

UNIVERSITÉ DE CAEN BASSE-NORMANDIE

RAPPORT DE STAGE

1 AVRIL 2015 - 30 SEPTEMBRE 2015

Open orchestra

Auteur :

Amaury LAVIEILLE

Enseignants :

Marc SPANIOL

Jean-Marc LECARPENTIER

20 août 2015



Open Orchestra est un CMS open-source, multi-sites, multi-langues, multi-devices, réalisé avec le framework PHP symfony2, conçu pour être hautement flexible et personnalisable. En version bêta depuis un an, la première version est prévue pour le mois de septembre.

Il est développé par une équipe, de l'agence digitale Interakting du groupe Business & Decision, selon les principes de la méthode agile Scrum.

Open Orchestra is a open-source CMS, multi-sites, multi-languages, multi-devices, developed with the PHP framework Symfony2, designed to be highly adaptable and expandable. In beta since one year, the first version is planned for september.

It is developed by a team of the digital agency Interakting of Business & Decision group, according to the agile Scum method.

Remerciements

Tout d'abord, je tiens à remercier l'ensemble de l'agence Interakting pour m'avoir offert l'opportunité de réaliser mon stage au sein de l'agence.

Je tiens aussi à remercier plus particulièrement les différents membres de l'équipe d'Open Orchestra présent durant mon stage pour tous leurs conseils et leur accueil au sein de l'équipe :

— Nicolas Bouquet, Product Owner

— Nicolas Thal, Scrum master

— Nicolas Anne

— Noël Guilain

— Romain Boëssel

Table des matières

Remerciements	1
Introduction	4
I L'entreprise	5
1 Présentation de l'entreprise	6
1.1 Business & Decision	6
1.2 Interakting	7
II Projet	8
2 Open Orchestra	9
2.1 Présentation	9
2.2 Architecture	9
2.3 Fonctionnalités	10
2.3.1 Blocs	10
2.3.2 Nodes	11
2.3.3 Content type	12
2.4 Caractéristiques	13
2.4.1 Performance	13
2.4.2 Modularités	14
III Symfony2	16
3 Concepts clés	17
3.1 Bundles	17
3.2 Conteneur de service	18
3.2.1 Services	18
3.2.2 Conteneur	19
IV Organisation du développement	20
4 Développement	21
4.1 Gestion de projet agile	21
4.1.1 Méthodes agiles	21
4.1.2 Scrum avec Open Orchestra	22
4.2 Intégration continue	24

4.2.1	Git	24
4.2.2	Travis CI	26
4.2.3	Déploiement	27
4.3	Qualité	27
4.3.1	Qualité du code	27
4.3.2	Test	28
4.3.3	Qualité fonctionnel	29

Bilan	30
--------------	-----------

Annexes	31
----------------	-----------

4.4	Aperçus	31
-----	-------------------	----

Introduction

Dans le cadre de ma deuxième année du master DNR2I (Master document numérique en réseau, ingénierie de l'Internet), j'ai effectué un stage chez Business & Decision au sein de l'agence Interakting à Caen.

Pour une durée de six mois, du 1^{er} Avril au 30 Septembre 2015, j'ai été intégré au sein de l'équipe de développement du projet Open Orchestra.

Dans un premier temps, je vais vous présenter succinctement l'entreprise. Ensuite, je vous décrirais le projet Open Orchestra, son origine, ses particularités, etc. Enfin, je vous parlerais de l'organisation et des méthodes mises en place pour la réalisation du projet.

Première partie

L'entreprise

Chapitre 1

Présentation de l'entreprise

1.1 Business & Decision

Business & Decision est un groupe international spécialisé dans trois grands domaines, la Business Intelligence¹ (BI), la gestion de la relation client et enfin le e-business.

Le groupe a été créé en 1992 par Patrick Bensabat, aujourd'hui il est présent dans 15 pays et emploie plus de 2500 personnes.

Événements majeurs et évolution du groupe depuis 1992 :

1992 : Création de Business & Decision par Patrick Bensabat autour de projets de Business Intelligence

1993-1996 : Mise en place des premiers Datawarehouses² et applications de prévisions financières

1999-2000 : Création de la division CRM (Gestion de la relation client)

2002-2003 : Acquisition en Grande-Bretagne et au Benelux. Création d'agences régionales en France

2004 : Acquisition en Suisse et aux Pays-bas et implantation en Tunisie

2005 : Implantation aux États-Unis et acquisition de Metaphora³

2007-2008 : Création de la marque Interkating

2011 : Déploiement d'offres Cloud et Mobilité. Inauguration du Datacenter éco-responsable d'Eolas

2013 : Lancement de Datalyse, programme de recherche en Big Data. Implantation au Pérou

2014 : Création de Herewecan, agence de communication digitale, Lancement des pôles d'expertise Big Data, Transformation et Hub Mobile)

1. La Business Intelligence ou l'informatique décisionnelle représente les différentes solutions informatiques qui permettent l'exploitation des données d'une entreprise dans le but de faciliter la prise de décisions

2. Les Datawarehouses ou entrepôt de données désigne une base de données utilisée pour stocker et structurer des données de production d'une entreprise afin de fournir des informations stratégiques pour la prise de décisions

3. Metaphora est une société de service spécialisée dans la conduite du changement. Elle intervient dans toutes les étapes d'un projet pour faciliter l'appropriation du futur système d'informations par les utilisateurs finaux.

1.2 Interakting

Interakting est l'agence web du groupe B&D composée de 340 employés répartis principalement entre Caen et Paris, les équipes de l'agence interviennent sur des projets web de grande envergure (Canal +, Bnp Paribas, Inra, Moët Hennessy, PSA Peugeot Citroën, ...).

Les différents projets de l'agence sont développés avec différents outils : eZ publish (système de gestion de contenu créé par l'entreprise eZ Systems As), PHP Factory (plateforme réalisée conjointement par Interakting et Zend Technologies) et de plus en plus avec Symfony2 (framework php écrit par SensionLabs) avec notamment Open Orchestra.

Deuxième partie

Projet

Chapitre 2

Open Orchestra

2.1 Présentation

Open Orchestra est un CMS (Content Management System) open source en développement depuis un peu plus d'un an. Il est actuellement en version bêta, la sortie de la première version est prévu pour le mois de septembre.

Il est basé sur le framework PHP MVC Symfony 2 et MongoDB. Open Orchestra offre bien-sûr les fonctionnalités attendues par tous CMS : gestion de contenu, médiathèque, contribution de page, gestion de version, utilisateurs, workflow, rôles, multi-site, multi- langue, multi-device , etc.

La grande particularité d'Open Orchestra est son faible couplage au niveau du code, mais aussi au niveau des solutions techniques, c'est-à-dire que l'on peut facilement remplacer ou étendre un de ses composants. Je reviendrais plus en détail sur ce point dans une section ultérieure.

2.2 Architecture

Avec Open Orchestra le « Back-office » et le « Front office » peuvent être dissociés dans deux applications Symfony différentes.

Ce découpage a de nombreux avantages, tout d'abord un seul « Back-office » peut gérer plusieurs sites (« Front-office »), de plus cela permet de mettre « Back-office » et le « Front-office » sur des serveurs différents, l'unique condition est qu'ils doivent tous utiliser la même base de données.

Pour finir, Open Orchestra utilise une API RESTFull¹ pour accéder, gérer ces différentes entités (pages, utilisateurs, contenus, etc).

Le schéma 2.1 présente l'architecture d'Open Orchestra avec un « Back-office » qui gère plusieurs « Fronts ».

