vous avez défini une nouvelle classe « mappée ».

Prenez le temps de bien réfléchir pour vous convaincre que toutes les informations nécessaires à la constitution du lien objet entre une instance de la classe `Film` et une instance de la classe `Pays` sont là. Pour le dire autrement : toutes les informations permettant à Hibernate d'engendrer la requête qui précède ont bien été spécifiées par les annotations.

Exercice : afficher la liste des films et leur pays

À vous de jouer : créez une action `listeFilms` qui affiche la liste des films et le pays où ils ont été tournés. Dans la JSTL, l'affichage du pays ne devrait pas être plus compliqué que :

```
${film.pays.nom}
```

Au passage regardez dans la console les requêtes transmises par Hibernate : instructif.

Important : Vous noterez que nous avons défini l'association du côté `Film` mais *pas* du côté `Pays`. Etant donné un objet pays, nous de pouvons pas accéder à la liste des films qui y ont été tournés. *L'association est uni-directionnelle*. Gardez cela en mémoire mais ne vous grillez pas les neurones dessus : nous allons y revenir.

Vous devriez maintenant pouvoir implanter avec Hibernate l'action qui affiche la liste des films avec leur metteur en scène.

Exercice : afficher la liste des films et leur metteur en scène

Aide : il faut définir la classe `Artiste`, mappée, et la lier à `Film` pour représenter l'association. Je vous aide : dans une approche « graphe d'objet », le metteur en scène est un objet propriété de `Film`.

À l'execution, examinez les requêtes SQL produites par Hibernate et méditez sur ce que cela implique. Prenez également le recul pour apprécier ce qu'apporte Hibernate par rapport à la gestion manuelle que nous avons envisagée dans le chapitre précédent.

7.3 Résumé : savoir et retenir

Les notions suivantes devraient être claires maintenant :

- une application objet représente les données comme un *graphe d'objet*, liés par des références ;
- une couche ORM transforme une base relationnelle en graphe d'objets et permet à l'application de *naviguer* dans ce graphe en suivant les références entre objets ;
- la transformation est effectuée à la volée en fonction des navigations (accès) effectués par l'application ; elle repose sur la génération de requêtes SQL ;

Hibernate est une implantation de la spécification JPA, et un peu plus que cela.

CHAPITRE 8

JPA : le *mapping*

Nous abordons maintenant une étude plus systématique de JPA/Hibernate en commençant par la définition du modèle de données, ou plus exactement de l'association (*mapping*) entre la base de données relationnelle et le modèle objet java. Nous parlerons plus concisément de *mapping O/R* dans ce qui suit. Rappelons le but de cette spécification : transformer *automatiquement* une base relationnelle en graphe d'objets java. Cette transformation est effectuée par Hibernate sur des données extraites de la base avec des requêtes SQL.

Note : nous considérons pour l'instant que la base est pré-existante, comme notre base *webscope*. Une autre possibilité, plus radicale, est de définir un modèle objet et de laisser Hibernate générer le schéma relationnel correspondant.

Le but de ce chapitre est principalement de finaliser notre modèle java pour la base *webscope*, ce qui couvrira les options les plus courantes du *mapping* O/R. Il existe plusieurs méthodes pour définir un modèle O/R.

— par un fichier de configuration XML ;
— par des annotations directement intégrées au code java.

L'option « annotation » présente de nombreux avantages, pour la clarté, les manipulations de fichier, la concision. C'est donc celle qui est présentée ci-dessous. Reportez-vous à la documentation Hibernate si vous voulez inspectez un fichier de configuration XML typique.

Prudence : Attention, quand vous regardez les (innombrables) documentations sur le Web, à la date de publication ; les outils évoluent rapidement, et beaucoup de tutoriaux sont obsolètes.

Pour utiliser les annotations JPA, il faut inclure le *package* `javax.persistence.*`. Il nous semble préférable de suivre JPA le plus possible plutôt que la syntaxe spécifique à Hibernate, la tendance de toute façon étant à la convergence de ce dernier vers la norme.

8.1 S1 : Entités et composants

Supports complémentaires :

— Diapos pour la session « S1 : Entités et composants »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3594c278sxffxru/

Une entité, déclarée par l'annotation `@Entity` définit une classe Java comme étant *persistante* et donc associée à une table dans la base de données. Cette classe doit être implantée selon les normes des beans : propriétés déclarées comme n'étant pas publiques (*private* est sans doute le bon choix), et accesseurs avec *set* et *get*, nommés selon les conventions habituelles.

Par défaut, une entité est associée à la table portant le même nom que la classe. Il est possible d'indiquer le nom de la table par une annotation `@Table`. En voici un exemple (parfaitement inutile en l'occurrence) :

```
@Entity
@Table(name="Film")
public class Film {
  ...
}
```

De nombreuses options JPA, et plus encore Hibernate, sont disponibles, mais nous allons nous limiter à l'essentiel pour comprendre les mécanismes. Vous pourrez toujours vous reporter à la documentation pour des besoins ponctuels.

La norme JPA indique qu'il est nécessaire de créer un constructeur vide pour chaque entité. Le *framework* peut en effet avoir à instancier une entité avec un appel à `Constructor.newInstance()` et le constructeur correspondant doit exister. La norme JPA indique que ce constructeur doit être public ou protégé :

```
public Film() {}
```

La méthode ci-dessus a l'inconvénient de créer un objet dans un état invalide (aucune propriété n'a de valeur). Cela ne pose pas de problème quand c'est Hibernate qui utilise ce constructeur puisqu'il va affecter les propriétés en fonction des valeurs dans la base, mais du point de vue de l'application c'est une source potentielle d'ennuis. Avec Hibernate, le constructeur vide peut être déclaré `private` (mais ce n'est pas la recommandation JPA).

Finalement, il est recommandé, au moins pour certaines classes, d'implanter des méthodes *equals()* et *hashCode()* pour qu'Hibernate puisse déterminer si deux objets correspondent à la même ligne de la base. C'est un sujet un peu trop avancé pour l'instant, et nous le laissons donc de côté.

Important : Nous nous soucions essentiellement pour l'instant d'opération de *lecture*, qui suffisent à illustrer et valider la modélisation du *mapping*.

8.1.1 Identifiant d'une entité

Toute entité doit avoir une propriété déclarée comme étant l'identifiant de la ligne dans la table correspondante. Il est beaucoup plus facile de gérer une clé constituée d'une seule valeur qu'une clé composée de plusieurs. Les bases de données conçues selon des principes de *clé artificielle* ou *surrogate key* (produite à partir d'une séquence) sont de loin la meilleure solution. On peut être confronté à une base qui n'a pas été conçue sur ce principe, auquel cas JPA/Hibernate fournit des méthodes permettant de faire face à la situation, mais c'est plus compliqué. Nous présentons une solution à la fin de ce chapitre.

L'identifiant est indiqué avec l'annotation `@Id`. Pour produire automatiquement les valeurs d'identifiant, on ajoute une annotation `@GeneratedValue` avec un paramètre `Strategy`. Voici deux possibilités pour ce paramètre :

— `Strategy = GenerationType.AUTO`. Hibernate produit lui-même la valeur des identifiants grâce à une table `hibernate_sequence`.

— `Strategy = GenerationType.IDENTITY`. Hibernate s'appuie alors sur le mécanisme propre au SGBD pour la production de l'identifiant. Dans le cas de MySQL, c'est l'option `AUTO-INCREMENT`, dans le cas de postgres ou Oracle, c'est une séquence. C'est à vous de vous assurer que pour chaque table, ce mécanisme est en place.

Voici les commandes pour `Film` ou `Artiste` en utilisant le mécanisme automatique d'Hibernate, *en supposant que les clés primaires ne sont pas auto-incrémentées*.

```
@Id
@GeneratedValue(Strategy = GenerationType.AUTO)
private Integer id;
private void setId(Integer i) { id = i; }
public Integer getId()  { return id; }
```

Note : en stratégie `GenerationType.AUTO`, Hibernate doit se charger de créer une table `hibernate_sequence`.

Si la clé est auto-incrémentée dans MySQL, l'annotation est la suivante.

```
@Id
@GeneratedValue(Strategy = GenerationType.IDENTITY)
private Integer id;
```

Faut-il fournir des accesseurs `setId()` et `getId()` ? Conceptuellement, l'id est utilisé pour lier un objet à la base de données et ne joue aucun rôle dans l'application. Il n'a donc pas de raison d'apparaître publiquement. d'où l'absence de méthode pour le récupérer dans la plupart des objets métiers.

— fournir une méthode *publique* `setId()` pour un identifiant auto-généré ne semble pas une bonne idée, car dans ce cas l'application pourrait affecter un identifiant en conflit avec la méthode de génération ; il est donc préférable de créer une méthode privée ;

— fournir une méthode `getId()` n'est pas nécessaire, sauf si l'application souhaite inspecter la valeur de l'identifiant en base de données, ce qui est de fait souvent utile.

En résumé, la méthode `setId()``devrait être ``private`. Notez qu'Hibernate utilise l'API `Reflection` de java pour acéder aux propriétés des objets, et n'a donc pas besoin de méthodes publiques.

Note : Avec Hibernate, l'existence de `setId()` ne semble même pas nécessaire. Il semble que JPA requiert des accesseurs pour chaque propriété mappée, donc autant respecter cette règle pour un maximum de compatibilité.

8.1.2 Les colonnes

Par défaut, toutes les propriétés non-statiques d'une classe-entité sont considérées comme devant être stockées dans la base. Pour indiquer des options (et aussi pour des raisons de clarté à la lecture du code) on utilise le plus souvent l'annotation `@Column`, comme par exemple :

```
@Column
private String nom;
public void setNom(String n) {nom= n;}
public String getNom() {return nom;}
```

Cette annotation est utile pour indiquer le nom de la colonne dans la table, quand cette dernière est différente du nom de la propriété en java. Dans notre cas nous utilisons des règles de nommage différentes en java et dans la base de données pour les noms composés de plusieurs mots. `@Column` permet alors d'établir la correspondance :

```
@Column(name="annee_naissance")
private Integer anneeNaissance;
public void setAnneeNaissance(Integer a) {anneeNaissance = a;}
public Integer getAnneeNaissance() {return anneeNaissance;}
```

Voici les principaux attributs pour `@Column`.

- `name` indique le nom de la colonne dans la table ;
- `length` indique la taille maximale de la valeur de la propriété ;
- `nullable` (avec les valeurs `false` ou `true`) indique si la colonne accepte ou non des valeurs à `NULL` (au sens « base de données » du terme : une valeur à `NULL` est une absence de valeur) ;
- `unique` indique que la valeur de la colonne est unique.

Note : Un objet métier peut très bien avoir des propriétés que l'on ne souhaite pas rendre persistantes dans la base. Il faut alors impérativement les marquer avec l'annotation `@Transient`.

8.1.3 Les composants

Une *entité* existe par elle-même indépendamment de toute autre entité, et peut être rendue persistante par insertion dans la base de données, avec un identifiant propre. Un *composant*, au contraire, est un objet sans identifiant, qui ne peut être persistant que par rattachement (direct ou transitif) à une entité.

La notion de composant résulte du constat qu'une ligne dans une base de données peut *parfois* être décomposée en plusieurs sous-ensemble dotés chacun d'une logique autonome. Cette décomposition mène à une granularité fine de la représentation objet, dans laquelle on associe *plusieurs* objets à *une* ligne de la table.

On peut sans doute modéliser une application sans recourir aux composants, au moins dans la définition stricte ci-dessus. Ils sont cependant également utilisés dans Hibernate pour gérer d'autres situations, et notamment les clés composées de plusieurs attributs, comme nous le verrons plus loin. Regardons donc un exemple concret, celui (beaucoup utilisé) de la représentation des *adresses*. Prenons donc nos internautes, et ajoutons à la table quelques champs (au minimum) pour représenter leur adresse.

```
ALTER TABLE Internaute ADD adresse TEXT,
    ADD code_postal VARCHAR(10), ADD ville VARCHAR(100);
```

Utilisez phpMyAdmin pour insérer quelques adresses aux internautes existant dans la base, et passons maintenant à la modélisation java.

On va considérer ici que l'adresse est un *composant* de la représentation d'un internaute qui dispose d'une unité propre et distinguable du reste de la table. Cela peut se justifier, par exemple, par la nécessité de contrôler qu'un code postal est correct, une logique applicative qui n'est pas vraiment pertinente pour l'entité *Internaute*. Un autre argument est qu'on peut réutiliser la définition du composant pour d'autres entités, comme par exemple `Société`.

Toujours est-il que nous décidons de représenter une adresse comme un composant, désigné par le mot-clé `Embeddable` en JPA.

```
package modeles.webscope;

import javax.persistence.*;

@Embeddable
public class Adresse {
    String adresse;
        public void setAdresse(String v) {adresse = v;}
```

(suite sur la page suivante)

(suite de la page précédente)

```
        public String getAdresse() {return adresse;}

        @Column(name="code_postal")
            String codePostal;
        public void setCodePostal(String v) {codePostal = v;}
        public String getCodePostal() {return codePostal;}

        String ville;
        public void setVille(String v) {ville = v;}
        public String getVille() {return ville;}
    }
```

La principale différence visible avec une entité est qu'un composant n'a pas d'identifiant (marqué `@Id`) et ne peut donc pas être sauvegardé dans la base indépendamment. Il faut au préalable le rattacher à une entité, ici `Internaute`.

```
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Internaute {

      @Id
      private String email;
      public void setEmail(String e) {email = e;}

      @Column
      private String nom;
      public void setNom(String n) {nom = n;}
      public String getNom() {return nom;}

      @Column
      private String prenom;
      public void setPrenom(String p) {prenom = p;}
      public String getPrenom() {   return prenom;}

      @Embedded
      private Adresse adresse;
      public void setAdresse(Adresse a) {adresse = a;}
      public Adresse getAdresse() {return adresse;}
}
```

Au lieu de l'annotation `Column`, on utilise `@Embedded` et le tour est joué. La propriété `adresse` devient, comme `nom` et `prénom`, une propriété *persistante* de l'entité.

Exercice : affichez la liste des internautes.

Implantez une action qui affiche la liste des internautes avec leur adresse.

Vous remarquerez que la classe `Adresse` contient des annotations qui le *mappent* vers la table, notamment avec les noms de colonne. Mais que se passe-t-il alors si on place deux composants du même type dans une entité ? Par exemple, si on veut avoir une adresse personnelle et une adresse professionnelle pour un internaute ? On *mapperait* dans ce cas

deux propriétés distinctes vers la *même* colonne. Pour illustrer le problème (et la solution), commençons par modifier la table.

```
ALTER TABLE Internaute ADD adresse_pro TEXT,
    ADD code_postal_pro VARCHAR(10), ADD ville_pro VARCHAR(100);
```

On indique alors, *dans le composé* (l'entité), le *mapping* entre la table de l'entité et le nouveau composant, autrement dit les colonnes de la table qui doivent stocker les propriétés du composant. C'est une *surcharge* (*override* en anglais), d'où la syntaxe suivante en JPA, à ajouter à la classe `Internaute`.

```
   @Embedded
   @AttributeOverrides( {
   @AttributeOverride(name="adresse", column = @Column(name="adresse_pro") ),
   @AttributeOverride(name="codePostal", column = @Column(name="code_postal_pro") ),
   @AttributeOverride(name="ville", column = @Column(name="ville_pro") )
}
   )
   private Adresse adressePro;
   public void setAdressePro(Adresse a) {adressePro = a;}
   public Adresse getAdressePro() {return adressePro;}
```

L'attribut `codePostal` du composant `adressePro` sera donc par exemple stocké dans la colonne `code_postal_pro`.

Exercice : affichez les adresses personnelle et professionelle.

Etendez l'action précédente pour affichez les deux adresses. Au préalable, utilisez phpMyAdmin pour saisir quelques valeurs d'adresse professionnelle.

Exercice : mappez toutes les entités de la base *webscope*

Définissez une classe pour chaque *entité* de la base *webscope*, avec un *mapping* utlisant les annotations vues jusqu'à présent. Attention : nous parlons bien des *entités* (cf. le modèle UML) et pas des associations qui parfois, sont aussi représentées par des tables.

8.2 S2 : associations un-à-plusieurs

Supports complémentaires :

- Diapos pour la session « S2 : associations un-à-plusieurs »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f3595092c8o8in5x/

Venons-en maintenant aux associations. Nous laissons de côté les associations « un à un » qui sont peu fréquentes et nous en venons directement aux associations « un à plusieurs ». Notre exemple prototypique dans notre base de données est la relation entre un film et son (unique) réalisateur. Un premier constat, très important : en java nous pouvons représenter l'association de trois manières, illustrées sur la figure *Trois possibilités pour une association un à plusieurs*.

- dans un film, on place un lien vers le réalisateur (unidirectionnel, à gauche) ;
- dans un artiste, on place des liens vers les films qu'il a réalisés (unidirectionnel, centre) ;
- on représente les liens des deux côtés (bidirectionnel, droite).

Rappelons que dans une base relationnelle, le problème ne se pose pas : une association représentée par une clé étrangère est par nature bidirectionnelle. Cette différence est due aux deux paradigmes opposés sur lesquels s'appuient le modèle relationnel d'une part (qui considère des *ensembles* sans liens explicites entre eux, une association étant reconstitué

par un calcul de jointure) et le modèle objet de java (qui établit une navigation dans les objets instantiés grâce aux références d'objets).

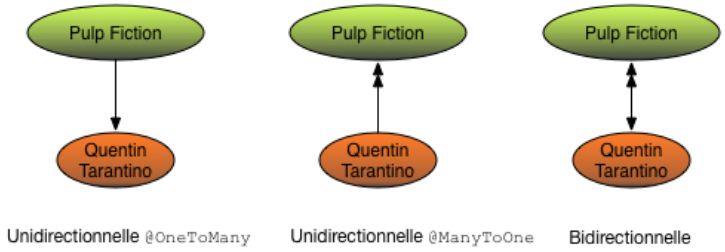

FIG. 1 – Trois possibilités pour une association un à plusieurs

Ceci étant posé, voyons comment nous pouvons représenter les trois situations, pour un même état de la base de données.

8.2.1 L'annotation `@ManyToOne`

Nous avons déjà étudié ce cas de figure. Dans la classe `Film`, nous indiquons qu'un objet lié nommé `réalisateur`, de la classe `Artiste`, doit être cherché dans la base par Hibernate.

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
public void setRealisateur(Artiste a) {realisateur = a;}
public Artiste getRealisateur() {return realisateur;}
```

L'annotation `@JoinColumn` indique quelle est la *clé étrangère* dans `Film` qui permet de rechercher l'artiste concerné. En d'autres termes, quand on appelle `getRealisateur()`, Hibernate doit engendrer et exécuter la requête suivante ;

```
select * from Artiste where id = ?film.id_realisateur
```

où `?film` désigne l'objet-film courant.

Important : Vous pouvez noter qu'avec le comportement simpliste décrit ci-dessus, on effectue une requête SQL pour rechercher *un* objet, ce qui constitue une utilisation totalement sous-optimale de SQL et risque d'engendrer de très gros problèmes de performance pour des bases importantes. Gardez cette remarque sous le coude, nous y reviendrons plus tard.

Hibernate sait bien entendu représenter l'association (clé primaire / clé étrangère) dans la base, en transposant la référence objet d'un artiste par un film. Voici concrètement comment cela se passe. Créons l'action `insertFilm` ci-dessous.

```
public void insertFilm() {
        session.beginTransaction();
        Film gravity = new Film();
        gravity.setTitre("Gravity");
        gravity.setAnnee(2013);

        Genre genre = new Genre();
        genre.setCode("Science-fiction");
        gravity.setGenre(genre);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        Artiste cuaron = new Artiste();
        cuaron.setPrenom("Alfonso");
        cuaron.setNom("Cuaron");

        // Le réalisateur de Gravity est Alfonso Cuaron
        gravity.setRealisateur(cuaron);

        // Sauvegardons dans la base
        session.save(gravity);
        session.save(cuaron);
        session.getTransaction().commit();
}
```

On a donc créé le film *Gravity*, de genre *Science-Fiction* et mis en scène par Alfonso Cuaron. L'association est créée, *au niveau du graphe d'objets java* par l'instruction :

```
gravity.setRealisateur(cuaron);
```

On a ensuite sauvegardé les deux nouvelles entités dans la base. Exécutons cette action, et regardons ce qui s'affiche dans la console :

```
Hibernate: select genre_.code from Genre genre_ where genre_.code=?
Hibernate: insert into Film (annee, genre, code_pays,
            d_realisateur, resume, titre) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Artiste (annee_naissance, nom, prenom) values (?, ?, ?)
Hibernate: update Film set annee=?, genre=?, code_pays=?, id_realisateur=?,
            resume=?, titre=? where id=?
```

Ce que l'on voit : Hibernate a engendré et exécuté des requêtes SQL. La première sélectionne le genre *Science-Fiction* : comme nous avons indiqué la valeur de la clé primaire dans l'instance Java, Hibernate vérifie automatiquement si cet objet `Genre` existe dans la base et va le lire si c'est le cas opur le lier par référence à l'objet `Film` en cours de création. Hibernate cherche toujours à synchroniser le graphe des objets et la base de données quand c'est nécessaire.

Le premier `insert` est déclenché par l'instruction `session.save(gravity)`. Il insère le film. À ce stade l'identifiant du metteur en scène dans la base est à `NULL` puisque Cuaron n'a pas encore été sauvegardé. Nous pouvons donc, à un moment donné, associer un objet persistant (le film ici) à un objet transient (l'artiste).

Le second `insert` correspond à `session.save(cuaron)`. Il insère l'artiste, qui obtient alors un identifiant dans la base de données. Et du coup Hibernate effectue un `update` sur le film pour affecter cet identifiant à la colonne `id_realisateur`.

Note : On remarque que tous les champs de `Film` sont modifiés par l'"`update` alors que seul l'identifiant du metteur en scène a changé. Hibernate ne gère pas les modifications au niveau des attributs mais se contente - c'est déjà beaucoup - de détecter toute modification de l'objet java pour déclencher une synchronisation avec la base. Nous reviendrons sur tout cela quand nous parlerons des transactions.

En résumé, tout va bien. Pour un effort minime nous obtenons une gestion complète de l'association entre un film et son metteur en scène. Hibernate synchronise les actions effectuées sur le graphe des objets persistants en java avec la base de données, que ce soit en lecture ou en écriture.

C'est aussi le bon endroit pour noter concrètement la différence, en ce qui concerne les associations, entre la représentation relationnelle et le modèle de données en java. Il n'est pour l'instant pas possible de récupérer la liste des films

réalisés par Alfonson Cuaron car nous n'avons pas représenté dans le modèle objet (Java) ce côté de l'association. En revanche, une simple jointure en SQL nous donnerait cette information.

8.2.2 L'annotation `@OneToMany`

De l'autre côté de l'association, on utilise l'annotation `@OneToMany` : à un objet correspondent plusieurs objets associés. Dans notre exemple, à un artiste correspondent de 0 à plusieurs films réalisés.

Important : Mettez en commentaires, pour l'instant, la clause `@ManyToOne` définissant le réalisateur dans la classe `Film`, puisque nous étudions les associations unidirectionnelles :

```
/* @ManyToOne
  @JoinColumn (name="id_realisateur")
  private Artiste realisateur;
  public void setRealisateur(Artiste a) {realisateur = a;}
  public Artiste getRealisateur() {return realisateur;}
*/
```

Nous allons voir dans la prochaine session ce qui se passe quand on combine les deux.

Spécifions l'association `@OneToMany` dans la classe `Artiste`.

```
@OneToMany
@JoinColumn(name="id_realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
public void addFilmsRealise(Film f) {filmsRealises.add(f) ;}
public Set<Film> getFilmsRealises() {return filmsRealises;}
```

Il n'y a pratiquement pas de différence avec la représentation `@ManyToOne`. On indique, comme précédemment, que la clé étrangère est `id_realisateur` et Hibernate comprend qu'il s'agit d'un attribut de `Film`, ce qui lui permet d'engendrer la même requête SQL que précédemment.

L'association est représentée par une collection, ici un `Set` (notez qu'on l'initialise avec un ensemble vide). On fournit deux méthodes, `add` (au lieu de `set` pour le côté `@ManyToOne`) et `get`.

Tentons à nouveau l'insertion d'un nouveau film et de son metteur en scène. Il suffit d'exécuter la même méthode `insertFilm` que précédemment, en remplaçant la ligne :

```
gravity.setRealisateur(cuaron);
```

par la ligne :

```
cuaron.addFilmsRealise(gravity);
```

Supprimez éventuellement les données insérées précédemment, dans phpMyAdmin, avec :

```
DELETE FROM Film where titre='Gravity';
DELETE FROM Artiste WHERE nom='Cuaron'
```

et exécutez cette action. Hibernate devrait afficher les requêtes suivantes dans la console :

```
Hibernate: select genre_.code from Genre genre_ where genre_.code=?
Hibernate: insert into Film (annee, genre, code_pays, id_realisateur,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            resume, titre) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Artiste (annee_naissance, nom, prenom) values (?, ?, ?)
Hibernate: update Film set id_realisateur=? where id=?
```

Hibernate a effectué les insertions et les mises à jour identiques à celles déjà vues pour l'association `@ManyToOne`. Là aussi, c'est normal puisque nous avons changé l'association en java, mais au niveau de la base elle reste représentée de la même manière.

Cette fois il n'est plus possible de connaître le metteur en scène d'un film puisque nous avons supprimé un des côtés de l'association. Il va vraiment falloir représenter une association bidirectionnelle. C'est ce que nous étudions dans la prochaine session. Faites au préalable l'exercice ci-dessous.

Exercice : afficher la liste des artistes avec les films qu'ils ont réalisés

Ecrire une action qui affiche tous les artistes et la liste des films qu'ils ont réalisés, quand c'est le cas. Aide : on devrait donc trouver deux boucles imbriquées dans la JSTL, une sur les artistes, l'autre sur les films.

8.3 S3 : Associations bidirectionnelles

Supports complémentaires :

- Diapos pour la session « S3 : Associations bidirectionnelles »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950b5fhlm5fi0/

Maintenant, que se passe-t-il si on veut représenter l'association de manière bi-directionnelle, ce qui sera souhaité dans une bonne partie des cas. C'est tout à fait pertinent par exemple pour notre exemple, puisqu'on peut très bien vouloir naviguer dans l'association `réalise` à partir d'un metteur en scène ou d'un film.

On peut tout à fait représenter l'association des deux côtés, en plaçant donc simultanément dans la classe `Film` la spécification :

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
public void setRealisateur(Artiste a) {realisateur = a;}
public Artiste getRealisateur() {return realisateur;}
```

et dans la classe `Artiste` la spécification :

```
@OneToMany
@JoinColumn(name="id_realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
public void addFilmsRealise(Film f) {filmsRealises.add(f) ;}
public Set<Film> getFilmsRealises() {return filmsRealises;}
```

Cette fois l'association est représentée des deux côtés.

8.3.1 Le problème

Cela soulève cependant un problème dû au fait que là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel. Effectuons une dernière modification de `insertFilm()` en affectant les *deux* côtés de l'assocation.

```
// Le réalisateur de Gravity est Alfonso Cuaron
gravity.setRealisateur(cuaron);
// Films réalisés par A. Curaon?
for (Film f : cuaron.getFilmsRealises()) {
        System.out.println("Curaron a réalisé " + f.getTitre());
}
// Alfonso Cuaron a réalisé Gravity
cuaron.addFilmsRealise(gravity);
```

Au passage, on a aussi affiché la liste des films réalisés par Alfonso Caron. Première remarque : le code est alourdi par la nécessité d'appeler `setRealisateur()` *et* `addFilmRealises()` pour attacher les deux bouts de l'association aux objets correspondants. Exécutez à nouveau `insertFilm()` (après avoir supprimé à nouveau de la base l'artiste et le film) et regardons ce qui se passe. Hibernate affiche :

```
Hibernate: select genre_.code from Genre genre_ where genre_.code=?
Hibernate: insert into Film (annee, genre, code_pays,
              id_realisateur, resume, titre) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Artiste (annee_naissance, nom, prenom) values (?, ?, ?)
Hibernate: update Film set annee=?, genre=?, code_pays=?,
              id_realisateur=?, resume=?, titre=? where id=?
Hibernate: update Film set id_realisateur=? where id=?
```

Avez-vous d'abord noté ce que *l'on ne voit pas* ? Normalement notre code devrait afficher la liste des films réalisés par Alfonso Cuaron. Rien ne s'affiche, car nous avons demandé cet affichage *après* avoir dit que Cuaron est le réalisateur de *Gravity*, mais *avant* d'avoir dit que *Gravity* fait partie des films réalisés par Cuaron. En d'autres termes, java *ne sait pas*, quand on renseigne un côté de l'association, *déduire* l'autre côté. Cela semble logique quand on y réfléchit, mais ça va mieux en le disant et en le constatant. Il faut donc bien appeler `setRealisateur()` *et* `addFilmRealises()` pour que notre graphe d'objets java soit cohérent. Si on ne le fait, l'incohérence du graphe est potentiellement une source d'erreur pour l'application.

Seconde remarque : Hibernate effectue deux `update`, alors qu'un seul suffirait. Là encore, c'est parce que les deux spécifications de chaque bout de l'association sont indépendantes l'une de l'autre.

Ouf. Prenez le temps de bien réfléchir, car nous sommes en train d'analyser une des principales difficultés conceptuelles du *mapping* objet-relationnel.

(réflexion intense de votre part)

8.3.2 Le remède

Bien, pour résumer nous avons deux objets java qui sont déclarés comme persistants (le fim, l'artiste). Hibernate surveille ces objets et déclenche une mise à jour dans la base dès que leur état est modifié. Comme la réalisation d'une association implique la modification des *deux* objets, alors que la base relationnelle représente l'association en un seul endroit (la clé étrangère), on obtient des mises à jour redondantes.

Est-ce *vraiment* grave ? A priori on pourrait vivre avec, le principal inconvénient prévisible étant un surplus de requêtes transmises à la base, ce qui peut éventuellement pénaliser les performances. Ce n'est pas non plus très satisfaisant, et Hibernate propose une solution : déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit.

```
@OneToMany(mappedBy="realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
```

On indique donc que le *mapping* de l'association est pris en charge par la classe `Film`, *et on supprime l'annotation* `@JoinColumn` *puisqu'elle devient inutile*. Ce que dit `mappedBy` en l'occurrence, c'est qu'un objet associé de la classe `Film` maintient un lien avec une instance de la classe courante (un `Artiste`) grâce à une propriété nommée `realisateur`. Hibernate, en allant inspecter la classe `Film`, trouvera que `realisateur` est *mappé* avec la base de données par le biais de la clé étrangère `id_realisateur`. Dans `Film`, on trouve en effet :

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
```

Toutes les informations nécessaires sont donc disponibles, et représentées une seule fois. Cela semble compliqué ? Oui, ça l'est quelque peu, sans doute, mais encore une fois il s'agit de la partie la plus contournée du *mapping* ORM. Prenez l'exemple que nous sommes en train de développer comme référence, et tout ira bien.

En conséquence, une modification de l'état de l'association au niveau d'un artiste *ne déclenchera pas* d'`update` dans la base de données. En d'autres termes, l'instruction

```
// Alfonso Cuaron a réalisé Gravity
cuaron.addFilmsRealise(gravity);
```

ne sera pas synchronisée dans la base, et rendue persistante. On a sauvé un `update`, mais introduit une source d'incohérence. Une petite modification de la méthode `addFilmsRealises()` suffit à prévenir le problème.

