Collin Maxime

stage CARTAHU

Manuel d'Administration

Explication détaillées de ManageChart pour pouvoir modifier les fichiers sources

Référence du document	LETG_CM_MA_3-0
Version du document	3.0
Date de version	10/10/14
État du document	Fini
Rédacteurs	Maxime Collin
Relecteurs	Sarah Bourriaud

Historique des Modifications

Version	Commentaire	Date de modification	Auteur
1.0	Création du document	19/08/14	Maxime Collin
1.1	Rédaction parties I.1 à I.3	19/08/14	Maxime Collin
1.2	Rédaction partie I sauf I.12 et I.14	20/08/14	Maxime Collin
1.3	Fin rédaction partie I	27/08/14	Maxime Collin
2.0	Début rédaction partie II	27/08/14	Maxime Collin
2.1	Rédaction parties II.7 à II.10	29/08/14	Maxime Collin
2.2	Rédaction partie II.10 + petites corrections	01/09/14	Maxime Collin
2.3	Rédaction partie II.11	02/09/14	Maxime Collin
2.4	Suite rédaction partie II.11	15/09/14	Maxime Collin
2.5	Fin rédaction partie II.11	18/09/14	Maxime Collin
3.0	Finalisation document	10/10/14	Maxime Collin

Table des matières

I) Comprendre Symfony

Ce chapitre n'a pas pour vocation d'être un tutoriel sur Symfony, mais plus une synthèse pour fournir les connaissances strictement nécessaires propres à Symfony pour comprendre ManageChart. Pour des connaissances plus approfondies, il y a un excellent tutoriel qu'il faut adapter un peu puisqu'il a été écrit pour Symfony 2.1. Au moment où je rédige ce document nous en sommes à la version 2.5. De plus la documentation de Symfony est très bien faite et très complète, les différentes versions sont conservées et dans plusieurs langues. Par ailleurs cette synthèse est grandement inspirée de ce tutoriel et de la documentation officielle.

Symfony respecte le modèle MVC en utilisant un contrôleur frontal. Comme un schéma vaut mieux qu'un grand discours, un schéma issu du tutoriel expliquant le parcours d'une requête dans Symfony est joint en annexe.

Il existe deux environnements pour les projets Symfony : l'environnement de développement et l'environnement de production. En développement, le cache est vidé régulièrement, on a plus d'informations sur les erreurs provoquées et on a un outil supplémentaire qui apparaît, la Web Debug Toolbar qui donne beaucoup d'autres informations très utiles pour le développeur. En production, le cache n'est pratiquement jamais vidé, et on n'a aucune information sur les erreurs provoquées. Cela dit on améliore aussi grandement les performances.

Symfony a plusieurs outils puissants pour développer une application web et notamment la console ainsi que la Web Debug Toolbar.

1) La console

Symfony embarque une console permettant un grand nombre de choses, de la gestion du cache à la génération de code, en passant par la gestion de l'ORM. J'ai joint en annexe une liste de commandes utiles.

2) La Web Debug Toolbar

La Web Debug Toolbar de Symfony est une barre apparaissant en bas de chaque page web lorsqu'on est en environnement de développement. Elle fournit beaucoup d'informations sur le temps de chargement de la page, la mémoire utilisée, les requêtes exécutées, l'utilisateur courant, les événements soulevés, les formulaires et leurs données, le routing...

C'est un outil extrêmement puissant lorsqu'on développe.

3) L'arborescence de Symfony

Il y 4 répertoires importants dans Symfony :

- app/
- src/
- vendor/
- web/

Le répertoire app/ contient tous les fichiers concernant le site en dehors des fichiers sources, à quelques exceptions près. On y trouve notamment le cache, les logs, la configuration, la sécurité, les paramètres globaux...

src/ contient évidement tous les fichiers sources écrits nous-mêmes.

vendor/ contient toutes les librairies extérieures.

Et web/ est le dossier accessible depuis l'extérieur. Ne s'y trouve donc que les ressources publiques telles que les fichiers .js et .css, les images, les polices... On y trouve donc le sous-répertoire js/ qui contient tous les fichiers .js communs à tous les bundles, le sous-répertoire css/...

Le sous-répertoire bundles/ de web/ est intéressant : il contient toutes les ressources de chacun des bundles. Symfony pourra ensuite détecter de quelles ressources a besoin chacune des pages du site, et de générer les fichiers minifiés les regroupant. Cela permet d'envoyer au client uniquement les ressources nécessaires pour la page qu'il consulte. En réalité, ce sous-répertoire (bundles/) ne contient que des liens vers les fichiers ressources des bundles, qui sont eux stockés dans l'arborescence du bundle. Pour générer ces liens, il suffit d'exécuter la commande

```
$php app/console assets:install web/ --symlink
```

4) Cache et logs

Symfony optimise les temps de chargements en faisant de la mise en cache. Cela implique que toutes les modifications ne sont pas prises en compte à moins de vider le cache, notamment lorsqu'on modifie une ressource. La règle d'or lorsqu'on a une erreur est de toujours vider le cache puis de recommencer.

Il y a 2 solutions pour vider le cache :

- utiliser une commande Symfony

```
$php app/console cache:clear
```

- vider le cache « manuellement »

```
$sudo rm -rf app/cache/* app/logs/*
```

On peut remarquer que l'on vide également les logs, cela n'est pas obligatoire. Attention toutefois, j'ai remarqué qu'en local on peut perdre les droits sur ces dossiers, dans ce cas il faut faire

```
$sudo chmod -R 777 app/cache app/logs
```

Bien entendu, lorsqu'on crée des nouvelles ressources ou qu'on les modifie, il ne faut pas oublier de le dire à Symfony :

```
$php app/console assets:install web/ --symlink
```

En production cela donne :

```
$php app/console assetic:dump --env=prod

$php app/console cache:clear --env=prod
```

5) Les variables globales

Les variables globales sont définies dans le fichier parameters.yml. Pour celles communes a un bundle, le fichier est défini dans src/Mc/NomBundle/Ressources/config. Pour celles communes à l'ensemble de l'application, le fichier est défini dans app/config. Il faut ensuite les déclarer dans le fichier config.yml du même répertoire dans la section « parameters » ou dans le service que l'on désire.

6) Le bundle à la loupe

Une notion importante dans Symfony est le bundle. Un bundle est une partie du site qui fonctionne indépendamment des autres bundles. Bien sûr un bundle peut en utiliser un autre. Dans l'arborescence Symfony chaque bundle a sa propre arborescence, contenant tous les fichiers nécessaires à son fonctionnement spécifique. Je vais distinguer deux types de bundles :

- Les bundles écrits par nous-mêmes, stockés dans le répertoire src/
- et les bundles extérieurs, stockés dans le répertoire vendor/

Les premiers sont les bundles que nous n’avons pas réussi à trouver tout fait, contrairement aux seconds. Les bundles extérieurs sont en fait les bibliothèques de Symfony écrits par tout membre de la communauté ayant publié ses sources.

Du point de vue de leur arborescence, ils sont découpés ainsi :

- Controller/ : Contient les contrôleurs
- DependencyInjection/ : Contient des informations sur le bundle (chargement automatique de la configuration par exemple)
- Entity/ : Contient les modèles
- Form/ : Contient les éventuels formulaires
- Resources/ :
  - config/ : Contient les fichiers de configuration du bundle (nous placerons les routes ici, par exemple)
  - public/ : Contient les fichiers publics du bundle : fichiers .css et .js, images, etc.
  - views/ :Contient les vues du bundle, les templates Twig
- Tests/ : Contient les éventuels tests unitaires et fonctionnels.

7) Le routing

Le routing dans Symfony est traduit par des fichiers de mapping d'URL vers une action d'un contrôleur. Chaque bundle a son fichier de routing dans son répertoire ressources/config/. Il existe différents formats pour définir une route, donc notamment les formats .xml et .yml. Le format .xml est très verbeux, tandis que je trouve que le format .yml est simple et lisible. Dans le format .yml une route se décompose ainsi :

```
chart_show:

    pattern: /show/{id}

    defaults: { _controller: McChartBundle:Chart:show }

    requirements:

        id: \d+
```

En premier *chart_show* est le nom de la route. C'est grâce à son nom que l'on pourra la référencer dans le reste de l'application plutôt que d'utiliser l'URL en dur.

Ensuite vient l'URL elle-même, le *pattern*. Ici on a également un paramètre de route, *{id}*. Les paramètres sont placés entre accolades et séparés par des « / ».

Ensuite, on indique à quelle méthode de quel contrôleur cette route est associée. On précise donc le namespace du contrôleur (« McChartBundle »), puis le contrôleur (« Chart »), et enfin l'action (« show »). À savoir que Symfony utilise des conventions de nommage et l'action *show* devra être définie comme *showAction* dans le contrôleur. Cette méthode devra attendre comme paramètre *id*.

Vient après la section *requirements* qui est la liste des contraintes sur cette route. Ici on a une contrainte sur le paramètre *id* : il doit être un entier. Les expressions régulières sont autorisées ici. On peut également définir des valeurs par défaut aux paramètres comme ceci :

```
 defaults: { _controller: McChartBundle:Chart:show, id: 0 }
```

Cela dit il faut indiquer à Symfony d'importer ce fichier de route. Pour cela, on utilise un fichier de route central, placé donc dans le répertoire app/ et plus précisément dans le sous-répertoire config/. app/config/routing.yml contient donc l'import des fichiers de routing des bundles, plus la route d'index du site.

Voici comment on importe un fichier de route :

```
mc_chart:

    resource: "@McChartBundle/Resources/config/routing.yml"

    prefix:   /{_locale}/chart

    requirements:

        _locale: en|fr
```

On commence par indiquer le nom de l'import (*mc_chart*), puis l'emplacement de la ressource. On peut ensuite ajouter un préfixe à toutes les routes de la ressource, et on peut également définir des contraintes sur toutes les routes de la ressource. Ici, on a ajouté un paramètre système (*_locale*), ces paramètres systèmes effectuent en plus des actions supplémentaires automatiquement. Par exemple ici *_locale* définit la langue dans laquelle on désire obtenir les pages. Symfony chargera automatiquement les fichiers de langue correspondants.

On pourra ensuite générer des URLs dans le reste de l'application pour effectuer des redirections ou autres, que ce soit dans le contrôleur ou dans le fichier de template. On utilise dans les deux cas, le nom de la route pour que l'URL associée ne soit définie qu'en un seul endroit.

Dans le contrôleur on utilise la fonction

```
$url = $this->generateUrl('chart_show', array('id' => $id));
```

pour générer l'URL correspondante. On lui passe en argument le nom de la route et un tableau contenant les paramètres de la route. Si on ajoute un troisième paramètre définit à true, on génère une URL absolue. On peut ensuite faire une redirection avec la fonction

```
    return $this->redirect($url);
```

Dans le moteur de template on utilise la fonction

```
{{ path('chart_show', { 'id': chart.id }) }}
```

Ici *chart.id* est une variable Twig que l'on a passé depuis le contrôleur. Pour générer des URLs absolues on utilise de la même manière la fonction *url*.

8) Twig : le moteur de template

Symfony permet d'utiliser le moteur de template Twig qui définit sa propre syntaxe, se voulant plus simple et plus lisible que du PHP et spécialisé dans le rendu d'HTML. Cela dit, Twig est loin de ne se limiter qu'au HTML et permet ainsi d'écrire des e-mails, des flux RSS... Les fichiers Twig se placent dans le répertoire Ressources/views/ d'un bundle.

Du point de vue de la syntaxe, Twig définit 3 balises :

{# pour les commentaires #}

{{ pour l'affichage d'une variable }}

{% pour les actions %}

On appelle une vue depuis un contrôleur avec la méthode

```
return $this->render('McChartBundle:Chart:index.html.twig',

                    array('varTwig' => $varPhp));
```

Ici on appelle la vue index.html.twig qui est placée dans le répertoire src/Mc/Chartbundle/Ressources/views/Chart/. Il s'agit encore une fois d'une convention de nommage, pour que ce soit plus facile à lire. On peut remarquer que la vue a deux extensions : « .html.twig », c'est un fichier Twig qui contient du html. L'extension « .html » est juste une information pour les développeurs et n'est pas du tout une obligation, mais elle permet de mieux s'y retrouver. On a également passé une variable à la vue.

Exécuter une action

On exécute une action avec la syntaxe {% %}. Twig définit des tags que l'on peut exécuter comme les structures de contrôle ou la gestion de l'inclusion d'autres templates et même de l'héritage.

Afficher une variable

On affiche une variable simplement avec {{ varTwig }}. Bien sûr, *varTwig* peut être un objet ou un tableau, on accède alors à ses attributs ou index avec *varTwig.attr1* ou *varTwig['attr1']*. Ce qui se passe :

- Elle vérifie si *varTwig* est un tableau, et si *attr1* est un index valide. Si c'est le cas, elle affiche *varTwig['attr1']*.
- Sinon, et si *varTwig* est un objet
  - Elle vérifie si *attr1* est un attribut valide (public donc). Si c'est le cas, elle affiche *varTwig->attr1*.
  - Sinon, elle vérifie si *attr1()* est une méthode valide (publique donc). Si c'est le cas, elle affiche *varTwig->attr1()*.
  - Sinon, elle vérifie si *getAttr1()* est une méthode valide. Si c'est le cas, elle affiche *varTwig->getAttr1()*.
  - Sinon, elle vérifie si *isAttr1()* est une méthode valide. Si c'est le cas, elle affiche *varTwig->isAttr1()*.
- Sinon, elle n'affiche rien et retourne `null`.

On peut également appliquer des filtres et quelques fonctions sur la variable tels que s*triptags* qui suppriment toutes les balises XML, comme ceci :

```
{{ varTwig|striptags }}
```

Symfony enregistre aussi quelques variables globales dans Twig :

- {{ app.request }}
- {{ app.session }}
- {{ app.environment }}
- {{ app.debug }}
- {{ app.security }}
- {{ app.user }}

on peut également définir nos propres variables globales, voir #1.5.Variables globales

Le modèle « Triple-héritage »

Il existe une pratique permettant de bien organiser ses templates : le modèle « Triple-héritage ». Il s'agit de définir un héritage de templates sur 3 niveaux :

1. Le layout général de l'application. Il définit le header, le footer... Ce que l'on retrouve sur toutes les pages. Comme c'est un fichier commun, il est stocké dans app/ et plus précisément dans Ressources/views/.
2. Le layout du bundle. Il définit les parties communes de toutes les pages du bundle. Il est placé à la racine du répertoire Ressources/views/ du bundle. Il hérite du layout général.
3. Le template de page. Il contient le contenu central de la page. Il est placé dans un répertoire dans le répertoire Ressources/views/ du bundle et hérite du layout du bundle.

Pour hériter d'un template on utilise la fonction

```
{% extends 'lePere.html.twig' %}
```

On peut définir des blocs dans un template, qu'un template fils peut venir surcharger comme ceci :

lePere.html.twig :

```
{% block body %}

{% endblock %}
```

leFils.html.twig

```
{% extends 'lePere.html.twig' %}

{% block body %}

 toto

{% endblock %}
```

9) Traductions

Pour traduire l'application dans différentes langues, on utilise des fichiers catalogues regroupés dans les répertoires Ressources/translations/ de chaque bundle et également dans le répertoire app/ pour les traductions communes. On y trouve un fichier par langue. Leur fonctionnement est simple : ils associent une chaîne de caractères à un identifiant, la chaîne de caractères étant la traduction. Par la suite, on peut faire appel au service de traduction depuis un contrôleur ou une vue, en lui passant l'identifiant en paramètre. Le service renvoie alors la traduction associée en fonction de la locale de l'URL. On peut également transmettre des portions de chaînes à remplacer, comme un nom par exemple. Voici un exemple :

src/Mc/ChartBundle/Ressources/translations/messages.fr.yml :

```
msgconfirm: Voulez-vous vraiment supprimer %nameChart% ?
```

MaVue.html.twig

```
{{ 'msgconfirm'|trans({'%nameChart%': chart.nameChart}) }}
```

La chaîne *%nameChart %* de messages.fr.yml est alors remplacée par la valeur passée en argument de l'appel au service de traduction dans MaVue.html.twig (ici *chart.nameChart*).

Si il n'y a pas de locale définie dans l'URL, c'est alors la locale par défaut qui est utilisée. Elle est définie dans le fichier app/config/parameters.yml à fr.

Si il n'y a pas de traductions disponibles dans la langue demandée, il prend dans une autre langue. Si il n'y a pas de traductions dans aucune des langues, il renvoie l'identifiant de la chaîne.

