|  |
| --- |
| AUTEUR : NGOUMNAI ETIENNE LEON GAUSS  PROFESSEUR SUPERVISEUR : OLIVIER LIECHTI |
| ANNEE : 2012-2013 |



|  |  |
| --- | --- |
| ***Proposé par Lotaris SA*** | ***Développement d'un outil de benchmarking pour JPA/JavaEE*** |

TRAVAIL DE DIPLOME

Table des matières

[1. Introduction 2](#_Toc363445089)

[2. Java Persistence API 4](#_Toc363445090)

[2.1 L’unité de persistance 5](#_Toc363445093)

[2.1.1 Les Annotations 5](#_Toc363445094)

[2.2 Le manageur d’entité 6](#_Toc363445095)

[2.3 Les classes d’entité 7](#_Toc363445096)

[2.3.1 Héritage 9](#_Toc363445097)

[2.3.2 Les associations entre les classes d’entité 11](#_Toc363445098)

[2.4 Création d’un projet web utilisant JPA sur Netbeans IDE 11](#_Toc363445099)

[3 Application de référence 19](#_Toc363445100)

[3.1 Web 20](#_Toc363445101)

[3.2 Service 21](#_Toc363445102)

[3.3 Le modèle 22](#_Toc363445103)

[3.4 La Base de données 23](#_Toc363445104)

[3.5 Les données de test 26](#_Toc363445105)

[4 Obtention et présentation des métriques 27](#_Toc363445106)

[4.1 Obtention des métriques 27](#_Toc363445107)

[4.1.1 SessionEventListner 27](#_Toc363445108)

[4.1.2 Configuration d’une SessionEventListner 28](#_Toc363445109)

[4.2 Présentation des métriques 31](#_Toc363445110)

[4.2.1 Synthèse des données 31](#_Toc363445111)

[4.2.2 Affichage des données 32](#_Toc363445112)

[5 Conclusion 33](#_Toc363445113)

[6 Bibliographie 34](#_Toc363445114)

# Introduction

Ce document fait foi de rapport du travail de diplôme de bachelor dont l’intulé du sujet est : *Développement d’un outil de benchmarking pour JPA/JavaEE*. Il a été proposé par la start-up Lotaris qui développe une plate-forme de gestion de licences. L’outil devra intégrer une chaine permettant de faire face aux contraintes de performance et de disponibilité qu’impose l’application.

Le développement d’applications à plusieurs couches interagissant entre elles (applications multi-tiers), en particulier avec une base de données comme couche, nécessite l’utilisation d’outils répondant le mieux à nos besoins. Java Enterprise Edition (Java EE), qui est l’une des plates-formes les plus utilisées dans le marché du développement informatique, met à disposition plusieurs interfaces de programmation (en anglais on utilise l’acronyme API) parmi lesquels *Java Persistence API (JPA)*.

JPA, par ses interfaces, définit des règles permettant la gestion de la correspondance entre les objets Java et une base de données indépendamment de celle-ci. Elle offre la possibilité aux développeurs d’écrire un code simple et minimaliste ayant une faible dépendance aux serveurs d’applications et aux différentes couches d’une application multi-tiers.

La conception du programme (les relations entre les différentes classes du programme, le type des attributs) étant pris en compte par JPA, il serait capital de faire attention aux différents choix d’implémentation. Mettons-nous dans le cadre d’un développeur qui utilise JPA pour la persistance des données de son application dans une base de données décrite comme suite : une table A en relation bidirectionnelle avec une table B et celle-ci ayant une relation bidirectionnelle avec une table C. Le développeur, en voulant lister les données de la table A où juste quelques informations de cette table uniquement lui seront intéressantes, pourra se retrouver avec comme résultat, selon les stratégies adopter lors de l’implémentation de son application, aussi bien les données de la table A que celles des tables B et C. Ceci pourra fortement dégrader les performances en termes de temps, l’utilisation de la mémoire, du processeur et même plus, si les données doivent transiter sur un réseau internet, une bonne partie de la bande passante sera utilisée inutilement. Il faut donc faire attention aux différentes stratégies utilisées.

Dans une équipe de développeur utilisant JPA pour un projet donné, plusieurs personnes pourraient être appelées à apporter des modifications au projet tout au long de son élaboration. Une modification pourrait apporter un gain ou une perte de performance, elle pourrait entrainer un mauvais disfonctionnement du projet ou d’une partie du projet. L’utilité avantageuse de JPA ne devant pas priver les développeurs de comprendre l’impact des choix d’implémentation faits, il leur sera important d’avoir à disposition un outil de test ou de monitoring, d’où la naissance du projet qui fait l’objet de notre travail de diplôme.

L’objectif de ce travail de diplôme est de :

* Etudier et mettre en pratique la technologie JPA par le biais d’une petite application. Cette application servira de référence et permettra d’effectuer des tests de performance sous la base des données de test et de différente architecture de l’application.
* Voir comment des sondes peuvent être développées pour faire différents types de mesures sur les requêtes effectuées. A savoir, le nombre de fois qu’une requête a été faite, les différentes dates d’exécutions de cette requête, la durée d’exécution de chaque requête.
* Intégrer les sondes dans un processus automatisés pour que les résultats puissent être obtenus et partagés chaque jour. Ce qui permettra voir les variations des performances chaque jour et de savoir à quel moment il y a eu une dégradation de performance et à quelles entités sont concernées par cette dégradation.

La figure suivante illustre le travail à faire.

Figure 1‑1

Représentation des métriques

Application de référence

Requête HTTP

Code de l’application

JPA

Rest Client

Sonde

cURL

Utilisateur

Base de données des métriques

Base de données de l’application

Synthèse des données

Un utilisateur effectuera des requêtes HTTP via un outil permettant de le faire (navigateur, cURL[[1]](#footnote-1)…). Ces requêtes seront reçues et interprétées par l’application (au niveau du code de l’application). L’application, via JPA va interroger la base de données et répondre à l’utilisateur. La sonde est connectée sur un point d’entrée/sortie de JPA avec la base de données de façon à intercepter toutes les opérations. Les données reçues de la sonde seront par la suite synthétisées puis archivées dans la base de données et représentées à l’utilisateur.

En d’autres termes, les utilisateurs auront à naviguer à travers l’application en utilisant par exemple un navigateur de leur choix pour charger des données ou obtenir une ressource de la base de données et notre outil analysera les opérations effectuées entre l’application et la base de données. Les résultats de l’analyse seront sauvegardés dans une autre base de données et représentés sur une page web pour qu’on puisse observer la variation des performances.

La suite de ce document ce présentera comme suite :

* Un résumé de l’étude de JPA, qui permet de comprendre ses spécificités et sa mise en œuvre. Cette étude permettra de développer une application utilisant JPA.
* La présentation de l’application référence utilisant la technologie JPA. Cette application devant être simpliste, devra intégrer une partie du concept de la programmation orienté objet (liaison dynamique, héritage), elle devra avoir des classes en relation unidirectionnelle, bidirectionnelle avec plusieurs types de cardinalité.
* Une explication du mode opératoire pour obtenir des métriques et leur représentation. Ces métriques permettront d’observer le comportement de JPA. Elles permettront de développer les sondes pour collecter les données qui serviront aux prises de mesure.
* Un état des lieux du travail effectué

# Java Persistence API

JPA est issue de *Java Specification Requests* (JSR), plus précisément la JSR 220 et s’inscrit dans la liste des ORM tout comme Hibernate de Java, iBATIS de .Net Active record de Ruby Doctrine de PHP ou encore Django de Python.

Une ORM ou object-relational mapping en français mapping objet-relationnel est une technique de programmation qui permet, par le biais des frameworks, de faire le lien entre les objets d’un langage de programmation et une base de donnée relationnelle, ce qui donne l’illusion d’avoir une base de données orientée objet.

Pour les développeurs Java, un ORM est une alternative à JDBC (Java DataBase Connectivity). JDBC est une interface de programmation utilisant la plateforme Java et permettant aux applications d’accéder à une base de données grâce à des pilotes. L’utilisation de JDBC pour communiquer avec une base de données nécessite la conversion des données objets en données relationnelles et inversement. Par conséquent, les développeurs seront appelés à programmer cette conversion pour chaque objet ce qui conduit à produire du code similaire pour chaque conversion. Ils devront également écrire toutes les requêtes nécessaires à leur programme. L’écriture de ces opérations de conversion ainsi que des différentes requêtes devient fastidieux dès lors que les relations entre objets deviennent complexes et que la base de données comporte un important nombre de tables. A chaque modification faite au niveau du programme, il faudra tenir compte des requêtes existantes, de la base de données et agir en conséquence.