```
public void addFilmsRealise(Film f) {
        f.setRealisateur(this);
        filmsRealises.add(f) ;
}
```

L'astuce consiste à appeler `setRealisateur()` pour garantir la mise à jour dans la base dans tous les cas.

8.3.3 En résumé

Il est sans doute utile de résumer ce qui précède. Voici donc la méthode recommandée pour une association *un à plusieurs*. Nous prenons comme guide l'association entre les films et leurs réalisateurs. `Film` est du côté `plusieurs` (un réalisateur met en scène *plusieurs* films), `Artiste` du côté `un` (un film a *un* metteur en scène). Dans la base relationnelle, c'est du côté *plusieurs* que l'on trouve la clé étrangère.

Du côté « plusieurs »

On donne la spécification du *mapping* avec la base de données. Les annotations sont `@ManyToOne` et `@JoinColumn`. Par exemple.

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
```

Dans le jargon ORM, ce côté est « responsable » de la gestion du *mapping*.

Du côté « un »

On indique avec quelle *classe responsable* on est associé, et quelle *propriété* dans cette classe représente l'association. Aucune référence directe à la base de données n'est faite. L'annotation est `@OneToMany` avec l'option `mappedBy` qui

indique que la responsabilité de l'association est déléguée à la classe référencée, dans laquelle l'association elle-même est représentée par la valeur de l'attribut `mappedBy`. Par exemple :

```
@OneToMany(mappedBy="realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
```

Hibernate inspectera la propriété `realisateur` dans la classe responsable pour déterminer les informations de jointure si nécessaire.

Les accesseurs

Du côté `plusieurs`, les accesseurs sont standards. Pour la classe `Film`.

```
public void setRealisateur(Artiste a) {realisateur = a;}
public Artiste getRealisateur() {return realisateur;}
```

et pour la classe `Artiste`, on prend soin d'appeler l'accesseur de la classe responsable pour garantir la synchronisation avec la base et la cohérence du graphe.

```
public void addFilmsRealise(Film f) {
        f.setRealisateur(this);
        filmsRealises.add(f) ;
}
public Set<Film> getFilmsRealises() {return filmsRealises;}
```

Et voilà ! Dans ces conditions, voici le code pour créer une association.

```
public void insertFilm() {
        session.beginTransaction();
        Film gravity = new Film();
        gravity.setTitre("Gravity");
        gravity.setAnnee(2013);

        Genre genre = new Genre();
        genre.setCode("Science-fiction");
        gravity.setGenre(genre);

        Artiste cuaron = new Artiste();
        cuaron.setPrenom("Alfonso");
        cuaron.setNom("Cuaron");

        // Alfonso Cuaron a réalisé Gravity
        cuaron.addFilmsRealise(gravity);

        // Sauvegardons dans la base
        session.save(gravity);
        session.save(cuaron);
        session.getTransaction().commit();
}
```

L'association est réalisée des deux côtés en java par un seul appel à la méthode `addFilmsRealises()`. Notez également qu'il reste nécessaire de sauvegarder individuellement le film et l'artiste. Nous pouvons gérer cela avec des annotations `Cascade`, mais vous avez probablement assez de concepts à avaler pour l'instant.

Lisez et relisez ce qui précède. Pas de panique, il suffit de reproduire la même construction à chaque fois. Bien entendu c'est plus facile si l'on a *compris* le pourquoi et le comment.

Exercice : vérifier qu'Hibernate ne génére plus de requête redondante.

Modifiez `insertFilm()` comme indiqué, et consultez les requêtes Hibernate pour vérifier qu'un seul `update` est effectué.

Que se passe-t-il si l'application appelle directement `setRealisateur()` et pas `addFilmsRealise()` ? Que faudrait-il faire ?

8.4 S4 : Associations plusieurs-à-plusieurs

Supports complémentaires :

— Diapos pour la session « S4 : Associations plusieurs-à-plusieurs »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950bdesbxcgui/

Nous en venons maintenant aux associations de type « plusieurs-à-plusieurs », dont deux représentants apparaissent dans le modèle de notre base *webscope* :

— un film a *plusieurs* acteurs ; un acteur joue dans *plusieurs* films ;
— un internaute note *plusieurs* films ; un film est noté par *plusieurs* internautes.

Quand on représente une association plusieurs-plusieurs en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association. C'est bien ce qui a été fait pour cette base : une table `Role` représente la première association avec une clé composée `(id_film, id_acteur)` et une table `Notation` représente la seconde avec une clé `(id_film, email)`. Et il faut noter de plus que chaque partie de la clé est elle-même une clé étrangère.

On pourrait généraliser la règle à des associations ternaires, et plus. Les clés deviendraient très compliquées et le tout deviendrait vraiment difficile à interpréter. La recommandation dans ce cas est de *promouvoir* l'association en entité, en lui donnant un identifiant qui lui est propre.

Note : Même pour les associations binaires, il peut être justicieux de transformer toutes les associations plusieurs-plusieurs en entité, avec deux associations un-plusieurs vers les entités originales. On aurait donc une entité `Role` et une entité `Notation` (chacune avec un identifiant). Cela simplifierait le *mapping* JPA qui, comme on va le voir, est un peu compliqué.

On trouve de nombreuses bases (dont la nôtre) incluant la représentation d'associations plusieurs-plusieurs avec clé composée. Il faut donc savoir les gérer. Deux cas se présentent en fait :

— l'association n'est porteuse d'aucun attribut, et c'est simple ;
— l'association porte des attributs, et ça se complique un peu.

Nos associations sont porteuses d'attributs, mais nous allons quand même traiter les deux cas par souci de complétude.

8.4.1 Premier cas : association sans attribut

Dans notre schéma, le nom du rôle d'un acteur dans un film est représenté dans l'association (un peu de réflexion suffit à se convaincre qu'il ne peut pas être ailleurs). Supposons pour l'instant que nous décidions de sacrifier cette information, ce qui nous ramène au cas d'une association sans attribut. La modélisation JPA/Hibernate est alors presque semblable à celle des associations un-plusieurs. Voici ce que l'on représente dans la classe `Film`.

```
@ManyToMany()
@JoinTable(name = "Role", joinColumns = @JoinColumn(name = "id_film"),
           inverseJoinColumns = @JoinColumn(name = "id_acteur"))
Set<Artiste> acteurs = new HashSet<Artiste>();
```

(suite sur la page suivante)

(suite de la page précédente)

```
public Set<Artiste> getActeurs() {
        return acteurs;
}
```

C'est ce côté qui est « responsable » du *mapping* avec la base de données. On indique donc une association `@ManyToMany`, avec une seconde annotation `JoinTable` qui décrit tout simplement la table représentant l'association, `joinColumn` étant la clé étrangère pour la classe courante (`Film`) et `inverseJoinColumn` la clé étrangère vers la table représentant l'autre entité, `Artiste`. Essayez de produire la requête SQL qui reconstitue l'association et vous verrez que toutes les informations nécessaires sont présentes.

De l'autre côté, c'est plus simple encore puisqu'on ne répète pas la description du *mapping*. On indique avec `mappedBy` qu'elle peut être trouvée dans la classe `Film`. Ce qui donne, pour la filmographie d'un artiste :

```
@ManyToMany(mappedBy = "acteurs")
Set<Film> filmo;
public Set<Film> getFilmo() {
        return filmo;
}
```

C'est tout, il est maintenant possible de naviguer d'un film vers ses acteurs et réciproquement. L'interprétation de `mappedBy` est la même que dans le cas des associations un-plusieurs.

Exercice : afficher tous les acteurs d'un film.

Reprenez l'action qui affiche les films, et ajoutez la liste des acteurs. De même, pour la liste des artistes, affichez les films dans lesquels ils ont joué.

8.4.2 Second cas : association avec attribut

Maintenant, nous reprenons notre base, et nous aimerions bien accéder au nom du rôle, ce qui nécessite un accès à la table `Role` representant l'association.

Nous allons représenter dans notre modèle JPA cette association comme une `@Entity`, avec des associations un à plusieurs vers, respectivement, `Film` et `Artiste`, et le tour est joué.

Oui, mais… nous voici confrontés à un problème que nous n'avons pas encore rencontré : la clé de `Role` est composée de deux attributs. Il va donc falloir tout d'abord apprendre à gérer ce cas.

Nous avons fait déjà un premier pas dans ce sens en étudiant les *composants* (vous vous souvenez, les adresses ?). Une clé en Hibernate se représente par un objet. Quand cet objet est une valeur d'une classe de base (`Integer`, `String`), tout va bien. Sinon, il faut définir cette classe avec les attributs constituant la clé. En introduisant un nouveau niveau de granularité dans la représentation d'une entité, on obtient bien ce que nous avons appelé précédemment un composant.

La figure *Gestion d'un identifiant composé* montre ce que nous allons créer. La classe `Role` est une entité composée d'une clé et d'un attribut, le nom du rôle. La clé est une instance d'une classe `RoleId` constituée de l'identifiant du film et de l'identifiant de l'acteur.

Voici la méthode illustrée par la pratique. Tout d'abord nous créons cette classe représentant la clé.

```
package modeles.webscope;

import javax.persistence.*;

```

(suite sur la page suivante)

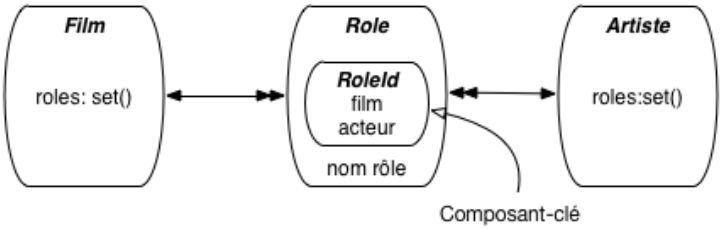

FIG. 2 – Gestion d'un identifiant composé

(suite de la page précédente)

```
@Embeddable
public class RoleId implements java.io.Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @ManyToOne
    @JoinColumn(name = "id_acteur")
    private Artiste acteur;
    public Artiste getActeur() {
            return acteur;
    }

    public void setActeur(Artiste a) {
            this.acteur = a;
    }

    @ManyToOne
    @JoinColumn(name = "id_film")
    private Film film;
    public Film getFilm() {
            return film;
    }

    public void setFilm(Film f) {
            this.film = f;
    }
}
```

Notez que ce n'est pas une entité, mais un composant annoté avec `@Embeddable` (reportez-vous à la section sur les composants si vous avez un doute). Pour le reste on indique les associations `@ManyToOne` de manière standard.

Important : une classe dont les instances vont servir d'identifiants doit hériter de `serializable`, la raison - technique - étant que des structures de *cache* dans Hibernate doivent être sérialisées dans la session, et que ces structures indexent les objets par la clé.

Voici maintenant l'entité `Role`.

```
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Role {

     @Id
     RoleId pk;
     public RoleId getPk() {
             return pk;
     }

     public void setPk(RoleId pk) {
             this.pk = pk;
     }

     @Column(name="nom_role")
     private String nom;
     public void setNom(String n) {nom= n;}
     public String getNom() {return nom;}
}
```

On indique donc que l'identifiant est une instance de `RoleId`. On la nomme `pk` pour *primary key*, mais le nom n'a pas d'importance. La classe comprend de plus les attributs de l'association.

Et cela suffit. On peut donc maintenant accéder à une instance `r` de rôle, afficher son nom avec `r.nom` et même accéder au film et à l'acteur avec, respectivement, `r.pk.film` et `r.pk.acteur`. Comme cette dernière notation peut paraître étrange, on peut ajouter le code suivant dans la classe `Role` qui implante un racourci vers le film et l'artiste.

```
public Film getFilm() {
        return getPk().getFilm();
}

public void setFilm(Film film) {
        getPk().setFilm(film);
}

public Artiste getActeur() {
        return getPk().getActeur();
}

public void setActeur(Artiste acteur) {
        getPk().setActeur(acteur);
}
```

Maintenant, si `r` est une instance de `Role`, `r.film` et `r.acteur` désignent respectivement le film et l'acteur.

Il reste à regarder comment on code l'autre côté de l'association. Pour le film, cela donne :

```
@OneToMany(mappedBy = "pk.film")
private Set<Role> roles = new HashSet<Role>();
public Set<Role> getRoles() {
        return this.roles;
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
public void setRoles(Set<Role> r) {
        this.roles = r;
}
```

Seule subtilité : le `mappedBy` indique un *chemin* qui part de `Role`, passe par la propriété `pk` de `Role`, et arrive à la propriété `film` de `Role.pk`. C'est là que l'on va trouver la définition du *mapping* avec la base, autrement le nom de la clé étrangère qui référence un film. Même chose du côté des artistes :

```
@OneToMany(mappedBy = "pk.acteur")
private Set<Role> roles = new HashSet<Role>();
public Set<Role> getRoles() {
        return this.roles;
}
public void setRoles(Set<Role> r) {
        this.roles = r;
}
```

Exercice : affichez tous les rôles

Créez une action qui affiche tous les rôles, avec leur film et leur artiste.

Nous avons fait un bon bout de chemin sur la compréhension du *mapping* JPA. Nous arrêtons là pour l'instant afin de bien digérer tout cela. Voici un exercice qui vous permettra de finir la mise en pratique.

Exercice : compléter le *mapping* de la base *webscope*

Vous en savez maintenant assez pour compléter la description du *mapping* O/R de la base *webscope*. Une fois cela accompli, vous devriez pouvoir créer une action qui affiche tous les films avec leur metteur en scène, leurs acteurs et les rôles qu'ils ont joué. Vous devriez aussi pouvoir afficher les internautes, les notes qu'ils ont données et à quels films.

8.5 Résumé : savoir et retenir

À l'issue de ce chapitre,, assez dense, vous devez être en mesure de définir le *mapping* ORM pour la plupart des bases de données, une exception (rare) étant la représentation de *l'héritage*, pour lequel les bases relationnelles ne sont pas adaptées, mais que l'on va trouver potentiellement dans la modélisation objet de l'application. Il faut alors savoir produire une structure de base de données adaptée, et la *mapper* vers les classes de l'application. C'est ce que nous verrons dans le prochain chapitre.

Il resterait encore des choses importantes à couvrir pour être complets sur le sujet. Entre autres :

- les méthodes `equals()` et `hashCode()` sont importantes pour certaines classes ; il faudrait les implanter ;
- des options d'annotation, dont celles qui indiquent comment *propager* la persistence en *cascade*, méritent d'être présentées.

Concentrez-vous pour l'instant sur les connaissances acquise dans le présent chapitre, et gardez en mémoire que vous n'êtes pas encore complètement outillés pour faire face à des situations relativement marginales. Pour consolider vos acquis, il serait sans doute judicieux de prendre un schéma de base de données que vous connaissez et de produire le mapping adapté.

CHAPITRE 9

Règles avancées de *mapping*

Nous abordons dans ce chapitre quelques aspects avancés des règles de *mapping*, et notamment la gestion de *l'héritage* orienté-objet. On est ici au point le plus divergent des représentations relationnelle et OO, puisque l'héritage n'existe pas dans la première, alors qu'il est au cœur des modélisations avancés dans la seconde.

Il n'est pas très fréquent d'avoir à gérer une situation impliquant de l'héritage dans une base de données. Un cas sans doute plus courant est celui où on effectue la modélisation objet d'une application dont on souhaite rendre les données persistantes. Dans tous les cas les indications qui suivent vous montrent comme réaliser simplement avec JPA la persistance d'une hiérarchie de classes java.

9.1 S1 : Gestion de l'héritage

Supports complémentaires :

- Diapos pour la session « S1 : Gestion de l'héritage »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950c87i8ctjim/

Nous prenons comme exemple illustratif le cas très simple d'un raffinement de notre modèle de données. La notion plus générale de *vidéo* est introduite, et un film devient un cas particulier de vidéo. Un autre cas particulier est le *reportage*, ce qui donne donc le modèle de la figure *Notre exemple d'héritage*. Au niveau de la super-classe, on trouve le titre et l'année. Un film se distingue par l'association à des acteurs et un metteur en scène ; un reportage en revanche a un lieu de tournage et une date.

Le cas est assez simplifié, mais suffisant pour étudier nos règles de *mapping*.

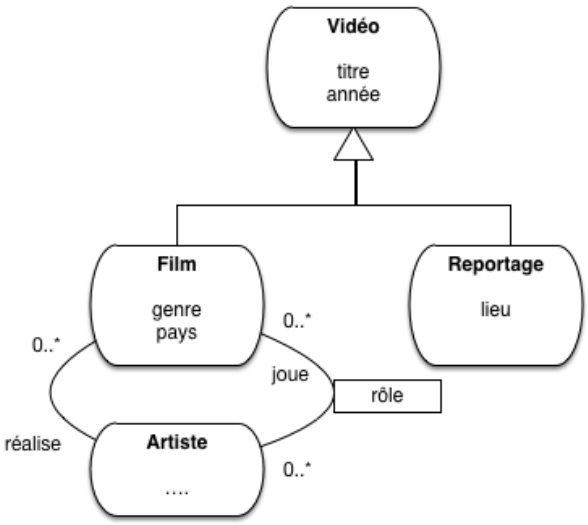

FIG. 1 – Notre exemple d'héritage

9.1.1 Les solutions possibles

Comme indiqué ci-dessus, il n'existe pas dans le modèle relationnel de notion d'héritage (d'ailleurs la notion d'objet en général est inconnue). Il faut donc trouver un contournement. Voici les trois solutions possibles, aucune n'est idéale et vous trouverez toujours quelqu'un pour argumenter en faveur de l'une ou l'autre. Le mieux est de vous faire votre propre opinion (je vous donne la mienne un peu plus loin).

— *Une table pour chaque classe*. C'est la solution la plus directe, menant pour notre exemple à créer des tables *Vidéo*, *Film* et *Reportage*. Remarque très importante : on doit *dupliquer* dans la table d'une sous-classe les attributs persistants de la super-classe. Le titre et l'année doivent donc être dupliqués dans, respectivement, *Film* et *Reportage*. Cela donne des tables indépendantes, chaque objet étant complètement représenté par une seule ligne.
Remarque annexe : si on considère que *Vidéo* est une classe abstraite qui ne peut être instanciée directement, on ne crée pas de table *Vidéo*.

— *Une seule table pour toute la hiérarchie d'héritage*. On créerait donc une table *Vidéo*, et on y placerait tous les attributs persistants de *toutes* les sous-classes. La table *Vidéo* aurait donc un attribut *id_realisateur* (venant de *Film*), et un attribut *lieu* (venant de *Reportage*).
Les instances de *Vidéo*, *Film* et *Reportage* sont dans ce cas toutes stockées dans la même table *Vidéo*, ce qui nécessite l'ajout d'un attribut, dit *discriminateur*, pour savoir à quelle classe précise correspondent les données stockées dans une ligne de la table. L'inconvénient évident, surtout en cas de hiérarchie complexe, est d'obtenir une table fourre-tout contenant des données difficilement compréhensibles.

— Enfin, la troisième solution est un mixte des deux précédentes, consistant à créer une table par classe (donc, trois tables pour notre exemple), tout en gardant la spécialisation propre au modèle d'héritage : chaque table ne contient que les attributs venant de la classe à laquelle elle correspond, et une *jointure* permet de reconstituer l'information complète.
Par exemple : un film serait représenté partiellement (pour le titre et l'année) dans la table *Vidéo*, et partiellement (pour les données qui lui sont spécifiques, comme *id_realisateur*) dans la table *Film*.

Aucune solution n'est totalement satisfaisante, pour les raisons indiquées ci-dessus. Voici une petite discussion donnant mon avis personnel.

La duplication introduite par la première solution semble source de problèmes à terme, et je ne la recommande vraiment pas. Tout changement dans la super-classe devrait être répliqué dans toutes les sous-classes, ce qui donne un schéma douteux et peu contrôlable.

Tout placer dans une même table se défend, et présente l'avantage de meilleures performances puisqu'il n'y a pas de

jointure à effectuer. On risque de se retrouver en revanche avec une table dont la structure est peu compréhensible.

Enfin la troisième solution (table reflétant exactement chaque classe de la hiérarchie, avec jointure(s) pour reconstituer l'information) est la plus séduisante intellectuellement (de mon point de vue). Il n'y a pas de redondance, et il est facile d'ajouter de nouvelles sous-classes. L'inconvénient principal est la nécessité d'effectuer autant de jointures qu'il existe de niveaux dans la hiérarchie des classes pour reconstituer un objet.

9.1.2 Application : une table pour chaque classe

Dans ce qui suit, nous allons illustrer cette dernière approche et je vous laisse expérimenter les autres si cela vous tente. Nous aurons donc les trois tables suivantes :

— `Video (id_video, titre, annee)`
— `Film (id_video, genre, pays, id_realisateur)`
— `Reportage(id_video, lieu)`

Nous avons nommé les identifiants `id_video` pour mettre en évidence une contrainte qui n'apparaît pas clairement dans ce schéma (mais qui est spécificiable en SQL) : *comme un même objet est représenté dans les lignes de plusieurs tables, son identifiant est une valeur de clé primaire commune à ces lignes*.

Un exemple étant plus parlant que de longs discours, voici comment nous représentons deux objets *vidéos*, dont l'un est un un film et l'autre un reportage.

Tableau 1 – La table `Vidéo`

id_video	titre	année
1	Gravity	2013
2	Messner, profession alpiniste	2014

Rien n'indique dans cette table est la catégorie particulière des objets représentés. C'est conforme à l'approche objet : selon le point de vue on peut très bien se contenter de voir les objets comme instances de la super-classe. De fait, *Gravity* et *Messner* sont toutes deux des vidéos.

Voici maintenant la table *Film*, contenant la partie de la description de *Gravity* spécifique à sa nature de film.

Tableau 2 – La table `Film`

id_video	genre	pays	id_realisateur
1	Science-fiction	USA	59

Notez que l'identifiant de *Gravity* (la valeur de `id_video`) est le *même* que pour la ligne contenant le titre et l'année dans *Vidéo*. C'est logique puisqu'il s'agit du même objet. Dans *Film*, `id_video` est *à la fois* la clé primaire, et une clé étrangère référençant une ligne de la table *Video*. On voit facilement quelle requête SQL permet de reconstituer l'ensemble des informations de l'objet.

```
select * from Video as v, FilmV as f
where v.id_video=f.id_video
and titre='Gravity'
```

C'est cette requête qui est engendrée par Hibernate quand l'objet doit être instancié. Dans le même esprit, voici la table *Reportage*.

Tableau 3 – La table `Reportage`

id_video	lieu
2	Tyroll du sud

En résumé, avec cette approche, l'information relative à un même objet est donc éparpillée entre différentes tables. Comme souligné ci-dessus, cela mène à une particularité originale : la clé primaire d'une table pour une sous-classe est *aussi* clé étrangère référençant une ligne dans la table représentant la super-classe. Voici les commandes de création des tables sous MySQL (nous nommons la seconde `FilmV` pour éviter la collision avec la table existante dans notre schéma).

```
 CREATE TABLE Video (id_video INT AUTO_INCREMENT,
         titre VARCHAR(255) NOT NULL,
         annee INT NOT NULL,
         PRIMARY KEY (id_video)
        );
CREATE TABLE FilmV (id_video INT,
         genre VARCHAR(40) NOT NULL,
         pays VARCHAR(40) NOT NULL,
         PRIMARY KEY (id_video),
         id_realisateur INT NOT NULL,
         FOREIGN KEY(id_realisateur) REFERENCES Artiste(id),
         FOREIGN KEY (id_video) REFERENCES Video(id_video)
        );
CREATE TABLE Reportage (id_video INT,
         lieu VARCHAR(40) NOT NULL,
         PRIMARY KEY (id_video),
         FOREIGN KEY (id_video) REFERENCES Video(id_video)
        );
```

Note : La clé primaire de la classe racine est en `AUTO_INCREMENT`, mais pas celles des autres classes. Un seul identifiant doit en effet être généré, pour la table-racine, et propagé aux autres.

9.1.3 Implantation JPA/Hibernate

Sur la base du schéma qui précède, il est très facile d'annoter les classes du modèle Java. Voici tout d'abord la super-classe.

```
package modeles.webscope;

import javax.persistence.*;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Video {

  @Id
  @GeneratedValue
  @Column(name = "id_video")
  private Long idVideo;

  public Long getIdVideo() {
            return this.idVideo;
      }

  @Column
```

(suite sur la page suivante)

(suite de la page précédente)

```
  private String titre;
      public void setTitre(String t) {
              titre = t;
      }
      public String getTitre() {
            return titre;
      }

  @Column
  private Integer annee;
      public void setAnnee(Integer a) {
            annee = a;
      }
      public Integer getAnnee() {
            return annee;
      }

  public Video() {  }
}
```

La nouvelle annotation est donc :

```
@Inheritance(strategy=InheritanceType.JOINED)
```

qui indique que la hiérarchie d'héritage dont cette classe est la racine est évaluée par *jointure*. Voici maintenant une des sous-classes, *FilmV* (réduite au minimum, à vous de compléter).

```
package modeles.webscope;

import javax.persistence.*;

@Entity
@PrimaryKeyJoinColumn(name="id_video")
public class FilmV extends Video {
      @ManyToOne
      @JoinColumn(name = "id_realisateur")
      private Artiste realisateur;
      public void setRealisateur(Artiste a) {
            realisateur = a;
      }

      public Artiste getRealisateur() {
            return realisateur;
      }
}
```

La seule annotation originiale de *FilmV* (dont on indique qu'elle hérite de *Video*) est :

```
@PrimaryKeyJoinColumn(name="id_video")
```

qui spécifie donc que `id_video` est à la fois clé primaire et clé étrangère pour la résolution de la jointure qui reconstitue un objet de la classe.

Exercice : afficher films et reportages

Implantez les classes *Video*, *Film* et *Reportage*, insérez quelques données dans la base (par exemple celles données dans les tableaux ci-dessus) et implantez une action qui affiche les films, les reportages, et finalement toutes les vidéos sans distinction de type.

Exercice (option) : appliquez les autres stratégies d'héritage

Reportez-vous à la documentation JPA/Hibernate pour tester les deux autres stratégies d'héritages. Consultez les requêtes SQL produites.

9.2 Résumé : savoir et retenir

Le *mapping* JPA/Hibernate des situations d'héritage objet est relativement simple, le seul problème relatif état le choix de la structure de la base de données. Retenez que plusieurs solutions sont possibles, aucune n'étant idéale. Celle développée dans ce chapitre semble la plus lisible.

Retenez la particularité de la représentation relationnelle dans ce cas : la clé primaire des tables, à l'exception de celle représentant la classe-racine, est *aussi* clé étrangère. Il est important de spécifier cette contrainte dans le schéma pour assurer la cohérence des données.

CHAPITRE 10

Lecture de données

Une fois résolus les problèmes d'association entre le modèle objet et le schéma relationnel, nous pouvons nous intéresser à l'accès aux données. Nous en avons déjà eu un rapide aperçu dans les chapitres précédents, mais il est temps maintenant d'adopter une approche plus systématique, de comprendre quels sont les mécanismes à l'œuvre, et de s'interroger sur les performances d'un accès à une base de données *via* un ORM.

Pratique : contrôleur, modèle et vue

Pour les exercices et exemples de ce chapitre, je vous propose de créer un nouveau contrôleur, nommé `Requeteur`, associé à l'URL `requeteur`. Pour les différentes méthodes de lecture étudiées, créez une classe `Lectures.java` qui tiendra lieu de modèle.

10.1 S1 : Comment fonctionne Hibernate

Supports complémentaires :

- Diapos pour la session « S1 : Comment fonctionne Hibernate »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950d193bfs1jp/

Nous avons défini le *mapping* ORM, sur lequel Hibernate s'appuie pour accéder aux données, en écriture et en lecture. Regardons maintenant comment sont gérés, en interne, ces accès. Pour l'instant nous allons nous contenter de considérer les *lectures* de données à partir d'une base existante, sans effectuer aucune mise à jour. Celà exclut donc la question des *transactions* qui, comme nous le verrons plus tard, est assez délicate. En revanche cela nous permet d'aborder sans trop de complication l'architecture d'Hibernate et le fonctionnement interne du *mapping* et de la matérialisation du graphe d'objet à partir de la base relationnelle.

Note : Le titre de ce chapitre utilise le terme *lecture*, et pas celui de *requête* plus habituel dans un contexte base de données. Cette distinction a pour but de souligner qu'une application ORM accède aux données (en lecture donc) par différents mécanismes, dont la *navigation* dans le graphe d'objet. Les requêtes effectuées sont souvent déterminées et exécutées par la couche ORM, sans directive explicite du programmeur. Gardez à l'esprit dans tout ce chapitre que notre application gère un graphe d'objet, pas une base de données tabulaires.

10.1.1 L'architecture

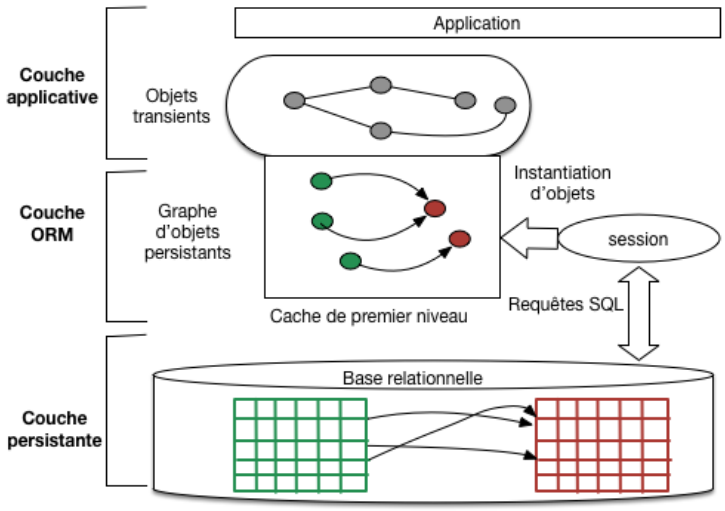

FIG. 1 – Les 3 couches d'une application ORM

Regardons à nouveau la structuration en couches d'une application s'appuyant sur un *framework* ORM (Figure *Les 3 couches d'une application ORM*). L'application, écrite en Java, manipule des objets que nous pouvons séparer en deux catégories :

— les objets *transients* sont des objets Java standard, instanciés par l'opérateur *new*, dont le cycle de vie est géré par le *garbage collector* ;
— les objets *persistants* sont également des objets java, instances d'une classe persistante (définie par l'annotation `@Entity`) et *images d'une ligne dans une table de la base relationnelle*.

L'appartenance à une classe persistante est une condition *nécessaire* pour qu'un objet devienne persistant, mais ce n'est pas une condition *suffisante*. Il faut également que l'objet soit placé sous le contrôle du composant chargé de la persistance, soit, pour Hibernate, l'objet *session*. Pour le dire autrement, on peut très bien instancier un objet d'une classe persistante et l'utiliser dans un cadre de programmation normal (dans ce cas c'est un objet transient), sans le stocker dans la base.

Note : Il existe en fait une troisième catégorie, les objets *détachés*, que nous présenterons dans la chapitre *Applications concurrentes*.

Les objets persistants sont placés dans un espace nommé *le cache de premier niveau* dans Hibernate, que l'on peut simplement voir comme l'emplacement où se trouve matérialisé (partiellement) le graphe des objets utilisés par l'application. Cette matérialisation est donc contrôlée et surveillée par un objet *session* que nous avons déjà utilisé pour accéder aux données, mais qu'il est maintenant nécessaire d'examiner en détail car son rôle est essentiel.

10.1.2 La session et le cache de premier niveau

La session Hibernate définit l'espace de communication entre l'application et la base de données. Essentiellement, cet objet a pour responsabilité de synchroniser la base de données et le graphe d'objet. Pour nous en tenir aux lectures, cette synchronisation consiste à transmettre des requêtes `SELECT` via JDBC, pour lire des lignes et instancier des objets à partir des valeurs de ces lignes.

Revenons à la figure *Les 3 couches d'une application ORM*. Deux tables (associées) sont illustrées, une verte, une rouge. Des lignes de chaque table sont représentées, sous forme d'objet persistant (rouge ou vert), dans le cache de premier niveau associé à la session. L'instantiation de ces objets a été déclenchée par des demandes de lecture de l'application. Ces demandes passent *toujours* par la session, soit *explicitement*, comme quand une requête HQL est exécutée, soit *implicitement*, quand par exemple lors d'une navigation dans le graphe d'objet.