1. REST (Representational State Transfer) est un style d'architecture qui définit différentes règles (ressource identifiée unitairement, utilisation des verbes HTTP POST, DELETE, PUT, GET) pour accéder et manipuler des ressources

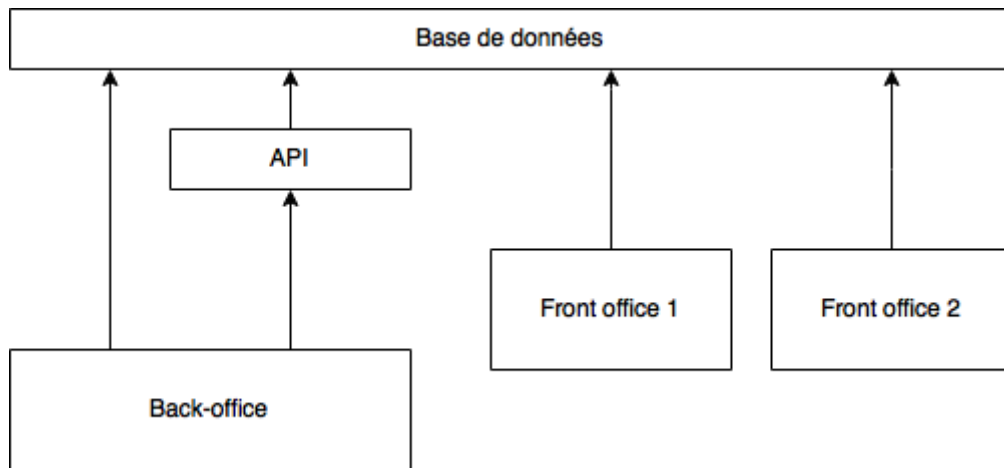


FIGURE 2.1 – Exemple d’architecture d’utilisation d’Open Orchestra

2.3 Fonctionnalités

Dans ce chapitre je vais vous présenter les différentes fonctionnalités d’Open Orchestra. Bien sûr, je ne vais pas toutes les détailler, mais uniquement les points clés qui permettent une bonne compréhension du projet.

2.3.1 Blocs

Dans Open Orchestra tous les éléments visibles en front sont représentés par des blocs. Un bloc est simplement une entité avec des attributs qui varient selon le type de bloc. Chaque type de bloc est indépendant, c’est-à-dire que eux seuls connaissent leurs attributs, leurs façon de s’afficher en front ou en back-office ou encore la façon dont ils peuvent être renseigné en back-office.

Pour permettre cette indépendance, Open Orchestra utilise le design pattern stratégie. Le pattern stratégie permet de rendre une famille d’algorithmes interchangeables et ainsi la possibilité d’exécuter un traitement spécifique selon le contexte.

Comme le présente le diagramme de classe 2.2 chaque bloc possède une classe qui indique comment il doit s’afficher, pour cela la classe a deux méthodes, la première qui indique le bloc qu’elle supporte (**support**) et la méthode **show** qui fournit le rendu, d’un autre côté, il y a une classe appelée **manager** qui est la seule à connaître toutes les stratégies des différents blocs.

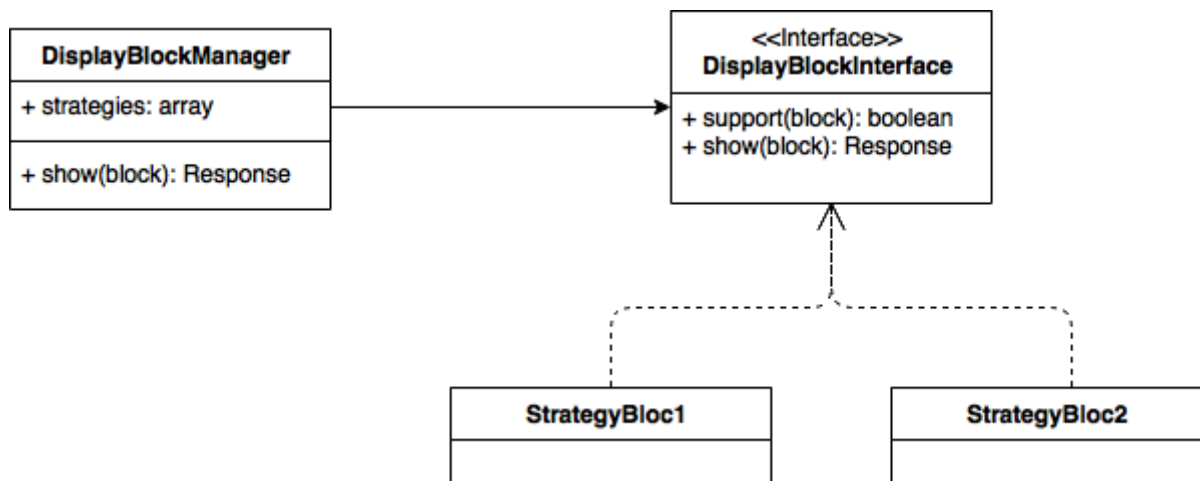


FIGURE 2.2 – Utilisation du pattern stratégie pour l’affichage des bloc en front

Ainsi, lorsque que l’on désire afficher un bloc, il suffit de demander au manager d’afficher ce bloc, ce dernier va chercher parmi les stratégies qu’il connaît celle qui supporte (méthode `support`) le bloc et s’il en trouve une alors il exécute la méthode (`show`) de la stratégie.

Par défaut, le CMS propose de nombreux types blocs (annexe 4.6) comme par exemple un bloc pour lister un type de contenu, afficher un texte formaté, afficher un carrousel ou un média, un menu, une carte, un bloc de contact, etc.

Bien-sûr, il est possible pour un intégrateur², de créer son propre type de bloc, il lui suffit de développer les différentes stratégies (affichage en front et back-office, formulaire en back-office, etc) nécessaires à un bloc.

2.3.2 Nodes

Un des points central d’un CMS est les pages. Sur Open Orchestra les pages sont identifiées comme des noeuds (« nodes »). Les nodes sont simplement des conteneurs qui contiennent des zones. Les zones permettent d’organiser la page, elles contiennent des sous-zones ou des blocs (annexe 4.5) ce qui permet un découpage fin de la page pour simplifier sont organisation.

Il existe trois types de nodes :

- Les nodes qui représentent les pages visibles en « front ».
- Les « node tranverse » qui contiennent les blocs transverses, c’est-à-dire les blocs qui sont communs à plusieurs pages comme un bloc « menu » ou encore un bloc « footer ».
- Le dernier type de node est celui qui permet de contribuer les pages d’erreurs (404, 503) d’un site.

2. Dans le cadre de ce rapport un intégrateur indique une personne ou une équipe qui utilise Open Orchestra afin de l’étendre pour des besoins particuliers

Le schéma 2.3 montre un exemple typique de page avec trois zones, le header qui contient un bloc menu, le footer un bloc footer et une zone principale découpée elle-même en deux zones avec un bloc de contact pour la zone de droite et un bloc de texte à gauche.

