Notes sur le Code D'ASKER

Le code d'ASKER peut paraitre bien organisé et solide au premier regard, cependant ce document devrait potentiellement vous permettre d'appréhender le code (contenu-structures-principes) plus rapidement, du moins je l'espère. Le ton du document ne se veut pas académique ou procédurier, j'espère qu'il restera suffisamment clair.

1- Technologies et « Framework » Utilisés

Le code a subi plusieurs modifications majeures pour être adapté à la plateforme d'enseignement CLAROLINE CONNECT (plateforme co-produite par L'Université de Lyon et du Québec (Montréal ?)), puis en être séparé. Le code, au moment où j'écris ces lignes, début juin 2016, a été techniquement re-séparé de la plateforme.

Si vous reprenez le code de cette date, vous devriez alors vous retrouver avec une application basée sur le Framework Symfony. Ce Framework assez connu dans le web dispose d'une architecture MVC et d'une séparation en « bundle », autrement dit des blocs, des « briques » d'application représentant du code applicatif particulier. Il est donc fortement conseiller de se renseigner un peu sur ce Framework pour mieux comprendre le code. Commencer par poser un « environnement » dans lequel une application Symfony fonctionne est probablement le premier pas. Concernant la version de Symfony à utiliser, je conseillerais la 2.8 et inférieure. Lorsque j'ai repris le code, certains éléments utilisés par ASKER étaient déjà en « DEPRECATED » en 2.5, et Symfony version 3 casse la rétrocompatibilité avec les applications antérieures. J'ai normalement déjà réglé quelques soucis pour passer en version 3, mais il en reste deux ou trois concernant l'interface php/base de données (bd) qui ne sont pas réglés. De plus, il est encore beaucoup plus facile de trouver de l'aide déjà résolue sur la version 2.7 que la version 3…

L'application ne se contente pas cependant d'utiliser Doctrine et ses bundles. L'interface utilise amplement les patterns twig et surtout Angular et tout ce qui concerne les contrôleur Angular. En plus d'Angular, ASKER utilise une bibliothèque php que je n'apprécie que moyennement du fait de son fonctionnement : JMSSerializeBundle. Ce dernier utiliser les commentaires php pour ajouter des informations sur comment sérialiser des objets PHP pour la base de données ou pour l'application client.

Pour les parties suivantes, je risque d'utiliser du jargon des différents Framework Symfony. Il est donc fortement conseillé de se familiariser un petit peu avec si vous ne connaissez absolument pas ce dernier.

2- Ce qu'il manque pour déploiement

L'application contient deux bundles « importants ». Le premier, AppBundle, est un bundle au nom par défaut qui pour le moment défini une entité (objet php) utilisateur pour l'application. Cette entité utilise un bundle créé par FOS (Friends of Symfony) et qui propose

une gestion de la connexion utilisateur. Ce bundle, nommé user-bundle, propose également de gérer les différents rôles utilisateurs. Pour l'instant, il existe une hiérarchie de 4 rôles. Deux rôles fonctionnels (utilisateur ou créateur de contenu), et deux rôles Administrateurs. Les rôles fonctionnels sont sous les noms de ROLE_USER et ROLE_WS_CREATOR. Les rôles Administrateurs ont la totalité des droits que pourrait avoir un ROLE_WS_CREATOR et sont nommés ROLE_ADMIN et ROLE_SUPER_ADMIN. La hiérarchie est décrite dans le fichier « app\config\security.yml », et les accès sont gérés sous la catégorie access_control du même fichier, via un système d'expressions régulières.

Actuellement, seul l'utilisateur avec un rôle au moins ROLE_WS_CREATOR a accès à la génération des exercices à partir d'un modèle. Il manque en effet un moyen de donner aux utilisateur normaux l'accès à certains modèles. L'idée probablement la plus simple à mettre en place est un système de groupe. Le bundle Friend-Of-Symfony/user-bundle propose d'ailleurs déjà cette fonctionnalité. En fonction du groupe, ou des groupes, l'utilisateur a accès ou non à certains modèles. Le système fonctionnerait alors ainsi :

Côté ROLE_WS_CREATION :

- Créer un groupe, supprimer un groupe, créer un lien d'invitation au groupe
- Associer certains modèles créés au groupe

Côté ROLE_USER :

- En cliquant sur le lien, il s'inscrit au groupe en question
- Dans « Mes tentatives » (probablement à renommer mes exercices), il peut utiliser les modèles en question pour générer ces derniers.

