Université de Montpellier
L2 Informatique

Bracket Creator

Un outil d'édition de tournois en ligne

Rapport de T.E.R.
Projet Informatique HLIN405

Encadrant :
M. David Delahaye

Étudiants :
M. Arda Uzan
M. Quentin Baiget
M. Benoît Huftier
M. Johann Rosain

Code source : https://gitlab.com/Ephesiel/projet-hlin405/

Table des matières

Introduction

Dans le cadre du projet informatique du second semestre de deuxième année de licence à l'Université de Montpellier, nous avons été amenés à développer un site web qui permet la création et la gestion de tournois.

Notre groupe était au départ composé de quatre personnes : Arnaud Peler, Quentin Baiget, Benoît Huftier et Johann Rosain. Suite à des problèmes personnels, Arnaud Peler a malheureusement quitté le groupe, avant de faire une quelconque contribution au projet. Nous avons ensuite, vers le milieu du projet, été rejoints par Arda Uzan. Le projet a été réalisé sur une période de 3 mois, du 8 février au 7 mai 2021, et notre groupe a été encadré par M. David Delahaye.

Notre groupe étant composé de deux étudiants ayant déjà développé un site professionnel, nous nous sommes fixés pour objectif une chose : d'avoir un code propre, qu'il est facile de maintenir, d'utiliser, et de réutiliser.

Nous avons ainsi défini une liste d'éléments nécessaires à l'élaboration d'un site structuré :

- Un outil de travail collaboratif avec du code qui fonctionne sur toutes les plateformes.
- Des règles de programmation, pour que le style du code soit uniforme.
- Une structure flexible, qui permet une factorisation maximale des différentes parties à programmer.
- Un système de *tests*, pour s'assurer que ce que nous faisons fonctionne correctement et continue de fonctionner dans le futur.
- Des *code reviews* faites par nos pairs, qui permettent d'éviter des *bugs* flagrants, et qui s'assurent que tout le monde est à jour dans le projet.

Nous allons donc tout d'abord parler de l'organisation du groupe et de la division des tâches, pour ensuite s'arrêter sur l'architecture de notre site et sa base de données associée, et enfin aborder les particularités techniques, et conclure sur l'expérience que ce projet nous a apporté.

Chapitre 1

Organisation du groupe

1.1 Outil recherché

Il y a 3 points parmi ceux définis précédemment qui entrent dans le domaine de l'organisation, et plus précisément, dans le domaine du développement collaboratif. Notre but était donc de trouver une solution web qui permettait :

1. **Le partage de code**, pour que tout le monde puisse utiliser la dernière version du site, et puisse travailler de manière agréable.
2. **L'attribution de tâches**, pour savoir en détail ce que chaque personne doit faire, et avoir un historique de ce que chacun a fait.
3. **D'avoir un espace « wiki »**, afin de noter les règles importantes à respecter, et décrire l'utilisation de l'architecture développée.
4. **De pouvoir faire des *Code Reviews***, facilement et rapidement.

Après s'être renseignés sur les solutions existantes, nous avons choisi l'outil *GitLab*[1], sur un serveur hébergé par les développeurs de cette solution (car celui de la faculté était souvent hors service, ce qui nous empêchait d'avancer).

Nous avons ainsi pu compartimenter le travail à faire, et défini des règles simples pour l'utilisation de cet outil.

1.2 Notre utilisation de *GitLab*

Pour garder une organisation du projet structurée, nous nous sommes servis des modules proposés par l'outil, en mettant en place :

— Un système d'attribution de tâches, en utilisant des *issues* avec le *label*[2] « Nouvelle fonctionnalité », qui définit chaque tâche à faire et la date de rendu, et qui assigne la tâche à un membre du groupe.

1. GitLab : https://gitlab.com/Ephesiel/projet-hlin405
2. Label : Étiquette en français, cela permet de catégoriser les l'*issues*.

— Plusieurs pages de *wiki*, qui référencent les règles d'écriture des différents langages, l'utilisation de la structure *php* pour créer une page, et quelques guides, notamment pour l'utilisation des commandes de base de *git*.
— Un système de *Code Reviews* en utilisant des *issues* avec le *label* « Code Review », où un *template* de description de la *review* est fourni et le développeur n'a qu'à compléter.
— Une *pipeline* qui exécute automatiquement les *tests php*[3] pour vérifier que rien n'a été cassé sur un *commit*[4].

1.3 Répartition des tâches

Les tâches ont rapidement été découpées pour que tout le monde ait du travail à faire. Nous avons fractionné le travail en 3 catégories :

— HTML / CSS : Quentin BAIGET
— Javascript : Arda UZAN, Benoît HUFTIER, Johann ROSAIN
— PHP :

1. Benoît HUFTIER, qui a conçu et développé le système de validation de formulaire et de contraintes dans le module *core*, ainsi que plusieurs pages du site.
2. Johann ROSAIN, qui a conçu et développé l'architecture du site (notamment les deux systèmes de *renderer*), et qui a aussi travaillé sur la plupart des validations de formulaire.

Ainsi, tout le monde pouvait travailler sur une fonctionnalité différente en même temps, grâce au *git* bien organisé et à la création d'une *branche* par fonctionnalité à programmer.

Au début du projet, nous avions l'intention de travailler pour sortir une version 1 en un peu plus d'un mois, puis une version 2 après le temps qu'il nous restait avant de clôturer le projet. Néanmoins, petit à petit, au cours du projet, nous nous sommes aperçus que nous n'aurions pas le temps de faire cela, car nous avons privilégié un code propre, relu, et testé plutôt que d'aller vite et avoir de nombreux bugs.

Finalement, notre développement n'a donc pas du tout suivi le planning posé au début du projet, et nous nous sommes retrouvés avec le planning décrit dans l'annexe A.

3. Une section sera dédiée aux tests dans la suite du document
4. Commit : Procédé de git qui permet de valider les modifications faites.

Chapitre 2

L'architecture

Comme évoqué précédemment, deux membres de l'équipe ont déjà eut affaire à des projets professionnels. Ils connaissent bien les problèmes liés à la mauvaise gestion d'un projet, en particulier le fait de ne pas avoir une architecture flexible.

