Rapport de conception

Adonis Najimi,
Valentin Stern,
Vincent Albert,
Théo Gerriet

11 janvier 2014

1 Présentation du projet

1.1 Étude de l'existant

Nous avons trouvé deux logiciels qui sont utilisés pour la création de partitions et l'édition de celles-ci.
GuitarPro est un logiciel payant qui permet de réaliser des partitions et de les enregistrer via un micro.
`http://www.guitar-pro.com/fr/index.php`
TuxGuitar est un logiciel opensource qui permet d'éditer et de créer une partition.
`http://tuxguitar.herac.com.ar/`
Cependant TuxGuitar ne permet pas de transformer un son joué en une partition éditable.
Aucun de ces logiciels ne permet de partager ces partitions sur un réseau social. Ce réseau est donc ce qui permet à notre programme de se distinguer des autres solutions disponibles.

1.2 Cahier des charges fonctionnel

Priorité haute
- Analyse et enregistrement des notes
- Envoi des partitions
- Accordeur
- Jouer une partition

Priorité moyenne
- Apprentissage de la composition

Priorité basse
- Analyse automatique d'une partition

1.3 Repartition des charges

Adonis NAJIMI : Toute l'application web
Valentin STERN : Model du client(50%)
Vincent ALBERT : Controleur et vue du client
Théo GERRIET : Model du client(50%)

1.4 Echéancier

Le projet sera ponctué d'étapes décisives dont l'échéance sera à respecter.

20 janvier :

- Création de la base de données
- Création de la classe abstraite de la vue
- Création du modèle
- Création du modèle Active Record sur le site web et mise en place des tests unitaires
- Création des routes sur le site web

30 janvier :

- Implémentation des algorithmes dans le modèle
- Création des vues du site web
- Création des différents widgets : accordeur, partition, note, liste de partitions, options

10 février :

- Création des contrôleurs du site web
- Outils de manipulation de la base
- Création et implémentation du contrôleur de l'application dans l'interface graphique

20 février :

- ajouts de fonctionnalités non essentielles sur le site et finition des contrôleurs
- Liaison du logiciel avec la base de données

30 février :

- Finalisation du projet

2 Client C++

2.1 Présentation du client

La fonctionnalité majeure du client et de permettre à l'utilisateur de composer et enregistrer ses propres musiques en jouant avec sa guitare. Il n'aura qu'a jouer ce qu'il veut et l'application traitera les notes jouées et les enregistrera dans une partition. Il pourra par la suite modifier les notes de la partition en cas d'erreur.

Le client comporte d'autres fonctionnalitées annexes non essentielles mais intéressantes cependant. L'utilisateur pourra, avant de commencer à jouer, accorder sa guitare afin de jouer le plus juste possible. Le client a pour but de permettre à l'utilisateur de composer ses chansons, et il y aura donc une partie permettant l'apprentissage des bases de la composition à la guitare de manière interactive. Il pourra apprendre à créer des accords, jouer sur différentes gammes ou encore apprendre le rythme.

Une fonctionnalitée permettant d'envoyer directement sa partition au serveur web sera également présente afin de simplifier le partage de ses créations.

Enfin, l'application permettra aussi d'analyser une partition afin de trouver quelle est la gamme principale de celle-ci.

2.2 Description du contenu du modele

Cette partie de l'application est principalement basée sur l'analyse et l'enregistrement des notes jouées par l'utilisateur. Le principe est d'analyser en temps réel les fréquences à la base du son de la guitare, et de faire en sort de constituer un accord avec les notes ayant le volume le plus fort et l'enregistrer. De plus, le modèle comporte également toutes les informations sur les préférences de l'utilisateur pour l'application, ainsi que des fonctions pour enregistrer les données. Nous avons choisi de sauvegarder les fichiers au format JSON.
Afin de jouer une partition, nous utiliserons des sons de guitare en midi d'une durée courte, que nous jouerons plusieurs fois afin de correspondre à la durée de la note sur la partition, il y aura également la possibilité de changer la note jouée pour une note qui correspond plus à sa propre guitare pour une meilleure expérience utilisateur.

Nous avons décidé d'utiliser la librairie FMOD Ex pour le modèle, car cette librairie est complète et fournit toute les fonctionnalités dont nous avons besoin. En effet, elle nous permet de lire et d'enregistrer des sons, et également de récupérer les fréquences d'un son grâce à la courbe de ce son. Elle utilise pour cela une tranformée de Fourrier rapide.

Fréquences des hauteurs (en hertz) dans la gamme tempérée

Note\octave	0	1	2	3	4	5	6	7
Do	32,70	65,41	130,81	261,63	523,25	1046,50	2093,00	4186,01
Do♯ ou Ré♭	34,65	69,30	138,59	277,18	554,37	1108,73	2217,46	4434,92
Ré	36,71	73,42	146,83	293,66	587,33	1174,66	2349,32	4698,64
Ré♯ ou Mi♭	38,89	77,78	155,56	311,13	622,25	1244,51	2489,02	4978,03
Mi	41,20	82,41	164,81	329,63	659,26	1318,51	2637,02	5274,04
Fa	43,65	87,31	174,61	349,23	698,46	1396,91	2793,83	5587,65
Fa♯ ou Sol♭	46,25	92,50	185,00	369,99	739,99	1479,98	2959,96	5919,91
Sol	49,00	98,00	196,00	392,00	783,99	1567,98	3135,96	6271,93
Sol♯ ou La♭	51,91	103,83	207,65	415,30	830,61	1661,22	3322,44	6644,88
La	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00	7040,00
La♯ ou Si♭	58,27	116,54	233,08	466,16	932,33	1864,66	3729,31	7458,62
Si	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07	7902,13

