Rapport de projet

Développement Web II

SOFONEA Axel
JEAMME Christopher
GRANJON Thomas
SIRACUSA Rémi
BRUYERE Dimitri

Introduction

Notre projet est une application de gestion de dépense de groupe. Organiser un budget lors d'un événement n'est pas toujours simple. Le but de notre application est de rendre cette tâche facile et intuitive via deux interfaces possibles : une page web et une application java. Ce rapport est là pour vous présenter ce projet et la manière dont on a procédé.

Rapport de conception

Diagramme UML

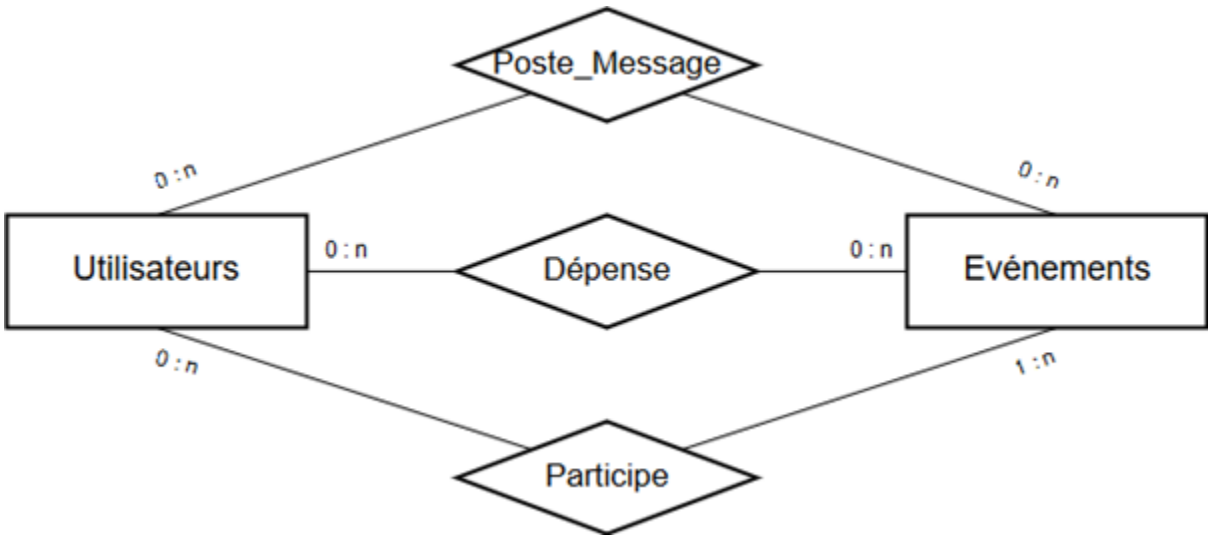
Voir le fichier DealWithItUML.png

Plan du site / application

Notre application permet de s'inscrire via un formulaire d'inscription ou de s'authentifier si nous sommes déjà inscrits. Un système de gestion des utilisateurs est donc présent.

Une fois connectés sur le site, nous avons la possibilité de consulter les dernières opérations effectuées sur le site, de créer un événement et enfin, de consulter les événements auxquels on participe. Si on accède à "Mes événements", il nous apparaît alors tous les événements auxquels on participe, et le fait de cliquer sur l'un de ces événement nous emmène sur l'événement en question. Depuis cette interface, on peut consulter le total des dépenses effectuées par l'ensemble des participants sur l'événement, mais également l'ensemble des participants et l'historique des dépenses de l'événements (triées par date). Nous pouvons ajouter une dépense à l'événement en ajouter une description et un montant, et aussi ajouter un participant grâce à son pseudo sur le site.

Base de données



Utilisateurs (**idUtilisateur**, nom, prénom, email, pseudo, motDePasse)
Evénement (**idEvenement**, nomEvenement, budget)
Participe (**#idUtilisateur, #idEvenement**)
Poste_Message (**#idUtilisateur, #idEvenement, date**, message)
Dépense (**#idUtilisateur, #idEvenement, date**, montant, description)

La table Utilisateurs contient les informations relatives aux personnes inscrites sur le site. La table Événement contient les événements. La table Participe permet de faire le lien entre un utilisateur et les événements auxquels il participe, et la table Dépense permet de stocker les dépenses effectuées par les utilisateurs dans les événements.