C'est ainsi que la première semaine du projet a été passée à concevoir et à développer une architecture qui soit :

— Flexible
— Facile à utiliser
— Facile à modifier

Nous avons décidé de regrouper tout ce qui n'impacte pas directement le site dans un module *core*, qui contient alors deux systèmes pensés pour suivre tous les points précédents, pour finalement arriver au schéma de conception de la figure 1.

2.1 Le système de rendu de page

Nous avons donc conçu un système avec deux fonctionnalités principales. Il permet de **gérer les sections affichées** de la page et de **configurer l'agencement de la page**.

2.1.1 La gestion des sections

Ce système permet de contrôler l'affichage de certaines sections de deux manières différentes :

1. Grâce à des *permissions*, qui est en fait un système de *flags*[1]. Nous associons ces permissions à des classes HTML de la page, et selon les permissions de l'utilisateur actuel, ces sections sont supprimées s'il n'a pas le droit de les voir.
2. Grâce à des *prédicats*, avec un système assez similaire à celui des permissions. Un certain prédicat est associé à une classe HTML, et s'il est faux, alors cette section est supprimée.

1. Flag : c'est un ensemble de *bits* qui fournissent une information selon leur état.

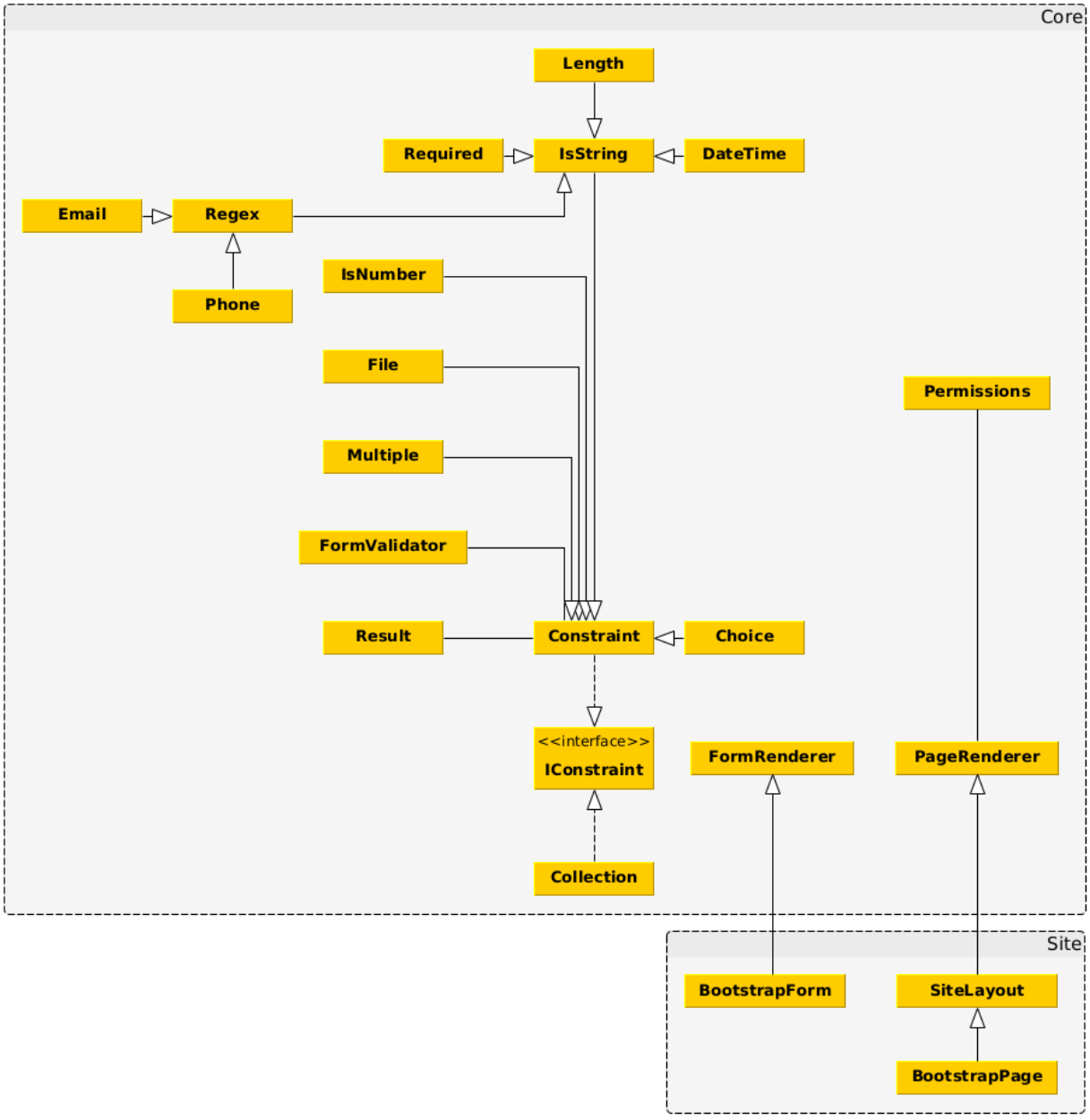


FIGURE 1 – Conception simplifiée de l'architecture du site.

De cette manière, le développeur *frontend*, donc HTML / CSS, ne s'occupe que de son code à lui. Il n'y a pas (ou très peu) de PHP dans les pages HTML, et cela évite d'avoir une page incompréhensible et dure à maintenir. Il n'y a que le strict nécessaire, ce qui fait que refaire la page ou bien en changer une partie n'est pas difficile.

2.1.2 L'agencement d'une page

Nous nous sommes également dit, qu'outre le système précédent, mettre en place un moyen efficace d'avoir toujours le même agencement de pages est très pratique pour éviter la répétition de code, ainsi que pour éviter de devoir changer les sections qui structurent notre site dans les 15 pages qui le composent.

