Sorbonne Université

Rapport de projet

Développement d'Applications Réticulaires
Projet CLARINET

Amel ARKOUB
Chao LIN
Ling-Chun SO

Responsable :
Romain Demangeon

Septembre - Novembre 2018

Introduction

Dans le cadre du projet de DAR, nous devions développé une application réticulaire incluant une composante sociale, dont la thématique reste libre. Nous avons donc décidé de développer Clarinet. Il s'agit d'une application web sociale basée sur la participation des internautes à des **événements musicaux** en France. Les utilisateurs peuvent indiquer s'ils souhaitent participer à un événement musical particulier et discuter avec les autres utilisateurs qui participent à cet évènement. Pour une lecture plus agréable, nous réduisons au terme **événements** les événements musicaux.

Un utilisateur peut rechercher des événements qui l'intéressent en tapant des mots-clés. S'il trouve un événement, il peut accéder à sa fiche détaillée, et même faire apparaître le lieu sur une carte de France. Si un utilisateur indique sa participation, il peut ensuite rejoindre un salon avec d'autres utilisateurs qui participent au même événement.

Chaque utilisateur possède un profil. Il peut charger une photo de profil et écrire une biographie. On y trouve également les listes des événements auxquels il a déjà participé, des événements auxquels il va participer, des abonnements aux autres utilisateurs, et de ses abonnés. Il peut également démarrer un chat privé un utilisateur parmi ses abonnements ou ses abonnés.

En somme, Clarinet est une application qui permet aux utilisateurs de découvrir les divers événements musicaux qui prennent place partout en France. Mais aussi, les utilisateurs peuvent faire de nouvelles rencontres, discuter, et par la suite se réunir pour profiter de l'événement ensemble.
Le code de l'application web Clarinet est disponible sur Github (https://github.com/Shaolans/clarinet) et est visible sur heroku (https://clarinet.herokuapp.com/).

Manuel utilisateur

Description de l'interface

Clarinet est une application web qui permet à ses utilisateurs de découvrir de divers évènements musicaux de France. Les utilisateurs peuvent rechercher des événements en donnant un mot clé de leur choix ou une date. Lorsqu'il participe à un évènement, le salon de conversation de cet évènement lui ait ouvert pour discuter avec les autres participants. De plus, il est possible pour l'utilisateur de rechercher d'autres utilisateurs par le nom et se s'abonner à eux. Et bien sûr, les utilisateurs sont libres de discuter avec ses abonnées et les personnes dont ils se sont abonnés. Les évènements participés, les évènements qu'on va participer, les abonnées et les abonnements sont tous affiché sur la page de profile de l'utilisateur.

Uses cases

Cas n°1: Rechercher un évènement par mot clé ou par date

Acteur: Utilisateur
Précondition: L'acteur doit être connecté et être sur la page de la carte.
Scénario:

1. L'acteur entre dans la barre de recherche d'évènements le mot clé de son choix.
2. Le système recherche les évènements correspondant aux critères de recherche.
3. Le système affiche les évènements trouvés par ordre chronologique décroissant.
4. L'acteur clique sur le nom d'un évènement
5. Le système affiche la page correspondant à l'évènement

Alternatives:

A1. Au S1, l'acteur choisit la recherche par date. Continue au S2.

A2. Au S1, l'acteur n'a donné aucun critère de recherche, le système affiche tous les évènements. Continue au S2.

Postcondition: L'acteur peut voir les évènements qu'il recherche.

Cas n°2: Situer sur une map le lieu d'un évènement

Acteur: Utilisateur

Précondition: L'acteur doit être connecté et a déjà procédé le cas n°1

Scénario:

1. L'acteur clique sur le bouton Carte situé à droite d'un évènement.
2. Le système récupère l'adresse correspondant à l'évènement.
3. Le système recherche le geocode de l'adresse.
4. Le système centre la carte sur le lieu de l'évènement.

Exception:

E1. Au S2, le système n'a pas pu récupérer le géocode de l'adresse.

Postcondition: L'acteur peut voir le lieu où se passe l'évènement sur une carte.

Cas n°3: Participer à un évènement

Acteur: Utilisateur

Précondition: L'acteur doit être connecté, être sur la page d'un évènement et de n'avoir pas encore participer à cet évènement.

Scénario:

1. L'acteur clique sur le bouton participer.
2. Le système ajoute l'acteur sur la liste des évènements.
3. Le système ajoute l'évènement dans la liste des évènements passées de l'acteur.

Alternative:

A1. Au S3, le système ajoute l'évènement dans la liste des évènements futurs de l'acteur s'il s'agit d'un évènement futur.

Postcondition: L'acteur fait partie des participants de l'évènement.

Cas n°4: Ne plus participer à un évènement

Acteur: Utilisateur

Précondition: L'acteur doit être connecté, être sur la page d'un évènement et d'avoir participer à cet évènement.

Scénario:

1. L'acteur clique sur le bouton ne plus participer.

2. Le système supprime l’acteur sur la liste des évènements.
3. Le système supprime l’évènement dans la liste des évènements passées de l’acteur.

Alternative:

A1. Au S3, le système supprime l’évènement de la liste des évènements futurs de l’acteur s’il s’agit d’un évènement futur.

Postcondition: L’acteur ne fait plus partie des participants de l’évènement.

Cas n°5: Rejoindre le salon de conversation d’un évènement

Acteur: Utilisateur

Précondition: L’acteur doit être connecté, être sur la page de profile et d’avoir participer à un évènement.