FIGURE 1 – Toutes les frequences possibles

2.3 UML du modele de l'application

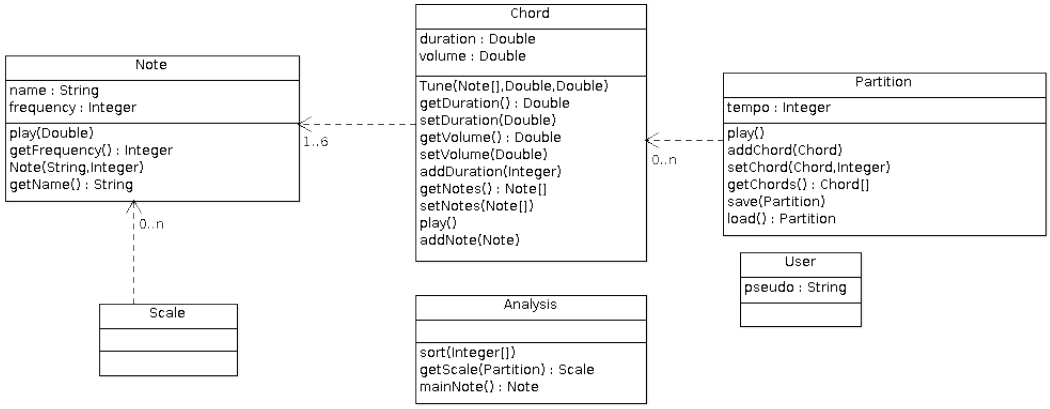


FIGURE 2 – UML du modele

La classe Note représente une Note qui n'est pas jouée, c'est-à-dire qu'elle contient la définition d'une note seulement.
La classe Chord permet donc de représenter de 1 à 6 note jouée simultanément tel un accord à la guitare.
La classe Partition représente donc une partition comme l'utilisateur va la voir.
La classe Analysis propose différentes fonctions essentielles à l'application. Cette classe propose également des fonctionnalités pour l'apprentissage de la composition de musiques.
La classe Scale représente une gamme. Une gamme est une suite de notes, et la classe est donc composée d'un certain nombre de note. La classe User est présente pour enregistrer le pseudonyme de l'utilisateur pour pouvoir envoyer

des partitions sur le site. Lorsque l'utilisateur va jouer une note, le logiciel va récupérer toutes les fréquences entrant par le micro et les trier en fonction de leur volume. Il est donc aisé par la suite de trouver quelles sont les notes qui sont le plus jouées.

2.4 Algorithmes

Structures de donnees utilisees dans les algorithmes :
Note < frequence : entier, nom : chaine >
EntreeMicro : Type correspondant à l'entrée micro de l'ordinateur. Il permet de récupérer les fréquences à un certain instant
Accord : < notes : tableau Note[0..6] , courant : entier>
Partition : < accords : Liste(Accord) >
Gamme : < notes : Liste(Note)>
Récupérer une note grâce à sa fréquence

```
fonction recupererNote(frequence : entier, notes : Note[0..n], n : entier) : Note
debut
    min <- 0
    max <- n
    trouve <- faux
    Tant que min <= max et non trouve faire
        indice <- (max + min) /2
        frequenceNote <- notes[indice].frequence
        Si frequence = frequenceNote
        alors
            retour <- notes[indice]
            trouve <- vrai
        sinon
            Si frequence < frequenceNote
            alors
                max <- indice
            sinon
                min <- indice
            fsi
        fsi
    ftantque
    Si non trouve
    alors
        retour <- notes[min]
    fsi
    retourne retour
fin
```

```
lexique
frequence : entier : Fréquences de la note à récupérer
notes : Note[0..n] : Toutes les notes possibles
n : entier : Nombre de notes possibles
max : entier : indice maximal dans le tableau
```

min : entier : indice minimal dans le tableau
trouve : booleen : Booléen à vrai si on a trouvé la note (avec exactement la même fréquence)
indice : entier : Indice en cours dans le tableau
frequenceNote : entier : Frequence de la note en cours d'analyse
retour : Note : Note correspondant à la fréquence

Fonction de placement nécessaire à la fonction trier

```
fonction placer(meta : tableau entier[0..n], frequences : tableau entier[0..n], n : entier, inf : entier, sup : entie
debut
    inda <- inf
    a <- frequences[meta[inf]]
    inf <- inf + 1
    Tant que sup >= inf faire
        Si frequences[meta[inf]] > a
        alors
            Tant que sup >= inf ou frequences[meta[[sup]] > a faire
                sup <- sup - 1
            ftantque
            temp <- meta[sup]
            meta[sup] <- meta[inf]
            meta[inf] <- temp
            sup <- sup - 1
        fsi
        inf <- inf + 1
    ftantque
    temp <- meta[sup]
    meta[sup] <- meta[inda]
    meta[inda] <- temp
    retourne sup
fin
```

lexique
meta : tableau entier[0..n] : Meta donnée contenant les numéros de frequences
frequences : tableau entier[0..n] : Les indices sont les numéros de fréquences et les valeurs sont le volume
n : entier : Taille des tableaux frequences et meta
inf : entier : Borne inférieur utilisée pour le placement
sup : entier : Borne supérieure utilisée pour le placement
inda : entier : Valeur de inf avant modification
a : entier : Valeur du volume de la fréquence
temp : entier : Variable temporaire

Fonction de tri des fréquences

```
fonction trier(meta : tableau entier[0..n], frequences : tableau entier[0..n], n : entier, inf : entier, sup : entier)
debut
```

```
    Si inf < sup
    alors
        indice <- placer(meta, frequences, inf, sup)
        trier(meta, frequences, n, inf, indice - 1)
        trier(meta, frequences, n, indice + 1, sup)
    fsi
fin
```

lexique
meta : tableau entier[0..n] : Meta donnée contenant les numéros de frequences
frequences : tableau entier[0..n] : Les indices sont les numéros de fréquences et les valeurs sont le volume
n : entier : Taille des tableaux frequences et meta
inf : entier : Borne inférieur utilisée pour le tri
sup : entier : Borne supérieure utilisée pour le tri
indice : entier : Indice retournée par le placement pour diviser le tableau

Montrer la note principale(pour l'accordeur)

```
fonction montrerNote(mic : entreeMicro)
debut
    accorder <- vrai
    Tant que
        n <- 512
        freqs <- recupererFrequences(mic, n)
        Pour i de 0 a n faire
            meta[i] <- i
        fpour
        meta <- trier(meta, freqs, n, 0, n - 1)
        ecrire recupererNote(meta[0])
    ftantque
fin
```

lexique
mic : entreeMicro : Entrée micro utilisée
accorder : booleen : Booleen à vrai si on continue à accorder. Il peut être changer grâce à un clic dans l'application
freqs : freqs : tableau entier[0..n] : Frequences envoyées par mic

