Année 2014-2015

Rapport de projet - EI5 AGI - IAIE

Développement d'une application web avec Joomla et Symfony 2

Projet réalisé par :

Guillaume Fache
Rajesh Santhanam
Marouane Laouina

Projet encadré par :

Jean-Baptiste Fasquel

ISTIA l Institut supérieur des sciences et techniques de l'ingénieur d'Angers www.istia.univ-angers.fr l istia@univ-angers.fr
62 avenue Notre-Dame du Lac l 49000 ANGERS l Tél. 02 41 22 65 00 l Fax 02 41 22 65 01

Table des matières

Introduction

L'équipe de basket-ball de Beaupréau-Le Fief Sauvin compte plusieurs de dizaines de membres, plusieurs équipes regroupant toutes les catégories d'âge, et organise ou participe à de nombreuses rencontres chaque saison. Actuellement, le site du club est essentiellement articulé autour de l'affichage de renseignements dont les membres pourraient avoir besoin, à l'aide du CMS (Content Management System, ou Système de Gestion de Contenu) Joomla. Un certain nombre de nouvelles problématiques sont apparues lors de leur volonté de mettre à jour ce site, en ajoutant bon nombre de fonctionnalités. Quand le besoin d'administrer le club en ligne a émergé, la question de la technologie à utiliser pour réaliser ces fonctionnalités s'est vite posée. Une étude de faisabilité s'est donc révélée indispensable, et nous a occupé pour les premières séances. De plus, ce projet étant réalisé par trois étudiants, l'utilisation d'une plate-forme en ligne permettant de mettre en commun le travail réalisé s'est vite avéré indispensable. Le versioning a apporté, malgré quelques difficultés, une solution pratique à ce problème.

Nous consacrerons une première partie du rapport à l'étude de faisabilité qui a précédé toute réalisation technique. La deuxième partie s'attardera sur la première ossature du projet, et la découverte du framework Symfony 2. Enfin, dans la dernière partie nous nous pencherons sur la version finale du projet, et notamment l'utilisation de Bootstrap pour des vues adaptatives. Le rapport sera illustré de quelques fragments de code explicités soit par des commentaires, soit par une explication plus détaillée dans le corps de texte.

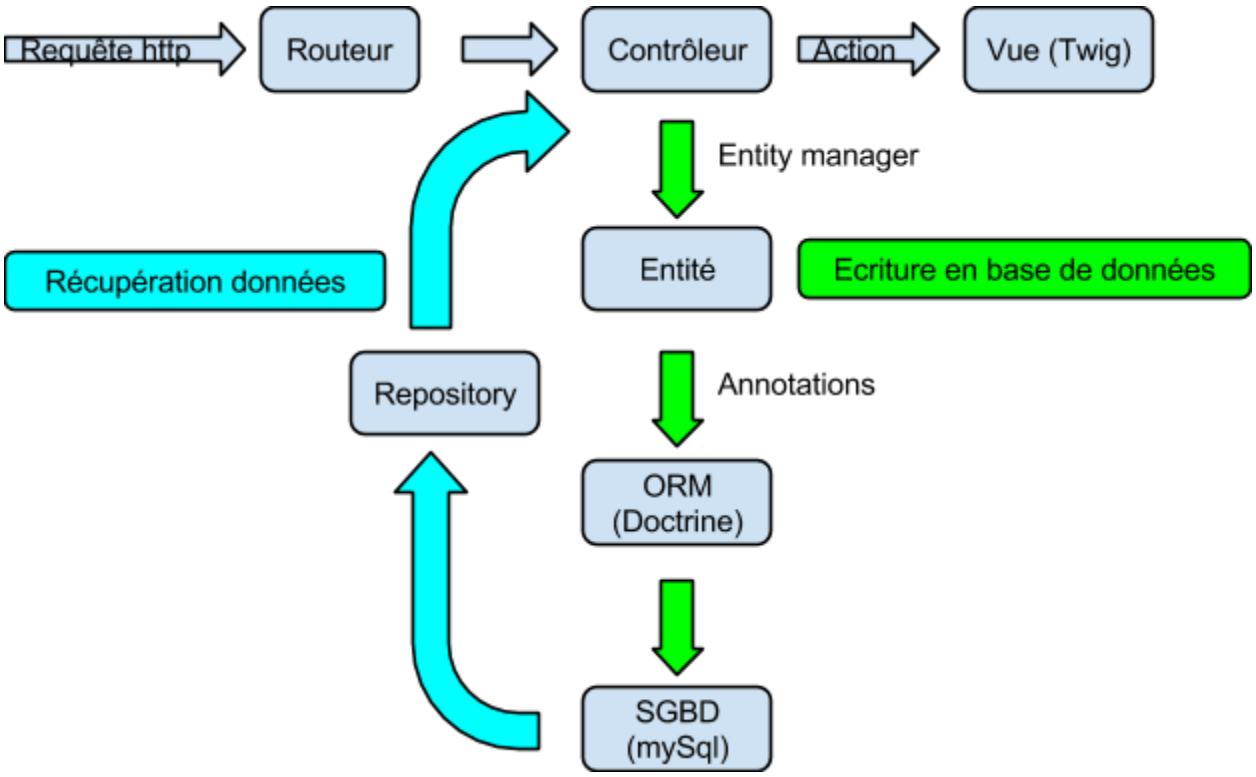
I Étude préliminaire du sujet

Le site web du club de basket de Beaupréau - Le Fief Sauvin est aujourd'hui réalisé à l'aide du CMS Joomla. Ce choix est parfaitement raisonnable puisqu'il s'agit avant tout d'une vitrine servant à mettre à disposition aux membres du club des informations pratiques dont ils pourraient avoir besoin, comme la date de leur prochain match, ou de leur prochain entraînement. Certaines fonctionnalités du cahier des charges ne remettaient pas en question ce choix technologique, comme la création d'un calendrier plus pratique pour l'utilisateur, parfaitement réalisable avec Joomla, mais d'autres, comme toute la gestion des personnes, des équipes, des entraînements, rendaient nécessaire la réalisation d'une étude de faisabilité, car pas toujours réalisable avec Joomla. Nous nous sommes donc penchés sur les possibilités offertes par Joomla, et conformément à ce que nous avions pu soupçonner au premier abord, un changement de technologie allait s'avérer nécessaire. Nous avons choisi de diviser ce projet en deux: une première partie serait consacrée à la mise à jour du site, en utilisant Joomla, pour que le site soit au moins partiellement mis à jour, dans la mesure des libertés offertes par le CMS. En plus de cela, nous avons décidé d'utiliser le framework PHP Symfony 2, pour créer un tout nouveau site web, en partant de zéro, et qui pourrait répondre aux spécifications du cahier des charges.

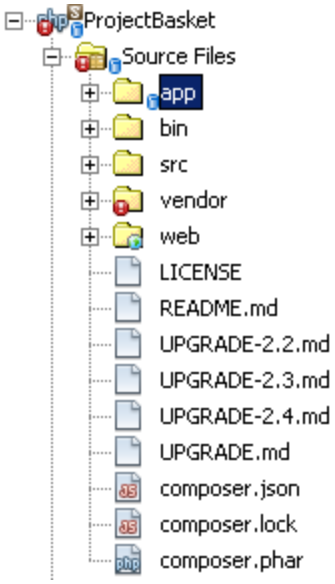
Le fonctionnement de Symfony

Pourquoi le choix de Symfony et pas celui de Java, ASP.NET, ou d'autres frameworks PHP comme Zend Framework ? Tout d'abord parce qu'il permet, grâce à une prise en main relativement aisée, de développer le projet rapidement. Une documentation facile d'accès et illustrée de nombreux exemples simples a permis de se plonger dans le coeur du problème sans passer trop de temps à maîtriser l'aspect technique du framework dans ses moindres détails. On a d'ailleurs pu rapidement se rendre compte des similitudes avec ASP.NET, et exploiter les compétences déjà apprises en cours pour dresser des parallèles entre les deux technologies à chaque fois que l'opportunité se présentait. De plus, au moins un membre du groupe avait déjà une expérience avec Symfony. Dans l'optique de privilégier les compétences déjà acquises, on a donc décidé de ne pas chercher ailleurs ce que l'on avait déjà sous la main. Enfin, ce framework gagne chaque jour en popularité, et savoir le maîtriser constitue une valeur ajoutée importante sur un CV pour qui veut se spécialiser en développement web.