Concernant la gestion des utilisateurs ; pour l'instant la fonction de validation des utilisateurs par e-mail de confirmation est désactivée ; il s'agit encore ici d'une option du bundle user-bundle mais qui nécessite encore quelques paramétrages -et test- concernant le contenu et l'envoie des mails.

3- Concernant la réutilisabilité du code

Un des principaux soucis du code n'est pas sa fonctionnalité, mais plutôt dans la facilité avec laquelle on peut le modifier. Le code est « clair » dans le sens où il parait plutôt bien rangé et respectant le modèle MVC prôné par Symfony. Cependant... S'il respecte plus ou moins (je reviendrais là-dessus) le modèle MVC, il n'utilise absolument pas les possibilités offertes par le modèle objet de php. Ou plutôt, il l'utilise, mais en partie, avec des gros défauts. Ces « gros défauts » étant une mauvaise utilisation de l'héritage en php dans le but de faciliter les accès lors de la génération du html et des communications JSON (pour l'interface) et sérialisation dans la BD (création de requêtes SQL), il est très compliqué de modifier certains éléments sans devoir changer l'ensemble de la chaine… Par exemple, pour rajouter un modèle d'exercice (comme un nouveau type de QCM), ce n'est pas moins de 128 fichiers à modifier, plus la création de 4 réservés à définir la structure/génération de notre nouveau type d'exercices.

L'erreur dans l'utilisation de l'objet (j'espère que vous vous connaissez un peu en code orienté objet au passage ^^) est pourtant très simple, et malheureusement souvent vue. Supposez un objet CommonModel, dont hérite chacun des nouveaux Modèles d'Exercices (QCM, QROC). De là :

- Chaque classe objet fille de CommonModel porte le même nom : Model. Chaque fichier Model.php est ensuite contenu dans un dossier portant le nom du Modèle d'Exercice.

- Dans la classe CommonModel, au début, est listé l'intégralité des noms possibles pour les modèles d'exercices, sous la forme de constantes. On retrouve par exemple const OPEN_ENDED_QUESTION = « open-ended-question »
- Dans le Service ExerciseModelService, qui va faire la liaison avec l'interface, on intègre chaque modèle php en renommant la classe Modèle contenue à l'intérieur. Le service utilise ensuite une instruction de type switch sur le nom du modèle d'exercice pour savoir lequel générer.
- Le nom avec les tirets (« open-ended-question » par exemple) est d'ailleurs utilisé à de nombreux autres endroits. Tout d'abord l'objet php représentant l'Exercice, qui va disposer de « open-ended-question » dans son type, puis à de nombreux endroits de l'interface, principalement pour servir de filtres lorsqu'on veut afficher certains types d'exercices...

Le dernier point que je cite au-dessus est ensuite directement lié à un autre élément que je considère également comme un défaut du code. Si vous regardez le contenu de \src\AskerBundle\Resources\public\frontend\js\modules\teacher\controllers.js, vous trouverez dans « $scope.resourceContext » un autre élément qui devrait potentiellement vous étonner. Il s'agit de la description JSON de chacune des classes php utilisées par l'interface avec des valeurs par défaut…

En fait, ces descriptions sont utilisées lorsqu'un utilisateur veux faire un nouvel objet de ce type. Lorsque l'utilisateur veut créer un nouveau modèle d'exercice de type open-ended-question par exemple, il va cliquer sur le bouton correspondant de l'interface, qui va envoyer une requête POST utilisant le JSON présent dans le fichier controller.js. De là, le contrôleur du serveur gérant les requêtes POST va utiliser un switch sur le type envoyé par l'interface et ainsi retrouver le bon type d'entité en interne… Contrôleur qui contient donc aussi un switch sur le type d'exercice (class ExerciseObjectFactory).