Le format .yml permet d'organiser les fichiers messages.xx.yml en les factorisant : si dans l'identifiant de la chaîne à traduire on place des « . », on peut le factoriser dans messages.xx.yml. Exemple :

```
chart:

    table:

        titlecolumn:

            id: id

            name: Nom

            type: Type

            actions: Actions

            url: URL pour l'iframe
```

Ici on accède à la traduction de l'*id* avec la chaîne *chart.table.titlecolumn.id* et idem pour le nom, le type, les actions... Si l'on avait du mettre la chaîne complète pour chacun, on aurait perdu beaucoup de lisibilité :

```
chart.table.titlecolumn.id: id

chart.table.titlecolumn.name: Nom

chart.table.titlecolumn.type: Type

chart.table.titlecolumn.actions: Actions

chart.table.titlecolumn.url: URL pour l'iframe
```

Et cela n'est possible qu'avec le format .yml.

10) Gestion des ressources

Assectic est une bibliothèque présente par défaut sous Symfony qui permet de gérer les ressources. Assetic regroupe et minifie les ressources en production. On précise de quelles ressources a besoin chacune de nos pages, et Assetic se charge de créer un fichier .js et un fichier .css contenant uniquement ce dont la page a besoin. Bien sûr, si plusieurs pages ont exactement les mêmes besoins, ce sont les mêmes fichiers qui sont utilisés. Pour créer ces fichiers, il faut utiliser la commande

```
$php app/console assetic:dump –env=prod
```

Cela va créer autant de fichiers que nécessaire dans les répertoires web/js et web/css, qui seront ensuite disponibles pour les clients.

Pour préciser les ressources nécessaires d'une page, on utilise les balises *stylesheets* et *javascripts* dans Twig comme ceci :

```
{% stylesheets

 '@McChartBundle/Resources/public/css/main.css'

%}

       <link  rel="stylesheet"  href="{{  asset_url  }}"  type="text/css"
/>

{% endstylesheets %}


{% javascripts

 '@McChartBundle/Resources/public/js/main.js'

%}

       <script type="text/javascript" src="{{ asset_url }}"></script>

{% endjavascripts %}
```

On peut bien sûr mettre plusieurs ressources dans ces balises, en réalité tout l'intérêt est là.

11) Installation d'une bibliothèque

On peut les intégrer facilement à notre projet grâce au composer. Le composer est un service Symfony qui regroupe entre autres les dépendances du projet, tels que les bibliothèques. Il se présente sous la forme d'un fichier .json à la racine du site. Il a une section *require* qui contient la liste de toutes les dépendances du projet, telles que la version de PHP, celle de Symfony, les fichiers .css et .js extérieurs, et les bibliothèques. Pour ajouter une bibliothèque au projet, il suffit d'ajouter la ligne correspondante dans le fichier, et d'exécuter la commande

```
$php composer.phar update
```

à la racine du projet. Ensuite, si ce n'est pas déjà fait, il faut l'activer dans le fichier app/AppKernel.php en rajoutant la ligne

```
new Le\Nouveau\Bundle(),
```

Il existe des sites regroupant les bibliothèques existantes tels que http://knpbundles.com/ ou https://packagist.org/packages/symfony/.

Le répertoire vendor/ est donc réservé à ces bibliothèques et aucun des fichiers présents dans ce répertoire ne devrait être modifié puisque toute modification est susceptible d'être supprimée au prochain

```
$php composer.phar update
```

Pour modifier ces fichiers, il faut les surcharger.

12) Doctrine : l'ORM

L'ORM Doctrine est celle par défaut de Symfony. Elle possède ses propres commandes pour la gérer avec la console. On peut par exemple créer une entité en ligne de commande :

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

Il suffit alors de se laisser guider par la console. Je décris en annexe des valeurs à saisir. À noter que Doctrine se charge de rajouter un champ id automatiquement à toutes ses entités.

De même on peut mettre une entité à jour avec

```
$php app/console doctrine:generate:entities {{NameSpaceBundle}}:
{{Entity}}
```

Il faut ensuite faire le lien avec la base de données avec

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

(option --force pour exécuter concrètement les requêtes).

On peut voir que Doctrine a généré l'objet correspondant dans le répertoire src/Mc/NomBundle/entity/ avec tous ses attributs privés et les getter et setter. On remarque également la présence d'annotations sur les attributs telles que :

```
    /**
     * @var string
     *
     * @ORM\Column(name="nameChart", type="string", length=255)
     */
    private $nameChart;
```

L'annotation *@ORM\Column(...)*, indique à Doctrine que cet attribut correspond à la colonne *nameChart* dans la table correspondante de la base de données, que c'est un attribut de type *string* qui ne doit pas dépasser 255 caractères.

On peut voir aussi qu'un attribut *id* a été généré et qu'il prendra sa valeur de façon automatique lors de son premier enregistrement en base de données.

Enfin, il y a un fichier EntityRepository.php généré avec l'entité. Il s'agit d'une classe qui permettra d'exécuter des requêtes spécialisées pour récupérer nos objets. Je n'en ai jamais utilisé mais j'ai préféré les générer au cas où ils servent un jour.

Les relations

On peut bien entendu placer des relations entre nos objets. Pour Doctrine il y a trois types de relations :

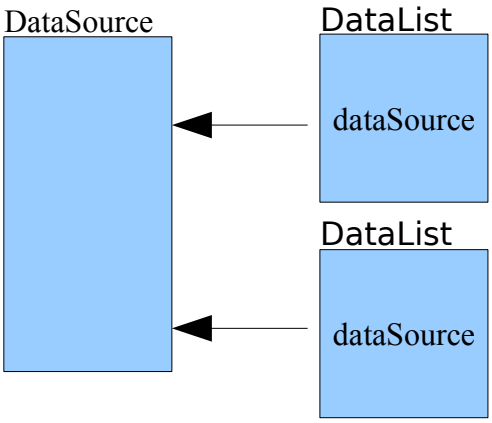
- OneToOne
- ManyToOne
- ManyToMany

Ces relations sont assez parlantes d'elles-mêmes. Pour en mettre une en place il faut commencer par définir le « propriétaire » de la relation, l'objet qui contiendra la référence de l'autre objet. Et ensuite si l'on veut que la relation soit unidirectionnelle ou bidirectionnelle. Dans une relation unidirectionnelle, seul le propriétaire de la relation peut accéder à l'objet visé, tandis que dans une relation bidirectionnelle l'objet visé peut également accéder au propriétaire. On parle d'« inverse » pour l'objet visé. Dans le cas d'une relation *ManyToOne,* c'est nécessairement du côté *Many* qu'est le propriétaire.

Pour établir une relation il faut ajouter à la main dans la classe de l'entité propriétaire (ici *DataList*) un attribut comme ceci :

```
  /**

                                                                                *
@ORM\ManyToOne(targetEntity="Mc\DataSourcesBundle\Entity\DataSource")

   * @ORM\JoinColumn(nullable=false)

   */

    private $dataSource;
```

Ici, l'attribut *$dataSource* vise l'entité *DataSource* en *ManyToOne*, donc schématiquement nous avons une relation comme ceci :



De plus l'annotation *@ORM\JoinColumn(nullable=false)* indique que cet attribut ne peut pas être null, autrement dit qu'il doit forcément pointer vers une *DataSource*.

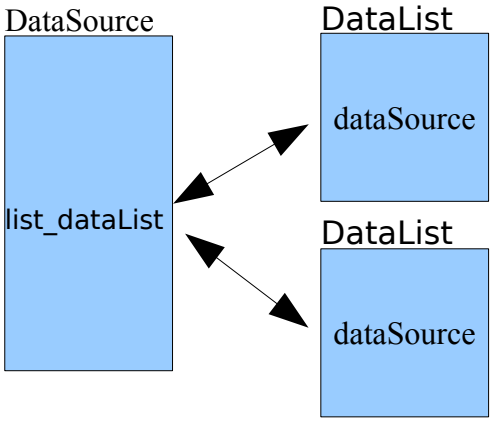
Maintenant si nous désirons avoir une relation bidirectionnelle, donc de telle sorte que l'objet *DataSource* puisse accéder à l'objet *DataList*, il faudrait modifier l'annotation dans *DataList* pour indiquer à Doctrine quel est l'attribut présent dans *DataSource* qui peut accéder à la *DataList* comme ceci :

```
  /**

                                                                                *
@ORM\ManyToOne(targetEntity="Mc\DataSourcesBundle\Entity\DataSource",
 inversedBy="list_dataList")

   * @ORM\JoinColumn(nullable=false)

   */

    private $dataSource;
```

Et placer un attribut dans *DataSource* comme ceci :

```
    /**

                                                                                *
@ORM\OneToMany(targetEntity="Mc\DataSourcesBundle\Entity\DataList",
       mappedBy="dataSource", cascade={"persist", "remove"})

     */

    private $list_dataList;
```

On aurait alors un schéma comme ceci :



Le *cascade={"persist", "remove"}*indique que lors de la modification de cet objet en base de données (que ce soit un enregistrement, une modification ou une suppression) les *DataLists* présentes dans *$list_dataList* doivent également être modifiées.

Après avoir modifié une entité, il faut toujours la mettre à jour pour être sûr que Doctrine prenne bien en compte les modifications. Si on a modifié ses attributs il faut également mettre à jour la base de données.

Après avoir fait la mise à jour on peut regarder les getter et setter des nouveaux attributs. Dans l'entité *DataList* on peut voir que *$dataSource* est un objet *DataSource* et non un identifiant, Doctrine gère tout seul la récupération de l'objet visé. Et dans l'entité *DataSource*, dans le cas d'une relation bidirectionnelle, *$list_dataList* est une Collection Doctrine d'objets *DataList*. Il manque l'initialisation de cette collection, qu'il faut rajouter dans le constructeur de *DataSource* :

```
public function __construct()

{

    $this->list_dataList =
    new \Doctrine\Common\Collections\ArrayCollection();

}
```

Il manque une dernière chose : modifier un des setters (soit celui de *DataList* soit celui de *DataSource*) pour bien mettre à jour les deux objets. Par exemple dans *DataList* :

```
                                    public          function
 setDataSource(\Mc\DataSourcesBundle\Entity\DataSource
             $dataSource)

    {

        $this->dataSource = $dataSource;

   $this->dataSource->addList_dataList($this);



        return $this;

    }
```

Ainsi lorsqu'on appellera *setDataSource(...)* de *DataList*, on ajoutera automatiquement la *DataList* dans la collection de *DataList* de *DataSource*.

Il faut bien entendu faire la même chose avec la suppression. Pour cela on peut utiliser un événement Doctrine. Doctrine définit un certain nombre d'événements, et l'on peut créer des méthodes dans nos entités qui écoutent ces événements. Ici c'est l'événement *PreRemove* que l'on va écouter. Pour cela il faut ajouter l'annotation *@ORM\HasLifecycleCallbacks* dans les annotations figurant au-dessus du nom de l'entité :

```
/**

 * DataList

 *

 * @ORM\Table()

                                                                        *
@ORM\Entity(repositoryClass="Mc\DataListBundle\Entity\DataListReposito
ry")

 * @ORM\HasLifecycleCallbacks

 */

class DataList

{
```

Ensuite on peut créer la méthode qui va mettre à jour l'objet *DataSource* :

```
/*
 * @ORM/PreRemove
 */
public function updateDataSource()
{
   $this->dataSource->removeList_dataList($this);
}
```

La manipulation des objets

Pour manipuler les objets Doctrine définit plusieurs fonctions :

- *persist($object)* : dit à Doctrine de gérer l'objet passé en paramètre. Doctrine l'enregistrera en même temps que toutes les autres modifications en base de données au prochain appel de flush()
- *clear()* : annule tous les *persist()* effectués depuis le dernier flush()
- *detach($object)* : annule tous les *persist* effectués sur l'objet passé en paramètre depuis le dernier *flush()*
- *flush()* : enregistre toutes les modifications en base de données
- *refresh($object)* : met à jour l'objet passé en paramètre dans l'état où il est en base de données. Écrase toutes les modifications depuis le dernier *flush()*.
- *remove($object)* : supprime l'objet passé en paramètre au prochain *flush()*.
- *contains($object)* : retourne true si l'objet passé en argument est géré par Doctrine.

13) Les formulaires

Doctrine permet la génération de formulaire pour ses entités grâce à la commande

```
$php app/console doctrine:generate:form {{NameSpaceBundle}}:{{Entity}}
```

Cette commande va générer la classe *EntityType* dans le répertoire Form/ du bundle. On y trouve la fonction *buildForm()*, qui ajoute les champs du formulaire à celui-ci. Doctrine a par défaut ajouté tout les champs correspondants aux attributs de l'entité, avec leur type. Bien que le formulaire est utilisable en l'état il est préférable de le regarder de plus près pour vérifier les types des champs. Il y a une liste officielle des types de champs disponibles. Attardons nous sur le type *entity*. Ce type permet de récupérer toutes les entrées de l'entité visée présentes en base de données, pour remplir une relation.

On peut également ajouter un *array* d'option comme troisième argument à la fonction *add()*. Les options dépendent du type de champ mais il en existe quelque unes universelles à tous les champs, telles que le *label* et *required*, qui définissent respectivement le label de ce champ tel qui sera affiché et si ce champ est obligatoire (true par défaut).

Il est possible de définir une valeur par défaut au champ. Pour cela il suffit de donner une valeur à son attribut dans le constructeur de l'entité.

Symfony ajoute de lui-même un dernier champ qui sera caché, le *token CSRF* permettant de gérer la sécurité. C'est parfois un champ oublié qui provoque une erreur s'il n'est pas présent.

Pour afficher un formulaire, Twig définit plusieurs fonctions de références dont la plus classique est *{{ form_widget(form) }}*. Cette fonction affiche tout simplement l'intégralité du formulaire. On peut également personnaliser un peu plus l'affichage du formulaire en utilisant des fonctions qui n'affichent qu'une partie de celui-ci telles que :

- *{{ form_label(form.attribut) }}*, qui n'affiche que le label de l'attribut en question.
- *{{ form_widget(form.attribut) }}*, qui n'affiche que la balise de l'attribut.
- *{{ form_errors(form.attribut) }}*, qui n'affiche que les erreurs de l'attribut lors de la soumission du formulaire.
- *{{form_row(form.attribut) }}*, qui affiche le label, la balise et les erreurs de l'attribut.

Lorsqu'on utilise ces fonctions il ne faut pas oublier d'appeler la fonction *{{ form_rest(form) }}* pour que le *token* ajouter automatiquement par Symfony soit présent. Cette fonction affiche tous les champs manquants.

On peut aller encore plus loin dans la personnalisation du formulaire grâce au Theming. Chaque partie de l'affichage d'un formulaire est appelé un fragment, ainsi *{{ form_label(form.attribut) }}*, et *{{form_row(form.attribut) }}* sont deux fragments différents. Pour personnaliser leur affichage, on peut les surcharger, puisque ce sont des simples blocs Twig.

Exemple de surcharge d'un bloc :

```
{% block form_row%}

 toto

      {{ form_label(form) }}

      {{ form_errors(form) }}

      {{ form_widget(form) }}

{% endblock %}
```

Le bloc *form_row* va maintenant afficher « toto » au début du champ sur le lequel il est appelé.

On peut surcharger un bloc en deux endroits différents :

- Dans le même fichier où est affiché le formulaire, le bloc n'est surchargé que pour ce formulaire. On appelle alors la fonction *{% form_theme form _self %}* au début du fichier pour lui dire de s'importer lui-même pour l'habillage du formulaire (attention : ceci ne fonctionne que si ce template en étend un autre ! Sinon il faut externaliser la surcharge du bloc).
- Dans un fichier externe, on peut alors :
  - Soit importer ce fichier dans un autre pour ne surcharger que le formulaire présent dans le 2e fichier. On appelle alors la fonction *{% form_theme form with 'McFormBundle:Form:fields.html.twig' %}* pour lui dire d'importer ce fichier pour l'habillage du formulaire.
  - Soit dire à Twig de surcharger tous ces formulaires avec celui-ci. Pour cela, il faut se rendre dans le fichier de configuration général (app/config/config.yml) et placé dans le champ twig->form->ressources, le nom du fichier qui surcharge les blocs (ici McFormBundle:Form:fields.html.twig).

Par la suite au niveau du comportement de Twig, il va d'abord chercher la surcharge du bloc, et dans le cas où il ne la trouve pas, il passe au thème global. Ce thème global est définit dans la librairie TwigBridge dans le fichier form_div_layout.html.twig.

On peut aller encore plus loin et ne surcharger qu'un champ individuel (propre à un attribut).

Les formulaires imbriqués

Lorsque l'on a un attribut de l'entité qui est une *Collection*, comme dans le cas d'une relation entre entités *ManyToOne* ou *ManyToMany*, on peut vouloir ajouter plusieurs objets visés par l'attribut dans le formulaire de l'entité.

