Université de Technologie de Compiègne

Génie Informatique

Rapport de projet - Projet Java EE

SR03

Steve LAGACHE & Romain PELLERIN

Chargé de TD : Enrico Natalizio

Printemps 2016 (P16)

Dernière mise à jour : 24 mai 2016

Table des matières

1 Projet : plateforme de QCM

Le but initial du projet était de créer une plateforme en ligne proposant de remplir des QCM à des stagiaires ; la plateforme étant administrable par des utilisateurs particuliers (administrateurs). Un certain nombre de fonctionnalités était attendu notamment la gestion du parcours des stagiaires, l'affichage des scores, la gestion fine des questions et réponses, etc.

De plus, nous avons pris la liberté de rajouter des fonctionnalités :

— Gestion des exceptions Java avec feedback utilisateur

— Gestion/vérification des *input* utilisateurs

— Construction et définition rigoureuse de la base de données MySQL pour avoir une cohérente maximale et une redondance minimale

Par ailleurs, il était intéressant d'essayer de faire du code Java le plus réutilisable possible en utilisant notamment de l'héritage, et d'autres *Design Patterns*.

1.1 Choix technologiques : language, base de données, *libraries*, IDE

Comme imposé le projet a donc été fait en Java Enterprise Edition. Nous avons néanmoins souhaité utiliser **la version 8** pour pouvoir profiter des nouvelles fonctionnalités offertes sur les listes (faire un `map` par exemple ou un `filter`).

À cela, s'ajoute l'utilisation de Tomcat 8 dans sa version stable. Enfin, quant à la base de données, il s'agit d'une base MySQL proposée par l'UTC sur le serveur `tuxa.sme.utc` (accessible uniquement sur le réseau de l'UTC) : il nous aura fallu utiliser le connecteur `mysql-connector-java-5.1.39-bin.jar`.

Le rendu des pages Web est fait avec JSP (JSP Standard Tag Library). Il nous aura fallu utiliser la bibliothèque `jstl-1.2.jar`.

Au niveau du développement, nous avons utlisé Eclipse Mars comme IDE [1] pour sa portabilité et sa robustesse (bien que l'IDE tende de plus à plus à devenir une usine à gaz, et dont les performances sont discutables, en comparaison à d'autres IDE tels qu'IntelliJ).

1.2 Rappels sur JEE

Un bref rappel, néanmoins important, quant au fonctionnement d'une application JEE.

Les routes sont définies dans un fichier `web.xml`. À chaque route est associée une ***servlet***. Une ***servlet*** est une classe Java particulière qui va effectuer plusieurs traitements sur deux objets particuliers (requête et réponse HTTP) avant de renvoyer une réponse au client, le plus souvent sous la forme de code HTML. Ce code HTML peut être généré à la volée ou écrit au préalable dans un fichier JSP qui sera importé. Ce fichier pré-écrit contient du code qui sera exécuté au moment de l'import, il peut s'agir de code Java ou de balises particulières JSP, mais dans les deux cas le résultat est le même.

1. *Integrated Development Environment*

L'application contient également des fichiers publics, accessibles sans ***servlet***, stockés dans un sous-dossier du dossier `WEB-INF`. Il s'agit souvent de fichier statiques tels que du CSS, JavaScript, ou des images.

Les ***servlets*** peuvent faire appel à n'importe quel autre fichier Java, instancier n'importe quels objets. C'est également ici que les appels à la base de données se font (par exemple après réception d'une requête HTTP de type POST).

1.3 Architecture du projet

1.3.1 Java sources

Nous avons décidé de créer plusieurs packages, et cela dans plusieurs buts :

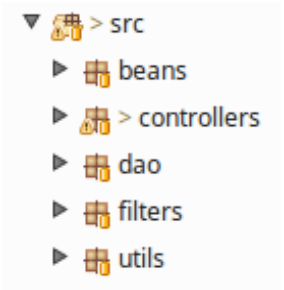

Figure 1.1 – Nos sources Java

- **beans** : Contient les classes métiers, c'est-à-dire les modèles. Il s'agit de classes simples qui sont des représentants de nos classes UML créées lors de la modélisation du projet. À chaque classe correspond une table en base de données. Ces classes contiennent les mêmes attributs que les colonnes des tables correspondantes en base de données. Les seules méthodes qui existent sont des *getters/setters* et des constructeurs.
- **controllers** : Ce sont les *servlets* qui traitent les requêtes HTTP.
- **dao** : Ce sont les classes qui gèrent l'accès à la base de données. Nous avons créé une classe abstraite `DAO.java` dont toutes les classes héritent afin d'éviter autant de possible de répéter du code, notamment pour les fonctions génériques telles que `find()`, `insert()`, etc.
- **filters** : Contient un filtre permettant d'assurer que certaines pages ne soient atteintes que par des utilisateurs connectés.
- **utils** : Contient des classes Java simples avec quelques fonctions `static` afin, à nouveau, d'éviter de réécrire deux fois le même code. Ce sont des fonctions génériques utilitaires.