La session est donc un objet absolument essentiel. Pour les lectures en particulier, son rôle est étroitement associé au cache de premier niveau. Elle assure en particulier le respect de la propriété suivante.

Propriété : unicité des références d'objet

Dans le contexte d'une session, chaque ligne d'une table est représentée par au plus un objet persistant.

En d'autres termes, une application, quelle que soit la manière dont elle accède à une ligne d'une table (requête, parcours de collection, navigation), obtiendra toujours la référence au même objet.

Cette propriété d'unicité est très importante. Imaginons le cas contraire : je fais par exemple plusieurs accès à un même film, et j'obtiens deux objets Java *distincts* (au sens de l'égalité des références, testée par ==), *A* et *B*. Alors :

- si je fais des modifications sur *A et* sur *B*, quelle est celle qui prend priorité au moment de la sauvegarde dans la base ?
- toute modification sur *A* ne serait pas immédiatement visible sur *B*, d'où des incohérence, sources de *bugs* très difficiles à comprendre.

Le cache est donc une sorte de copie de la base de données au plus près de l'application (dans la machine virtuelle java) et sous forme d'objets. Voici encore une autre manière d'exprimer les choses.

Corollaire : identité Java et identité BD

Dans le contexte d'une session, l'identité des objets est équivalente à l'identité base de données. Deux objets sont identiques (== renvoie `true`) si et seulement si ils ont les mêmes clés primaires.

Le test :

```
a == b
```

est donc toujours équivalent à :

```
A.getId().equals(B.getId())
```

Concrètement, cela implique un fonctionnement assez contraint pour la session : à chaque accès à la base ramenant un objet, il faut vérifier, *dans le cache de premier niveau*, si la ligne correspondante n'a pas déjà été instanciée, et si oui renvoyer l'objet déjà présent dans le cache. Pour reprendre l'exemple précédent :

- l'application fait une lecture (par exemple par une requête) sur la table *Film*, l'objet *A* correspondant à la ligne *l* est instancié, placé dans le cache de premier niveau, et sa référence est transmise à l'application ;
- l'application fait une seconde demande de lecture (par exemple en naviguant dans le graphe) ; cette lecture a pour paramètre la clé primaire de la ligne *l* : alors la session va d'abord chercher dans le cache de premier niveau si un objet correspondant à *l* existe ; si oui sa référence est renvoyée, si non une requête est effectuée ;
- l'application ré-exécute une requête sur la table *Film*, et parcourt les objets avec un itérateur ; alors à chaque itération il faut vérifier si la ligne obtenue est déjà instanciée dans le cache de premier niveau.

Vous devez être bien conscients de ce mécanisme pour comprendre comment fonctionne Hibernate, et le rôle combiné de la session et du cache de premier niveau.

Note : Notez bien la restriction « dans le contexte d'une session. ». Si vous fermez une session *s1* pour en ouvrir une autre *s2*, tout en gardant la référence vers *A*, la propriété n'est plus valable car *A* ne sera pas dans le cache de premier niveau de *s2*. À plus forte raison, deux applications distinctes, ayant chacune leur session, ne partageront pas leur cache de premier niveau.

Le cache de premier niveau est structuré de manière à répondre très rapidement au test suivant : « Donne moi l'objet dont la clé primaire est *C* ». La structure la plus efficace est une table de hachage. La figure *Une table de hachage du cache de premier niveau* montre la structure. Une fonction *H* prend en entrée une valeur de clé primaire *k, l, m* ou *n* (ainsi que le nom de la table *T*) et produit une *valeur de hachage* comprise entre h_1 et h_u. Un répertoire associe à chacune de ces *valeurs de hachage* une *entrée* comprenant un ou plusieurs objets java o_k, o_l, o_m ou o_n. Notez qu'il peut y avoir des *collisions* : deux clés distinctes mènent à une même entrée, contenant les objets correspondants.

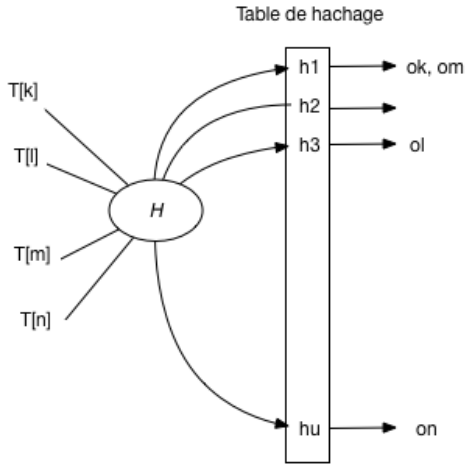

FIG. 2 – Une table de hachage du cache de premier niveau

En résumé, pour tout accès à une ligne de la base, c'est toujours le même objet qui est retourné à l'application. Le cache de premier niveau conserve tous les objets persistants, et c'est dans ce cache que l'application (ou plutôt la session courante) vient piocher. Ce cache *n'est pas* partagé avec une autre application (ou plus précisément avec une autre session).

Note : Comme vous vous en doutez peut-être, il existe un cache *de second niveau* qui, lui a des propriétés différentes. Laissons-le de côté pour l'instant.

10.1.3 À propos de `hashCode()` et `equals()`

Il est recommandé, *dans certains cas*, d'implanter les méthodes `hashCode()` et `equals()` dans les classes d'objet persistants. La situation qui rend cette implantation nécessaire est caractérisée comme suit.

Quand faut-il implanter `hashCode()` et `equals()` ?

Si des instances d'une classe persistante doivent être conservées dans une collection (`Set`, `List` ou `Map`) qui couvre plusieurs sessions Hibernate, alors il est nécessaire de fournir une implantation spécifique de `hashCode()` et `equals()`

Disons-le tout de suite avant d'entrer dans les détails : il vaut mieux éviter cette situation, car elle soulève des problèmes qui n'ont pas de solution entièrement satisfaisante. À ce stade, vous pouvez donc décider que vous éviterez toujours de vous mettre dans ce mauvais cas, ce qui vous évite de plus d'avoir à lire ce qui suit. Sinon (ou si vous voulez vraiment comprendre de quoi il s'agit), continuez la lecture de cette section et faites-vous une idée par vous-mêmes de la fragilité introduite dans l'application par ce type de pratique.

Note : La discussion qui suit est relativement scabreuse et vous pouvez l'éviter, au moins en première lecture. Il n'est pas forcément indispensable de se charger l'esprit avec ces préoccupations.

Ce qui pose problème

Voici, sous forme de code synthétisé, un exemple de situation problématique.

```
// On maintient une liste des utilisateurs
Set<User> utilisateurs;

// Ouverture d'une première session
Session s1 =  sessionFactory.openSession();

// On ajoute l'utilisateur 1 à la liste
User u1 = s1.load (User.class, 1);
utilisateurs.add (u1);

// Fermeture de s1
s1.close();

// Idem, avec une session 2
Session s2 =  sessionFactory.openSession();
User u2 = s2.load (User.class, 1);
utilisateurs.add (u2);
s2.close();
```

Ce code instancie deux objets persistants `u1` et `u2`, correspondant à la même ligne, mais qui ne sont pas *identiques* puisqu'ils ont été chargés par deux sessions différentes. Ils sont insérés dans le `Set utilisateurs`, et comme leur `hashCode` (qui repose par défaut sur l'identité des objets) est différent, ce `Set` contient donc un doublon, ce qui ouvre la porte à toutes sortes de *bugs*.

Rappel sur le rôle de `hashCode()` et `equals()`.

Vous pouvez vous reporter à la documentation.

— http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode%28%29

— http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29

Conclusion : il faut dans ce cas implanter `hashCode()` et `equals()` non pas sur l'identité objet, mais en tenant compte de la *valeur* des objets pour détecter que ce sont les mêmes. Notez que le problème ne se pose pas si on reste dans le cadre d'une seule session.

Solution

Une solution immédiate, mais qui ne marche pas, est d'implanter l'égalité sur l'identifiant en base de données (l'attribut `id`). Cela ne marche pas dans le cas des identifiants auto-générés, car la valeur de ces identifiants n'est pas connue au moment où on les instancie. Exemple :

```
utilisateurs.add (new User("philippe"));
utilisateurs.add (new User("raphaël"));
```

On crée deux instances, pour lesquellles (tant qu'on n'a pas faite de `save()`) l'identifiant est à `null`. Si on base la méthode `hashCode()` sur l'`id`, seul le premier objet sera placé dans le `Set utilisateur`.

La seule solution est donc de trouver un ou plusieurs attributs de l'objet persistant qui forment une clé dite « naturelle », à savoir :

— identifiant l'objet de manière unique,
— dont la valeur est toujours connue,
— et qui ne change jamais.

C'est ici que l'on peut affirmer qu'il n'existe à peu près jamais un cas où ces critères sont satisfaits à 100%. Il restera donc toujours un doute sur la robustesse de l'application.

Ces réserves effectuées, voici un exemple d'implantation de ces deux méthodes, en supposant que le nom de l'utilisateur est une clé naturelle (ce qui est faux bien sûr).

```
@Override
public int hashCode() {
    HashCodeBuilder hcb = new HashCodeBuilder();
    hcb.append(nom);
    return hcb.toHashCode();
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof User)) {
        return false;
    }
    User autre = (User) obj;
    EqualsBuilder eb = new EqualsBuilder();
    eb.append(nom, autre.nom);
    return eb.isEquals();
}
```

Conclusion :

— Je vous déconseille tout à fait d'ouvrir et fermer des sessions, à moins d'avoir une excellente raison et de comprendre l'impact, illustré par l'exercice précédent.

— Je vous déconseille également de stocker dans des structures annexes des objets persistants : la base de données est là pour ça.

Note : Pour votre culture, un objet qui a été persistant et est soustrait du cache (par exemple parce que la session est fermée est un objet *détaché*. C'est un troisième statut, après *transient* et *persistant*.

Exercice : vérifier l'unicité, dans le contexte d'une session

Ecrivez une action qui lit deux fois le même film dans deux objets *a* et *b*, et vérifiez que ces deux objets sont identiques.

Complétez le test en utilisant deux sessions successives, en fermant la première et en en ouvrant une seconde. Lisez le même film dans un objet *c* et vérifiez qu'il n'est pas *identique* aux précédents.

Si votre application conserve des références à *a*, *b*, et *c*, vous obtenez donc (en ouvrant/fermant des sessions) des objets *distincts* correspondant à la *même* ligne.

NB : la lecture d'une ligne avec la clé primaire s'effectue avec :

```
session.load(Film.class, idFilm)
```

Exercice : comprendre le rôle de `hashCode()` et `equals()`

Pour chaque fragment de code ci-dessous, indiquez si l'assertion est un succès ou un échec, dans les cas suivants :
- `hashCode()` et `equals()` par défaut ;
- `hashCode()` et `equals()` basés sur la clé relationnelle (`id`) ;
- `hashCode()` et `equals()` basés sur une clé métier.

Justifiez votre réponse (et testez vous-mêmes pour vérifier).

```
HashSet someSet = new HashSet();
someSet.add(new PersistentClass());
someSet.add(new PersistentClass());
assert(someSet.size() == 2);
```

```
PersistentClass p1 = sessionOne.load(PersistentClass.class, new Integer(1));
PersistentClass p2 = sessionTwo.load(PersistentClass.class, new Integer(1));
assert(p1.equals(p2));
```

```
HashSet set = new HashSet();
User u = new User();
set.add(u);
session.save(u);
assert(set.contains(u));
```

10.2 S2 : Les opérations de lecture

Supports complémentaires :

- Diapos pour la session « S2 : Les opérations de lecture »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950dc7l4zh92k/

Maintenant que vous comprenez le fonctionnement interne d'Hibernate, au moins pour les grands principes, nous allons regarder rapidement les différentes options de lecture de données. Voici la liste des possibilités :

- par *navigation* dans le graphe des objets, si nécessaire chargé à la volée par Hibernate ;
- par identifiant : méthode basique, rapide, et de fait utilisée implicitement par d'autres méthodes comme la navigation ;
- par le langage de requêtes HQL ;
- par l'API `Criteria`, qui permet de construire par programmation objet une requête à exécuter ;
- enfin, directement par SQL, ce qui n'est pas une méthode portable et devrait donc être évité.

Nous passons ces méthodes en revue dans ce qui suit.

10.2.1 Accès par la clé

Deux méthodes permettent d'obtenir un objet par la valeur de sa clé. La première est `get`, dont voici une illustration :

```
return (Film) session.get(Film.class, id);
```

La seconde est `load` dont l'appel est strictement identique :

```
return (Film) session.load(Film.class, id);
```

Dans les deux cas, Hibernate examine d'abord le *cache* de la session pour trouver l'objet, et transmet une requête à la base si ce dernier n'est pas dans le cache. Les différences entre ces deux méthodes sont assez simples.

- si `load()` ne trouve par un objet, ni dans le cache, ni dans la base, une exception est levée ; `get()` ne lève jamais d'exception ;
- la méthode `load()` renvoie parfois un *proxy* à la place d'une instance réelle.

Notion : que'est-ce qu'un *proxy*

Un *proxy*, en général, est un intermédiaire entre deux composants d'une application. Dans notre cas, un *proxy* est un objet non persistant, qui joue le rôle de ce dernier, et se tient prêt à accéder à la base si *vraiment* des informations complémentaires sont nécessaires.

Retenez qu'un *proxy* peut décaler dans le temps l'accès réel à la base, et donc la découverte que l'objet n'existe pas en réalité. Il semble préférable d'utiliser systématiquement `get()`, quitte à tester un retour avec la valeur `null`.

Exercice : accès direct par la clé

Ecrivez une action `lectureParCle()` qui appelle `get()` et `load()`, d'abord pour un identifiant de film qui existe (par exemple, 1), puis pour un qui n'existe pas (par exemple, -1). Dans le second cas, gérez la non-existence par un test approprié pour chaque méthode.

10.2.2 Accès par navigation

Considérons l'expression *film.getRealisateur().getNom()* en java, ou plus simplement *film.realisateur.nom* en JSTL. Deux objets sont impliqués : le film et le réalisateur, instance de `Artiste`. Seul le premier (le film) est *à coup sûr* instancié sous forme d'objet persistant. L'artiste peut avoir déjà été instancié, ou pas.

Hibernate va d'abord tenter de trouver l'objet *Artiste*, en cherchant dans la table de hachage avec la valeur de l'attribut `id_realisateur` du film. Si l'objet ne figure pas dans le cache, Hibernate transmet une requête à la base,

```
select * from Artiste where id=:film.id_realisateur
```

charge l'objet persistant dans le cache, et le lie au film. Cette méthode de matérialisation progressive du graphe en fonction des actions de l'application est appelée « par navigation ». Elle nous amène à une question très intéressante : dans quelle mesure Hibernate peut-il « anticiper » la matérialisation du graphe pour éviter d'effectuer trop de requêtes SQL ?

Pour comprendre la navigation, et les enjeux du mécanisme de matérialisation associé, exécutez la vue suivante qui, en plus d'accéder au réalisateur du film, va chercher tous les films mis en scène par ce réalisateur.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

    <html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Accès à un film par la clé, et navigation</title>
    </head>
    <body>

      <h2>Le film no 1</h2>

      Nous avons bien ramené le film ${film.titre}

      <h2>Son réalisateur</h2>

      Et son réalisateur est ${film.realisateur.nom}

      <h2>Lequel a également réalisé ...</h2>

      <ul>
            <c:forEach items="${film.realisateur.filmsRealises}" var="film">
                  <li>${film.titre}</li>
            </c:forEach>
      </ul>
    </body>
    </html>
```

Associez cette vue à une action qui lit un film par sa clé, et suivez les instructions de l'exercice.

Exercice : étude de la méthode de chargement d'Hibernate

Testez la vue avec Hitchock ou Eastwood, et regardez soigneusement les requêtes générées par Hibernate affichées dans

la console. Que peut-on en conclure sur la stratégie de chargement ? Essayez de la décrire.

10.2.3 Le langage HQL

Bien entendu l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes. HQL (pour *Hibernate Query Language* bien sûr) est un langage de requêtes *objet* qui sert à interroger le graphe (virtuel au départ) des objets java constituant la vue orientée-objet de la base. Autrement dit, on interroge un ensemble d'objets java liés par des associations, et pas directement la base relationnelle qui permet de les matérialiser. Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête.

Voici un exemple simple de recherche de films par titre avec HQL.

```
public List<Film> parTitre(String titre)
{
        Query q = session.createQuery("from Film f where f.titre= :titre");
        q.setString ("titre", titre);
        return q.list();
}
```

Remarquez que la clause `select` est optionnelle en HQL : on interroge des *objets*, et la projection sur certains attributs offre peu d'intérêt. Elle a également le grave inconvénient de produire une structure (un ensemble de listes de valeurs) qui n'est pas pré-existante dans l'application, contrairement au modèle objet *mappé* sur la base. Sans la clause `select`, on obtient directement une collection des objets du graphe, sans acun travail de décryptage complémentaire.

Seconde remarque : comme en jdbc, on peut coder dans la requête des paramètres (ici, le titre) en les préfixant par « : » (« ? » est également accepté). Hibernate se charge de protéger la syntaxe de la requête, par exemple en ajoutant des barres obliques devant les apostrophes et autres caractères réservés.

Important : Il est totalement déconseillé de construire une requête comme une chaîne de caractères, à grand renfort de concaténation pour y introduire des paramètres.

Insistons sur le fait que HQL est un langage *objet*, même s'il ressemble beaucoup à SQL. Il permet de naviguer dans le graphe par exemple avec la clause `where`. Voici un exemple.

```
from Film f
where f.realisateur.nom='Eastwood'
```

10.2.4 L'API Criteria

Hibernate propose un ensemble de classes et de méthodes pour *construire* des requêtes sans avoir à respecter une syntaxe spécifique très différente de java.

```
public List<Film> parTitreCriteria(String titre)
{
        Criteria criteria = session.createCriteria(Film.class);
        criteria.add (Expression.eqOrIsNull("titre", titre));
        return criteria.list();
}
```

On ajoute donc (par exemple) des *expressions* pour indiquer les restrictions de la recherche. Il n'y a aucune possibilité de commettre une erreur syntaxique, et une requête construite avec `Criteria` peut donc être vérifiée à la complilation. C'est, avec le respect d'une approche tout-objet, l'argument principal pour cette API au lieu de HQL. Cela dit, on peut aussi estimer qu'une requête HQL est plus concise et plus lisible. Le débat est ouvert et chacun juge selon ses goûts et ses points forts (êtes-vous plus à l'aise en programmation objet ou en requêtage SQL ?). Nous avons choisi dans ce qui suit de présenter uniquement HQL, et d'ignorer l'API Criteria que nous vous laissons explorer par vous-mêmes si vous pensez qu'il s'agit d'une approche plus propre.

Exercice : un premier formulaire de recherche

Vous en savez assez pour créer une première fonction de recherche de films combinant quelques critères comme : le titre, l'année (ou un intervalle d'année), le genre. Proposez un formulaire HTML pour saisir ces critères, exécutez la requête en HQL et/ou avec l'API `Criteria`, affichez le résultat.

10.3 Résumé : savoir et retenir

L'information essentielle à retenir de ce chapitre est le rôle joué par la session Hibernate et le cache des objets maintenu par cette session. Il doit être clair pour vous qu'Hibernate consacre beaucoup d'efforts à maintenir dans le cache une image objet cohérente et non redondante de la base. Cela impacte l'exécution de toutes les méthodes d'accès dont nous avons donné un bref aperçu.

Manipulez l'objet `Session` avec précaution. Une méthode saine (dans le contexte d'une application Web) est d'ouvrir une session en début d'action, et de la fermer à la fin.

CHAPITRE 11

Le langage HQL

Voici donc maintenant une présentation des aspects essentiels de HQL. Mon but n'étant pas de ré-écrire la documentation en ligne, et donc d'être exhaustif, le contenu qui suit se concentre sur les principes, avec des exemples illustratifs.

11.1 S1 : HQL, aspects pratiques

Supports complémentaires :

- Diapos pour la session « S1 : HQL, aspects pratiques »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950e76wmhuozw/

Pour tester HQL, je vous recommande d'utiliser, sous Eclipse, le *plugin Hibernate Tools*. Voici comment l'installer, et quelques instructions d'utilisation.

- Sous Eclipse, dans le menu *Help*, accédez au choix *Marketplace* (ou *Software search and updates*, selon les versions).
- Lancez une recherche sur *Hibernate Tools*.
- Lancez l'installation, puis relancez Eclipse quand elle est terminée.

Une fois le *plugin* installé, vous pouvez ouvrir des *vues* Eclipse. Dans le menu *Windows*, puis *Show view*, choisissez l'option *Hibernate* comme le montre la figure suivante :

11.1.1 Configuration

Lancez tout d'abord la fenêtre *Hibernate Configurations*. Vous pouvez créer des configurations (soit, essentiellement, le contenu d'un fichier `hibernate.cfg.xml`). Une configuration peut être créée à partir d'un projet existant, ce qui revient à utiliser les paramètres Hibernate déjà définis pour le projet.

Quand vous êtes sur l'onglet *Configuration* de la nouvelle fenêtre, des boutons sur la droite permettent de créer une nouvelle configuration ; pour une configuration existante, le bouton droit donne accès à l'édition de la configuration, au lancement d'un éditeur HQL, etc.

Le plus simple est donc d'indiquer le projet associé à la configuration, et d'utiliser le fichier de *mapping* `hibernate.cfg.xml` de ce projet. Cela assure que les classes persistantes et la connexion à la base sont automatiquement détec-

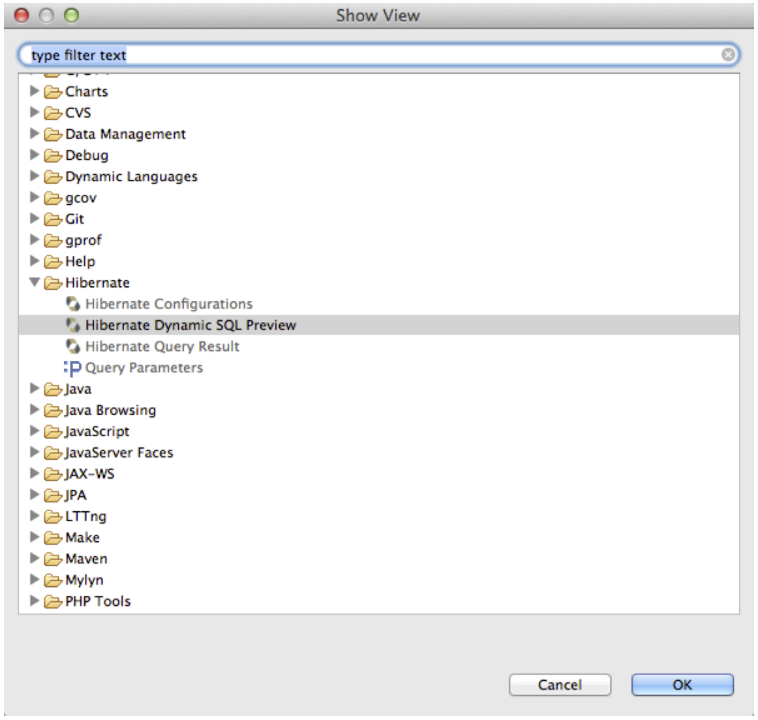

FIG. 1 – Lancement des fenêtres de Hibernate Tools

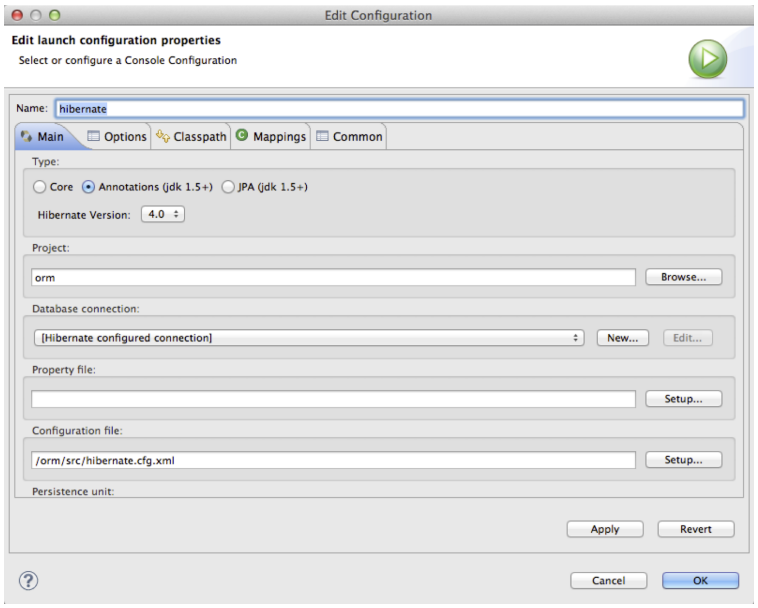

FIG. 2 – Configuration d'une session Hibernate

tées. L'outil vous permet également de créer ou modifier un fichier de configuration avec une interface graphique asez agréable.

11.1.2 Utilisation

Une fois la configuration créée, vous pouvez lancer un éditeur HQL et y saisir des requêtes. Pour l'éditeur HQL, sélectionnez votre configuration de session, et utilisez soit le bouton droit, soit l'icône « HQL » que vous voyez sur la barre d'outils. Vous pouvez saisir des requêtes HQL dans l'éditeur, et les exécuter avec la flèche verte.

Trois autres vues sont utiles :

— *Hibernate Dynamic SQL Preview* vous montre la/les requête(s) SQL lancée(s) par Hibernate pour évaluer la requête HQL ; très utile pour comprendre comment Hibernate matérialise le graphe requis par une requête HQL.
— *Hibernate Query Result* vous donne simplement le résultat de la requête HQL.
— *Query parameters* sert à définir la valeur des paramètres pour les requêtes qui en comportent.

Pour résumer, votre environnement de test HQL devrait ressembler à la figure ci-dessous.

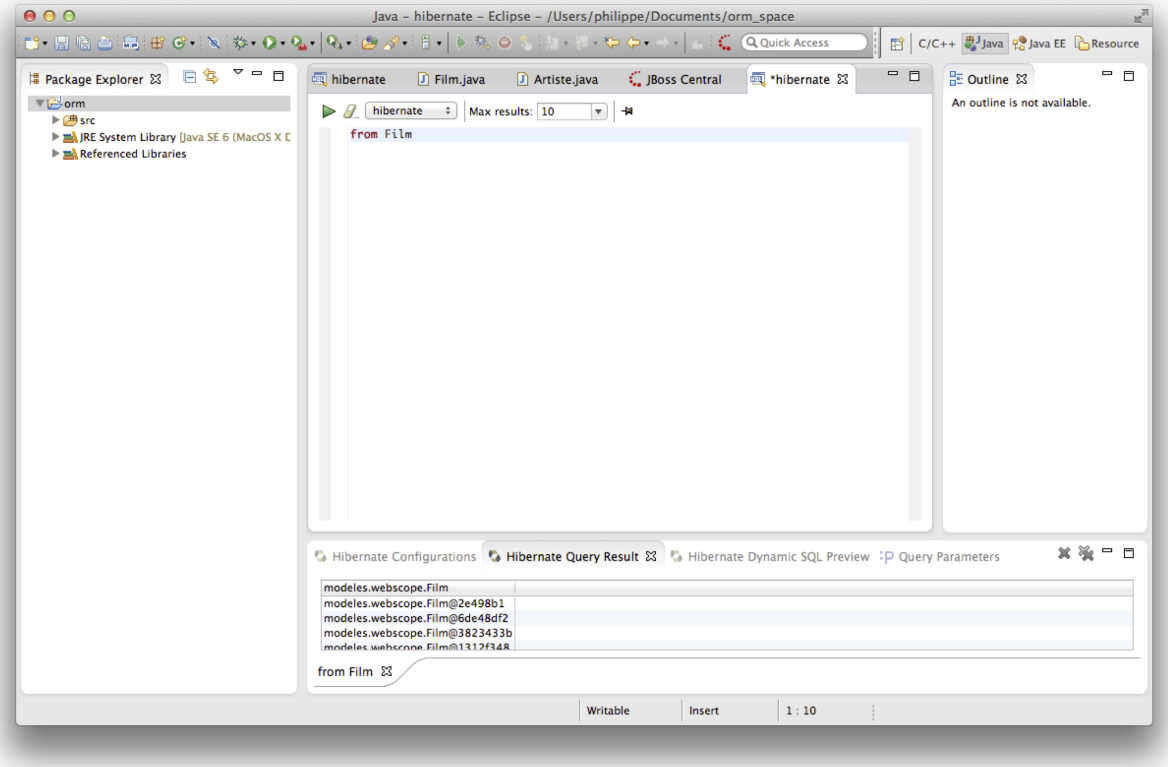

Fig. 3 – Les fenêtres Hibernate Tools en action

Une dernière astuce : les requêtes HQL renvoient en général des *objets*, et Hibernate Tools affiche la *référence* de ces objets, ce qui n'est pas très parlant. Pour consulter plus facilement le résultat des requêtes, vous pouvez ajouter une clause `select` (optionnelle en HQL) :

```
select film.titre, film.annee
from Film as film
```

11.1.3 HQL et java

L'intégration de HQL à Java se fait par des méthodes de la `Session`. Le code suivant recherche un (ou plusieurs) film(s) par leur titre.

```
public List<Film> parTitre(String titre)
{
        Query q = session.createQuery("from Film f where f.titre= :titre");
        q.setString ("titre", titre);
        return q.list();
}
```

La clause `select` est optionnelle en HQL, et offre peu d'intérêt dans le cadre de l'application java car on cherche en général à récupérer automatiquement des instances des classes *mappées*, ce qui implique de récupérer tous les attributs de chaque ligne.

Commme en jdbc, on peut introduire dans la requête des *paramètres* (ci-dessus, le titre) en les préfixant par « : » (« ? » est également accepté). Hibernate se charge de protéger la syntaxe de la requête, par exemple en ajoutant des barres obliques devant les apostrophes et autres caractères réservés. Voici une requête un peu plus complète, tout en étant formulée plus concisément.

```
session.createQuery("from Film as film where film.titre like :titre and film.annee <␣
↪:annee")
               .setString ("titre", "%er%")
               .setInteger("annee", 2000)
               .list();
```

On applique la technique dite de « chaînage des méthodes », chaque méthode `set` renvoyant l'objet-cible, auquel on peut donc appliquer une nouvelle méthode, et ainsi de suite, pour une syntaxe beaucoup plus concise.

Notez également que l'affectation des paramètres tient compte de leur type : Hibernate propose des `setDate()`, `setInteger()`, `setTimestamp()`, etc. On peut également utiliser comme paramètre un objet persistant, comme le montre l'exemple suivant :

```
// bergman est une instance de Artiste
Artiste bergman = ...;

session.createQuery("from Film as film where film.realisateur= :mes")
               .setEntity ("mes", bergman)
               .list();
```

HQL offre la possibilité de *paginer* les résultats, une option souvent utile dans un contexte d'application web. L'exemple suivant retourne les lignes 10 à 19 du résultat.

```
session.createQuery("from Film")
               .setFirstResult (10)
               .setMaxResults(10)
               .list();
```

La méthode `list()` de l'objet `query` est la plus générale pour exécuter une requête et constituer le résultat. Si vous êtes sûr que ce dernier ne contient qu'un seul objet, vous pouvez utiliser `uniqueResult()`.