Pour cela, il faut autoriser dans le formulaire l'ajout d'objet et potentiellement leur suppression, grâce aux options *allow_add* et *allow_delete*. Ensuite, tout ce passe en JavaScript. Il faut commencer par mettre un bouton d'ajout de l'objet dans le formulaire. Un clic sur ce bouton déclenchera l'appel d'une fonction qui insérera le formulaire de l'objet en question. Cette fonction peut également placer un bouton de suppression de l'objet, qui se contentera de supprimer le formulaire. Ensuite, les données sont automatiquement envoyées lors de la soumission, et les options *allow_add* et *allow_delete* permettent de rajouter des objets ou d'en supprimer dans la Collection.

Pour insérer le formulaire de l'objet, Symfony nous ajoute une balise * * avec un attribut *data-prototype* contenant le formulaire de l'objet. La fonction d'ajout va alors récupérer la valeur de cet attribut et l'ajouter à la balise * * de l'attribut.

On peut vouloir personnaliser l'affichage du sous-formulaire. Il pour cela il y a toutes les solutions présentées plus haut, ou une autre que j'ai choisi puisque j'avais besoin d'une gestion très fine du sous-formulaire. Attention : la solution que j'ai utilisée impose peu de ré-utilisabilité, ce que n'est pas dérangeant dans mon cas. Personnellement j'ai créé ma propre balise * * avec son attribut *data-prototype* qui est défini ainsi : *{% include 'McChartBundle:Form:formImbriqueYAxis.html.twig'%}*. Cela va placer tout ce qu'il y a dans le fichier formImbriqueYAxis.html.twig, dans l'attribut *data-prototype*. Et formImbriqueYAxis.html.twig contient toutes les balises HTML du sous-formulaire.

La validation

Lorsqu'on valide le formulaire, l'action du contrôleur appelé a à sa disposition la fonction *form->bind()* qui va remplir l'objet avec les données du formulaire. Ensuite, il peut appeler la fonction *form->isValid()*, pour valider les données de l'objet. Pour cela il faut ajouter un ensemble de règles de validation (ou contraintes) à l'objet. Il existe plusieurs solutions, j'ai choisi celle où ces contraintes sont définies dans l'objet lui-même, au niveau de ses attributs. Cela donne une meilleure vision des choses à mon sens.

On utilise donc encore une fois les annotations. Ici elles sont préfixées par *@Assert* et se placent juste au-dessus de l'attribut sur lequel elles s'appliquent. Voici la liste des annotations existantes.

14) ParamConverters

Les *ParamsConverters* vont agir juste avant la méthode du contrôleur. Ils permettent de convertir les paramètres de route dans le format souhaité, avant de les transmettre au contrôleur.

Un *ParamConverter* connu est *DoctrineParamConverter*. Doctrine est l'ORM utilisé dans ManageChart. Ce *ParamConverter* va être capable de récupérer une entité à partir de son id. Si on reprend la route c*hart_show*

```
chart_show:

    pattern: /show/{id}

    defaults: { _controller: McChartBundle:Chart:show }

    requirements:

        id: \d+
```

On comprend bien le but de cette route : afficher le graphique qui a l'identifiant *id*. Regardons maintenant l'entête de la méthode *showAction()* du contrôleur.

```
public function showAction(Chart $chart)
```

On voit que cette fonction attend un objet *Chart* et non un entier. C'est là que *DoctrineParamConverter* intervient : il écoute l'événement *kernel.controller* ; cet événement est propagé juste avant d'exécuter l'action du contrôleur. Le *ParamConverter* va alors voir que la méthode *showAction*() attend un objet *Chart*, qui figure dans sa liste d'entités. Or, de l'autre côté c'est un entier dont on dispose. Il va donc récupérer en base de données l'objet d'identifiant correspond et le transmettre au contrôleur. Tout cela est complètement transparent pour nous. Il existe bien sûr d'autres *ParamsConverters* et on a la possibilité de créer les nôtres.

15) La sécurité

La sécurité de l'application est définie dans le fichier app/config/security.yml. Elle se décompose en plusieurs sous-parties :

- L'encryptage des mots de passe des utilisateurs qui se fait avec la fonction de hachage SHA512.
- Le pare-feu.
- La gestion des rôles.

Provider

Parlons d'abord du provider. Il s'agit tout simplement du fournisseur d'utilisateur. Ici on utilise celui fournit par le *FOSUserBundle*, qui va récupérer nos utilisateurs enregistrés en Base de Données.

Pare-feu

On a ensuite le pare-feu, qui va s'assurer qu'on est connecté avant de nous laisser passer. On a en général deux pare-feu avec le *FOSUserBundle* :

- Le pare-feu de développement, qui permet de désactiver la sécurité sur certaines URLs à savoir celles qui ont cette expression régulière : _(profiler|wdt)|css|images|js ; il s'agit d'URLs utilisés en développement.
- Et le pare-feu principal pour le reste de l'application.

Dans un pare-feu, on trouve le pattern des routes catchées par ce pare-feu, le provider utilisé pour authentifier nos utilisateurs, le formulaire de connexion, la redirection en cas de déconnexion et l'option « remember me » ou non.

Il faut bien entendu autoriser les anonymes pour qu'on puisse s'authentifier.

Rôles

On utilise les rôles pour protéger nos ressources. Si un utilisateur n'a pas l'autorisation nécessaire pour accéder à une ressource, son accès lui est refusé.

Tout utilisateur enregistré et connecté a le rôle « IS_AUTHENTICATED_REMEMBERED ». On peut ensuite définir des rôles personnalisés simplement en les plaçant dans l'attribut *$roles* de l'utilisateur. Par convention les rôles personnalisés qu'on peut ajouter commencent tous par « ROLE_ ». il est également possible d'établir une hiérarchie dans la section *role_hierarchy.* Voici un exemple :

```
ROLE_ADMIN:       [ROLE_SCIENTIFIC]
```

Cette ligne signifie que quelqu'un qui a le rôle « ROLE_ADMIN » a également le rôle « ROLE_SCIENTIFIC ». On peut bien entendu définir plusieurs rôles hérités pour un même rôle.

On protège donc nos ressources en testant si l'utilisateur courant a le rôle nécessaire. On peut protéger nos ressources à plusieurs niveaux :

- Au niveau des routes, grâce à la section *access_control* du fichier security.yml.
- Au niveau des actions des contrôleurs, grâce à l'annotation *@Secure* et au service security.context.
- Au niveau de l'affichage des pages, avec la fonction Twig *is_granted()* qui utilise également le service security.context.

Les *access_control* catch une route et définissent le ou les rôles nécessaires pour accéder à cette route. On peut également n'autoriser que certaines adresses IP, ou certains canaux (comme HTTPS).

L'annotation *@Secure*, placée avant la déclaration d'une action du contrôleur définit le ou les rôles nécessaires pour accéder à cette action. Cette méthode offre une granularité plus fine que les *access_control*.

Enfin, on peut tester dans la vue si l'utilisateur à tel ou tel rôle, et faire un affichage différent en fonction.

FOSUserBundle

J'ai utilisé le *FOSUserBundle* pour gérer les utilisateurs de l'application. J'utilise donc ses routes pour la connexion et la déconnexion et son *provider*.

Il est conseillé de n'utiliser qu'un seul pare-feu avec ce bundle pour la compatibilité (le pare-feu dev ne comptant pas). C'est donc ce que j'ai fait.

Il faut le personnaliser un peu pour qu'il fonctionne. Et notamment créer un bundle UserBundle qui hérite de *FOSUserBundle* (méthode *getParent()* de McUserBundle.php), avec une entité qui hérite de *BaseUser*. Il ne reste plus qu'à le configurer : ajout de la section suivante dans le fichier app/config/config.yml

```
fos_user:

    db_driver:     orm

    firewall_name: main

    user_class:    Sdz\UserBundle\Entity\User
```

II) ManageChart

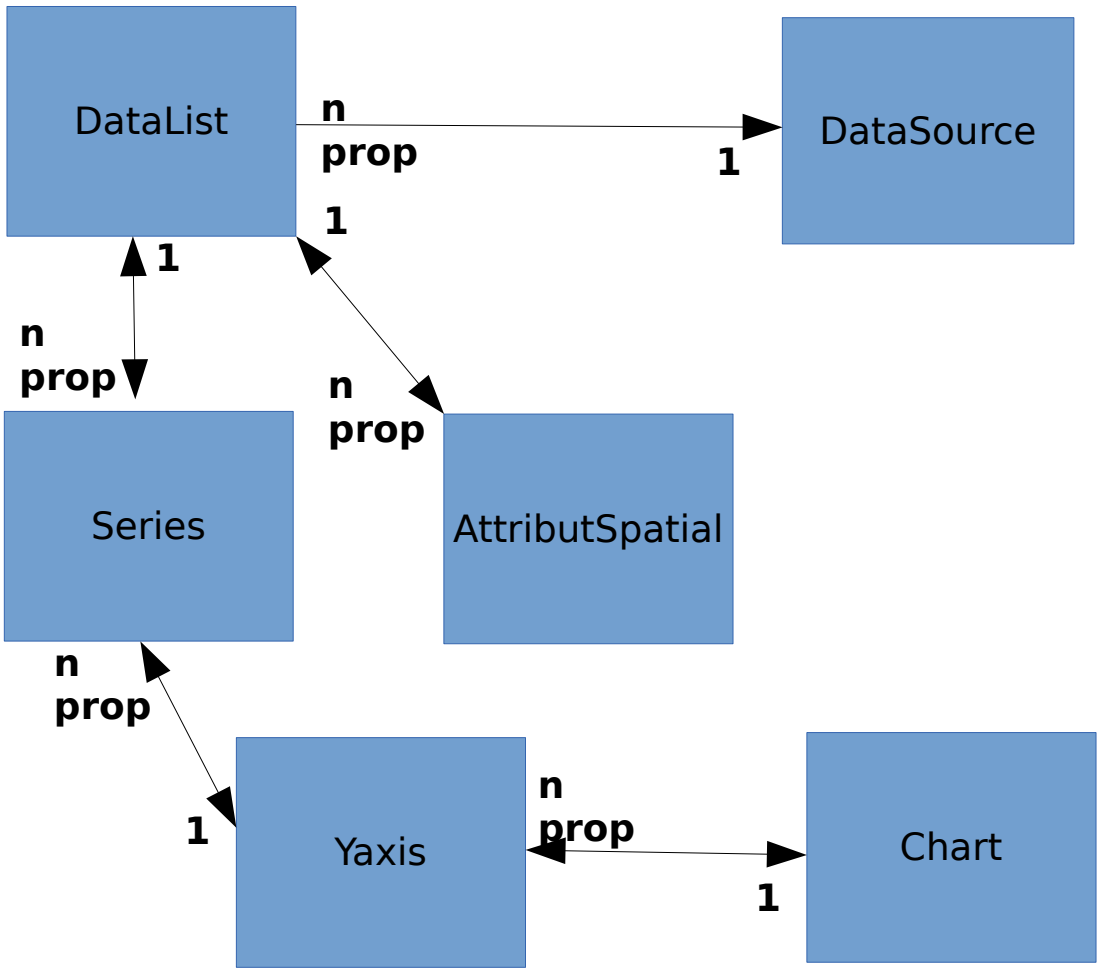
1) Base de données

Bien que nous ne sommes pas sensés toucher à la Base de Données mais utiliser l'ORM Doctrine, cela est parfois nécessaire. Attention toutefois, y toucher directement ne garantit en rien l'intégrité de la Base de Données.

Je ne présenterai que la façon dont elle est conçue vu par Doctrine.

On a une table *Account* qui contient toutes les informations propres aux comptes des utilisateurs, elle est gérée par *FOSUserBundle*. Vu la complexité de cette table je conseille d'utiliser plutôt les commandes Doctrine via la console pour la gérer. Pour rappel, elles sont fournies en annexe 2.

On entre ensuite dans le cœur de l'application. Voici un schéma simplifié des tables :



On peut noter la présence de flèches bidirectionnelles qui représentent des relations Doctrine bidirectionnelles, et le mot-clé « prop » qui représente quel objet est propriétaire de la relation (propre à Doctrine).

On peut voir ce schéma en 2 parties :

- Une pour la récupération des données.
- Et une autre la génération des graphiques.

On a 3 tables pour gérer la récupération des données :

- Une table *DataSources* qui contient les informations (hôte, port, nom de la base, login, password...) pour se connecter aux Bases de Données distantes (ex : marel) . Cette table est reliée à la table *DataList* par une relation un/plusieurs.
- La table *DataList* contient les requêtes SQL qui seront exécutées pour récupérer les données des Bases de Données distantes. Cette table est reliée à la table *AttributSpatial* et à la table *Series* par des relations un/plusieurs.
- Et *AttributSpatial,* la table qui contient les attributs spatiaux éventuels d'une requête. Ces « attributs spatiaux » permettent, lorsqu'on a une table avec plusieurs points géographiques, de ne sélectionner les données propres qu'à un seul ou à plusieurs de ces points. Cela dit ils permettent aussi de sélectionner une année, ou un autre type de donnée.

On a autant de tables pour gérer la génération des graphiques :

- la table *Series* qui fait le lien entre les données et les graphiques. Elle contient toutes les informations pour l'afficher sur le graphique et pour récupérer les données qui lui sont propres. Elle est liée à la table *YAxis* par une relation un/plusieurs.
- *YAxis* qui contient les informations qui sont propres à chaque axe des ordonnées avec la liste des séries à afficher sur cet axe. Elle est reliée à la table *Chart* par une relation un/plusieurs.
- Et la table *Chart*, qui contient toutes les informations restantes du graphique.

2) Configuration générale, paramètres globaux, import des routes et sécurité

Je vais détailler dans cette section les différents fichiers de configuration et notamment les lignes que j'ai modifiées.

app/config/config.yml

Le fichier de configuration générale de l'application.

Modification :

```
framework:

    translator:      { fallback: "%locale%" }
```

Je précise ici la locale par défaut si elle n'est pas précisée dans l'URL. Les paramètres entre %% sont définit dans le fichier parameters.yml

```
twig:

    form:

        resources:

            - 'McFormBundle:Form:fields.html.twig'
```

Ici je précise à Twig d'utiliser la vue fields.html.twig du bundle FormBundle pour l'habillage des formulaires. Dans cette vue je surcharge quelques fragments de formulaire.

```
    globals:

        timeout_redirection: %timeout_redirection%
```

Et ici je donne à Twig une variable globale *timeout_redirection*.

```
fos_user:
    db_driver:      orm
    firewall_name: main
    user_class:    Mc\UserBundle\Entity\Account
    from_email:
        address:        admin@indigeo.fr
        sender_name:    webmaster
        form:
            type: mc_user_registration
    profile:
        form:
            type: mc_user_profile
```

Là j'active *FOSUserBundle*. Je lui donne l'entité qui servira à enregistrer les utilisateurs (« `user_class:  Mc\UserBundle\Entity\Account` »), l'adresse e-mail à utiliser et le nom, ainsi que le nom du formulaire à utiliser pour l'enregistrement et celui pour l'édition du profil.

app/config/parameters.yml

Ici j'ai défini les paramètres de connexion à la base de données de ManageChart, avec la locale par défaut et un timeout servant pour la redirection après un enregistrement en Base de Données. Il y a également un paramètre nommé « secret » servant pour la sécurité.

app/config/routing.yml

Dans ce fichier, j'importe les routes des bundles mais je définit également la route servant pour l'index.

Je commence d'ailleurs par celles-ci :

```
index:
    path:       /
    defaults:   { _controller: McChartBundle:Chart:index }


index2:
    path:       /{_locale}
    defaults:   { _controller: McChartBundle:Chart:index }
    requirements:
        _locale: en|fr


index3:
    path:      /{_locale}/
    defaults:   { _controller: McChartBundle:Chart:index }
    requirements:
        _locale: en|fr
```

J'ai définit 3 routes pour reconnaître les URLs « managechart/ », « managechart/(fr|en) » et « managechart/(en|fr)/ ». Toutes renvoient vers la même action du contrôleur *Chart*. La route « index2 » me sert notamment pour conserver la locale de l'utilisateur pendant sa navigation sur ManageChart.

Ensuite je passe à l'import des routes des bundles. Je les préfixe toutes avec le nom de leur bundle pour assurer leur unicité. Pour beaucoup je les préfixe également avec « /admin » ou « /scientific ». J'ai définit des *access_control* sur les URL qui ont ces préfixes pour faire une gestion des rôles.

app/config/security.yml

C'est ici qu'est définie la sécurité de l'application. Il y a plusieurs sections comme le *provider*, la hiérarchie des rôles, le pare-feu, les *access_control*... La plupart de ces sections n'ont pas été modifiées, et sont identiques à la présentation que j'en ai faite dans la section 1.15.La sécurité.