C'est ici, qu'encore une fois, la classe *PageRenderer* entre en jeu. Cette classe est *générique*, c'est une classe-mère qui gère tout le système qu'on ne veut pas avoir à refaire dans ses classes-filles. Elle fournit la fonction *render*, qui appelle une méthode qui construit un objet *DOMDocument*, et l'écrit sur la page.

C'est aux classes-filles d'implémenter cette méthode, et c'est le système qui nous permet d'agencer la page comme nous l'entendons. Par exemple, le modèle utilisé pour toutes les pages de notre site, appelé *BootstrapPage* (car il inclut la feuille css et le script js bootstrap), affiche les éléments suivants :

- La balise *head*, ainsi que son contenu
- Le *corps* de la page, avec :
  - Le *header* du site, configuré au préalable
  - Une section avec la classe *container*
  - Les messages de succès, s'il y en a
  - Le corps de la page en lui même
  - Le footer
  - L'inclusion des fichiers javascript

De plus, les sections restreintes sur le *header* et le *footer* peuvent aussi être gérées dans cette classe-fille, ce qui nous permet également de factoriser la gestion des accès.

2.2 La construction d'une page

En plus de ce système de rendu, nous avons fait en sorte qu'il soit facile à utiliser. Pour ce faire, nous nous sommes inspirés de *wordpress*[2], et à la création d'une page du site, il faut inclure un fichier : *site-header.php*. Ce fichier sert à charger tout ce dont nous avons besoin :

- Définition de constantes du site lues depuis le fichier *env.cfg*.
- Inclusion de tous les fichiers de fonction.
- Inclusion de l'autoloader.
- Installation et mise à jour de la base de données.
- Définition des constantes de l'utilisateur.

2. Wordpress : https://github.com/WordPress/WordPress/

Nous savons que tout le monde n'est pas sur le même environnement. Nous avons envisagé l'installation pour tous les membres de l'équipe d'un conteneur *Docker*, qui règlerait ce problème. Seulement, nous nous sommes dit que ce serait peut-être beaucoup demander à chacun de faire ceci, et comme personne n'avait déjà travaillé avec cet outil, les problèmes rencontrés auraient pris plus de temps à résoudre que d'autres problèmes de compatibilité.

Ainsi, chaque développeur peut définir ses propres paramètres depuis le fichier *config.php*. Notamment, il définit le compte administrateur, la base de données à laquelle se connecter, ainsi que l'ajout ou non des *placeholders* dans la base de données.

Tout ce système permet de créer une page, minimale, en quelques lignes :

```php
<?php

require_once __DIR__ . '/site-header.php';

// Création de la page
$page = new PH\Templates\BootstrapPage();

// Ajout de notre fichier HTML
$page->setBody(ph_get_body('accueil.php'));

// Ajout d'un titre (optionnel, mais conseillé)
$page->setTitle("Accueil de mon super site !");

// On rend la page !
$page->render();
```

Chapitre 3

La base de données

La base de données, tout comme le planning de développement, est *vivante*. En effet, elle née au début du projet, où les concepteurs essaient de faire au mieux pour tout prévoir. Cependant elle évolue tout au long du projet, pour répondre aux besoins grandissants des développeurs.

La structure globale reste inchangée, mais de nombreuse petites modifications sont effectuées sur les différentes tables pour mieux répondre aux besoins réels des développeurs.

Nous allons ici décrire le schéma final, avec la version simplifiée consultable sur la figure 2, et une version complète sur l'annexe B.

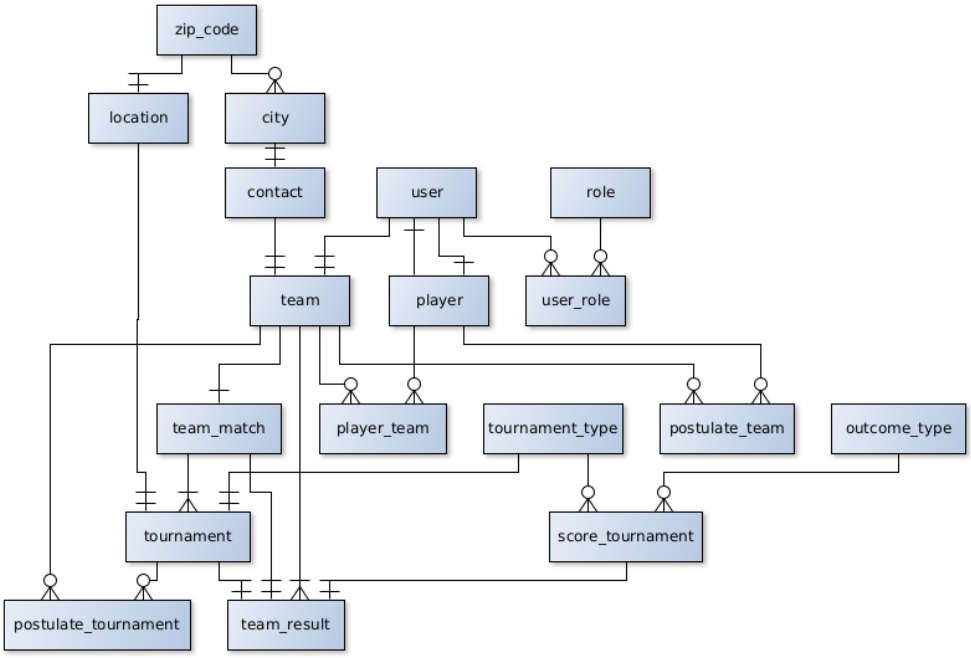


FIGURE 2 – Schéma simplifié de la base de données.

3.1 Les utilisateurs

Nous allons parler dans cette première partie des utilisateurs. C'était la première chose à mettre en place dans le site pour avoir accès aux différentes pages. Il y a trois tables intéressantes :

— *user*, qui enregistre les données d'un utilisateur.
— *role*, qui est une liste des rôles disponibles dans le site.
— *user_role*, qui est une liste de rôles par utilisateur.

Nous avons procédé de cette façon pour une raison principale : un utilisateur *peut avoir* plusieurs rôles. En effet, un utilisateur peut être *en même temps* un joueur et un gestionnaire de tournois.