Liste des fonctionnalités

Web

- Création de compte (Inscription)
- Connexion
- Déconnexion
- Créer un événement
- Ajouter des utilisateurs à notre événement créé
- Participer aux événements d'autres utilisateurs
- Ajouter une dépense avec une description et un montant à un événement auquel on participe
- Voir la liste de nos événements dans la page d'accueil connecté
- Visualiser un événement avec ses dépenses et ses collaborateurs
- Voir la liste de toutes nos dépenses dans la page d'accueil connecté

Swing

- Connexion

Technologies utilisées

- **Thread**
- **JSP**
- **Servlet**
- **JSTL**
- **Expression Language (EL)**
- **Swing**
- **XML / DTD**
- **Sockets TCP**
- **Parser SAX**
- **JavaScript**

Client léger

En ce qui concerne la partie client léger, c'est-à-dire la partie de notre application accessible via le web, notre projet respecte le patron de conception **MVC** (Modèle-Vue-Contrôleur).

Le modèle correspond à la base de données **dealWithIt.sql**, qui contient les tables nécessaires pour le stockage des utilisateurs, des événements et des dépense (voir partie Base de Données).

Les vues, qui correspondent aux fichiers **jsp** contenus dans le dossier WebContent, réalisent l'affichage des données sur le site web. Elles contiennent également les formulaires nécessaires pour la modification du modèle.

Les contrôleurs sont les **servlets** situées dans le package `interfaceWeb`. Les contrôleurs reçoivent et valident les informations rentrées par les utilisateurs. Ils effectuent des requêtes au modèle pour récupérer des informations et les transmettre par la suite aux vues, et modifient le modèle dans le cas d'une réception par formulaire (dans la méthode `doPost`).

Dans les jsp, on utilise les technologies **EL** et **jstl** :

- Pour afficher les message d'erreur : par exemple, dans `evenement.jsp`, si le montant entré dans le formulaire n'est pas valide, le contrôleur nous envoie un message d'erreur qui s'affiche dans un cadre rouge. On pourrait simplement afficher grâce à l'expression language `${erreur}`. Mais le cadre rouge serait présent même sans message à afficher. On utilise donc la balise `<c:if>` de jstl pour effectuer le test :

```
<c:if test="${not empty erreur}">
      <span class="alert alert-danger">${erreur}</span>
</c:if>
```

- Pour effectuer des boucles : par exemple, pour afficher les dépenses d'un utilisateur, on récupère du contrôleur une `LinkedHashMap<Depense,Evenement>`, qui contient chaque dépense d'un utilisateur associée à l'événement dans lequel la dépense a été effectuée. On utilise la balise `<c:forEach>` de jstl pour afficher les dépenses :

```
<c:forEach var="dep" items="${  depenses  }">
      <span class="..."></span> ${ dep.key.date }
      <span class="..."></span> ${ dep.value.nomEvenement }
      <span class="..."></span> ${ dep.key.description }
      <span class="..."></span> ${ dep.key.montant }€
</c:forEach>
```

On peut également voir dans l'exemple ci-dessus que les classes `Evenement` et `Depense` sont des JavaBeans. Il en est de même pour toutes les classes du package `donnees`. Ainsi, on peut facilement accéder aux variables des objets avec des EL.

Client lourd

Le client lourd est donc une interface Swing qui permet d'effectuer les mêmes fonctionnalités que le client léger.

De part divers problèmes nous n'avons pas pu la finir, néanmoins, toutes les structures sont présentes et il y a des fonctionnalités présentes. Il ne manque que l'implémentation pour tout faire fonctionner ensemble.

Ce client va envoyer des demandes à un serveur dédié aux clients lourds (**Multi-thread**, lancé par le serveur du web grâce à un **Listener**) basé sur des **sockets TCP**.