Symfony est un framework MVC, dont on pourrait schématiser le fonctionnement ainsi :



La structure du projet (comme tout projet Symfony) est la suivante :



Dans le répertoire **app** nous trouverons l'essentiel des fichiers servant au paramétrage de l'application dans son ensemble (routage, sécurité, paramètres généraux …). Le répertoire **bin** est géré uniquement par Symfony et nous ne sommes jamais amenés à l'utiliser. De même **vendor** n'est pas destiné à ce que nous nous en servions directement. Il contiendra

tous les composants de Symfony nécessaire à son bon fonctionnement, et tous les bundles externes qu'on pourra être amenés à installer. C'est essentiellement par Composer, un outil de gestion de dépendances pour PHP, que nous allons gérer ces dépendances externes. Il suffit d'exécuter la commande (une fois qu'on est dans le répertoire du projet) :

```
> php composer.phar update
```

Composer va alors se référer au fichier composer.json, dans le même répertoire, et dans lequel on a inscris toutes nos références. Pour ajouter une référence à une ressource externe, on a plusieurs possibilités : à l'aide de la commande

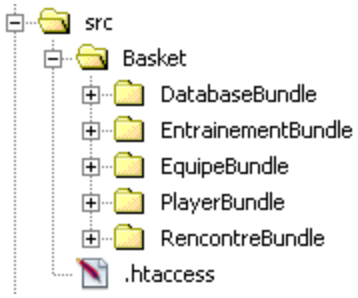
```
> php composer.phar require
```

ou bien directement en ajoutant manuellement au fichier composer.json :

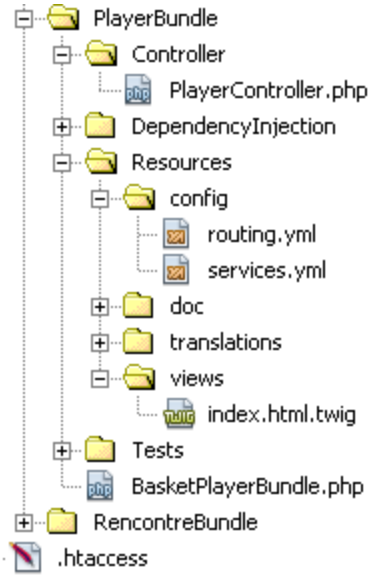
```
"require": {
        "php": ">=5.3.3",
        // Symfony lui même est un bundle !
        "symfony/symfony": "2.5.*",
        // L'ORM Doctrine
        "doctrine/orm": "~2.2,>=2.2.3",
        "doctrine/doctrine-bundle": "~1.2",
        // Le moteur de template Twig
        "twig/extensions": "~1.0",
        // Bundle qu'on a installé par nous même
        "friendsofsymfony/user-bundle": "~2.0@dev"
    },
```

On retrouve là tous les outils dont on sera amenés à se servir un peu plus tard dans le projet.

Le répertoire **src** par contre contiendra l'essentiel de notre code. C'est là notamment qu'on trouve tous nos différents bundles :



Chaque bundle dispose de la même organisation :



Le répertoire Controller contient le ou les controllers du bundle, qui permettront, en fonction de la route reçue par le routeur, d'exécuter l'action correspondant. Dependency Injection est un répertoire apparu dans les toutes dernières versions de Symfony, et nous n'avons pas eu à nous en servir, ni même en connaître le fonctionnement. Dans le répertoire Resources on va trouver le fichier de routing du bundle, et qui "hérite" d'un fichier de routage global (dans le répertoire app). On y trouve aussi la ou les vues associées au bundle. Enfin, le répertoire Tests sert à réaliser des tests unitaires de l'application, mais nous ne l'avons pas exploité. Mais avant de voir plus en détail le fonctionnement de notre application Symfony, jetons un oeil sur le travail réalisé pour mettre à jour l'application Joomla.

Implémentation d'un module calendrier dans Joomla

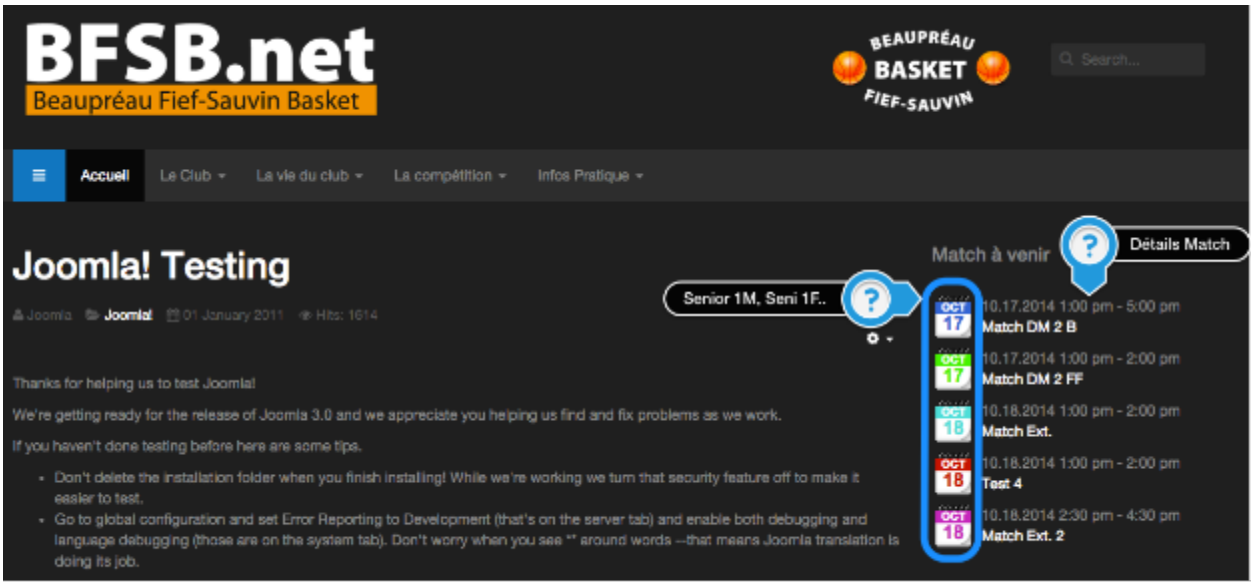


Fig : Joomla 3.0

Afin de répondre à la première demande du cahier des charges, il fallait passer le site actuel (réalisé avec Joomla 2.5, qui ne supporte pas les vues adaptatives (ou "responsive")) en Joomla 3.0, pour tenir compte de cet objectif d'adaptabilité, sur mobile et tablette. Bien que nous avons essayé de mettre à jour le site actuel vers la version 3.0, l'option "Joomla Update" était premièrement invisible. Après avoir recherché comment l'activer, nous étions confrontés à un autre problème, plus compliqué, lié aux fichiers manquant dans l'ancien site web. Nous avons donc décidé de partir d'un site totalement neuf, directement en version 3.0.

La deuxième partie du Joomla consistait à l'implémentation d'un calendrier responsive, afin que les membres du club puissent consulter les plannings des match sur n'importe quelle plates-forme. L'avantage des CMS (Content Management System) est qu'il existe plusieurs milliers de modules dans le "repository" (index général de tous les modules pris en charge officiellement par le CMS). Nous avons ajouté le module DPCalendar, qui correspondait le mieux au cahier des charges.