Ce qu'on peut noter comme différence, c'est les routes utilisées pour la connexion et la déconnexion, qui sont celles de *FOSUserBundle*, et la redirection après ces actions. J'ai également définit un certain nombre d'*access_control*, pour permettre la protection par les rôles. Ainsi, les routes contenant « /admin » sont réservées aux administrateurs et les routes contenant « /scientific » sont réservées aux scientifiques. Tout le reste du site demande à être connecté à l'exception de quelques routes telles que le login, la visualisation de graphiques... qui ne demandent aucun droit.

J'ajouterai que toutes les actions des contrôleurs sont protégées à l'exception de quelque unes, ainsi que l'affichage de toutes les vues.

3) Layout général et menu

Je vais présenter ici les ressources Twig communes à toute l'application, et donc présentes dans le répertoire app/Resources/views.

app/Resources/views/layout.html.twig

Le layout général de l'application définit la structure du site (position du menu, footer...). Pour rappel, j'utilise le modèle « Triple-héritage » des vues : le layout général de l'application, le layout du bundle et la partie centrale de la page.

C'est dans cette vue que je définit les balises *<html>*, *<head>*, *<body>* et *<footer>*. Je définit également plusieurs blocs qui pourront ensuite être surchargés par les héritiers. Il y a 4 blocs dans cette vue :

- Le bloc *stylesheets*, qui définit les styles à importer.
- Le bloc *body*, qui définit le corps de la page
- le bloc *javascripts*, qui définit les scripts à importer.
- Et le bloc *document_ready*, qui définit le contenu de la fonction *$ (document).ready()*.

On peut noter que j'importe déjà des ressources dans les bloc *stylesheets* et *javascripts*, ce sont les ressources communes à l'ensemble de l'application.

Je définit également ici deux fonctions JavaScript dans la balise *<head>* puisque j'en ai besoin dans le reste de l'application, elles doivent être immédiatement disponibles. La première empêche la visualisation d'un graphique lorsqu'on clique sur le champ input contenant l'URL de l'Iframe associée. Et la deuxième empêche également la visualisation de l'élément du tableau lorsque l'on clique sur l'action supprimer, et retourne une popup de confirmation de suppression de l'élément.

Enfin on peut noter l'attribut *style* du * * qui contient tout le contenu de la page. Je n'ai pu externaliser ce style pour une raison qui m'échappe. J'ai constaté que l'utilisation de la police « SourceSansPro-Light » ne se faisait que lorsqu'elle était précisée dans l'attribut *style* de cette balise.

app/Resources/views/menu.html.twig

Il s'agit de la vue qui contient le menu de l'application. Elle est incluse dans le layout général. Chaque menu est une balise *<a>* qui redirige vers la bonne URL.

Si l'utilisateur courant n'est pas connecté, il ne voit que « ManageChart » et « Connexion ». « ManageChart » renvoie vers l'index, à savoir la liste des graphiques existants, et « Connexion » vers le formulaire de connexion à l'application.

Si l'utilisateur courant est un scientifique, « Connexion » est remplacé par son nom et une liste déroulante lui permet de voir son profil, de l'éditer et de se déconnecter.

Enfin si l'utilisateur courant est un administrateur il voit apparaître les volets « DataSources », « DataList », « Chart » et « Users », qui lui permettent d'accéder à ces différentes sections. À noter que « ManageChart » et « Chart » redirigent vers la même URL, « Chart » est juste là pour offrir plus de lisibilité à un administrateur.

app/Resources/views/addRegexJQuery.js.twig

Cette vue permet d'ajouter la prise en compte des expressions régulières dans JQuery. Je l'ai trouvée à l'adresse http://james.padolsey.com/javascript/regex-selector-for-jquery.

4) Css et Js généraux

Je décris ici les fichiers de styles et JavaScripts qui sont utilisés dans toutes l'application. Il sont bien entendu placés dans les répertoires web/css et web/js.

Css

J'utilise Bootstrap sur toute l'application. C'est un framework CSS très puissant. Il est basé sur un système de grille responsive de 12 colonnes. Je l'utilise avec l'exemple Boostrap starter template.

Il y a n'y a qu'un seul fichier de style général que j'ai écris : styleGlob.css. Ce fichier importe les deux seules polices utilisées dans toute l'application et définit les classes *loader* et *btn-delete*, qui sont utilisées pour afficher un gif de chargement et espacer les boutons de suppression des éléments de ceux d'édition dans les tableaux.

Js

J'utilise également JQuery sur toute l'application. On peut aussi noter que Bootstrap à un fichier JavaScript.

5) FormBundle

Ce bundle n'est là que pour surcharger les formulaires, afin :

- D'afficher une astérisque sur les champs obligatoires.
- Et d'utiliser Bootstrap dans les formulaires.

Pour cela, il a juste une vue fields.html.twig, qui surcharge les fragments _label et *_row* des formulaires (cf. 1.13.Les formulaires).

Ensuite, dans le fichier app/config/config.yml, j'ai précisé à Twig d'utiliser cette vue pour l'habillage des formulaires (cf. #2.2.1.app/config/config.yml).

6) UserBundle et surcharge FOSUserBundle

Ce bundle sert à utiliser *FOSUserBundle*. Pour cela il hérite de *FOSUserBundle* :

src/Mc/UserBundle/McUserBundle.php

```
class McUserBundle extends Bundle

{

 public function getParent()

 {

        return 'FOSUserBundle';

 }

}
```

Seul un administrateur peut enregistrer un nouvel utilisateur. Et seul l'utilisateur courant peut modifier son profil. Un utilisateur ne peut pas se supprimer lui-même.

Routes

On peut voir dans le répertoire src/Mc/UserBundle/Resources/config, deux fichiers de routes. Il y en a un réservé aux routes pour les administrateurs et toutes les routes de ce fichier sont préfixées par « /admin ». La seule route de ce bundle utilisable pour tout utilisateur connecté, est la visualisation du compte. Ensuite un administrateur a accès aux routes de suppression et de visualisation de tous les comptes.

Entités

Ce budnle définit une entité *Account* héritant de *BaseUser*, qui ne sert qu'à avoir une classe non abstraite.

Formulaires

Ce bundle a deux formulaires :

- Un pour l'enregistrement d'un nouvel utilisateur.
- Et un autre pour l'édition d'un utilisateur.

Ces formulaires ajoutent par rapport aux formulaires existants de *FOSUserBundle* la sélection du rôle de l'utilisateur. Pour les utiliser, je l'ai déclaré comme services dans le fichier config.yml du bundle.

Contrôleur

Le contrôleur de ce bundle ne définit que trois actions :

- *indexAction()*, qui est réservée au administrateur. Elle permet la visualisation de tous les comptes existants.
- *deleteAction()*, également réservée au administrateur. Qui permet de supprimer un compte existant.
- Et *showAction()*, qui permet à un utilisateur connecté de voir son compte et à un administrateur de voir n'importe quel compte.

Listener

Lors de l'enregistrement d'un nouvel utilisateur, *FOSUserBundle* propage l'événement *fos_user.registration.completed*, sur lequel il a un listener qui connecte automatiquement le nouvel utilisateur. Or cela a pour effet dans ManageChart de déconnecter l’administrateur qui vient de créer le compte de l'utilisateur au profit de celui-ci.

Pour empêcher cela, il y a plusieurs solutions. J'ai opté pour l'une d'entre elle, qui est certainement critiquable mais a le mérite de fonctionner. J'ai créé mon propre listener sur cet événement, avec une priorité plus forte (255), et je stoppe la propagation de cet événement dans mon listener. Je l'ai déclaré comme service dans le fichier de config du bundle.

Vues

Il n'y a que deux vues de définies dans ce bundle :

- index.html.twig qui affiche la liste des utilisateurs. Un message flash est affiché si l'utilisateur a essayé de se supprimer lui-même.
- show.html.twig qui affiche les détails d'un compte. Il affiche les boutons d'édition et de changement de mot de passe du compte uniquement si le compte visualisé est celui de l'utilisateur courant.

Elles héritent toutes les deux du layout du bundle qui ne fait rien de particulier mais qui est là pour respecter le modèle « Triple-héritage », et pour faciliter la modularité.

Surcharge de FOSUserBundle

Bien que ce ne soit pas lié à ce bundle, je place cette partie ici puisqu'elle traite de la gestion des utilisateurs. Les vues de *FOSUserBundle* sont volontairement simplistes pour être surchargées. C'est donc ce que j'ai fais en créant un répertoire FOSUserBundle/Resources/views dans le répertoire app/. On y voit un certain nombre de vues surchargées. La plupart ne présentent que peu d'intérêt, c'était simplement pour intégrer Bootstrap à ces formulaires. Il n'y en a que deux qui ont un fonctionnement légèrement différent de celui de *FOSUserBundle* :

- « edit_content.html.twig » qui active ou désactive la modifications du rôle de l'utilisateur en fonction de si l'utilisateur courant est un administrateur ou non.
- Et « email.txt.twig » qui réécrit l'URL pour qu'elle soit accessible depuis l'extérieur.

7) DataSourcesBundle

Ce bundle gère les informations de connexion aux Bases de Données distantes comme marel. Il est réservé aux administrateurs.

Routes

Comme ce bundle est réservé aux administrateurs, toutes ses routes sont importées avec le préfixe « /admin ». On a les routes classiques pour visualiser l'ensemble des *DataSources*, en supprimer, en éditer, en créer, et en visualiser que l'on retrouve dans les bundles *DataList* et *Chart*. Et celui-ci a une route qui lui est spécifique en plus : *data_sources_connection*; elle permet lors de la création ou de l'édition d'une *DataSource*, de tester si les informations saisies sont correctes en établissant une connexion avec la Base de Données distante.

Entités

Il n'y a qu'une seule entité pour ce bundle : *DataSource*. Elle a toutes les informations nécessaires pour établir une connexion avec la Base de Données distante, a savoir :

- le nom de Base de Données.
- Le serveur sur lequel elle est hébergée.
- Son port.
- Le login et le password pour s'y connecter.
- Et le type de base de données (MySQL, Postgres...).

Il y a quelques champs en plus comme une description. Il y a un champ qui peut faire un peu doublon avec un autre : le champ *typeBDD* avec t*ypeStrBDD*. Le premier est un *integer* utilisé plus loin et le deuxième permet simplement d'afficher le type de Base de Données.

Le login et password sont bien sûr encryptés lors de leur enregistrement en base de données grâce au bundle *EncryptBundle*. On les crypte pour pouvoir ensuite les décrypter lorsqu'on a besoin de se connecter.

La connexion elle-même est dépendante du type de Base de Données, pour cela c'est la responsabilité du bundle *BDDBundle*.

Il y a deux méthodes définies en plus dans cette entité, qui sont :

- *connect()* qui permet de se connecter à la Base de Données. Elle décrypte le login et le mot de passe, puis les ré-encrypte une fois la connexion faite.
- Et *testConnect()* fait la même chose que *connect()*, mais elle est appelée depuis le formulaire de création/édition et les login et mot de passe ne sont pas encore encryptés.

Ces deux méthodes sont susceptibles de soulever une erreur si la connexion a échoué.

Formulaires

Ce bundle n'a qu'un seul formulaire, « DataSourceType.php » dans son répertoire Form/. C'est un formulaire simple puisque tous ces champs sont des types primitifs (il y a une relation que lie son entité aux *DataList*, mais il n'en n'est pas propriétaire et elle est unidirectionnelle, donc l'entité *DataSource* n'a pas d'attributs sur cette relation).

On peut tout de même noter deux champs spéciaux :

- *typeBDD* qui est de type *choice*. cela se traduit par une balise *<select>*, dont les *<option>* valent le tableau passé en option à « choices ». Ici ce tableau est *AvailableType::$types*, c'est un tableau statique et publique du bundle *BddBundle*.
- Et *typeStrBDD* qui est caché. C'est juste la chaîne littérale pour afficher simplement le type de Base de Données. Elle est calculée en fonction de *typeBDD.*

Contrôleur

On y trouve les cinq actions classiques correspondantes aux routes, à savoir la visualisation de l'ensemble des *DataSources*, la suppression d'une, l'édition, l'enregistrement et la visualisation d'une. Elles sont toutes réservées aux utilisateurs ayant le rôle d'administrateur. Ce bundle a également une sixième action, *connectAction()*, correspondant à la route *data_sources_connection*. Cette action permet de récupérer les données du formulaire pour les tester en essayant de se connecter à la Base de Données visée par ces données. Elle renvoie un message d'erreur si la connexion à échoué, ou « Connexion réussie » dans le cas contraire.

On peut noter que les actions *newAction()* et *editAction()*, gèrent elles-mêmes la soumission du formulaire. Elles testent si la requête est de type *POST*, auquel cas on est en soumission du formulaire, et valident celui-ci. Pour le valider, elles commencent par créer un objet *DataSource* avec les données du formulaire, et valident l'objet en fonction des règles de contraintes définies dans celui-ci grâce à l'annotation *@Assert*. Si l'objet est valide, elles l'enregistrent en Base de Données, après avoir définit l'attribut *$typeStrBDD* à la valeur d'index *$typeBDD* du tableau *AvailableType::$types* du bundle *BddBundle*, et après avoir encrypté le login et le mot de passe. Elles redirigent ensuite vers la route de confirmation d'enregistrement.

Vues

Il n'y a que les vues classiques pour ce bundle :

- Le layout du bundle, qui ne définit rien de particulier.
- confirmed.html.twig, qui affiche juste un message de confirmation d'enregistrement d'une *DataSource*.
- form.html.twig, qui est le formulaire d'édition et de création d'une *DataSource*. Il ajoute simplement un bouton pour tester la connexion et une fonction JavaScript qui exécute un appel Ajax à la route *data_sources_connect* avec les données du formulaire en *POST*. La fonction affiche le retour de l'action associée à cette route, qui tente de se connecter à la Base de Données visée par les informations du formulaire.
- index.html.twig, qui affiche juste la liste des *DataSource*.
- Et show.html.twig, qui affiche les informations d'une *DataSource.*

8) EncryptBundle

Ce bundle n'a pour vocation que d'encrypter et de décrypter les chaînes de caractères qu'on lui donne. Pour se faire il n'a qu'une classe *Encrypt* dans son répertoire Controller/ et aucune route ne permet d'accéder directement aux actions de cette classe. En réalité ce n'est pas un contrôleur mais juste une classe PHP.

Encrypt crypte et décrypte les chaînes de caractères avec un OU-exclusif sur une chaîne définie en attribut privé. De plus il encode les chaînes de caractères cryptées en base64 (et bien sûr les décode avant de les décrypter).

9) BddBundle

Ce bundle permet de fournir des drivers adaptés à ManageChart pour abstraire le type de Base de Données sur lesquels on effectue nos requêtes SQL.

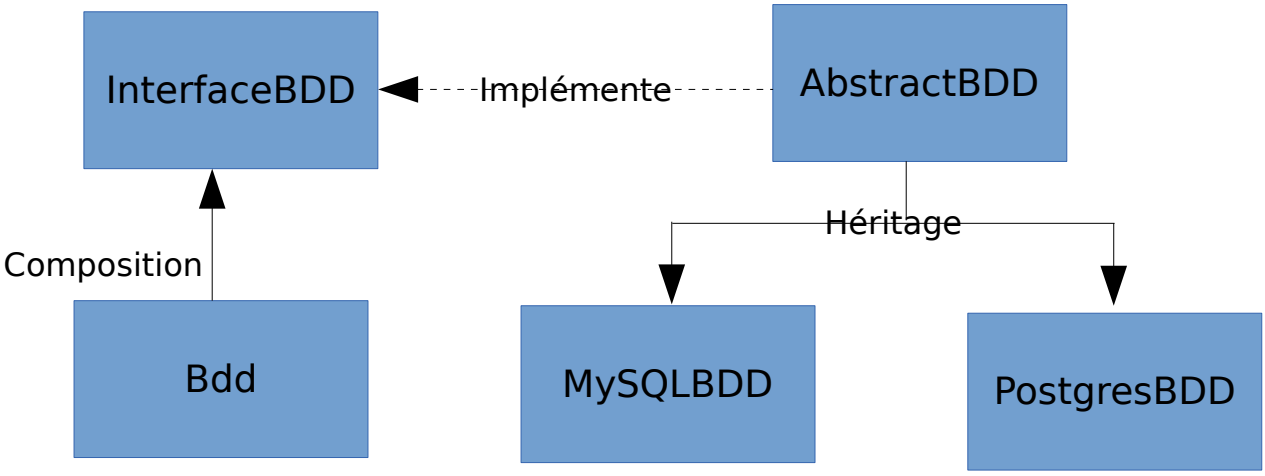
Il a pour vocation d'être utilisé uniquement par d'autres bundles et par conséquent n'a aucune route et aucun contrôleur.