Scénario:

1. L’acteur clique sur le bouton Rejoindre le salon situé à droite de l’évènement.
2. Le système crée un chatbox et affiche l’historique des dialogues de ce salon de conversation dans le chatbox.
3. Le système annonce aux autres utilisateurs qui se sont rejoint au salon l’arrivée de l’acteur.
4. L’acteur envoie une message au salon.
5. Le système stocke le message dans l’historique des dialogues du salon.
6. Le système transmet le message à tous les autres utilisateurs qui ont rejoint le salon.

Postcondition: L’acteur peut communiquer avec les autres participants de l’évènement utilisant le chatbox.

Cas n°6: Quitter le salon de conversation d’un évènement

Acteur: Utilisateur

Précondition: L’acteur doit être connecté, être sur la page de profile, d’avoir participer à un évènement et d’avoir rejoint un salon de conversation.

Scénario:

1. L’acteur clique sur le boutton de fermeture du chatbox.
2. Le système aux autres utilisateurs qui se sont rejoint au salon que l’acteur quitte le salon.
3. Le système ferme le chatbox.

Postcondition: L’acteur ne voit plus le chatbox correspondant au salon de conversation de l’évènement.

Cas n°7: Rechercher et suivre une personne

Acteur: Utilisateur

Précondition: L’acteur doit être connecté.

Scénario:

1. L’acteur se déplace sur la page des utilisateurs.
2. Le système affiche tous les utilisateurs du site et une barre de recherche.
3. L’acteur recherche un utilisateur en donnant un nom.
4. Le système affiche les utilisateurs correspondant au critère donné.
5. L’acteur sélecte un utilisateur.
6. Le système affiche la page de profile de l’utilisateur sélectionné
7. L’acteur choisit de suivre cet utilisateur.
8. Le système ajoute dans les abonnés de cet utilisateur l’acteur.
9. Le système ajoute dans les abonnements de l’acteur cet utilisateur.

Alternative:

A1. Au S1, l’acteur peut se déplacer sur la page d’un évènement dont il peut voir les participants. Passe au S5.

Postcondition: L’acteur est abonné à l’utilisateur qu’il a décidé de suivre.

Cas n°8: Ne plus suivre une personne

Acteur: Utilisateur

Précondition: L’acteur doit être connecté, être sur la page de profile d’un autre utilisateur dont il est abonné.

Scénario:

1. L’acteur clique Ne plus suivre.
2. Le système supprime l’acteur des abonnés de cet utilisateur.
3. Le système supprime cet utilisateur des abonnements de l’acteur.

Postcondition: L’acteur n’est plus abonné à l’utilisateur dont il a décidé de ne plus suivre.

Cas n°9: Conversation privée avec un autre utilisateur

Acteur: Utilisateur

Précondition: L'acteur doit être connecté, être sur sa page de profile et il est abonné à quelqu'un ou quelqu'un s'est abonné à lui.

Scénario:

1. L'acteur clique la rubrique des abonnements.
2. Le système affiche les abonnements de l'acteur.
3. L'acteur clique sur le bouton Chat situé à droite d'un abonnement.
4. Le système crée un chatbox et affiche l'historique de la conversation privée.
5. L'acteur envoie un message.
6. Le système stocke le message dans l'historique de la conversation privée.
7. Le système transmet le message dans le chatbox de l'utilisateur adverse.

Alternative:

A1. En S1, l'acteur peut aussi cliquer la rubrique des abonnées et le système affiche les abonnées. Continue au S3.

A2. En S7, si l'utilisateur adverse n'a pas ouvert le chatbox correspondant à l'acteur, un chatbox sera crée automatiquement dans la session de l'utilisateur adverse.

Exception:

E1. En S7, si l'utilisateur adverse n'est pas connecté sur la page de profile, le message ne sera pas transmit (mais sera récupérer par l'utilisateur adverse dans les historiques).

Postcondition: L'acteur peut discuter avec un autre utilisateur.

Cas n°10: Editer sa page de profile

Acteur: Utilisateur

Précondition: L'acteur doit être connecté et être sur la page de profile

Scénario:

1. L'acteur choisit d'éditer sa bio.
2. Le système affiche un cadre d'édition de texte.
3. L'acteur entre sa bio et envoie les données.
4. Le système modifie la bio de l'utilisateur dans la base de donnée.
5. Le système sauvegarde la nouvelle bio sur la page de profile.

Alternative:

A1. Au S1, l’acteur choisit de changer sa photo de profile. Le système modifie le photo de profile et transmet la modification à la base de donnée.

Exception:

E1. Au S3, l’acteur choisit d’annuler. Fin d’interaction.

Postcondition: Les informations de l’acteur sont modifiées.

.

Architecture de Clarinet

Dans le cadre de la réalisation du projet, nous avons utilisé plusieurs technologies, telles que l'API ou les bases de données, et ce pour répondre à différents besoin.