Enregistrement d'une partition

```
fonction enregistrer(mic : entreeMicro, seuilHaut : entier, seuilBas : entier, tempo : entier) : Partition
debut
    enregistrer <- vrai
    j <- 0
    accord <- accordVide() //Renvoi un accord vide
```

```
    Tant que enregistrer faire
        frequences <- recupererFrequences(mic)
        n <- 512
        freqs <- recupererFrequences(mic, n)
        Pour i de 0 a n faire
            meta[i] <- i
        fpour
        meta <- trier(meta, frequences, n, 0, n - 1)
        complet <- faux
        fini <- faux
        i <- 0
        new <- false
        Tant que non complet et non fini faire
            indiceFrequence <- meta[i]
            frequence <- frequences[indiceFrequence]
            si frequence > seuilHaut
            alors
                new <- vrai
                ajouterAccord(partition, accord)
                accord <- accordVide()
                complet <- ajouterNote(accord, recupererNote(frequence)
            sinon
                si non new
                alors
                    Si frequence > seuilBas
                    alors
                        acc <- accordVide()
                        complet <- ajouterNote(accord, recupererNote(frequence))
                    sinon
                        fini <- vrai
                    fsi
                sinon
                    fini <- vrai
                fsi
        ftantque
        Si non new
        alors
            si non estVide(acc)
            alors
                si acc = accord
                alors
                    ajouterTemps(1000 / tempo / 16)
                fsi
            fsi
        fsi
        i <- i + 1
        attendre(1000 / tempo / 16)
    ftantque
    retourne retour
fin
```

Nous attendons pendant 1000 / tempo / 16 car c'est la plus petite unité de temps représentable dans une partition. L'utilisateur doit rentrer le tempo dans lequel il compte jouer. Nous attendons en milliseconde.

```
fonction ajouterNote(accord : Accord, note : Note) : booleen
debut
    si accord.courant = 5
    alors
        retour <- vrai
    sinon
        accord[courant] = note
        accord.courant <- accord.courant + 1
    fsi
    retourne retour
fin


fonction ajouterAccord(partition : Partition, accord : Accord)
debut
    adjqlis(partition, accord)
fin


fonction analyserGamme(partition : Partition, gammes : tableau Gamme[0..n], n : entier) : Gamme
debut
    notes <- partition.notes
    indice <- tete(notes)
    Tant que non finliste(notes, indice) faire
        accord <- val(notes, indice)
        Pour chaque note dans accord faire
            tnotes[note] <- tnotes[note] + 1
        fpourchaque
indice <- suc(notes, indice)
    ftantque
    Pour i de 0 a n faire
        gamme <- gammes[i]
        pourcentage[i] <- pourcentageGamme(gamme, tnotes)
    fpour
pourcentMax <- 0
    Pour i de 0 a n faire
        si pourcentage[i] > pourcentMax
        alors
            gammeMax <- gammes[i]
        fsi
    fpour
    retourne gammeMax
fin
```

Algorithme utilisé pour déterminer combien il y a de notes d'une gamme dans un ensemble de notes

```
fonction pourcentageGamme(gamme : Gamme, notes : table (Note => entier)) : entier
debut
    indice <- tete(gamme)
    nbNotesGammes <- 0
    Tant que non finliste(gamme, indice) faire
        note <- val(gamme, indice)
        nbNotesGammes <- nbNotesGammes + accestab(notes, note)
    ftantque
    nbTotal <- 0
    Pour chaque valeur dans notes faire
        nbTotal <- nbTotal + valeur
    fpourchaque
    retourne (nbNotesGammes / nbTotal ) * 100
fin
```

2.5 UML de l'interface graphique

L'interface graphique sera codée en C++ et respectera le modèle MVC.
Ainsi seul le contrôleur modifiera le modèle, tandis que la classe NoSkin, représentant l'interface, ne fera qu'afficher les données.
Ces deux classes sont donc représentées sur l'UML et sont complémentées par des classes complémentaires telles que la classe FormatException qui permet de gérer plus finement la conversion JSON/Objet.
Les classes TabWidget, Options et Chord seront utilisées par NoSkin pour créer les fenêtres de l'accordeur, des paramètres et le widget particulier de la tablature.