J'ai définit une classe statique, *AvailableType*, qui déclare au moyen d'un tableau statique les types de Bases de Données gérées par ce bundle. Pour ajouter un nouveau type de Base de Données, il faut l'ajouter à la suite de ce tableau.

Toutes les classes de ce bundle sont définies dans le répertoire Controller.

Conception

C'est moi qui est conçu ce bundle. Je me suis appliqué à abstraire au maximum le type de base de données qui est interrogé derrière. Voici le schéma de conception :



InterfaceBDD

Cette classe est l'interface que doit respecter un driver. Elle permet d'abstraire l'implémentation.

AbstractBDD

Cette classe implémente l'interface du driver et est abstraite. Elle permet de centraliser la définition des attributs communs et leur getter et setter avec le constructeur. Elle n'implémente pas les méthodes déclarées dans *InterfaceBDD*, c'est le rôle de ses classes filles.

Comme attribut il y a toutes les informations pour se connecter à une Base de Données plus un attribut $*msgGetTimestamp*, qui est une chaîne de caractères décrivant la fonction SQL a utiliser pour récupérer un UNIX TIMESTAMP à partir d'un champ date. La valeur de cet attribut est dépendante du type de Base de Données et par conséquent est définie dans les classes filles. Elle est ensuite récupérée pour être affichée lors de la création et de l'édition d'une requête SQL.

PostgresBDD

Cette classe hérite de *AbstractBDD* et doit donc implémenter les méthodes déclarées dans *InterfaceBDD*. Elle définit également l'attribut statique $*msgGetTimestamp.*

On peut noter dans la méthode *connect()* la ligne

```
pg_set_client_encoding($this->connection, "UNICODE");
```

qui définit l'encodage des caractères récupérés en UTF-8 pour qu'ils s'affichent correctement dans ManageChart.

MysqlBDD

C'est une classe identique à *PostrgresBDD*. On peut noter que je l'ai implémentée en style procédural et non en orientée objet. C'est parce que le style orienté objet ne semble pas compatible avec la version PHP du serveur, en tout cas au moment où je l'ai développée. Par conséquent on trouve à chaque lecture du résultat un

```
mysqli_data_seek($result, 0);
```

pour revenir au début du résultat pour les lectures suivantes. On trouve également l'encodage en UTF-8 dans la fonction *connect()*.

Bdd

Cette classe ajoute une abstraction sur le type de base de données. Elle a un attribut *bdd*, qui est « de type » *InterfaceBDD* pour résumer. En réalité, cet attribut est instancié à partir des classes filles de *AbstractBDD* en fonction du type de base de données. Par la suite, cette classe renvoie son attribut et l'on peut appeler les méthodes de l'interface sans savoir si l'on a une base de données Postrgres ou MySQL.

Le constructeur de cette classe se crée à partir d'une *DataSource*. Il récupère son type et instancie une des classes filles de *AbstractBDD* en fonction de ce type avec tous les paramètres de la *DataSource*.

Il permet aussi de récupérer les fonctions SQL pour récupérer un UNIX TIMESTAMP de chaque type de base de données implémentés dans le driver, afin de les afficher dans l'interface de construction de requêtes SQL.

Enfin il affiche un message d'erreur si le type qu'on lui donne ne correspond à aucun type de base de données figurant dans *AvailableType*.

10) DataListBundle

Ce bundle gère les requêtes SQL effectuées sur les *DataSources*. Il est réservé aux administrateurs.

!! ATTENTION !!

J'ai noté une étrangeté sur le comportement de Doctrine entre le mode développement et le mode production :

- Lorsque Doctrine récupère les entités en base de données en mode production, tout se passe bien. Il renvoie un tableau avec toutes les entités voulues.
- Mais lorsqu'il le fait en mode développement, pour une raison inconnue, il inverse l'ordre des entités.

Cela à de l'importance dans ce bundle, on doit donc rajouter la fonction *array_reverse()* lorsqu'on est en développement et la supprimer en production. Il faut ajouter/supprimer cette fonction dans les fichiers :

- src/Mc/DatalistBundle/Controller/DataListController.php, ligne 177 :

```
$attributsSpatiaux = array_reverse($dataList->getAttributsSpatiaux()->toArray());
```

- src/Mc/DataListBundle/Entity/DataList.php, ligne 200 :

```
foreach (array_reverse($this->attributsSpatiaux->toArray()) as $attributSpatial)
```

Bien entendu ce comportement se répète partout, mais il n'a vraiment de l'importance qu'ici.

Routes

C'est un bundle entièrement réservé aux administrateurs donc toutes ses routes sont protégées avec le préfixe « /admin ». On y trouve les routes classiques pour gérer les *DataLists* (visualisation de l'ensemble, suppression, édition, création, visualisation d'une seule), avec en plus les deux routes :

- *data_list_query_traitment* qui permet de tester en création/édition d'une requête la DataList. On applique le traitement sur les données récupérées, tel qu'il sera fait par le serveur avant de les donner aux graphiques.
- Et *data_list_query_brut* qui permet de tester de la même manière une requête mais sans traiter les données.

Entités

On trouve deux entités dans ce bundle qui viennent compléter la partie récupération des données. En premier lieu on a les *DataList*, qui sont les requêtes elles-mêmes. Elles sont liées aux *DataSources* par une relation un/plusieurs dont elles-sont propriétaires. Et on a les *AttributSpatial*, qui sont la sélection d'un jeu de données associé à un point dans la table de la base de données visée par la requête. Ce point est classiquement géographique mais peut être temporel ou d'un autre type. Par exemple dans la base Obsera, on veut pouvoir sélectionner les données correspondantes à « Capesterre ». c'est donc un *AttributSpatial*. D'un point de vue implémentation, les attributs spatiaux viennent s'insérer dans la requête au niveau de la clause WHERE. Pour reprendre l'exemple de Obsera, on aurait :

SELECT [...] WHERE loc.name='Capesterre' [...];

L'entité *AttributSpatial* est liée aux *DataList* par une relation un/plusieurs bidirectionnelle dont elle est propriétaire.

Rentrons maintenant dans le détail de ces entités. Je vais commencer par *AttributSpatial* puisque elle est utilisé dans *DataList*.

AttributSpatial à plusieurs champs dont :

- Un champ *nameAttribut* qui est l'identifiant de la colonne de la base de données sur laquelle est vient s'appliquer.
- Un champ *valueAttribut*, qui permet de donner une valeur à l'attribut pour tester la requête et visualiser les séries des graphiques associés. Cette valeur est ensuite modifiée lorsqu'on appelle un graphique depuis une application extérieure pour le visualiser dans une Iframe.
- Un champ *typeAttribut* pour différencier si l'attribut est de type littéral ou numérique, puisqu'il faut le placer entre ' s'il est de type littéral.
- Et un champ *keywordAttribut*, qui est le mot clé SQL utilisé dans la requête pour insérer l'attribut spatial. Il peut être « AND », « OR » ou vide. Il vient se placer après l'attribut dans la requête de cette manière :

  WHERE {{attributSpatial}} {{keyword}}

Cette entité est donc liée au *DataList* par une relation un/plusieurs dont elle est la propriétaire. Elle a donc un attribut *dataList* qui fait le lien. La relation est bidirectionnelle, cela est précisé par l'option *inversedBy="attributsSpatiaux",* « attributsSpatiaux » étant un attribut de *DataList*. Pour mettre à jour l'entité *DataList* j'ai défini dans le setter *setDataList()* l'appel au setter *addAttributsSpatiaux()* de *DataList*, et une méthode qui écoute l'événement *@PreRemove*, pour supprimer l'attribut spatial de la collection de sa *DataList*.

DataList elle à un champ nom et un champ requête. Elle a deux autres attribut qui sont *dataSource* pour sa relation avec l'entité *DataSource*, et *attributsSpatiaux* pour sa relation bidirectionnelle avec *AttributSpatial*. Enfin elle a quatre autres champs qui ne sont pas propagés en base de données. Il s'agit d'attributs servant à stocker le résultat de sa requête, ils ne sont donc utilisés que de façon temporaire lorsqu'on exécute la requête. On trouve :

- *data*, pour le résultat brut de la requête.
- *dataColumn*, pour le résultat brut de la requête rangé par colonne.
- *fields*, pour le nom des colonnes de la requête.
- *dataByParameter*, le résultat traité de la requête rangé par paramètres.
- Et *parameter*, la liste des paramètres de la requête.

Cette entité à tous les getters et setters correspondants, plus quatre méthodes servant à exécuter la requête et à en récupérer les données. La première, *executeQuery()*, va servir de chef d'orchestre. On peut noter quelle a deux paramètres facultatifs qui vont lui permettre d'insérer correctement les attributs spatiaux dans la requête. Voici le déroulement de cette méthode :

1. Elle commence par se connecter à la base de données distantes.
2. Puis elle vérifie si elle attend des attributs spatiaux, auquel cas elle les insère dans la requête avant d'exécuter celle-ci. Lorsqu'elle insère les attributs spatiaux dans la requête, elle le fait juste après la clause WHERE. Si elle ne l'a trouve pas, elle rajoute la clause WHERE à la fin de la requête sans tenir compte d'éventuel clause ORDER BY ou LIMIT BY. C'est donc à l'administrateur qui écrit la requête que revient la responsabilité de placer la clause WHERE (même vide) dans la requête si nécessaire.
3. Pour finir elle récupère les données, et les traites pour les rendre affichable dans un graphique avant de libérer le jeu de résultat.
4. Enfin cette méthode retourne la chaîne de caractères de la requête exécutée avec l'insertion des attributs spatiaux. Cela permet de mieux comprendre d'éventuelles erreurs.

On a deux méthodes pour insérer les attributs spatiaux dans la requête, *insertAttributsInQuery()* et *insertAttributInQuery()*. Ce qui les différencie c'est le pluriel : la première va insérer tous les attributs spatiaux dans la requête à l'aide de la deuxième qui va elle renvoyer la chaîne à insérer pour un attribut spatial.

La méthode « singulière » renvoie le nom de l'attribut suivi de la valeur qui lui est passé en paramètre placé entre ' ou non en fonction de type de l'attribut, et avec le mot clé à la fin.

La première elle, a trois cas à distinguer en fonction de la provenance des attributs spatiaux :

- Celui où ils sont enregistrés en base de données.
- Celui où ils proviennent du formulaire de création/édition lorsqu'on teste la requête.
- Et celui où ils sont dans l'URL.

Dans le premier cas, on insère tous les attributs spatiaux enregistrés en base de données avec leur valeur de test. *AttributsSpatiauxURL* vaut alors « default » et *editCreate* « false ».

Dans le deuxième cas on en en édition/creation, *editCreate* vaut alors « true ». Il n'y a aucun attribut spatial enregistré en base de données. Ils proviennent de la requête et sont dans le paramètre *attributsSpatiauxURL*. On trouve tous les champs d'un attribut spatial séparé par des « @ », les attributs spatiaux étant eux-même séparés par des virgules. On créé donc un tableau à partir de *attributsSpatiauxURL* en utilisant la virgule comme séparateur. On se retrouve alors avec un tableau dont chaque case contient un attribut spatial avec tous ses champs. On itère sur ce tableau, et on redécoupe chaque case en tableau en utilisant cette fois l'arobase comme séparateur. Ce qui donne un tableau avec chacun des champs d'un attribut spatial. On peut donc l'insérer dans la requête.

Et dans le dernier cas, on trouve uniquement les valeurs des attributs spatiaux dans l'URL séparées par des virgules. Elles sont stockées dans le paramètre *attributsSpatiauxURL* et *editCreate* vaut « false ». De la même façon on sépare ces valeurs dans un tableau. Pour chaque attribut spatial enregistré en base de données, on l'insère dans la requête avec la valeur qui provient de l'URL. Si on a pas suffisamment de valeurs dans la requête par rapport au nombre d'attributs spatiaux enregistrés en base de données, on soulève une erreur de code 24. Et si une valeur est nulle on soulève une erreur de code 42.

Enfin la quatrième méthode de *DataList* est *decoupeData()* qui va traiter les données pour les rendre affichables dans un graphique. Elle créé un tableau pour les résultats traités en fonction des valeurs des colonnes de la requête. Ce tableau aura cette forme :

```
[
        'col3@col4' => [
                            [col1, col2], [col1, col2], [col1, col2] ...
                        ],
        'col3@col4' => [
                            [col1, col2], [col1, col2], [col1, col2] ...
                        ],
        'col3@col4' => [
                            [col1, col2], [col1, col2], [col1, col2] ...
                        ]
]
```

Où col1 correspond à la valeur en x, col2 à la valeur en y, col3 au nom du paramètre et col4 à son unité. À noter que col4 peut ne pas être définit.

Voici un exemple avec de vraies valeurs :

```
[
        'Température@C°' =>   [
                                    [14586058, 16], [14587058, 18]
                              ],
        'Salinité@PSU'    =>  [
                                    [14586058, 35], [14587058, 36]
                              ]
]
```

Formulaires

Ce bundle a un formulaire pour chacune de ses entités. Ainsi on a *AttributSpatialType* et *DataListType*. Le premier est simple puisqu'il n'a que des types primitifs. Le deuxième lui à un champ *entity* pour sélectionner la *DataSource* sur laquelle va s'exécuter la requête, et un champ *collection* pour ajouter ou supprimer des attributs spatiaux. Le champ *entity* affiche le nom des *DataSources* pour les sélectionner, et le champ *collection* autorise l'ajout et la suppression de *AttributSpatialType*.

Contrôleur

Le contrôleur de ce bundle a les actions classiques de gestion d'une *DataList* à savoir la visualisation de l'ensemble des *DataList* enregistrées en base de données et la suppression/visualisation/création/édition d'une. Pour ce qui est de la création/édition une petite particularité vient s'ajouter dû à la relation bidirectionnelle avec les *AttributsSpatiaux*. En effet les *AttributsSpatiaux* de la *DataList* attendent pour être enregistrés l'identifiant de la *DataList*. Or celle-ci n'a pas d'identifiant tant qu'elle n'est pas elle-même enregistrée, puisque son identifiant est en « auto-incrément ». Doctrine gère cela de façon transparente sur les relation unidirectionnelle mais pas sur les relations bidirectionnelle. On doit donc enregistrer la *DataList* seule, puis ses *attributsSpatiaux* après leur avoir réaffecté la *DataList* avec son id.

J'utilise également dans ces méthodes un tableau de variables pour la traduction de certaines parties du formulaires, qui sont ajoutées dynamiquement en JavaScript. Et en édition je transmet la liste des attributs spatiaux initiaux, pour les afficher puisqu'ils ont été supprimés de la collection de la *DataList*.

Ils restent deux actions qui sont *queryBrutAction()* et *queryTraitmentAction().* Elles permettent de tester la requête lorsqu'on est en création/édition de celle-ci, et d'afficher ses résultats bruts ou traités. Elles font appel à une troisième méthode interne au contrôleur, *getDataList()*. Cette méthode renvoie un objet *DataList* à partir des données du formulaire après avoir exécuté et affiché la requête avec les attributs spatiaux si il y en a. Dans le cas de *queryTraitmentAction()* il y a un traitement pour afficher le nombre de valeur en Y nulles.

Vues

En plus des vues classiques (on peut noter le double tableau dans show.html.twig pour afficher la *DataList* et ses attributs spatiaux), on peut voir la vue formImbriqueAttributSpatial.html.twig. Cette vue va permettre d'ajouter dynamiquement des formulaires personnalisés d'attributs spatiaux.

form.html.twig est le formulaire utilisé pour la création et l'édition d'une *DataList*. Elle a une balise * * ayant *data-prototype* comme attribut qui a pour valeur formImbrique.html.twig. C'est cet attribut qui sera utilisé pour ajouter des formulaires d'attributs spatiaux dynamiquement.

On trouve une fonction JavaScript (*query()*) dans cette vue permettant de tester la requête. Elle prend en paramètre la route à appeler et fait une requête Ajax sur cette route avec la *dataSource*, la requête et les *attributsSpatiaux* du formulaire en paramètre. Cette fonction va afficher un message d'erreur si un des champs du formulaire n'est pas renseigné. S'il n'y a pas eu de message d'erreur, elle affiche le gif de chargement et lance la requête Ajax. Elle arrête le gif lorsqu'elle reçoit une réponse du serveur.