Ainsi, ces 3 tables permettent de mettre facilement plusieurs rôles à un utilisateur. Elles permettent aussi d'ajouter autant de rôles que nous le souhaiterions dans le futur, et qu'ils pourront toujours être assignés facilement. Changer le nom d'un rôle peut-être fait tout aussi facilement. C'est une structure facile à utiliser et flexible, qui permet de faire des modifications dans le futur sans avoir à repenser la base de données.

3.2 Les équipes

Les équipes sont assez liées aux utilisateurs. En effet, dans notre système, un joueur doit être dans une équipe pour pouvoir jouer un tournoi. Ainsi, c'est la seconde étape après les utilisateurs pour pouvoir créer un tournoi. Ici encore, il y a 3 tables en particulier qui nous intéressent :

— *team*, qui enregistre les informations d'une équipe.
— *postulate_team*, qui gère les postulats à une équipe.
— *player_team*, qui enregistre quel joueur est dans quelle équipe.

Un joueur crée une équipe, il est donc son capitaine. Ensuite, d'autres joueurs peuvent postuler dans son équipe, et c'est à lui de choisir de les accepter ou de les refuser. De plus, aucune donnée de *postulate_team* n'est effacée, pour que nous puissions garder un historique des postulats. Ainsi, il y a 3 états possibles :

— *accepted*, qui signifie que le joueur a été accepté dans l'équipe. Dans ce cas, les informations vont aussi être renseignées dans *player_team*.
— *pending*, qui signifie que le postulat n'a pas encore été accepté ou refusé, le joueur est en attente.
— *refused*, qui signifie que le joueur a été refusé. Il ne peut ainsi plus postuler dans cette équipe.

3.3 Tournois

3.3.1 Inscription

Nous avons utilisé un système de postulats similaire pour les inscriptions. Cependant, ici, il n'y a pas de 3ème table pour savoir quelle équipe est inscrite. Il suffit simplement de regarder quelles équipes ont été acceptées dans *postulate_ tournament*.

3.3.2 Type

Il y a plusieurs types de tournois renseignés dans la base de données : des coupes, des championnats, et des coupes avec poules. Ainsi, pour chaque type de tournoi, nous devons pouvoir assigner un score différent selon le résultat. C'est là que 3 tables entrent en jeu :

- *tournament_ type*, pour le type de tournoi.
- *outcome_ type*, pour les différents résultats possibles. Pour le moment, seuls « défaite », « victoire » et « nul » existent.
- *score_ tournament*, qui répertorie le score selon le type de tournoi et le résultat d'un match. Par exemple, en coupe, il n'y a que victoire et défaite. Impossible d'avoir un match nul. Donc par exemple, 1 point pour une victoire, et 0 pour une défaite. Cependant, en championnat, il est possible d'avoir un match nul. Il faut donc un système de points correspondant. Cette table existe pour configurer ce système.

3.3.3 Matches

Les matches sont un peu délicats à représenter. Il faut d'un côté connaître les deux équipes et le tournoi, mais d'un autre, il faut aussi savoir quand le match a été joué, pour pouvoir faire un arbre de tournoi correct. Ainsi, nous avons décidé de faire un arbre binaire *inversé*, c'est à dire, qu'au lieu de partir de la racine et d'avoir des enfants, tous les noeuds, à l'exception des feuilles, référencent leurs parents.

De cette façon, nous partons des feuilles pour remonter petit à petit vers la racine de l'arbre, qui sera la finale. Cela permet aussi de gérer un nombre d'équipes aléatoire, et pas seulement une puissance de 2.

3.4 Le lieu

Pour cette partie, nous avons choisi de laisser les utilisateurs remplir notre base de données pour nous. Nous avons supposé que les tournois se déroulaient seulement en France pour cette première version du site. Il y a 3 tables importantes :

- *city*, avec juste le nom d'une ville.
- *zip_ code*, qui prend un code postal et une ville.
- *location*, qui récupère une adresse et l'associe à un code postal.

Une fois que les utilisateurs entrent une ville et un code postal, ces informations seront ensuite proposées aux autres utilisateurs lorsqu'ils veulent entrer un nouveau lieu.

Chapitre 4

Les particularités techniques

4.1 PHP

Nous avons fait en sorte de factoriser au maximum notre code PHP. En effet, nous voulions éviter de faire la même chose sur plusieurs pages différentes, chose qui s'avère compliquée sur un site, notamment au niveau des formulaires.

C'est pourquoi nous avons développé deux systèmes pour éviter de répéter les mêmes lignes pour chaque formulaire que nous créions.

4.1.1 Les validations

Le premier système est celui des validations. En faisant des recherches, nous sommes tombés sur le système de validations de *symfony*[1]. Nous nous sommes inspirés de ce modèle pour développer notre propre système. Il permet de créer des contraintes sur chaque champ d'un formulaire puis de les tester.

Pour chaque contrainte, un objet *Result* est créé. Il possède deux propriétés. La première est la validité du champ. La seconde concerne les messages d'erreurs sur le champ. Si le champ est valide, cette dernière recevra un tableau vide.

C'est un système facilement extensible, attribut qu'il nous fallait absolument. En effet, un ensemble de contraintes a été développé au début de ce système. Cependant, nous n'avons pas pensé à certaines contraintes, et donc nous avons dû les ajouter par la suite. C'était très facile, autant à ajouter qu'à utiliser. La flexibilité de ce système et sa facilité d'utilisation nous a permis de faire des vérifications extensibles sur les différents formulaires très facilement.

4.1.2 Le rendu de formulaires

Suite au développement de ce premier système, nous avons eu l'idée de factoriser les erreurs des formulaires. En effet, comme dit précédemment, nous voulions éviter au maximum d'avoir du PHP dans nos pages HTML.

1. Validator Symfony : https://symfony.com/doc/current/validation.html

De plus, nous avions maintenant un système « normalisé » de résultats, le mieux serait donc aussi de faire un système « normalisé » de formulaires.