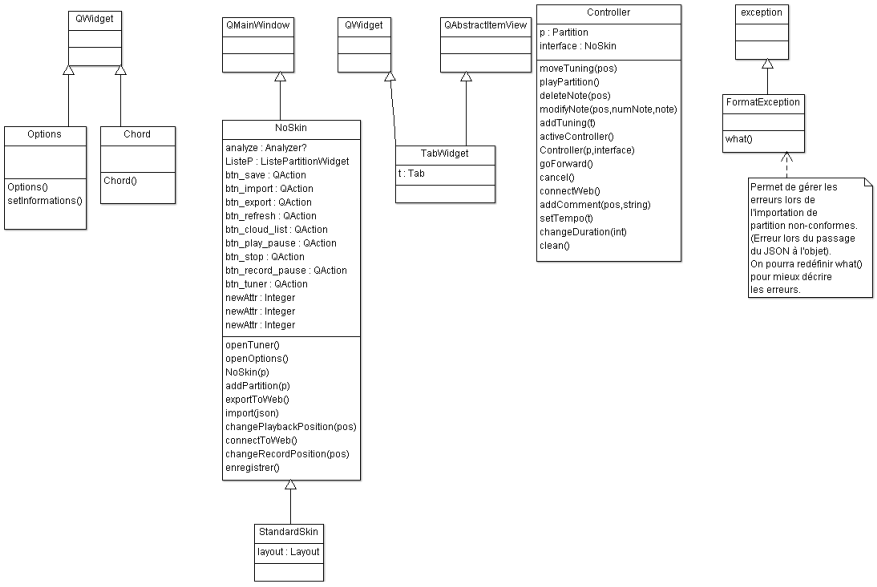


FIGURE 3 – UML de l'interface graphique

2.6 Maquettes du client

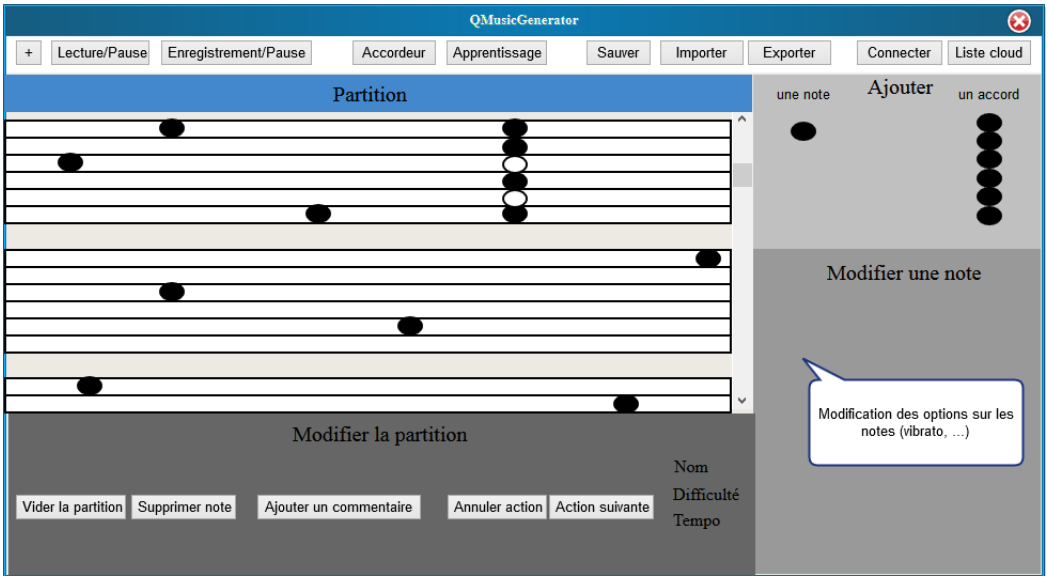


FIGURE 4 – Page principale du client

Cette fenêtre est la fenêtre principale de notre client. A partir de celui-ci, il sera possible de créer des partitions vierges, d'en ouvrir des préexistantes, d'en lire et d'enregistrer des notes à partir de l'entrée audio ainsi que de les sauvegarder.

Mais il sera aussi possible d'ouvrir un accordeur, de télécharger des partitions hébergées sur le site web, et d'en uploader.

Il sera en effet possible de connecter directement le client avec le site à l'aide des identifiants de l'utilisateur.

De plus, il y aura aussi évidemment toutes les options permettant de modifier la partition en cours, tels que son nom, sa difficulté, son tempo, ...

Enfin, on pourra modifier la partition manuellement en déplaçant les notes ou en en ajoutant.

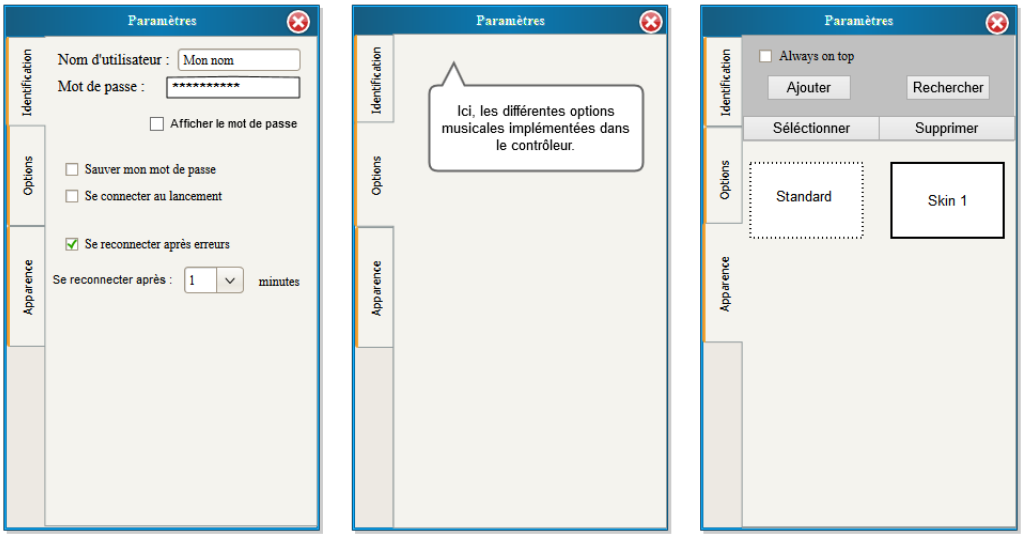


FIGURE 5 – Fenêtre des paramètres du logiciel
Cette fenêtre permettra de modifier les paramètres du logiciel.
Cela recouvre les paramètres d'authentification, ceux spécifiques à la synthèse de musique en elle-même, mais aussi l'apparence du client via l'utilisation de skins.

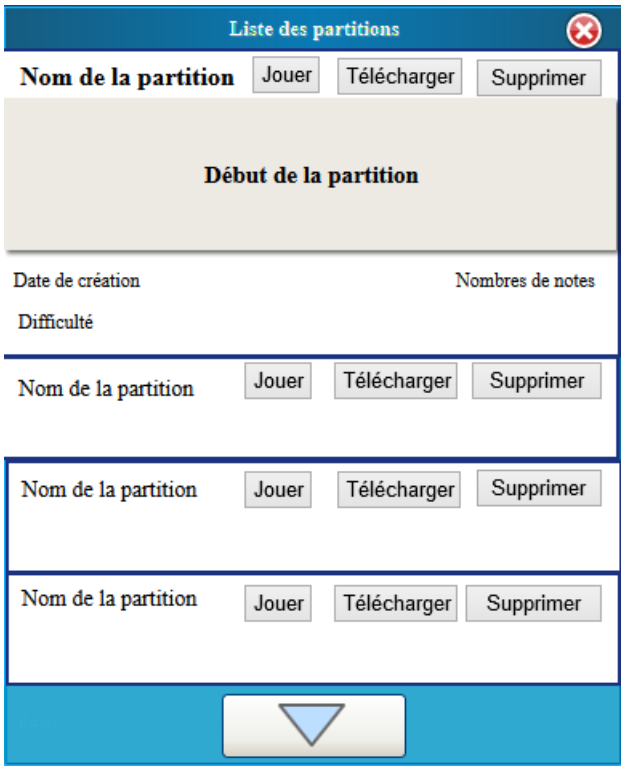


FIGURE 6 – Fenêtre des listes de partitions hébergées sur le site web
Cette fenêtre listera toutes les partitions que l'utilisateur a uploadé sur le site internet si celui-ci est connecté.
A partir de cette fenêtre, il pourra soit les supprimer du site web, soit les télcharger, ou encore les jouer.