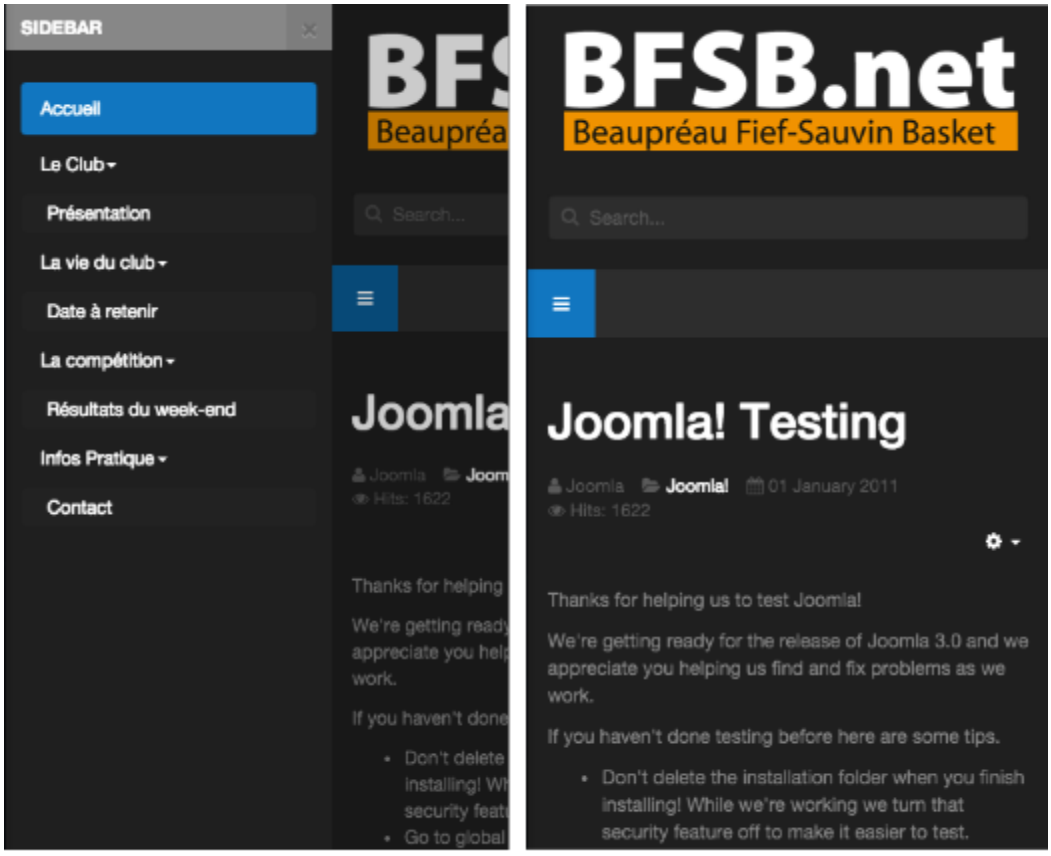
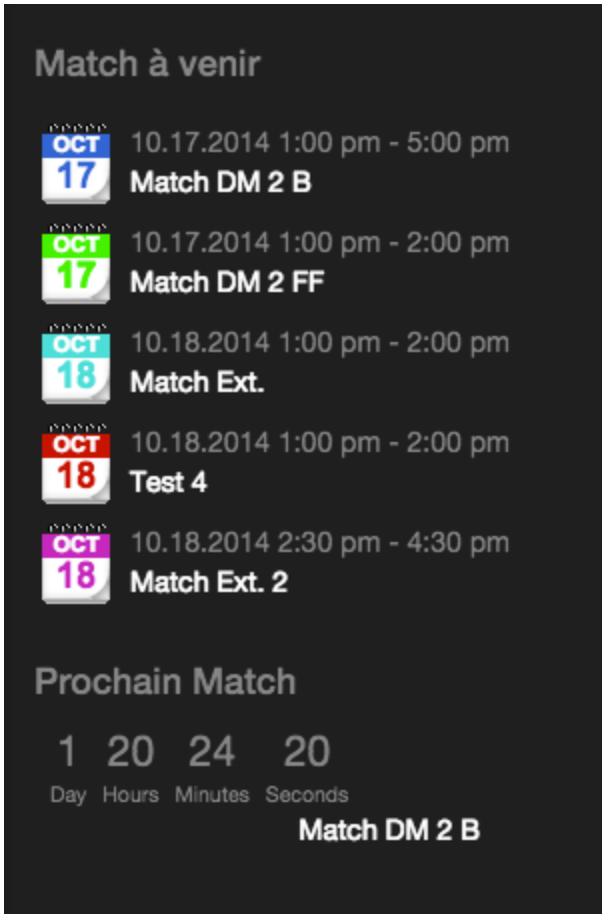


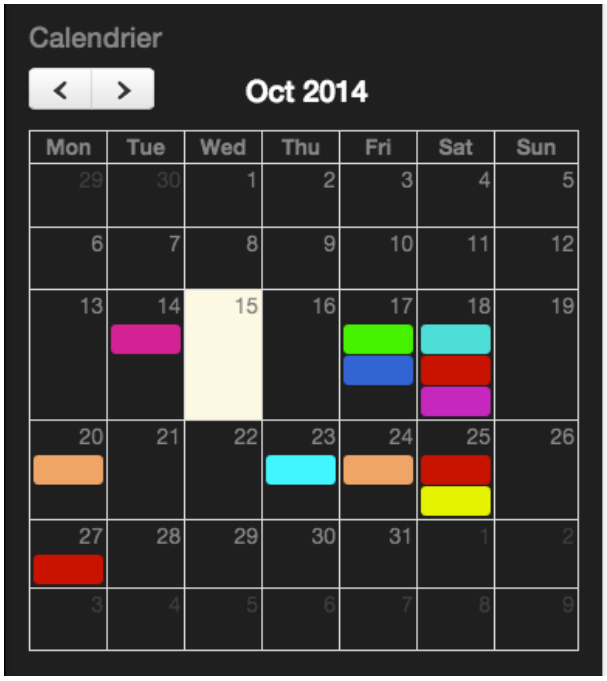
Fig : Joomla 3.0 - "Responsive"

Calendrier :

Le module DPCalendar possède des avantages importants, comme la création des événements avec une interface claire. On peut enregistrer des catégories en leur attribuant des couleurs uniques, pour pouvoir les identifier facilement sur le site. De plus, nous pouvons donner une description à chaque événement en indiquant la date, l'heure et la localisation du match.

Ce module comporte trois autres "sous-modules" (Match à venir, Calendrier & Prochain match), à l'aide desquels nous pourrions afficher des informations concernant les différents matchs sur le page principale du site.





L'inconvénient de ce module est qu'il devient payant (*cf: Annexe 1*), dès lors qu'on souhaite importer des événements à partir de liens externes (google calendar, facebook ...) ou de fichiers (csv, json ...). De plus, il possède sa propre base de données ce qui peut être un inconvénient majeur, si on souhaite faire des liens entre la base de données actuelle (Access) et la base de données du module. La procédure pour créer un match à l'aide de ce module est détaillée dans l'annexe 2.

Le versioning du projet

Avant même le début du codage à proprement parler, nous avons décidé d'utiliser un suivi de projet à l'aide d'un outil de versioning. Notre premier choix a été google code, car la perspective de rendre notre code public avec une plate-forme comme GitHub ne nous convenait pas entièrement. La prise en main de l'outil n'a pas été aussi simple que nous l'imaginions initialement, et des problèmes de sauvegarde du projet sont souvent apparus. A cause d'une mauvaise maîtrise des concepts de base du versioning, la synchronisation avec un repository est vite devenu plus une source de soucis qu'une véritable aide, comme elle était censée l'être. C'est là que nous avons décidé d'étudier plus en détail le fonctionnement

de Git, notamment grâce à la documentation officielle, qui est particulièrement claire et utile, pour partir sur une base de projet saine, cette fois sur GitHub. Le repository créé pour le projet est public, comme pour tous les comptes gratuit de GitHub, et est consultable à l'adresse : https://github.com/blety/projetBasket. En respectant quelques règles simples, comme rédiger des commit concis mais précis sur l'avancement du projet, en veillant à toujours "puller" (c'est à dire uploader le projet depuis le web), avant de "pusher" (c'est à dire télécharger sa propre version), pour travailler sur des versions à jour du projet, ou encore en faisant attention à ne commiter (c'est à dire "inscrire dans le contexte de sauvegarde") que les fichiers utiles (pas le répertoire vendor, ou encore le cache de l'application, qui vient se loger dans le répertoire app de l'application, comme nous l'avions fais dans un premier temps), nous avons réussi à gérer notre projet efficacement, et le suivi du projet a dès lors été nettement plus simple et agréable. De plus, cela nous a permis d'apprendre à maîtriser un outil très populaire en entreprise, et dont la connaissance constitue un plus certains dans un profil de développeur web.