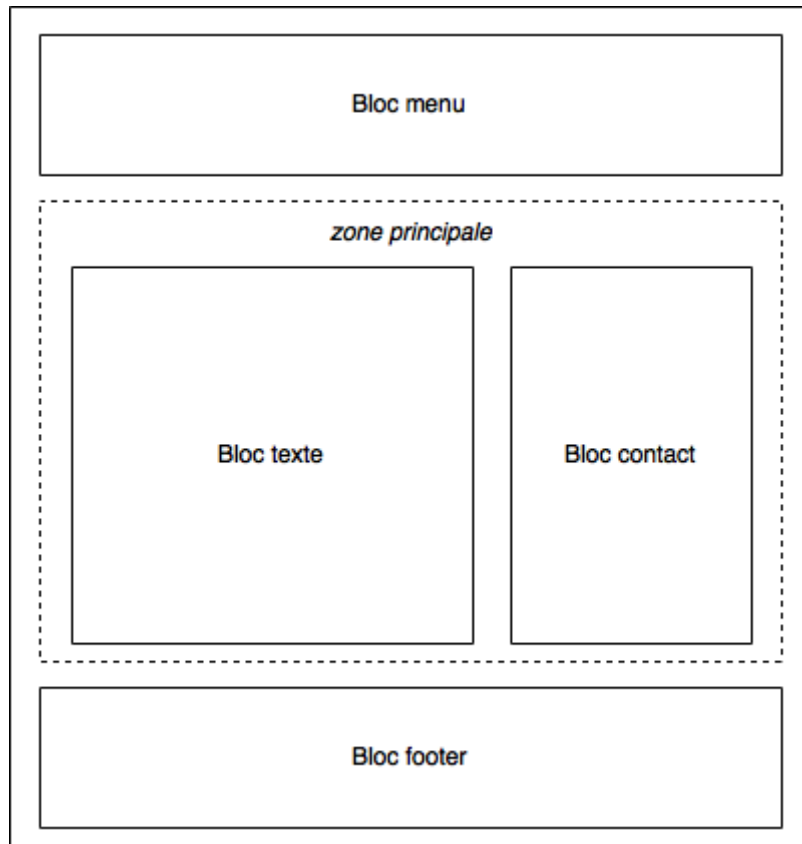


FIGURE 2.3 – Exemple de page

2.3.3 Content type

Le second point important d'un CMS est les contenus, avec Open Orchestra il est facile de créer des types de contenus (actualité, client, etc, ...).

En effet, Open Orchestra propose une approche graphique grâce à un formulaire pour créer ou éditer des types de contenus comme l'illustre la figure 2.4.

Content type id* news

Names in all the languages* en* fr*

Edit form custom template

Link to the current site OFF

Visible by default in BackOffice list

Name ON

Status label ON

Version ON

Language ON

Linked to site ON

Fields*

Field id* title

Label* en* fr*

Title

Searchable ON

Translatable ON

Type* Text line

Visible in BackOffice list OFF

Max length* 25

Field required ON

Default value

FIGURE 2.4 – Formulaire d’édition du type de contenu news

Un type de contenu est composé de différents champs avec différentes options, par exemple pour un champ de type texte il peut y avoir une option pour limiter le nombre de caractère ou encore une valeur par défaut.

Par défaut, Open Orchestra offre une liste de types de champs (date, texte, monnaie, média, email, entier, zone de texte riche, etc) qui comme pour tous les composants d’Open Orchestra peut être étendu par d’autre type de champ personnalisé.

2.4 Caractéristiques

2.4.1 Performance

Open Orchestra a été développé pour supporter une charge de trafic importante. Pour cela le CMS exploite au niveau du front l’ESI (Edge Side Includes) couplé à un reverse proxy³.

L’ESI est un langage de balisage HTML qui permet de diviser une page en différents éléments dont les rendus sont faits dans différentes requêtes par le serveur web. Ce qui permet d’avoir un cache HTTP sur les différents éléments et ainsi rafraichir seulement les éléments obsolètes de la page et non toute la page.

3. Un reverse proxy est un serveur qui traite les requêtes en amont du serveur web. L’intérêt d’un reverse proxy est multiple gestion de cache, chiffrement, répartition de charge

Comme nous l'avons vu dans la section 2.3.1 sur Open Orchestra les pages sont déjà découpées en différents blocs ainsi l'utilisation de l'ESI est adapté et permet une amélioration significative des performances.

Si nous reprenons l'exemple de notre page (schéma 2.3), le schéma 2.5 présente le processus effectué par le serveur dans le cas où les blocs **header** et **footer** sont déjà en cache lorsqu'un utilisateur demande la page.

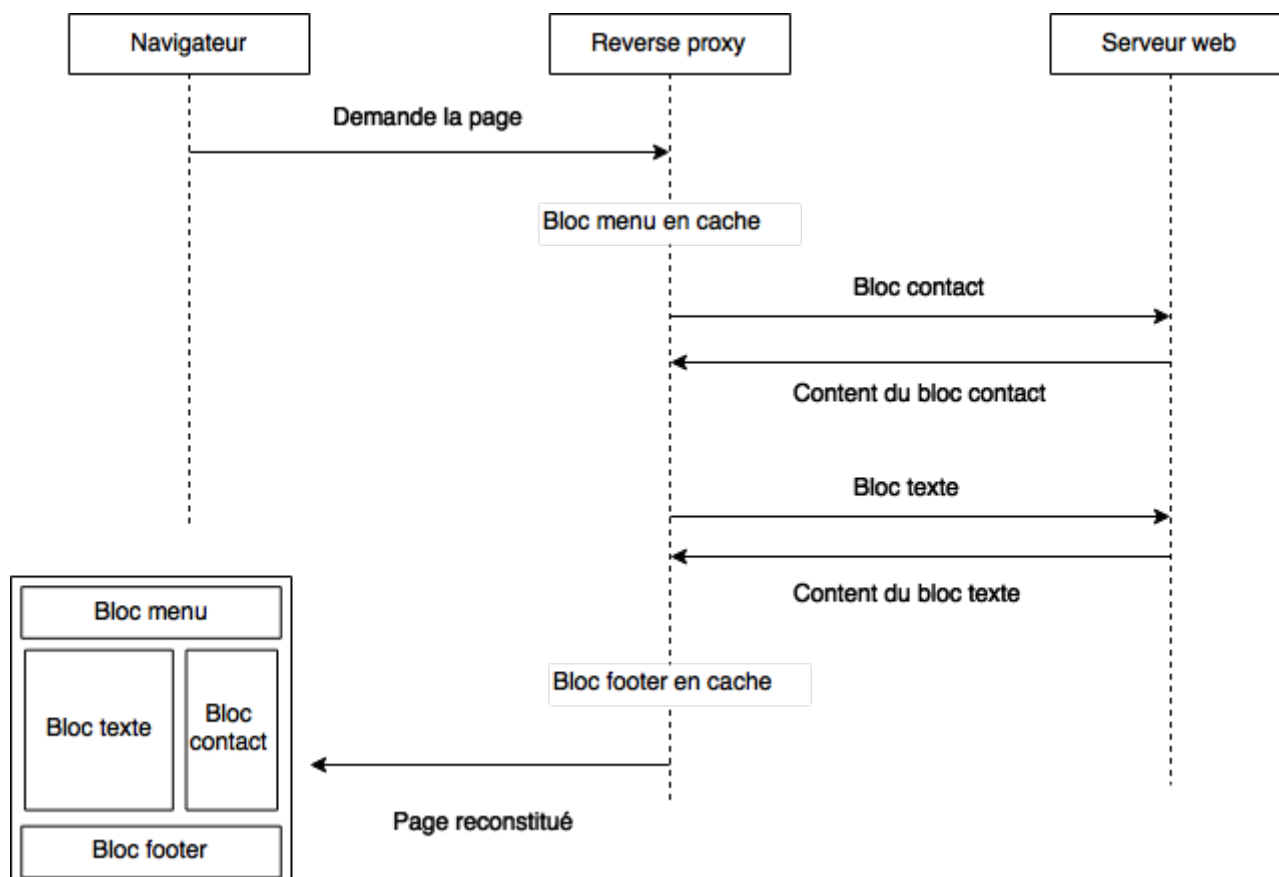


FIGURE 2.5 – Processus d'affichage d'une page qui utilise l'ESI

2.4.2 Modularités

Comme je vous l'expliquais en introduction la principale particularité d'Open Orchestra est sa modularité. Pour cela, les différents composants du CMS sont répartis dans différents bundles 3.1

- **open-orchestra-base-bundle** : Classes communes au back-office et front-office
- **open-orchestra-base-api-bundle** : Api d'Open Orchestra
- **open-orchestra-base-api-mongo-model-bundle** : Implémentation des models nécessaires à l'api pour MongoDB

- **open-orchestra-cms-bundle** : Logique du back-office
- **open-orchestra-front-bundle** : Logique du front-office
- **open-orchestra-display-bundle** : Logique d’affichage des blocs en Front-office
- **open-orchestra-media-bundle** : Médiathèque d’Open Orchestra
- **open-orchestra-media-admin-bundle** : Administration de la médiathèque d’Open Orchestra
- **open-orchestra-model-interface** : Interfaces des models utilisés par les autres bundles
- **open-orchestra-model-bundle** : Implémentation des models pour MongoDB
- **open-orchestra-user-bundle** : Gestion des utilisateurs
- **open-orchestra-workflow-function-bundle** : Système de workflow pour le back-office

Une application front-office n’a aucun intérêt à charger toute la logique du back-office ainsi grâce à ce découpage chaque application (front-office ou back-office) charge uniquement les composants qui lui est nécessaire.