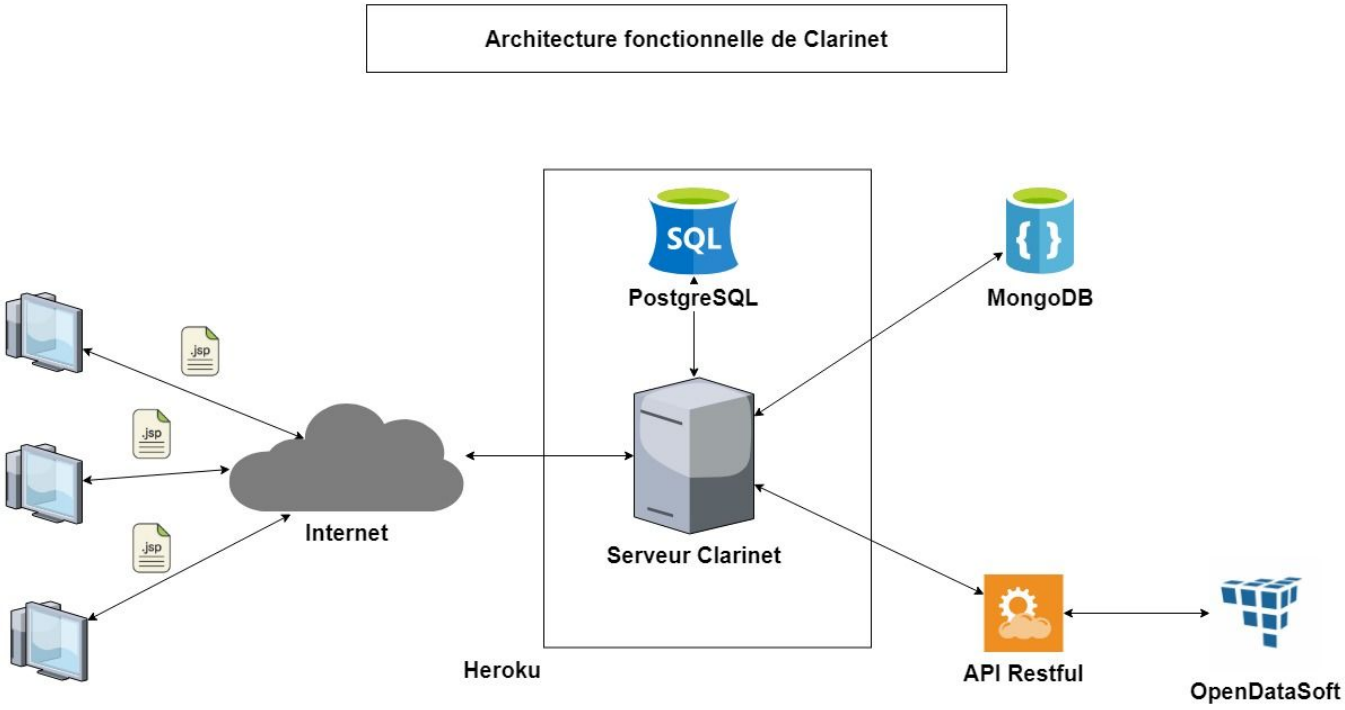


Figure 2.1. Architecture fonctionnelle de Clarinet

La figure ci-dessus représente l'architecture de notre application Clarinet.

Le serveur est un serveur Apache Tomcat réalisé selon l'architecture REST. Nous avons écrit des aussi bien des servlet java et que des JavaServer page. Nous avons utilisé des JSP car elles permettent de créer dynamiquement des pages html, et donc avoir un contenu adapté aux différents besoins de l'application.

En ce qui concerne l'hébergement du serveur, nous avons opté pour Heroku, il s'agit d'un service de cloud computing de type plate-forme en tant que service qui permet de déployer le code du serveur rapidement et simplement. En effet, nous n'avions qu'à lier le répertoire git à Heroku pour qu'il déploie automatiquement à chaque push.

Dans l'application Clarinet, nous utilisons deux types de bases de données différentes : une base de données relationnelle PostgreSQL et une base de données NoSQL.
La base de données relationnelle a pour but de contenir les données utilisateurs (pseudo, mot de passe, etc) ainsi que les abonnements des utilisateurs. Elle est hébergée par Heroku et est dédié à notre application.
Pour la base de données NoSQL, nous avons utilisé MongoDB. Nous y stockons les messages, les informations relative au profil de l'utilisateur (hormis les abonnements) telles que la photo de profil, la biographie et les événements. Elle nous est fournie par la plateforme mLab qui offre 500 Mo d'espace gratuitement.

Le serveur exploite une API RESTful externe d'OpenDataSoft pour récupérer toutes les informations liée à un événement. OpenDataSotf offre une API très complète qui permet de récupérer les informations sous forme JSON.

Pour la partie client, le serveur utilise JSP ce qui permet de créer dynamiquement les pages html en fonction des données qu'on veut exploiter. Par exemple, nous avons représenté l'utilisateur par un objet et pour générer la page de profil, nous utilisons une page JSP, qui génère le code html un peu différemment en fonction du fait si la page de profile correspond à celle de l'utilisateur connecté, ou bien si elle appartient à un autre utilisateur.

Nous avons utilisé un template Bootstrap pour le design de notre application web.

Partie serveur

Architecture du serveur

La partie serveur de Clarinet est composée de plusieurs classes réparties dans plusieurs packages différents. Elle a pour rôle de gérer et traiter toute la partie métier de l'application.

Voici les différents rôles des classes contenues dans chaque package :

- ***authentification*** : Il s'agit des classes qui gère l'authentification et la vérification de la session de l'utilisateur qui se connecte.
- ***chat*** : Ce package contient le websocket et les classes permettant d'envoyer et de recevoir des messages.
- ***database*** : Ce package contient les classes qui gèrent la connexion pour les bases de données (nous utilisons le design pattern singleton afin d'éviter de créer une connexion à chaque fois).
- ***database.utils*** : Ce package contient les méthodes pour récupérer et stocker les informations liés aux événements, utilisateurs et messages.
- ***launch*** : Ce package contient la classe pour lancer le serveur en local.
- ***map_geolocalisation*** : Ce package contient la classe permettant de récupérer les positions géographique d'un lieu à partir d'une adresse.
- ***opendatasoft*** : Ce package permet d'effectuer des requêtes à l'API externe OpenDatasoft et de récupérer les événements sous forme de classe
- ***servlet*** : Ce package contient tous les servlets nécessaires au bon fonctionnement de l'application tels que l'abonnement des utilisateurs, l'inscription aux événements, la déconnexion, etc.
- ***src/main/webapp*** : Ce dossier contient tous les JSP.