1.3.2 WEB-INF

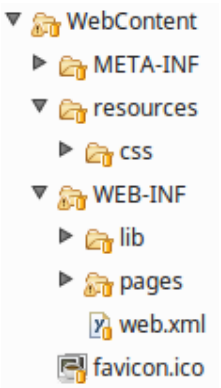

Figure 1.2 – Le dossier WebContent

L'autre dossier intéressant du projet est sans aucun doute `WebContent`. Il contient beaucoup d'éléments importants répartis dans des sous-dossiers :

- Les fichiers statiques servis par le serveur sur simple requête (dans notre cas, uniquement du CSS) dans le dossier `resources`.
- Les 2 bibliothèques que nous utilisons mentionnées plus haut (MySQL connecteur et JSTL) dans `WEB-INF/lib`.
- Les pages JSP dans `WEB-INF/pages`.
- Le fichier `web.xml` qui définit le lien entre les routes et les servlets, dans le dossier `WEB-INF`.

1.4 Base de données

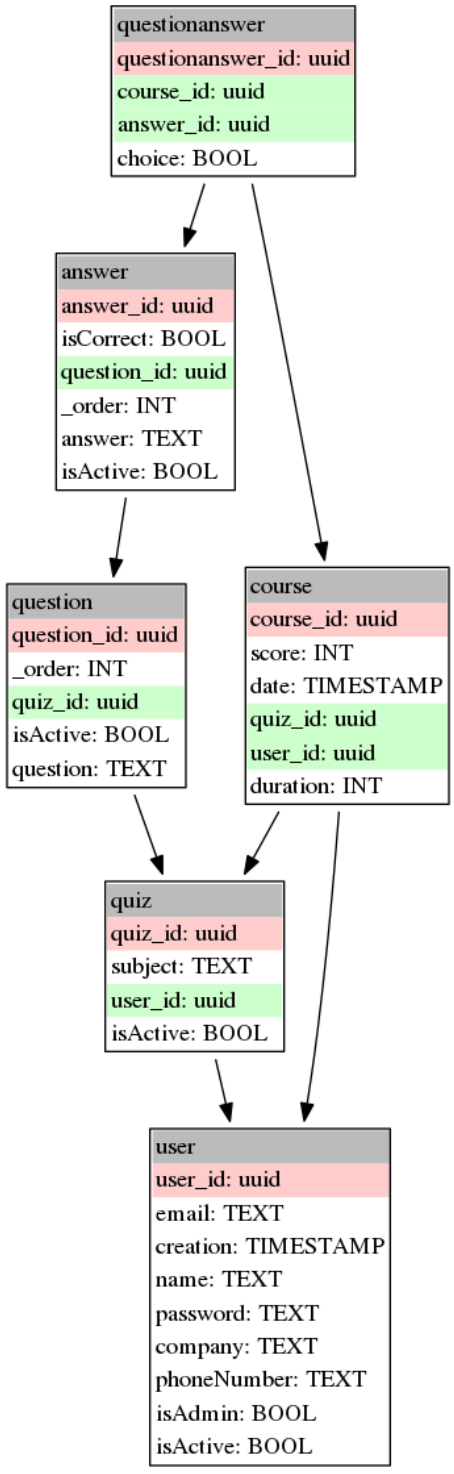

Figure 1.3 – Schéma de notre base de données

Ce projet repose également sur l'utilisation d'une base MySQL mise à disposition par l'UTC. Cette base sert à stocker toutes nos données.

L'une des premières étapes de ce projet, avant d'attaque tout code, a été de réfléchir à un modèle UML pour notre base de données. C'est ce même modèle qui servirait plus tard à créer nos classes métiers (*beans*). Nous avons voulu éviter toute redondance (3ème forme normale). Le

schéma auquel nous avons abouti se trouve sur l'image ci-dessus.