De plus, cela permet de désactiver facilement une fonctionnalités. Par exemple, si un intégrateur n’a pas besoin de gérer des médias il peut alors désactiver la médiathèque en n’utilisant pas les bundles (**open-orchestra-media-bundle** et **open-orchestra-media-admin-bundle**).

Un autre avantage est la facilité de remplacement d’un composants. Par défaut Open Orchestra utilise MongoDB comme système de gestion de base de données, mais si un intégrateur veut utiliser un autre système comme Mysql alors il lui suffit d’écrire son propre **open-orchestra-model-bundle** qui utilise bien-sûr les différentes interfaces de **open-orchestra-model-interface** adaptés a Mysql.

Troisième partie

Symfony2

Chapitre 3

Concepts clés

Symfony2 est un framework MVC PHP open source développé par l'entreprise SensioLabs. Il dispose d'une très grande communauté et d'un large panel de modules (« Bundles ») développés par cette communauté.

Dans ce chapitre, je ne vais pas revenir sur les concepts de base et tous les composants de Symfony 2 (contrôleur, routing, etc) mais je vais vous présenter quelque concepts clé utilisé par Symfony 2 qui permettent justement le faible couplage d'Open Orchestra.

3.1 Bundles

Un bundle Symfony est un repertoire qui contient un ensemble de fichiers structurés (contrôleur, entité commande, listener, formulaire) qui permettent d'implémenter une fonctionnalité réutilisable dans n'importe quelle application Symfony2. Sous Symfony2 presque tout est découpé en bundle même le kernel du framework.

Les différentes classes et fichiers d'un bundle sont organisé selon l'architecture suivante :

```
Bundle/  
  Command/  
  Controller/  
  DependencyInjection/  
  EventListener/  
  Tests/  
  Ressources/  
    config/  
    public/  
    translations/  
    views/
```

- **Command** Contient les commandes du bundles
- **Controller** Contient les contrôleurs du bundles
- **DependencyInjection** Contient les classes qui peuvent importer la configuration de services, ajouter des passes de compilation, je reviendrai plus en détail sur ce repertoire par la suite
- **EventListener** Les listeners d'évènements du bundle
- **Tests** Tests unitaires et fonctionnels du bundle

- **Resources/config/** Configuration du bundle (routage, déclarations de service, etc)
- **Resources/views/** Templates (vues) utilisé dans le bundle
- **Resources/public/** Ressources web (js, css, images, etc)

Pour permettre une plus grande flexibilité un bundle peut définir sa propre configuration . En effet chaque bundle possède une classe **Extension** qui se trouve dans le répertoire **DependencyInjection**, cette classe permet de charger la configuration du bundle qui se trouve dans le répertoire **config**, de plus elle permet de manipuler cette configuration comme par exemple l'étendre avec la configuration générale de l'application (**app/config**) ou encore déclarer des services dynamiquement selon une configuration.

3.2 Conteneur de service

3.2.1 Services

Un service est simplement un objet qui réalise une fonction spécifique comme par exemple envoyer un mail. L'avantage d'avoir de nombreux services pour gérer les différentes fonctionnalités de son application est de permettre la séparation de son code et donc une plus grande flexibilité.

Avec Symfony2, pour créer un service il suffit de le déclarer dans un fichier de configuration (yaml, xml, php), en général ce fichier est appelé **services.yml** et de charger ce fichier lors du chargement du bundle (3.1). Lorsque l'on déclare un service, il faut indiquer sa classe et éventuellement les différents arguments dont a besoin le service.

Par exemple, le fichier de configuration yaml suivant déclare deux services, **service2** et **service2**. La classe **Service1** a besoin d'une instance de **Service2** pour cela, on peut utiliser la syntaxe **@service2** qui indique au conteneur de service de chercher un service appelé **service2** et de transmettre une instance au constructeur de **\verbService2**?

```
services:
    service2:
        class: "\MonBundle\Service2"
    service1:
        class: "\MonBundle\Service1"
        arguments: ["@service2"]
        tags:
            - { name: tag }
```

Pour finir, lorsque l'on déclare un service, il peut être taggé comme le montre l'exemple ci-dessus avec le paramètre **tags**. L'intérêt d'avoir des services taggés et de pouvoir récupérer tous les services qui possèdent un certains tag. c'est notamment utilisé par le pattern stratégie, que je vous ai présenté dans la section précédente, pour récupérer les différentes stratégies.

Ce type d'architecture est appelé « architecture orientée services » et n'est pas spécifique à Symfony ou même à PHP.

3.2.2 Conteneur

Les différents services sont gérés par le conteneur de service. Le conteneur de service est un objet qui gère tous les services, récupération d'un service, instanciation d'un service, etc.

Par exemple, supposons que nous avons deux services (service1, service2) et que le service 1 a besoin du service2. Lorsque l'on fait appelle au service1 le schéma 3.1 décrit le fonctionnement du conteneur de service pour récupérer le service.

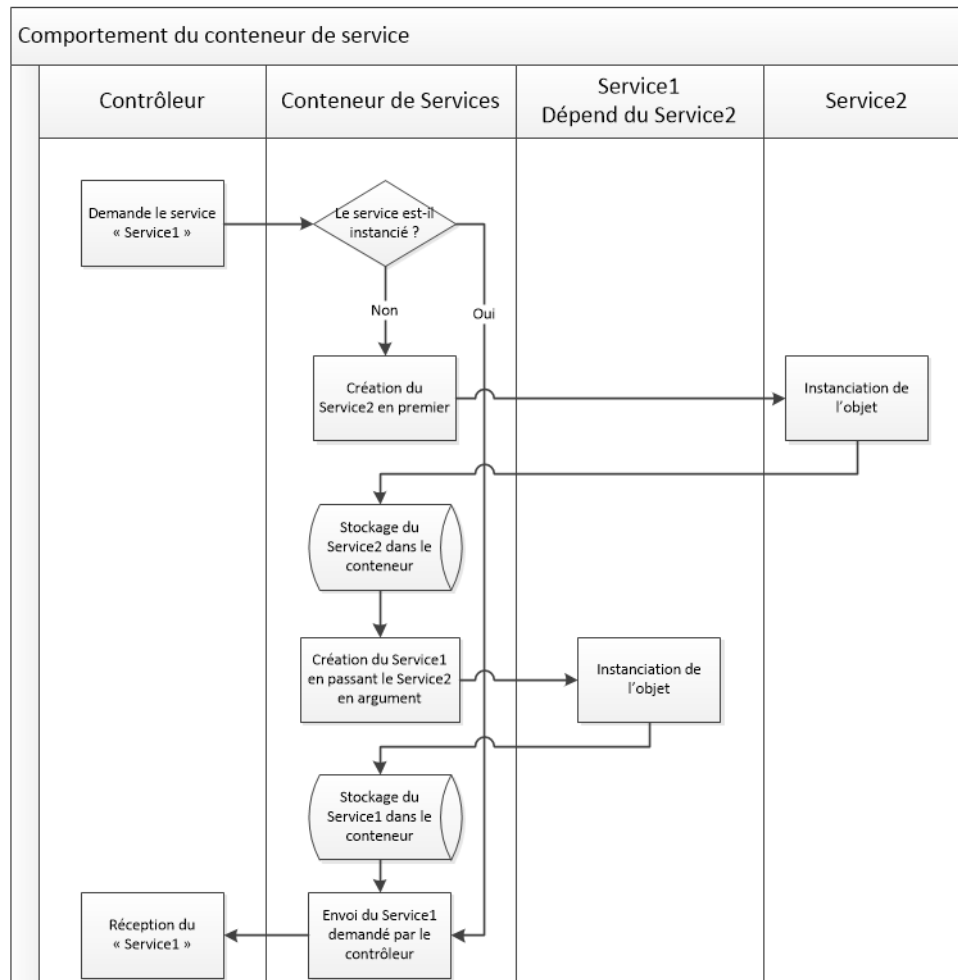


FIGURE 3.1 – Fonctionnement du conteneur de serveur lors de la demande d'un service

Les différents services sont partagés au sein d'un conteneur, c'est-à-dire qu'un service est instancié une seule et unique fois à la première demande de celui-ci. Ce qui permet à différents objets de manipuler la même instance d'un service tout au long de la requête.