Le serveur va déterminer le type, lire la requête, la traiter et envoyer un retour. Les demandes peuvent être, soit des demandes d'objets (*Evenement*, *Depense, ...*), de connexion, d'inscription ou encore des demandes d'ajout d'objets dans la base de données **MySQL.**

Ces demandes sont faites à l'aide de fichiers **XML** et lues par un parseur **SAX.**

L'interface Swing est simple et a pour but de ressembler à celle du web afin de montrer la similitude entre les deux versions. A l'exception de la gestion des données exposée ci-dessus, toutes les fonctionnalités du site web auraient été les mêmes dans l'application.

L'interface possède deux pages : un accueil non connecté et un accueil connecté. Le premier présente l'application et permet de se connecter ou de s'inscrire. Le second offre (en plus de l'option de se déconnecter) trois boutons qui passent les informations sur les événements liés à l'utilisateur connecté, ses opérations et la procédure pour déclarer un nouvel événement, le tout dans la même page.

L'inscription ou la connexion se font via une fenêtre popup accessible depuis `AccueilNonConnecte.java` qui demande à l'utilisateur ses informations, les stockent dans la base de données (réciproquement les comparent à celles déjà entrées dans la base de données), vérifie le bon comportement des interactions avec le serveur puis lance `AccueilConnecte.java`.

Une fois la connexion, ou l'inscription, aboutie et validée, les informations liées à l'utilisateur (id, nom, prenom, pseudo, mot de passe) sont stockés dans une variable de type Utilisateur pour imiter une session.

Provenance du code

Nous avons écrit l'intégralité du code nous-même excepté une partie du CSS pour laquelle nous avons utilisé Bootsrap.
La gestion de la base de données, la connexion TCP et les parseurs ont été basés sur nos travaux de TP ou sur des exemples vus en cours.

Organisation

Nous avons utilisé `git` pour gérer les versions de notre code :
https://github.com/ChrisJeamme/X18A/
Pour ce qui est de l'organisation

- **Christopher JEAMME** était le chef de projet. Il s'est occupé de la conception / l'assignement des tâches, ainsi que de la gestion de la base de données côté Java, du serveur du client lourd, Swing, de la transmission de donnée entre le client lourd et le serveur, du CSS du site.
  Temps de travail estimé : environ 100 heures.

- **Dimitri BRUYÈRE** a majoritairement oeuvré sur le client léger, c'est à dire les jsp et les contrôleurs qui constituent la partie web de l'application. Il a également développé des fonctions d'interaction à la base de données, ainsi que quelques méthodes pour le XML.
  Temps de travail estimé : environ 50 heures.

- **Axel SOFONÉA** a aidé à l'élaboration de l'interface Swing.
  Temps de travail estimé : environ 10 heures.

- **Rémi SIRACUSA** a participé à l'élaboration de l'interface Swing. Il a travaillé sur le client lourd, le XML, les handlers SAX.
  Temps de travail estimé : environ 15 heures.

- **Thomas GRANJON** a longtemps travaillé sur le chat, sans succès. Puis a œuvré sur le client lourd, le XML, les handlers SAX et la création de la base de données. A écrit la majorité des commentaires et construit la Javadoc puis a participé à l'écriture du rapport. Temps de travail estimé : entre 40 et 50 heures.

Ressources utilisées

- Cours de Abderrazak Daoudi
- Cours de Rémi Emonet
- Cours d'OpenClassroom sur Java EE
- Site de TutorialPoint
- Site de W3Schools
- Divers forums
- Maven
- Eclipse

En tant que librairies externes, nous avons seulement utilisé Bootstrap pour le CSS du site.

Améliorations possibles

Modification des informations du compte
Suppression des éléments
Serveur multi-utilisateurs multi thread
Implémenter toutes les fonctions du web dans le Swing
Ajout et gestion d'un chat par événement entre tous ses participants

Conclusion

Ce projet nous a permis de pratiquer quasiment toutes les technologies de développement web qui nous été enseignées ce semestre. Il nous a donc permis de progresser.

La partie client léger est fonctionnelle et ressemble à ce que nous avions imaginé au départ. Malheureusement, le manque de temps ou d'organisation, notamment due au manque d'implication de certains membres du groupe, nous a empêché de terminer ce projet. La partie client lourd n'est pas terminée, même si les classes et les parseurs nécessaires ont été implémentés. Le temps consacré au développement inachevé du client lourd nous a empêché de pouvoir développer d'autres fonctionnalités qui semblaient réalisables sur la partie web.