1.4.1 Explications

Ce schéma se veut le plus clair possible afin de représenter les liens qui existent entre les différentes tables. Cependant, il convient de l'expliquer un minimum et de préciser les différences avec l'implémentation réelle.

— La plupart des champs sont en `NOT NULL` par défaut sauf ceux sans importance comme l'entreprise ou le numéro de téléphone.

— Les clés primaires sont en `AUTO_INCREMENT`.

— Les dates sont créées automatiquement grâce à `CURRENT_TIMESTAMP`.

— La clé primaire de la table des utilisateurs est en fait son email (`user_id` a été renommée en `email`).

— La clé étrangère `user_id` de la table des quizzes pointe vers le créateur du quiz.

— Nous avons rajouté de nombreux `CHECK` et `UNIQUE KEY` afin de renforcer la cohérence des données. Pour cela, voir notre fichier joint à ce rapport.

1.4.2 Améliorations possibles

Le serveur ne nous permettait pas de créer de `triggers`. C'est dommage car cela aurait pu nous permettre d'assurer une plus grande cohérence des données (par exemple vérifier qu'un créateur de quiz est administrateur, ou qu'un parcours – *course* – est fait par un stagiaire, donc quelqu'une qui n'est pas administrateur). De même pour les colonnes `_order` : avec des triggers nous aurions pu les incrémenter correctement sans devoir le faire dans le code Java.

1.5 Fonctionnement de l'application entière

Comme expliqué plus haut, le fichier `web.xml` sert à *forward* la requête vers la bonne servlet en fonction du *pattern* des routes. Des *wildcards* peuvent être utilisés afin de *match* plusieurs routes.

La requête est ensuite envoyée soit à un filtre, soit directement à une servlet, cela dépend de ce qui est écrit dans le fichier `web.xml`. Dans notre cas, toutes les requêtes sont transmises à un filtre, sauf une : celle à destination de la page de *login*. Dans les autres cas, le filtre nous permet de nous assurer que c'est un **utilisateur connecté** qui tente d'accéder aux pages.
Par ailleurs, nous utilisons également un autre filtre, cette fois-ci pour **toutes les requêtes**, qui nous permet de forcer l'encodage en UTF-8. Cela est notamment très utile à la réception des requêtes de type POST. Avant cela, lorsque nous insérions des *input* utilisateurs en base, le contenu reçu était implicitement converti en *latin1*. Maintenant, grâce à ce filtre, il est conservé en UTF-8 et est correctement sauvegardé en base de données.

XML

```xml
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-class>org.apache.catalina.filters.
       SetCharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
```

```
    <init-param>
        <param-name>ignore</param-name>
        <param-value>false</param-value>
    </init-param>
</filter>
```

À la sortie d'un filtre, la requête est transmise à une servlet. À l'éxécution de la servlet, nous avons à disposition deux objets Java (`HttpServletRequest request`, `HttpServletResponse response`). Ils nous permettent respectivement de traiter la requête HTTP et la réponse HTTP que l'on enverra (accès aux *headers*, au *body*, paramètres GET, URL, etc.).