II La première ossature

La structure en bundles

Au commencement était le cahier des charges, donnant les objectifs du projet, à savoir ajouter au site actuel du club la possibilité de l'administrer assez finement, en créant de nouvelles rencontres, de nouveaux entraînements, en ajoutant, modifiant et supprimant des membres du club ou des équipes. Toutes ces entités devaient interagir entre elles, la base de données finale devait être complétée par les valeurs réelles fournies par le client, les vues devaient être adaptative, et l'on devait essayer de faire avec ce que l'hébergeur du site permettait. Le cahier des charges était déjà articulé en entités facilement modélisables, et les interactions entre chacune d'elles étaient clairement définies. Une première version a donc été réalisé, assez sommaire dans un premier temps, mais qui allait être amenée à s'étoffer par la suite. On a opté pour une architecture où toutes les entités (c'est à dire les classes dans Symfony), et tous les formulaires issus de ces entités, seraient compris dans un Bundle nommé Database. Un Bundle chez Symfony représente un bloc fonctionnel, généralement articulé autour d'une entité. L'atout majeur de cette décomposition est de pouvoir exporter et réutiliser des pans entiers d'application dans d'autres projets. On a vu un peu plus haut que Symfony lui même n'est rien d'autre qu'un (gros) bundle. De plus, on a ainsi une meilleure lisibilité du projet, chaque élément étant dans son bloc propre. On a donc choisi de faire un Bundle par bloc fonctionnel, comme il est normalement prévu de le faire. Dans cette première version, nous avions donc un PlayerBundle et un EntrainementBundle, qui interagissaient entre eux. La création des Bundles s'est faite simplement par l'invite de commande :

```
> php app/console generate:bundle
```

Une fois la commande saisie, on peut renseigner les différents paramètres de notre Bundle :

```
Welcome to the Symfony2 bundle generator



Your application code must be written in bundles. This command helps
you generate them easily.

Each bundle is hosted under a namespace (like Acme/Bundle/BlogBundle).
The namespace should begin with a "vendor" name like your company name, your
project name, or your client name, followed by one or more optional category
sub-namespaces, and it should end with the bundle name itself
(which must have Bundle as a suffix).

See http://symfony.com/doc/current/cookbook/bundles/best_practices.html#index-1
```

```
for more
details on bundle naming conventions.

Use / instead of \ for the namespace delimiter to avoid any problem.

Bundle namespace:_
```

L'ensemble est très guidé. On choisit ici le namespace du Bundle, avec les bonnes pratiques de Symfony étant rappelées pour avoir un projet à l'architecture propre et lisible. Un peu plus loin on choisira le nom du Bundle, le type de format de configuration. Par préférence personnelle, tous les bundles seront configurés par des fichiers YAML (bien qu'il soit possible de choisir XML, ou encore les annotations). Ce choix n'a pas grande importance car la documentation est très claire quelque soit le type choisis. Symfony va aussi en profiter pour mettre à jour le Kernel (le noyau dur de l'application), ainsi que le routage (pour que le routage du Bundle soit ajouté au fichier de routage global à toute l'application). Notre bundle est créé. Il possède une route par défaut, qui pointe vers le contrôleur par défaut, qui renvoie la vue par défaut. Pour le DatabaseBundle, on a supprimé contrôleur, routage et vues, car il ne sert qu'à stocker toutes nos entités et nos formulaires.

Les entités, formulaires et l'ORM Doctrine

On se sert encore une fois de l'invite de commande pour créer nos entités.

```
> php app/console generate:doctrine:entity
```

Et là encore, un assistant va nous aider de façon très transparente à créer notre entité.

```
Welcome to the Doctrine2 entity generator



This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name:_
```

On choisit donc le nom de son entité (précédé du nom du bundle dans lequel elle se trouve, donc pour nous tout le temps DatabaseBundle:MonEntité), ainsi que le type et le nom de chacun des propriétés de notre entité. Une fois la saisie terminée, Symfony se chargera de générer l'entité correspondante, avec toutes les annotations Doctrine correspondant. On

pourra en rajouter par la suite, notamment pour définir les relations entre propriétés de deux entités.

Doctrine est l'ORM (Object-Role Mapping) associé par défaut à Symfony. C'est l'outil qui, grâce aux annotations placées au niveau des entités, permet d'établir des relations entre celles ci, ou encore gère la persistance en base données. Elle sert d'intermédiaire entre les classes de notre programme orienté objet, et les tables de notre base de données. Une des grandes force de Doctrine, est qu'on peut, en une ligne de commande, générer toute sa structure de base de données à partir des entités et des annotations renseignées sur celles-ci.

```
> php app/console doctrine:schema:update --dump-sql
```

Cette commande va nous permettre de visualiser les mises à jour de la base de données, et tester s'il n'y a pas d'erreurs dans nos relations entre nos entités. Si la commande s'exécute sans erreurs, on peut alors entrer :

```
> php app/console doctrine:schema:update --force
```

Doctrine mettra alors à jour nos entités et nos relations.

L'édition des entités se fait de façon minimaliste. Il suffit de renseigner le nom des propriétés souhaitées, et là encore, une simple commande permet l'édition des getters et setters associés.

```
> php app/console doctrine:generate:entities DatabaseBundle/Entity/MonEntité
```

On a essayé autant que possible de tirer parti des outils proposés par Symfony pour avoir un code correspondant au standards du Web.

Une fois nos premières entités créées, avec un nombre de propriétés réduites, qu'on pourrait complexifier par la suite, il fallait passer aux opérations CRUD (Create, Read, Update, Delete), qui nous permettrait de manipuler nos entités, et de les enregistrer en base de données. La première étape consistait à créer les formulaires permettant la saisie des informations utiles à la création de ces entités. Encore une fois, Symfony propose, plutôt que l'écriture manuelle et laborieuse, la génération de formulaire à partir d'entité, et ce avec une simple commande :

```
> php app/console generate:doctrine:form DatabaseBundle:MonEntité
```

Symfony générera un formulaire de la forme :

```
class PersonneType extends AbstractType
{
    /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('username')
            ->add('surname')
            ->add('genre','choice',array(
                'choices'=> array('masculin' => 'masculin', 'féminin' => 'féminin')
            ))
            ->add('birthday')
            ->add('save','submit')
        ;
    }
```

On retrouve très explicitement le même nom que les propriétés données à nos entités (en l'occurrence l'entité Personne), et c'est ce qui permet de ne pas avoir à préciser le type des champs du formulaire. Si le nom des propriétés n'étaient pas exactement les mêmes, on devrait les configurer nous même. C'est cette aspect "convention plutôt que configuration" qui fait que Symfony est un framework facilement prenable en main. On remarquera par contre que certains champs se sont vus assignés un second paramètre : il s'agit du type de champ. Ici, on a ajouté le type "Submit" au champ "save", pour qu'il permette la validation de notre formulaire. Un peu plus haut, on a assigné au champ "Genre" le type "Choice", qui permettra d'afficher une liste de choix prédéfinis à l'utilisateur. Les formulaires restent encore relativement simple, et se complexifieront par la suite.