```
session.createQuery("from Film where titre='Vertigo'")
                .uniqueResult();
```

Attention, une exception est levée si plus d'une ligne est trouvée. Cette méthode ne devrait être appliquée que pour des recherches portant sur des attributs déclarés `unique` dans le schéma de la base de données (dont la clé primaire).

Note : À partir de maintenant je donne les requêtes HQL indépendamment du code java, dans la forme « pure » que vous pouvez tester sous Hibernate Tools.

11.2 S2 : base de HQL

Supports complémentaires :

- Diapos pour la session « S2 : Base de HQL »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950ee5kzsay1m/

Rappelons que HQL est un langage de requêtes *objet* qui sert à interroger le graphe (virtuel au départ) des objets java constituant la vue orientée-objet de la base. Autrement dit, on interroge *un ensemble d'objets java* liés par des associations, et pas directement la base relationnelle qui permet de les matérialiser. Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête. Cela donne une syntaxe et une interprétation parfois différente de SQL, même si les deux langages sont en apparence très proches.

11.2.1 Sélection d'objets

Toute cette section se concentre sur l'interrogation d'une seule classe.

La clause `from`

La forme la plus basique de HQL comprend la clause `from`, et indique la classe dont l'extension est sujet de la recherche :

```
from Film
```

Le point intéressant est que, en cas de hiérarchie d'héritage, il est possible d'interroger les super- et sous-classes. La liste des vidéos est obtenue par :

```
from Video
```

et la liste des films (sous-classe de `Video`) par :

```
from FilmV
```

On peut rechercher tous les objets persistants avec :

```
from java.lang.Object
```

On peut créer des *alias* pour désigner les objets et leur appliquer des traitements de sélection ou projection. La requête suivante définit un alias `film` :

```
select film.titre
from Film as film
where film.annee < 2000
```

Le mot-clé `as` est optionnel (comme la clause `select`). Je l'utilise systématiquement pour plus de clarté. Préfixer un attribut par l'alias n'est indispensable que s'il y a risque d'ambiguité, ce qui n'est pas le cas dans l'exemple ci-dessus.

La clause `where`

Les opérateurs de comparaison sont très proches de ceux de SQL : `=`, `<`, `>`, `between`, `in`, `not between`, etc. Les opérateurs `like` et `not like` s'utilisent comme en SQL et expriment la même chose. Les opérateurs logiques sont les mêmes qu'en SQL : `and`, `or` et `not`, avec ordre de priorité défini par le parenthésage. De même, HQL dispose d'une clause `order by` semblable à celle de SQL. Voici un exemple un peu complet :

```
from Film as film
where (titre like '%er%' and annee between 1970 and 2000)
or film.genre='Drame'
order by film.titre
```

Attention à la valeur `NULL` ou `null`. Comme en SQL, elle indique *l'absence* de valeur. On la teste avec l'expression `is null`, comme en SQL :

```
from Film as film
where  film.resume is null
```

Note : il existe beaucoup de fonctions pré-définies que peuvent être appliquées aux propriétés des objets : `size()`, `minElement()`, `trunc()`, `trim()`, etc. Je vous renvoie à la documentation pour une liste complète.

Jointures implicites

Contrairement à SQL où les critères de sélection ne peuvent concerner que les propriétés de la table interrogée, HQL permet la navigation vers des objets associés (et donc vers d'autres tables), avec une syntaxe abrégée et intuitive. On peut par exemple chercher les films dirigés par Clint Eastwood :

```
from Film as film
where film.realisateur.nom = 'Eastwood'
```

Cela ne fonctionne *que* pour les associations `many-to-one` et `one-to-one`. Vous ne pouvez pas écrire :

```
from Film as film
where film.roles.acteur.nom = 'Eastwood'
```

Attention, il n'y a pas de magie : cette syntaxe nécessite une *jointure* au niveau de la requête SQL générée par Hibernate, et correspond en fait à une manière dite *implicite* d'exprimer une jointure. Voici la requête SQL engendrée par Hibernate.

```
select *
from Film film0_ cross join Artiste artiste1_
where film0_.id_realisateur=artiste1_.id
and artiste1_.nom='Eastwood'
```

On peut bien entendu exprimer *explicitement* des jointures, avec des options assez riches détaillées dans la section suivante.

Exercice : expression de restrictions

- Trouver lez films dont le titre comprend le mot "parrain"
- Trouvez tous les drames
- Trouvez tous les artistes nés avant 1970
- Trouvez tous les réalisateurs (c-à-d. les artistes qui ont mis en scène un film)

11.3 S3 : Les jointures

Supports complémentaires :

— Diapos pour la session « S3 : Les jointures »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35950f6ca878jyg/

Jusqu'à présent, nous prenons les objets d'une classe, ce qui correspond donc à une recherche de ligne dans une seule table pour les instancier. Comme en SQL, il est nécessaire d'exprimer des *jointures* si l'on veut exprimer des critères de recherche sur les objets associés.

11.3.1 Jointures HQL et jointures SQL

Comme en SQL, une jointure s'exprime avec le `join`.

— en HQL, en exprime des jointures par *navigation* en exploitant les associations d'objets définies dans le *mapping*,
— en SQL, on applique (conceptuellement) le produit cartésien de deux tables, et on restreint le résultat à des combinaisons de lignes satisfaisant certains critères.

Il faut bien réaliser qu'une jointure par HQL qui, en apparence, accède directement aux objets associés, est en fait implantée par Hibernate comme une jointure SQL. Soyons concrets avec un premier exemple : on veut tous les films mis en scène par Clint Eastwood :

```
select film
from Film as film
join film.realisateur as a
where a.nom='Eastwood'
```

On accède donc directement à l'objet associé `realisateur`, sans mention explicite de la classe `Artiste`. En sous-main, Hibernate engendre et exécute la requête SQL suivante.

```
select film.*
from Film film0_  inner join  Artiste artiste1_
  on film0_.id_realisateur=artiste1_.id
where artiste1_.nom='Eastwood'
```

Vous voyez que la version SQL comprend une contrainte d'égalité entre la clé primaire de `Artiste` et la clé étrangère dans `Film`, contrainte qui n'apparaît pas en HQL puisqu'elle est représentée par une association entre objets dans le graphe virtuel. La définition du *mapping* permet à Hibernate d'engendrer le code SQL correct.

On peut adopter en HQL une expression plus proche de la logique SQL, mais moins naturelle du point de vue d'une logique « graphe d'objet ».

```
select film
from Film as film,  Artiste as a
where film.realisateur = a
and a.nom='Eastwood'
```

Le code SQL engendré est le même. Il n'y a donc aucune différence en termes de performance ou d'expressivité. Cette dernière version illustre une autre spécificité « objet » de HQL : on compare directement *l'identité* de deux objets (un artiste, un réalisateur) avec la clause :

```
where film.realisateur = a
```

et pas, comme en SQL, *l'égalité* des clés primaires de ces objets. Cela étant, il est possible d'effectuer directement une jointure sur les clés en HQL :

```
select film
from Film as film,  Artiste as a
where film.realisateur.id = a.id
and a.nom='Eastwood'
```

Ici, `id` est un mot-clé représentant la clé primaire, quel que soit son nom réel.

Le rôle de la clause `select`

Voici un autre exemple de jointure, impliquant une association un à plusieurs. Elle sélectionne tous les films dont un des rôles est *McClane* :

```
select distinct film
from Film as film
join film.roles as role
where role.nom= 'McClane'
```

La clause `select` prend ici une importance particulière : dans la mesure où nous accédons aux extensions de deux classes (`Film` et `Role`), il est préférable de préciser quels sont les objets que nous plaçons dans le résultat. Sans clause `select`, on obtient une liste de paires d'objets (un film, un rôle), soit une instance de `List<Object[]>`. Pour chaque élément de la liste, le premier composant du tableau est un `Film`, le second un `Rôle`.

De plus, le film est dupliqué autant de fois qu'il a de rôles associés, ce qui rend indispensable l'utilisation du mot-clé `distinct`. Executez par exemple la requête suivante.

```
select film.titre, role.nom
from Film as film
join film.roles as role
```

On obtient autant de fois le titre du film qu'il y a de rôles. Un tel résultat est assez laborieux à traiter, et on perd en grande partie les avantages de HQL (à savoir instancier un résultat directement exploitable).

Dernier exemple, tous les films où Clint Eastwood joue un rôle :

```
select distinct film
from Film as film
join film.roles as role
join role.pk.acteur as acteur
where acteur.nom= 'Eastwood'
```

Notez le chemin `role.pk.acteur`, exprimant un *chemin* dans le graphe des objets et composants. Je vous invite à consulter la requête SQL engendrée par Hibernate pour cette jointure.

La clause `join` de HQL

Les jointures de HQL sont les mêmes que celles du SQL ANSI.

- `[inner] join` : exprime une jointure standard pour impliquer (pas forcément matérialiser) les objets associés ;
- `left [outer] join` : exprime une *jointure externe à gauche* : si par exemple un film n'a pas de metteur en scène, il (le film) fera quand même partie du résultat, avec un objet `realisateur` associé à `null` ;
- `right [outer] join` est la complémentaire de `left join` (et a donc peu d'intérêt en pratique).

Pour bien voir la différence, exécutez les deux requêtes suivantes :

```
from Film as film
inner join film.roles as role
```

et :

```
from Film as film
left outer join film.roles as role
```

La seconde devrait vous ramener *aussi* les films pour lesquels aucun rôle n'est connu.

Comme indiqué plus haut, cette requête renvoie une instance de `List<Object[]>`, qu'il faut ensuite décoder, pour obtenir, respectivement, des films et des rôles.

Ce qui soulève la question suivante : *si, à partir de l'un des films sélectionnés, je veux accéder aux rôles par la collection* `film.roles`, *Hibernate a-t-il déjà instancié cette collection, ou va-t-il devoir à nouveau accéder à ces rôles avec une requête SQL ?*.

La réponse est : une nouvelle requête sera effectuée pour trouver les rôles du film, ce qui semble dommage car la jointure SQL sera effectuée deux fois. *Les requêtes HQL que nous présentons de créent pas de graphe, mais instancient simplement les objets sélectionnés, sans lien entre eux*. Dit autrement : le fait d'effectuer une jointure entre un film et des rôles *n'implique pas* que le graphe *film-rôles* est matérialisé dans le cache de premier niveau.

Ce n'est pas si illogique que cela en a l'air, si on prend en compte le fait qu'il est possible d'appliquer des *restrictions* sur les objets sélectionnés, comme dans l'exemple ci-dessous.

```
from Film as film
left outer join film.roles as role
where role.nom='McClane'
```

Si on matérialisait le graphe à partir des objets sélectionnés, la collection `roles` de chaque film serait incomplète puisqu'elle ne comprendrait que ceux dont le nom est `McClane`. En résumé, il faut bien distinguer deux motivations différentes, et partiellement incompatibles :

- la *sélection* d'objets satisfaisant des critères de restriction ;
- la *matérialisation* d'une partie du sous-graphe de données.

Les requêtes HQL que nous présentons ici correspondent à la première motivation. Il est possible d'utiliser HQL *aussi* pour la matérialisation du graphe de données, avec une option `fetch` que nous étudierons dans le prochain chapitre.

Important : Retenez que ces deux motivations peuvent être *incompatibles* : si je *restreins* avec la clause *where* les objets sélectionnés (les films avec un rôle McClane), je ne peux pas obtenir en même temps un graphe complet (tous les rôles des films, dont un des rôles est McClane).

En résumé, concentrez-vous pour l'instant sur l'utilisation de HQL pour sélectionner et matérialiser les objets qui vous intéressent, sans chercher à optimiser les performances ou éliminer les redondances de requêtes SQL engendrées. Cela revient à toujours utiliser la clause `select` pour garder les objets qui vous intéressent, et à utiliser les jointures pour exprimer des restrictions. Par exemple :

```
select distinct film
from Film as film
left outer join film.roles as role
where role.nom='McClane'
```

Comme en SQL, une autre manière d'exprimer une jointure est d'utiliser des sous-requêtes, qui sont présentées un peu plus loin.

Exercice : pour quelques jointures de plus

— Trouvez les films avec l'acteur Bruce Willis.
— Affichez les noms des acteurs ayant joué le rôle de "Jack Dawson", et le metteur en scène qui les a dirigés.
— Affichez la liste des metteurs en scène, et les films qu'ils ont réalisés
— Affichez les films dans lesquels le metteur en scène est également un acteur (pensez : « HQL, langage objet » !)
— Affichez la liste des genres, et pour chaque genre les films correspondant.

11.4 S4 : Compléments

Supports complémentaires :
— Diapos pour la session « S4 : Compléments »
— Vidéo associée : à venir

Nous avons vu l'essentiel de HQL dans ce qui précède. Voici quelques compléments utiles.

11.4.1 Sous-requêtes

Comme en SQL, HQL permet l'expression de sous-requêtes, ce qui revient souvent à une nouvelle manière d'exprimer des jointures. Voici donc une nouvelle manière d'obtenir les films dirigés par Clint Eastwood.

```
from Film as film
where  film.realisateur in (from Artiste where nom='Eastwood')
```

Par rapport à l'expression standard avec un `join`, il existe une différence qui mérite d'être mentionnée : comme la classe `Artiste` n'apparaît que dans la sous-requête, les instances de `Artiste` ne figurent pas dans le résultat. Il n'est donc pas nécessaire d'utiliser une clause `select` pour se restreindre aux films.

Voici une autre version de la requête qui ramène les films dans lesquels figure un rôle "McClane".

```
from Film as film
where  film in (select role.pk.film from Role as role
              where role.nom='McClane')
```

Et une autre version de la requête qui recherche les films dirigés par Clint Eastwood, cette fois avec une sous-requête `exists`.

```
select film.titre
from Film as film
where  exists (from Artiste as a where a = film.realisateur and a.nom='Eastwood')
```

Il est bien clair que cette expression est très compliquée par rapport à celle basée sur la jointure par navigation (et en particulier la jointure implicite, voir ci-dessus).

Les sous-requêtes en HQL sont tout à fait similaires à celles de SQL. Je n'entre donc pas dans l'énumération des diverses possibilités. Pour rappel, les quantificateurs suivants sont composables avec les opérateurs de comparaison, et s'appliquent à des sous-requêtes qui ramènent plusieurs lignes.

— `any` : vrai si *au moins* une ligne ramenée par la sous-requête satisfait la comparaison ;
— `all` : vrai si *toutes* les lignes ramenées par la sous-requête satisfontt la comparaison.

Des synonymes pour `any` sont `in` et `some`. Voici deux exemples. Le premier recherche les films dont au moins un acteur est né avant 1940.

```
from Film as film
where 1940 > any (select  role.pk.acteur.anneeNaissance
                  from Role as role
                  where role.pk.film=film)
```

Il est beaucoup plus simple (et plus lisible) d'exprimer ce genre de requête avec un `join`. L'utilisation de `all` (pour : « les films dont *tous* les acteurs sont nés avant 1940 ») paraît plus justifiée :

```
from Film as film
where 1940 > all (select  role.pk.acteur.anneeNaissance
                  from Role as role
                  where role.pk.film=film)
```

Comme en SQL, il existe plusieurs syntaxes possibles pour une même requête. Par exemple, celle ci-dessus s'exprime aussi avec `not exists` (à vous de jouer).

Exercice : requêtes imbriquées

Les requêtes suivantes doivent être exprimées avec des sous-requêtes

— Quels internautes n'ont donné que des notes supérieures à 3 ?
— Quels sont les films dans lesquels joue Brad Pitt ?
— Et quels sont les films qui n'ont pas obtenu une note de 5 ?

11.4.2 Création d'objet spécifiques

Revenons sur les requêtes *ad hoc* qui constituent des résultats constitués de *projection* sur certains attributs. Voici un exemple

```
select film.titre, film.annee, film.realisateur.prenom, film.realisateur.nom
from Film as film
```

Quel est l'intérêt de cette requête ? A priori elle implique un travail de décodage fastidieux du résultat, constitué d'une liste de tableaux, chaque tableau contenant 4 objets correspondant aux 4 attributs projetés.

Il existe quand même un avantage potentiel : comme la requête ne nécessite pas la création d'objets persistants, on évite le placement de ces objets dans le cache de la session, avec le (modeste) surcoût induit par le test d'existence, et l'insertion dans la table de hachage.

On gagne donc (un peu) en performance. Il se peut que dans certains cas le gain vaille le surcroît de programmation nécessaire.

11.4.3 Regroupement et agrégation

Le `group by`, le `having` et les fonctions d'agrégation sont identiques en HQL et en SQL (ou, pour être plus précis, elles ne présentent aucune spécificité supplémentaire par rapport à celles que nous avons déjà vues). Quelques exemples suffiront.

— La note moyenne des films

```
select avg(note)
from Notation
```

— Les films et leur nombre d'acteurs

```
select film, count(*)
from Film as film join film.roles as role
group by film
```

Notez qu'on aimerait faire `select film, count(film.roles) from Film as film`, mais HQL n'accepte pas cette syntaxe.

— Les metteurs en scène ayant réalisé au moins trois films

```
select artiste.nom, count(*)
from Artiste as artiste join artiste.filmsRealises as film
group by artiste
having count(*)  > 3
```

Exercice : requêtes imbriquées

— Les internautes et le nombre de films qu'ils ont notés
— Donnez les films avec au moins 4 rôles.
— Films dont la note moyenne est supérieure à la note moyenne de tous les films.

CHAPITRE 12

Optimisation des lectures

Le moment est venu de chercher à contrôler les requêtes SQL engendrées par Hibernate pour éviter de se retrouver dans la situation défavorable où de nombreuses requêtes sont exécutées pour charger, petit à petit, des parties minimes du graphe de données. Cette situation apparaît typiquement quand le graphe à matérialiser est découvert par la couche Hibernate au fur et à mesure des navigations de l'application. Or, cette dernière (ou plus exactement le concepteur de l'application : vous) sait souvent *à l'avance* à quelles données elle va accéder, et peut donc anticiper le chargement grâce à des requêtes SQL (idéalement, une seule) qui vont matérialiser l'intégralité la partie du graphe contenant ces données visitées.

12.1 S1 : Stratégies de chargement

Supports complémentaires :

— Diapos pour la session « S1 : Stratégies de chargement »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35951031abhhhtt/

La *stratégie de chargement* spécifie cette anticipation. Essentiellement, elle indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé. La stratégie de chargement peut être paramétrée à deux niveaux.

— *dans la configuration du mapping*, autrement dit sous forme d'une annotation JPA ;
— *à l'exécution de requêtes HQL*, grâce au mot-clé `fetch`, déjà évoqué.

Ces deux niveaux sont complémentaires. La spécification au niveau de la configuration a l'inconvénient d'être insensible aux différents contextes dans lesquels la base de données est interrogée. Pour prendre l'exemple de notre base, dans certains contextes il faudra, quand on accède à un film charger les acteurs, dans d'autre cas ce seront les notes, et enfin dans un troisième cas ni l'une ni l'autre des collections associées ne seront nécessaires.

La bonne méthode consiste donc à indiquer une stratégie de chargement *par défaut* au niveau de la configuration, et à la *surcharger* grâce à des requêtes HQL spécifiques dans les contextes où elle ne fait pas l'affaire. C'est ce que nous étudions dans ce chapitre.

Note : Nous reprenons les requêtes du contrôleur `Requeteur` et nous allons nous pencher sur les requêtes SQL engendrées par Hibernate.

L'étude des stratégies de chargement est rendue compliquée par plusieurs facteurs défavorables.

— la stratégie par défaut (si on ne spécifie rien) tend à changer d'une version Hibernate à une autre ;
— la stratégie par défaut n'est pas la même en Hibernate et en JPA ;
— la documentation n'est pas limpide, c'est le moins qu'on puisse dire.

Il est donc préférable de ne pas se fier à la stratégie par défaut mais de toujours la donner explicitement, en se basant sur des principes que nous allons identifier. Et il est sans doute nécessaire de vérifier le comportement d'Hibernate sur les requêtes les plus importantes de notre application, pour éviter des accès sous-optimaux à la base sous-jacente.

Nous commençons par une petite étude de cas pour bien comprendre le problème.

12.1.1 Petite étude de cas : le problème

Vous avez déjà écrit une action `lectureParCle` dans votre contrôleur, qui se contente de charger un film par son id avec `load()` ou `get()`. Nous allons associer à cette action maintenant la vue minimale suivante.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Accès à un film par la clé, sans navigation</title>
</head>
<body>

      <h2>Le film no 1</h2>

      Nous avons bien ramené le film ${film.titre}. Et c'est tout.

 </body>
</html>
```

Nous n'accédons donc à aucun objet associé. Seul le titre est affiché. Exécutez cette action et examinez la requête SQL engendrée par Hibernate. Si vous avez utilisé les annotations JPA, cette requête devrait être la suivante (j'ai retiré la liste du `select` pour ne conserver que les jointures).

```
select [...]
from Film film0_ left outer join Genre genre1_ on film0_.genre=genre1_.code
   left outer join Pays pays2_ on film0_.code_pays=pays2_.code
     left outer join Artiste artiste3_ on film0_.id_realisateur=artiste3_.id
where film0_.id=?
```

Hibernate a donc engendré une requête qui effectue une jointure (externe) pour toutes les associations de type `@ManyToOne`. C'est clairement inutile ici puisque nous n'accédons ni au genre, ni au pays, ni au metteur en scène.

Faisons la même expérience, cette fois en utilisant HQL. Vous avez dû écrire une action `parTitre()` qui exécute la méthode suivante :

```
public List<Film> parTitre(String titre)
{
        Query q = session.createQuery("from Film f where f.titre= :titre");
        q.setString ("titre", titre);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return q.list();
}
```

Associez à cette action la vue suivante, qui affiche pour chaque film son titre et le nom du réalisateur.

```
<h2>Les films</h2>

Voici les films ramenés:

<ul>
        <c:forEach items="${films}" var="film">
                <li>Nous avons bien ramené le film ${film.titre} dont le
                        réalisateur est ${film.realisateur.nom}</li>
        </c:forEach>
</ul>
```

Exécutez l'action. Vous devriez voir 4 requêtes SQL.

— la première correspond à l'exécution de la requête HQL.

— les trois suivantes correspondent, respectivement, à la recherche du genre, du pays et du réalisateur ; soit :

```
select * from Genre genre0_ where genre0_.id=?
select * from Pays pays0_ where pays0_.id=?
select * from Artiste artiste0_ where artiste0_.id=?
```

Les id recherchés sont bien entendu les valeurs des clés étrangères trouvées dans `Film`.

Les seconde et troisième requêtes sont ici inutiles. La quatrième ne l'est que parce que nous affichons le réalisateur, mais en fait Hibernate effectue « aveuglément » et systématiquement la recherche des objets liés au film par une association `@ManyToOne`, dès que ce film est placé dans le cache, sans se soucier de savoir si ces objets vont être utilisés. Il s'agit d'une stratégie dite de *chargement immédiat* (ou « glouton », *eager* en anglais), en général peu efficace.

12.1.2 Petite étude de cas : la solution

Bien, nous allons remédier à cela en indiquant explicitement la stratégie de chargement à appliquer. Editez la classe `Film.java` et complétez toutes les annotations `@ManyToOne` comme suit :

```
@ManyToOne(fetch=FetchType.LAZY)
```

Nous souhaitons un chargement « paresseux » (*lazy*). Maintenant, exécutez à nouveau l'action de recherche par clé : la requête SQL devrait être devenue :

```
select [...] from Film film0_ where film0_.id=?
```

Beaucoup mieux n'est-ce pas ? On ne charge que ce qui est nécessaire, sans jointure superflue. Considérons la seconde action, celle qui effectue une recherche HQL. Si vous la ré-exécutez avec la stratégie indiquée, vous devriez voir deux requêtes SQL :

```
select [...] from Film film0_ where film0_.id=?
select * from Artiste artiste0_ where artiste0_.id=?
```

Pourquoi ? La première requête est toujours celle correspondant à la requête HQL. La seconde est déclenchée *par navigation* : quand on veut afficher le nom du réalisateur, Hibernate déclenche l'accès à l'artiste et son chargement. Cet accès est nécessaire car l'objet n'est pas présent dans le cache, Hibernate étant maintenant en mode `Lazy`.

La stratégie indiquée est donc parfaite pour notre première action, mais pas pour la seconde où une jointure avec la table `Artiste` quand on recherche le film serait plus appropriée. En fait, cela montre qu'aucune stratégie générale, indiquée au niveau de la configuration, ne peut être optimale dans tous les cas. Il nous reste une solution, qui est d'adapter la stratégie de chargement au contexte particulier d'utilisation.

Remplacez la requête HQL dans la méthode `parTitre()` par la suivante :

```
select film from Film as film join fetch film.realisateur
where film.titre= :titre
```

Ré-exécutez l'action de recherche par titre, et vous ne devriez plus voir qu'une seule requête SQL effectuant la jointure entre `Film` et `Artiste`. Cette requête charge le film, l'artiste, *et surtout matérialise le graphe d'association entre les deux objets*. Du coup, quand on navigue du film vers son réalisateur, l'objet `Artiste` est trouvé dans le cache et il n'est plus nécessaire d'effectuer une requête SQL supplémentaire. Avez-vous noté le mot-clé `fetch` dans la requête HQL ? C'est lui qui entraine une surcharge de la stratégie de chargement par défaut.

Et voilà ! Récapitulons.

12.1.3 Résumé des leçons tirées

Qu'avons-nous appris ? Et bien, en fait, l'essentiel de ce qu'il faut comprendre pour adopter une méthode fondée de choix d'une stratégie de chargement. Pour résumer :

- Hibernate (ou JPA) applique une stratégie par défaut qui tend à multiplier les requêtes SQL ou les jointures externes inutilement.
- Cette stratégie par défaut peut être remplacée par une configuration explicite dans les annotations de *mapping*.
- Une stratégie au niveau de la configuration ne peut être optimale dans tous les cas : on peut alors, *dans un contexte donnée*, la remplacer par un chargement adapté au contexte, exprimé en HQL avec l'option `fetch`.

J'ai pris l'hypothèse un peu simpliste que l'on cherche à minimiser le nombre de requêtes SQL. C'est une bonne approche en général, car elle laisse l'optimiseur du SGBD déterminer la bonne méthode d'exécution, en tenant compte de ses ressources propres (index, mémoire). Il se peut qu'une requête SQL engendrée soit trop complexe et pénalise au contraire les performances. Là, on tombe dans l'expertise des plans d'exécution de requête : je vous renvoie à mon cours sur les aspects systèmes des bases de données pour en savoir plus (http://sys.bdpedia.fr).

12.2 S2 : Configuration des stratégies de chargement

Supports complémentaires :

- Diapos pour la session « S2 : Configuration des stratégies de chargement »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f359510eeb5q0tyq/

La stratégie de chargement peut donc être spécifiée avec la configuration des classes persistantes, et « surchargée » par l'exécution de requêtes HQL indiquant explicitement une stratégie particulière, adaptée au contexte local. Dans cette section nous étudions la spécification par la configuration.

12.2.1 Les stratégies Hibernate/JPA

Note : La documentation Hibernate étant particulièrement confuse sur le sujet (cf. chapitre 20), je tente une simplification/clarification : si quelqu'un me démontre qu'elle ne correspond pas complètement à la réalité je serai ravi de faire un ajustement.

Un petit rappel sur la terminologie avant de commencer : dans ce qui suit, étant donné un objet *x* chargé par Hibernate, désigné par *entité principale*, l'expression *entité (secondaire)* désigne un objet *y* lié à *x* via une assocation `@ToOne`, et l'expression *collection* désigne un ensemble d'objets, liés à *x* donc via une association `@ToMany`. Si, par exemple un objet instance de `Film` est l'entité principale, le réalisateur (instance de `Artiste`) est une entité (secondaire) et les rôles du film constituent une collection.

Nous avons donc essentiellement deux stratégies de chargement, s'appliquant aux entités et collections :

— `Eager` (« glouton », « gourmand » en anglais) consiste à charger une entité ou une collection le plus tôt possible, indépendamment du fait que l'application les utilise ou non par la suite.
— `Lazy` (« paresseux » en anglais) indique au contraire qu'une entité ou une collection est chargée le plus tard possible, au moment où l'application cherche à y accéder.

Par défaut, JPA charge les entités avec une stratégie `Eager`, et les collections avec une stratégie `Lazy`. C'est cette stratégie qui est appliqué quand on utilise les annotations JPA, même quand le moteur implantant JPA est Hibernate.

Toujours sur le même exemple, avec la stratégie par défaut JPA, le réalisateur, le pays et le genre sont des entités chargées en mode `Eager` quand un film est chargé dans la session ; les rôles ne sont chargés que quand l'application tente de les récupérer avec la méthode *getRoles()*. Reprenez notre petite étude de cas en début de chapitre pour vérifier.

Note : Hibernate applique une stratégie par défaut différente de JPA : tout est chargé en mode `Lazy`.

Examinons maintenant *comment* ces stratégies sont implantées.

12.2.2 `Eager`, le chargement glouton

Dans ce mode, Hibernate va tenter d'associer le plus tôt possible les entités secondaires à l'entité principale. On peut constater en pratique que deux méthodes sont utilisées :

— *Par des jointures externes*. C'est notamment le cas quand on charge un objet par *load()* ou *get()*. Hibernate engendre alors une requête comprenant des `outer join`.
— *Par des requêtes SQL complémentaires*. Si Hibernate n'a pas pu anticiper en plaçant des `outer join`, des requêtes SQL sont effectuées, une par entité secondaire. C'est le cas par exemple quand l'entité principale est obtenue par une requête HQL, ou quand elle fait partie d'une collection chargée de manière paresseuse.

Le mode `Eager` ne se justifie que dans deux cas

— on est sûr que l'application accèdera toujours ou presque toujours aux entités secondaires quand une entité principale est chargée ;
— on est sûr que les entités secondaires sont dans un cache de premier ou de second niveau.

Il me semble difficile d'assurer qu'une application, pour toujours et en toutes circonstances, satisfaira une des deux conditions ci-dessus. Cette stragégie a le grave inconvénient d'engendrer parfois en grande quantité des requêtes SQL élémentaires, ramenant chacune une seule ligne. Je propose donc la recommandation suivante.

Recommandation : toujours adopter le mode `Lazy` par défaut

Au niveau de la configuration du *mapping*, je vous recommande d'appliquer systématiquement une stratégie `lazy` par défaut. Ce qui revient à compléter les annotations `@OneToOne` ou `@ManyToOne` comme suit :

```
@ManyToOne (fetch=FetchType.LAZY)
```

Pour les associations de type `@ToMany`, le mode par défaut est toujours `Lazy` et il est donc inutile de compléter l'annotation (mais faites-le si vous voulez privilégier la clarté).

Note : La documentation Hibernate parle de `Immediate` fetching, `Eager` fetching et `Join` fetching, de manière à vrai dire assez confuse et mélangeant le *quand* et le *comment*. Il semble suffisant de considérer qu'il n'y a que deux méthodes, gloutonne (*Eager*) ou paresseuse (`Lazy`), et pour chacune plusieurs implantations possibles (par requête indépendante, par `outer join`, etc.) selon les circonstances.

12.2.3 Le mode `Lazy`

Le mode `Lazy` est le mode par défaut pour les collections. Cela semble indispensable car la taille de la collection n'est pas connue à priori, et se mettre en mode `Eager` ferait courir le risque de charger un très grand nombre d'objets, sans garantie sur leur utilisation, et un coût d'initialisation et de stockage en mémoire élevé. Pensez par exemple à ce que donnerait le chargement de *tous* les films d'un pays comme les USA si on associait une collection `films` en mode `Eager` à la classe `Pays`.

Par défaut, toutes les collections devraient donc être en mode `Lazy`. Aucune solution n'étant idéale en toutes circonstances, ce mode peut donner lieu à un chargement inefficace, caractérisé par l'expression "1+n requêtes".

12.2.4 Le problème des 1+n requêtes

Supposions que vous vouliez parcourir tous les films de la base, et pour chacun analyser les notes reçues. La structure de votre action serait la suivante :

```
// On recherche les films
Set<Film> films = session.createQuery ("from Film");

for (Film film: films) {
  // Pour chaque film on traite les notes
  for (Notation notation: films.getNotation()) {
    // Faire qq chose avec la notation
  }
}
```

Nous avons deux boucles imbriquées. *Quelles sont les requêtes SQL engendrées* ? Il y en a une par boucle.