L’utilisation de JPA permet de palier à ce problème, elle permet de générer des requêtes, de construire et maintenir les tables de la base de données automatiquement. Ceci ce fait par le biais des annotations et d’un fichier de configuration assez particulier appelé *Persistence*.xml (XML pour Extensible Markup Language en français langage de balise extensible).

L’on devrait noter que, lorsqu’on parle de persistance des données en informatique comme c’est le cas ici, c’est qu’il s’agit d’une technique permettant de sauvegarder l’état d’une donnée (d’un objet) de façon à ce qu’il soit toujours accessible même après que le programme qui lui a donné naissance soit terminer.

JPA repose sur trois principaux éléments qui sont : l’unité de persistance (en anglais *Persistence Unit*), le manageur d’entité (en anglais *Entity Manager*) et les classes d’entité (en anglais *Enties class*). Dans la suite, on parlera de ces éléments plus en détail.



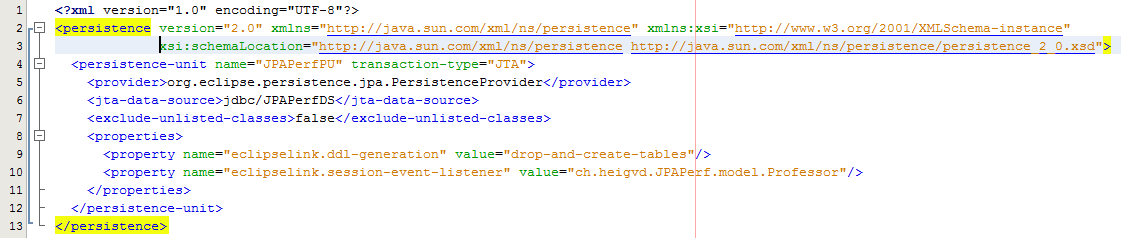
## L’unité de persistance

L’unité de persistance est l’élément qui, par le biais des annotations ou du fichier XML équivalent, gère la façon dont les classes d’entité sont mappées dans la base de données. Elle est définie par trois autres éléments :

* Les métadonnées sur les classes d’entité. Ce sont les annotations dont nous parlerons plus en détail au point 2.1.1.
* La *Persistence Manager Factory*. Elle est définie par la classe *EntityManagerFactory*. Un objet de cette classe définit la configuration complète de l’unité de persistance lors de l’exécution de JPA.
* Le fichier de configuration *Persistence.xml*. Ce fichier permet de définir :
* Le nom de l’unité de persistance. Il peut y avoir plusieurs unités de persistance, d’où l’intérêt de les nommer pour pouvoir les distinguer.
* Le *Persistence Provider*. Il fournit l’implémentation de la spécification de JPA. C’est lui qui gère réellement le couplage entre les classes d’entité et la base de données tel que c’est spécifié dans les métadonnées de ces classes d’entité. On peut citer comme *Persistence Provider*, *HibernatePersistence*, *EclipseLink* et *Sprint*…
* La base de données dans laquelle les données seront enregistrées. Et le type de transaction à utiliser.
* Les classes qui sont concernées par la persistance.
* Les différentes propriétés de l’unité de persistance telles que la façon dont la base de données sera générée, le monitoring des sessions liées à l’application etc.

On peut voir un exemple de fichier *Persistence*.*xml* sur la figure suivante.

Figure 2‑1



### Les Annotations

Les annotations ont été introduites dans Java SE 5. Elles permettent d’ajouter des métadonnées à un code source Java. C’est une stratégie ou la façon dont un objet devra être utilisé. Elles peuvent être ajoutées aux classes, méthodes, attributs, paramètres de méthodes. Leur syntaxe est sous la forme « @ métadonnée», par exemple : *@Entity*, *@Id*. On retrouve ces annotations dans le package *javax.Persistence*.

Lorsqu’on utilise les annotations, lors de la compilation du code source, compilateur Java va se charger de mettre ces métadonnées dans un fichier classe et l’unité de persistance les utilisera comme expliqué plus haut.

Les annotations sont une alternative aux fichiers de configurations XML. Au lieu d’utiliser les annotations dans un programme, on peut écrire un fichier XML équivalent qui produira le même comportement. Mais en général on préfère utiliser les annotations pour leur simplicité.

JPA utilise un grand nombre d’annotation qu’on peut regrouper comme suites :

* Les annotations pour les classes
* Les annotations pour les champs des classes
* Les annotations pour les relations entre classes
* Les annotations pour les modes d’accès
* Les annotations pour la génération des valeurs
* Les annotations pour les méthodes de callback
* Les annotations pour les requêtes JPQL
* Les annotations pour Java EE
* Les annotations pour le mappage (ORM)
* Les annotations pour les requêtes SQL

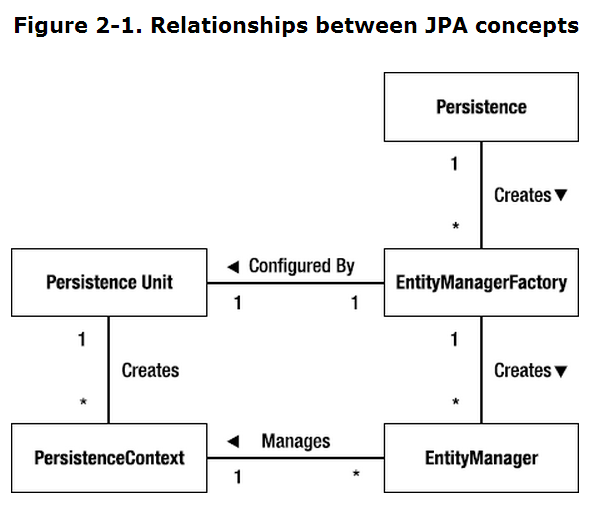
On peut annoter les classes, les attributs et les méthodes. Pour le faire, il suffit de placer l’annotation soit sur la même que le composant à annoter soit la ligne d’avant. S’il y a plusieurs annotations pour le même composant, on peut les placer les uns après les autres.

## Le manageur d’entité

Comme son nom l’indique, le manager d’entité est le composant qui va manager une classe d’entité. Il est défini par la classe Java *EntityManager*. Sa création se fait par le biais de la classe *EntityManagerFactory* qui, pour le faire, va récupérer les métadonnées de l’unité de persistance et les interpréter. Une fois l’objet *Entitymanager* créé, il ne pourra que manager un seul type de classe d’entité.

Pour une meilleure compréhension des relations, on se réfère à la figure 2-1 du livre Pro JPA 2 : Mastering the Java *Persistence* API.

Figure 2‑2



On y voit qu’un *EntityManager* est créé par un seul *EntityManagerFactory* et peut manager un seul contexte de persistance. Un contexte de persistance existe dans une seule unité de persistance.

Un contexte de persistance est un groupe d’entité (des objets Java) de même type managé par un *EntityManager*. On rappelle qu’objet *EntityManager* est configuré pour opérer sur un seul type de classe d’entité et sur une seule base de données. Un objet d’une classe d’entité donnée est unique dans un contexte de persistance et après sa création, sa référence est passée à l’objet *EntityManager* pour être persister dans la base de données.

Les opérations des *EntityManager* sur les classes d’entité sont appelées CRUD, c’est-à-dire *Create*, *Read*, *Update*, *Delete*. Ces méthodes font appel aux méthodes *persist(), find(), merge() , remove()* qui interagissent avec la base de données via des requêtes SQL.

## Les classes d’entité

Une classe d’entité est une classe qui sera mappée dans la base de données comme étant une table de cette base de données. Les champs de cette classe seront automatiquement persistés dans cette table et représenteront les différentes colonnes da la table. Pour transformer une classe en entité, il faut lui ajouter deux annotations :

* « *@Entity* ». Cette annotation est placée avant la déclaration de la classe à laquelle elle est associée et indique la classe devra être traitée comme étant une entité. La table associée à cette classe dans la base de données portera le nom de la classe. Si on veut changer ce nom, il suffira d’ajouter l’élément « name » comme tel *@Entity(name = ``XXX ´´)* où XXX représente le nom qu’on veut attribuer à la table.
* « *@Id* ». Elle permet d’annoter le champ qui sera utilisé comme clé primaire dans la base de données. Ce champ ne peut être de n’importe quel type. Il est restreint aux types suivants :
* Type primitif Java : *char, byte, short, int, long*
* Classe wrapper Java : *Character, Byte, Short, Integer, Long*
* Type String de *java.lang.String*
* Type numeric *: java.math.BigInteger*
* Types temporal : *java.util.Date et java.sql.Date*

Les types *float, double,* *Float, Double,* *java.math.BigDecimal* sont aussi possibles mais sont sujets à erreur à cause des arrondis et la non fiabilité du résultat de leur comparaison.