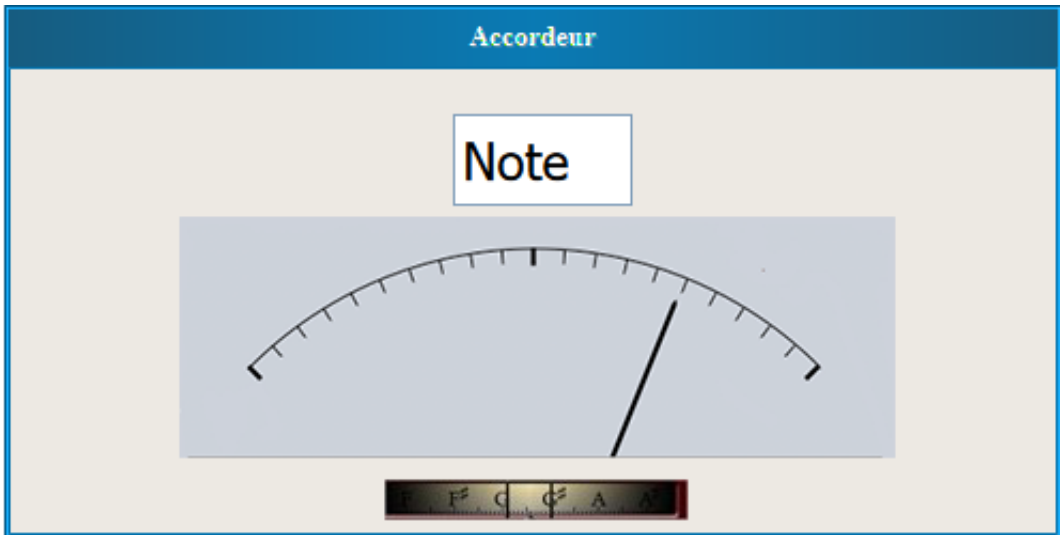


FIGURE 7 – Accordeur
Cette fenêtre affichera un accordeur qui analysera l'entrée sonore pour permettre à l'utilisateur de réaccorder son instrument directement depuis le logiciel.

2.7 Liaison avec l'application web

Afin de communiquer avec l'application web, le client sera doté d'un module réseau qui enverra et recevra des requêtes http. Les partitions seront envoyées et reçues en format JSON afin de faciliter ces échanges.
Le JSON aura format semblable à ceci :

```
{
    "Name" : "string",
    "Tempo" : 0,
    "private" : true/false
    "tunes" : [{
        "volume" : 0-1,
        "nb_notes" : 0,
        "notes" : [{
            "name" : "do-la-...",
            "frequence" : 0
        }]
    }]
}
```

3 Application Web

3.1 Maquettes d'écran

Les differentes maquettes du site permettent d'améliorer la vision des besoins de l'application. Ces visuels ne sont pas définitifs et sont simplement là pour nous aider à mieux structurer l'application et organiser les objectifs nécessaires à la réalisation de celle-ci.

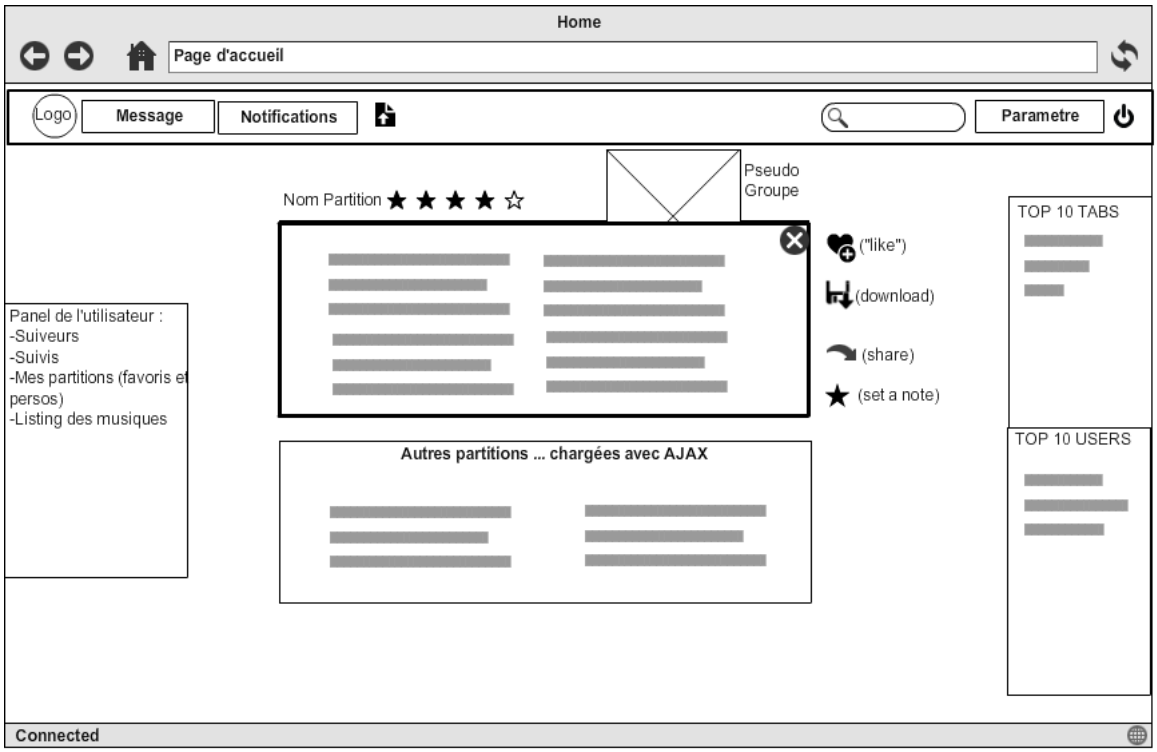


FIGURE 8 – Maquette de la page principale du site

La barre du haut est une barre de menu qui sera disponible sur toute les pages du site, elle n'est pas représentée sur toutes les maquettes par soucis de clarté. En effet, le principal sujet des maquettes suivantes n'est pas la présence de la barre de menu mais le contenu de celles-ci