— La première sélectionne tous les films et ressemble simplement à :

```
select * from Film
```

— La seconde sélectionne les notes d'un film donné

```
select * from Notation where id_film=:film.id
```

Comment sont-elles exécutées ? La première est exécutée *une fois*, la seconde *autant de fois qu'il y a de films*, d'où la caractérisation par l'expression "1+n" requêtes. Le problème est que cette stratégie est inefficace car elle soumet potentiellement beaucoup de requêtes au SGBD. Si vous avez 10 000 films, vous exécuterez 10 000 fois la même requête, alors que les SGBD sont conçus, grâce à l'opération de jointure, pour ramener tous les objets en une seule requête.

On peut envisager plusieurs solutions au problème.

— en changeant la configuration pour accéder à la collection en mode `Eager` : comme je l'ai déjà signalé, utiliser une configuration générale pour résoudre un problème survenant dans un contexte spécifique ne fait que transposer le problème ailleurs ;
— en utilisant le *chargement par lot*, décrit ci-dessous ;
— en utilisant une requête HQL de chargement.

La troisième solution me semble de loin la meilleure. Pour votre culture Hibernate, voici une brève présentation du chargement par lot.

12.2.5 Le chargement par lot, une variante de `Lazy`

Le chargement par lot (*batch fetching*) permet de factoriser les requêtes effectuées pour obtenir les collections associées aux objets de la boucle extérieure (celle sur les films dans notre exemple). Il est caractérisé par la taille d'un lot, k, exprimant le nombre de collections recherchées en une seule fois. La syntaxe de l'annotation est la suivante :

```
@BatchSize(size=k)
```

Reprenons l'exemple de nos 10 000 films, avec les 10 000 requêtes cherchant la collection *roles* de chaque film. Avec un chargement par lot de taille 10, on va grouper la recherche des collections 10 par 10. Supposons pour simplifier que les identifiants des films sont séquentiels : 1, 2, 3, …., etc. Quand l'application cherche à accéder à la collection *roles* du *premier* film, la requête suivante sera effectuée :

```
select * from Notation where id_film in (1, 2, 3, 4, 5, 6, 7, 8 , 9, 10)
```

et les 10 collections des 10 premiers films seront initialisées. L'accès aux *roles* pour les films 2, 3, 4, …, 10 se fera donc dans le cache, sans avoir besoin d'effectuer une nouvelle requête SQL. L'accès aux roles du film 10 déclenchera la requête :

```
select * from Notation where id_film in (11, 12, 13, 14, 15, 16, 17, 18 , 19)
```

et ainsi de suite. Avec ce paramétrage, on est passé du problème des 1+n requêtes au problème des 1+n/10 requêtes !

C'est donc un mode intermédiaire entre `Eager` et `Lazy`. Incontestablement, il va amener une amélioration des performances de l'application. Cela peut être une solution simple et satisfaisante, même si elle repose sur un paramétrage dont la valeur n'est pas évidente à fixer.

Cela dit, on peut aisément faire la même critique : le paramétrage du lot dans la configuration n'est sans doute pas adapté à tous les contextes présents dans l'application. Dans la mesure où on peut, *localement*, dans un contexte donné, opter pour une stratégie `Eager`, pourquoi se priver de le faire, ce qui est à la fois simple, efficace et compréhensible ? Cela passe par l'utilisation de requêtes de chargement HQL.

Exercice : appliquez le chargement par lot

Ecrivez une action qui parcourt tous les films et affiche leur rôles : vous devriez constater l'effet des 1+n requêtes. Appliquer le chargement par lot pour limiter son impact, et consultez les requêtes SQL engendrées par Hibernate.

12.3 S3 : Charger un sous-graphe avec HQL

Supports complémentaires :

— Diapos pour la session « S3 : Charger un sous-graphe avec HQL »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35951199aqwzxqg/

Nous avons étudié HQL dans une optique de *sélection* de données à matérialiser. Dans cette section nous abordons l'option `fetch` qui indique que l'on veut charger les objets associés dans le cache. Si, par exemple, on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe.

12.3.1 L'option `fetch`

On ne peut pas utiliser `fetch` sans `left join`. Le `fetch` indique que les objets de la table associée doivent être placés dans le cache.

```
select film
from Film as film
left join fetch film.roles as role
```

Pour chaque film placé dans le cache, on initialise donc la collection des rôles.

12.3.2 Combiner requêtes de chargement et de sélection

Attention à ne pas combiner sans précaution une requête de chargement (avec `fetch`) et de sélection (avec `where`). Prenons la requête suivante : on veut les films dont un des rôles est « McClane ». Et on l'exprime de la manière suivante :

```
select film
from Film as film
left join fetch film.roles as role
where role.nom= 'McClane'
```

Le `fetch` indique que l'on veut charger dans le cache le graphe `Film-Role` *pour les lignes sélectionnées dans la base*. Mais ici nous avons un problème : comme nous avons exprimé une *restriction* avec la clause `where` sur les rôles, seuls les rôles dont le nom est McClane seront chargés. La mauvaise surprise est que quand nous naviguerons du film vers ses rôles, nous n'en trouverons qu'un, ce qui n'était pas l'objectif initial.

La solution est d'utiliser deux fois la table `Role`. La première occurence, sans restriction, apparaît dans le `left join fetch`, la seconde dans le `where`. Voici une expression qui me semble claire (d'autres sont possibles) :

```
from Film as film
left join fetch film.roles as role
where film in (select r2.pk.film from Role as r2 where nom= 'McClane')
```

Ce n'est pas très élégant, mais cela correspond à la satisfaction de deux besoins différents : *sélectionner* des données et les *charger* sous forme d'objet.

Résoudre les 1+n requêtes avec HQL

Reprenons le problème des 1+n requêtes. Il serait préférable d'effectuer la jointure suivante en lieu et place de la requête SQL qui sélectionne simplement les films :

```
select * from Film as film, Notation as note
where film.id=note.id_film
```

Il se trouve que nous savons maintenant obtenir cette jointure *et le chargement du sous-graphe Film-Notation* avec la requête HQL suivante :

```
select distinct film from Film as film left join fetch film.notations
```

En utilisant cette expression HQL, les notations seront placées dans le cache, et la navigation sur les notations pour chaque film ne nécessitera plus de l'exécution de la seconde requête SQL.

```
// On recherche les films
Set<Film> films = session.createQuery ("select distinct film from Film as film "
                                       + "left join fetch film.notations");

for (Film film: films) {
  // Pour chaque film on traite les notes
  for (Notation notation: films.getNotation()) {
    // Faire qq chose avec la notation
  }
}
```

Exercice : testez la solution au problème des 1+n requêtes

Reprenez l'action qui affiche un film et ses rôles. Utilisez la requête HQL *de chargement* appropriée pour avoir une seule requête et prendre tous les objets dans le cache.

Exercice : minimisation des requêtes SQL

Prenez l'application de votre projet, qui devrait consister en un ensemble d'actions consistant, chacune, à naviguer dans une partie du graphe d'objet. Le but est de minimiser le nombre de requêtes SQL, si possible avec une seule requête par action. À vous de jouer.

Exercice (difficile) : validation des performances

Nous prenons jusqu'à présent l'hypothèse implicite qu'il faut minimiser le nombre de requêtes SQL. Pour vérifier que c'est justifié, nous pouvons explorer les deux options suivantes :

— Charger une base de données volumineuse, mesurer les temps de réponse avec la configuration par défaut, puis notre configuration optimisée, au moins pour quelques requêtes importantes.
— Etudiez le plan d'exécution de MySQL pour quelques requêtes et vérifier qu'il est optimal.

Cet exercice demande un travail conséquent et constitue un complément significatif au projet. À vous de voir si vous êtes intéressé.

Pour charger une base de données avec beaucoup de lignes, vous pouvez vous reporter au site http://www.generatedata.com/.

12.4 Résumé : savoir et retenir

La question du chargement des objets par des requêtes SQL est importante et malheureusement assez complexe pour des raisons déjà évoquées : changements d'une version à une autre, confusion de la documentation, différences JPA/Hibernate, et aussi tout simplement la complexité des options disponibles. Sachez que j'ai essayé de vous dissimuler certains aspects (les *proxy*, les *wrappers*) dont le rôle est peu clair et qui ne sont sans doute pas indispensables à la bonne gestion du problème. Voici un résumé de mes recommandations, pour une approche qui me semble solide et gérable sur le long terme.

- Dans la configuration, déclarez toutes vos associations en `Lazy`.
- Configurez la session pour produire l'affichage des requêtes SQL.
- Explorez votre application systématiquement pour analyser les requêtes SQL qui sont produites ; dans une application MVC c'est assez simple : exécutez chaque action et regardez quelle(s) requête(s) SQL est/sont produite(s).
- Si vous constatez trop de requêtes : remplacez les opérations de navigation par une requête HQL initiale qui charge les objets auxquels l'action va accéder ; si possible réduisez à une seule requête SQL produite par action.
- Si vous constatez qu'une ou plusieurs requêtes SQL paraîssent trop complexes, produisez son plan d'exécution, et modifiez éventuellement vos expressions HQL pour la décomposer.

Le bon réglage des requêtes SQL est un art qui implique une très bonne compréhension de l'exécution des requêtes dans un système relationnel, l'utilisation des index, des algorithmes de jointure, de tri, des ressources mémoire, etc. Si vous n'êtes pas spécialiste, l'aide d'un DBA pour optimiser votre application peut s'avérer nécessaire.

CHAPITRE 13

Dynamique des objets persistants

Jusqu'à présent nous avons travaillé sur une base de données pré-existante, et nous n'avons donc effectué aucune *mise à jour* avec JPA/Hibernate. En soi, il n'y a rien de beaucoup plus compliqué que pour les recherches : étant donné un objet *persistant*, on modifie une de ses propriétés avec un des « setteurs ». La session Hibernate, qui surveille cet objet placé dans son cache, va alors le marquer comme étant à modifier, et engendrera la requête `update` au moment approprié.

La problématique abordée dans ce chapitre est un peu plus générale et aborde le *cycle de vie* des objets de notre application. Un même objet peut en fait changer de *statut*. Le statut « persistant » indique une synchronisation avec la base de données pour préserver les modifications de l'état de l'objet. Mais cet objet peut également être (ou devenir) *transient* ou *détaché*. À quel moment un objet devient-il persistant ? Quand cesse-t-il de l'être ? Comment Hibernate détecte-t-il l'apparition d'un objet persistant et gère-t-il les mises à jour ? La première session considère ces problématiques de mises à jour affectant un seul objet, et introduit la notion de *transaction* dans JPA/Hibernate.

Ces questions peuvent devenir relativement complexes quand on ne considère plus un seul objet, mais un graphe dont le comportement doit obéir à une certaine cohérence. Il faut alors définir des comportements répercutant, « en cascade », les mises à jour d'un objet pour qu'elles affectent également les objets dépendants et connexes dans le graphe. Ce sujet est traité dans la seconde session.

13.1 S1 : Objets transients, persistants, et détachés

Supports complémentaires :

- Diapos pour la session « S1 : Objets transients, persistants, et détachés »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35951226gpyblcw/

Commençons par la notion de base, celle de *transaction*. Rappelons qu'une transaction est une séquence d'accès (lectures ou mises à jour) à la base de données qui satisfait 4 propriétés, souvent résumées par l'acronyme ACID.

- A comme Atomicité : les accès d'une même transaction forment un tout solidaire ; ils sont validés ensemble ou annulés ensemble.
- C comme cohérence : une transaction doit permettre de passer d'un état cohérent de la base à un autre état cohérent.
- I comme isolation : une transaction est totalement isolée des transactions concurrentes qui s'exécutent en même temps.

— D comme durabilité : avant la validation (*commit*), les mises à jour d'une transaction sont invisibles par toute autre transaction, après, ils deviennent visibles et définitifs.

Ce sont des notions de base que je vous invite à réviser si elles ne sont pas claires. Hibernate fournit une interface `Transaction` qui encapsule la communication avec deux types de systèmes transactionnels : les transactions JDBC, et les transactions JTA (*Java Transaction API*). Le second type de système est réservé à des applications complexes ayant besoin de recourir à un gestionnaire de transactions réparties. Dans notre cas nous ne considérons que les transactions JDBC.

Hibernate fonctionne toujours en mode `autocommit=off`, ce qui est recommandé pour éviter de valider aveuglément toutes les requêtes. D'une manière générale, toute série de requêtes / mises à jour effectuée avec Hibernate devrait être intégrée à une *transaction* bien identifiée.

13.1.1 L'interface `Transaction`

L'interface `Transaction` est donc en charge d'interagir avec la base pour initier, soumettre et valider des transactions. Cette interface peut lever des exceptions qui doivent être soigneusement traitées pour éviter de se retrouver dans des états incohérents après l'éventuel échec d'une opération. La *pattern* standard de gestion des exceptions est illustré ci-dessous.

```
// 1 - Instanciation d'un objet
        Pays pays = new Pays();
        pays.setCode("is");
        pays.setLangue("Islandais");
        pays.setNom("Islande");

// 2 - cet objet devient persistant
        Transaction tx = null;
        try {
                tx = session.beginTransaction();
                session.save(pays);
                tx.commit();
        } catch (RuntimeException e) {
                if (tx != null)
                        tx.rollback();
                throw e; // Gérer le message (log, affichage, etc.)
        } finally {
                session.close();
        }

// 3 - Ici, l'objet 'pays' est détaché de la session !
```

On marque donc le début d'une transaction avec `beginTransaction()` et la fin soit avec `commit()`, soit avec `rollback()`. Entre les deux, Hibernate charge des objets persistants dans le cache, et surveille toute modification qui leur est apportée par l'application. Un objet modifié est marqué comme *dirty* et sera mis à jour dans la base par une requête `update` (ou `insert` pour une création) le moment venu, c'est-à-dire :

— à la fin de la transaction, sur un `commit()` ;
— au moment d'un appel explicite à la méthode *flush()* de la session (vous ne devriez pas avoir besoin de le faire) ;
— quand Hibernate estime nécessaire de synchroniser le cache avec la base de données.

Le dernier cas correspond à une situation où des données ont été modifiées dans le cache, et où une requête ramenant ces données depuis la base est soumise. Dans ce cas, Hibernate peut estimer nécessaire de synchroniser au préalable le cache avec la base (avec un *flush()*) pour assurer la cohérence entre le résultat de la requête et le cache. Attention : *synchroniser* (effectuer les requêtes de mise à jour) ne veut pas dire *valider* : il reste possible d'effectuer un `rollback()` ramenant la base à son état initial.

Le code précédent montre la gestion des exceptions. Il y a deux choses à faire impérativement :

— annuler l'ensemble de la transaction en cours par un `rollback` ;
— fermer la session : une exception peut signifier que le cache n'est plus en phase avec la base de données, et toute poursuite de l'activité peut mener à des résultats imprévisibles.

La fermeture de la session implique celle de la connexion JDBC, et la suppression de tous les objets persistants situés dans le cache. La base elle-même est dans un état cohérent garanti par le `rollback`. Donc tout va bien.

Note : Le code montré ci-dessus est un condensé des actions à effectuer dans le cadre d'une session. Dans une application, vous devez bien entendu gérer les exceptions de manière plus systématique.

13.1.2 Statut des objets Hibernate

En examinant le code qui précède, on constate qu'un même objet (*pays*) n'est pas uniformément *persistant*. Il passe en fait par trois statuts successifs.

— jusqu'au début de la transaction (et très précisément jusqu'à l'appel à *save()*), l'objet est *transient* : c'est un objet java standard géré par le *garbage manager* ;
— après l'appel à *save()*, l'objet est placé sous le contrôle de la session qui va observer (par des mécanismes d'inspection java) tous les événements qui affectent son état, et synchroniser cet état avec la base : le statut est *persistant* ;
— enfin, après la fermeture de la session, l'objet *pays* existe toujours (puisque l'application maintient une référence) mais il n'est plus synchronisé avec la base : son statut est dit *détaché* (sous-entendu, de la session).

On peut constater que le statut Hibernate d'un objet peut être géré de manière complètement indépendante des services fonctionnels qu'il fournit à l'application. En d'autre termes, cette dernière n'a pas à se soucier de savoir si l'objet qu'elle manipule est persistant, transient ou détaché. Pensez à l'inconvénient qu'il y aurait à devoir insérer en base un objet alors que l'on veut simplement faire appel à ses méthodes. Pensez inversement à l'intérêt de pouvoir utiliser (par exemple dans des tests unitaires) des objets sans les synchroniser avec la base de données.

Le statut d'un objet change par l'intermédiaire d'interactions avec la session Hibernate. La figure *Cycle de vie des objets du point de vue JPA/Hibernate* résume ces interactions et les changements de statut qu'eles entrainent.

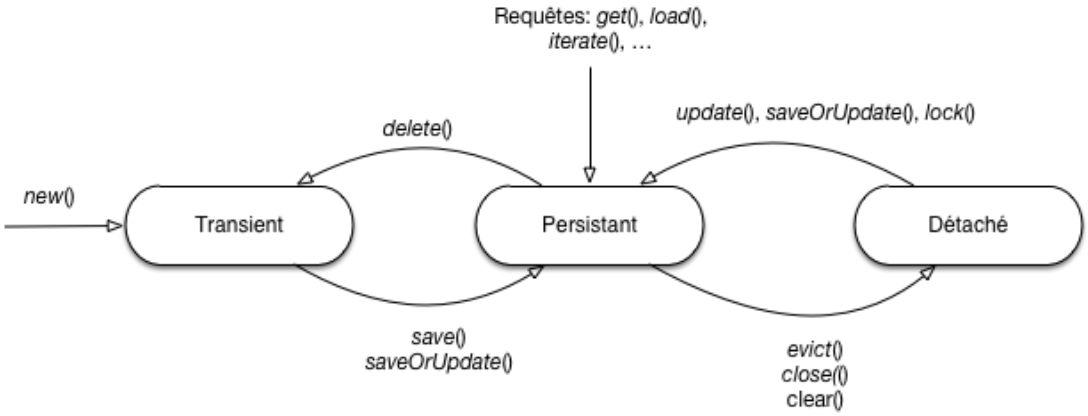

Fig. 1 – Cycle de vie des objets du point de vue JPA/Hibernate

Le statut « persistant » est central dans la figure. On constate qu'il existe essentiellement trois méthodes pour qu'un objet *devienne* persistant :

— un objet transient, instancié avec *new()*, est associé à la session par *save()* ou *saveOrUpdate()* ; c'est l'exemple du code ci-dessus ;

— une ligne de la base de données, sélectionnée par une requête ou une opération de navigation, est *mappée* en objet persistant ;
— un objet *détaché* est « ré-attaché » à une session.

Voici des détails sur chaque statut.

13.1.3 Objets persistants

Un objet persistant est une instance d'une classe *mappée* qui est associée à une session par l'une des méthodes mentionnées précédemment. Par définition, un objet persistant est *synchronisé* avec la base de données : il existe une ligne dans une table qui stocke les propriétés. la session surveille l'objet, détecte tout changement dans son état, et ces changements sont reportés automatiquement sur la ligne associée par des requêtes `insert`, `delete` ou `update` selon le cas. Le moment où ce report s'effectue est choisi par Hibernate. Au plus tard, c'est à l'appel du *commit()*.

L'association à une ligne signifie qu'un objet persistant a également la propriété de disposer d'un identifiant de base de données (celui, donc, de la ligne correspondante). Le mode d'acquisition de la valeur pour cet identifiant explique les différentes méthodes disponibles.

Dans le cas le plus simple, l'identifiant est engendré par une séquence. Quand on instancie un objet avec *new()* et que l'on appelle la méthode *save()*. Hibernate va détecter que l'objet est nouvellement instancié et ne dispose par d'identifiant. Un appel `insert` à la base va créer la ligne correspondante, et lui affecter une valeur d'identifiant auto-générée.

Si l'identifiant n'est pas auto-généré par une séquence, il doit être fourni par l'application (c'est le cas dans notre exemple pour le pays, identifié par son code). L'exécution de l'"`insert` risque alors d'être rejetée si une ligne avec le même identifiant existe déjà. Si, donc, on veut rendre persistant un objet *dont l'identifiant est déjà connu*, il est préférable d'appeler *saveOrUpdate()*. Hibernate déclenchera alors, selon le case un `insert` ou un `update`.

La destruction d'un objet persistant avec la méthode *delete()* implique d'une part son passage au statut d'objet transient, et d'autre part l'effacement de la ligne correspondante dans la base (par un `delete`).

13.1.4 Objets détachés

La différence entre un objet *transient* et un objet *détaché* est que ce dernier a été, à un moment donné, associé à une ligne de la base. On peut donc *affirmer* que l'objet transient dispose d'une valeur d'identifiant. On peut aussi *supposer* que la ligne dans la base existe toujours, mais ce n'est pas garanti puisqu'un objet détaché, du fait de la fermeture de la session, n'est plus synchronisé à la base.

Dans ces conditions, la méthode *saveOrUpdate()* s'impose naturellement pour les objets détachés qui sont ré-injectés dans une session. Hibernate effectue un `update` de la ligne existante, ou un `insert` si la ligne n'existe pas.

Les deux autres méthodes pour rendre persistant un objet détaché se comportent différemment.

— la méthode *update()* indique à Hibernate qu'une synchronisation immédiate avec la commande SQL `update` doit être effectuée ; c'est nécessaire quand l'objet détaché a été modifié hors de toute session, et que son état n'a donc pas été synchronisé avec la base.
— la méthode *lock()* associe l'objet détaché à la session (il devient donc persistant) mais les changements d'état intervenus *pendant* le détachement ne sont pas reportés dans la base.

Voilà l'essentiel de ce que vous devez savoir pour comprendre et gérer *individuellement* le statut des objets. La prochaine session va se pencher sur le cas ou des modifications affectent des ensembles d'objets, et plus particulièrement des objets connexes dans le graphe, comme par exemple un film et l'ensemble de ses acteurs. En suivant l'approche présentée jusqu'à présent, il faudrait appeler *save()* ou *saveOrUpdate()* sur *chaque* objet créé ou modifié, ce qui n'est ni naturel, ni élégant.

Exercice : une transaction insérant un graphe d'objets

Créez une transaction qui insère un film, son metteur en scène et ses acteurs. Voici un bout de code ci-dessous, que vous pouvez compléter en insérant les acteurs Georges Clooney et Sandra Bullock.

```
Film gravity = new Film();
gravity.setTitre("Gravity");
gravity.setAnnee(2013);

Genre genre = new Genre();
genre.setCode("Science-fiction");
gravity.setGenre(genre);

// Alfonso Cuaron a réalisé Gravity
Artiste cuaron = new Artiste();
cuaron.setPrenom("Alfonso");
cuaron.setNom("Cuaron");
cuaron.addFilmsRealise(gravity);

// Ajoutez les acteurs...

// Sauvegardons dans la base
session.save(gravity);
```

Que se passe-t-il à l'exécution ? Comment l'expliquer ? Comment corriger le problème ?

Hibernate permet de spécifier une propagation des commandes de persistance dans un graphe d'objet, ce qui est plus élégant et évite des erreurs potentielles. Le sujet est abordé dans la seconde session.

13.2 S2 : Persistance transitive

Supports complémentaires :

- Diapos pour la session « S2 : Persistance transitive »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f359512cdh5tip39/

Si vous avez effectué l'exercice qui précède, vous avez constaté une occurrence du problème général suivant : si je crée, avec mon application, un graphe d'objet connecté, les instructions rendant ces objets persistants doivent préserver la *cohérence* de ce graphe. En d'autres termes, on ne peut pas rendre certains objets persistants et d'autres transients sous peine d'aboutir à une version incohérente de la base.

Un des rôles de la transaction est d'assurer la *cohérence (transactionnelle)* des commandes de mise à jour. C'est le « C » dans le fameux acronyme ACID. Assurer cette cohérence « manuellement » en gérant les situations au cas par cas est source d'erreur et de programmation répétitive. Avec JPA/Hibernate, on peut introduire dans le modèle de données une spécification des contraintes de cohérences, qui sont alors automatiquement appliquées.

13.2.1 Cohérence transactionnelle

La figure *Un graphe d'objet à rendre persistant* montre le *graphe d'objet* de notre film. Si, dans ce graphe d'objet, une partie seulement est rendu persistante (par exemple le film, en grisé), alors il est clair que la représentation en base de données sera incomplète. Au niveau de la base elle-même, les contraintes d'intégrité référentielle (décrites par les clés étrangères) ne seront pas respectées, et le SGBD rejettera la transaction. Mais avant même cela, Hibernate détectera que la méthode *save()* est appliquée à un objet transient qui référence d'autres objets transients, avec risque potentiel d'incohérence. Une exception sera donc levée avant même la tentative d'insertion en base.

La solution « manuelle » (que vous avez sans doute mise en application dans l'exercice de la session précédente) consiste à appliquer la méthode *save()* sur *chaque* objet du graphe, *et ce dans un ordre bien défini* pour éviter de rendre persistant un objet référençant un objet transient.

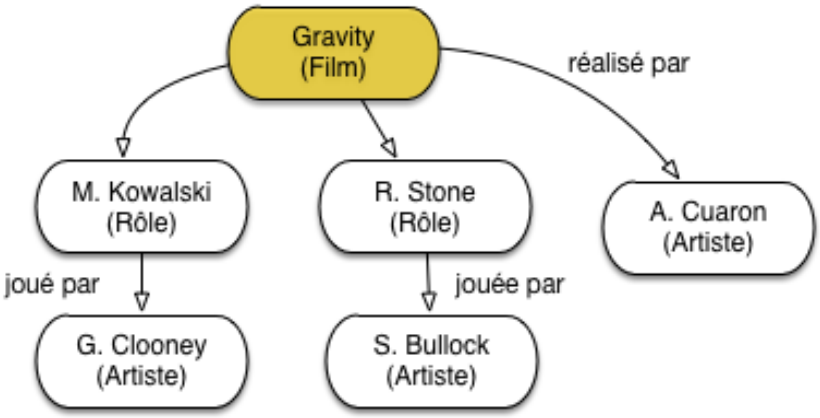

FIG. 2 – Un graphe d'objet à rendre persistant

Note : Dans les bases objets (peu répandues), cette cohérence transactionnelle est prise en charge par le SGBD qui applique un principe dit de *persistance par atteignabilité*. Ce principe n'existe pas dans les SGBD relationnelles, la notion la plus proche étant celle d'intégrité référentielle.

L'option `cascade` sert à spécifier la prise en charge par Hibernate de la cohérence transactionnelle.

13.2.2 L'option cascade

La propagation « en cascade » des instructions de persistance dans un graphe d'objet est spécifiée par une annotation `@Cascade`.

Important : *Dans ce qui suit nous utilisons les options de JPA, et pas celles d'Hibernate*. Attention, il semble que JPA et Hibernate soient partiellement incompatibles sur ce point, donc faites bien attention aux *packages* que vous importez (pour JPA, c'est `javax.persistence.*`).

De plus, la méthode *save()* de la session Hibernate semble ne pas reconnaitre les annotations JPA, alors que cela fonctionne bien pour la méthode *persist()* que nous utilisons donc. Ces problèmes seront probablement réglés dans une prochaine version.

Voici donc simplement comment on indique que le réalisateur doit être rendu persistant quand le film est lui-même rendu persistant (classe *Film.java*).

```
@ManyToOne (fetch=FetchType.LAZY, cascade=CascadeType.PERSIST)
@JoinColumn(name = "id_realisateur")
private Artiste realisateur;
```

Si vous appelez maintenant la méthode *session.persist(gravity)*, la persistance s'appliquera par transitivité au metteur en scène (essayez).

Les opérations de persistance à « cascader » sont les créations, modifications et destructions, et peuvent se placer des deux côtés de l'association. Voici les annotations (JPA) correspondantes, énumérées par `CascadeType`.

— `PERSIST`. C'est l'exemple montré ci-dessus. Quand on appelle la méthode *persist()* sur l'objet référençant (le film dans notre exemple), l'objet référencé (l'artiste dans notre exemple) devient également persistant par appel à la méthode *persist()*, qui est donc éventuellement propagée transitivement à son tour.

— `REMOVE`. Un appel à *remove()* sur l'objet référençant est propagé à l'objet référencé.
Attention à bien considérer cette option. Sur notre schéma, la suppression d'un film *ne devrait pas* entraîner la suppression du réalisateur (vous êtes bien d'accord ?). En revanche la suppression d'un artiste peut entraîner celle des films qu'il/elle a réalisés (à décider à tête reposée), et entraîne certainement celle des rôles qu'il/elle a joués.
— `MERGE`. La méthode *merge()* JPA est équivalente à *saveOrUpdate()* en Hibernate. Cette option de cascade indique donc une propagation des opérations d'insertion *ou* de mise à jour sur les objets référencés.
— `REFRESH`. La méthode *refresh()* synchronise l'objet par lecture de son état dans la base. Cette option de cascade indique donc une propagation de l'opération sur les objets référencés.
— `ALL`. Couvre l'ensemble des opérations précédentes.

N'utilisez surtout pas `CascadeType.ALL` aveuglément ! La persistance transitive complète s'impose pour les associations de nature compositionnelle (par exemple une commande et ses lignes de commande), dans lesquelles le sort d'un composé est étroitement lié à celui du composant (mais l'inverse n'est pas vrai !). Elle est aussi normale pour les entités qui représentent (conceptuellement) une association plusieurs-plusieurs dans le modèle de l'application. Dans notre schéma, c'est le cas pour les rôles par exemple, dont la préservation n'a pas de sens dès lors que le film ou l'acteur est supprimé. Dans toutes les autres situations, une réflexion au cas par cas s'impose pour éviter de déclencher automatiquement des opérations non souhaitées (et surtout des destructions).

Notez qu'il est possible de combiner plusieurs options en les plaçant entre accolades :

```
cascade = {CascadeType.PERSIST, CASCADE.MERGE}
```

Vous voilà équipés pour spécifier la persistance transitive avec JPA/Hibernate.

Exercice : cohérence transactionnelle pour le graphe d'un film

Ajoutez les options `cascade` à votre modèle pour que la persistance d'un film entraîne automatiquement celle des objets liés. Vérifiez que cela fonctionne en insérant les objets de la figure *Un graphe d'objet à rendre persistant*.

13.3 Résumé : savoir et retenir

La question des mises à jour est souvent plus délicate que celle des recherches, tout simplement parce que son impact sur la base de données est plus grand. Retenez :

— que toute mise à jour s'effectue dans le contexte d'une *transaction*, qui s'exécute de manière *atomique* (tout est validé, ou rien) ;
— une transaction peut lever des exceptions qui risquent de laisser la couche ORM dans un état incohérent (synchronisation incomplète entre la base et la session) : toute levée d'exception doit entraîner la fermeture de la session courante ;
— les objets passent par divers statuts, distingués techniquement par la présence d'une valeur pour l'identifiant ; le passage au statut *persistant* assure la synchronisation avec la base : essayez de produire un code dans lequel ce statut est clair à chaque instant ;
— la persistance d'un graphe d'objet peut s'obtenir avec des annotations `@Cascade`, dont l'option plus simple et la plus sûre à manipuler est `PERSIST` ; les autres options sont à considérer avec précaution et en connaissance de cause.

Nous n'en avons pas fini avec les transactions, considérées jusqu'à présent de manière isolée, alors qu'elle s'exécutent souvent en *concurrence*. C'est le sujet du prochain chapitre.

CHAPITRE 14

Applications concurrentes

La notion de transaction est classique dans le domaine des bases de données, et très liée à celle de *contrôle de concurrence*. Tous les SGBD implantent un système de contrôle, et la première section de ce chapitre propose quelques rappels et un exercice sur les niveaux d'isolation pour un rafraichissement (probablement utile) de vos connaissances.

Hibernate ne ré-implante pas un protocole de contrôle transactionnel qui serait redondant avec celui du SGBD. En première approche, une application Hibernate est donc simplement une application standard qui communique avec la base de données, éventuellement en concurrence avec d'autres applications, et lui soumet des transactions. Là où Hibernate mérite une développement spécifique, c'est dans le traitement de transactions « longues », particulièrement utiles dans le cadre d'une application Web.