Nous avons décidé d'écrire notre serveur majoritairement avec des JSP. Cependant, certaines fonctionnalité ne nécessite pas de charger une page ou alors cela reviendrait à recharger

complètement un page inutilement tel que l'inscription à un événement, ce qui explique la présence de servlet Java (ce qui implique l'utilisation du script Javascript).

JavaServer Page

Nous utilisons les JSPs lorsque les utilisateurs visent des ressources qui nécessitent de rendre du code HTML pour la simplicité et l'efficacité. Ca nous a notamment permis d'adapter la page en fonction de la nature de l'utilisateur (comme le cas de la page de profil comme mentionné précédemment).

Nous les avons utilisées pour toutes les pages du site :

- La connexion
- L'inscription
- La fiche d'événement
- Le profil
- La page de recherche d'événements

A chaque fois qu'un utilisateur arrive sur une page (hormis les pages de connexion et d'inscription), on vérifie qu'il possède la bonne clé de session. Dans le cas contraire, il est redirigé vers la page de connexion.

Servlet

Dans le cadre de notre application Clarinet, nous utilisons principalement les JSP. Cependant certaines fonctionnalités ne requièrent pas de recharger une page, donc il est inutile de le gérer avec un JSP. C'est la raison pour laquelle que nous avons développé des servlets Java implantant les fonctionnalités nécessaires, appelés par des script Javascript.
Nous avons un servlet chaque fonctionnalité qui suit :

- Concernant la gestion de l'utilisateur : l'inscription, la déconnexion.

- Concernant le profile : L'ajout d'une biographie, la récupération de l'image de profil, la mise à jour de l'image de profile.
- Concernant la gestion des événements : L'inscription et la désinscription du participant à un événement, la récupération des événements d'un utilisateur.
- Concernant les activités sociales : les conversations via les WebSockets, et l'abonnement à un utilisateur.

Il s'agit là de servlets classiques qui récupèrent des paramètres et effectuent des traitements sur les bases de données.

Base de données

Nous utilisons deux types de bases de données. Au début, nous ne voulions utiliser qu'une base de données relationnelle, car nos schémas sont bien définis et nous n'avons pas besoin de souplesse. Cependant il n'existe à ce jour aucune offre gratuite offrant une base de données de taille conséquente (Heroku offre une base de données relationnelle PostgreSQL de 10 000 lignes). Nous nous somme donc tournés vers d'autres alternatives : les bases NoSQL. Notre choix s'est porté sur le site mLab, qui offre une base de données MongoDB de 500Mo.

Nous avons choisi PostgreSQL pour contenir les informations relatives aux utilisateurs pour des questions de sécurité. En effet, nous pouvons crypter certains champs notamment les mots de passes mais aussi appliquer des contraintes sur les champs comme le type, clé primaire/étrangère mais aussi dans la taille des champs ce que nous ne pouvons pas faire avec MongoDB.

Nous avons utilisé MongoDB pour contenir des informations moins critiques telles que les inscriptions aux évènements et les messages, et pour ne pas subir les contraintes des bases de données relationnelles et bénéficier de la flexibilité des bases NoSQL.

MongoDB

Nous avons utilisé MongoDB pour contenir les messages et les profils. Ils possèdent les structures suivantes :