La même chose a été faite pour ce qui est des Exercices eux-mêmes. Il existe un fichier CommonExercise qui va lister l'ensemble des exercices existant dans l'application. Une part de confusion se rajoute en plus par le fait que certains exercices et certains modèles d'exercices ont été nommé de manières identiques. Par exemple un modèle d'exercice QROC (open-ended-question) aura la même chaine de caractère pour représenter son type que l'exercice de type QROC lui-même.

Il n'y a probablement pas de manière de changer le code sans le modifier en profondeur mais il est possible de proposer quelques pistes pour aider à retravailler ce dernier. Tout d'abord, les éléments qui ne changeront pas sont probablement les entités au sens de Symfony et les codes génériques (classes php n'ayant aucune référence aux exercices/ressources/modèles d'exercices particulier – ce qui fait quand même un bon 50% des fichiers). Pour le reste, il me semble important de pouvoir regrouper tout ce qui a rapport à un même type d'exercice au même endroit. Pour cela, je pensais principalement à deux solutions possibles ; la deuxième demandant toutefois de faire la première dans une certaine mesure :

1- Une séparation par type de fichier

Il s'agit de la solution « facile » dans le sens où elle se rapproche le plus de la solution déjà mise en place ici. Les sous-dossiers sont principalement déjà séparés par fonctionnalité mais la généricité est

mise à mal. Dans les deux solutions, c'est probablement par cela qu'il faut commencer. De mon point de vue, les contrôleurs javascript n'ont besoin pour leur requête POST de n'envoyer qu'un élément (ie sans avoir à transmettre les valeurs par défaut de création de l'objet) : le type de l'objet. Etant donné la nature de Symfony, le seul fichier ayant potentiellement ensuite besoin de savoir quel objet créer (et donc posséder un « switch » sur les types potentiel à créer) est le contrôleur récupérant ces requêtes POST. De plus, il serait de bon ton de faire en sorte que ce contrôleur utilise un fichier de configuration à part pour reconnaitre les classes en question. Le fichier en question pourrait être un simple fichier texte lu par le contrôleur recevant les requêtes POST à sa création. Une fois ceci fait, cela permettrait de se débarrasser des autres switchs sur les types des classes filles dans les autres php, qui ne servent essentiellement qu'à cela. Ainsi, l'utilisateur aurait juste besoin de modifier le code à 5 endroits pour ajouter un modèle d'exercice :

- Le contrôleur js pour ajouter le nom du type
- Le fichier de configuration pour ajouter le nom du type
- L'interface pour ajouter les boutons nécessaires coté apprenant/coté enseignant
- La classe modèle et les informations de générations pour le modèle
- Créer le service correspondant pour récupérer la classe dans la base de données

-

2- Un sous-dossier pour tous les éléments d'un même modèle

La solution précédente devrait déjà faciliter grandement la tâche de quelqu'un voulant modifier le code, mais va un tout petit peu plus loin. Il s'agirait de regrouper tout ce qui est utile à un modèle/une ressource dans un même sous dossier.

Il existerait alors pour chaque modèle un sous dossier contenant : classes pour utiliser le modèle (factory-entity) ; « partials » html à intégrer dans les modèles twigs ; fichier js contenant le contrôleur pour le nouveau modèle ; services.yml et config.yml pour le nouveau modèle. Dans le dossier contenant l'ensemble des sous dossiers se trouverait alors le fichier énumérant tous les modèles possibles. Ce fichier serait alors lu par les contrôleurs Symfony en ayant besoin, c'est-à-dire ceux s'occupant de l'interface. Les modèles twigs pour récupérer chaque partial et l'intégrer dans son interface, et le contrôleur recevant les requêtes POST en JSON.

Cette deuxième solution n'est pas plus optimale que la première, mais regrouperait l'ensemble des fichiers pour un même exercice en un seul endroit. Dans les deux cas, faire bien attention à différencier un modèle d'un certain type et une ressource du même nom. Si certaines ressources ne sont utilisées que dans un modèle (QCM, QROC), d'autres (textes et images) sont utilisées dans plusieurs. Il convient

alors de se demander les ressources dépendantes d'un seul modèle sont à placer dans le même sous dossier ou si il faut garder le clivage actuel entre ressources et modèles. Quoi qu'il en soit, bien je pense que donner un nom différent à la ressource et au modèle (ce qui n'est pas le cas actuellement) serait une bonne chose.