Quatrième partie

Organisation du développement

Chapitre 4

Développement

Durant la période du stage, j'ai réalisé de nombreuses tâches sur le projet de différents types :

- Ajout de fonctionnalités aux projets
- Correction de bug
- Documentation du projet (en anglais)
- Re factorisation et Optimisation du code

Je ne vais pas vous énumérer toutes les tâches qui portent en général sur un point précis du projet, cela n'apporterait aucune valeur à ce rapport. Toutefois, je vais vous présenter l'organisation générale du développement, les techniques, les concepts et les outils utilisés.

4.1 Gestion de projet agile

4.1.1 Méthodes agiles

Le projet a été mené selon une approche agile. La gestion de projet agile repose sur un développement itératif qui consiste à découper un projet en plusieurs itérations appelé sprint. Les méthodes agiles définissent plusieurs valeurs fondamentales :

- La communication au sein de l'équipe
- La collaboration avec le client, celui-ci doit être impliqué tout au long du développement.
- L'adaptation au changement, c'est-à-dire que la planification initiale doit être flexible pour permettre l'évolution de la demande.

Il existe différents types de méthode agile (Scrum, XP, RAD...) qui possèdent leurs propres caractéristiques.

Pour le développement d'Open Orchestra, c'est la méthode Scrum qui est utilisée. Scrum définit trois rôles au sein d'une équipe :

- Le « Product Owner (PO) » qui porte la vision du produit
- Le « Scrum Master » est le responsable de la mise en œuvre de la méthode, il doit s'assurer que cette dernière est correctement appliquée
- L'équipe de développement qui est chargée de transformer les besoins exprimés par le Product Owner

La réalisation d'un projet utilisant la méthode est rythmée par différents événements :

Tous d'abord, le sprint qui représente une période de courte durée, de une à quatre semaines, durant laquelle l'équipe effectue un nombre de tâches définies à l'avance.

La réunion de planification où l'équipe répond à deux questions, « Quoi ? » c'est-à-dire les tâches du backlog¹ qu'elle réalisera au prochain sprint et « Comment ? » c'est à ce moment que l'équipe estime les tâches choisies au « Quoi »

Le « daily scrum » est une réunion réalisée quotidiennement. Durant cette réunion chaque membre de l'équipe indique les tâches qu'il a réalisées depuis le dernier daily et celles qu'il va effectuer jusqu'au prochain daily et pour finir, les difficultés qu'il a ou pense rencontrer. Cette réunion permet à tous les membres de connaître les tâches de chacun afin de s'entraider et d'anticiper plus facilement les obstacles.

Pour finir à la fin du sprint, les membres de l'équipe se réunissent pour la revue de sprint durant laquelle les différentes tâches réalisées durant le sprint sont présentées et validées. Puis pour la rétrospective qui permet de mettre en évidence les points positifs et les actions à mettre en place pour améliorer le prochain sprint.

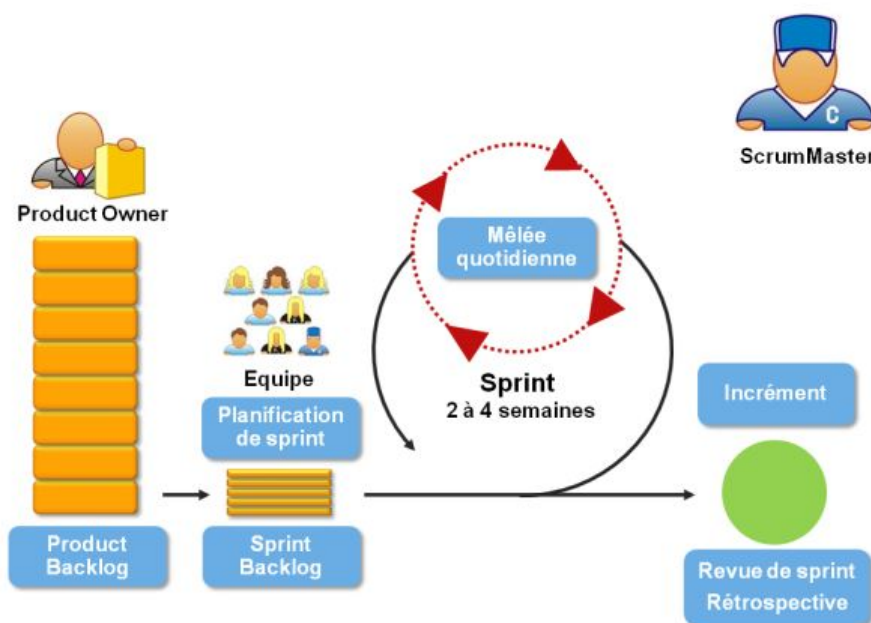


FIGURE 4.1 – Schéma décrivant la méthode Scrum

Source : <http://www.agiliste.fr/fiches/guide-demarrage-scrum/>

4.1.2 Scrum avec Open Orchestra

Au sein du projet Open Orchestra, la méthode Scrum est appliquée. Ainsi des sprints de une semaine avec un daily scrum tous les jours à midi. De plus, tous les mercredis ou mardis selon les disponibilités une « cérémonie » qui comprend la réunion de planification et la rétrospective.

1. Le backlog est un ensemble de fonctionnalités ou de tâches nécessaires pour la réalisation satisfaisante d'un projet

Concernant, la revue de sprint (validation des tâches du sprint) celle-ci n'est pas faite durant une réunion défini mais tout au long du sprint par le product owner.

Pour faciliter la mise en place de la méthode Scrum, l'équipe utilise différents outils. Tout d'abord Skype et Slack² pour la communication.

Et enfin Trello pour organiser les tâches, en effet l'équipe possède un **board** avec différentes colonnes pour organiser les tâches :

- **Backlog** : le backlog qui contient les différentes tâches à effectuer pour le projet, les tâches du backlog sont organisées en quatre colonnes (Nice to have, Good, Great, Must have) selon la priorité de la tâche.
- **Support** : Les demandes effectuées par les différentes équipes d'intégrateur d'Open Orchestra au sein d'Interakting.
- **Proposition** : Les différentes propositions d'amélioration ou de re-factorisation du code faites par les membres de l'équipe.
- **Bugs** : Les différents bugs rencontrés durant le développement
- **Todo** : Les tâches à effectuer durant le sprint en cours.
- **Doing** : Les tâches actuellement en développement.
- **Blocked** : Les tâches bloquées pour différentes raisons (manque de précision de la tâche, bug empêchant la réalisation de la tâche)
- **ToDeploy** : Tâches réalisées qui sont prêtes à être déployées sur le serveur d'intégration.
- **ToValidate** : Tâches du sprint en attente de validation par le product owner.
- **Failed** : Tâches du sprint non validées par le product owner
- **Done** : Tâches du sprint validées par le product owner

Les colonnes ToDeploy, Done sont uniques à un sprint, il y a donc une colonne ToDeploy, Done différente pour chaque sprint.