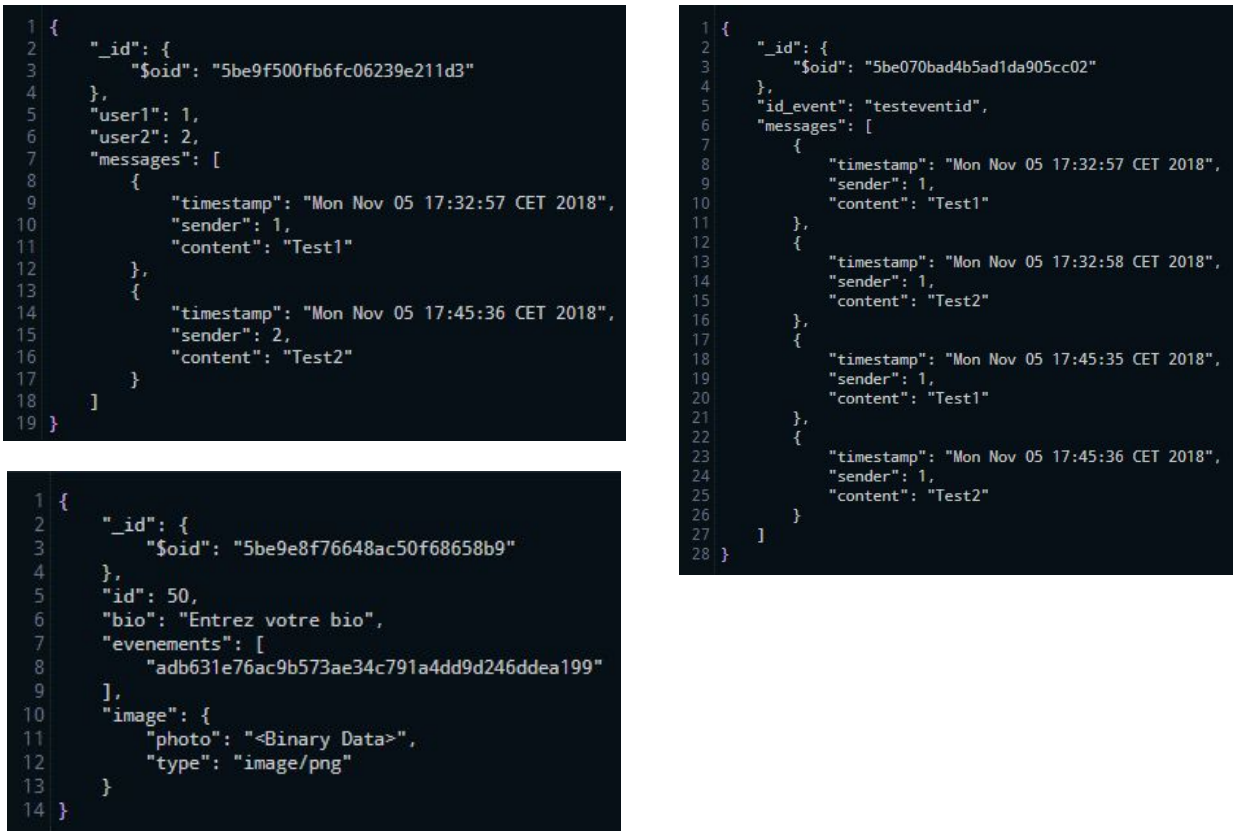


Figure 2.3 Schéma des documents MongoDB

Nous avons deux collections pour les messages, une correspondant aux conversations privées entre deux utilisateurs, et l'autre correspondant aux conversations multi-utilisateurs liées à un événement spécifique.

Dans les deux cas, l'entité message est définie par son contenu (String), sa date (String) et l'id de son auteur (int).

Pour définir une conversation privée, nous utilisons les identifiants des deux interlocuteurs (int), et une liste de messages liés à la conversation.

Quant à une conversation multi-utilisateurs, nous la définissons par l'identifiant de l'événement concerné (String), ainsi que la liste des messages de cette conversation.

Le profil est construit de l'identifiant de l'utilisateur (int), d'une biographie (String), d'un tableau événements qui contient les identifiants des événements (String) auxquels l'utilisateur a déjà participé ou va participer, et finalement d'une image de profil par défaut qu'il peut changer par la suite. L'image est un document qui contient le type de l'image (String) et l'image en ByteArray.

PostgreSQL

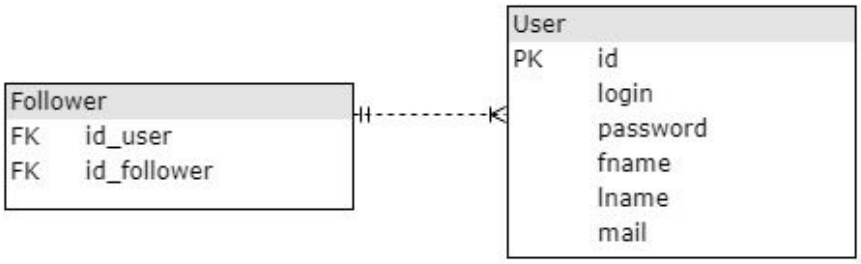


Figure 2.4. Schéma de la base de données relationnelle

Pour la base de données PostgreSQL nous avons créé une table qui contient les informations d'un utilisateur et une autre qui contient les abonnements.

WebSocket

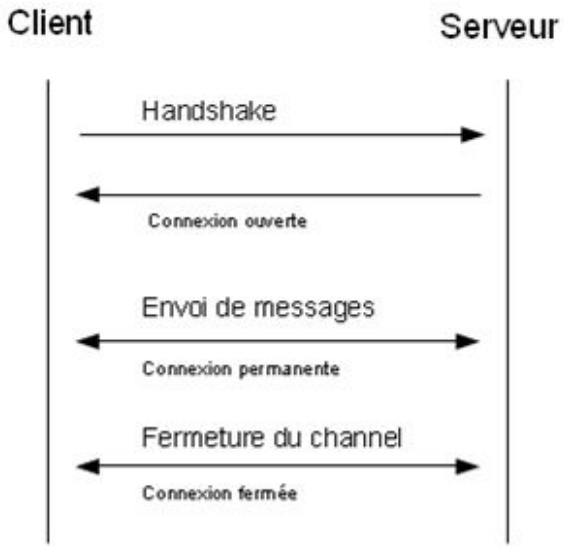


Figure 2.5. Fonctionnement d'un websocket

Pour gérer le chat, nous avons utilisé le protocole Websocket pour permettre les échanges entre les utilisateurs. WebSocket est une alternative à Ajax qui permet de mettre en place des solutions quasi temps réel pour recevoir des données. Cependant sa compatibilité est limitée aux navigateurs récents. Le protocole complet est supporté par Internet Explorer 10, Chrome depuis la version 16, Firefox depuis la version 11, Safari depuis la version 6.0.

Au niveau du serveur, nous avons besoin de créer un @ServerEndpoint qui attend une demande de connexion d'un client. Une fois la connexion établie, le client et le serveur peuvent alors émettre et recevoir des messages. Cette connexion peut être fermée par le client ou par le serveur. Les messages échangés entre utilisateurs sont reçus et traités par la méthode annotée par @OnMessage. Ces messages sont des chaînes de caractères avec un format JSON pour simplifier la récupération des données. Dans notre application, il existe plusieurs types de message :

- Les messages du système: il s’agit de récupérer les données de l’utilisateur, signaler l'arrivée d’un nouveau utilisateur dans un salon et signaler qu’un utilisateur a quitté un salon.
- Les messages vers un salon de conversation multi-utilisateurs
- Les messages privés vers un autre utilisateur.