Lorsque la page est chargée, on transmet quelques variables dont celles de traduction à JavaScript. Si l'on est en édition on créé un formulaire pour chaque attributs spatiaux de la *DataList* avec les valeurs pré-remplies. Pour cela il faut ajouter un bouton d'ajout de ces attributs spatiaux puisque l'insertion de leur formulaire se fait en fonction de ce bouton.

CSS

J'ai défini quelques styles pour le formulaire de ce bundle dans le fichier form.css du répertoire Resources/public/css. On trouve un agrandissement des champ, l'utilisation de la police italique pour le nombre de valeurs nulles de chaque paramètre lors de l'appel à *queryTraitementAction()* et un espacement de bouton.

JS

Il n'y a qu'un seul fichier JavaScript, formAttributSpatial.js dans ce bundle. Il est en charge des attributs spatiaux du formulaire. Pour cela il définit plusieurs fonction :

- *ajouterAttributSpatial()*, qui ajoute un formulaire d'attribut spatial au formulaire. On récupère le contenu de l'attribut *data-prototype* de la balise * * et on l'ajoute juste avant le lien d'ajout. On ajoute également un lien de suppression de ce formulaire.
- *ajouterLienSuppression()*, créé le lien de suppression et l'ajoute au formulaire de l'attribut spatial. Au clic sur ce lien, on supprime le formulaire.
- *insertAttributInQuery()*, insère les attributs spatiaux présent dans le formulaire dans le champ requête, tel qu'ils le seront par la suite par le serveur. À noter que cette fonction les insère en commentaire ce qui ne changera en rien leur insertion par le serveur. Si un champ d'un attribut spatial n'est pas renseigné, un message d'erreur est affiché en-dessous de cet attribut. Les autres attributs sont insérés.
- Et *addControlNewDeleteAttributSpatial()*, qui ajoute un lien d'insertion de formulaire d'attribut spatial si on est pas en édition (déjà fait dans le cas contraire). Cette fonction définie également un compteur d'attributs spatiaux à partir du nombre d'attributs spatiaux déjà présent (cas en édition). Ce compteur est incrémenté lors du clic sur le lien d'ajout d'attribut spatial.

11) ChartBundle

Voici le cœur de l'application. C'est le bundle qui gère les graphiques de leur création à leur visualisation dans une application externe comme Indigeo.

AvailableChoice

Ce répertoire regroupe les possibilités sur les types d'axes, de séries, leurs couleurs et leurs styles. Ce sont les possibilités qui viennent de la librairie Highchart, à l'exception des couleurs. Elles ont donc lieu d'être peu modifiées, et uniquement par un administrateur. J'ai préféré les placer en dur dans le code, plutôt que de toutes les enregistrer en base de données et faire des requêtes systématiquement. Je les ais regroupé dans ce répertoire, en créant une classe pour chacun des types de possibilités et en les plaçant dans un tableau statique et publique. Pour les modifier il faut rentrer dans le code et modifier ces fichiers, mais cela n'impactera en rien le reste de l'application. On peut noter que j'ai utilisé la même technique pour les types de bases de données gérées par le *BddBundle.*

Routes

On trouve trois fichiers de routing dans de bundle. C'est parce qu'on a besoin d'une gestion des permissions fines sur ce bundle. En effet tout le monde doit pouvoir consulter des graphiques, mais seul des scientifiques et des administrateurs peuvent en créer et en modifier, et seul les administrateurs peuvent en supprimer.

Pour cela on a donc trois fichiers de routing qui sont importés avec des préfixes différents :

- routing.yml, qui est le fichier de routes ouvertes à tous. Il est importé avec le préfixe « /chart ». On y trouve les routes pour la visualisation d'un graphique dans ManageChart ou dans une Iframe et pour récupérer les données du graphique. Les routes prennent des paramètres facultatifs qui sont les attributs spatiaux par défaut à 0, et si on est en édition/création par défaut à false. C'est pour le comportement de l'insertion des attributsSpatiaux dans la requête.
- routing_scientific.yml, qui est le fichier de routes réservées aux utilisateurs ayant le rôle ROLE_SCIENTIFIC, c'est-à-dire les scientifiques et les administrateurs. Il est importé avec le préfixe « /scientific/chart ». On y trouve les routes de création et d'édition. On a deux formulaires différents (un pour Highchart et un Highstock) donc ces routes sont doublées.
- Et routing_admin.yml qui est le fichier de route réservées aux administrateurs. Il est importé avec le préfixe « /admin/chart ». On y trouve la route de suppression d'un graphique.

Pour rappel la route de visualisation de l'ensemble des graphiques est l'index de ManageChart et est définie dans le fichier de routing général app/config/routing.yml. Elle est ouverte à tous.

Entités

On trouve trois entités dans ce bundle : *Chart*, *YAxis* et *Series*. Elles gèrent entièrement la partie graphique. Pour reprendre le schéma des relations entre les entités en 2.1.Base de données, on a

- *Chart* qui est est liée aux *YAxis* par une relation un/plusieurs. En effet un graphique peut avoir plusieurs axes des ordonnées. Cette relation est bidirectionnelle et c'est *YAxis* qui en est propriétaire.
- *YAxis* qui est liée aux *Series* par une relation un/plusieurs également, puisqu'on peut avoir plusieurs séries sur un même axe des ordonnées. Elle est bidirectionnelle et c'est *Series* qui en est propriétaire.
- Et *Series* qui est liée au *DataList* pour récupérer les données. La relation est aussi d'ordre un/plusieurs puisqu'une *DataList* peut avoir plusieurs *Series*. Elle est également bidirectionnelle et c'est *Series* qui en est propriétaire.

Un *Chart* à :

- Un nom pour l'identifier dans ManageChart. C'est un champ obligatoire.
- Un titre et un sous-tire à afficher sur le graphique. Ce sont des champs non-obligatoires.
- Un champ crédits obligatoire.
- Une légende non-obligatoire. Elle est affiché par défaut.
- Un type qui est rempli par l'application avec les valeurs « Highchart » ou « Highstock ».
- Un titre et une unité pour l'axe des abscisses non-obligatoires.
- Un type pour l'axe des abscisses obligatoire.
- Un gapSize obligatoire et supérieur ou égal à 0. Ce champ sert pour les graphiques temporels ayant des données régulières. Il définit un nombre d'unité de temps (calculé par Highchart comme étant le minimum entre deux données) au-delà du quel si il n'a pas de mesures, il affiche un trou dans le graphique. C'est utile pour les données manquantes. Il a pour valeur par défaut 0.
- Et deux champs pour l'export du graphique dans un format d'impression et l'export des données bruts en CSV, non-obligatoires.

Un *YAxis* à :

- Un titre non-obligatoire.
- Et un type obligatoire ayant pour valeur par défaut « linear ».

Il s'ajoute également automatiquement dans la collection d'*YAxis* de *Chart* avec son setter *setChart()*. Et ce supprime de celle-ci sur l'événement *@PreRemove*.

Une *Series* à :

- Un titre obligatoire.
- Une unité non-obligatoire.
- Un type, une couleur et un style (ligne pleine, pointillés...) obligatoires.
- Des marqueurs sur chacun des points non-obligatoire.
- Et un paramètre qui est le numéro du paramètre de sa *DataList*. Pour rappel une requête peut avoir plusieurs paramètres comme la température et la salinité.

Elle s'ajoute automatiquement dans les collections de *Series* d'*YAxis* et de *DataList* avec les setters correspondants. Et se supprime de ces collections sur l'événement *@PreRemove*.

Formulaires

On trouve bien évidemment un formulaire pour chacune de ces entités. Ainsi *ChartType* définit tout les champs qui lui sont nécessaires, mais ils peuvent être différents en fonction de si l'on a un graphique Highchart ou un graphique Highstock. Les différences sont les suivantes :

- Un graphique Highchart peut définir le type d'axe des abscisses et ajouter ou supprimer des axes des ordonnées.
- Un graphique Highstock peut définir le *gapSize*.

YAxisType est très simple, il a juste un champ *titre* et un champ *type* et peut ajouter ou supprimer des *Series*.

SeriesType est plus complexe. Il a les champs simples comme son titre, son unité, sa couleur... mais il a deux champs plus complexes : *dataList* et *parameterDataList*. *parameterDataList* est dépendant de *dataList*, il faut donc ajouter dans le formulaire cette dépendance pour le valider. Le champ *dataList* est de type *entity* et affiche le nom des *DataList* dans la balise *<select>*. Le champ *parameterDataList* est de type *choice* et affiche les paramètres de la *DataList* sélectionnée.

On définit donc deux listener sur les événements PRE_SET_DATA et POST_SUBMIT, afin de réaliser la dépendance à l'initialisation du formulaire et à sa soumission. Le but de ces listener est d'ajouter le champ *parameterDataList* au formulaire avec les paramètres de la *DataList* sélectionnée au moment de leur événement. On on a donc trois cas :

- en création sur l'événement PRE_SET_DATA, il n'y a pas de *DataList* sélectionnée. Il faut donc ne pas mettre de valeur dans le champ *parameterDataList*.
- En édition sur l'événement PRE_SET_DATA, il y a une *DataList* sélectionnée. Il faut donc placer ses paramètres dans le champ *parameterDataList*.
- Et en soumission sur l'événement POST_SUBMIT, il y a également une *DataList* sélectionnée. Il faut aussi placer ses paramètres dans le champ *parameterDataList*.

Pour évider la redondance de code, j'ai défini une fonction qui ajoute ce champ avec les bonnes valeurs en fonction de la *DataList* qui lui est passée en paramètre. J'ai placé la définition de cette fonction dans une variable pour y faire appel dans les listener. Ceux-ci doivent exécuter la requête de la *DataList* pour récupérer ses paramètres. Il est à noter que c'est de là que vient le temps de soumission long du graphique : puisqu'on ré-exécute toutes les *DataList* sélectionnée autant de fois qu'il a de séries (y compris si c'est la même *DataList*), le temps de soumission est nécessairement beaucoup plus long.

Contrôleur

Ce contrôleur à un bon nombre de méthodes. Il y a les actions classiques de gestion de graphiques (visualisation, suppression, édition, création), qui sont doublées dans le cas de l'édition et de la création en raison des deux types de graphiques (Highchart/Highstock). On trouve également une action pour renvoyer les données d'une *DataList* en JSON, et une autre pour afficher le graphique dans une Iframe. Enfin il y a plusieurs méthodes internes au contrôleur évitant la redondance de code. Prenons toutes les méthodes dans l'ordre :

- *initialiseVarTwig()* est une méthode interne. Les vues de ce contrôleur notamment celles du formulaire d'édition/création ont beaucoup de variables. Cette méthode permet de centraliser leur initialisation. Il y a une liste de traduction pour les formulaires dynamiques, des listes de choix traduites pour les balises <*select*>, la liste des *DataList* de la base de données et les premières valeurs de ses listes.
- *getListDataList()* est également une méthode interne, elle est utilisée pour afficher la liste des *DataList* sur le formulaire.
- *getDataParameterDataListAction()* est une action du contrôleur, donc atteignable avec une route. En l'occurrence il s'agit de la route *get_data_parameterdatalist*, qui est ouverte à tous. En effet cette action permet de renvoyer les données d'une *DataList* au format JSON pour ensuite les afficher dans un graphique. L'action et la route sont mal nommées : elles devraient plutôt s'appeler *get_data_datalist* ; c'est parce que cette action à changée et ne renvoyait avant que les données d'un seul paramètre de la DataList. Cette méthode va commencer par exécuter la requête, avec des attributs spatiaux dans l'URL si on est en visualisation, sinon avec ceux de la base de données si on est dans le formulaire d'édition/création. On récupère ensuite les données de la *DataList* et on les encode en JSON. Il faut distinguer les cas où la valeur de X peut être de type *string* ou *number* et idem pour la valeur de Y. De plus la valeur en Y peut valoir *null*. Dans le cas d'une valeur de type *string* il faut l'encoder en JSON pour échapper les caractères spéciaux. Enfin on envoi l'ensemble des données en JSON. Il est possible d'appeler cette méthode toute seule pour déboguer un graphique. On appelle alors l'URL *www-iuem.univ-brest.fr/wapps/managechart/{_locale}/chart/get-data-parameterdatalist/{idDataList}/0/true* et on voit la réponse JSON.
- *indexAction()* renvoi simplement la page pour afficher la liste des graphiques. La variable *$tabInitial* en commentaire n'a strictement rien à faire là et peut être supprimée. Je l'avais créée pour aider quelqu'un sur un forum et j'ai oublié de la supprimer.
- *deleteAction()* supprime un graphique. Elle est réservée aux administrateurs.

- *register()* est une méthode interne permettant d'enregistrer un graphique. Elle est utilisée lors de l'édition et de la création. L'enregistrement des diverses entités liées à un graphique est complexe et nécessite d'être externalisé. Pour rappel un graphique à un ou plusieurs axes des ordonnées qui ont chacun une ou plusieurs séries. Lors de leur enregistrement chacune de ces entités à besoin de la référence (id) de leur entité en relation (*YAxis* pour *Series* et *Chart* pour *YAxis*). Or cette référence ne leur sera donnée qu'une fois qu'elles seront enregistrée puisqu'elles sont en auto-incrément. Toutes ces relations sont bidirectionnelles et Doctrine ne gère pas très bien ce type de relation. C'est donc à notre charge d'enregistrer les entités dans le bon ordre et de leur passer les références. On commence donc par vider la collection d'axes Y du graphique pour l'enregistrer (en prenant soin de les conserver dans une variable). Puis pour chaque axe, on fait la même chose avec leurs séries, on leur affecte le graphique et on les enregistre. À noter que le setter *setChart()* appelle le setter *addYAxis()* de l'entité *Chart*. Et on recommence avec les séries.
- *edit()* est également une méthode interne permettant d'éditer un graphique et de valider sa soumission. Il faut supprimer les axes des ordonnées et leur séries en base de données avant de les enregistrer sous peine de conflit. À noter que la méthode *remove()* n'est effective qu'au prochain *flush(),* effectué donc par *register()* si le graphique soumis est correct. Dans le cas d'un graphique Highstock, il ne peut avoir qu'un seul axe, la possibilité d'en ajouter et d'en supprimer dans le formulaire a donc été supprimée. Mais il faut lui créer un axe par défaut. *edit()* retourne un graphique si elle a validé le formulaire soumis ou les variables du formulaire dans le cas de la création de celui-ci.
- Vient ensuite les deux actions *editHighchartAction()* et *editHighstockAction()* qui font appel à *edit()* et retourne un formulaire Highchart ou Highstock ou la confirmation de l'enregistrement du graphique.
- *create()* est une méthode interne similaire à *edit()*. Contrairement à celle-ci elle n'a pas à supprimer la liste des axes et des séries en base de données. En revanche elle affecte le type d'un graphique qui ne peut changer.
- On a ensuite les actions *newHighchartAction()* et *newHighstockAction()* qui font globalement la même chose que *editHighchartAction()* et *editHighstockAction()*.
- On trouve également les actions *confirmedAction()* pour confirmer l'enregistrement d'un graphique et *showAction()* pour en visualiser un dans ManageChart.

– Enfin on a *showIframeAction()* pour visualiser un graphique dans une Iframe avec des attributs spatiaux dans l'URL si besoin. Le paramètre *test* sert à définir si on utilise les attributs spatiaux présent dans l'URL ou ceux présent en base de données.

Vues

On peut voir beaucoup de dossier dans ce répertoire. Je vais les détailler avec leurs fichiers un à un.

– Chart/ est le répertoire pour les vues classiques de ce bundle autre que le formulaire, à savoir l'index pour visualiser l'ensemble des graphiques, et leur visualisation dans ManageChart ou dans une Iframe.

– Form/ regroupe les vues propres à la visualisation du formulaire d'édition/création.

– Highchart/ et Highstock/ regroupent les vues new et edit adaptées en fonction du type de graphiques.

– Et JS/ regroupe des vues ne contenant que du code JavaScript. Si j'utilise Twig pour ce code JavaScript c'est soit parce que j'utilise des fonctions Twig dans le JavaScript, soit pour intégrer tout le code dans la fonction *document_ready()* de JQuery.