Une classe d’entité ressemble beaucoup à une classe JavaBeans, qui est en fait un composant logiciel écrit en Java, encapsulant plusieurs objets pour répondre à un besoin métier [1]. Cette ressemblance est dû au fait qu’une classe d’entité possède en général les propriétés suivantes :

* Un constructeur sans argument (constructeur par défaut).
* Des accesseurs pour tous les attributs déclarés privés.
* La classe devrait implémenter l’interface « *Serializable[[2]](#footnote-2)* » pour pouvoir sauvegarder et restaurer l’état des objets lors des transactions.
* La classe ne doit pas être déclarée « final ».

A ce niveau, on peut citer quelques annotations qui sont souvent rencontrées :

* *@GeneratedValue* : elle s’applique à l’attribut qui sert de clé primaire et permet de définir la stratégie utilisée pour générer cette clé.
* *@Colum(name = ``XXX´´)* : elle s’applique à n’importe quel attribut et permet de modifier son nom dans la base de donnée. XXX représente le nom à attribuer.
* *@Enumerated* : Elle s’applique à un attribut de type énuméré.

On peut voir un exemple de classe d’entité sur la figure suivante.

Figure 2‑3



### Héritage

Comme nous l’avons mentionné plus haut, une classe d’entité est avant tout une classe Java. De ce fait, elle bénéficie des mécanismes Java tel que l’héritage. Mais le comportement attendu de la classe dérivée pourra être surprenant.

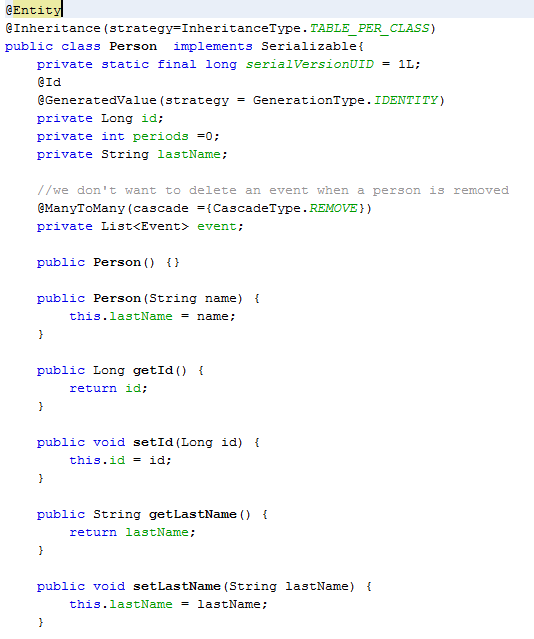
Une classe d’entité peut héritée d’une autre classe n’étant pas une. Dans ce cas, un objet d’une telle classe héritera bien de la super classe mais l’état de la super classe ne sera pas persisté. En d’autres termes, les attributs et méthodes hérités de la super classe ne seront pas pris en compte par le *Persistence Provider* définit dans le fichier *Persistence*.*xml* et dans la base de données on retrouvera uniquement les attributs propres à la sous classe. De plus, la super classe ne sera pas persisté.

Une classe d’entité peut étendre une autre classe d’entité. Dans ce cas, les deux classes seront persistées et leur représentation dans la base de données dépendra de la stratégie optée par le développeur. On peut citer comme stratégie :

* Une seule table pour la super classe et la classe dérivée. Cette stratégie est mise en place via l’annotation *« @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)* ». Les attributs de la super classe et de toutes ses classes dérivées seront regroupés dans une seule table pour former les colonnes de cette table. La valeur « NULL » sera affectée aux colonnes n’étant pas référencées dans une classe donnée.
* Une table par classe d’entité. Cette stratégie est mise en place via l’annotation « *@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)* ». Les attributs de la super seront recopiés dans chaque sous classe lors de la génération de la base de données et constitueront ainsi des colonnes dans les tables représentant les sous classes.

La figure suivant montre un exemple de super classe.

Figure 2‑4



Dans le cas où une classe d’entité dérive d’une autre, on n’aura plus besoin de définir un attribut qui servira de clé primaire (annotation *@Id*). Cet attribut sera hérité de la super classe ainsi que sa stratégie de génération.

### Les associations entre les classes d’entité

JPA permet de définir les relations entre les classes. C’est-à-dire la façon dont les classes sont liées entre elles. Les différents types d’associations et leur annotation sont les suivants :

* Association one to one, annotée par « *@OneToOne* ». Elle indique un objet d’une classe A est en relation avec un objet d’une classe B uniquement.
* Association one to many, annotée par « *@OneToMany* ». Elle indique un objet d’une classe A est en relation avec plusieurs objets d’une classe B.
* Association many to one, annotée « *@ManyToOne* ». Elle est quasiment identique à la précédente et indique plusieurs objets d’une classe A peuvent être en relation avec un même objet d’une classe B.
* Association many to many, annotée « *@ManyToMany* ». Elle indique plusieurs objets d’une classe A peuvent être en relation avec les objets d’une classe B et inversement.

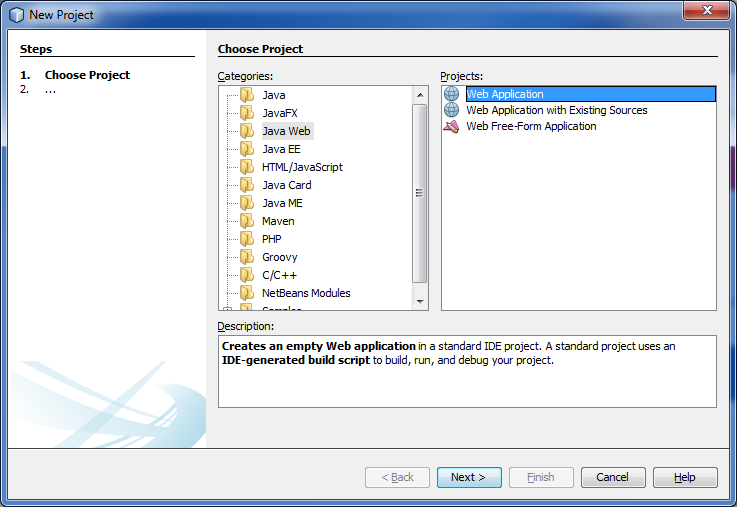
Les associations sont unidirectionnelles ou bidirectionnelles. Dans le cas d’une association unidirectionnelle, une seule classe possède la référence sur l’autre classe. Dans le cas d’une association bidirectionnelle, chaque classe possède une référence sur l’autre classe et il faut spécifier la classe qui possède la relation en ajoutant l’attribut « *mappedBy* ». On peut voir un exemple dans la Figure 2-2 *@ManyToMany(mappedBy= ``courses´´)* ce qui veut dire que dans la relation *Student-Course*, la classe *Student* possède la relation et le mappage se fait sur l’attribut *courses* de *Student*.

## Création d’un projet web utilisant JPA sur Netbeans IDE

Démarrer un projet web utilisant JPA comme *ORM* est assez simple. La démarche à suivre est la suivante :

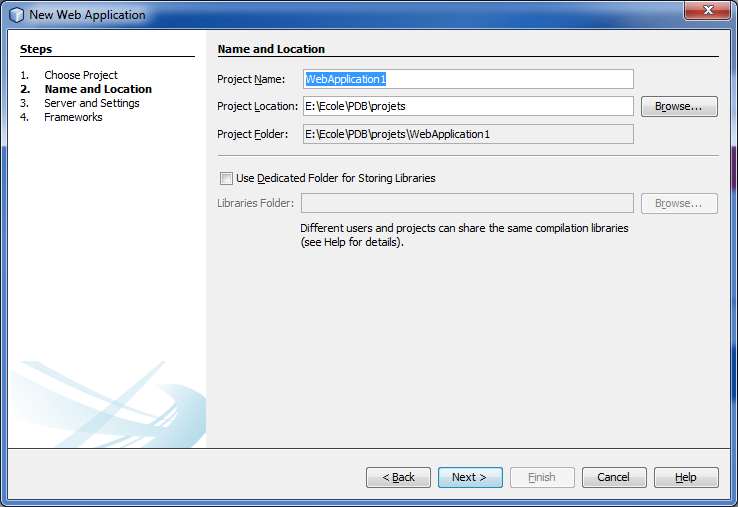
* Depuis la page d’accueil de Netbeans, sélectionner nouveau projet.
* Dans Categories sélectionner « Java Web » et dans Projets, sélectionner « web Application »

Figure 2‑5



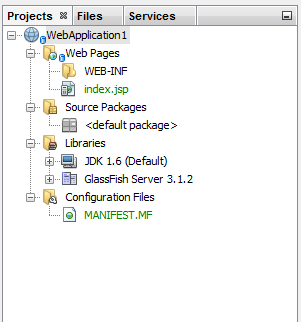
* En suite cliquer sur « next », donner le nom du projet, l’emplacement, puis cliquer sur « Next » et en fin sur « Finish »

Figure 2‑6



Le projet web est créer et on obtient l’arborescence suivante.