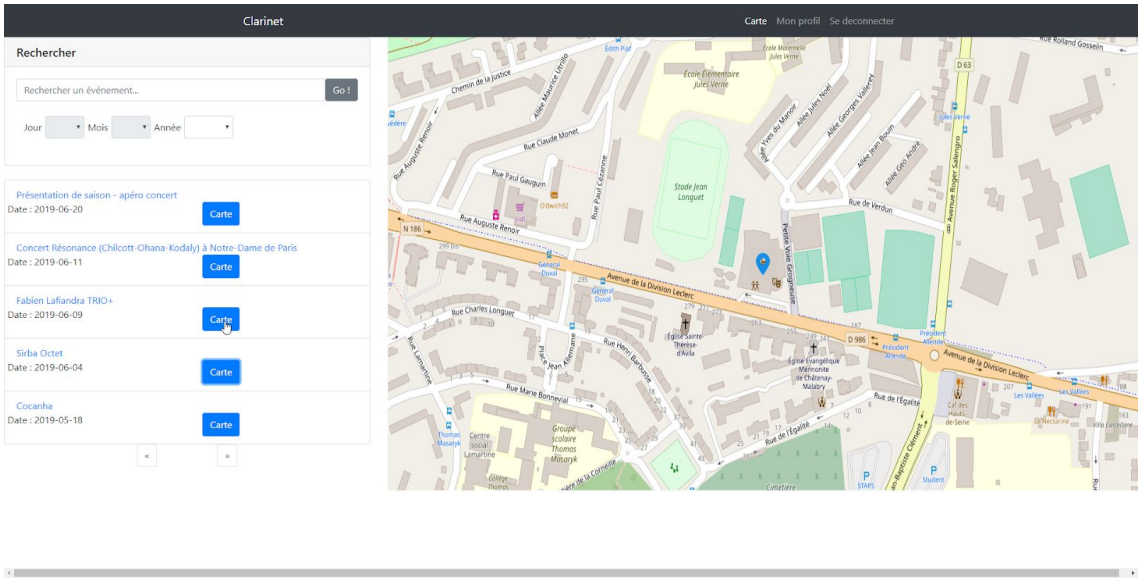
Tous les messages qui ne sont pas du type message du système sont stockés dans la base de données afin de garder l’historique des dialogues.
Les messages privés reçu par le Websocket sont envoyés vers la session du récepteur. Les messages dont destinataire est un salon de conversation sont envoyés à tous les utilisateurs qui ont rejoint le salon.

Partie client

Bootstrap

Pour le design du site, nous nous sommes servi d’un template utilisant le framework Bootstrap. Les tendances depuis ces dernières années en terme de design vont vers une interface simple et très épurée tel que le Material Design de Google.

Interface de recherche d’événement



L'interface de recherche d'événements de Clarinet permet de rechercher des événements partout en France. Elle offre aussi une option de tri par date et propose de montrer sur la carte le lieu de déroulement de l'événement. Pour indiquer le lieu, nous nous sommes servi des APIs offertes par OpenLayers et OpenStreetMap.

Affichage de la carte

Afin d'afficher une carte sur notre interface utilisateur, nous utilisons l'objet Map offert par l'API OpenLayer qui permet d'afficher une carte et d'y ajouter un marker à des latitude et longitude données. Les informations qu'on récupère de l'événement ne précise pas ses coordonnées géographique mais l'adresse du lieu. Ainsi, pour trouver les longitude et latitude suivant une adresse, nous avons utilisé l'API OpenStreetMap. Pour trouver les coordonnées d'une adresse, nous faisons un appel AJAX comme ceci :

```
function updateMap(map, address){
      var position = unescape(encodeURIComponent(address));
      console.log(address);
      $.ajax({
           type: "GET",
           url: "https://nominatim.openstreetmap.org/search",
           data: "q="+position+"&format=json",
           success: function(resp){
               console.log(resp);
               var latitude = parseFloat(resp[0].lat);
               var longitude = parseFloat(resp[0].lon);

               map.getView().setCenter(ol.proj.fromLonLat([longitude, latitude]));
               map.getView().setZoom(17);
               var marker = new ol.Feature({
                   geometry: new ol.geom.Point(
                     ol.proj.fromLonLat([longitude, latitude])
                   )
               });
               var iconStyle = new ol.style.Style({
                   image: new ol.style.Icon(({
                     src: 'marker.png'
                   }))
               });
               marker.setStyle(iconStyle);
               var vectorSource = new ol.source.Vector({
                   features: [marker]
               });
               var markerVectorLayer = new ol.layer.Vector({
                   source: vectorSource,
               });
               map.addLayer(markerVectorLayer);
          }

         });

}
```

Lorsque la réponse de la requête est retournée, nous récupérons les coordonnées associées à l'adresse et nous pointons l'endroit correspondant sur la carte.

Recherche d'événements

Nous ne pouvions pas charger directement tous les événements existants sur le navigateur car la base de données est de taille conséquente. Nous ne pouvions pas non plus à chaque fois recharger le JSP et ajoutant en paramètre la page souhaitée ; cela impacterait les performances et donc l'expérience utilisateur. C'est pourquoi nous avons donc décider de créer un servlet qui renvoie sous forme JSON un certain nombre d'événements à partir de l'indice de début d'intervalle. Ensuite, on modifie le code HTML directement à partir de JavaScript.

Nous ne faisons pas directement appel à l'API OpenDatasoft pour récupérer les événements mais passons par un servlet de notre serveur. Tout d'abord, la connexion du serveur est plus performante, et nous filtrons les données et envoyons le strict nécessaire au navigateur, ce qui a pour effet de réduire la bande passant consommée pour la requête.