Le traitement du contrôleur

A ce stade, nous aimerions bien tester le fonctionnement de notre application. Il ne nous reste plus qu'à "mapper" notre formulaire et notre entité (c'est à dire que les informations saisies dans le formulaire soit bien transmises à notre contrôleur, et que celui-ci puisse effectuer l'opération de l'enregistrer dans la base de données), et à créer une vue minimaliste comme première interface. Plaçons nous tout d'abord dans notre contrôleur (ici, pour continuer avec les exemples précédents, PlayerController, situé dans PlayerBundle) :

```
class PlayerController extends Controller
{
    public function indexAction(Request $request)
    {
        $player = new Personne();
        $form = $this->createForm(new PersonneType(),$player);
        $form->handleRequest($request);

        if ($form->isValid()){
          $em = $this->getDoctrine()->getManager();
          $em->persist($player);
          $em->flush();
        }

        $em = $this->getDoctrine()->getEntityManager();
        $allPlayers = $em->getRepository('BasketDatabaseBundle:Personne')->findAll();

        return $this->render('BasketPlayerBundle::index.html.twig', array(
            'form' => $form->createView(),
            'allPlayers'=>$allPlayers
                ));
    }
}
```

La fonction IndexAction est l'action appellée par le routage, configuré auparavant dans le fichier src/PlayerBundle/Resources/config/routing.yml :

```
basket_player_homepage:
    path:     /player/index
    defaults: { _controller: BasketPlayerBundle:Player:index }
```

IndexAction reçoit comme paramètre un objet de type Request, qui représente simplement la requête Http du client. On commence par créer une nouvelle Personne, et la ligne suivant s'occupe de mapper le formulaire nouvellement créé PersonneType à l'entité Personne créée juste auparavant. On s'occupe ensuite d'hydrater le formulaire à partir de la requête reçue en paramètre d'entrée. Si les opérations précédentes se sont passées sans exception levée, on peut faire appel à l'Entity Manager ($em) de Symfony, qui nous permettra d'effectuer les opérations CRUD. On commence par "persister" l'entité, c'est à dire l'inscrire dans le contexte de persistance géré par Doctrine. L'entité est ensuite réellement sauvegardée en base de données par l'instruction flush(), qui prend toutes les entités persistées et effectue les opérations correspondantes (INSERT si l'entité est persistée pour la première fois, UPDATE sinon).

Une fois l'inscription en base de données réalisée, on appelle à nouveau l'Entity Manager, pour récupérer toutes les personnes déjà enregistrées, et les stocker dans une variable ($allPlayers). On peut enfin renvoyer à la vue les différents éléments qu'on a généré dans le contrôleur : notre formulaire et notre variable $allPlayers. Ceux-ci seront accessibles très facilement grâce au moteur de templates TWIG.

Les vues et TWIG

Toutes nos vues seront construites en utilisant le moteur de templates TWIG, qui en plus de permettre l'affichage d'HTML classique, inclut quelques fonctions bien pratiques, comme la possibilité de parcourir des tableaux, d'hériter d'autres templates, ou encore d'afficher très facilement des formulaires. Ainsi, notre premier template ressemblait au suivant :

```
{% extends '::layout.html.twig' %}

    {% block title %} Player Index {% endblock %}

    {% block body %}
        {{ form(form) }}

        <table>
            <tr>
                <th>Nom</th>
                 <th>Prenom</th>
                 <th>Genre</th>
                 <th>Birthday</th>
            </tr>

        {% for player in allPlayers %}
            <tr>
                <td>
                {{player.nom}}
                </td>
                <td>
                {{player.prenom}}
                 </td>
                 <td>
                 {% if player.genre == '1' %}
                  Masculin
                  {% else %}
                  Feminin
                 {% endif %}
```

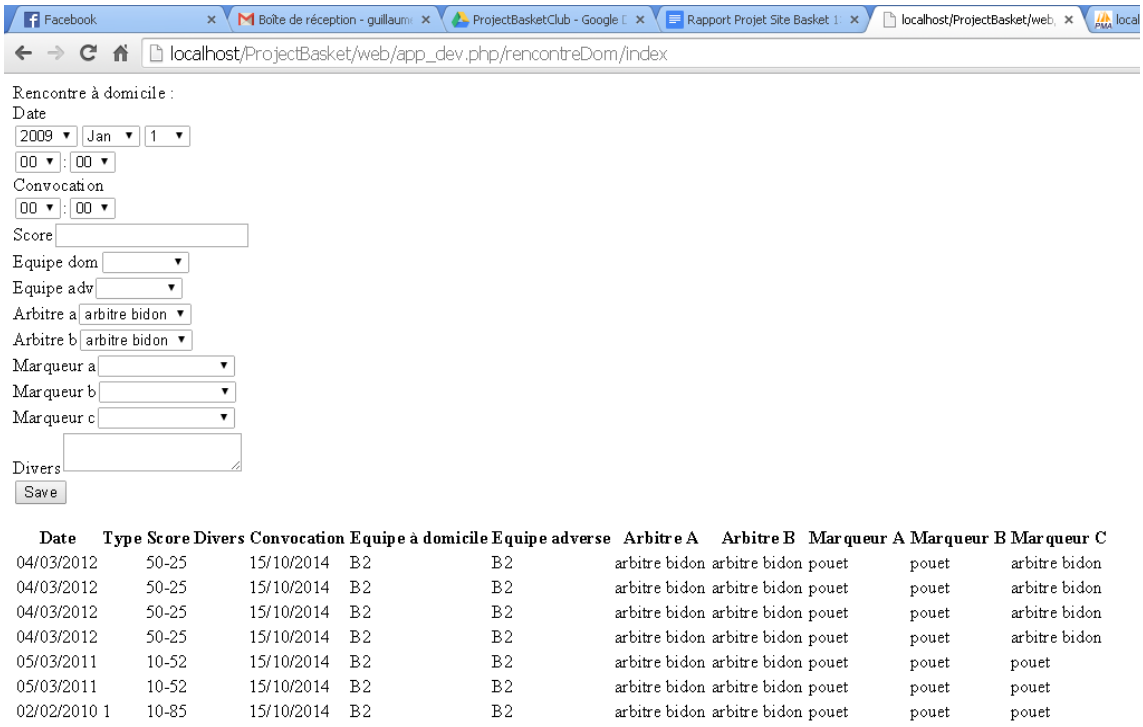
```
            </td>
            <td>
            {{player.birthday}}
            </td>
        </tr>
    {% endfor %}
  </table>
{% endblock %}
```

Pour bien comprendre la structure de cette vue, il faut présenter le principe d'héritage de template proposé par TWIG. Dans notre projet, conformément aux bonnes pratiques de Symfony, nos vues héritent toutes d'un layout général, qui contiendra toutes les informations communes à toutes les vues, comme la barre de navigation, le footer, etc. C'est le mot clé {{extends '::layout.html.twig'}} qui permet cela. Dans le layout principal seront définis des "block", qui définissent les zones ou les templates fils s'incrusteront. Traditionnellement on trouve un block {% block title %}, un {% stylesheets %}, un {% scripts %}, et surtout un {% body %}, où se trouvera l'essentiel de notre contenu. Ici, on remplira les blocks "title" et "body" de notre layout. Notre body commencera par afficher l'intégralité du formulaire, par la simple instruction {{form(form)}}. Il est bien sûr possible de détailler son affichage en le faisant champ par champ, mais ici ce n'est pas encore nécessaire, on cherche juste a tester son bon fonctionnement. On affiche ensuite un tableau qui permet d'afficher toutes les personnes récupérées en base de données et transmises par la variable $allPlayers. On remarquera que TWIG dispose de quelques fonctionnalités très pratiques, comme la possibilité de parcourir un tableau (ici notre tableau de joueurs $allPlayers), où encore l'affichage conditionnel d'une propriété (à l'aide de {% if … %}).

III La version finale

Implémentation de l'interface utilisateur (UI)

Afin de rendre la page créée par Symfony plus ergonomique, on décide d'utiliser Bootstrap. L'avantage de Bootstrap est de pouvoir construire le site sous forme de grille ("grid") afin de rendre le site modulable sur toute les plate-formes.