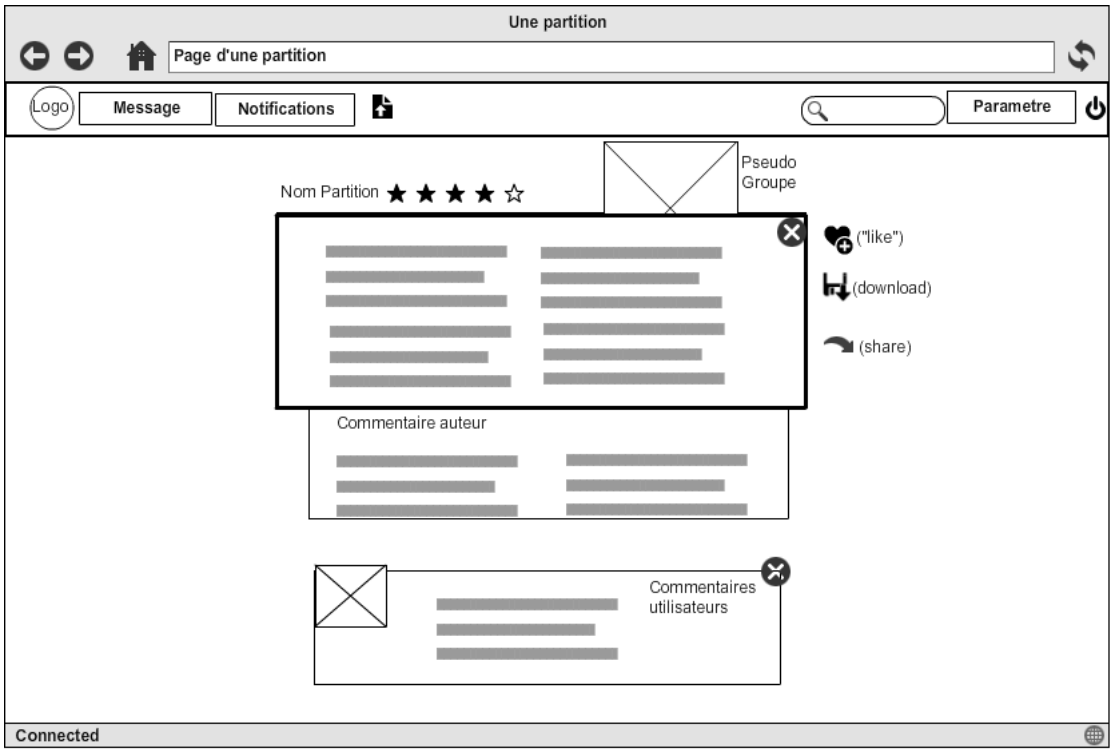


FIGURE 9 – Page principale d'une partition de musique
Cette page permettra de visionner une partition. De plus, si l'utilisateur possède les droits nécessaires, il pourra la supprimer ou la modifier. Enfin, les utilisateurs pourront poster des commentaires sur la partition et partager cette dernière sur différents réseaux sociaux.

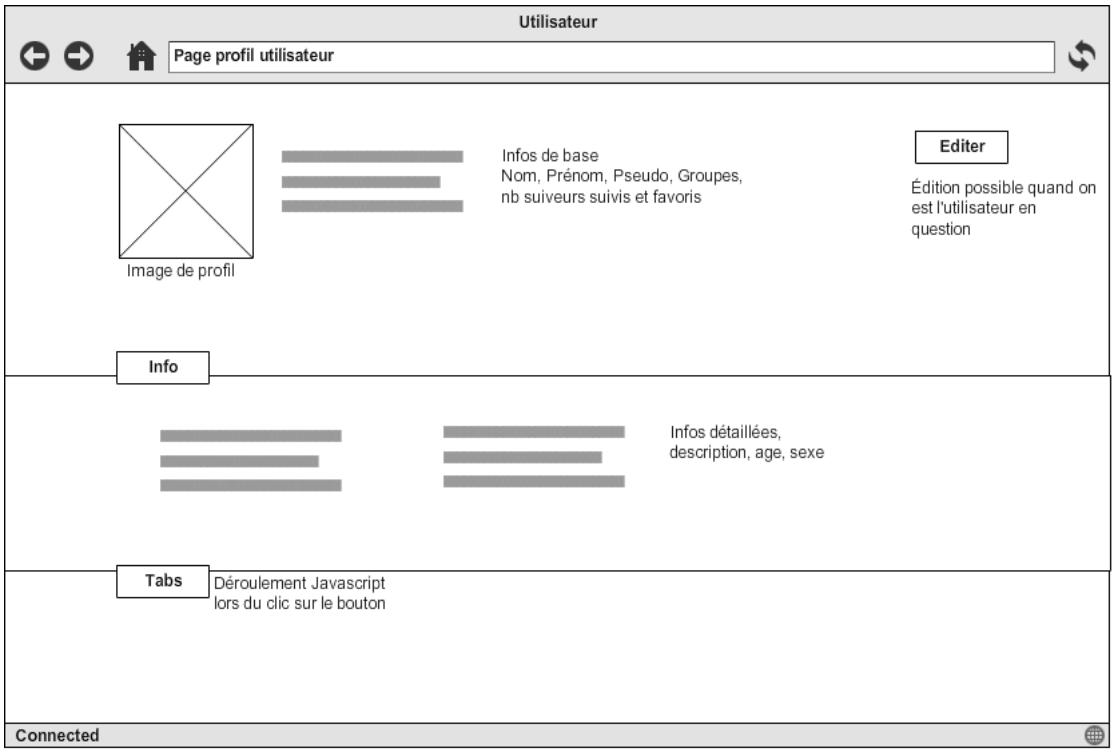


FIGURE 10 – Page de profil utilisateur
Cette page permettra à l'utilisateur concerné de modifier ses informations s'il le souhaite.
Les autres utilisateurs verront sur sa page de profil les informations rendues visibles par l'utilisateur en question.