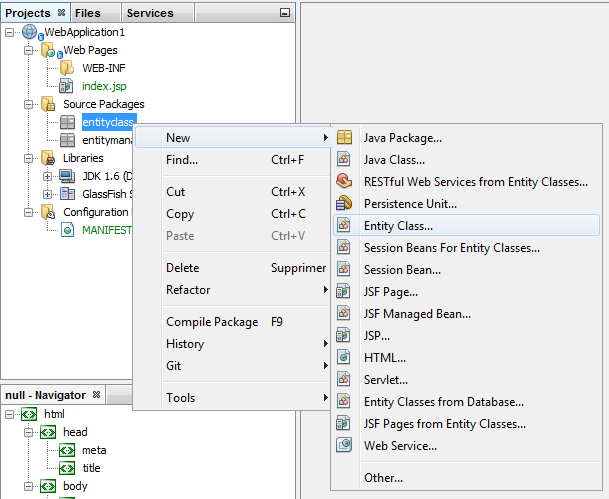
Figure 2‑7



Dans Source Packages on va créer les packages « entityclass » et « entitymanager ».

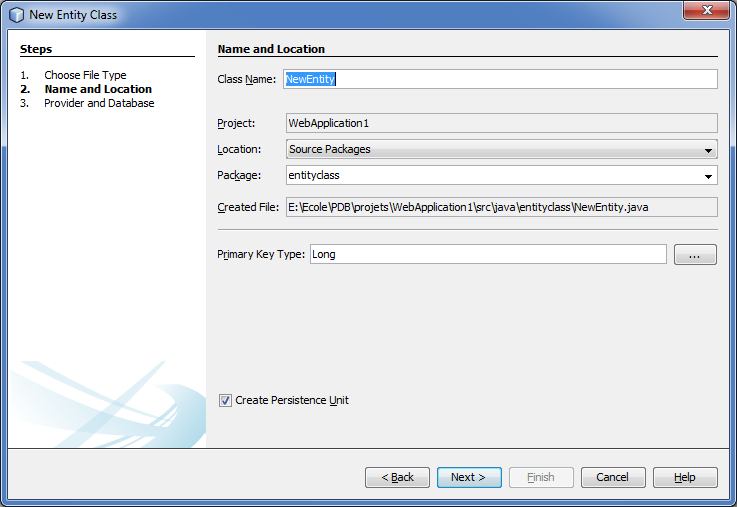
Dans le package « entityclass », on fait un clic droit et on sélectionne New > Entity Class

Figure 2‑8



* Dans la fenêtre qui s’affiche, saisir le nom de la classe. On peut remarquer le champ Primary Key Type qui est par défaut de type « Long ». On ne va pas le changer, il indique que la clé primaire sera de ce type. On remarquera aussi qu’il y a une case « Create Persistence Unit » cocher par défaut.

Figure 2‑9

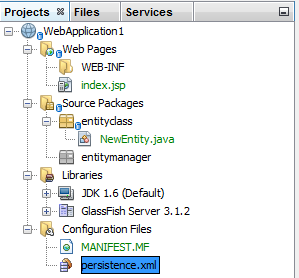


* Cliquer sur Next. Dans la fenêtre suivant, on donne les informations qui seront contenues dans le fichier peristence.xml.
* *Persistence* *Unit* Name : On peut ne pas donner le nom de l’unité de persistance dans le cas où il n’y a qu’un seul.
* *Persistence* *Provider* : Par défaut, c’est *EcliseLink(JPA 2.0)* mais on peut le modifier
* Data Source : On indique la base de données à utiliser pour persister les entités. Elle devrait avoir été créée au préalable.
* Table Generation Strategy : On indique ici la stratégie de création des tables la base de données.

Toutes ces informations pourront être modifiées dans le fichier *Persistence*.xml.

* Après avoir cliqué sur Finish, on obtient une nouvelle arborescence contenant le fichier *Persistence*.xml

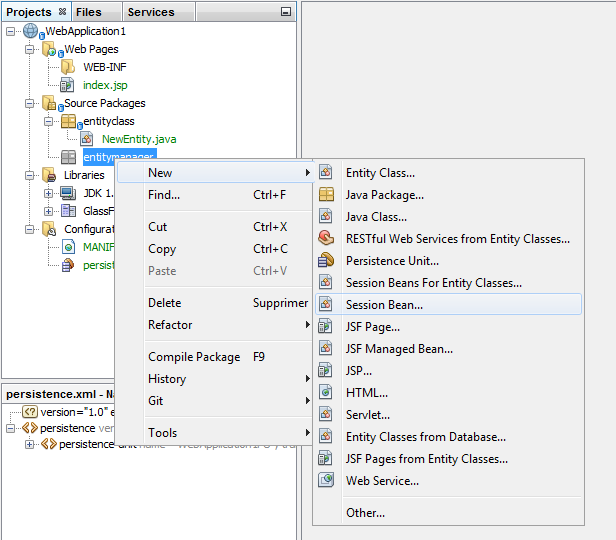
Figure 2‑10



La création du manager d’entité est quasiment similaire.

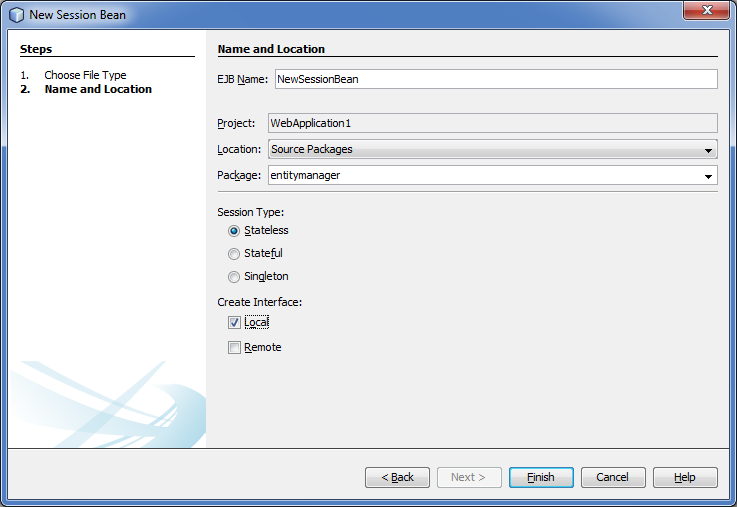
* On fait un clic droit sur le package *entitymanager* puis New > Session Bean

Figure 2‑11



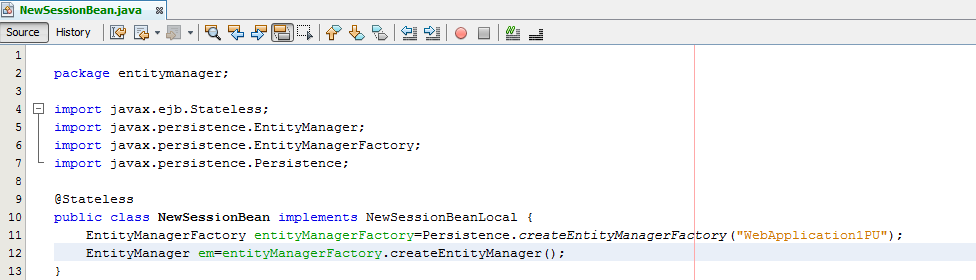
* Dans la fenêtre qui apparait, on peut sélectionner « Stateless » et « Local » puis cliquer sur Finish

Figure 2‑12



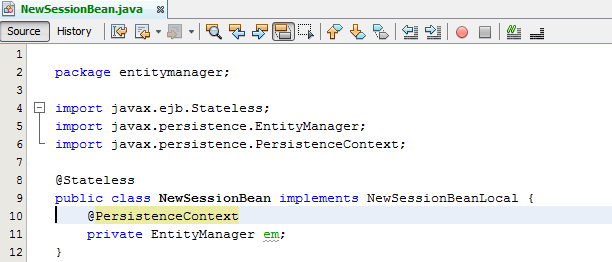
On obtient un EJB dans lequel on peut créer et initialiser le manageur d’entité pour la classe d’entité précédemment créée. Cette création peut se faire soit à main comme suite :

Figure 2‑13



On peut également dédier la création du manager d’entité au conteneur, *Persistence* *Provider* *EclipseLink* par exemple.

Figure 2‑14



On pourra retenir de ce qui précède que JPA est une technologie qui s’appuie sur plusieurs autres technologies masquées par l’utilisation des annotations ce qui confère au développeur une grande facilité dans leur travail. Mais ces derniers devraient bien se document er sur JPA avant de se lancer dans un projet.

Les contours de la technologie JPA ayant été explorés, on peut se lancer dans une application pour la mettre en pratique et plus tard servira à automatiser les tests de performance.