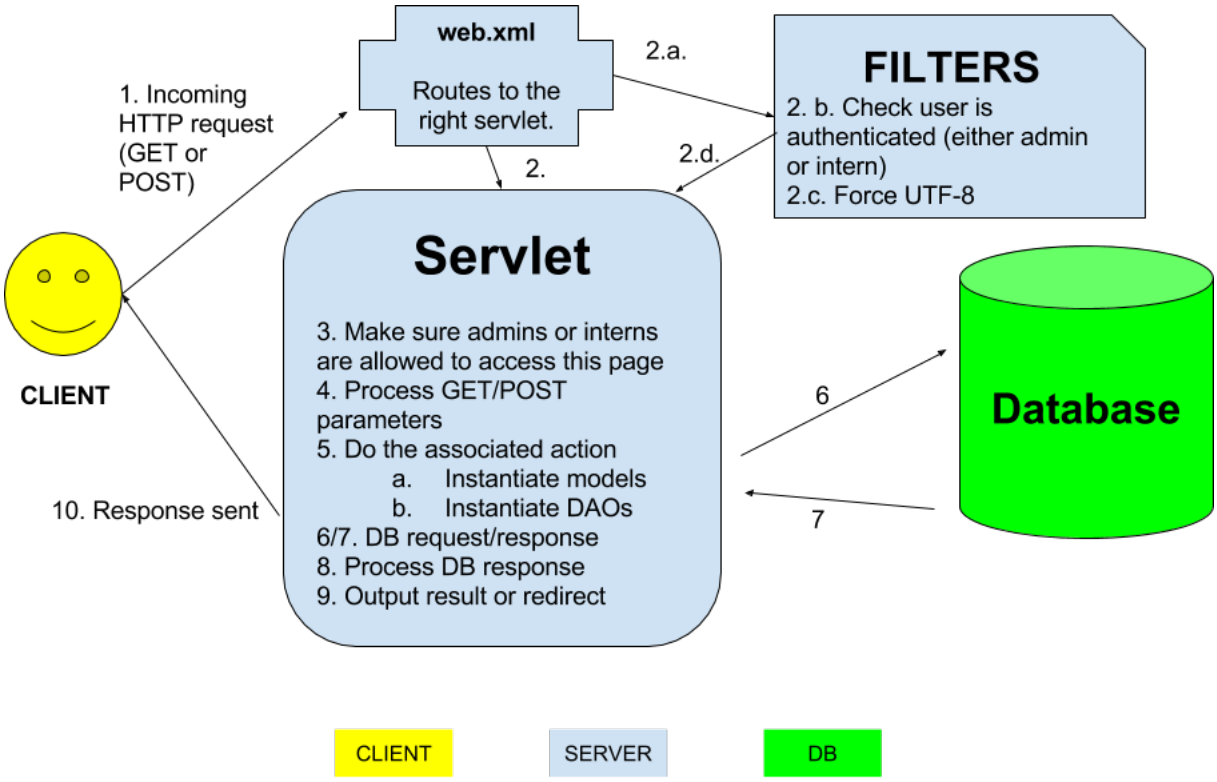

Figure 1.4 – Fonctionnement de notre application

En fonction des paramètres GET (dans l'URL) ou POST (dans le *body*) reçus, nous effectuons diverses opérations, souvent sur la base de données. Puis nous "injectons" des variables dans l'objet requête que nous "passons" finalement à une page JSP. Cette page va utiliser les variables injectées, qui peuvent être de type primitif, des objets, des tableaux, etc, afin d'afficher leur contenu au client.

Java

```
public static final String REQUEST_QUESTION = "question";
...

// Injection de variables
request.setAttribute(REQUEST_QUESTION, qd.find(q));

// Appel d'une page JSP
this.getServletContext().getRequestDispatcher("/WEB-INF/pages/
   answersmanagement.jsp").forward(request, response);
```

1.6 Implémentation

1.6.1 Servlets

Nous avons essayé autant que possible de réutiliser les servlets plusieurs fois, afin de ne pas dupliquer du code et aussi pour simplifier notre application. Pour cela, nous avons avantageusement tiré profit de deux méthodes offertes par les servlets :

Java

```java
@Override
public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
}
@Override
public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
}
```

Ces deux fonctions seront appelées en fonction du type de requête HTTP reçue : GET ou POST. Ainsi, sur plusieurs pages, notamment le login, nous pouvons utiliser la même servlet pour l'affichage du formulaire mais aussi pour la réception des identifiants de *login*. Par la suite, si les identifiants sont corrects, nous redirigeons l'utilisateur, comme cela se fait sur de nombreux sites web :

Java

```java
@Override
public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    // Vérifications des identifiants reçus auprès de la BDD

    // Sauvegarde du cookie de session
    session.setAttribute(SESSION_USER, user);
    // Redirection
    response.sendRedirect("dashboard");

    // Code ...
}
```

1.6.2 Beans et DAO

Nous avons autant de classes *bean* que de DAO. La couche DAO fait office d'abstraction par rapport au modèle, afin d'effectuer des opérations en base de données.

1.6.3 Filtres

Comme dit précédemment, nous avons utilisé deux filtres dont un créé par nous. Celui-ci permet de vérifier en amont qu'un utilisateur qui tente d'accéder à une page est bien connecté (vérification des cookies). L'autre filtre permet de forcer UTF-8 sur les données reçues par l'utilisateur.

1.7 Résultat final

Ci-dessous deux extraits de deux de nos pages.