Chart/

On trouve dans ce dossier les vues index.html.twig, show.html.twig et showIframe.html.twig.

index.html.twig

Affiche la liste des graphiques dans un tableau. Elle ajoute un champ input contenant l'URL de ce graphique pour l'afficher dans une Iframe. Ce champ est en lecture seule et un traitement à été effectué au préalable sur l'URL pour quelle soit visualisable depuis l'extérieur de l'iuem :

```
{% set urlIframeExt = urlIframe|replace({'wapps.univ-brest.fr' : 'www-iuem.univ-brest.fr'}) %}
```

show.html.twig

Affiche un graphique dans ManageChart. Elle fait appel à l'action *showIframeAction()* pour afficher le graphique dans une Iframe dans ManageChart. Cela permet de conserver la création du graphique sur *showIframeAction()*. Les seules choses qu'ajoute show.html.twig c'est l'intégration dans ManageChart et l'appel aux attributs spatiaux présent en base de données. En effet lorsqu'on visualise les graphiques dans ManageChart c'est pour visualiser son rendu, non analyser les résultat. C'est donc la valeur des tests des attributs spatiaux des *DataList* associées à ce graphique qui sont utilisées.

showIframe.html.twig

Est la seule vue de ManageChart n'utilisant pas le modèle triple-héritage. En effet cette vue a pour but d'afficher un graphique dans une Iframe, le menu de ManageChart n'a donc rien à faire ici. Cette vue définie ses propres balises <*html*> <*header*> et <*body*>. De plus contrairement à ManageChart elle n'utilise pas le style d'exemple *starter-template* de Bootstrap.

Elle ajoute un bouton pour télécharger les données bruts du graphiques en CSV si l'option à été définie à *true* dans le formulaire, sans passer par le serveur de Highchart. À noter que ce bouton n'est actuellement pas compatible avec Internet Explorer.

On a ensuite une balise <*div id=«container»*> dans lequel sera inséré le graphique. On importe les fichiers JavaScripts nécessaires à la création du graphique en fonction de si c'est un graphique Highchart ou Highstock.

Vient après la création du graphique. Je défini les options du graphique dans une variable puis je crée le graphique à partir de ces options. Cela permet d'ajouter les données du graphique plus tard. Avec cette méthode il est nécessaire de définir le *renderTo* de *chart* à l'id de la balise <*div*> qui va contenir le graphique (ici « container »).

Une particularité de Doctrine est lorsqu'on enregistre une valeur booléenne en base de données à partir d'une balise <*checkbox*> d'un formulaire, c'est la valeur *null* qui est enregistrée si cette case n'est pas cochée. Il est donc nécessaire de tester la valeur en base de données pour réellement placer une valeur booléenne.
Je ne vais détailler les options de Highchart, leur API en ligne est bien mieux documentée que ce que je pourrais faire.

À fin de showIframe.html.twig j'ajoute les données au graphique. Pour cela je commence par définir deux variables globales :

- *dataGlob* qui va contenir toutes les données du graphique.
- Et *fileAttente* qui va permettre d'afficher toutes les séries liées à une même *DataList* lorsque les données de celle-ci ont été récupérées.

J'appelle ensuite la fonction *setSeries()* définie dans new-edit-showIframe_js.js.twig sur toutes les séries. Cette fonction va permettre d'ajouter les données aux séries et de mettre à jour le bouton d'export des données CSV que j'ai créé.

Form/

Ce répertoire ne contient que les balises HTML des formulaires d'édition/création d'un graphique.

formChart.html.twig

Est la vue globale du formulaire d'édition/création. On y trouve le graphique et les champs globaux au formulaire comme son nom ou son titre. Je personnalise beaucoup l'affichage des champs sur la grille Bootstrap pour faire un formulaire ergonomique. Une autre particularité de cette vue sont les deux balises * * à la fin avec l'attribut *data-prototype* définit comme étant le contenu d'un autre fichier. Il s'agit des vues qui seront utilisées pour les formulaires imbriqués des axes des ordonnées et des séries.

formImbriqueYAxis.html.twig

Permet d'inclure le formulaire d'axes des ordonnées. Un axe des ordonnées n'a que trois champs : un titre, un type et ses séries.

formImbriqueSeries.html.twig

Comme formImbriqueYAxis.html.twig, cette vue permet d'inclure des formulaires de séries. Il a beaucoup de balises *<select>* dont les choix possibles ont été passés en variables à Twig depuis le contrôleur.

Highchart/

Ce répertoire contient les vues de création et d'édition d'un graphique Highchart. Elles font appels aux vues du répertoire Form/.

new.html.twig

Je commence par redéfinir le bloc *form_row* pour l'adapter à ce formulaire, puis j'inclue le formulaire formChart.html.twig.
On passe ensuite au JavaScript qui va gérer cette page. On a tout d'abord l'inclusion des divers scripts qui vont être utilisés puis la fonction document*_ready()* de JQuery. Là on commence par préciser qu'on a pas de données à charger, contrairement en édition où il faut charger les données des *DataList*. Cela est utilisé par les fonctions qui chargent les données pour optimiser un peu le chargement de la page. On prépare le graphique avec le code JavaScript se trouvant dans JS/createChart.js.twig, on s'occupe d'ajouter le type de l'axe des abscisses (possible uniquement en Highchart) puis on crée le graphique. Highchart crée automatiquement un premier axe des ordonnées. Cela ne nous convient pas pour la suite de la gestion des axes du graphique. On supprime donc cette axe et on ajoute un listener sur le changement de type d'axes des abscisses (possible uniquement en Highchart). Enfin on permet l'ajout et la suppression de formulaire d'axes des ordonnées et on inclut le code de JS/new-edit_document_ready.js.twig. On peut préciser que les trois fonctions utilisées dans cette vue (*getTypeAxis()*, *updateTypeAxis()* et *addControlNewDeleteYAxis()*) sont définies dans public/js/formYAxis.js.twig.

edit.html.twig

Cette vue ressemble à new.html.twig mais rajoute d'autres choses dues à l'édition. Par exemple :

- il faut redéfinir le bloc *mc_chartbundle_chart_list_yaxis_widget* à une simple balise * * avec un id qu'on pourra récupérer par la suite. Cela permet d'éviter que Symfony ajoute les formulaires d'axes des ordonnées tout seul. Je prend la main sur cela en raison de la complexité derrière et de l'ergonomie.
- On retrouve ensuite le même code que new.html.twig avec en plus l'ajout des formulaires d'axes des ordonnées et leur contenu. Il faut commencer par ajouter un lien d'ajout d'axe des ordonnées car c'est le point de repère utilisé pour intégrer les formulaires imbriqués d'axes.
- Ensuite pour chaque axe on ajoute son formulaire avec la fonction *addYAxis()* de public/js/formYAxis.js et sa balise * * pour ajouter des séries. Cette partie est plus complexe car on peut ajouter une seule série ou toutes celles d'une *DataList*.
- On peut passer à l'ajout de chacune des séries de cet axe. On utilise donc la fonction *addSeries()* de public/js/formSeries.js et on ajoute le contenu des champs. Il ne faut pas oublier de prévenir les listeners qui écoute ces champs du changement de leur valeur. Et on passe en mode synchrone pour les requêtes Ajax pour éviter de dupliquer les requêtes de récupération de données.
- Une fois qu'on a ajouté toutes les séries on peut ajouter le contrôle de l'ajout et de la suppression de celles-ci avec la fonction correspondante dans public/js/formSeries.js. On peut également définir le contenu des champs de l'axe.
- Une fois l'ajout des axes et de leurs séries terminé on peut ajouter le contrôle de l'ajout et de la suppression des axes, et inclure le code JavaScript contenu dans JS/new-edit_document_ready.js.twig.

Highstock/

Les vues de ce répertoire sont très similaires à celles du répertoire Highchart/ mais s'adaptent aux graphiques Highstock.

new.html.twig

Par rapport à la vue new.html.twig du répertoire Highchart/ on trouve une modification du style du formulaire puisqu'il y a déjà un formulaire d'axe présent. Ensuite contrairement à cette vue on ne supprime pas l'axe des ordonnées existant puisqu'on ne peux pas en ajouter ou en supprimer en Highstock. En revanche on doit supprimer les deux premières séries créées automatiquement par Highstock (la deuxième sert à l'affichage dans la barre de navigation sous le graphique).

edit.html.twig

On retrouve nettement la vue edit.html.twig du répertoire Highchart/, mais avec la différence qu'on ne peut ajouter ou supprimer d'axes des ordonnées. On a aussi les modifications de style du formulaire qu'on trouve dans la vue new.html.twig du même répertoire.

JS/

On ne trouve dans ce répertoire que du code JavaScript qui est inclut dans les vues new, edit et showIframe soit dans la balise <*script*> soit dans la fonction *document_ready()*.

createChart.js.twig

On trouve toutes les options du graphique pour le créer mais également des variables globales. Commençons par les variables globales :

- *dataGlob* est un tableau qui va contenir toutes les données des *DataList* indexé par *DataList*.
- On a ensuite les variables Twig que l'on partage avec JavaScript pour pouvoir les utiliser dans les scripts telles que les traductions ou la liste des *DataList*.
- On a ensuite deux tableaux très importants : *arrayCorresIndex?????Chart*. Ces deux tableaux font la correspondance entre les index des axes des ordonnées et des séries du formulaire et ceux du graphique. En effet on peut ajouter et supprimer des axes et des séries dans tous les sens du côté du formulaire mais il ne faut pas perdre la référence côté graphique ! Le comportement de Highchart est tel que lorsqu'il a 3 séries et qu'on supprime la première, il décrémente l'index des deux autres. Pour cette raison j'ai créé ces deux tableaux associatifs, qui ont pour index le numéro de la série ou de l'axe dans le formulaire et pour valeur celui du graphique. Une valeur négative signifie qu'il n'y a pas de correspondance. Pour la correspondance entre les séries j'ai dû résoudre un petit problème : la série 1 de l'axe 1 à le même numéro que la série 1 de l'axe 2 ! J'ai donc défini mes index de cette manière : *indexYAxis* * *nbSerieMaxParAxe* + *indexSerie*. Ainsi ma série 1 de l'axe 1 à pour index '000' et ma série 1 de l'axe 2 '100', si l'on définit *nbSerieMaxParAxe* à 100. J'ai entouré de ' mes index car ils sont associatifs.

- Vient ensuite la récupération des valeurs des champs du formulaire et l'inclusion du thème définit avec Laurence David.
- Enfin on a les options du graphique. On peut noter l'option *turboTreshold* définie à 1000. il s'agit de sa valeur par défaut. Cette option sert à accélérer le chargement des données lorsqu'elles sont très nombreuses (ici supérieure à 1000). J'avais essayé de l'utiliser avec une valeur à 1 mais cela provoquait des bugs. Je l'ai tout de même laissé pour pouvoir la changer plus facilement.

highchart_theme.js.twig

On définit ici les éléments de thème généraux à tous les graphiques. On définit donc la liste des couleurs possibles comme celles-ci utilisées pour indigeo. Le séparateur de millier devient l'espace et non la virgule (convention USA), et le séparateur de décimale devient la virgule et non le point (convention USA). Je défini également les polices de caractères et les tailles de polices. Enfin j'autorise la sélection d'un camembert sur un graphique circulaire.

new-edit_document_ready.js.twig

Ce code JavaScript concerne les vues new et edit des répertoires Highchart/ et Highstock/, et vient se placer dans la fonction *document_ready()* de JQuery. On y trouve surtout des listeners pour le formulaire. Au niveau des champs globaux (autres que axes et séries). Il y a :

- Le listener pour modifier la propriété CSS du div du graphique de « fixed » à « absolute » et inversement.
- Celui pour afficher ou cacher le graphique.
- La fonction qui va remplir les options du *<select>* du type d'axe des abscisses, à partir de celles transmises depuis le contrôleur pour qu'elle soient traduites.
- La suppression d'un classe pour rendre l'affichage plus propre.
- Et les listeners du formulaire au niveau global (titre et sous-titre du graphique ainsi que titre et unité de l'axe des abscisses, et submit pour affichage d'un gif de chargement). Tous ne sont pas là, tel que les boutons d'export ou la légende ou encore les crédits car ces paramètres doivent être définis à la création du graphique Highchart. Une modification d'un de ces paramètres nécessite donc la reconstruction complète du graphique. Cela à été jugé trop peu performant et trop complexe à mettre en œuvre par rapport au gain ajouté.

new-edit-showIframe_js.js.twig

Le dernier template de ce répertoire est le code JavaScript commun aux templates new, edit, et showIframe. Ce code vient s'insérer dans une balise *<script>* pour ensuite faire appel aux fonctions qu'il définit. Il a été placé dans un template pour pouvoir faire appel aux fonctions définies par Twig, notamment au niveau de l'appel d'URL. On y trouve des fonctions communes aux fichiers JavaScript et aux templates, des fonctions faisant appel aux fonctions Twig, et celles qui sont internes à son fonctionnement.

Ce fichier commence avec *produceTitleWithUnit()* qui renvoi un titre avec une unité entre parenthèses en fonction de ses paramètres. C'est une fonction commune aux template new et edit pour l'axe des ordonnées, et au fichier JavaScript formSeries.js pour les séries.

On a ensuite *getData()* qui va effectuer une requête Ajax à partir de ses paramètres pour récupérer les données d'une *DataList,* et les ajouter à la variable globale *dataGlob* définie dans createChart.js.twig. Elle appelle en Ajax la route *get_data_parameterdatalist* avec l'identifiant de la *DataList* qui lui est passé en paramètre, et les attributs spatiaux si on est pas en visualisation de test. Il ne faut pas oublier que les requêtes Ajax sont asynchrones donc le temps que la requête Ajax se termine et qu'on remplisse la variable global *dataGlob*, d'autres séries pourraient vouloir s'afficher à partir des mêmes données. Or elles testent si leurs données sont présentes dans *dataGlob*, ce qui n'est pas encore la cas. On définit donc un tableau vide pour éviter qu'elles lance à leur tour la même requête Ajax. Pour finir elle fait appel à une fonction de callback passée en paramètre si celle-ci est définie. Cette fonction est utilisée pour le fonctionnement interne de ce fichier mais également par formSeries.js.

Vient après *displayParameters()* qui va afficher dans le *<select>* de parameters d'une série, les paramètres de la *DataList* sélectionnée. Elle affiche également sur le graphique le premier paramètre si on est pas en chargement de données (en édition ou lors de l'ajout de tous les paramètres d'une série). C'est une fonction propre au fonctionnement interne de ce fichier, appelée par *addAjaxQueryOnDataList()*.

Cette fonction justement ajoute un listener sur le <*select*> de la *DataList* qui lui est passé en paramètre, pour modifier la liste de ses paramètres lorsque une nouvelle *DataList* est sélectionnée. Une requête Ajax est effectuée si nécessaire, ie. si il n'y a pas encore eu de requête Ajax sur cette *DataList*. Cela se traduit par l'absence de cette *DataList* dans la variable globale *dataGlob*. Dans ce cas on définit une fonction de callback pour afficher les paramètres dans le <*select*> une fois la requête terminée, et on fait appel à *getData()*. Sinon on modifie juste le <*select*> de parameters. À noter que si c'est l'option -1 qui a été choisie, on vide les options <*select*> de parameters, et on supprime les données de la série du graphique. C'est une fonction utilisée par formSeries.js.

On a ensuite une petite fonction, *getKeyByValue()* qui renvoi la clé de la valeur passée en paramètre du tableau passé lui aussi en paramètre. Cette clé est renvoyée sous forme d'entier. Elle utilisée sur le tableau associatif *arrayCorresIndexSerieChart*, qui a comme clés *indexYAxis* * *nbSerieMaxParAxe* + *indexSerie* sous forme de chaînes de caractères. Cette fonction renvoi donc cette clé mais comme entier. Elle est utilisée par *setData()* du même fichier.

Vient après une autre petite fonction qui vérifie s'il n'y pas de valeurs de type chaînes de caractères dans la série. Pour des raisons de performances la vérification n'est effectué que sur la première valeur, puisque tel qu'à été conçue l'application toutes les données sont du même type. Cette fonction est également utilisée par *setData()* pour les graphiques Highstock, car ils ne supportent pas ce type de données.

setData() justement va afficher les données sur le graphique. Elle va commencer par faire des vérifications en fonction du type de graphique. Par exemple si c'est un graphique Highchart et qu'on a un axe logarithmique, il faut vérifier qu'il n'y a pas de valeur négative ou nulle dans la série. Si c'est le cas on change le type d'axe sinon le comportement de Highchart est de supprimer ces données de la série mais définitivement. Ce qui implique que ces données ne sont pas ré-affichées si l'on rechange le type d'axe. C'est une fonction interne au fonctionnement de ce fichier, appelée par *setSeries()*.