Par exemple, une tâche suit un processus, présenté par le schéma 4.2, bien particulier entre l'intégration de la tâche dans le sprint et sa validation par le product owner. Ce processus permet de suivre facilement l'avancement ou encore les différents problèmes rencontrés sur les différentes tâches par tous les membres de l'équipe.

2. Slack est une plateforme de communication réalisée en 2014 qui permet d'intégrer facilement différents outils de service en ligne telle que github, trello, dropbox, google drive, ...)

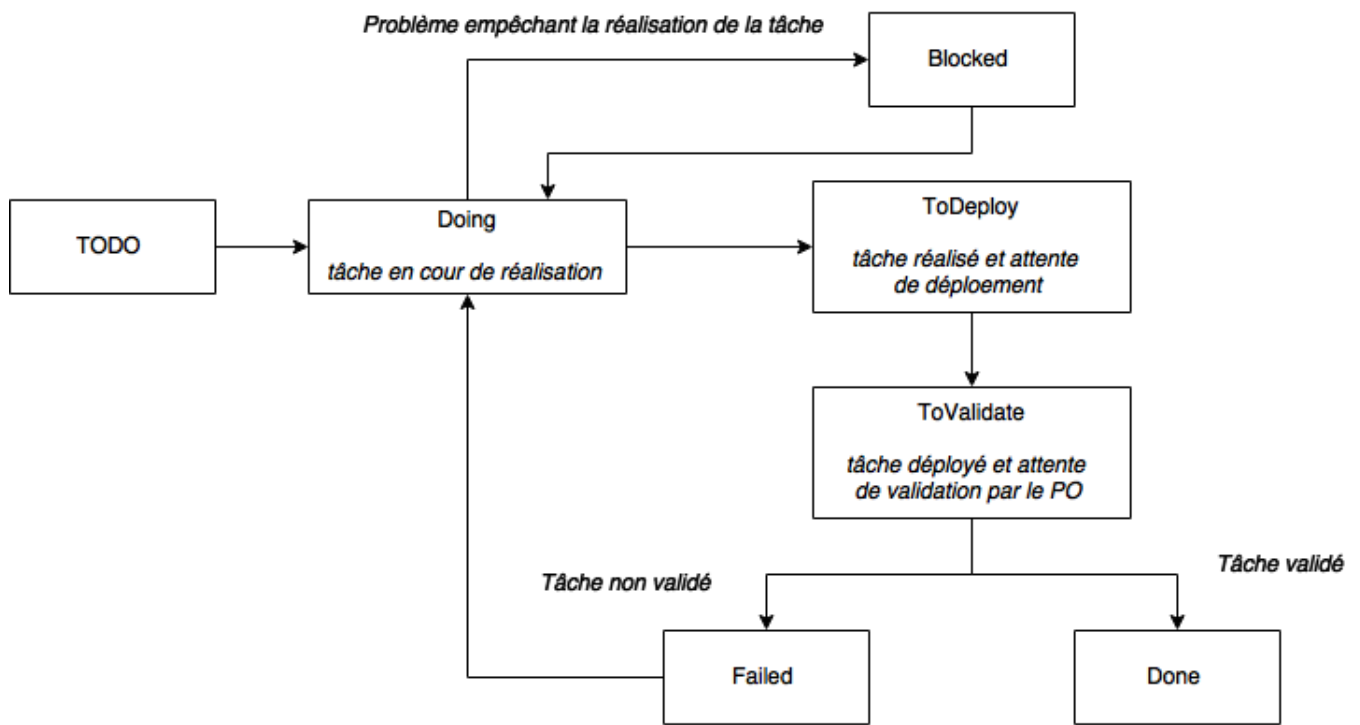


FIGURE 4.2 – Schéma représentant la progression d'une tâche lors d'un sprint

4.2 Intégration continue

Pour son développement Open Orchestra utilise les principes de l'intégration continue.

L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

Source wikipedia

L'utilisation de ces pratiques apporte de nombreux avantages, tout d'abord comme il est indiqué dans la définition cela permet de minimiser les régressions mais aussi, d'avoir à tout moment un produit (logiciel, site, etc) utilisable.

La mise en place de cette technique nécessite différents pré-requis :

- Dépôt unique de code source versionné
- Automatisation des tests
- Un environnement similaire à celui de production
- Automatiser le déploiement

4.2.1 Git

Dans le cadre d'Open Orchestra, le code source est versionné avec git³ et utilise GitHub qui est un service web permettant d'héberger un dépôt git. Comme je l'expliquais lors de la

3. Git est un logiciel de versionning décentralisé, c'est-à-dire qu'il n'existe pas un dépôt unique mais un dépôt local pour chaque développeurs.

présentation d'Open Orchestra, ce dernier est découpé en plusieurs bundles et donc chaque bundle possède son propre dépôt sur GitHub.

Open orchestra étant toujours en version bêta, il n'y a pas de realease à proprement parlé. Ainsi le workflow de développement sur github est simplifié car il y a qu'une seule branche stable, la branche master. Les autres branches sont des branches temporaires d'ajout de fonctionnalités ou de correction de bugs. Ce type de workflow est appelé **GitHub flow** et est représenté par le schéma 4.3.

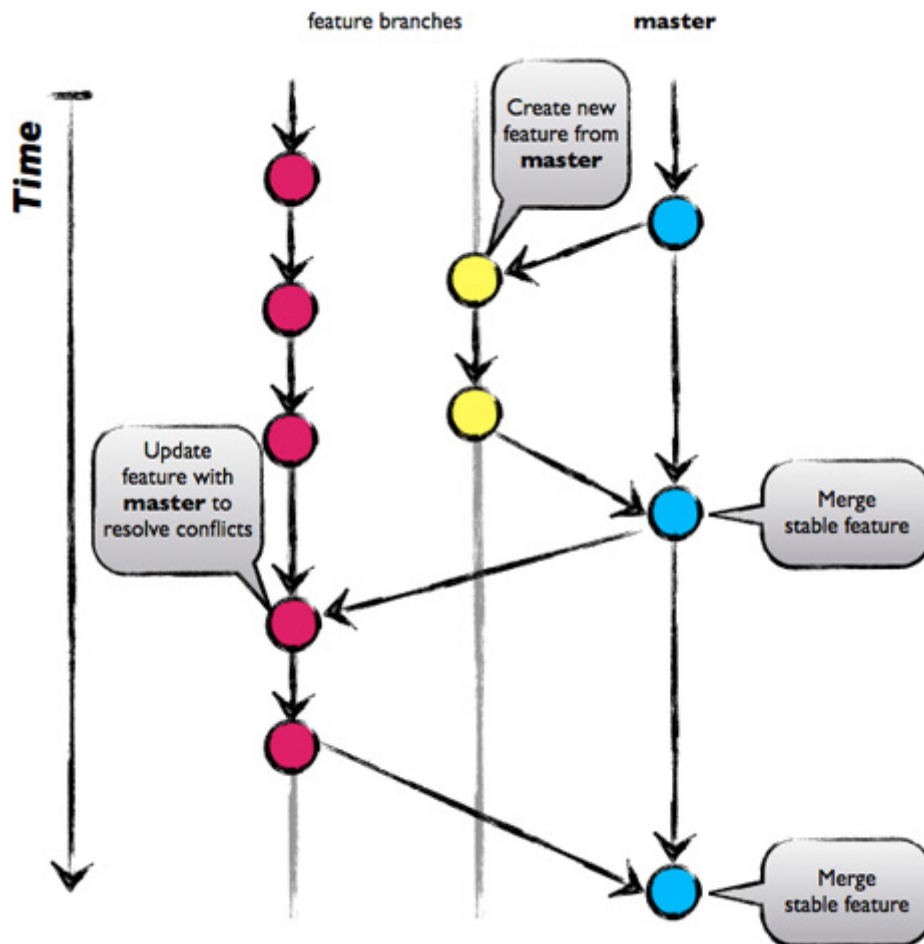


FIGURE 4.3 – GitHub flow

Source : <http://nicoespeon.com/fr/2013/08/quel-git-workflow-pour-mon-projet>

Ainsi avec ce type de workflow, la modification du code source que ce soit pour corriger un bug ou ajouter une fonctionnalité s'effectue toujours de la même manière :

1. Mettre à jour sa branche master en local
2. Créer une nouvelle branche en local à partir de master
3. Effectuer les modifications dans cette branche
4. Pousser sa branche sur GitHub (origin)
5. Vérifier que les tests fonctionnent, je reviendrai plus en détail sur les tests par la suite

6. Ouvrir une pull-request, sur GitHub une pull-request consiste à faire une proposition de modifications, à partir de cette pull-request les différents membres peuvent commenter, valider ou non la modification.