# Application de référence

Pour mieux comprendre le fonctionnement de JPA et faire des tests de performance, on s’est proposé de développer une application de référence.

Le matériel utilisé pour cette application est le suivant :

* Netbeans IDE 7.3
* GlassFish 3.1.2.2
* Base de données MySQL

L’architecture de notre application devra respecter le schéma suivant :

Figure 3‑1

Base de données

web

modèle

service

Chacune des structures web, service, modèle représente un package dont nous parlons dans la suite.

## Web

On se trouve ici dans le package ch.heigvd.jpaperf.web de notre application. Les classes de ce package utilisent Java API for RESTful Web Services (*JAX-RS*) qui est une interface de programmation Java permettant de créer des services Web avec une architecture *REST* (Representational Satate Transfert).

Ces classes (appelées objet métier) nous donnent la possibilité d’écrire des requêtes sous forme d’URL pour interroger la base de données. Ces requêtes peuvent être écrites via un navigateur (Internet Exploreur, Mozilla, Google Chrome…), elles peuvent être écrites en ligne de commande grâce au client *REST cURL* (Client URL Request Library) qui est une interface en ligne de commande permettant de récupérer une ressource désignable par une URL. Un exemple de classe est illustré si dessous.

Figure 3‑2



Lorsqu’on envoie une requête, c’est le moteur GlassFish qui va se charger de le diriger vers la classe concernée. Ensuite, grâce au chemin indiqué dans la requête, on arrivera sur la méthode en charge de traiter cette requête. Le chemin est indiqué dans la classe par l’annotation « *@Path* ». Le type de données traitées par l’objet métier est indiqué par l’annotation « *@Produces* ». Dans notre cas, on utilise des objets JSON (JavaScript Object Notation) qui encapsule des DTO (Data Transfer Object).

Les DTO ont une importance sécuritaire ici en ce sens que, hors de l’application, l’on n’est pas obligé de savoir exactement comment sont formées nos classes d’entité ou qu’est ce qui est en registré dans la base de données. Alors, pour chaque classe d’entité, on crée une DTO associée qui renferme les informations (attributs) que nous acceptons faire connaitre hors de l’application. Ainsi, lorsqu’on répond à une requête, en l’occurrence une requête de type GET, on ne transmet pas l’objet résultant de la base de données mais plutôt la DTO qui est en quelque sorte son image. De la même façon, lorsqu’on reçoit un objet DTO, on le transforme en entité avant de le manipuler.

## Service

On est ici dans le package ch.heigvd.jpaperf.services. Ce sont en fait des EJB (Enterprise JavaBeans) Session Statelless dans lesquels on a injecté un *EntityManager* pour manager les classes d’entité. Ces EJB implémentent des interfaces qui mettent à disposition des méthodes de service.

Un EJB Session Stateless est un objet Java qui met à disposition des services et méthode qu’il contient [2]. Du fait qu’il soit Stateless, il reste les propriétés suivantes :

* Aucune donnée n’est retenue ni enregistrée, c’est-à-dire qu’aucun état n’est retenu, d’où Stateless.
* Aucun mécanisme ne garantit que deux appels consécutifs à une méthode d’un tel EJB visent une seule et même instance.
* Les accès concurrents sont impossibles.

Il existe aussi les EJB Session Stateful, mais ceci sont imports dès lorsqu’on a besoin de conserver l’état d’une session. Ce qui n’est pas le cas pour nous.

Nous avons dit qu’un *EntityManager* créé par un *EntityManagerFactory*. Ici, nous ne créons pas nous même les *EntityManager*. On les injecte dans les EJB via l’annotation « *@PersistenceContext* » et le conteneur *EclipseLink* se chargera de leur création et leur initialisation. On peut voir un exemple sur cette figure.

Figure 3‑3



Sur cette figure, l’annotation « *@Stateless* » veut dire qu’il s’agit d’un EJB Stateless. L’interface *CourseManagerLocal* définit les méthodes (services) qu’utilisera l’EJB. La super *classeAbstractManager* définit les méthodes permettant de manager les classes d’entité, en l’occurrence les méthodes :

* create. Elle fait appel à la méthode persiste qui permet de persister un objet passé en paramètre.
* read. C’est l’ensemble des méthodes appelées « finder ». Elles permettent de rechercher un objet.
* update. Cette méthode fait appel à la méthode merge qui permet de mettre à jour un objet déjà persisté qui a été édité.
* delete. Elle fait appel à la méthode remove qui permet de supprimer un objet persisté.

## Le modèle

Le modèle constitue le package *ch.heigvd.jpaperf.model*. Il contient les différentes classes d’entité reliées entre elles et qui seront persistées dans la base de données. Il représente notre application qui a pour but, l’affectation des cours à des étudiants et des professeurs. Ce qui nous permettait d’illustrer l’héritage, l’association unidirectionnelle et bidirectionnelle et plusieurs types de cardinalité. Dans le but d’ajouter une légère complexité, on a décidé de gérer les évènements : maladie, conférence, réception etc. Le diagramme UML de notre application est le suivant :

Figure 3‑4



Dans ce diagramme, la classe Person représente une classe d’entité, son état sera donc persisté dans la base de données. Les entités Professor et Student héritent de la classe. La relation Student-Course est une relation bidirectionnelle ManyToMany, ce qui veut dire que :

* connaissant un étudiant, on peut savoir quels cours il suit et inversement, connaissant un cours, on peut savoir quels étudiants suivent ce cours. Ceci est possible grâce à cette relation bidirectionnelle qui oblige chaque partie à garder une référence (plus précisément une liste en vertu de la multiplicité de la relation) sur l’autre comme nous l’avons expliqué plus haut.
* Dans la base de données, une nouvelle table sera créée pour représenter l’association entre les tables Student et Course. Cette table aura pour clé primaire, le couple de clé constitué des clés primaires des deux tables Course et Student.

La relation Course- Professor est bidirectionnelle ManyToOne. Dans ce cas, compte tenu de la stratégie de génération des tables héritées, au niveau de la base de données, la table PROFESSOR n’aura pas une colonne référençant la clé primaire de la table COURSE, une jointure sera faite avec celle-ci pour retrouver tous les cours dispensés par un professeur. La table COURSE par contre aura une colonne référençant la clé primaire de la table PROFESSOR. C’est sur cet attribut de COURSE que la jointure sera faite pour obtenir les cours dispensés par un professeur.

Specialization est un type énuméré qui fera office d’attribut dans la classe Person en vertu de l’héritage, elle constituera une colonne dans les tables PROFESSOR et STUDENT.

La relation Person-Event oblige la classe Person à avoir une référence sur la classe Event, on aura plus précisément une liste d’Event. Les classes Professor et Student dérivant de Person, on aura l’impression d’avoir les relations Professeur-Event et Student-Event et dans la base de données, on aura alors les tables mettant en relation les classes Professor et Event d’une part, Student et Event d’autres part, comme si l’association avait directement été faite sur ces deux classes. Ces tables sont PROFESSOR\_EVENT et STUDENT\_EVENT.

## La Base de données

On travaille avec une base de données MySQL. Elle a été créée et ajouté au fichier *Persistence*.xml pour être prise en compte par notre conteneur *EclipseLink*. Ainsi, JPA s’occupera de la génération et de la maintenance des tables de cette base de données en fonction des stratégies et annotations utilisées.

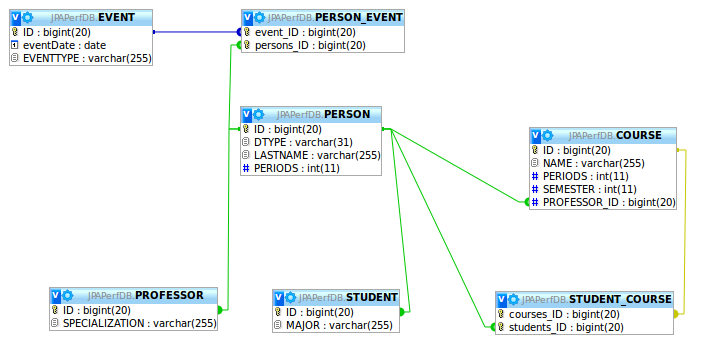
Comme stratégie, on a :

* Create. Elle indique à JPA qu’on veut que les tables soient créées et que la mise à jour des tables se fasse sans que la version créée au préalable ne soit effacée.
* Drop and Create. Cette stratégie indique qu’à chaque mise à jour, les tables précédemment créées doivent être supprimées et de nouvelles tables seront créées par la suite. Cette stratégie nous a paru assez intéressante dans la mesure où elle nous permet d’avoir une base de données fidèle au modèle de notre application.
* None. Elle indique qu’il n’y a pas de stratégie.