C'est ainsi qu'est née l'idée de la classe *FormRenderer*. Grâce à cette classe, le programmeur HTML ne s'occupe que du HTML. Il ne s'occupe pas des erreurs ou des résultats valides possibles. De plus, les erreurs sont gérées automatiquement par cette nouvelle classe. Elle facilite ainsi la vie de tout le monde.

Cette classe utilise une structure assez similaire à celle de *PageRenderer* :

— Calcul du *DOMDocument*.

— Rendu des erreurs, ou des champs valides, par les classes-filles.

— Affichage du formulaire sur la page.

Les données entrées par l'utilisateur sont sauvegardées, pour être replacées ensuite dans le formulaire, afin de savoir où il a des erreurs, et pourquoi c'est une erreur. Les champs valides sont aussi gardés, car c'est plus simple de ne pas les ressaisir.

De plus, un système de stockage des fichiers temporaires a été créé. Si l'utilisateur téléverse un fichier, et qu'il y a des erreurs dans le formulaire, les fichiers téléversés vont être sauvegardés dans un répertoire temporaire, et leurs noms seront écrits, par défaut, dans les champs associés. Cela permet d'éviter d'avoir à re-téléverser un fichier un peu lourd.

4.1.3 Les rôles

En plus de ces deux systèmes, nous avons aussi une classe de rôles, qui permet à un utilisateur d'avoir un ou plusieurs rôles.

Pour ce faire, nous avons utilisé un système de *flags* : un drapeau (bit) est levé pour chaque rôle que l'utilisateur possède. Par exemple, un administrateur aura les drapeaux suivants :

0	0	0	1

Si un utilisateur est joueur et administrateur, ses drapeaux ressembleront plutôt à ceci :

0	1	0	1

Bien sûr, c'est la représentation binaire des nombres qui nous intéresse ici. En PHP, cela sera stocké dans des entiers. Un administrateur aura alors le rôle qui est égal à 1, et un administrateur et joueur aura le rôle égal à 5.

4.2 Javascript

Nous avons réalisé beaucoup moins de *javascript* que de PHP car, dans le planning de développement initial, notre objectif était d'ajouter du javascript sur la version 2 du site. Cependant, nous avons tout de même développé deux systèmes assez complets.

4.2.1 Les filtres de recherche

Dans notre site, il y a plusieurs tableaux qui représentent des données. Notamment, nous avons le tableau qui liste les tournois, ainsi que celui qui liste les équipes. À défaut d'opter pour une barre de recherche classique, qui ne filtre qu'un seul champ, nous nous sommes dirigés sur un système de conditions.

Chaque condition correspond à une colonne du tableau de données. Dans la plupart des cas, une colonne du tableau correspond à une colonne de la table associée dans la base de données.

De plus, les conditions sont connectés logiquement par un « ET », ce qui permet d'affiner facilement les recherches.

Ce système est facilement utilisable sur plusieurs pages différentes. Il n'a besoin que d'un champ de type *select* dans la page HTML qui présente toutes les clés possibles, puis de configurer tous les champs d'entrée en *javascript*.

Lorsque l'utilisateur clique sur le bouton de soumission du formulaire, tous les filtres qu'il a ajoutés sont alors envoyés sur la page d'*action* définie du formulaire. Les champs qui n'ont pas été ajoutés ne seront pas envoyés.

4.2.2 L'ajax

Nous avons également fait un petit peu d'*ajax*[2], afin de faciliter la vie aux utilisateurs. Nous avons une tâche assez répétitive dans le site : accepter ou refuser des postulats.

Il peut y avoir une somme assez élevée de postulats, que ce soit pour un capitaine d'équipe ou bien pour un gestionnaire de tournois. De plus, pour gérer ces postulats, il faut ouvrir un modal. Ainsi, ce n'était pas vraiment *user-friendly* de devoir recharger la page à chaque postulat accepté ou refusé.

Par conséquent, nous avons développé un ensemble de fonctions pour chaque page où nous voulions ce système.

2. AJAX : Asynchronous JavaScript and XML

Chapitre 5

Conclusion

5.1 Travail réalisé

Pour résumé, notre site possède les fonctionnalités suivantes :

- Création de compte et permissions : joueur, gestionnaire de tournoi, administrateur.
- Création de tournoi de type coupe par l'administrateur.
- Gestion d'un ou plusieurs tournois par un gestionnaire.
- Création des équipes et inscription sur un ou plusieurs tournois.
- Affichage public des tournois et des matches.
- Gestion des rôles des différents comptes par l'administrateur.

5.2 Les difficultés organisationnelles

Durant ce projet, nous n'avons pas spécialement rencontré de difficultés techniques. Les difficultés principales étaient sur le plan de l'organisation. En effet, malgré un *git* méthodique et structuré, nous nous sommes aperçus d'une chose importante : tout le monde n'a pas le même niveau en programmation.

C'est une observation qui paraît évidente. Seulement, cela nous a frappé de plein fouet. En effet, comme le niveau des membres du groupe est hétérogène, nous n'avançons pas tous à la même allure. Ce qui est apparent pour certains, ne l'est pas pour d'autres.

Ainsi, cela nous a fait nous repencher sur le planning de développement, qui était initialement très ambitieux.

5.3 Perspectives

Le site est actuellement fonctionnel, et il pourrait être utilisé tel quel. Cependant, il y a plusieurs choses qui n'ont pas encore été faites et qui pourraient améliorer grandement l'expérience d'un utilisateur :

— La gestion de différent types de tournois (pour le moment, seulement les coupes sont gérées).
— Le moyen de réinitialiser le mot de passe d'un utilisateur grâce à un mail dédié.
— L'activation d'un compte grâce à un mail dédié.
— Un système de seeding pour que les meilleures équipes se rencontrent le plus tard possible dans la coupe.