14.1 S1 : Rappels sur la concurrence d'accès

Supports complémentaires :

- Diapos pour la session « S1 : Rappels sur la concurrence d'accès »
- Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f35951341n951v7h/

La première chose que je vous invite à faire est de lire soigneusement le chapitre d"introduction à la concurrence d'accès de mon cours de bases de données. Pendant la séance de cours, je donne une démonstration des principales caractéristiques pratiques de la concurrence d'accès et des niveaux d'isolation. Vous devriez être capable de reproduire cette démonstration (en comprenant ce qui se passe), et donc d'effectuer l'exercice suivant.

Exercice : simuler des transactions en ligne de commande

Pour cet exercice, vous allez étendre la base de données *Films* en ajoutant des *séances* diffusant des films, et en autorisant un internaute à effectuer la *réservation* d'un certain nombre de places pour une séance. Les tables *Seance* et *Reservation* existent déjà dans la base. Je vous laisse faire le *mapping* JPA pour ces deux entités.

Pour l'instant, on n'utilise pas Hibernate, mais deux terminaux connectés à MySQL, représentant deux sessions concurrentes. La commande `prompt` sous l'interpréteur MySQL permet de caractériser chaque session.

Sur la base de ma démonstration (pour ceux qui suivent le cours) ou du chapitre cité ci-dessus, reproduisez les situations suivantes :

— en mode `read uncommitted` : lecture sale, menant une des transactions à connaître une mise à jour de l'autre transaction, alors que cette dernière fait un `rollback` ;
— en mode `read committed` : des mises à jour perdues, résultant en un état de la base incohérent ;
— faire la même exécution concurrente en `repeatable read` : peut-on s'attendre à un changement ?
— en mode `serializable` : un interblocage.

14.2 S2 : Gestion de la concurrence avec Hibernate

Supports complémentaires :

— Diapos pour la session « S2 : Gestion de la concurrence avec Hibernate »
— Vidéo associée : https://mediaserver.cnam.fr/permalink/v125f359513b2m7pz6xh/

Comme expliqué au début du chapitre, les transactions concurrentes sont une source vicieuse d'anomalies, et il est important d'en être pleinement conscient dans l'écriture de l'application.

14.2.1 Gestion des niveaux d'isolation

Hibernate s'appuie sur le niveau d'isolation fourni par le SGBD, et ne tente pas de ré-implanter des protocoles de concurrence. Le mode par défaut dans la plupart des systèmes est `read committed` ou `repeatable read`. La première partie de ce chapitre doit vous avoir convaincu que ce niveau d'isolation (par défaut) n'offre pas toutes les garanties et qu'il est indipensable de se poser sérieusement la question des risques liés à la concurrence d'accès. Dans une application transactionnelle, le mode `read uncommitted` ne devrait même pas être envisagé. Il nous reste donc comme possibilités :

— d'accepter le mode par défaut, après évaluation des risques ;
— ou de passer en mode `serializable`, en acceptant les conséquences (performances moindres, risques de *deadlock*) ;
— ou, enfin, d'introduire une dose de gestion « manuelle » de la concurrence en effectuant des verrouillages préventifs (dits, parfois, « verrouillage pessimiste »).

La configuration Hibernate permet de spécifier le mode d'isolation choisi pour une application :

```
hibernate.connection.isolation = <val>
```

où `val` est un des codes suivants :

— 1 pour `read uncommitted`
— 2 pour `read committed`
— 3 pour `repeatable read`
— 4 pour `serializable`

Vous pouvez donc spécifier un niveau d'isolation, qui s'applique à *toutes* les sessions, même celles qui ne présentent pas de risque transactionnel. Dans ces conditions, choisir le niveau maximal (`serializable`) systématiquement représente une pénalité certaine. Il semble préférable d'utiliser le niveau d'isolation par défaut du SGBD, et de changer le mode d'isolation à `serializable` ponctuellement pour les transactions sensibles.

Il ne semble malheureusement pas possible, avec Hibernate, d'affecter simplement le niveau d'isolation pour une session. On en est donc réduit à passer par l'objet `Connexion` de JDBC. Le code est le suivant (ça ne s'invente pas…).

```
session.doWork(
    new Work() {
            @Override
            public void execute(Connection connection) throws SQLException {
                connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            }
        });
```

Le mode `serializable`, malgré ses inconvénients (apparition d'interblocages) est le plus sûr pour garantir l'apparition d'incohérences dans la base, dont la cause est très difficile à identifier. Une autre solution, décrite ci-dessous, consiste à verrouiller explicitement, au moment de leur mise en cache, les objets que l'on va modifier.

14.2.2 Verrouillage avec Hibernate

Tout d'abord, précisons qu'Hibernate se contente de transmettre les requêtes de verrouillage au SGBD : aucun verrou en mémoire n'est posé (cela n'a pas de sens car chaque application ayant son propre cache, un verrou n'a aucun effet concurrentiel).

La principale utilité des verrous est de gérer le cas de la *lecture* d'une ligne/objet, suivie de *l'écriture* de cette ligne. Par défaut, la lecture pose un verrou *partagé* qui n'empêche pas d'autres transactions de lire à leur tour. Reportez-vous au cours sur la concurrence (début du chapitre) et à l'exemple-prototype des *mises à jour perdues* pour comprendre comment ces lectures posent problème quand une écriture survient ensuite.

Le principe du verrouillage explicite est donc d'effectuer une lecture qui *anticipe* l'écriture qui va suivre. C'est l'effet de la clause `for update`.

```
select * from xxx where .... for update
```

Le `for update` déclare que les lignes sélectionnées vont être modifiées ensuite. Pour éviter qu'une autre transaction n'accède à la ligne entre la lecture et l'écriture, le système va alors poser un *verrou exclusif*. Le risque de mise à jour perdue disparaît.

Avec Hibernate, l'équivalent du `for update` est la pose d'un verrou au moment de la lecture.

```
Transaction tx = session.beginTransaction();
Film film = session.get (Film.class, filmId, LockMode.UPGRADE);

film.titre = ... ; // Mises à jour

tx.commit();
```

L'énumération `LockMode` implique l'envoi d'une requête au SGBD avec une demande de verrouillage. Les valeurs possibles sont :

- `LockMode.NONE`. Pas d'accès à la base pour verrouiller (mode par défaut).
- `LockMode.READ`. Lecture pour vérifier que l'objet en cache est synchronisé avec la base.
- `LockMode.UPGRADE`. Pose d'un verrou exclusif sur la ligne.
- `LockMode.WRITE`. Utilisé en interne.

Gérer la concurrence dans le code d'une application est une opération lourde et peu fiable. Je vous conseille de vous limiter au principe simple suivant : quand vous lisez une ligne que vous allez modifier ensuite, placer un verrou avec `LockMode.UPGRADE`. Pour tous les autres cas, n'utilisez pas les autres modes et laissez le SGBD appliquer son protocole de concurrence.

Exercice : réserver des séances de cinéma

Cet exercice consiste à implanter une fonction de réservation de places pour une séance qui soit préservée des mauvais effets de la concurrence d'accès. Vous devez implanter deux actions :

- la première affiche les séances, en montrant la capacité de chaque salle et le nombre de places restantes ; un petit menu déroulant doit permettre à l'utilisateur de choisir de réserver de 1 à 10 places ; un bouton « Réservation » permet de demander la réservation de ces places ;
- la seconde action doit effectuer la réservation.

Bien entendu il est essentiel de garantir l'intégrité de la base : on ne doit pas accepter plus de réservations que la capacité de la salle ne le permet. À vous de voir quelle technique de gestion de la concurrence vous devez appliquer (il

doit être clair maintenant que le mode par défaut présente des risques).

14.3 S3 : Transactions applicatives

Supports complémentaires :
- Diapos pour la session « S3 : Transactions applicatives »
- Vidéo associée : à venir

Dans le cadre d'une application Web (ou d'une application interactive en général), la notion de « transaction » pour un utilisateur prend souvent la forme d'une séquence d'actions : affichage d'un formulaire, saisie, soumission, nouvelle saisie, confirmation, etc. On dépasse donc bien souvent le simple cadre d'un ensemble de mises à jour soumises à un moment donné par une action pour voir une « transaction » comme un séquence d'interactions (typiquement une séquence de requêtes HTTP dans le cadre d'une application Web).

Un exemple typique (que je vous laisserai implanter à titre d'exercice à la fin de cette section) est l'affichage des valeurs d'un objet, avec possibilité pour l'utilisateur d'effectuer des modifications en fonction de l'état courant de l'objet. Par exemple, on affiche un film et on laisse l'utilisateur éditer son résumé, ou les commentaires, etc.

Les transactions vues jusqu'à présent (que nous appellerons *transactions en base*) s'exécutent dans le cadre d'une unique action, sans intervention de l'utilisateur. Nous allons maintenant voir comment gérer des *transactions applicatives* couvrant plusieurs requêtes HTTP. Disons tout de suite qu'il est exclu de *verrouiller* des objets en attente d'une action utilisateur qui peut ne jamais venir. La stratégie à appliquer (pour laquelle Hibernate fournit un certain support) est une stratégie dite de *contrôle optimiste* : on vérifie, au moment des mises à jour, qu'il y n'a pas eu d'interaction concurrentielle potentiellement dangereuse.

Note : Pour ceux qui ont suivi le cours de base de données dans son intégralité, l'algorithme de concurrence correspondant est dit « contrôle multiversions ».

Les transactions applicatives sont appelées également *transactions longues*, *transactions métier*, ou *transactions utilisateurs*. Elles supposent une mise en place au niveau du code de l'application, ce qui peut être assez lourd. La méthode et ses différentes variantes sont de toute façon bonnes à connaître, donc allons-y.

14.3.1 Les stratégies possibles

Reprenons l'exemple de notre édition du résumé d'un film : l'utilisateur consulte le résumé courant, le complète ou le modifie, et valide. Dans une situation concurrentielle, deux utilisateurs A et B éditent simultanément le film dans sa version f. Le premier valide une nouvelle version f_1 et le second une version f_2. Les trois phases de l'édition sont illustrées par la figure *Mécanisme d'une transaction applicative : qui l'emporte ?*.

La question est *quelle mise à jour l'emporte ?* On peut envisager trois possibilités :
- *Le dernier* `commit` *l'emporte*. Si B valide après A, la mise à jour de A (f_1) est perdue, et ce même si A a reçu un message de confirmation…
- *Le premier* `commit` *l'emporte*. Si B valide après A, il reçoit une message indiquant que la version f a déjà été modifiée, et qu'il faut resoumettre sa modification sur la base de f_1.

La seconde solution admet plusieurs variantes. On peut par exemple imaginer une tentative automatique de fusion des deux mises à jour. Toujours est-il qu'elle paraît préférable à la première solution qui efface sans prévenir une action validée. Vous aurez peut-être reconnu dans la seconde stratégie le principe de base du contrôle de concurrence multiversions.

Pour l'implanter, on peut s'appuyer sur une gestion automatique de versions incrémentées fournie par Hibernate.

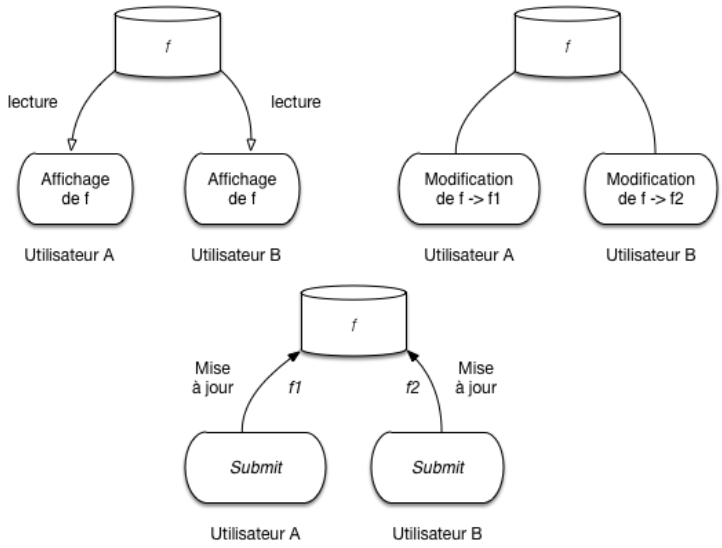

FIG. 1 – Mécanisme d'une transaction applicative : *qui l'emporte* ?

14.3.2 Versionnement avec Hibernate

Hibernate (JPA) permet la maintenance automatique d'un numéro de version associé à chaque ligne. Il faut ajouter une colonne à la table à laquelle s'applique le contrôle multiversions, et la déclarer dans l'entité de la manière suivante :

```
@Version
public long version;
```

Ici, la colonne s'appelle `version` (même nom que la propriété) mais on est libre de lui donner n'importe quel nom. On a choisi d'utiliser un compteur séquentiel, mais Hibernate propose également la gestion d'estampilles temporelles (marquage par le moment de la mise à jour).

L'interprétation de cette annotation est la suivante : Hibernate détecte toute mise à jour d'un objet le rendant « *dirty* ». Cela inclut la modification d'une propriété ou d'une collection. La requête `update` engendrée prend alors la forme suivante, illustrée ici sur notre classe `Film`, et en supposant que la version courante (celle de l'objet dans le cache) est 3.

```
update Film set [...], version=4 where id=:idFilm and version=3
```

Hibernate sait (par l'objet présent dans le cache) que la version courante est 3. La requête SQL se base sur l'hypothèse que le cache est bien synchrone avec la base de données, et ajoute donc une clause `version=3` pour avoir la garantie que la ligne modifiée est bien celle lue initialement pour instancier l'objet du cache.

Si c'est le cas, cette ligne est trouvée, la mise à jour s'effectue et le numéro de version est incrémenté.

Si ce n'est pas le cas, c'est qu'une autre transaction a effectué une mise à jour concurrente et incrémenté le numéro de version de son côté. L'objet dans le cache n'est pas synchrone avec la base, et la mise à jour doit être rejetée. Hibernate engendre alors une exception de type `StaleObjectStateException`. On se retrouve dans la stratégie 2 ci-dessus (le premier `commit` l'emporte), avec nécessité d'informer l'utilisateur qu'un mise à jour concurrente est passée avant la sienne.

Important : Vous aurez noté que le mécanisme n'est sûr que si *toutes* les mises à jour de la table `Film` passent par le même code Hibernate… C'est la limite (forte) d'une gestion de la concurrence au niveau de l'application.

Doté de ce mécanisme natif Hibernate, voyons comment l'utiliser dans le cadre de transactions applicatives. Auparavant, vérifions que, si nous faisons rien de plus, c'est la première solution, *le dernier* `commit` *l'emporte*, qui s'applique.

14.3.3 Par défaut, le dernier commit l'emporte !

Nous implantons simplement la fonction de mise à jour d'un film. La première action affiche le formulaire. Voici son code :

```
if (action.equals("editer")) {
     // Action + vue test de connexion
     Transactions trans = new Transactions();
     Film film = trans.chercheFilmParTitre("Gravity");
     request.setAttribute("film", film);
     maVue = "/vues/transactions/editer.jsp";
    }
```

Je vous laisse implanter la méthode `chercheFilmParTitre()` avec la requête HQL appropriée. Voici la page JSP (restreinte à la balise `body`).

```
<body>

      <h2>Edition d'un film: le formulaire</h2>

      <p>Vous éditez le film ${film.titre}, dont la version courante est ${film.version}
↪</p>

      <form action="${pageContext.request.contextPath}/transactions" method="get">
      <input type="hidden" name="idFilm" value="${film.id}"/>
      <input type="hidden" name="action" value="modifier"/>
      <textarea name="resume" cols="80" rows="7">${film.resume }</textarea>
      <input type="submit">
      </form>

</body>
```

Nous sommes dans le contexte d'un contrôleur `Transactions`. Vous voyez que sur validation de ce formulaire, on appelle une autre action du même contrôleur, `modifier`. Voici son code :

```
if (action.equals("modifier")) {
  // Initialisation du modèle
  Transactions trans = new Transactions();

  // Recherche du film
  String idFilm = request.getParameter("idFilm");
  Film film = trans.chercheFilmParId(Integer.parseInt(idFilm));

  // On conserve la version avant modif, pour l'afficher
  request.setAttribute("versionAvantModif", film.getVersion());

  // Modification et validation
  film.setResume (request.getParameter("resume"));
  trans.updateFilm(film);
```

(suite sur la page suivante)

(suite de la page précédente)

```
  // Affichage
  request.setAttribute("film", film);
  maVue = "/vues/transactions/modifier.jsp";
}
```

Je vous laisse implanter les méthodes `chercheFilmParId()` et `updateFilm()`, cette dernière selon le modèle de transaction présenté précédemment. Le code de la JSP est le suivant :

```
<h2>Mise à jour du film</h2>

<p>Vous avez demandé la mise à jour du film ${film.titre}, dont
la version courante est ${versionAvantModif }
   </p>

<p>
   Le résumé que vous avez saisi est: ${film.resume }
<p>
<p>
La version <i>après modification</i> est ${film.version}
 </p>
<a href="${pageContext.request.contextPath}/transactions?action=editer">Retour au
formulaire d'édition.</a>
```

Et voilà. Il est très facile de vérifier, en ouvrant deux navigateurs sur la fonction d'édition du film, que le dernier utilisateur qui clique sur *Valider* l'emporte : la mise à jour du premier est perdue.

Exercice : vérifier que le dernier l'emporte

Implantez les actions précédentes et vérifiez par vous-mêmes que deux utilisateurs qui éditent de manière concurrente un film voient l'un des deux (le dernier qui valide) l'emporter.

14.3.4 Granularité des sessions

Dans l'implantation qui précède, le numéro de version ne nous sert à rien, car nous relisons à nouveau le film dans la méthode `modifier()`, et il n'y a donc aucune vérification que la version du film que nous *modifions* est celle que nous avons *éditée*.

Jusqu'à présent, la portée d'une session coincide avec celle d'une requête HTTP. On ouvre la session quand une requête HTPP arrive, on la ferme avant de transmettre la vue en retour. C'est une méthode dite *session per request*, et la plus courante.

On peut gérer une transaction applicative en gardant la même granularité (une session pour chaque requête) mais en exploitant la capacité d'un objet persistant à être *détaché* de la session courante. Dans ce cas on procède comme suit :

- un objet persistant est placé dans le cache de la première session ;
- quand on ferme la première session, on *détache* l'objet et on le conserve bien au chaud (par exemple dans l'objet `HttpSession`) en attendant la requête HTTP suivante ;
- quand cette dernière arrive, on *ré-attache* l'objet persistant à la nouvelle session.

La méthode est dite *session-per-request-with-detached-objects*. Elle est illustrée par la figure *Mécanisme de la session avec objets détachés*.

Enfin il existe une méthode plus radicale, consistant à placer la session toute entière dans le cache applicatif survivant entre deux requêtes HTTP (typiquement l'objet `HttpSession`). Dans ce cas on ferme la connexion JDBC avec la

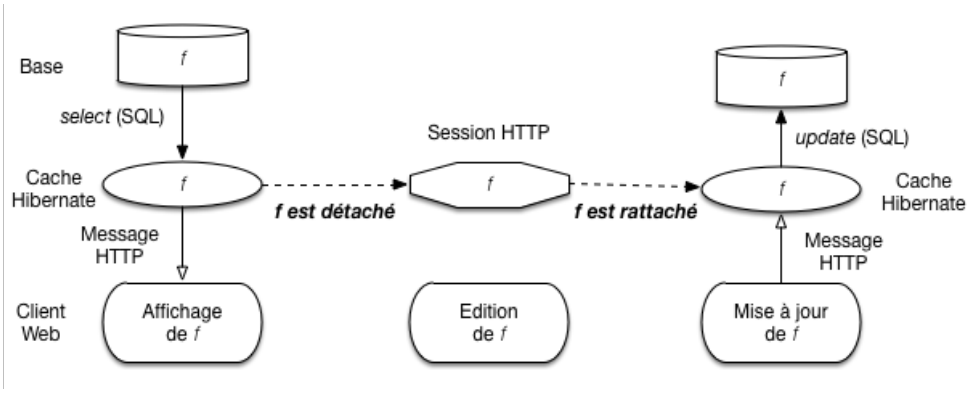

FIG. 2 – Mécanisme de la session avec objets détachés

méthode `disconnect()` et on en ouvre une nouvelle avec la méthode `reconnect()` quand la session est ré-activée. La méthode est dite *session-per-application-transaction* ou tout simplement session longue.

14.3.5 Méthode des objets détachés

Voici quelques détails sur la méthode *session-per-request-with-detached-objects*. Tout d'abord nous devons placer dans la session HTTP l'objet à gérer par transaction applicative. Le code est très simple :

```
HttpSession httpSession = request.getSession();
httpSession.setAttribute("object", obj)
```

La séquence est donc la suivante : (i) on ouvre une session Hibernate, et on cherche les objets que l'on souhaite éditer, (ii) on place ces objets dans la session HTTP pour les préserver sur plusieurs requêtes HTTP, (iii) on ferme la session Hibernate et on affiche la vue. Voici le code complet pour l'édition du film.

```
// Version avec session applicative
Session hibernateSession = sessionFactory.openSession();

// Recherche du film
Film film = (Film) hibernateSession
                                .createQuery("from Film where titre like :titre")
                                .setString("titre", "Gravity").uniqueResult();
// On le place dans la session HTTP
HttpSession httpSession = request.getSession();
httpSession.setAttribute("filmModif", film);

// On ferme la session Hibernate
hibernateSession.close();

// Et on affiche le film dans le formulaire
request.setAttribute("film", film);
maVue = "/vues/transactions/editer2.jsp";
```

À l'issue de cette action, l'objet `film` devient donc *détaché*. Dans l'action de mise à jour (quand l'utilisateur à soumis le formulaire), il faut *rattacher* à une nouvelle session Hibernate l'objet présent dans la session HTTP. Ce rattachement s'effectue avec la méthode `saveOrUpdate()` ou simplement `update()` : c'est notamment utile si on soupçonne que l'objet a déjà été modifié, et se trouve donc *dirty* avant même d'être réaffecté à la session.

Voici le code complet montrant la transaction.

```
Session hibernateSession = sessionFactory.openSession();
Transaction tx = null;
try {
        tx = hibernateSession.beginTransaction();

        // On prend le film dans la session HTTP
        HttpSession httpSession = request.getSession();
        Film film = (Film) httpSession.getAttribute("filmModif");

        // On le réinjecte dans la session Hibernate
        hibernateSession.saveOrUpdate(film);
        tx.commit();

        // Affichage
        request.setAttribute("film", film);
        maVue = "/vues/transactions/modifier2.jsp";
} catch (RuntimeException e) {
        if (tx != null)
                tx.rollback();
                throw e; // or display error message
        } finally {
                hibernateSession.close();
}
```

La différence essentielle avec la version précédente est donc qu'on ne va pas chercher le film dans la base mais dans la session HTTP, et qu'on obtient donc l'objet avec le numéro de version `v` que l'on a vraiment édité. Si vous avez mis en place le versionnement Hibernate, la requête SQL comprendra une clause `where version=v`.

```
update Film set resume=?, version=? where id=? and version=?
```

L'absence de cette version indiquerait qu'une mise à jour est intervenue entre temps : . Hibernate lève une exception `StaleObjectStateException` avec le message suivant :

```
Row was updated or deleted by another transaction
(or unsaved-value mapping was incorrect) : [modeles.webscope.Film#79]
```

À vous de faire l'expérience avec le code qui précède : éditez un même film avec deux navigateurs différents, et vérifiez que le premier `commit` doit gagner, le second étant rejeté avec une exception `StaleObjectStateException`.

14.3.6 Méthode des sessions longues

Si on veut éviter de détacher/attacher des objets, on peut préserver l'ensemble de l'état d'une session entre deux requêtes HTTP. Attention : l'état d'une session comprend la connexion JDBC qu'il est impératif de relâcher avec `disconnect()` si on ne veut pas « enterrer » des connexions et se retrouver face au nombre maximal de connexions autorisé.

Quand on reprend une action déclenchée par une nouvelle requête HTTP, il faut récupérer l'objet *session* Hibernate dans la session HTTP, et appeler `reconnect()`. On récupère alors l'ensemble des instances persistantes placées dans le cache.

Important : Il faut même penser à *fermer* la session quand la transaction applicative est terminée, ce qui suppose de savoir identifier une action « finale », ou au contraire de fermer/ouvrir la session pour toute action qui n'est pas intégrée

à la séquence des actions d'une transaction applicative.

Exercice : une transaction applicative

Modifiez vos deux actions `editer()` et `modifier()` pour être sûr de modifier la version du film que vous avez éditée. Vérifiez qu'Hibernate applique bien le protocole de gestion de concurrence dans ce cas.

14.4 Résumé : savoir et retenir

La question des transactions est délicate car elle implique des processus concurrents et une synchronisation temporelle qui sont difficiles à conceptualiser. Voici les informations essentielles que vous devez voir en tête, à défaut de mémoriser le détail des solutions.

— Hibernate se comporte comme une application standard, la gestion de la concurrence étant réservée au SGBD.
— Le niveau d'isolation transactionnel par défaut des SGBD est permissif et autorise potentiellement (même si c'est rare) l'apparition d'anomalies dues à des transactions concurrentes. Il faut parfois savoir choisir le niveau d'isolation maximal pour s'en prémunir.
— Du point de vue de l'utilisateur, certaines transactions couvrent plusieurs sessions Hibernate, et la gestion de concurrence du SGBD devient alors ineffective. Hibernate fournit un support pour implanter un contrôle de concurrence multiversion dans ces cas là.

CHAPITRE 15

Projets

15.1 Description

Le projet consiste à réaliser une mini-application fortement centrée sur la conception du modèle de données et sa réalisation avec JPA/Hibernate. Des énoncés sont donnés ci-dessous.

Pour chacun, la ligne directrice est la même. Je fournis des énoncés dans ce qui suit, vous êtes libres de choisir celui qui vous convient.

15.1.1 Ce qui est impératif

Il faut impérativement effectuer les étapes suivantes qui constituent le noyau des connaissances à acquérir.

- concevoir le modèle de données avec UML (ou équivalent) en distinguant bien les classes, les associations et les relations de généralisation (héritage) ; ce modèle peut être validé par votre enseignant avant tout début de réalisation, en me l'envoyant par mail (raphael.fourniersniehotta@lecnam.net, cette validation est optionnelle) ;
- créer le schéma de la base de données *et* le modèle Java, sur la base de la conception effectuée précédemment ; bien entendu, les deux doivent être liés par un *mapping* JPA ;
- insérer quelques données et implanter une mini-application Web MVC qui permet de naviguer dans la base et de consulter les données.

Avec JPA, le but et de concevoir des applications orientées-objet dont la persistance n'est qu'un aspect. La base de données ne sert qu'à représenter les données, « nues », les classes représentent les données *et* les comportements (logique métier). Pour que cela soit bien clair, il sera bon d'implanter quelques calculs « métiers » et de montrer leur résultat. Les énoncés sont prévus en ce sens.

15.1.2 Ce qui est optionnel

En plus des étapes impératives, l'évaluation du projet tiendra compte des fonctionnalités suivantes :
- intégration de l'interface Web dans un *template* avec menu, logo, etc., comme vu en cours ;
- utilisation intensive des liens HTML, pour « naviguer » dans la base et bien mettre en évidence sa structure de graphe d'objets ;
- minimisation des requêtes SQL par utilisation appropriée de HQL ;
- réalisation d'une transaction applicative.

15.1.3 Ce qui est apprécié

Enfin, des critères non spécifiques à JPA/Hibernate seront appréciés : bonne organisation de vos classes, normalisation des règles de nommage, documentation, commentaires. C'est l'aspect le moins essentiel, mais pensez à le soigner quand même.

15.1.4 Rendu du projet

Le code du projet doit être fourni sous la forme d'une archive d'application Web `.war`. Cette archive doit pouvoir s'installer directement sous un environnement Tomcat/MySQL/JPA/Hibernate conforme à celui décrit dans l'ensemble du cours. La validation par les enseignants consiste à installer l'application, à consulter les différentes pages et à vérifier les fonctionnalités implantées. Un extrait de la base de données avec laquelle vous avez travaillé devra être également fourni, pour faciliter le test des fonctionnalités de votre projet.

Un bref rapport devra accompagner le projet, décrivant l'installation, le schéma de la base, éventuellement un jeu de données (vous pouvez exporter votre base MySQL), une liste des fonctionnalités et quelques copies d'écran. Le fichier sera fourni au format pdf et ne devra pas excéder 10 pages (minimum 1 page).

Les règles de nommage à respecter sont les suivantes :
- **la base sql porte votre nom**. Le *dump* a pour extension *.sql*. Exemple : **fournier.sql**, contenant la base *fournier*.
- le fichier contenant le code porte aussi votre nom, suivi du type de projet. Exemples : **fournier-sport.war**, **fournier-mediatheque.war**
- le rapport a le même schéma de nommage que l'archive, mais pour extension *.pdf*. Exemple : *fournier-sport.pdf*
- les trois fichiers sont inclus dans une archive au format zip (exemple : **fournier-mediatheque.zip**), déposée sur la plateforme Moodle. La date butoir de rendu est fixée au **vendredi 22 février 2020 à 19h59**, la plateforme ne permettra pas les rendus au-delà du dimanche suivant (et les retards seront indiqués).

Le non-respect du schéma de nommage sera sanctionné. Les envois de pièces jointes par mail ne seront pas acceptés.

15.2 Projet 1 : Un club de sport

15.2.1 Les besoins

Un club de sport de raquettes propose à ses adhérents de faire du tennis (indoor), du badminton ou du squash. Vous allez être en charge de la réalisation d'une application qui permettra au propriétaire de connaître l'état de son club : quel est l'état de ses installations ? Et quel est l'état de l'abonnement de chaque joueur ?

Les joueurs effectuent des réservations de terrains, à une date et une heure données (par exemple : le 10 février 2018 à 20h30), pour une heure à chaque fois. Il y a, évidemment, un type de salle par sport, avec des attributs partiellement communs (dates de mise en service et état général avec une note entre 1 et 5 pour chaque salle) et partiellement spécifiques :
- les courts de tennis ont des surfaces différentes (appelées « greenset » et « terre battue ») ;
- les courts de squash peuvent être vitrés (ou non) et ont un parquet dont il faut stocker la date de remplacement ;
- les dates d'achats des filets et poteaux de badminton doivent être conservées.

Les joueurs ont deux types d'abonnement, au forfait, ou au ticket. Au ticket, ils ne règlent rien initialement, mais paient ensuite plus cher l'heure de sport (disons 22, 20 et 18 euros pour du tennis, du badminton et du squash, respectivement). Au forfait, ils paient un abonnement annuel, disons de 200 euros, qui leur donne accès aux courts pour 11, 10 et 9 euros de l'heure, respectivement.

15.2.2 La conception (diagramme UML)

— Proposez une modélisation du système d'informations, avec les différentes classes, les associations. Précisez soigneusement l'emplacement des attributs.
— Définissez une ou plusieurs méthodes permettant de calculer ce qui a été payé par un joueur le mois précédent (s'il est au forfait, intégrez dans le calcul 1/12 du montant du forfait annuel). Spécifiez chaque méthode avec précision et concision en fonction des attributs définis précédemment (pseudo-code).
— Maintenant complétez le modèle en intégrant les informations sur les types de sport. Spécifiez la ou les méthodes donnant le prix d'une heure de chaque sport pour chaque joueur.
— Définir une méthode qui indique le chiffre d'affaire du mois dernier.

15.2.3 L'implantation (JPA/Hibernate)

— Donnez le schéma de la base résultant du modèle UML. Implantez ce schéma avec MySQL.
— Réalisez le modèle JPA de l'application.
— Réalisez une mini-application MVC affichant les données de la base et capable de calculer le chiffre d'affaire.

15.2.4 Remarque :

Vous pouvez être créatifs et changer les valeurs des montants payés, ajouter des fonctionnalités.

15.3 Projet 2 : Le site d'une médiathèque municipale

15.3.1 Les besoins

Une médiathèque municipale prête des documents à différents abonnés (deux catégories : enfants et adultes), qui peuvent les emprunter pour des durées variables (selon la catégorie d'abonné et le type de documents). Les documents peuvent être de différentes nature (DVD, Livre, Périodique).