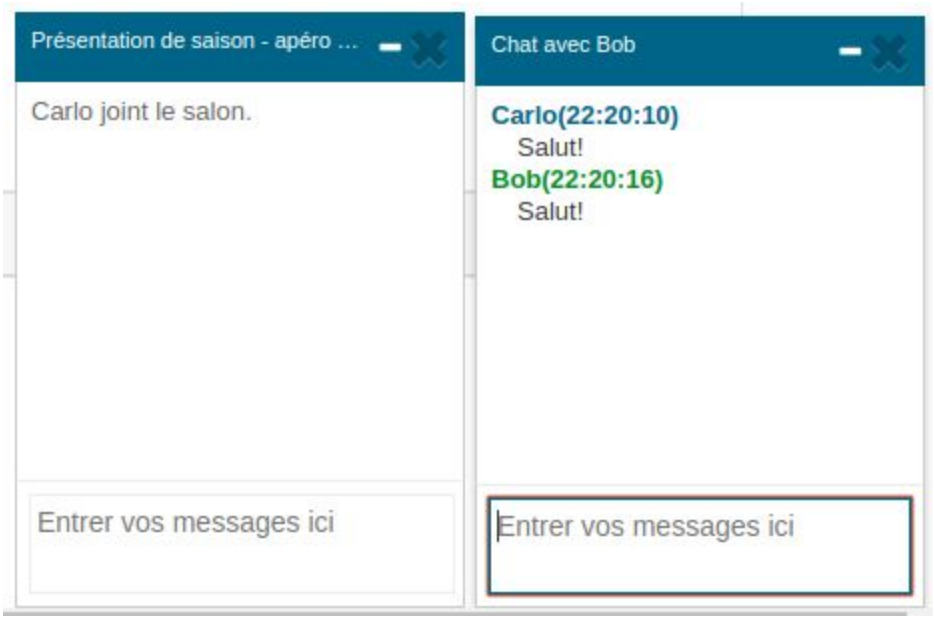
```
$.ajax({
      type: "GET",
      url: "/get/events",
      data: "query="+encodeURI(document.getElementById("myInput").value)+"&nbrows=5"+"&startrow="+(cpt*5)+"&date="+date,
      success: function(resp){
          var answer = "";
          var allRes ="";
          var results = resp["events"];
          console.log(resp);
          for(var i = 0; i < results.length; i++){
              var obj = results[i];

              var res="<div class=\"list-group-item\">"+
                    "<a href=\"/event.jsp?id_event="+obj.id+"\">"+obj.title+"</a>"+
                    "<div class=\"row\">" +
              "<p  >Date : "+obj.start_date+" </p> <div class =\"col-lg-4\"></div><button class=\"btn btn-primary\" onclick=\"updateMap(map, '"
              +obj.address.replace(/,/g,'').replace(/(\r\n\t|\n|\r\t)/gm,"")+"')\">Carte</button>"
              +"</div></div>";
              allRes +=res;
              var ans = "<li>"+
              "<b>Titre: </b>"+obj.title+" <br>"+
              "<b>Date: </b>"+obj.start_date+" <br>";
              if(!(typeof obj.tags === 'undefined')){
                  ans += "<b>Tags: </b>"+obj.tags+"<br>";
              }
              ans +=  "<button class=\"eventbtn\" onclick=\"updateMap(map, '"+obj.address.replace(/,/g,'').replace(/(\r\n\t|\n|\r\t)/gm,"")+"')\">Carte</button>"+
              "<a class=\"linkbtn\" href=\"/event.jsp?id_event="+obj.id+"\">Lien</a> <br>";
              ans += "</li>";
              answer += ans;
          }

          document.getElementById("events").innerHTML = allRes;
      }
    });
```

En somme, nous effectuons une requête AJAX sur notre serveur pour récupérer les événements. On récupère 5 événements à chaque fois, et on génère la liste d'événements. Dans la réponse retournée par le serveur se trouve notamment l'adresse de l'événement, pour permettre à l'utilisateur de pointer (via un bouton) l'événement sur la carte comme vu précédemment.

ChatBox



Boîtes de dialogues (gauche: salon de conversation, droite:conversation privée)

Lorsque l'utilisateur se présente sur la page de profil, un endpoint côté client sera initialisé et sera connecté au endpoint côté serveur. Une fois la connection réussie, l'utilisateur sera capable de recevoir des messages ou d'envoyer des messages vers le endpoint serveur. Après la connection, un message contenant les informations de l'utilisateur sera envoyé pour initialiser les propriétés de la session websocket de cet utilisateur.

Pour effectuer la communication entre utilisateurs nous avons créé des boîtes de dialogue à l'aide de jQuery. Lorsque l'utilisateur ouvre une boîte de dialogue, une demande sera envoyée sous forme de message pour chercher l'historique de ce dialogue. S'il existe, l'historique sera affiché directement dans la boîte de dialogue à l'initialisation.

Lorsque l'utilisateur envoie un message dans une boîte de dialogue, ce message sera envoyé au endpoint serveur avec les informations de l'expéditeur, du receveur et le contenu du message ainsi que l'heure courante.
Dans le cas de message privée entre deux utilisateurs, le endpoint côté serveur va d'abord enregistrer ce message dans la base de donnée correspondant aux historiques des dialogues

avec les id des deux personnes. Puis, l'endpoint va rechercher dans les sessions en ligne si la session du receveur est en ligne. Si le receveur est présent, un message sera envoyé vers le endpoint client de ce receveur obtenu à l'aide de sa session. Le endpoint client du receveur va examiner l'expéditeur du message reçu puis ouvre la boîte de dialogue correspondante si celui-ci n'est pas ouvert. Enfin, le message sera affiché dans la boîte de dialogue.

Dans le cas du salon de conversation multi-utilisateur, seuls les utilisateurs ayant ouvert la boîte de dialogue du salon peuvent recevoir des messages. En effet, lorsqu'un utilisateur rejoint un salon de conversation, il est ajouté dans le ChatRoom correspondant à cet évènement. Ce ChatRoom détient une liste de sessions, quand il reçoit un message, il transmet ce message à tous les sessions. Les nouveaux membres du salon ont accès à l'historique des dialogues du salon, cela leurs permet de ne pas rater des informations concernant l'évènement. Nous considérons qu'un utilisateur rejoint le salon lorsqu'il a ouvert la boîte de dialogue du salon, et quitte lorsqu'il la ferme.