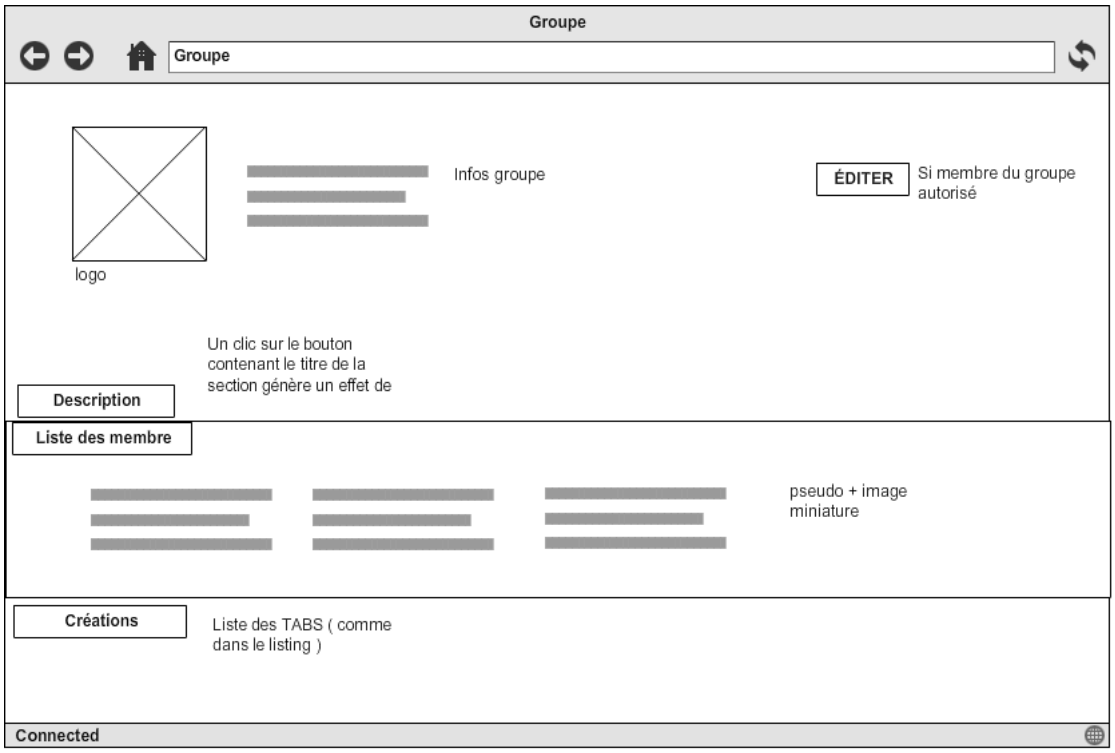


FIGURE 11 – Page d'un groupe de musique
Cette page permettra de modifier le groupe si l'utilisateur en possède les droits. Cette page contiendra la liste des membres du groupe ainsi que les différentes partitions liées au groupe concerné.

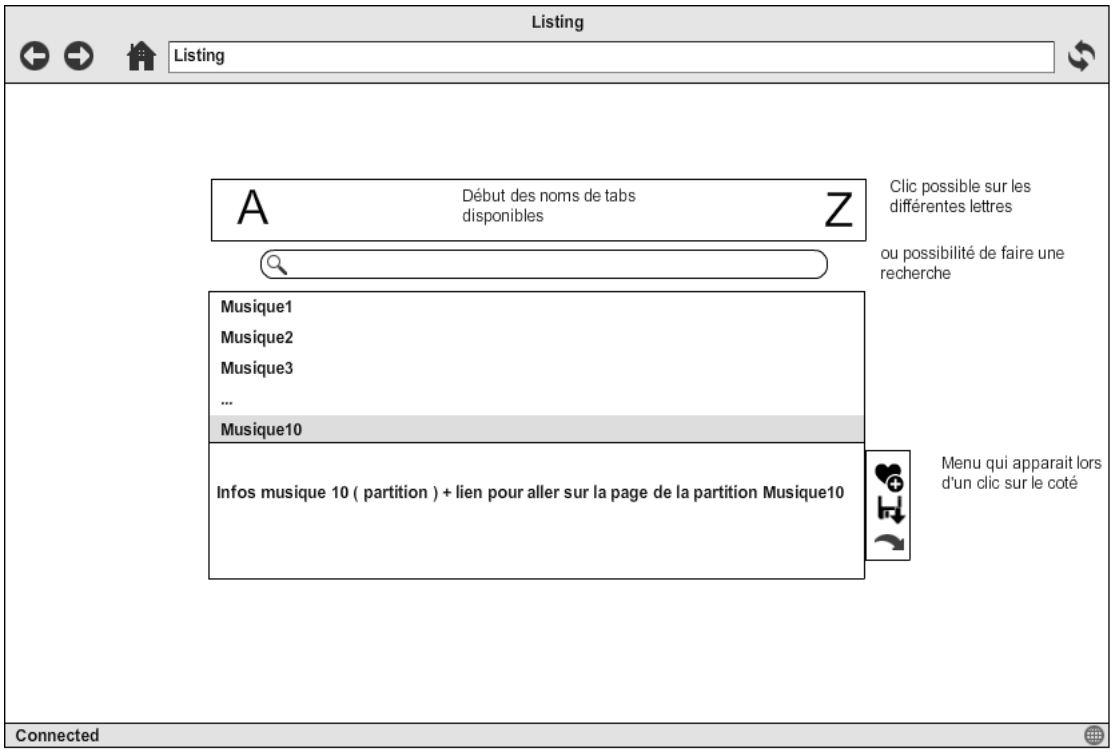
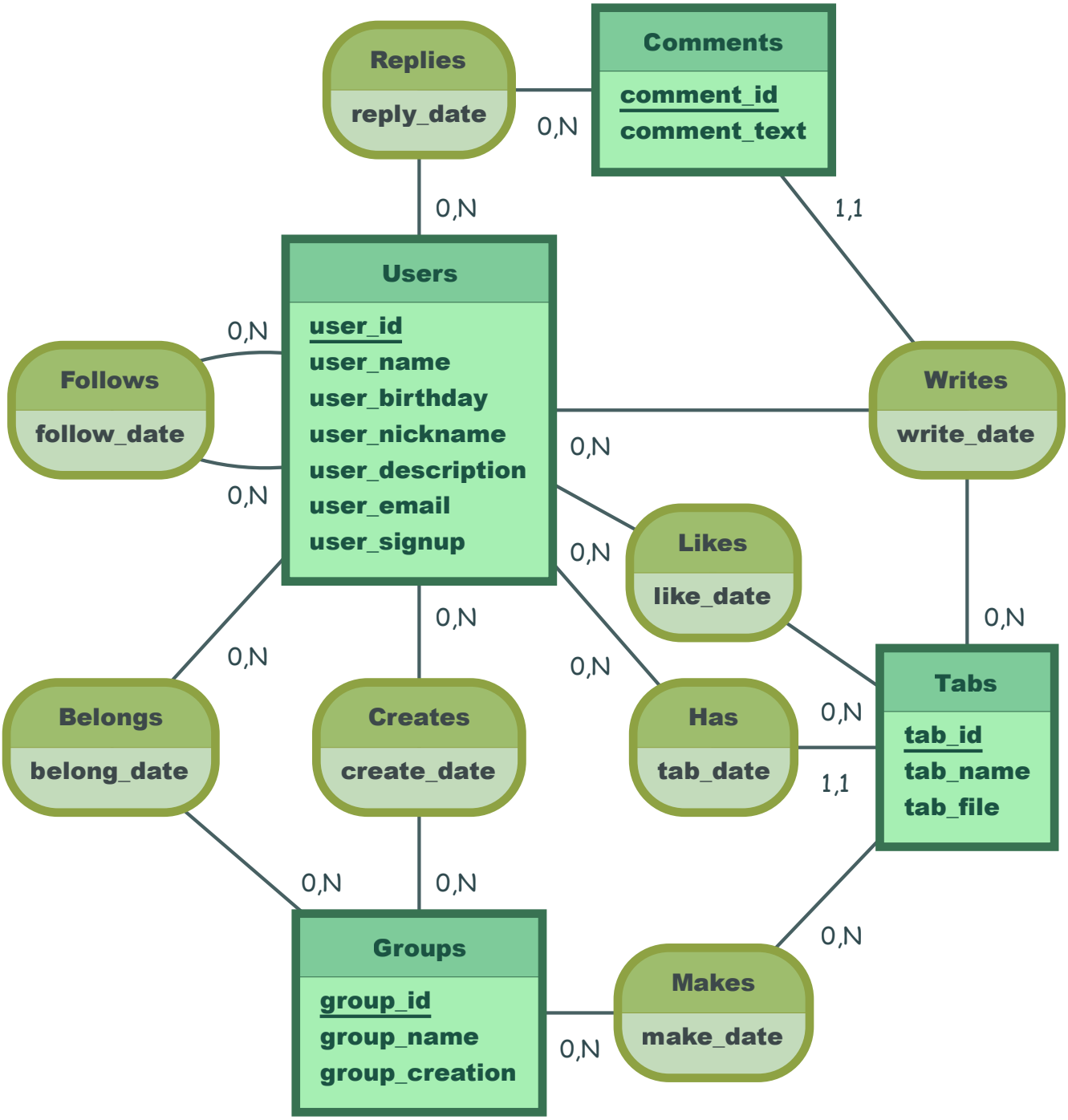