Les adultes ont le droit d'emprunter 10 documents simultanément, dont 3 DVD (maximum) et 2 périodiques. Les DVD ne peuvent être empruntés plus d'une semaine, livres et périodiques peuvent être conservés 3 semaines. Les enfants ne peuvent emprunter que 6 documents simultanément, dont 2 DVD et 1 périodique. Ils ne peuvent garder les documents que 2 semaines maximum.

Le site est une version simplifiée, avec des fonctionnalités réduites par rapport à ce que l'on attendrait d'un vrai site. Votre site Web doit permettre aux utilisateurs de réserver des documents. Faites simple, pas de gestion de « panier » comme sur des sites marchands ici : un utilisateur doit pouvoir afficher la liste des documents présents dans la base, puis cliquer sur « réserver » sur un bouton à côté du document pour l'emprunter (sauf, bien sûr, s'il dépasse alors le quota qui lui est autorisé). Pour éviter de gérer des « sessions utilisateurs », vous vous en tiendrez à un nombre restreint d'utilisateurs (par exemple 4, 2 adultes et 2 enfants), et afficher des liens sous la forme « réserver en tant que Adulte1 », « réserver en tant qu'Enfant2 ».

Le site doit aussi permettre aux documentalistes de voir si des utilisateurs ont des documents en retard. Là encore, version simplifiée : toutes les pages sont accessibles à tous les utilisateurs. Il faut donc une page dans laquelle est listé si un document est actuellement emprunté, par qui, et est-ce que le document devrait être de retour (à calculer avec les informations fournies ci-dessus).

15.3.2 La conception (diagramme UML)

— Proposez une modélisation des différentes entités et associations requise par l'application
— Indiquez comment calculer le retard des documents

15.3.3 L'implantation (JPA/Hibernate)

— Donnez le schéma de la base résultant du modèle UML. Implantez ce schéma avec MySQL.
— Réalisez le modèle JPA de l'application.
— Réalisez une mini-application MVC affichant les données de la base, comme décrit ci-dessus.

15.3.4 En option

— triez l'affichage des pages de documents par leurs catégories, n'affichez pas les documents déjà empruntés, permettez l'ajout de documents dans la base.

CHAPITRE 16

Liens utiles

16.1 Programmation et environnements Java

Il existe sur le Web d'innombrables tutoriels de toutes sortes sur la programmation Java et le développement Web java en particulier.

1. http://fr.openclassrooms.com/informatique/cours/creez-votre-application-web-avec-java-ee est une excellente ressource pour la programmation JEE. Très bien rédigé et expliqué, merci à l'auteur qui a beaucoup aidé certains aspects de la préparation du cours.

CHAPITRE 17

Annexe : Introduction aux gestionnaires de versions

Dans ce chapitre, nous proposons une introduction aux gestionnaires de versions, des outils importants pour les développeurs, qu'ils travaillent seuls ou en équipe. Nous verrons le principe de fonctionnement général dans une première partie. Dans la deuxième séquence, nous configurerons l'environnement qui vous permettra de travailler avec Git sur votre machine dans Eclipse et de vous synchroniser avec le GitLab du CNAM. Enfin, nous verrons les commandes de base pour que vous puissiez utiliser Git (prononcé « guite »).

Notre présentation de Git est ici volontairement restrictive. Git est un gestionnaire de versions très puissant et complexe, nous essayons de le rendre accessible à des développeurs débutants. En particulier, il est possible de se passer de l'intégration de git dans Éclipse et d'utiliser la ligne de commande, mais cet usage est à réserver à des utilisateurs plus avertis.

17.1 S1 : Principe de fonctionnement d'un gestionnaire de versions

Note : Des anciennes diapos sont encore disponibles, elles présentent le fonctionnement de Subversion, un gestionnaire de versions différent de Git. Elles ne collent donc plus au cours présenté mais peuvent intéresser des élèves. Introduction rapide à SVN

Un « gestionnaire de versions » ou « logiciel de contrôle de versions » est destiné, comme son nom l'indique, à gérer les différentes versions de fichiers. Cela va donc nous permettre, en tant que développeur, de pouvoir garder une trace des modifications successives effectuées sur un projet pour le mener à bien.

En effet, dès qu'un projet informatique dépasse quelques dizaines de lignes de code, il devient évident qu'il faut le structurer en plusieurs fichiers, voire en une arborescence complexe, afin que toutes les personnes impliquées puissent comprendre rapidement quelle est la structure du projet et où se trouvent les portions de code relatives à tel ou tel aspect (cf par exemple l'architecture MVC présentée dans le chapitre *Modèle-Vue-Contrôleur (MVC)*).

Que l'on soit seul ou plusieurs centaines de développeurs sur le même projet, il arrive alors fréquemment que l'on rencontre les problèmes suivants :

— un fichier comportant du code fonctionnel a été modifié *à un moment* et ne fonctionne plus ;
— on a ajouté de nouveaux fichiers au projet mais quelques jours/semaines plus tard, on ne se souvient pas pourquoi ;

— on a résolu un problème il y a 3 mois dans une partie du code et on aimerait pouvoir réutiliser la solution dans une autre partie du code ;
— on souhaite pouvoir travailler à plusieurs endroits (sur plusieurs machines) sans devoir s'envoyer la totalité du code par mail sous forme d'archive ;
— on souhaite pouvoir collaborer (pendant que l'un travaille sur la partie modèle, un autre se charge d'améliorer la vue).

Un gestionnaire de versions comme Git va permettre de résoudre ces situations problématiques En effet, les développeurs du projet pourront :

— garder une trace des modifications faites sur chaque fichier avec des commentaires associés,
— pouvoir facilement revenir à un état antérieur du code
— fusionner efficacement les modifications de plusieurs personnes pour avoir une version cohérente du code.

17.1.1 Les dépôts et les *commits*

À la base d'un gestionnaire de versions, il y a ce qu'on appelle un « dépôt » (en anglais *repository*). Il s'agit, en première approximation, d'un dossier dans lequel vous allez travailler et décider que tout ce qui se trouvera dans ce dossier sera *sous le contrôle de Git*.

Note : Quand on travaillera en synchronisation avec la plateforme GitLab, on effectuera en réalité la synchronisation de deux dépôts, celui sur notre machine et celui sur la plateforme (qui servira donc notamment de copie de sauvegarde, en cas de problème sur votre machine principale). De même, si vous travaillez sur plusieurs machines, pour démarrer sur une nouvelle machine il vous suffira de créer un dépôt et de le synchroniser avec celui du Gitlab pour revenir immédiatement à la dernière modification faite sur votre code.

Deuxième aspect important, la notion de **révision**, ou de **commit** (selon le gestionnaire utilisé). Il s'agit d'un état de l'arborescence à un moment donné. Git ne fera **pas d'enregistrement automatique** de vos fichiers : il va falloir lui indiquer, précisément, le moment où vous souhaitez enregistrer un état de votre code, avec un *message de commit* qui vous permettra plus tard de comprendre quel était l'état du code à ce moment-là. Voici des exemples de messages pour divers projets : « ajout d'une page de contact sur le site », « résolution du bug sur le bouton de validation de panier dans la page de Commande », « la communication avec la base de données a été établie et fonctionne. ». Chaque commit se voit attribuer un identifiant automatique par Git, de façon à ce qu'il soit unique au sein d'un projet. Plus tard, vous pourrez voir l'arborescence de votre projet comme sur la figure *Historique Git*, qui montre une partie du code destiné à ce cours, avec les différents messages de *commit* que j'ai utilisés.

À l'aide de l'identifiant, il sera ensuite facile d'indiquer à Git à quel moment de l'historique on souhaite revenir. Ou d'identifier à quel moment a été introduite telle ou telle modification.

Note : Même si ce n'est pas toujours immédiatement apparent, chaque commit est aussi signé avec le nom et l'adresse mail de son auteur. Donc il est possible de savoir non seulement *quand* a été introduit un bug, mais aussi de savoir *qui* est à l'origine du commit posant problème.

FIG. 1 – Historique Git

17.1.2 Un système décentralisé

Nous allons maintenant expliquer l'aspect décentralisé de GIT à partir de la figure *Le système décentralisé avec le GitLab du Cnam*. Cette figure montre l'organisation que nous allons adopter avec l'environnement proposé par le Cnam.

Pour commencer regardons en bas à gauche. Nous avons une machine `maMachineA` sur laquelle nous développons notre projet avec Eclipse. les fichiers de ce projet sont gérés par un GIT qui entrepose les versions successives dans un *dépôt local*. Ici, il se nomme `dépôtLocalA`.

Chaque fois que nous effectuons une mise à jour d'un fichier avec Eclipse, nous avons la possibilité de le sauvegarder dans le dépôt local. Nous avons en fait deux commandes :

- `add` permet d'ajouter un nouveau fichier ;
- `commit` permet de valider les modifications sur un fichier.

Il y a d'autres commandes, et notamment celles qui permettent de revenir à une version antérieure si on a fait une erreur. mais pour l'instant nous allons nous contenter de ces deux-là.

Maintenant, notre dépôt GIT est local ce qui n'est pas très pratique. D'une part si nous n'avons plus accès à notre `machineA`, nous ne pouvons plus accéder à nos fichiers, d'autre part nous ne pouvons pas travailler avec d'autres personnes en partageant des fichiers.

Nous allons donc synchroniser notre dépôt local avec un dépôt distant. En langage GIT, cela s'appelle *clôner* des dépots. GIT étant entièrement décentralisé, on peut se clôner avec plusieurs autres dépôts dans le cas très général. Mais dans notre cas particulier, **nous allons toujours nous clôner avec le dépôt du CNAM, qui est nommé `dépôtDistant` sur la figure**.

Pour clôner deux dépôts, on utilise initialement la commande GIT `clone`. Les deux dépôts sont alors connnectés et synchronisés.

Ensuite, chacun va vivre sa vie. Des modifications seront faites sur le dépôt local, d'autres sur le dépôt distant. À un moment donné on va vouloir synchroniser, dans un sens ou dans les deux sens. Supposons que l'on soit situé au niveau

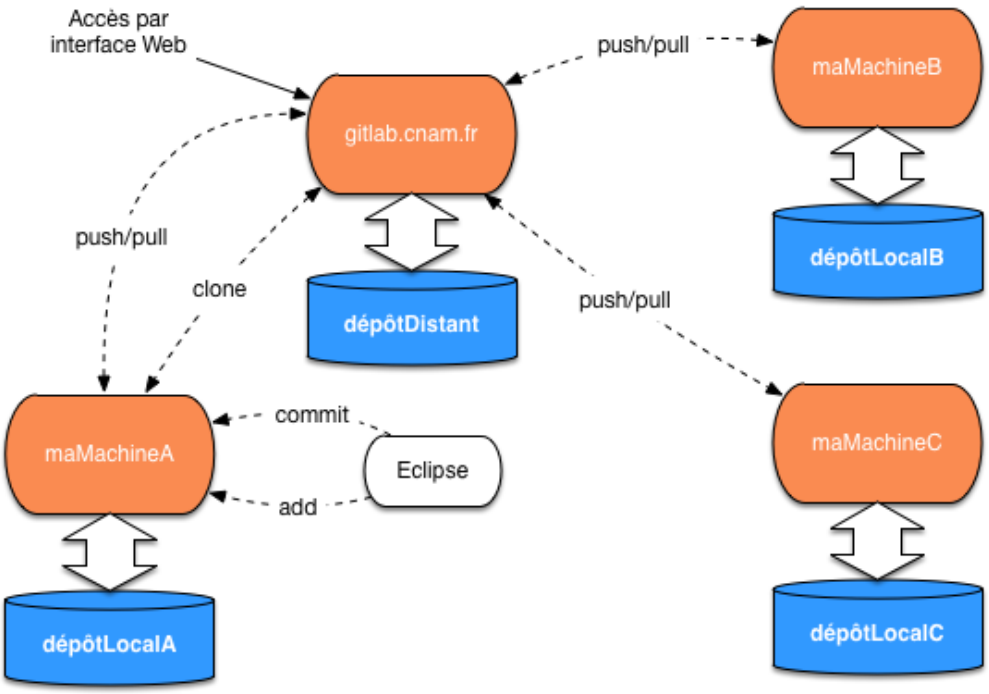

FIG. 2 – Le système décentralisé avec le GitLab du Cnam

du dépôt local, alors : GIT sont les suivantes :

— `push` va transmettre les modifications du dépôt local vers le dépôt distant ;
— `pull` va transmettre les modifications du dépôt distant vers le dépôt local.

Et voilà ! Il y a beaucoup d'autres choses (des branches, et autres complications) mais cela va nous suffire : quand nous maîtriserons ces commandes, nous serons déjà satisfaits.

Une simple remarque pour finir : supposons que `machineA` est celle que vous utilisez au Cnam. Vous rentrez chez vous après avoir synchronisé vos fichiers avec le dépôt distant, et vous voudriez continuer à travailler sur votre `machineB` en récupérant toutes vos mises à jour.

Vous voyez peut-être comment faire ? Il suffit de clôner le dépôt distant avec votre nouveau dépôt local, celui de la machineB, et le tour est joué : vous retrouvez tout votre environnement de travail.

C'est déjà très bien comme cela. Passons à la pratique.

17.2 S2 : Ce qu'il faut savoir faire

Dans cette section nous allons effectuer les principales manipulations avec Git, celles que vous serez amenés à faire régulièrement pour gérer votre projet. Les voici, en résumé.

— Créer votre *dépôt distant* Git sur l'environnement du Cnam.
— Clôner le dépôt distant vers votre *dépôt local*.
— Créer un projet avec Eclipse, et placer les fichiers du projet sous le contrôle Git du *dépôt local*.
— Synchroniser le dépôt local et le dépôt distant.
— Enfin, installer votre projet sur une nouvelle machine en clônant le dépôt distant.

À ce stade, vous devriez comprendre le but de ces manipulations. Prenez le temps de bien relire ce qui précède et de réfléchir à l'utilité de ce que nous allons faire : il s'agit tout simplement de votre capacité à pouvoir travailler sur votre projet avec n'importe quelle machine, en utilisant un dépôt Git distant comme pivot pour récupérer vos fichiers.

17.2.1 Le dépôt distant : le GitLab du Cnam

Le CNAM a mis en place une plateforme en ligne reposant sur le logiciel GitLab afin de permettre aux étudiants de déposer leur projets gérés par Git. Vous devriez être en mesure de vous connecter à cette plateforme à l'aide de vos identifiants traditionnels. L'interface est en anglais, mais nous allons la présenter pour que vous puissiez retrouver rapidement vos projets et naviguer au sein de chacun.

La plateforme GitLab du CNAM est accessible via l'adresse suivante : https://gitlab.cnam.fr. En cliquant sur le lien précédent, vous devriez arriver sur l'écran *Login sur Gitlab*.

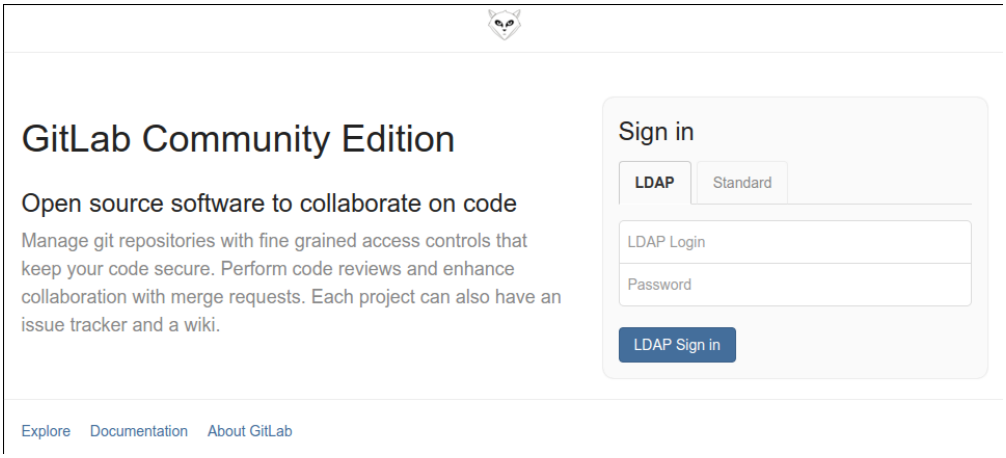

Fig. 3 – Login sur Gitlab

Après saisie de vos identifiants, vous devriez ensuite arriver sur un écran pour le moment assez vide (il sera plus tard dédié à la visualisation de l'activité dans votre projet, comme on le verra plus loin). En haut à droite de l'écran, se trouvent deux boutons pour créer un nouveau projet, l'un en forme de croix, l'autre en vert et portant la mention « New project », comme sur la figure *Créer un nouveau projet*.

Important : Cette mention « project » signifie ici « dépôt » : vous allez créer un dépôt, qui sera plus tard synchronisé avec les sources de votre projet sur votre machine. Ce n'est pas très heureux de donner des noms différents à un même concept mais nous n'y pouvons rien.

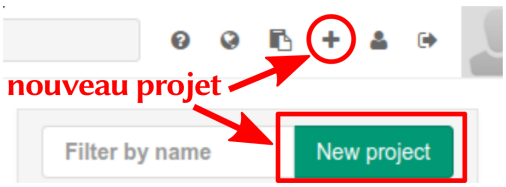

Fig. 4 – Créer un nouveau projet

Cliquer sur l'un de ces boutons vous amène à un écran où vous définirez quelques paramètres pour votre nouveau projet (cf Figure *Paramètres pour un nouveau projet*) :

— un nom (très important)
— un niveau de visibilité (choisissez « Private », de façon que votre travail ne soit, ici, pas visible des autres).

Vous devriez aboutir à un écran qui vous donne notamment l'URL pour synchroniser le travail entre votre machine et la plateforme Gitlab (il faut cliquer sur le bouton *HTTPS*, plutôt que *SSH*).

Exemple

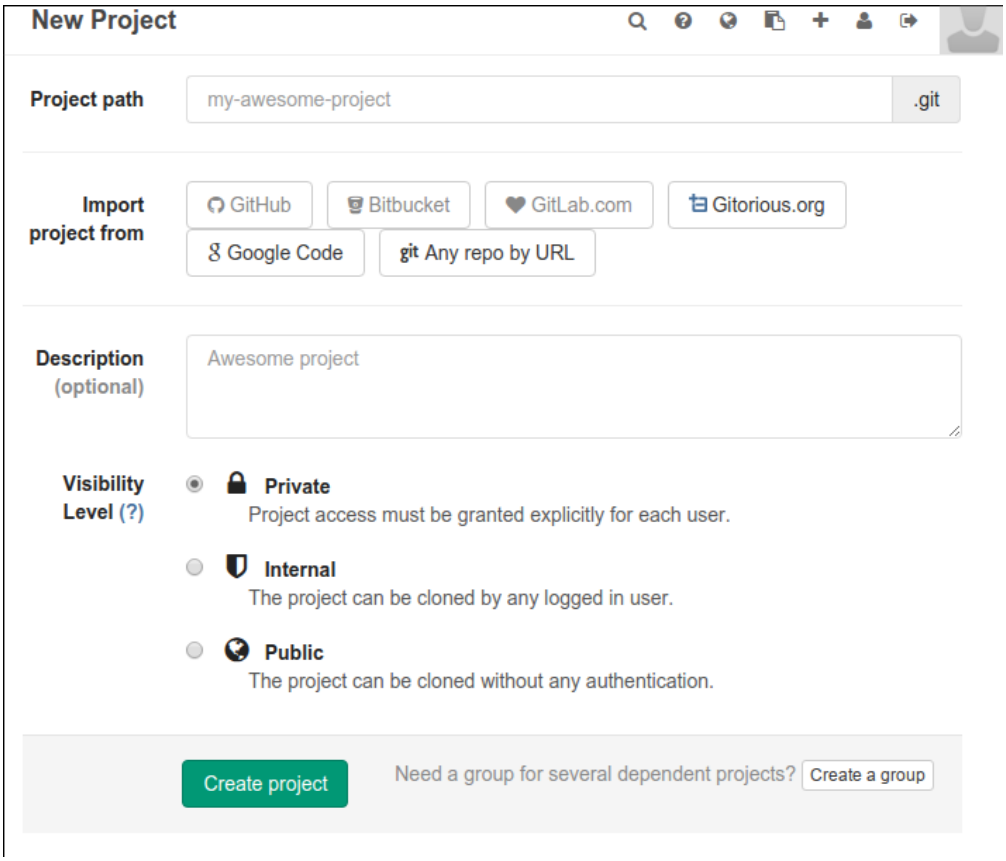

FIG. 5 – Paramètres pour un nouveau projet

Le projet des enseignants a pour URL https://gitlab.cnam.fr/gitlab/NSY135-profs/NSY135.git

Exercice.

Connectez-vous à gitlab et créez votre dépôt (projet) NSY135.

17.2.2 Clôner le dépôt distant vers le dépôt local

Maintenant nous allons clôner le dépôt distant vers le dépôt local. Il faut pour cela effectuer des commandes Git. On peut le faire en ligne de commande, mais on peut aussi utiliser Eclipse. Pour cela, nous allons devoir recourir à l'utilisation d'un plugin, très bien fait, qui permettra d'interagir avec le dépôt local et le dépôt distant sans utiliser la ligne de commande.

Quand le plugin est installé, il est possible avec Eclipse d'accéder à la *perspective GIT*. C'est celle qui va nous permettre de gérer les dépôts. Dans le menu *Window -> Open Perspective*, vérifier que vous pouvez ouvrir cette perspective. Si ce n'est pas le cas il faut installer le plugin.

Note : Les nouvelles versions d'Éclipse intègrent Git de façon native (sans plugin). J'utilise Eclipse 4.5.0, sortie en juin 2015.

Dans Éclipse, allez dans le menu « Aide » et cliquez sur « Installez de nouveaux logiciels et composants ». Dans le premier champ, entrez l'URL « http://download.eclipse.org/egit/updates » et cliquez sur Ajouter. Vous devriez ensuite être en mesure de cocher « Eclipse Git Team Provider » et de cliquer sur « Suivant » (cf figure *Ajout du site pour télécharger Eclipse Git Team Provider*). Poursuivez ensuite l'installation et redémarrez Éclipse pour que les changements soient pris en compte.

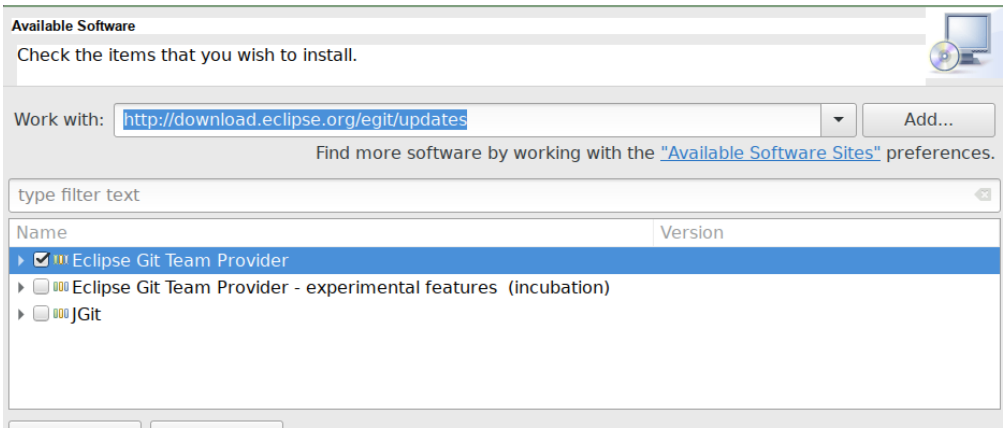

FIG. 6 – Ajout du site pour télécharger Eclipse Git Team Provider

Une fois le plugin installé, vous verrez une fenêtre semblable à celle de la figure *La perspective Git dans Eclipse*. Sur la gauche, nous voyons la liste des *dépôts locaux* qui sont connus d'Eclipse. Ici, nous en voyons deux. L'arborescence du premier a été déployée : vous pouvoir voir des *Branches*, des *tags*, un *Working tree* qui contient un répertoire *.git* avec les informations gérées par Git, et un répertoire *NSY135* : c'est la version courante des fichiers d'un projet nommé NSY135.

Dans votre cas, il n'y aura sans doute aucun dépôt local. Nous allons donc en créer un. Regardez à nouveau la figure : elle montre un menu sur la droite avec les options de création d'un dépôt. Vous pouvez :

— Importer un dépôt local existant mais pas encore connu d'Eclipse

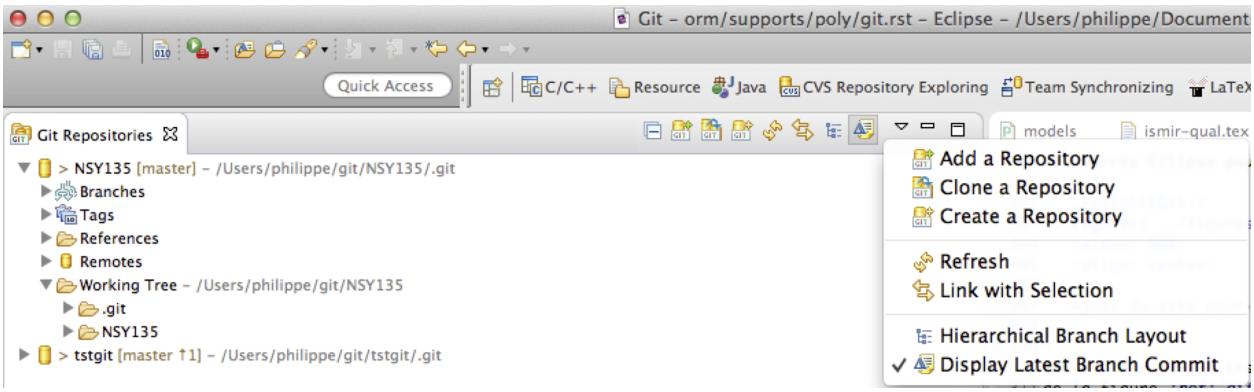

Fig. 7 – La perspective Git dans Eclipse

— Créer directement un dépôt local avec Eclipse.
— Créer un dépôt local par *clônage* d'un dépôt distant.

C'est cette dernière option que nous allons appliquer : nous allons clôner le dépôt distant du GitLab vers un dépôt local.

La figure *Spécifier le dépôt distant à clôner.* montre la première fenêtre de clônage. On y entre l'URL HTTPS du dépôt que vous avez créé dans le GitLab.

La fenêtre suivante demande quelle branche vous souhaitez prendre : acceptez le choix par défaut (`master`) et passez à la troisième fenêtre (figure *Spécifier le dépôt local.*). Vous pouvez choisir le nom du dépôt local ou reprendre le nom du dépôt distant. Notez que les dépôts locaux sont placés dans un répertoire sur votre machine dédié à Git (ici, le répertoire `HOME/git`.

Validez : votre dépôt distant sera clôné, et vos deux dépôts (le distant et le local) sont maintenants connectés. En cliquant sur le nom du dépôt qui a dû apparaître dans la fenêtre Git, vous verrez les informations sur cette connexion.

Et voilà ! Vos dépôts sont vides : l'étape suivante consiste à commencer à y placer des fichiers.

17.2.3 Gérer votre projet avec Eclipse et GIT

Créez un projet quelconque dans Eclipse. Ensuite, déclarez que vous souhaitez versionner ce projet (c'est-à-dire le placer sous le contrôle de Git) en utilisant le menu contextuel sur le nom de votre projet (un *Dynamic Web Project* comme vu dans le chapitre *Environnement Java*) puis en choisissant « Team » et « Share Project » (plus tard, j'utiliserai la syntaxe *Team* > *Share Project* pour désigner cette séquence d'opérations).

Eclipse vous demande dans quel dépôt local vous souhaitez placer votre projet. Choisissez-le et validez. Vous devez voir que votre projet comporte maintenant la mention *[Git master]* à sa droite et est précédée d'un symbole « > ». Cela signifie que des fichiers existent dans le dossier mais ne sont pas encore géré par Git. C'est parfaitement normal : nous avons dit que nous voulions que ce dossier soit contrôlé par Git, mais nous ne lui avons pas dit que nous voulions qu'il gère absolument tout…

Avec le menu contextuel (clic-droit) sur le nom du projet dans l'explorateur (partie gauche de l'interface), vous pouvez voir le menu *Team* en bas, qui contient les différentes actions liées à Git. Vous pouvez commencer par ajouter l'arborescence actuelle à votre dépôt, en sélectionnant *Add to index* (cf la figure *Ajout de l'arborescence du projet à Git*). Cette première opération ne valide pas encore : elle dit simplement que vous préparez votre projet en déclarant ce que vous allez placer sous le contrôle de Git (pour le moment *tout*, mais comme il n'y a presque rien dans votre projet…).

Il faut ensuite valider ces changements. Dans le langage Git, on parle de *commit*. Dans le menu *Team*, optez pour l'option tout en haut, appelée *Commit*. Une fenêtre s'ouvre et il ne vous reste plus qu'à indiquer un message de *commit utile* pour valider vos modifications (voir figure *Valider un commit en donnant un message d'explication*). Ici, comme c'est le premier message, « Premier commit, initialisation de l'arborescence » est un message suffisant. Validez avec

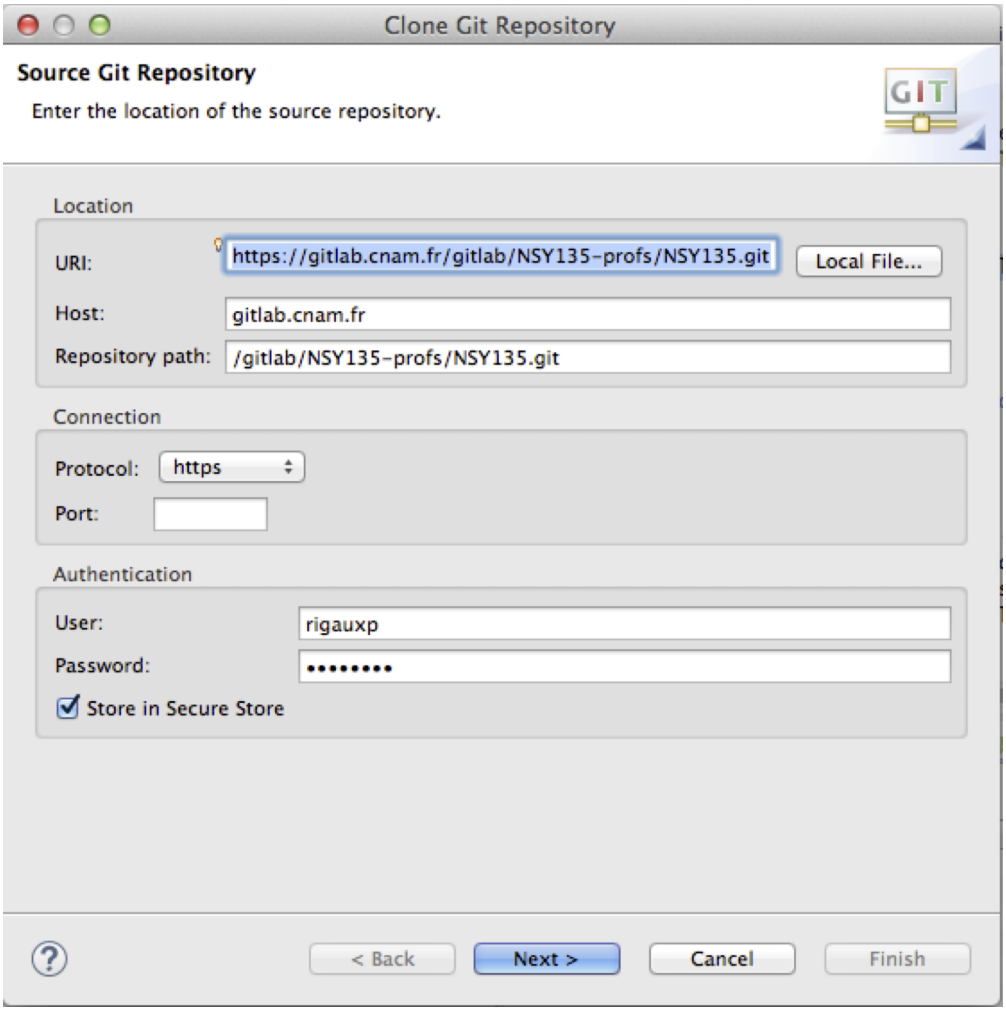

FIG. 8 – Spécifier le dépôt distant à clôner.