5.4 Conclusions tirées

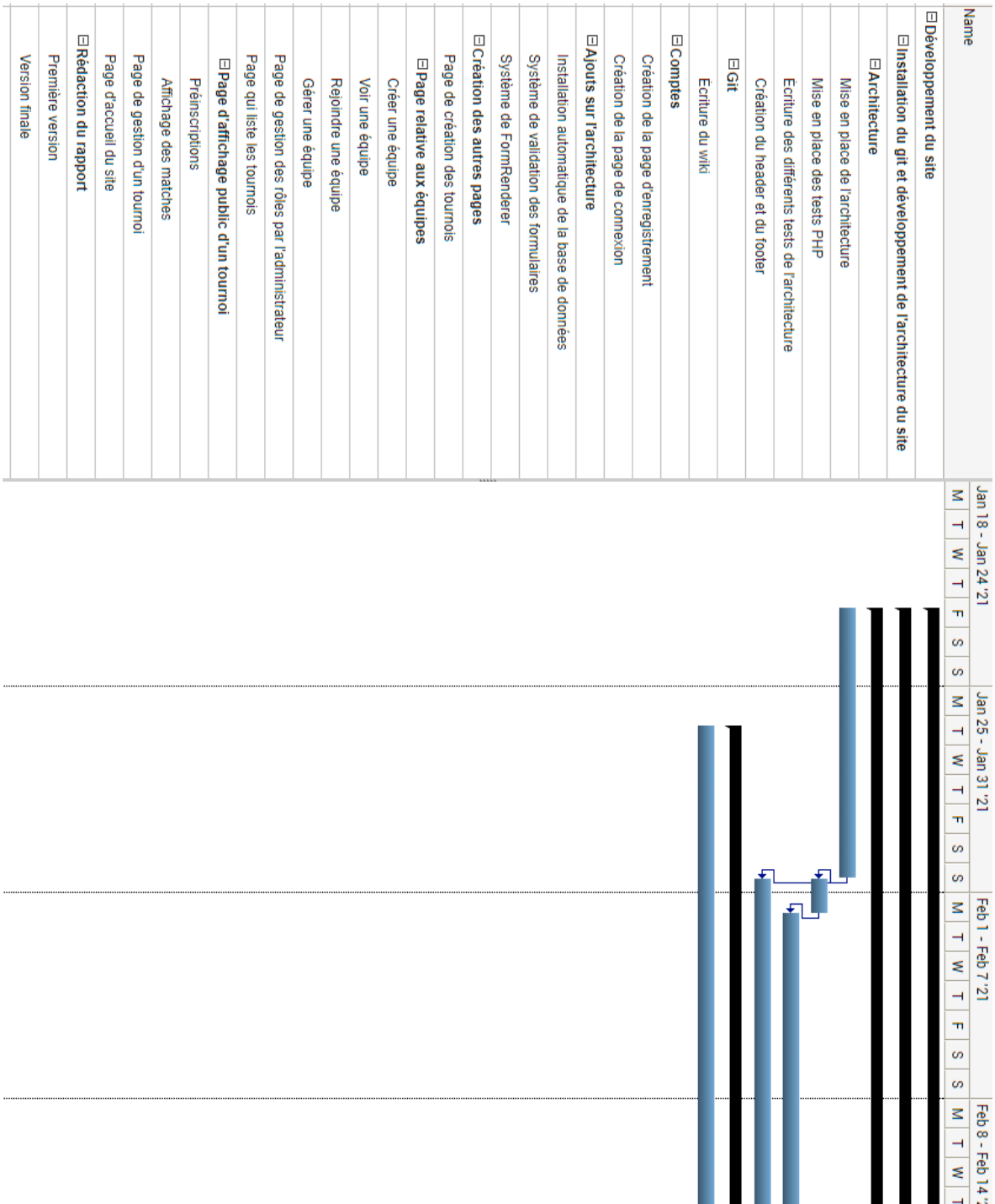
Ce projet nous a appris quelque chose d'important sur le travail en équipe. Nous avons compris qu'avoir une méthode de travail structurée et méticuleuse ne suffit pas à la réussite d'un projet, et qu'il faut aussi *savoir* travailler en équipe. Communiquer, collaborer, et s'entraider, plutôt que de faire chacun sa chose dans son coin et espérer que tout marche quand nous mettons en commun.