La stratégie optée pour l’application est Drop and Create. Nous avons dit plus haut que JPA prenait en compte les annotations pour la génération de la base de données. Reprenons le concept de l’héritage. On a à disposition trois annotations définissant comment représenter la classe parent Person et les classes enfants Professor et Student dans la base de données:

* *@Inheritance (strategy=InheritanceType.JOINED)*

Figure 3‑5



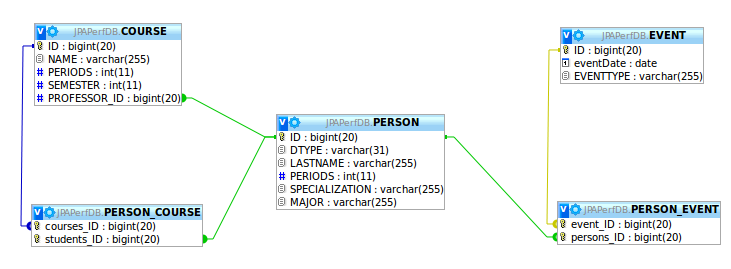
Bien que nous ayons établi les relations Professor – Course et Student – Course, l’annotation impose que dans la base de données, les associations avec les classes dérivant de la classe Person se fassent au niveau de la classe parent.

On peut remarquer que dans la table STUDENT\_COURSE, c’est bien la clé primaire (students\_ID) de la table STUDENT qui est utilisée et non celle de la table PERSON comme on pouvait le penser.

L’absence d’une table PROFESSOR\_COURSE s’explique par la cardinalité de l’association (OneToMany). On a plutôt une clé étranger référent la table PROFESSOR dans la table COURSE.

* *@Inheritance (strategy=InheritanceType.SINGLE\_TABLE)*

Figure 3‑6



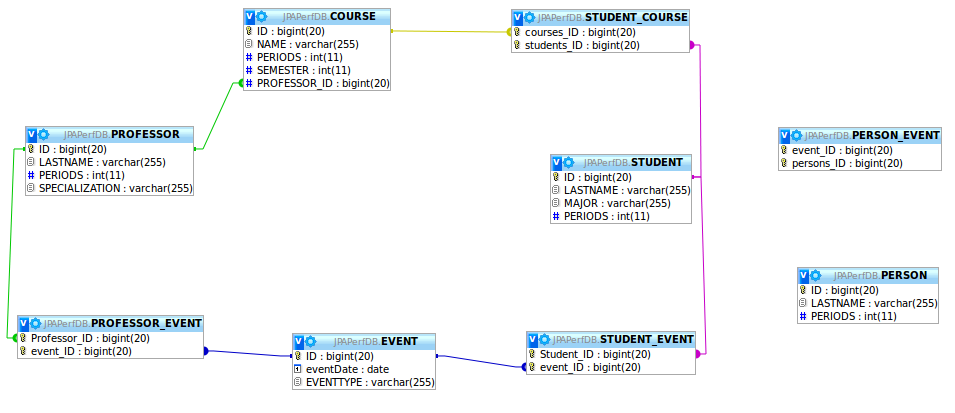
Cette annotation ou stratégie, impose à la base de données à avoir une seule classe pour représenter la classe parent Person et les classes enfant Professor et Student.

On peut remarquer que les attributs des classes Professor et Student sont regroupés dans la table PERSON.

Dans la table PERSON\_COURSE on a comme clé students\_ID comme si on avait une table STUDENT dans la base de données, tout comme dans la table COURSE ou on a la clé PROFESSOR\_ID.

* *@Inheritance ( strategy=InheritanceType.TABLE\_PER\_CLASS)*

Figure 3‑7



Ici, on crée une table pour chaque classe. Les classes enfants héritent des attributs de la classe parent en l’occurrence l’attribut PERIODS.

Bien que dans le modèle on avait l’association Person – Event, dans la base de données, l’association ce fait sur les tables des classes réellement référencées.

## Les données de test

Nous avons besoin d’un grand nombre de données pour peupler notre base de données à fin de faire tourner notre application et prendre des mesures de performance. Ces données correspondent aux noms des professeurs, aux noms des étudiants, aux noms des cours, aux différentes spécialisations et différents évènements.

Nous avons parcouru les sites web à la recherche des noms et on a créé un fichier de nom de professeur, d’étudiant, de cours. Ces fichiers nous permettent de générer 485 étudiants, 208 professeurs, 458 cours, 25 spécialisations et 200 évènements.

Parmi les mesures à prendre, on compte mesurer le temps de traitement de requête. Pour garder une cohérence lors des prises de mesures, nous avons développé dans nos EJB des méthodes de services appelées « Business Method ». Ces méthodes nous permettent par exemple, d’affecter des cours à un étudiant, à des professeurs. Elles prennent en paramètres, le nombre de cours à attribuer à un professeur ou un étudiant, le nombre d’évènements à affecter. Ainsi, notre base de données aura en permanence un nombre fixe d’entrées par table.

La simplicité de notre application réside dans le fait que nous désirons apprendre et non révolutionner ce type d’applications. De plus, nous avions besoin d’un benchmark pour réaliser des tests.

On peut remarquer que notre application fait intervenir plus concepts Java EE. On navigue entre plusieurs couches, du client Rest à la base de données en passant par le serveur d’application GlassFish, JAX-RS, les EJB et JPA. On a vu que les annotations avaient un impact dans la construction des tables de la base de données. A l’aide des métriques, on pourra mesurer le coup en termes de performance sur le choix des différentes stratégies.

# Obtention et présentation des métriques

Les métriques sont les indicateurs de qualité d’un logicielle. Dans notre cas en particulier, elles nous permettrons de déterminer les performances de JPA. N’étant pas par défaut à la disposition des utilisateurs, il faut faire une recherche pour obtenir ce qu’on veut comme métrique.

## Obtention des métriques

Pour retrouver ces métriques, nous nous sommes documentés sur *EclipseLink* qui est un service de persistance créé par la fondation Eclipse. *EclipseLink* met à la disposition des programmeurs Java des framework permettant d’interagir avec les bases de données, les services web, les objets XML mappés. Et il supporte plusieurs standards de persistance en l’occurrence JPA.

Comme indicateur de performance, on a trouvé l’interface *SessionEventListner*, *performance profiling*.

### SessionEventListner

Lorsqu’une opération doit être effectuée, une session est créée et cette session peut avoir plusieurs évènements. *EclipseLink* nous met alors à disposition l’interface *SessionEventListner*. Elle nous permet de recevoir tous les évènements générés lors d’une transaction entre l’application et la base de données. Nous recevons 36 évènements qui correspondent à 36 méthodes qui sont :

* missingDescriptor
* moreRowsDetected
* noRowsModified
* outputParametersDetected
* postAcquireClientSession
* postAcquireConnection
* postAcquireExclusiveConnection
* postAcquireUnitOfWork
* postBeginTransaction
* preCalculateUnitOfWorkChangeSet
* postCalculateUnitOfWorkChangeSet
* postCommitTransaction
* postCommitUnitOfWork
* postConnect
* postExecuteQuery
* postReleaseClientSession
* postReleaseUnitOfWork
* postResumeUnitOfWork
* postRollbackTransaction
* postDistributedMergeUnitOfWorkChangeSet
* postMergeUnitOfWorkChangeSet
* preBeginTransaction
* preCommitTransaction
* preCommitUnitOfWork
* preExecuteQuery
* prepareUnitOfWork
* preReleaseClientSession
* preReleaseConnection
* preReleaseExclusiveConnection
* preReleaseUnitOfWork
* preRollbackTransaction
* preDistributedMergeUnitOfWorkChangeSet
* preMergeUnitOfWorkChangeSet
* preLogin
* postLogin
* postLogout

Chacune de ces méthodes reçois un paramètre de type SessionEvent. L’introspection de ce paramètre nous a permis de relever les méthodes suivantes :

* getEventCode : Elle retourne un entier qui représente le code de l’évènement.
* getProperties :
* getQuery : Elle retourne la requête SQL créée par JPA pour interroger la base de données.
* getResult : Elle retourne le résultat d’une requête (la réponse de la base de données à une requête).
* getSession : Elle retourne la session sur laquelle les opérations sont effectuées.

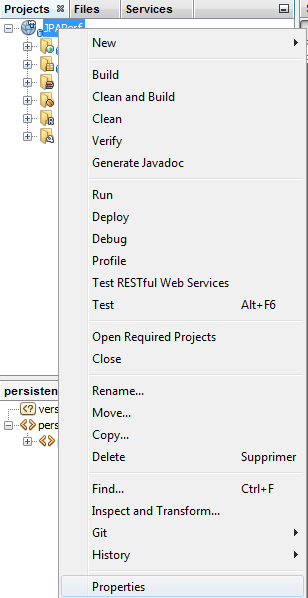
### Configuration d’une SessionEventListner

La configuration d’une *SessionEventListner* est assez simple et se fait à plusieurs niveaux.