Tout d'abord pour que le site puisse avoir des références sur les syntaxes utilisés lors de l'implémentation de Bootstrap, il faut inclure ces lignes dans la vue principale (layout.html.twig) :

Code : Pour "stylesheet"

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
```

Code : Pour "javascript"

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
```

Surcouche “Flat Design”

Afin de rendre plus ergonomique et de s’approcher de la tendance actuelle, on utilise le célèbre “Flat Design”. Le “Flat Design” est défini par son style graphique minimaliste et la simplification de l’interface en supprimant des éléments supplémentaires tels que les ombres portées, les surfaces brillantes, les textures et les dégradés qui créent habituellement un look tridimensionnel.

Code : Pour “stylesheet”

```
<link rel="stylesheet" href="../../../web/css/flat-ui.css">
```

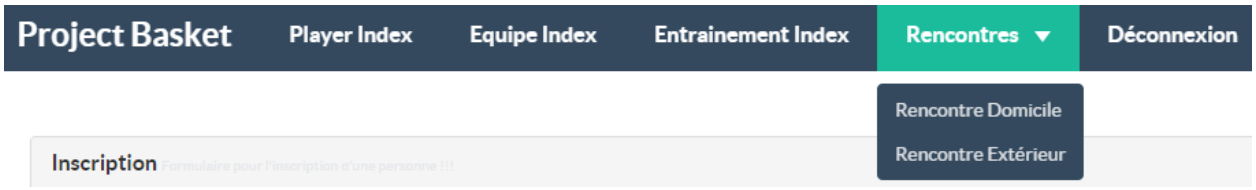
Code : Pour “javascript”

```
<script src="../../../web/js/flat.min.js">
```

En ajoutant ces lignes, on fait une surcouche du style de feuille “Bootstrap”.

Le menu de navigation du site

Nous avons décidé d’intégrer un menu global (présent sur toutes les pages) dans le site, afin de faciliter la navigation entre les différentes pages (Player, Entrainement, Rencontre Domicile, Rencontre Extérieur & Equipe).



Pour cela il faut ajouter ce code dans la page (layout.html.twig) qui sera le template par défaut des autres pages.

Code : Pour “menu navigation”

```
<nav class="navbar navbar-default"> ... </nav>
```

Décomposition du formulaire pour améliorer l'interface utilisateur

Une fois que nous avons généré le formulaire, nous avons d'abord testé son fonctionnement en l'affichant en une seule ligne :

Code : Formulaire Player	**Résultat :**
`{{ form(form) }}`	Rends le code HTML complet d'un formulaire.

Cependant, cet affichage est essentiellement destiné à la phase de test, et ne permet pas de spécifier les détails de l'affichage de notre formulaire. De plus pour pouvoir ajouter "Bootstrap", il faut décomposer le formulaire en plusieurs étapes.

Code : Personnalisé le formulaire

```
{{ form_start(form) }} //Rend le début du code formulaire
...
{{ form_end(form) }} //Rend la fin du code formulaire
```

Afin d'éditer chaque champ du formulaire on doit utiliser "form_widget" qui affiche le widget HTML d'un champ donné.

Code : Personnalisé chaque champ du formulaire

```
{{ form_widget(form.username, { 'attr': {'class': 'form-control'} }) }}
```

On définit le champ concerné dans le premier paramètre (ici dans l'exemple, le champ "username"), puis dans le deuxième paramètre on définit un tableau de variables qui seront affichées lors de l'affichage de la partie formulaire (ici, nous avons un tableau d'attributs qui nous permettra d'inclure la **classe** "*form-control*").

Nous pouvons aussi utiliser d'autres attributs HTML à la suite comme dans cet exemple de placeholder :

Code : Personnalisé chaque champ du formulaire

```
{{ form_widget(form.genre, { 'attr': {'class': 'form-control', 'placeholder':'Facultatif'} }) }}
```

Sauvegarde du formulaire et protection CSRF (Cross-site request forgery)

Lors de la sauvegarde du formulaire, on a constaté un problème lorsqu'on n'attribue pas de "jeton" à chaque formulaire : la fonction sauvegarde ne fonctionne pas. Il faut donc prendre en compte ce processus de la gestion de formulaire de Symfony et écrire :

Code : Protection CSRF

```
{{ form_widget(form.save, { 'attr': {'class': 'btn btn-success btn-lg btn-block'} }) }}

    {{ form_widget(form._token) }} // Génération de jeton pour la protection CSRF
```

Connexion à l'application comme administrateur

L'application était depuis le début pensée pour avoir deux interfaces distinctes : la première, une interface "administrateur", qui donne accès à tous les outils d'administration du club, et la seconde une interface "utilisateur" qui donnerait accès uniquement aux informations publiques comme les matchs ou les entraînements d'une personne. Nous avons décidé, plutôt que de réaliser une solution nous même, de se servir du FOSUserBundle. Ainsi, nous pouvions nous débarrasser de toutes craintes quant aux problèmes de sécurité inhérents à ce type d'opérations. En effet, ce Bundle est développé par une communauté spécialisée, qui veille à tenir le bundle à jour pour tout ce qui est relatif aux problèmes de sécurité. Il suffisait juste de se familiariser avec son fonctionnement et l'incorporer à notre projet. Le processus de son installation est détaillé dans la documentation officielle. On obtient au final que notre entité Personne dérive de la classe User du bundle. Cela implique que maintenant, chaque personne dispose de propriétés telles que "expired", si le compte n'est plus valide, "expires_at" pour fixer une date d'expiration, et parmi tant d'autre, surtout les propriétés "password" et "salt", dont on va se servir pour gérer l'authentification à l'application.

On commence par modifier notre paramètrage de la sécurité de l'application. Comme tout fichier de configuration, on le trouve dans le répertoire **app.** Avec les paramètres suivant, on ne peut accéder à l'application qu'une fois passé par le formulaire de connexion de FOSUserBundle.

```
security:
        encoders:
        // On définit l'encodage des mots de passe
        FOS\UserBundle\Model\UserInterface: sha512

       providers:
        fos_userbundle:
          // C'est l'username de la classe Personne qui sera utilisé comme identifiant
          id: fos_user.user_provider.username

        firewalls:
          main:
            pattern: ^/
            anonymous: ~
          form_login:
            provider: fos_userbundle
            login_path: fos_user_security_login
            check_path:     fos_user_security_check
            default_target_path: basket_player_homepage
          logout:
            path: fos_user_security_logout
            // On définit la redirection après la déconnexion
            target: basket_player_homepage

       access_control:
        // Pour accéder à la page de login, il faut qu'on ne soit pas identifié
        - { path: ^/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/admin/, role: ROLE_ADMIN }
       // Pour accéder à n'importe quelle autre page, il faut être passer par le login
        - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

Le formulaire d'authentification par défaut est très laid. On l'a donc surchargé pour en avoir un qui s'intègre plus harmonieusement à notre application. Il suffit pour cela de mettre un nouveau formulaire dans le répertoire app/Resources/FOSUserBundle/views/Security. Nous avons maintenant un joli formulaire d'authentification, mais lorsque l'on essaie de se connecter, les identifiants sont invalides. Pour que le mot de passe enregistré en base de données soit correct, on modifie notre controller de la sorte :

```
if ($form->isValid()){
        // on recupere le manager du fos user qui a partir du mot de passe donné le crypte
        // et le met en base
        $userManager = $this->container->get('fos_user.user_manager');
        $player->setPlainPassword($form->get('password')->getData());
        $userManager->updatePassword($player);

        $em = $this->getDoctrine()->getManager();
        $em->persist($player);
        $em->flush();
      }
```

Enfin, on ajoute au layout principal un lien "Déconnexion" vers le formulaire de déconnexion de FOSUser, et notre processus d'authentification fonctionne.