Extensions et Améliorations

Bien que notre projet Clarinet soit fonctionnel, des points peuvent être amélioré.

En effet, la géolocalisation des événements n'est parfois pas possible à cause de l'API d'OpenLayers qui ne trouve pas l'adresse, ceci peut être résolu en changeant d'API pour celui de Google (Maps) qui lui est plus performant. Cependant, l'utilisation de cet API est payante.

Une fonctionnalité à ajouter serait l'ajout de notification pour les messages, de notification de rappel pour les événements auxquels un utilisateur s'est inscrit et qui ont lieu le jour même.

Nous pouvons aussi améliorer la dimension sociale, par exemple afficher les utilisateurs connectés ou déconnectés, bloquer un utilisateur, etc…

Pour permettre à un utilisateur qui a oublié son mot de passe de récupérer son compte, on devrait rendre obligatoire la saisie d'une adresse e-mail et créer un serveur mail qui envoie des réponses automatiques.

On pourrait ajouter un peu d'intelligence artificielle et proposer dynamiquement des événements qui pourraient intéresser l'utilisateur en fonction des événements auxquels il a déjà participé, ou en fonction de son historique de recherche.

Le design est très minimaliste, et pourrait être revu pour ajouter un peu d'originalité, compte tenu du fait que le concept de notre projet est "fun" et ouvert à tout public.

Concernant la page de profil, nous voulions qu'une fois que l'utilisateur à charger sa nouvelle photo de profil, la photo qu'il vient d'envoyer apparaisse directement sur la page HTML (sans recharger la page). Après diverses tentatives et des heures de recherche, nous n'avons pas réussi à charger le nouveau tableau de bytes, qui correspond à la nouvelle image, sur la page.

La conversion du tableau de bytes en image ne s'effectue pas et ce qui s'affiche est les caractères du tableau.

Finalement, si nous avions disposé plus de temps, nous aurions homogénéisé la gestion des erreurs de base de données et des requêtes AJAX. En effet, comme c'est une production de trois personnes, les erreurs sont gérées différemment.

Etude financière

Coût du projet

Nous avons réalisé l'application Clarinet en utilisant des technologies open-sources (Apache Tomcat, PostgreSQL, MongoDB) ce qui nous évite de payer une licence d'utilisation. Au niveau des plateformes de services (Heroku, mLab), nous avons utilisés les offres gratuites. Concernant le matériel informatique, nous avons travaillé sur nos machines personnelles. Ainsi, le coût total du projet s'élève à 0 euro.

Cependant, il faudrait prendre en compte le coût de main d'oeuvre. Nous sommes trois étudiants qui ont travaillé en moyenne 2 jours par semaine depuis la semaine du 17 septembre 2018 jusqu'à la semaine du 12 novembre 2018. Ce qui fait 9 semaines au total, soit 18 jours de travails par personne. Nous avons donc au total 54 jours de travail cumulé pour le projet.
Si nous étions payés au minimum en tant que stagiaire à raison de 7 heures par jour, le coût total serait de 1417.5 euros, et au smic il serait de 2957.74 euros net.

De plus, il faudrait considérer investir dans des plateformes plus performantes. Les offres gratuites atteignent rapidement leurs limites, les bases de données offrent peu d'espace de stockage (MongoDB 500Mo, PosgreSQL 10 0000 lignes) et les connexions sont aussi limités. De plus, les performances du serveur sont moindres (512Mb de RAM/mode économie d'énergie forcé/lenteur).

Etude de marché

A notre connaissance, il n'existe pas à ce jour de concurrent direct à notre application Clarinet. Cependant, les réseaux sociaux classiques comme Facebook propose des pages à événements, mais les événements sont souvent consultés grâce aux partages et "bouche à oreille", on ne recherche pas forcément des événements sur Facebook. Un second concurrent

potentiel serait les billetteries comme la Fnacspectacle ou Leclerc-billetterie qui possèdent un catalogue très complet, cependant, ce sont des événements payants et donc ne recense pas les événements gratuit comme Clarinet. De plus, il n'y a aucune dimension social sur ces plateformes.
Ceci nous conforte dans l'idée que Clarinet possède un réel potentiel.

Monétisation

Il existe une multitude de moyen de monétiser notre application, en commençant par ajouter des rangs aux utilisateurs où ils faudraient payer pour monter son rang et débloquer des fonctionnalités plus poussées.
Les organisateurs d'événements peuvent aussi nous rémunérer pour que leurs événements soit plus visibles et mis en avant sur notre application.
Nous pourrions aussi proposer des partenariats avec les billetteries : il y aurait des liens vers ces billetteries sur la page des événements payants.

Conclusion

De part tout l'aspect social sur lequel nous avons travaillé, Clarinet remplit son objectif : de réunir des personnes qui ont les mêmes goûts musicaux. Ce projet nous a permis d'apprendre énormément sur le développement web, nous avons pu travailler avec des technologies qui nous étaient nouvelles tels que Bootstrap, MongoDB, PostgreSQL, JSP etc. Nous avons aussi appris à créer complètement une application web, et ainsi effectuer le travail d'un développeur full stack tels que la modélisation de la base de données, le déploiement de l'application, développer le back et front end.

Nous avons aussi pu approcher les problématiques d'entreprises grâce à ce projet. En effet, nous avons dû nous renseigner sur le cadre d'utilisation des technologies sous propriétés intellectuelles, étudier l'aspect financier d'une application tels que les coûts de production, de mise en service ou bien une étude de marché.