Une fois la pull-request validé, elle peut être mergé avec master, c'est-à-dire inclure la modification dans la branche master.

Comme je l'ai précisé en début de section pour le moment Open Orchestra ne possède pas de branche realease. Toutefois, le projet Open Orchestra est déjà utilisé par des intégrateurs pour réaliser des sites à distanciation de différents clients, ils ne peuvent donc pas utiliser directement la branche master, ils leur faut des points d'arrêts.

Pour cela, sauf cas particulier (bug bloquant, besoins particulier) la branche master est **taggée** toutes les deux semaines environ ainsi cela permet à l'équipe d'Open Orchestra de continuer le développement du projet et aux intégrateurs d'avoir une version fixe de la branche master qu'ils peuvent mettre à jour au moment le plus opportun pour eux. Lorsque Open Orchestra ne sera plus en bêta cela changera la version 1 aura sa propre branche dans laquelle aucune nouveauté sera intégré hormis les corrections de bug.

4.2.2 Travis CI

Lorsqu'une branche est poussée sur l'un des dépôts GitHub du projet, les tests sont automatiquement lancés. Pour cela, les différents dépôts du projet intègrent l'outil **Travis CI**. Travis CI est un service qui est lié à GitHub, il permet de déclencher un certain nombre de tâches lors d'événements sur le dépôt. Dans le cas d'Open Orchestra Travis CI permet de lancer les tests unitaires après chaque **push** sur un dépôt. Un des avantages de travis est qu'il se configure très facilement grâce à un fichier `yaml`.

Par exemple le fichier de configuration travis d'un des dépôts d'Open Orchestra

```
language: php

php:
  - 5.4

install:
  - composer install --prefer-dist --no-progress

script: ./bin/phpunit
```

Indique que le langage utilisé est `php` et que les tests doivent être exécutés avec la version 5.4 de `php` et enfin avant d'exécuter `phpunit` pour lancer les tests il est demandé à travis de faire un `composer install` pour récupérer les dépendances du projet. Ensuite, à partir de ce fichier de configuration Travis CI est capable de créer l'environnement pour effectuer les tests, il est bien sûr possible d'aller plus loin dans la personnalisation de l'environnement en allant chercher d'autres dépendances avec `apt-get install` par exemple ou encore de demander à être notifié par email si les tests ne passent pas.

4.2.3 Déploiement

Le déploiement sur un serveur de production n'est pas toujours une tâche facile (uploader le code, mettre à jour les dépendances, lancer les migrations de la base de données, vider le cache, etc) or pour permettre une intégration continue il est nécessaire de déployer régulièrement son application, il semble donc nécessaire d'automatiser cette tâche, pour cela il existe de nombreux outils.

En ce qui concerne Open Orchestra nous utilisons **Capistrano**. **Capistrano** est un outil écrit en Ruby qui permet d'exécuter des scripts sur un ou plusieurs serveurs. Il est principalement utilisé pour faire du déploiement, mais il permet aussi de créer ses propres tâches, par exemple pour Open Orchestra nous l'utilisons aussi pour lancer des tests de charge, avec Jmeter, directement sur le serveur d'intégration.

Un autre avantage de **Capistrano** est qu'il permet de revenir facilement sur une version précédente de votre application grâce à une architecture de dossier que met en place **Capistrano** dans le dossier de votre application sur serveur :

```
/app/  
  releases/ Contient un dossier pour chaque déploiement estampillé  
  current   Lien symbolique pointant vers un dossier de release spécifique  
  shared/   Contient les données partagées entre chaque déploiement
```

Ainsi, si un déploiement provoque des erreurs, il suffit de faire pointer le lien symbolique **current** vers une autre version.

4.3 Qualité

L'équipe d'Open Orchestra essaye de fournir un produit de qualité que ce soit au niveau du code, du couplage ou encore du fonctionnel. Pour cela différentes pratiques, outils sont mises en place.

4.3.1 Qualité du code

Tout d'abord au niveau du code Open Orchestra utilise le standard PSR-2. PSR-2 est un ensemble de recommandations sur le style et l'organisation du code dont voici les plus importantes :

- La tag de fermeture `\>`. doit être omis de tous les fichiers contenant uniquement du PHP
- Tous les fichiers PHP doivent se terminer par une ligne vide.
- La visibilité doit être déclarée sur toutes les propriétés et méthodes.
- L'ouverture des accolades pour les classes, les méthodes doivent figurer sur la ligne suivante, les accolades de fermeture doivent figurer sur la ligne suivante après le corps de la classe.
- Il doit y avoir un espace après la structure clé de contrôle et pas d'espace après la parenthèse ouvrante et avant la parenthèse fermante
- Il doit y avoir un espace entre la parenthèse fermante d'une structure de contrôle et de l'accolade ouvrante
- L'accolade fermante d'une structure de contrôle doit être sur la ligne suivante après le corps

Utiliser cette recommandation permet une harmonisation du code et ainsi il est plus simple pour toute l'équipe de relire et reprendre le code d'un autre membre, mais aussi pour tous les développeurs qui utilisent Symfony 2 puisque ce dernier utilise aussi les recommandations PSR-2

Lors de revue de code des pull-request, il n'est pas toujours aisé de détecter l'utilisation des bonnes pratiques ou non. Il existe des outils qui permettent d'analyser le code automatiquement et de détecter les oublis lors de la revue.

Pour le projet Open Orchestra deux outils sont utilisés :

Tous d'abord, Code Climate s'interface avec les dépôts c'est-à-dire que lors d'un push le code est analysé.

Code Climate permet de détecter la duplication de code, la couverture des tests, la qualité globale du projet. De plus, il fournit une note qui indique la qualité globale du dépôt (annexe 4.4). Le second outil est SensioInsight qui fonctionne de la même façon que Code Climate mais en supplément des vérifications sur les bonnes pratiques de Symfony 2.

4.3.2 Test

Le plus grand intérêt des tests sont qu'ils permettent de réduire les régressions lorsque l'on applique des modifications à une application. En effet, lorsqu'une application devient assez conséquente il n'est pas toujours aisé d'anticiper toutes les répercussions lorsque l'on modifie une portion de code. Les tests permettent donc de tester l'ensemble de l'application et ainsi vérifier que cette dernière fonctionne toujours correctement après les modifications effectuées. De plus, ils sont en général gage de qualité

Il existe deux grandes familles de test, Tous d'abord les tests unitaires qui permettent de vérifier que chaque méthode et fonction fonctionne correctement, les tests doivent être indépendant au maximum les uns des autres pour cela il existe les mocks qui sont des objets qui permettent de simuler le comportement d'objets « réels ». D'autre part les tests fonctionnels qui eux contrairement aux tests unitaires vérifient les fonctionnalités de bout en bout. c'est-à-dire que du routing jusqu'à la vue en passant par le modèle et le contrôleur.