```
<li><a href="{{ path('fos_user_security_logout')}}">Déconnexion</a></li>
```

Conclusion

Les missions principales du cahier des charges ont été réalisées, que ce soit à l'aide de Joomla ou du projet Symfony 2. La mise à jour du site, avec l'affichage d'un calendrier contenant les dates importantes pour le club a pu être faite, mais toute la partie administration du club n'étant pas réalisable grâce à ce CMS, c'est à l'aide d'un projet Symfony 2 vierge qu'on a pu la concevoir. Au final, une interface administrateur fonctionnelle constitue le livrable final, associée à une base de données entièrement réécrite contenant toutes les informations des membres du club.

Ce projet a été l'occasion pour chacun de se mettre dans un contexte de mission professionnelle, et donc de gérer le projet un peu différemment qu'en cours. L'interaction avec le client est une préoccupation majeure dans ce type de projet, et a posteriori nous aurions pu accorder plus d'importance à ces échanges, en organisant des rencontres plus fréquentes, ou encore en livrant plus de versions intermédiaires, sur lesquelles on aurait pu avoir plus de retours constructifs. Qui dit projet à plusieurs dit mise en commun des ressources produites, et ce projet a donc été l'occasion d'apprendre à connaître plus en détail les concepts de base du versioning, et la plate-forme github, sur laquelle nous avons centralisé tout notre projet dès les premières versions. Enfin, devant l'impossibilité de mener à bien plusieurs fonctionnalités cruciales du cahier des charges seulement avec le CMS Joomla, ce projet a été l'occasion de découvrir le framework PHP Symfony 2, et les différents éléments qui le compose, comme son ORM Doctrine, ou son moteur de template Twig. Grâce à des connaissances théoriques en programmation orientée objet et sur le pattern MVC, on a pu prendre en main ce nouvel outil rapidement, et en découvrir les nombreux atouts. On a pu ainsi compléter notre panel d'outil de développement web, et les technologies apprises en cours. On a vu lors de l'étude de faisabilité qu'il était utile pour un ingénieur de connaître les différentes possibilités technologiques pour répondre à un problème donné, et ce projet a donc été l'occasion d'ajouter quelques cartes à notre main.

On a pu le voir, toutes les missions du cahier des charges n'ont pas forcément toujours pu être réalisées, essentiellement par manque de temps, mais pourront l'être à l'avenir par une autre équipe. Le code est commenté et respecte un maximum de normes de codage et de bonnes pratiques, pour faciliter la reprise du code plus tard. Pour pouvoir donner un côté "blog" avec post de billet sur l'actualité du club, peut être que l'installation d'un bundle spécialisé pour cette fonctionnalité serait pertinent. La piste du SonataBundle semble intéressante à explorer, et semble répondre au problème.

Bibliographie

Un ouvrage général sur les bonnes pratiques du développement Web :
Bien développer pour le Web 2.0 : bonnes pratiques Ajax de Christophe Porteneuve et Tristan Nitot.

Webographie

La documentation officielle de Git :
http://git-scm.com/doc

La page d'aide de GitHub, qui répond aux questions les plus communes : https://help.github.com/

Documentation spécifique au remote push & pull :
https://help.github.com/articles/pushing-to-a-remote/

La documentation générale de Symfony :
http://symfony.com/doc/current/book/index.html

La documentation spécialisée de Symfony : http://symfony.com/doc/current/cookbook/index.html

La documentation officielle Bootstrap :
http://getbootstrap.com/css/

Histoire sur "Flat Design" :
http://www.kryzalid.net/blog/2013/09/09/un-regard-en-profondeur-sur-le-flat-design/

Pack "Flat UI" for Bootstrap :
http://designmodo.github.io/Flat-UI/

Documentation Module DPCalendar Joomla :
https://joomla.digital-peak.com/documentation/58-dpcalendar

Documentation décompisition formulaire :
http://symfony.com/fr/doc/current/reference/forms/twig_reference.html

Protection CSRF:
http://symfony.com/fr/doc/current/book/forms.html#protection-csrf
Documentation du Sonata project, pour le blog bundle :
http://sonata-project.org/bundles/news/master/doc/reference/introduction.html

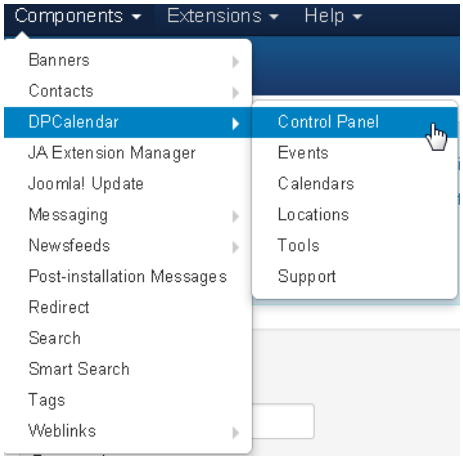
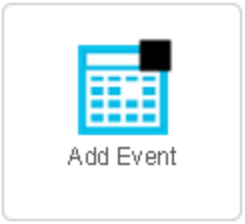
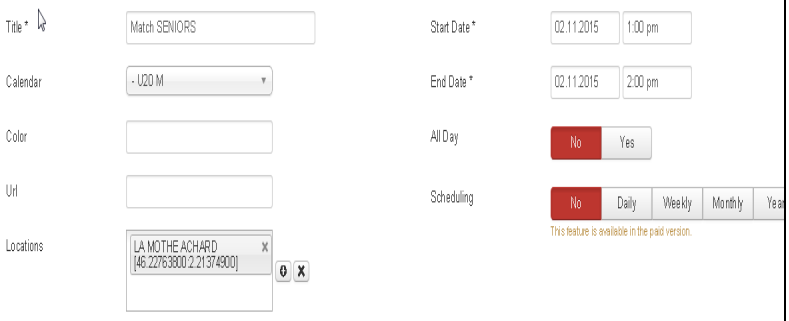
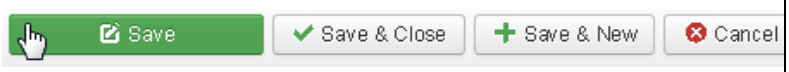
Documentation officielle du FOSUserBundle, qui décrit le processus d'installation :
https://github.com/FriendsOfSymfony/FOSUserBundle/blob/master/Resources/doc/index.md