La dernière fonction de ce fichier est *setSeries()*. Cette fonction va faire des vérifications pour voir si l'on est en visualisation ou non car, il n'y a pas le même fonctionnement derrière :

- Lorsqu'on est en édition/création cette fonction est nécessairement appelé, après qu'on est récupéré les données de la *DataList* pour pouvoir afficher ses paramètres. Donc les données sont déjà présentes dans la variable globale *dataGlob*.
- Tandis que lorsqu'on est en visualisation il n'y pas cette première étape, et donc les données peuvent ne pas être dans la variable globale. Il faut aussi prendre en compte qu'il peut y avoir plusieurs séries qui sont sur la même *DataList.* Il y a donc une sorte de file d'attente qui se crée, pour afficher toutes les séries d'une même *DataList* une fois les données de celle-ci récupérée. C'est la fonction de callback qui va lire ces files d'attentes et afficher les données sur le graphique tant qu'elles ne sont pas vides. On trouve dans une file d'attente toutes les informations nécessaires pour afficher les données. De plus je met à jour le lien d'export des données au format CSV que j'ai ajouté.

CSS

Je redéfinis ici quelques propriétés des champs du formulaire d'édition/création. Il est noter que ce n'est pas très bon pour le côté responsive de Bootstrap, et il faut faire attention à ce que l'on fait. Les propriétés parle d'elles-mêmes je ne vais donc pas les détailler. On peut juste noter celle-ci :

```
input[type="radio"], input[type="checkbox"] {

 width: auto;

}
```

qui permet de redéfinir la taille des boutons radios et checkbox, qui étaient bien trop gros sur Chrome car défini à 100 %.

JS

Il y a six fichiers JavaScript dans ce répertoire, quatre provenant de Highchart (export-csv.js, exporting.js, highchart.js et highstock.js) et deux que j'ai écrit pour le formulaire de création/éditon (formYAxis.js et formSeries.js). Je ne vais détailler que ceux que j'ai écrit.

formYAxis.js

Le premier, formYAxis.js gère l'ajout et la suppression de sous-formulaires d'axes des ordonnées, et les listeners sur leurs champs. La première fonction *getTypeAxis()* va vérifier lorsqu'on sélectionne un axe logarithmique, qu'il n'y a pas de valeur négatives ou nulles dans les séries liées à cet axe, et renvoi sinon un axe différent. Cette fonction est d'ailleurs utilisée pour l'axe des abscisses dans le cas d'un graphique Highchart. Il est à noté que s'il y a des valeurs négatives ou nulles, elle renvoie le premier type de *availableTypeAxis*, il ne doit donc pas être logarithmique. Cette fonction est utilisé par *updateTypeAxis()* du même fichier et par les vues new et edit de Highchart et showIframe.

updateTypeAxis() va mettre à jour le type d'axe, en vérifiant les valeurs négatives ou nulles dans le cas d'un axe logarithmique. Si la condition n'est pas vérifiée, elle affiche un message d'erreur en changeant l'axe à la valeur renvoyée par *getTypeAxis()*. Elle est utilisée par les listeners et par *setData()* de new-edit-showIframe_js.js.twig.

Vient après deux fonctions pour vérifier s'il n'y a pas de valeurs négatives ou nulles, la première faisant appel à l'autre.
checkNotNegativeValue() est appelée sur un ensemble de séries Highchart. Elle va donc les parcourir et appeler sur chacune d'entre elles *negativeValue(),* qui renvoie *true* si elle trouve une valeur négative dans la série qui lui est passée en paramètre.
negativeValue() doit vérifier si on lui a passé une série Highchart ou une série Json qui n'a pas encore été ajoutée.

Pour finir on a les trois fonctions qui vont contrôler l'ajout et la suppression de sous-formulaires d'axes des ordonnées, à savoir *ajouterYAxis()*, *ajouterLienSuppressionYAxis()* et *addControlNewDeleteYAxis()*.

ajouterYAxis() va commencer par récupérer le contenu de l'attribut *data-prototype* qui contient le sous-formulaire d'axes des ordonnées, pour l'ajouter avant le bouton d'insertion d'axe. On met à jour le tableau

arrayCorresYAxisChart, et on ajoute un axe à Highchart. On rajoute les listeners et le lien de suppression. Si l'on est en création, on ajoute également le contrôle de l'ajout et de la suppression de séries pour cet axe. Cette fonction est appelée par *addControlNewDeleteYAxis()* et par edit du répertoire Highchart/.
ajouterLienSuppressionYAxis() va simplement ajouter le lien de suppression pour l'axe passé en paramètre et son listener. Lorsqu'on supprime un axe il faut commencer par supprimer toutes ses séries. On parcours donc l'ensemble de la plage de valeurs pour un axe dans *arrayCorresIndexSerieChart*, et pour chaque index défini et supérieur à -1, on supprime la série. On supprime ensuite l'axe sur le graphique et on met à jour *arrayCorresYAxisChart*. Il reste plus qu'à supprimer le sous-formulaire. Elle est appelée par *ajouterYAxis()*.
Enfin *addControlNewDeleteYAxis()* va commencer par créer le lien d'ajout d'axe des ordonnées (en création puisque c'est déjà fait en édition), récupère ensuite le nombre d'axe des ordonnées qui existent et définit le listener d'ajout d'axe. Elle est appelée par les vues new et edit de Highchart.

formSeries.js

Le second fichier JavaScript, formSeries.js, réalise la même chose que formYAxis.js mais sur les séries.
Il commence par définir une fonction *getDashStyle(),* qui corrige un petit bug de Highchart : lorsqu'on définit explicitement le style 'Solid' il affiche un style 'Dot Dot', tandis que la valeur par défaut est bien 'Solid'. Cette fonction renvoi simplement *null* si on lui demande le style 'Solid', afin que Highchart utilise la valeur par défaut.

Vient après *updateTitleUnitWithParameter()* qui va mettre à jour les champs *title* et *unit* de la série en fonction du paramètre sélectionné.

preSelectDataListParameterColorNewSerie() permet, lors du clic sur le bouton ajouter une série, de récupérer la *DataList* et le paramètre associé de la série précédente du même axe, pour les pré-remplir avec le paramètre suivant. De même avec la couleur. Pour cela on commence par parcourir *arrayCorresIndexSerieChart* pour trouver la série précédente, et si elle existe et qu'on est pas en édition, on fait le changement. On passe en mode synchrone pour les requêtes Ajax, afin de ne pas afficher sur le graphique le premier paramètre. Puis on récupère les valeurs de la série précédente, pour calculer le paramètre suivant avec un modulo sur le nombre de paramètre de la *DataList*. On calcule également la couleur suivante en essayant de les rendre les plus différentes possibles. Les couleurs sont rangés dans l'ordre et prendre la couleur directement suivante ne donnera pas un affichage très visible, on prend donc le nombre de couleurs total divisé par le nombre de paramètres de la *DataList*. Cette fonction est appelée par *ajouterSerie()*.

ajouterSerie() fait la même chose que *ajouterYAxis()* à savoir ajouter le sous-formulaire de série, ajouter la série dans Highchart et les listeners sur les champs du formulaire. Elle récupère donc son attribut *data-prototype,* et ajoute le sous-formulaire avant le bouton d'insertion de série. Elle met à jour *arrayCorresIndexSerieChart* et ajoute la série dans Highchart. Elle s'occupe ensuite des listeners, à noter que le listener du champ *DataList* est la fonction *addAjaxQueryOnDataList()* de new-edit-showIframe-js.js.twig. Elle ajoute les styles sur la balise *select* de color et le lien de suppression. Enfin si ce n'est pas la première série de l'axe, elle pré-sélectionne le paramètre suivant de la *DataList* et la couleur suivante par rapport à la série précédente de l'axe. Cette fonction est appelée par *ajouterAllSeries()* et *addControlNewDeleteSeries()* du même fichier et par les vues edit.

On a ensuite *ajouterAllSeries(),* qui ajoute toutes les séries associées à chaque paramètre d'une *DataList* sélectionnée. Cette fonction me semble comporter une erreur : elle fait quoi qu'il arrive un appel Ajax pour récupérer les données de la *DataList*, sans vérifier dans *dataGlob* si elle existait déjà. Il faudrait modifier cela je pense. Elle a un fonctionnement assez simple, pour chaque paramètre de la *DataList* elle appelle *ajouterSerie()*. Elle définit au début une variable *firstSerie* à *true,* pour que le premier appel à *ajouterSerie()* ne fasse pas la pré-sélection du paramètre et de la couleur. Après ce premier appel, il est passé à *false* pour que tout les appels suivants fasse la pré-sélection et ainsi donne l'ensemble des paramètres. Il est à noter que l'index des séries doit être modifier, or on ne peut le faire que dans la fonction de callback, et une fonction de callback ne peut renvoyer quelque chose par définition. Une solution que j'ai trouvé est de passer par un objet puisque c'est la référence de l'objet qui est passé. Mais dans tout le reste du fichier c'est un entier qui est utilisé. J'ai donc fait un tour de passe-passe avec une deuxième fonction de callback, appelée par la première pour mettre à jour l'index. *ajouterAllSeries()* est appelée par *addControlNewDeleteSerie()*.

Vient ensuite *removeSerieChart()* qui va s'occuper de maintenir à jour *arrayCorresIndexSerieChart* lors de la suppression d'une série. Il faut décrémenter toutes séries qui ont un index côté graphique supérieur à celui qu'on supprime. Elle est appelée par le listener définit dans *ajouterLienSuppressionSerie()* et par celui définit dans *ajouterLienSuppressionYAxis()* dans formYAxis.js.

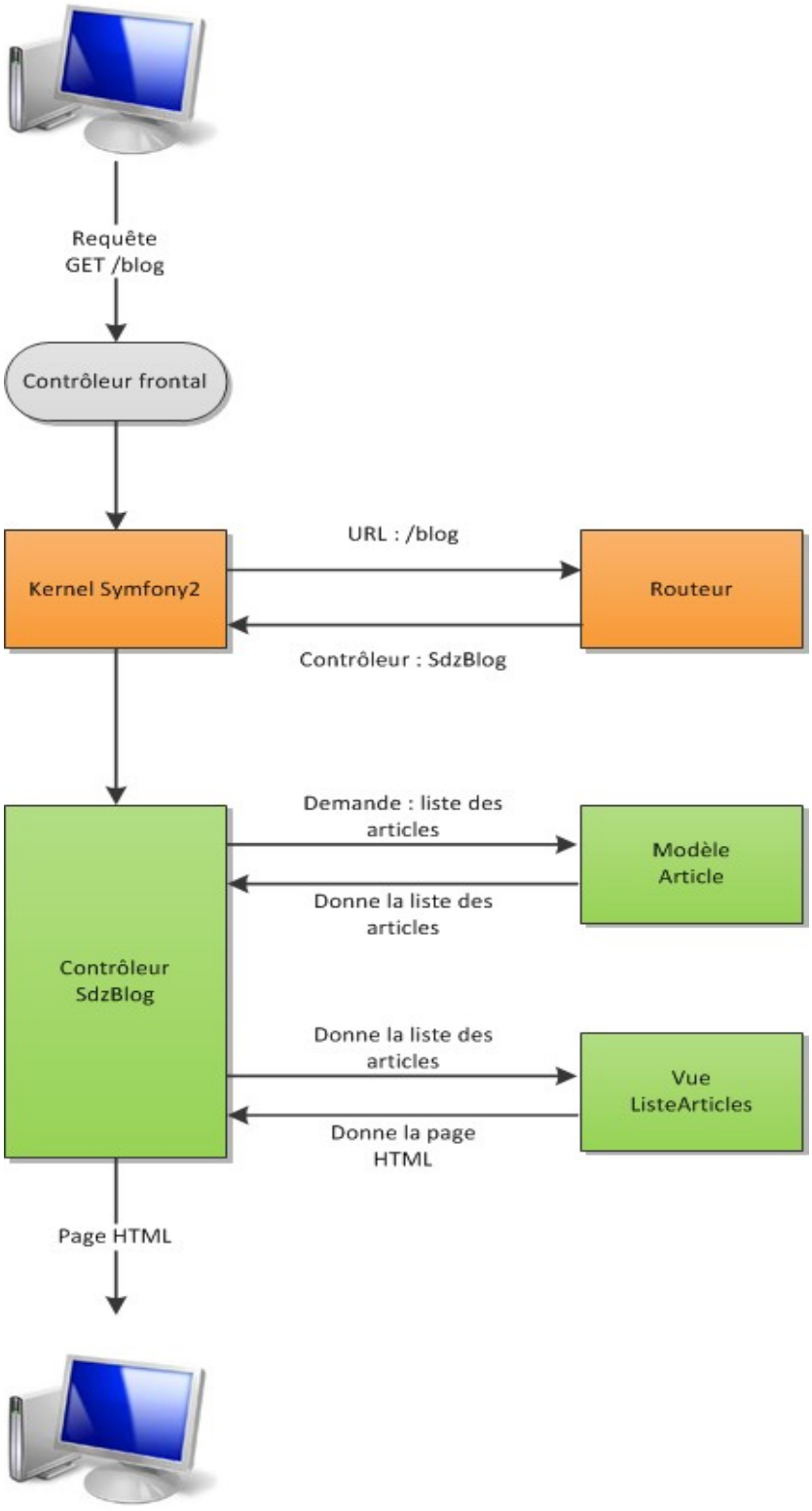
ajouterLienSuppressionSerie() va donc ajouter le bouton de suppression pour la série, et définir son listener qui va faire appel à *removeSerieChart()* et supprimer le sous-formulaire. Elle est appelée par *ajouterSerie()*.

addControlNewDeleteSeries() va définir les deux liens pour ajouter des séries si l'on est en création, puisque c'est déjà fait en édition. Ensuite elle va récupérer le nombre de séries déjà définies, et définir les listeners sur les boutons d'ajout d'une série, et d'ajout de toutes les séries d'une *DataList*. Pour ce dernier listener, on crée un objet pour l'index afin que la fonction de callback de *ajouterAllSeries()* puisse le modifier, et une fonction de callback pour mettre à jour l'index. *AddControlNewDeleteSeries()* est appelée par *ajouterYAxis()* de formYAxis.js et par les vues edit.

Enfin il a la fonction *addColor()* qui va parcourir la balise <*select*>, pour ajouter le style *background-color* sur toutes les options et sur la balise pour visualiser la couleur. Elle définit également un listener sur la balise pour changer sa couleur lorsqu'elle change de valeur. Elle est appelée par *ajouterSerie()*.

III) Annexe

1) Annexe 1 : parcours d'une requête



2) Annexe 2 : commandes Symfony

autocomplétion : tab

##########
Symfony
##########
DEV

vider le cache
php app/console cache:clear
ou
sudo rm -rf app/cache/* app/logs/*
sudo chmod -R 777 app/cache app/logs

création bundle
php app/console generate:bundle
nameSpace : {{NomDuSite}}/{{NomBundle}}Bundle
nom : par défaut (entrée)
répertoire destination : par défaut (entrée)
format de configuration : yml
struture à générer : yes
confirmer génération : yes

ajout d'un Bundle existant sur le net
rajout de la ligne correspondante au bundle dans la section "require" du composer.json, puis
php composer.phar update
enfin l'activer en rajoutant new Le\Nouveau\Bundle(), dans app/AppKernel.php

mettre à jour fichiers de traduction
php app/console translation:update --dump-messages {{locale}} {{NomBundle}}

```
## mettre a jour liens symboliques
php app/console assets:install web/ --symlink

## PROD ##

## importation ressources
php app/console assetic:dump --env=prod

## vider le cache
php app/console cache:clear –env=prod

###############
## FosUserBundle ##
###############

## ajouter un user ( --super-admin )
php app/console fos:user:create

## le desactiver
php app/console fos:user:deactivate {{user}}

## l'activer
php app/console fos:user:activate {{user}}

## ajouter un role ( --super pour super admin)
php app/console fos:user:promote {{user}} {{ROLE}}

## supprimer un role ( --super pour super admin)
php app/console fos:user:demote {{user}} {{ROLE}}

## changer le password
php app/console fos:user:change-password {{user}} {{newPassword}}
```

```
##########
## Doctrine ##
##########

## création BDD ("Symfony")
php app/console doctrine:database:create

## création entity
php app/console doctrine:generate:entity
# nom                   : {{NameSpaceBundle}}:{{Entity}}
# configuration format  : par défaut (entrée)
# field name            : {{nomAttribut}}
# field type            : {{typeAttribut}}
# stop ajout field      : touche entrée sur field name
# générer repository    : yes
# confirmer génération : yes

## mise à jour entity (sauvegarde : {{nomFichier}}~)
php app/console doctrine:generate:entities {{NameSpaceBundle}}:{{Entity}}

## mise à jour BDD (affichage requêtes)
php app/console doctrine:schema:update --dump-sql

## mise à jour BDD (exécution requêtes)
php app/console doctrine:schema:update --force

## création d'un constructeur de formulaire : xxxType.php
php app/console doctrine:generate:form {{NameSpaceBundle}}:{{Entity}}
```