FIGURE 12 – Liste de toutes les partitions
Le listing des partitions permettra une recherche plus approfondie qu'avec la recherche disponible dans la barre de menu (la barre du haut).

3.2 Modèle Merise MCD



3.3 Modèle Merise MLD

Replies (#user_id, #comment_id, reply_date)
Comments (comment_id, comment_text, #tab_id, #user_id, write_date)
Follows (#user_id, #user_id, follow_date)
Users (user_id, user_name, user_birthday, user_nickname, user_description, user_email, user_signup)
Likes (#tab_id, #user_id, like_date)
Belongs (#user_id, #group_id, belong_date)
Creates (#group_id, #user_id, create_date)
Tabs (tab_id, tab_name, tab_file, #user_id, tab_date)
Groups (group_id, group_name, group_creation)
Makes (#group_id, #tab_id, make_date)

3.4 Structure de l'application

La structure du site suivra le modèle MVC.
Des tests unitaires seront utilisés pour s'assurer du bon fonctionnement de l'application.
Le site sera à l'aide d'un framework MVC Javascript (Express) pour gérer la partie Serveur et un autre framework complémentaire qui sera utilisé pour les vues et les controllers en particulier (AngularJS).
Utiliser Javascript pour coder toute l'application web aussi bien du côté client que du côté serveur nous permettra d'avoir une première approche avec cette technologie qui devient de plus en plus demandée aujourd'hui. Ce qui rend donc ce projet très enrichissant pour notre avenir professionel.

Le site sera donc divisé en trois principales parties :
-Views
-Controllers
-Models
Plus une partie qui contiendra les routes (ie les chemins d'accès aux différents controllers)

Views :
-home => représentera l'accueuil du site -compte => Profil utilisateur
-groupe => Profil du groupe
-listing => Listing des différents partitions avec option de recherche
-topbar => barre du haut qui sera incluse sur toutes les pages
-tabpage => page principale d'une partition

Controllers :
Les controllers seront utilisés pour faire le lien entre le modèle et les différentes vues. De plus ils permettront de gérer les fonctionnalités spécifiques au site.
-home.js => controller général
-profil.js => controller du compte qui gère le profil utilisateur
-groupe.js => controller qui gère les groupes utilisateurs
-listing.js => controller qui gère la liste des partitions ainsi que la fonction recherche dans celles-ci
-topbare.js => permet de gérer la barre de menu présente sur le haut du site
-partition.js => permet de gérer les fonctionnalités d'une partition en particulier
-partition-page.js => controller qui gère la page d'une partition

Les modèles suivent l'architecture Active Record.
Ils seront donc liés à la base de données (cf MCD et MLD).
Une table équivaut donc à un module JavaScript.
Ces modules contiendront toutes les méthodes nécessaires au traitement et à la modification du modèle (findById,insert,delete,update ...) et suivront donc l'approche vue en cours.

Routes :
Chaque lien est une route qui peut contenir une action et qui actionne une méthode du controller lié à la route.

Une route est donc un chemin représenté par un URL.
Ces routes sont donc le coeur du fonctionnement de l'application Web.

Table des matières