Annexe :

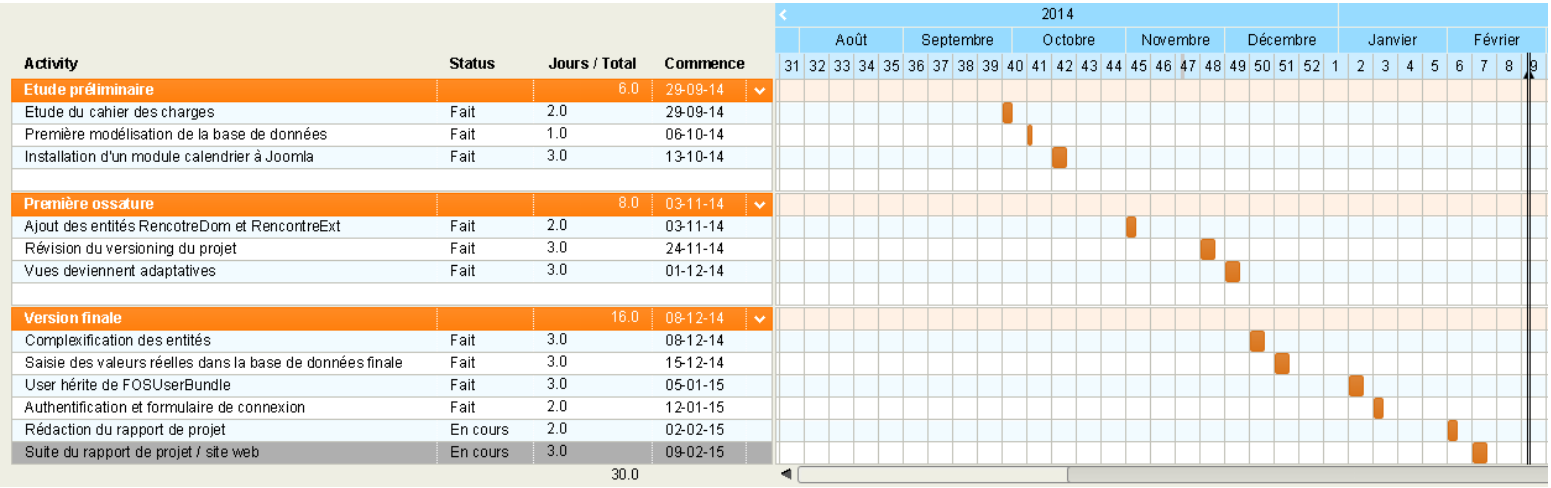
Annexe 1 : Prix de DPCalendar :

Buy now*	Download	Order	Order	Order
Duration	unlimited	3 months**	6 months**	12 months**
Domains	unlimited	unlimited	unlimited	unlimited
Extensions				
Component***	✓	✓	✓	✓
Upcoming Module	✓	✓	✓	✓
Counter Module	✓	✓	✓	✓
Mini Module	✓	✓	✓	✓
Map Module	⛔	✓	✓	✓
Search Plugin	✓	✓	✓	✓
Finder Plugin	⛔	⛔	✓	✓
Ical Plugin	⛔	✓	✓	✓
Jomsocial Plugin	⛔	⛔	✓	✓
CSV Plugin	⛔	✓	✓	✓
JEvents Plugin	⛔	✓	✓	✓
JCalPro Plugin	⛔	✓	✓	✓
Google Calendar Plugin	⛔	⛔	✓	✓
Facebook Event Plugin	⛔	⛔	✓	✓
CalDAV Plugin	⛔	⛔	✓	✓
MS Exchange Plugin	⛔	⛔	⛔	✓
Meetup Plugin	⛔	⛔	⛔	✓
Twitter Plugin	⛔	⛔	⛔	✓

Annexe 2 : Procédure pour créer un match à l'aide du module DP Calendar

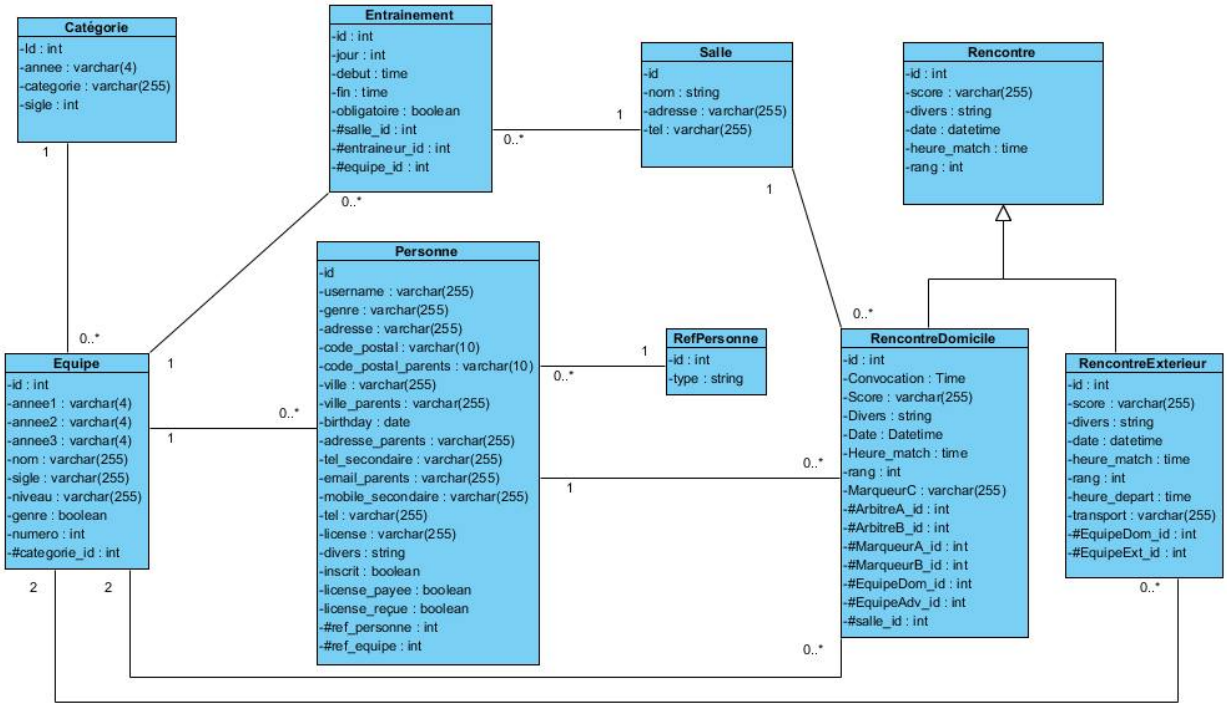
Etapes	**Résultats :**
Etape 1 : Components => DPCalendar => Control Panel	
Etape 2 : Cliquer sur "Add event"	
Etape 3 : Fournir un titre, Sélectionner la catégorie (l'equipe), choisir une date et l'heure de début et de fin du match. Ainsi que l'endroit ou va se dérouler le match (Localisation)	
Etape 4 : Appuyer sur "Save" pour sauvegarder le match	

Annexe 3 : Gantt du projet



Activity	Status	Jours / Total	Commence
Etude préliminaire		6.0	29-09-14
Etude du cahier des charges	Fait	2.0	29-09-14
Première modélisation de la base de données	Fait	1.0	06-10-14
Installation d'un module calendrier à Joomla	Fait	3.0	13-10-14
Première ossature		8.0	03-11-14
Ajout des entités RencotreDom et RencontreExt	Fait	2.0	03-11-14
Révision du versioning du projet	Fait	3.0	24-11-14
Vues deviennent adaptatives	Fait	3.0	01-12-14
Version finale		16.0	08-12-14
Complexification des entités	Fait	3.0	08-12-14
Saisie des valeurs réelles dans la base de données finale	Fait	3.0	15-12-14
User hérite de FOSUserBundle	Fait	3.0	05-01-15
Authentification et formulaire de connexion	Fait	2.0	12-01-15
Rédaction du rapport de projet	En cours	2.0	02-02-15
Suite du rapport de projet / site web	En cours	3.0	09-02-15
		30.0	

Annexe 4 : Meurise de la base de données finale



Résumé

Ce projet avait pour objectif la mise à jour du site d'un club de Basket, en lui apportant plusieurs nouvelles fonctionnalités, allant de l'ajout d'un calendrier pour les utilisateurs, jusqu'à rendre l'ensemble du site adaptable à toute plate-forme. Une partie du cahier des charges spécifiait cependant qu'il était nécessaire de pouvoir administrer le site, en ajoutant des rencontres, des entraînements, des personnes, etc… Ceci n'était pas réalisable avec Joomla, outil avec lequel était développé le site jusqu'alors, et a donc entraîné une étude de faisabilité afin de déterminer quel technologie utiliser pour implémenter ces fonctionnalités. Notre choix s'est porté sur Symfony 2, associé à l'ORM Doctrine 2, et le moteur de template Twig. Nous avons réalisé une interface d'administration répondant au cahier des charges, et permis l'authentification d'utilisateur à l'aide du FOSUserBundle. De plus, nous avons mis à jour le site Joomla, pour que les vues soient pleinement conformes au cahier des charges. Aujourd'hui, les deux applications sont pleinement fonctionnelles, et pourront toujours être améliorés l'an prochain par une autre équipe, grâce aux respect des bonnes pratiques du développement Symfony et les commentaires partout dans le code.

Mots-clés : Symfony 2, Joomla, vues adaptables, application web.

Summary

This project's goal was to update a basket club's website, bringing it brand new functionalities, from adding a calendar module to the existing website for the user to check intel more easily, or making all the views responsive, to be viewable on all devices. Another part of the specifications was to enable the administration of the club online, making it possible to create new tranings, new games, new persons, and many more.This was not manageable with the existing tool used to develop the website. We had to change it to a new technology, and studied what the possibilities were. We chose Symfony 2, with the ORM Doctrine 2 and the template motor Twig. We created an administrator interface according to all the specifications, and enabled the authentification through the use of the FOSUserBundle. Moreover, we updated the Joomla application, so that the view are fully responsive, according to the specifications. Today, both the applications are fully working, and can still be improved next year thanks to a great respect of the Symfony's good practices, and comments all across the code.

Keywords : Symfony 2, Joomla, responsive design, web application