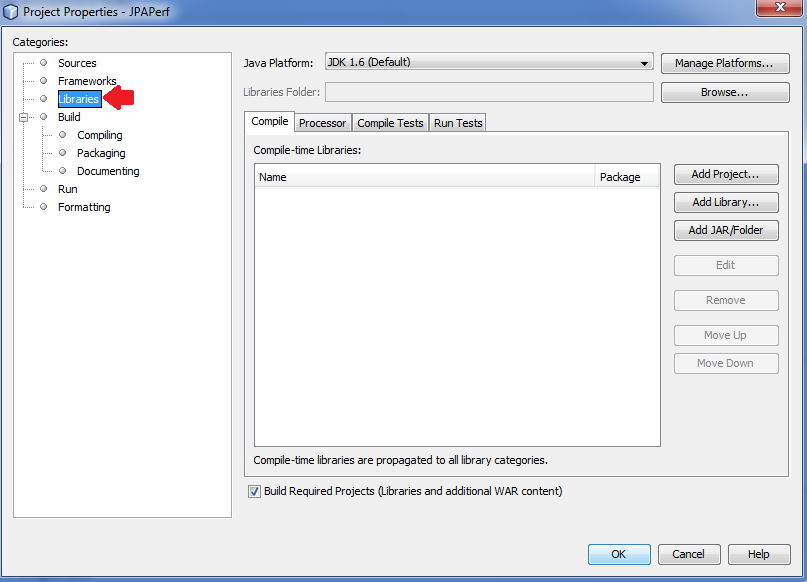
Il faut créer une classe java qui implémente l’interface *SessionEventListner*.

Pour éviter d’avoir une erreur lorsqu’on implémente l’interface *SessionEventListner*, il faut ajouter une dépendance à *EclipseLink* comme suite :

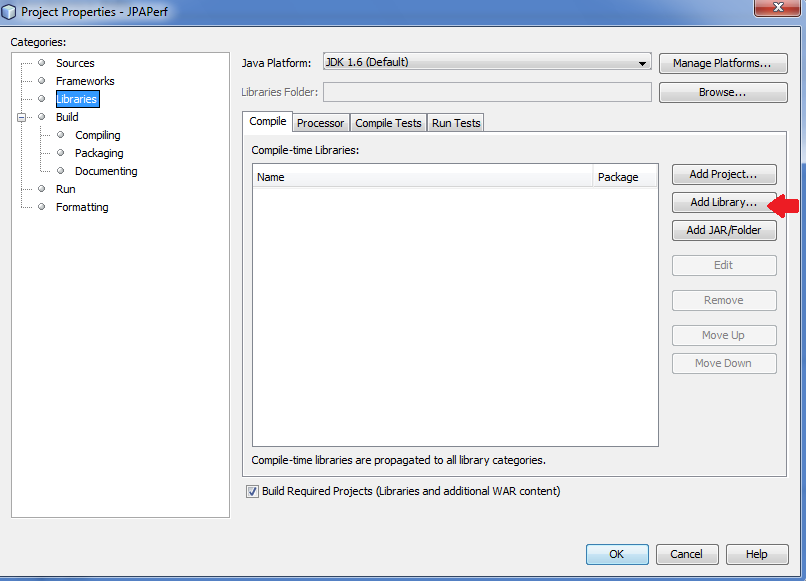
* Clic droit sur le projet et sélectionner  « Propriété »



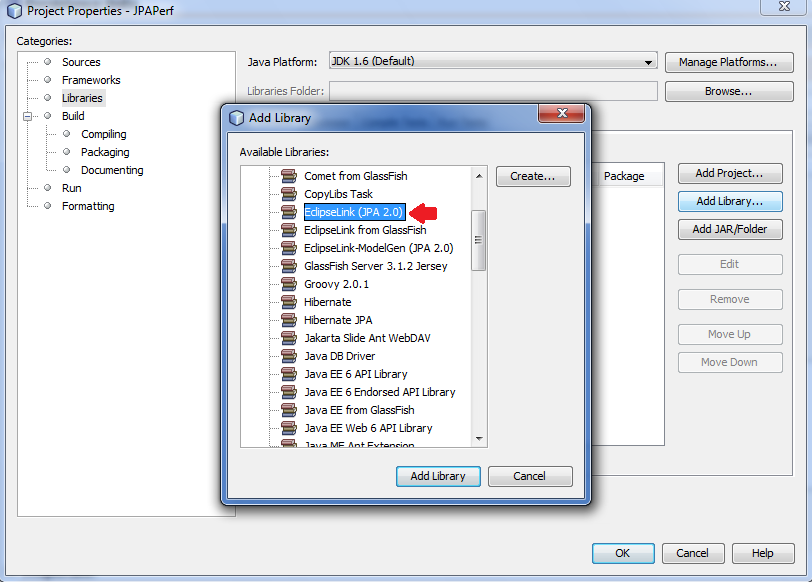
* Dans propriété, sélectionner « Librairies »



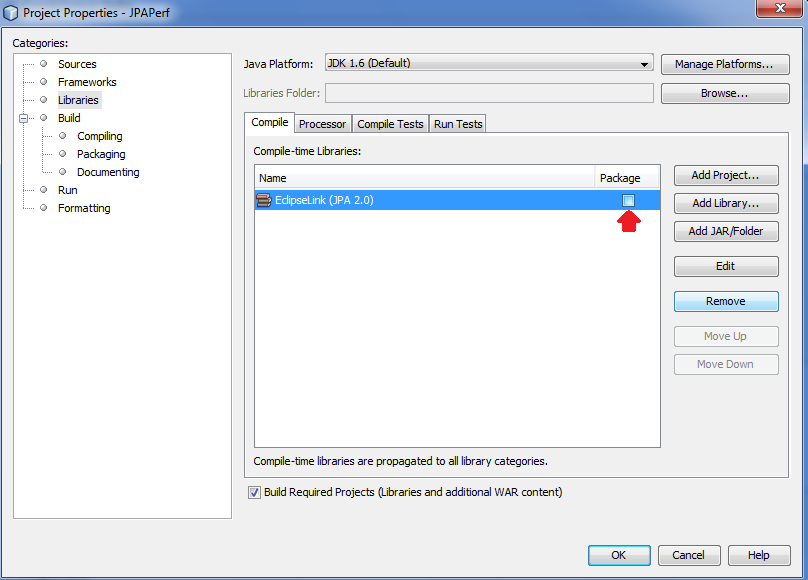
* Cliquer sur « Ajouter Librairie »



* Choisir EclipseLink (JPA 2.0) et cliquer sur « Ajouter Librairie » de la fenêtre active



* Décocher la case « Package » et cliquer sur « OK »



Dans le fichier *Persistence*.xml, il faut ajouter la propriété :

*<property name= ``eclipselink.session-event-listner´´ value=``mypackage.MyClass.class´´/>*

Myclass.class représente la classe qui implémente l’interface *SessionEventListner* et mypackage est le package où se trouve cette classe.

Après avoir effectué ces opérations, lorsqu’on fait des requêtes, la classe qui implémente l’interface *SessionEventListner* reçoit tous les évènements générés lors de la transaction entre JPA et la base de données. Ces évènements peuvent alors être enregistrés et utilisés comme on le veut.

## Présentation des métriques

La présentation des données nécessite deux opérations :

* Synthèse des données. Elle consiste à analyser les évènements reçus pour retrouver les informations pertinentes.
* L’affichage des données. Elle consiste à prendre l’information utile issue de la synthèse des données et la représenter sous un format simple et compréhensible.

### Synthèse des données

Comme nous l’avons dit plus haut, la classe implémentant l’interface *SessionEventListner* reçoit les notifications des différents évènements se produisant entre JPA et la base de données. Chaque notification déclenche l’exécution de la méthode de l’interface *SessionEventListner* à laquelle elle se rapporte. Chacune de ces méthodes reçoivent en paramètre un objet de type *SessionEvent*.

L’introspection du paramètre *SessionEvent* de chaque méthode nous a permis de faire sortir du lot de ces 36 méthodes quelles unes qui donnent des informations pertinentes.

* La méthode preExecuteQuery. L’évènement lié à cette méthode est déclenché avant l’exécution de chaque requête durant une session. La session ici est représentée par la connexion de JPA à la base de données et sa déconnexion. Le paramètre *SessionEvent* contient les informations sur la requête à exécuter :
* Le nom de la requête
* La classe concernée par la requête
* La requête JPQL, sous forme de chaine de caractères
* La requête SQL générée, sous forme de chaine de caractères.
* La méthode posExecuteQuery. L’évènement lié à cette méthode est déclenché après l’exécution de chaque requête durant une session. Le paramètre *SessionEvent* contient les mêmes informations que celles rencontrée dans la méthode preExecuteQuery sauf qu’ici on a en plus, le résultat de la requête.