Avec Symfony, il est assez facile de créer des tests puisqu'il intègre la bibliothèque PHPUnit qui fournit un framework de test complet, de plus pour les tests fonctionnels Symfony propose une classe `WebTestCase` qui permet d'initialiser le noyau de Symfony, cette classe fournit notamment un objet `client` qui permet d'effectuer des requêtes HTTP sur son application et un crawler pour vérifier le résultat retourné par la requête.

En plus des différents tests au niveau du code serveur, Open Orchestra possède aussi des tests de charges réalisés avec Jmeter. Jmeter est un logiciel qui permet de tester les performances d'une application en simulant des utilisateurs. Concrètement pour effectuer un test, il suffit décrire un scénario fonctionnel c'est-à-dire un enchaînement d'action (visiter une url, remplir un formulaire, etc) que Jmeter devra effectuer lors du test et un nombre d'utilisateur qui réaliseront ce scénario.

Lorsque que Jmeter effectue un test, il fournit un tableau de résultat avec différentes données telles que le temps de réponse d'une url, le pourcentage d'erreur, etc) ainsi cela permet de voir comment réagissent les différentes fonctionnalités de son application avec un nombre important d'utilisateur.

4.3.3 Qualité fonctionnel

Un point important d'une application est son ergonomie, concernant le projet Open Orchestra des consultants fonctionnels ont émit différentes propositions afin d'améliorer l'ergonomie. Les différentes propositions sont recueillies au sein d'un board Trello prévu à cet effet avec différentes colonnes (Propositions, Propositions validés par un expert fonctionnel, Propositions acceptées par Open Orchestra).

Ainsi lorsqu'une proposition est validée par un expert fonctionnel, nous en parlons en cérémonie pour savoir si elle doit être ou non intégrée au backlog du projet.

Bilan

Durant le stage, ce fut un réel plaisir de travailler avec l'équipe du projet Open Orchestra et plus en général avec les différentes personnes d'Interakting que j'ai côtoyé à l'agence de Caen ou de Paris lors de mes déplacements.

J'ai été directement intégré au projet comme aurait pu l'être un autre développeur de l'agence ce qui a été très constructif pour moi et important de voir que l'on me faisait confiance.

Travailler sur un projet Open Source d'un point de vue plus orienté éditeur que intégrateur (réalisation de site pour des clients) a été très intéressant et motivant. Cela m'a permis de voir et mettre en pratique la méthode agile Scrum sur un projet concret.

D'un point de vue technique, j'ai pu comprendre et approfondir des concepts avancés de Symfony2. Plus généralement, j'ai appris énormément sur les bonnes pratiques (test, psr2, principe SOLID⁴) mais aussi de découvrir et d'utiliser de nouveaux outils/langages (Travis, Ansible, Vagrant, CoffeeScript, Jmeter). De plus, durant mon stage j'ai assisté à des formations sur différents sujets (Git, Travis, Scrutinizer) proposé tous les jeudis de 13h à 15h par des développeurs de l'entreprise.

Pour conclure, ce stage a largement répondu à mes attentes que ce soit sur le projet en lui-même ou sur les points techniques.

4. Les cinq principes de bases de la programmation orientée objet :

- Single responsibility : Une classe doit avoir une seule responsabilité
- Open/closed : Une classe doit être ouverte à l'extension, mais fermée à la modification
- Liskov Substitution : Une instance de type A doit pouvoir être remplacé par une instance de type B tel que B implémente A
- Interface segregation : Préférer plusieurs interfaces spécifiques plutôt qu'une seule interface générale
- Dependency Inversion : Il ne faut pas dépendre des implémentations, mais des abstractions

Annexes

Liens utiles

- **Projet GitHub** : <https://github.com/open-orchestra>
- **Site de présentation d'Open Orchestra** : open-orchestra.com
- **Démonstration d'Open Orchestra** : open-orchestra.com/demo

4.4 Aperçus






Service	Badge
Travis	
CodeClimate (quality)	
CodeClimate (coverage)	
Sensio Insight	
VersionEye	

FIGURE 4.4 – Badges de qualité d'un bundle Open Orchestra

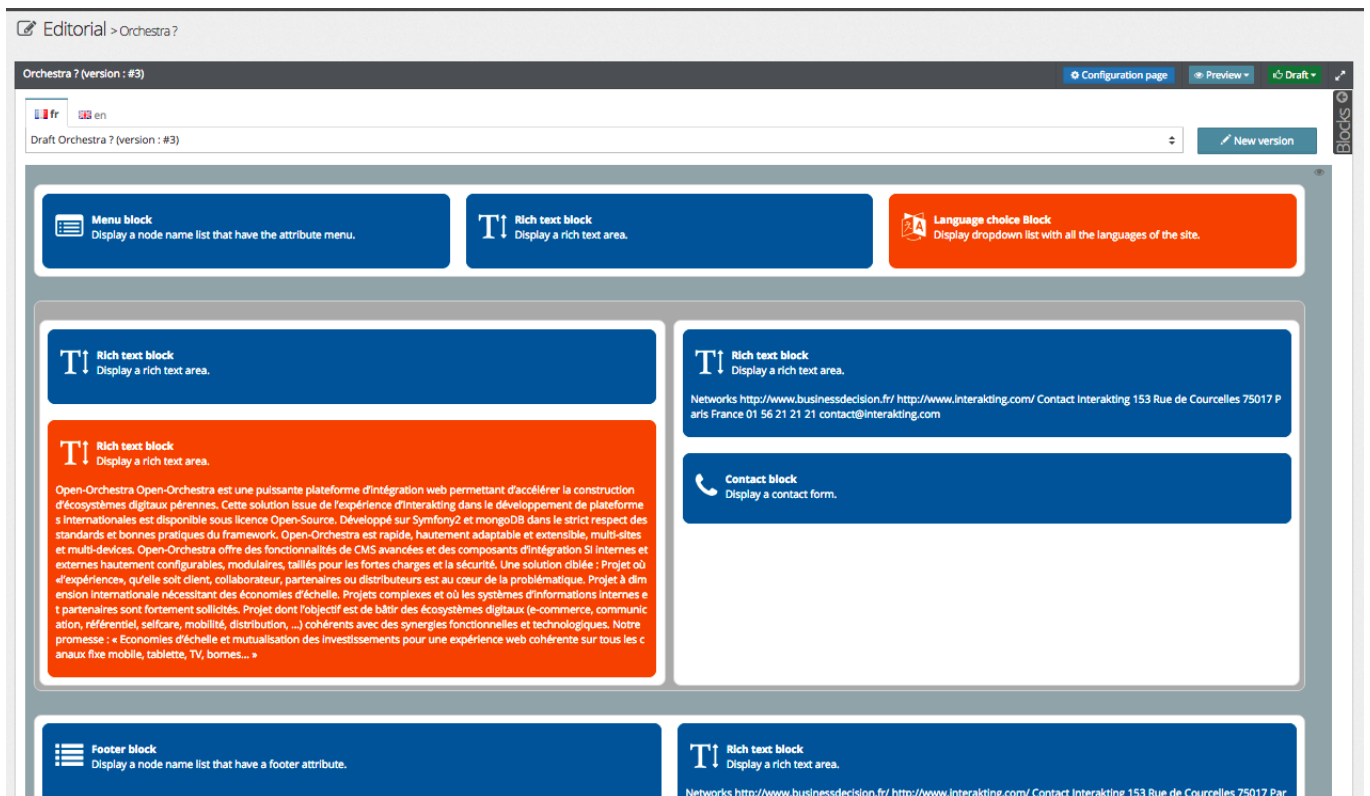


FIGURE 4.5 – Exemple de création d'un node avec des blocs



FIGURE 4.6 – Liste des blocs disponible par défaut