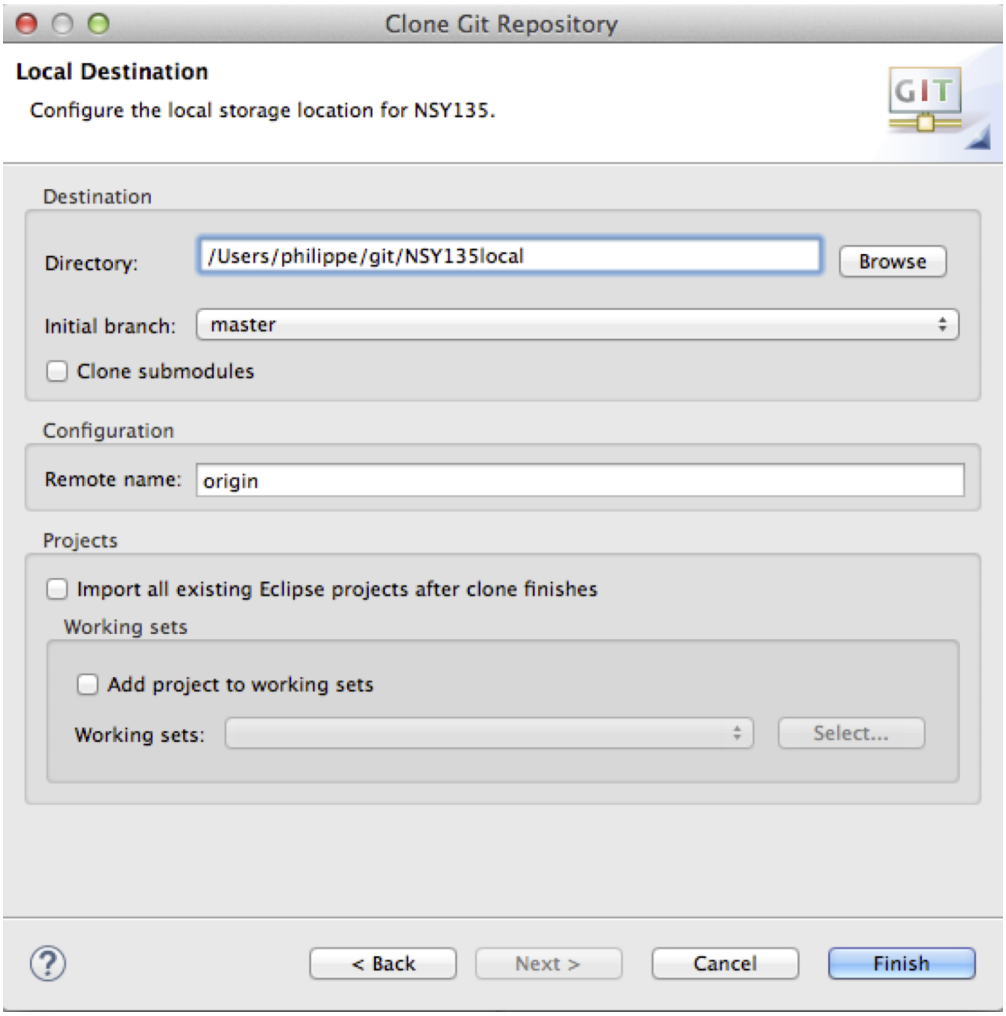

FIG. 9 – Spécifier le dépôt local.

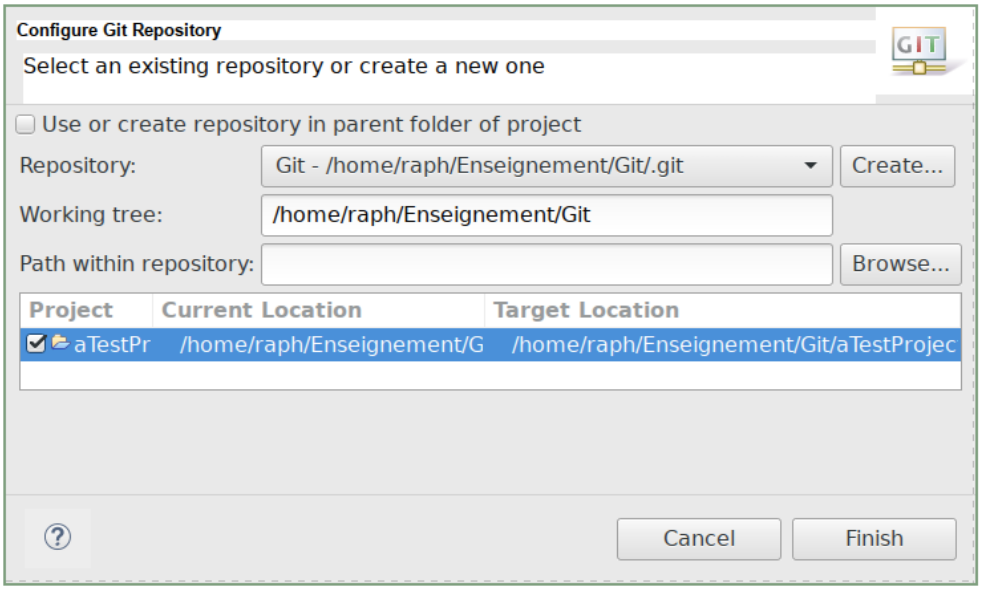

FIG. 10 – Sélection du dépôt Git sur votre machine.

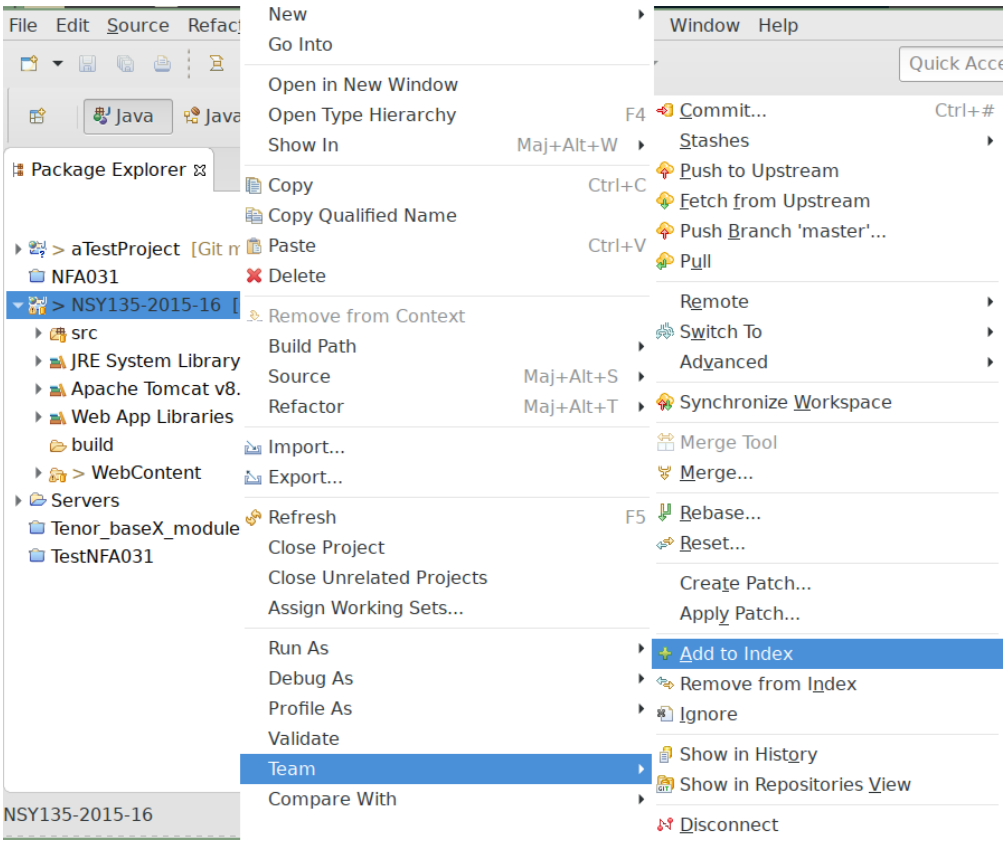

FIG. 11 – Ajout de l'arborescence du projet à Git

le bouton *Commit and push* et *voilà*, vous avez effectué votre premier *commit* avec Git (le *push* aura aussi permis de synchroniser avec la plateforme distante).

Plus tard, il faudra que vos messages décrivent quel a été l'apport d'un commit par rapport à la version précédente, de façon à avoir un historique compréhensible de la construction de votre projet. Revenez à la figure `histoGit` plus haut pour des exemples de messages.

Note : Une bonne pratique pour rédiger ses messages est de décrire « pourquoi » on a introduit telle ou telle modification. Il vaut mieux éviter de s'attarder sur le « quoi », puisque le lecteur intéressé pourra visualiser facilement ce qui a été introduit dans le code (avec *git diff*, cf section suivante, ou via l'interface Gitlab). Dans les références en fin de chapitre, vous trouverez un article dédié à la rédaction de messages de commit.

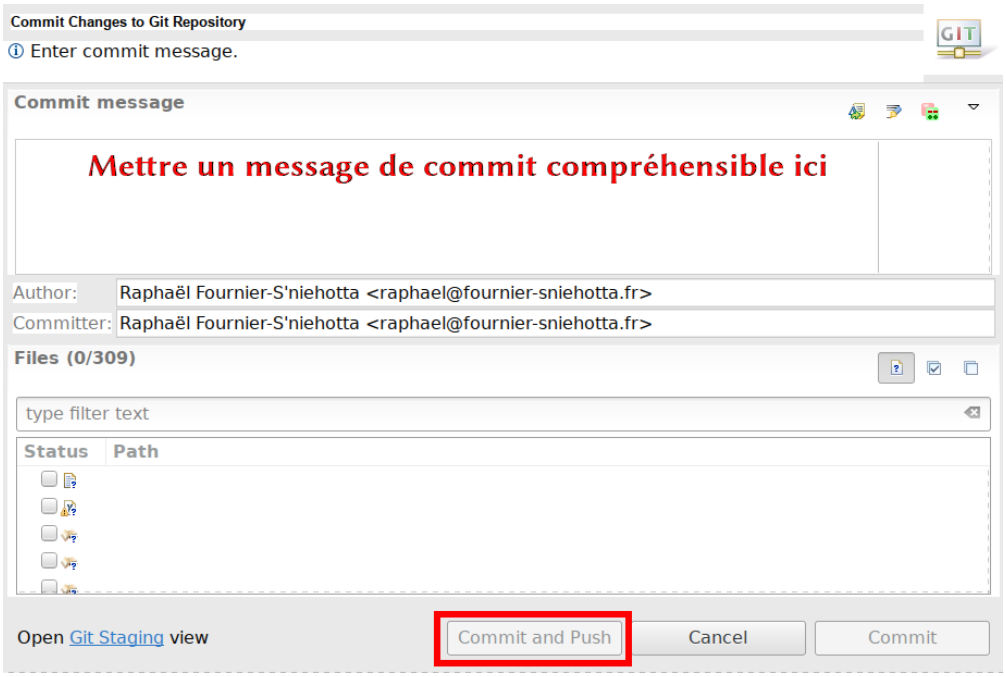

FIG. 12 – Valider un commit en donnant un message d'explication

17.2.4 Opérations sur le dépôt local

Les opérations de base permettent de travailler seul, avec un dépôt distant de sauvegarde. De cette façon, Git sert à enregistrer un historique du travail effectué.

Dans la section précédente, vous avez effectué un premier commit. Le travail ultérieur suivra une procédure similaire :

- ajouter du contenu à votre arborescence (écrire du code)
- décider de ce que vous souhaitez valider (**git add**)
- valider (**git commit**)
- envoyer votre ou vos *commits* au dépôt distant (**git push**)

Le premier point ne dépend pas de Git, seulement de votre éditeur de code. Le deuxième point fait appel à la commande *git add*, qu'on a utilisé précédemment à l'aide d'une commande graphique, sur la figure *Ajout de l'arborescence du projet à Git*. Nous avions choisi d'ajouter toute notre arborescence, mais vous pouvez effectuer des ajouts partiels, en ne choisissant qu'un sous-ensemble des dossiers/fichiers présents (en faisant un clic droit sur chacun des sous-dossiers ou des fichiers, puis *Add to index*).

Note : Remarquons ici que certains fichiers ne seront généralement pas « versionnés », sans quoi cela risquerait de surcharger inutilement le dépôt. Il s'agit notamment des bibliothèques lourdes que vous ajoutez à votre projet, et plus généralement des fichiers binaires (images, exécutables) : Git est, avant tout, fait pour « versionner » du texte (code source).

Vous pouvez utiliser la *Git Staging View* (vue préparatoire en français), une interface avec laquelle on visualise précisément l'ensemble des fichiers d'un dépôt qui ne seront pas contrôlés par Git, lesquels ont été modifiés depuis le dernier commit et on choisit lesquels seront incorporés au prochain commit (voir exemple figure *Git staging : espace pour choisir ce qui sera ou ne sera pas intégré au prochain commit.*). Depuis cette interface, vous pouvez aussi accéder à une comparaison entre l'état actuel du fichier et l'un de ses états passés (accessible également via *clic-droit* > *Compare With* > *Commit...*).

FIG. 13 – Git staging : espace pour choisir ce qui sera ou ne sera pas intégré au prochain commit.

17.2.5 Synchroniser avec le dépôt distant

Une fois que vous êtes satisfait de ce que vous avez réalisé dans l'interface de *staging*, vous pouvez écrire un joli message de commit dans la partie droite puis cliquer sur *Commit and Push* (valider et envoyer en français), ce qui permettra d'envoyer votre commit sur le Gitlab configuré précédemment.

Une fois l'opération d'envoi effectuée, vous devriez pouvoir voir les changements sur Gitlab, de façon similaire à ce qui est visible sur la figure *Historique Git*.

Note : Vous aurez peut-être remarqué qu'il y a le choix entre *Commit* et *Commit and Push*. Pour un usage simple et efficace, recourez seulement à **Commit and Push**. Parfois, vous pouvez avoir besoin de ne pas envoyer tout de suite un ensemble de modifications vers la plateforme, et les conserver sur votre machine, vous utiliserez alors le *Commit* seul.

17.2.6 Récupérer un projet sur une nouvelle machine

Maintenant, vous êtes en mesure de travailler, à des moments différents, sur deux machines (typiquement une chez vous et l'autre au CNAM), chacune pouvant envoyer ses changements sur la plateforme distante Gitlab. Il vous manque cependant la possibilité de recevoir les changements effectués sur l'autre machine. En effet, si vous travaillez sur les deux machines à des moments différents, vous allez avoir besoin de récupérer ce qui a été fait sur l'autre machine, pour travailler sur la dernière version et ne pas devoir refaire sur une machine les modifications que vous avez faites sur l'autre.

La **première chose à faire** quand vous changez de machine est donc, avant toute modification sur votre code, d'utiliser la commande *git pull* (ou *clic-droit* > *Team* > *Pull*). Cela devrait récupérer sur Gitlab ce que vous avez envoyé avec votre autre machine. Vous pouvez ensuite commencer à travailler, faire des *commits* et les envoyer.

17.3 S3 : Git avancé

17.3.1 Fonctions avancées

Note : Avec les fonctionnalités suivantes, on permet le travail collaboratif de plusieurs développeurs. Pour le moment, cette partie n'est pas écrite, l'usage de ces commandes est un peu plus complexe à détailler pour des débutants.

- git merge
- git diff
- git blame
- git bisect
- git apply (très avancé ?)

17.3.2 Les branches

Note : De même, l'usage des branches n'est pas encore présenté.

Il est parfois intéressant d'utiliser l'intégration de Git dans Eclipse, mais la ligne de commande permet souvent d'accéder à toute la puissance de Git. Nous allons voir dans cette section une petite démonstration de ce qu'il se passera pour vous quand vous utiliserez Git entre votre machine personnelle et la machine mise à votre disposition pour nos cours au CNAM.

Le scénario que nous illustrons sera le suivant :

- un projet de code sera créé durant une séquence introductive en cours, sur une machine du CNAM
- le code sera commité et envoyé sur votre espace Gitlab (au moins une fois à la fin de la séance, éventuellement à des moments intermédiaires)
- chez vous, le code est récupéré
- vous travaillez sur des modifications (exercices, variantes), que vous commitez et envoyez sur Gitlab, de façon à pouvoir les retrouver en cours
- la semaine suivante, votre code évolue pour suivre le cours, il est déposé sur Gitlab
- et ainsi de suite, vous faites des allers-retours entre vos deux machines de travail.

Nous verrons à la fin que cela peut bien entendu être étendu à plus de deux machines.

Note : Dans cette section, un certain nombre de choses dépendent de votre machine, des choix que vous ferez. J'utilise les symboles < et > pour indiquer ce qui varie. Exemple : si j'écris <votreLogin>, c'est à remplacer par votre login virtualia.

17.3.3 Sur la Machine du CNAM

On se déplace dans le dossier qui accueillera notre travail, en le créant éventuellement au préalable :

```
$ mkdir Git/CodeNSY135
$ cd Git/CodeNSY135
```

On initialise le dépôt Git dans ce dossier :

```
$ git init
Dépôt Git vide initialisé dans <cheminDuDossier>/.git
```

On regarde ce qu'il se passe (la commande *git status* est très importante pour savoir dans quel état est le dépôt). On n'a pour le moment rien fait, git nous l'indique :

```
$ git status
Sur la branche master

Validation initiale

rien à valider (créez/copiez des fichiers et utilisez "git add" pour les suivre)
```

On crée un premier fichier de code. J'utilise vim, un éditeur un peu particulier, vous pouvez en utiliser d'autres, comme *gedit*, *kate*, *nano* (installés au CNAM). Sur vos machines, vous pouvez bien sûr utiliser encore autre chose, comme *SublimeText*, *Notepad++*, etc.

```
$ vim maPremiereJsp.jsp // ou gedit maPremiereJsp.jsp
```

Après avoir fini l'édition d'un fichier, regardons ce que Git sait de notre dépôt :

```
      $ git status
      Sur la branche master

      Validation initiale

      Fichiers non suivis:
(utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

  maPremiereJsp.jsp

      aucune modification ajoutée à la validation mais des fichiers non suivis sont␣
↪présents (utilisez "git add" pour les suivre)
```

Pour le moment, il n'y a pas encore eu de commit, mais il y a des fichiers présents (en réalité, un seul, maPremiereJsp.jsp) dans le dossier qui ne sont pas dans l'index de git, ils sont « non suivis ». Pour l'ajouter, comme le suggère git, on utilise la commande suivante :

```
$ git add maPremiereJsp.jsp
```

Regardons l'état de notre dépôt (vous devriez observer un changement de couleur, le nom du fichier qui était rouge doit passer au vert) :

```
$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    nouveau fichier : maPremiereJsp.jsp
```

Nous pouvons maintenant valider nos changements, avec un message de commit expliquant ce qu'on vient de faire.

```
$ git commit
<ouverture de vim>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<saisir le message de commit, en appuyant d'abord sur "i" pour entrer dans le
mode Insertion, et en terminant par Échap, ":wq" pour quitter et valider le
message de commit>

[master (commit racine) 489c500] <Premier commit.>
 1 file changed, 15 insertions(+)
 create mode 100644 maPremiereJsp.jsp
```

J'ai saisi un message de commit simple, « Premier commit. ». Si je regarde l'état du dépôt, il est considéré comme « propre » :

```
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

En utilisant la commande *git log*, on peut retracer les différents commits, avec leurs auteurs et la date précise de leur validation :

```
$ git log
commit 489c500ebac812873fe4595c034b53fc89ca8e76
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 14:54:58 2016 +0100

        Premier commit.
```

Vous pouvez refaire cette séquence d'opérations (ajouter du code, valider les changements quand c'est satisfaisant) plusieurs fois consécutivement. Il est ensuite temps d'envoyer ce code sur Gitlab, pour synchroniser notre dépôt avec la plateforme et pouvoir simplement le récupérer sur une machine chez nous.

Une fois que vous aurez créé le projet et renseigné son nom (cf figures *Créer un nouveau projet* et *Paramètres pour un nouveau projet* ci-dessus), vous arriverez sur une page comme sur la figure newProjectOKGitlab. Cette fois, vous allez utiliser les commandes du bas de la page :

```
$ git remote add origin <VotreURLdeProjet>

$ git push -u origin master
Username for 'https://gitlab.cnam.fr': <VotreLogin>
Password for 'https://<votreLogin>@gitlab.cnam.fr': <VotrePassword>
Décompte des objets: 3, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (3/3), 468 bytes | 0 bytes/s, fait.
Total 3 (delta 0), reused 0 (delta 0)

To https://gitlab.cnam.fr/gitlab/fournier/demoProject.git
* [new branch]      master -> master
La branche master est paramétrée pour suivre la branche distante master
depuis origin.
```

Et voilà, la synchronisation pourra s'effectuer maintenant vers Gitlab. En allant sur la plateforme, vous devriez voir quelque chose qui ressemble à la figure *Vue du projet sur Gitlab après un premier commit synchronisé.* ci-dessous.

Après l'exercice, il est maintenant temps de passer à la configuration de votre machine personnelle.

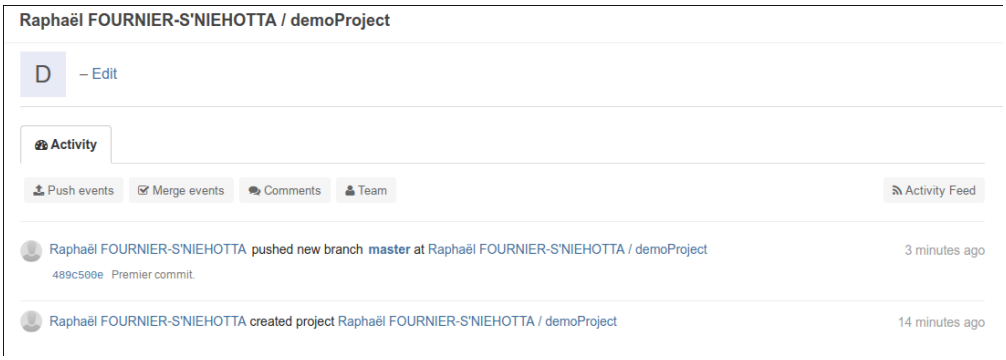

FIG. 14 – Vue du projet sur Gitlab après un premier commit synchronisé.

Exercice : Envoyer un second commit

Reprenez les opérations d'écriture de fichier, d'ajout à l'index et de validation du commit avec un nouveau fichier, *maDeuxiemeJsp.jsp*. Synchronisez avec Gitlab (en utilisant la commande *git push* après avoir commité). Regardez dans l'interface Gitlab l'apparition de vos changements, d'abord sur comme sur la figure précédente, puis en explorant les différents onglets dans le menu de gauche : « Files », « Commits », « Network », « Graphs ».

17.3.4 Chez vous

Sur cette nouvelle machine, vous souhaitez récupérer le code que vous avez envoyé sur Gitlab. Vous ne partez donc pas « de rien », mais d'une synchronisation avec un dépôt existant.

Si vous n'avez pas encore complètement configuré Git, il se peut qu'il vous soit demandé de saisir votre nom et votre adresse mail. Voici un exemple :

```
git config --global user.name "<votre nom complet>"
git config --global user.email "<votre adresse mail>"
```

Ensuite, placez-vous dans un dossier dans lequel vous voulez créer le dossier du dépôt (par exemple : *~/CoursC-NAM/NSY135*). La récupération du dépôt va se faire en utilisant la commande *git clone*, comme suit (l'url étant bien sûr celle que vous obtenez pour votre projet sur la plateforme, cf figure newProjectOKGitlab) :

```
$ git clone https://gitlab.cnam.fr/gitlab/fournier/demoProject.git
Clonage dans 'demoProject'...
Username for 'https://gitlab.cnam.fr': fournier
Password for 'https://fournier@gitlab.cnam.fr':
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Dépaquetage des objets: 100% (3/3), fait.
Vérification de la connectivité... fait.
```

Vous pouvez ensuite entrer dans le dossier et regarder où il en est :

```
$ cd demoProject
$ git status
Sur la branche master
```

(suite sur la page suivante)

(suite de la page précédente)

```
Votre branche est à jour avec 'origin/master'.
rien à valider, la copie de travail est propre
$ git log
commit 489c500ebac812873fe4595c034b53fc89ca8e76
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 14:54:58 2016 +0100

    Premier commit.
```

Vous retrouvez ici votre projet dans lequel vous l'avez laissé.

Écrivons un nouveau fichier, correspondant à une nouvelle vue pour notre projet :

```
$ vim maJspJolie.jsp
```

Une fois que c'est fait, vous obtenez ce qu'on avait vu auparavant, c'est-à-dire un état où il y a des fichiers dans le dossier qui ne sont pas sous le contrôle de Git.

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    maJspJolie.jsp

    aucune modification ajoutée à la validation mais des fichiers non suivis sont␣
↪présents (utilisez "git add" pour les suivre)
```

Évidemment, il faut ensuite ajouter et valider ce changement :

```
$ git add maJspJolie.jsp
$ git commit
[master 014416e] Deuxième commit, avec une nouvelle vue.
 1 file changed, 15 insertions(+)
  create mode 100644 maJspJolie.jsp
```

Maintenant, un *git status* nous informe :

```
$ git status
Sur la branche master
Votre branche est en avance sur 'origin/master' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)
  rien à valider, la copie de travail est propre
```

Nous avons effectué des changements locaux, mais ils n'ont pas encore été publiés sur Gitlab : pour le moment, Gitlab est toujours dans l'état dans lequel vous l'avez laissé en quittant votre machine CNAM la dernière fois (si vous n'êtes pas convaincus, allez vérifier sur la plateforme). Vous pouvez bien sûr effectuer plusieurs commits successifs sans envoyer les changements. Mais, si vous voulez pouvoir réutiliser votre code au CNAM la prochaine fois, il va vous falloir « pousser » (soumettre) vos commits à Gitlab, comme suit :

```
$ git push
Username for 'https://gitlab.cnam.fr': fournier
```

(suite sur la page suivante)

(suite de la page précédente)

```
Password for 'https://fournier@gitlab.cnam.fr':
Décompte des objets: 3, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 543 bytes | 0 bytes/s, fait.
Total 3 (delta 0), reused 0 (delta 0)
To https://gitlab.cnam.fr/gitlab/fournier/demoProject.git
   489c500..014416e  master -> master

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
rien à valider, la copie de travail est propre
```

Le *log* vous affiche donc un deuxième commit :

```
$ git log
commit 014416ec7a561387a63481f75eefff01af69865f
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 16:11:20 2016 +0100

Deuxième commit, avec une nouvelle vue.

commit 489c500ebac812873fe4595c034b53fc89ca8e76
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 14:54:58 2016 +0100

Premier commit.
```

Sur la plateforme, vous devriez maintenant voir votre commit apparaître (rechargez éventuellement la page).

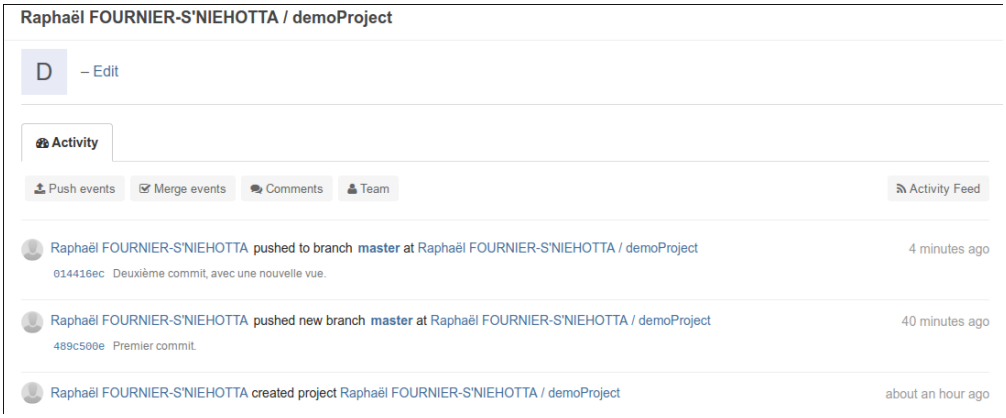

Fig. 15 – Vue du projet sur Gitlab après un deuxième commit, effectué depuis une nouvelle machine.

Voilà, vous pouvez continuer à travailler sur cette machine, en pensant bien à envoyer vos commits une fois que vous avez terminé, pour que vous puissiez récupérer tout cela en salle de cours.

17.3.5 De retour au CNAM

Quand vous arrivez dans la salle du CNAM pour débuter un nouveau cours, si vous avez effectué des modifications chez vous, elles ont été synchronisées avec Gitlab, mais **pas avec le dépôt local sur votre machine du CNAM**. Il est donc **fondamental de commencer par remettre à jour la version locale de votre dépôt** avec celle de Gitlab.

Pour vous en convaincre, regardez avec *git log* :

```
$ git log
commit 489c500ebac812873fe4595c034b53fc89ca8e76
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 14:54:58 2016 +0100

    Premier commit.
```

Pour effectuer cette synchronisation, vous allez utiliser la commande *git pull* :

```
$ git pull
Username for 'https://gitlab.cnam.fr': fournier
Password for 'https://fournier@gitlab.cnam.fr':
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Dépaquetage des objets: 100% (3/3), fait.
Depuis https://gitlab.cnam.fr/gitlab/fournier/demoProject
         489c500..014416e  master     -> origin/master
Mise à jour 489c500..014416e
Fast-forward
 maJspJolie.jsp | 15 +++++++++++++++
 1 file changed, 15 insertions(+)
 create mode 100644 maJspJolie.jsp
```

Celle-ci vous indique de nombreuses informations sur les téléchargements qu'elle effectue afin de synchroniser les versions. Vous obtenez ainsi un dépôt qui contient aussi vos modifications « de chez vous » :

```
$ git log
commit 014416ec7a561387a63481f75eefff01af69865f
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 16:11:20 2016 +0100

        Deuxième commit, avec une nouvelle vue.

commit 489c500ebac812873fe4595c034b53fc89ca8e76
Author: Raphaël Fournier-S'niehotta <raphael@fournier-sniehotta.fr>
Date:   Tue Mar 15 14:54:58 2016 +0100

        Premier commit.
```

Vous pouvez ensuite travailler, ajouter du code, valider (*git add* et *git commit*) et envoyer sur Gitlab (*git push*).

17.3.6 De retour chez vous

Maintenant que la synchronisation a été mise en place, votre machine doit normalement être en mesure de récupérer les modifications effectuées ailleurs à l'aide d'un *git pull*.

17.3.7 À retenir de cette séquence

Après la phase de configuration des machines, la séquence de travail est toujours :

- *git pull*
- *écrire du code*
- *git add <fichiers modifiés>*
- *git commit*
- *écrire du code*
- *git add <fichiers modifiés>*
- *git commit*
- *git push*

N'oubliez ni le **pull** initial, ni le **push** final !

17.4 Références pour aller plus loin

- La référence reste les pages de *man* (sous Unix) : pour chaque commande, vous pouvez taper *man git-commande*.
- Sur les messages de commit : https://ensiwiki.ensimag.fr/index.php/%C3%89crire_de_bons_messages_de_commit_avec_Git
- http://nvie.com/posts/a-successful-git-branching-model/
- le guide complet en anglais sur l'utilisation de Git dans Eclipse