Le temps étant un facteur déterministe de performance, il nous est possible ici prendre des mesures de temps d’exécution des requêtes. C’est-à-dire le temps qui s’écroule entre la réception d’une requête dans la base de données et la réponse de la base de données. Il est également possible de mesurer la durée d’une transaction. C’est-à-dire le temps qui s’écroule entre la réception par JPA, d’une requête venant de l’application et la réponse de JPA. Nous allons nous concentrer sur le temps d’exécution des requêtes.

L’information ayant été synthétisée, nous avons créé de nouvelles classes d’entité pour la représentation de celle-ci et nous avons utilisé JPA pour faire une sauvegarde des objets de ces classes dans une nouvelle base de données (la figure 1-1 illustre bien le travail effectué). Il ne restera qu’afficher le contenu de cette base de données.

### Affichage des données

L’affichage des données consiste à représenter sur une page web les informations issues de la synthèse de données relatives à l’exécution de chaque requête de façon à pouvoir suivre de jour en jour l’état des performances de l’application tout en gardant l’historique des requêtes.

Dans le but de faire une séparation entre les métriques issues de l’application et leur représentation, nous avons créé un nouveau projet qui s’appuie sur cette nouvelle base de données pour représenter son contenu.

Nous avons utilisé le paradigme *MVC2* [[3]](#footnote-3)(Modèle-Vue-Contrôleur). Où, le modèle est représenté non pas par la base de données dont le contenu doit être affiché, ou par la classe d’entité qui y dérive via JPA mais plutôt par une classe (*JavaBeans*) construite sous la base de la classe d’entité qui dérive de la base de données. Les vues sont représentées par des *JSP*[[4]](#footnote-4) et les contrôleurs par les *servlet*[[5]](#footnote-5). On a deux contrôleurs pour les deux vues.

Lorsque l’application est en marche, il y a une vue qui affiche les différentes requêtes et les statistiques associé à chacune (nom de la requête, classe d’entité concernée par la requête, la requête JPQL, la requête SQL générée par JPA, le nombre de fois que la requête a été exécutée, le temps d’exécution moyen des requêtes, le plus petit temps d’exécution et le plus grand temps d’exécution des requêtes du même type). On la possibilité de sélectionner le nom d’une requête pour avoir plus d’information sur les requêtes de ce type. En cliquant sur le nom de la requête (qui est un lien), on est dirigé vers une autre vue qui, pour chaque requête du type sélectionné, affiche le nom de la requête, l’heure à laquelle elle a été exécutée et sa durée d’exécution. Les pièces jointes à ce document, faisant fois de résultat de test, illustrent ces vues.

Les tests ont été faits en modifiant les annotations possibles pour représenter l’héritage tout en effectuant les mêmes requêtes à chaque changement pour comparer les différentes exécutions à savoir, les requêtes générées et leurs durées d’exécution.

Cette représentation nous permet de voir les variations de la performance. On devrait noter que les statisticiens utilisent très peu la moyenne comme indicateur de tendance. Car elle est très sensible aux valeurs atypiques[[6]](#footnote-6). Mais dans notre cas, elle est parfaitement adaptée car ce sont ces valeurs atypiques qui doivent être rapidement observées et elles représentent les piques de performance qui peuvent être des pertes ou des gains.

La création du nouveau projet pour récolter les données et faire des tests de performance dénote la faible dépendance entre l’application à tester et l’outil de test. Un utilisateur voulant tester son application aura juste à configurer la *SessionEventListner* comme expliqué au point 4.1.2 et connecter notre base de données « Base de données des métriques » comme illustré sur la figure 1-1.

# Conclusion

JPA est un ORM très intéressant qui bénéficie de la portabilité et la popularité de Java. Notre étude nous a permis de voir que cette ORM propose beaucoup de possibilités qui simplifient le travail des développeurs. Son utilisation nécessite une bonne connaissance de la programmation en Java, une bonne documentation sur le sujet JPA et fait intervenir plusieurs technologies de Java EE. Le couplage entre ces technologies est assez simple à mettre en œuvre.

La réalisation d’une plate-forme utilisant JPA tout en tenant compte des problèmes de performance n’est pas évident d’autant plus qu’une plate-forme se doit d’être évolutive et par conséquent, des modifications y seront apportées en permanence et plus, il est presqu’impossible de savoir à l’avance exactement ce qu’il faut faire pour ne pas dégrader les performances. Un tel outil permettra aux développeurs d’évoluer dans leur travail et revenir plus tard sur les points critiques qui auront été détectés.

Notre application de référence nous a permis de mieux nous familiariser avec JPA. Bien que simpliste, elle nous a permis mettre en pratique une partie non négligeable des spécificités de cette ORM et de faire des prises de mesure.

Les prises de mesure nous montrent des différences entre les temps d’exécution des requêtes en fonctions des annotations utilisées. Ce qui prouve que les annotations ont belles et bien un impact sur les performances. Mais, dans notre cas, on ne peut pas tirer de grande conclusion sous la base de notre application au vu de sa simplicité. Néanmoins, nous avons sous la main un outil de benchmarking dont la faible dépendance avec l’application à tester facilitera son intégration dans une chaine de test.

Toutefois, des améliorations sont à apporter à cet outil. Au niveau des métriques, on pourra ajouter au temps d’exécution des requêtes par la base de données, la durée des transactions. On pourra également ajouter en plus de la *SessionEventListner*, la *Performence Profiler* qui donne un grand nombre d’informations, on peut citer, le temps de préparation et d’exécution des requêtes, le nombre de table concernée par une requête. Pour une meilleure vue, on pourra ajouter un graphe des variations des performances. Vu l’importance de ces modifications, nous nous proposons de continuer le développement de cet outil.

# Bibliographie

[1] http://fr.wikipedia.org/wiki/JavaBeans

[2] <http://www.siteduzero.com/informatique/tutoriels/creez-votre-application-web-avec-java-ee/mise-en-place-29>

[3] <http://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur>

[4] <http://fr.wikipedia.org/wiki/JavaServer_Pages>

[5] <https://fr.wikipedia.org/wiki/Servlet>

<http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>

<http://wiki.eclipse.org/Introduction_to_EclipseLink_Sessions_%28ELUG%29#java.util_Logging>

http://eclipse.org/eclipselink/documentation/2.4/jpa/extensions/p\_session\_event\_listener.htm

<http://blog.ippon.fr/2011/10/11/jpa-une-magie-qui-se-merite-retour-aux-sources-de-jpa/>

Pro JPA 2 : Mastering the Java *Persistence* API de Mike Keith; Merrick Schincariol. Publier par Apress le 03 Décembre 2009

# Table des figures

[Figure 1‑1 3](#_Toc363445693)

[Figure 2‑1 5](#_Toc363445694)

[Figure 2‑2 7](#_Toc363445695)

[Figure 2‑3 9](#_Toc363445696)

[Figure 2‑4 10](#_Toc363445697)

[Figure 2‑5 12](#_Toc363445698)

[Figure 2‑6 13](#_Toc363445699)

[Figure 2‑7 13](#_Toc363445700)

[Figure 2‑8 14](#_Toc363445701)

[Figure 2‑9 15](#_Toc363445702)

[Figure 2‑10 16](#_Toc363445703)

[Figure 2‑11 17](#_Toc363445704)

[Figure 2‑12 18](#_Toc363445705)

[Figure 2‑13 18](#_Toc363445706)

[Figure 2‑14 19](#_Toc363445707)

[Figure 3‑1 19](#_Toc363445708)

[Figure 3‑2 20](#_Toc363445709)

[Figure 3‑3 22](#_Toc363445710)

[Figure 3‑4 23](#_Toc363445711)

[Figure 3‑5 24](#_Toc363445712)

[Figure 3‑6 25](#_Toc363445713)

[Figure 3‑7 26](#_Toc363445714)

1. cURL (Client URL Request Library) est une interface en ligne de commande permettant d’effectuer des requêtes http, https, ftp et bien d’autres. [↑](#footnote-ref-1)
2. La sérialisation est un processus de transformation de l’état d’un objet ou d’une structure de données en un format qui supporte la sauvegarde, la transmission des données sur un lien internet de façon à pouvoir reconstituer l’élément sans perte d’informations. [↑](#footnote-ref-2)
3. MVC : Modèle-Vue-Contrôleur. C’est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective [3] [↑](#footnote-ref-3)
4. JPS : Java Server Pages est une technique basée sur Java qui permet aux développeurs de créer dynamiquement des pages web [4] [↑](#footnote-ref-4)
5. Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d’un serveur http [5] [↑](#footnote-ref-5)
6. En statistique une valeur atypique est une valeur inhabituelle qui se démarque des autres valeurs observées [↑](#footnote-ref-6)