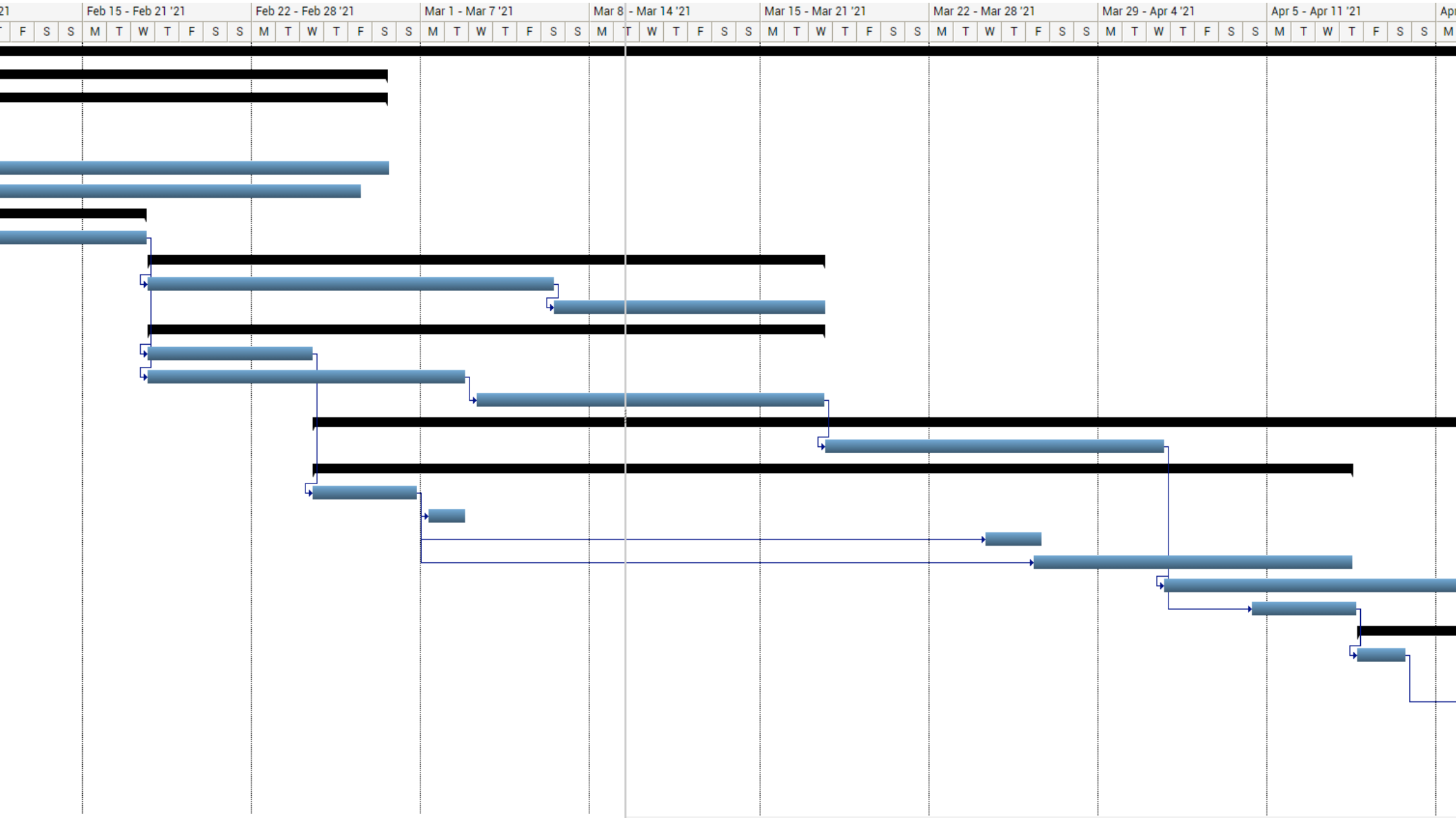
Pour conclure, ce projet nous a apporté de nouvelles perspectives sur le travail de groupe, et a ainsi grandement enrichi notre expérience.