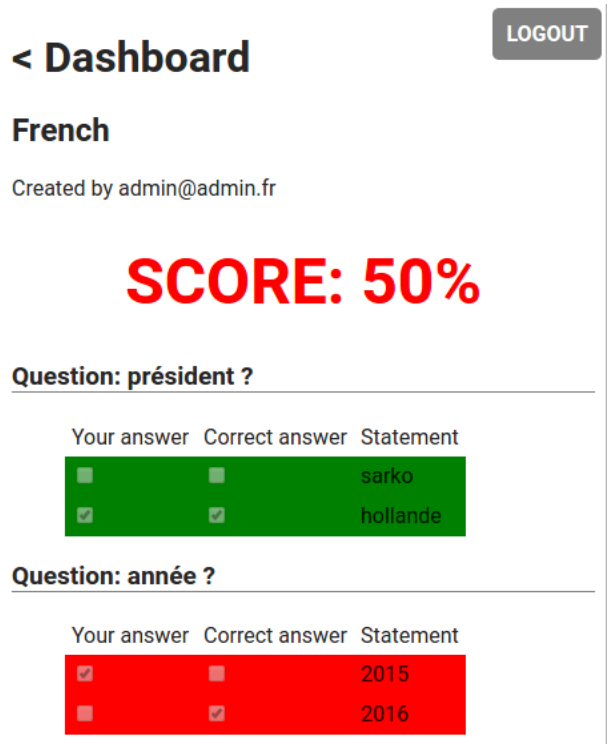

Figure 1.5 – Fin d'un QCM

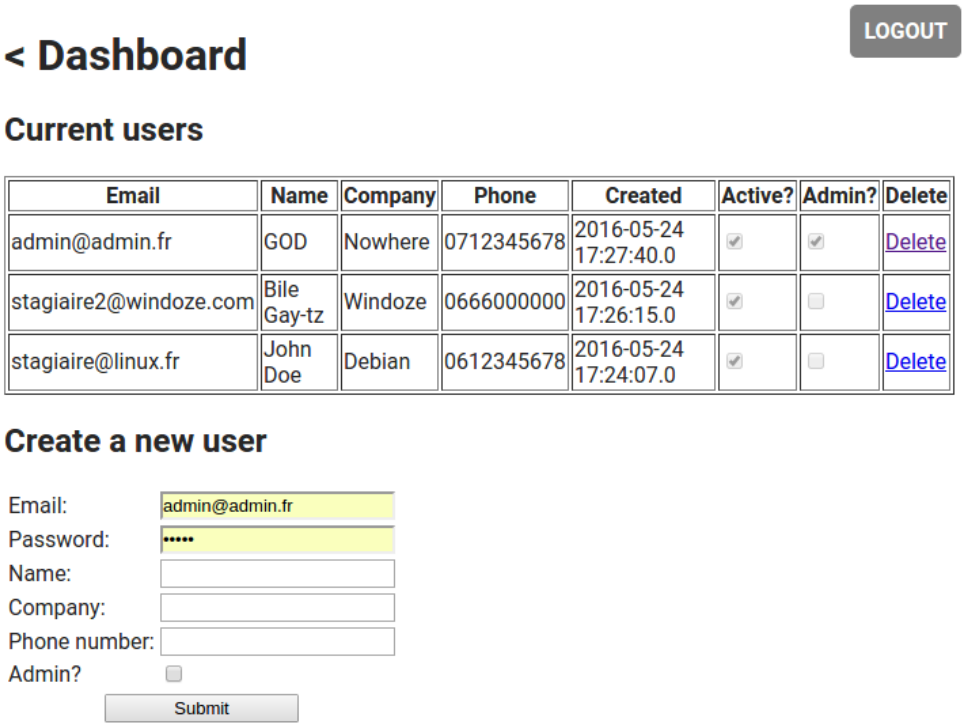

Figure 1.6 – Liste des utilisateurs

1.8 Difficultés rencontrées

Nous n'avons pas rencontré de difficulté majeure. En revanche le projet était très chronophage et répétitif ce qui vient ternir le ressenti global.

1.9 Comment exécuter le projet

1. Java
```
sudo apt-get install openjdk-8-jdk
```
2. Télécharger Eclipse for Java EE et l'installer.
3. Télécharger Tomcat 8.0.35 et extraire l'archive.
4. Importer le projet dans Eclipse en faisant "*File*" > "*Import*" > "*General*" > "*Existing Projects into Workspace*".
5. Corriger d'éventuels problèmes de *Build path*.
6. Installer Tomcat 8 dans Eclipse en précisant l'emplacement de Tomcat 8 qui vient d'être téléchargé et extrait.
7. Associer le projet au serveur et lancer le projet.
8. Puis ouvrir un navigateur Web et aller à `http://localhost:8080/Projet2/`.

1.10 Conclusion

En somme, un projet très instructif et complet qui permet d'appréhender dans son ensemble la création d'une application Java EE, liée à une base de données.