Annexes

Annexe A

Planning de développement

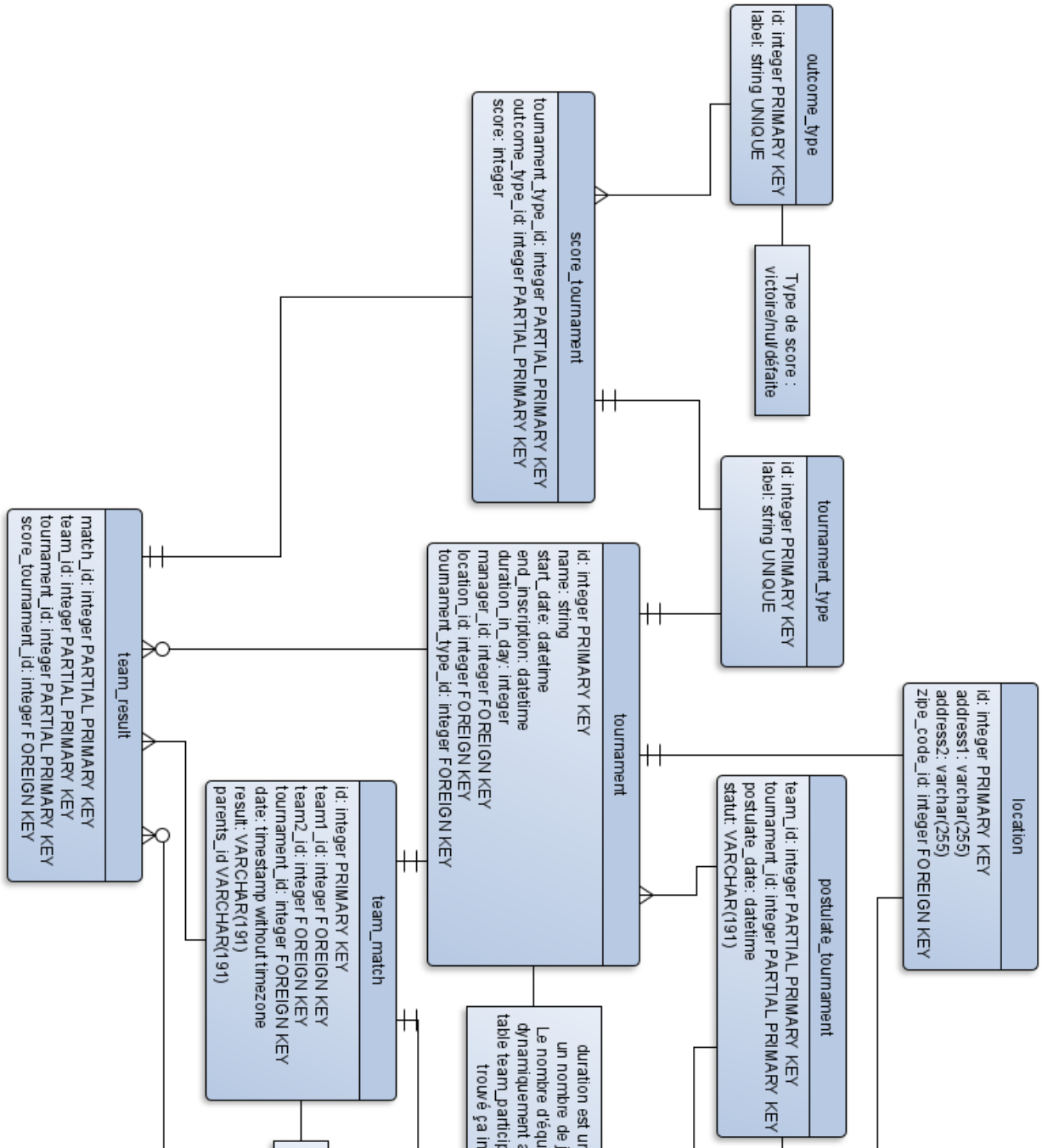


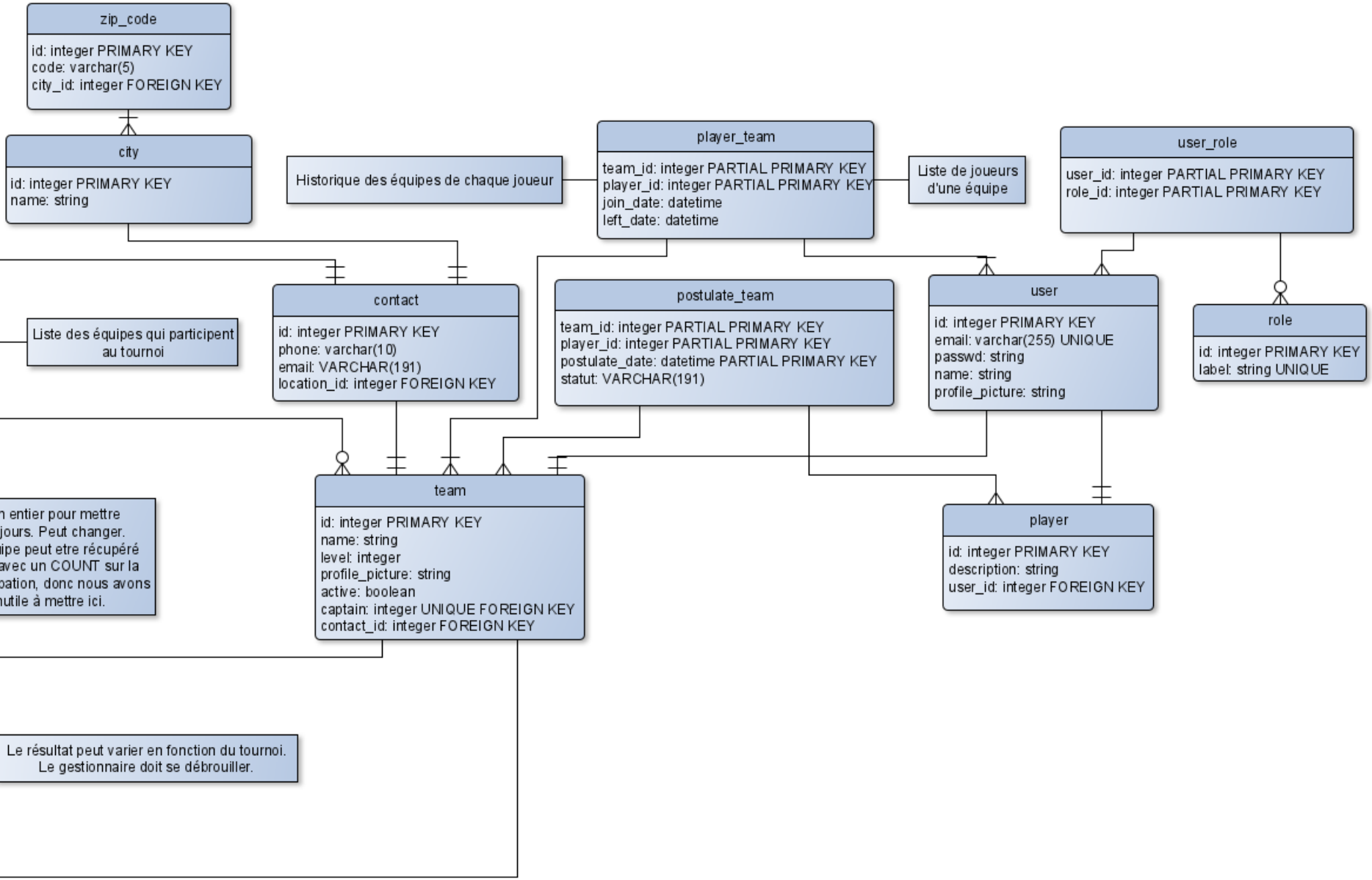

Feb 15 - Feb 21 '21
Feb 22 - Feb 28 '21
Mar 1 - Mar 7 '21
Mar 8 - Mar 14 '21
Mar 15 - Mar 21 '21
Mar 22 - Mar 28 '21
Mar 29 - Apr 4 '21
Apr 5 - Apr 11 '21
M T W T F S S

Apr 12 - Apr 18 '21
Apr 19 - Apr 25 '21
Apr 26 - May 2 '21
May 3 - May 9 '21
M T W T F S S

Annexe B

Schéma complet de la base de données




zip_code
id: integer PRIMARY KEY
code: varchar(5)
city_id: integer FOREIGN KEY
city
id: integer PRIMARY KEY
name: string
Historique des équipes de chaque joueur
player_team
team_id: integer PARTIAL PRIMARY KEY
player_id: integer PARTIAL PRIMARY KEY
join_date: datetime
left_date: datetime
Liste de joueurs d'une équipe
user_role
user_id: integer PARTIAL PRIMARY KEY
role_id: integer PARTIAL PRIMARY KEY
Liste des équipes qui participent au tournoi
contact
id: integer PRIMARY KEY
phone: varchar(10)
email: VARCHAR(191)
location_id: integer FOREIGN KEY
postulate_team
team_id: integer PARTIAL PRIMARY KEY
player_id: integer PARTIAL PRIMARY KEY
postulate_date: datetime PARTIAL PRIMARY KEY
statut: VARCHAR(191)
user
id: integer PRIMARY KEY
email: varchar(255) UNIQUE
passwd: string
name: string
profile_picture: string
role
id: integer PRIMARY KEY
label: string UNIQUE
n entier pour mettre
jours. Peut changer.
ipe peut etre récupéré
avec un COUNT sur la
pation, donc nous avons
nutile à mettre ici.
team
id: integer PRIMARY KEY
name: string
level: integer
profile_picture: string
active: boolean
captain: integer UNIQUE FOREIGN KEY
contact_id: integer FOREIGN KEY
player
id: integer PRIMARY KEY
description: string
user_id: integer FOREIGN KEY
Le résultat peut varier en fonction du tournoi.
Le gestionnaire doit se débrouiller.