Symfony

Symfony est un ensemble de composants réutilisables pour PHP qui permet de déployer rapidement des applications web.

Créer un projet

Avant de démarrer un projet, assurez vous de connaitre les bonnes pratiques de développement de projet.

Installer l'outils `symfony` :

```
  $ curl -LsS http://symfony.com/installer > symfony.phar
  $ mv symfony.phar ~/bin/symfony
  $ chmod a+x ~/bin/symfony
```

Créer le squelette de l'application :

```
  $ symfony new mon_application
```

Supprimer le bundle de démo :

```
  A default bundle, AcmeDemoBundle, shows you Symfony2 in action. After
playing with it, you can remove it by following these steps:
   * delete the src/Acme directory;
   * remove the routing entries referencing AcmeBundle in
app/config/routing_dev.yml;
   * remove the AcmeBundle from the registered bundles in app/AppKernel.php.
```

Connecter une base de données

```
  $ composer require propel/propel-bundle "~2.0@dev"
```

app/AppKernel.php

```
$bundles = array(
    // …
    new Propel\PropelBundle\PropelBundle(),
);
```

app/config/parameters.yml

```
parameters:
    database_driver: pdo_mysql
    database_host: 127.0.0.1
    database_port: null
    database_name: mon_application
    database_user: utilisateur
    database_password: mon_mot_de_passe
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    locale: en
    secret: xyz
```

app/config/config.yml

```
propel:
    dbal:
        driver:               %database_driver%
        user:                 %database_user%
        password:             %database_password%
        dsn:
%database_driver%:host=%database_host%;dbname=%database_name%;charset=U
TF8
        options:              {}
        attributes:           {}
```

```
$ mkdir -p src/AppBundle/Resources/config
```

src/AppBundle/Resources/config/schema.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="default" namespace="AppBundle\Model"
defaultIdMethod="native">
</database>
```

```
$ php app/console propel:database:create
Use connection named default in dev environment.
Database mon_application has been created.
$ php app/console propel:build
```

Après de nombreux essais, je ne réussis toujours pas à faire fonctionner `Propel` avec `SQLite`.
Cela me gène car j'aime faire les *dév* avec `SQLite`.
En pratique, ce n'est pas un problème car de toute façon la *prod* est en `MySQL` ou autre SGBD.

Travailler avec la base de données

```
$ vim src/AppBundle/Resources/config/schema.xml
```

src/AppBundle/Resources/config/schema.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="default" namespace="AppBundle\Model"
defaultIdMethod="native">

    <table name="user">
        <column name="id" type="integer" required="true"
primaryKey="true" autoIncrement="true" />
        <column name="first_name" type="varchar" size="100" />
        <column name="last_name" type="varchar" size="100" />
    </table>

</database>
```

```
$ php app/console propel:model:build
>>        AppBundle    Generated model classes from schema.xml
$ php app/console propel:migration:generate-diff
…
"PropelMigration_1426166067.php" file successfully created in […]
/app/propel/migrations
…
$ php app/console propel:migration:status
$ php app/console propel:migration:migrate
```

Pensez à `commit` le schéma et le code généré dans `app/propel/migrations/` et `src/AppBundle/Model/`.

Charger des données

Pour des données “normales” à insérer dans la base après sa création on peut utiliser les `fixtures` de Propel :

```
$ mkdir src/AppBundle/Resources/fixtures/
```

src/AppBundle/Resources/fixtures/001-users.yml

```
AppBundle\Model\User:
    u1:
        first_name: Jean
        last_name: Dupond
    u2:
```

```
        first_name: Marie
        last_name: Dupont
```

```
$ php app/console propel:fixtures:load @AppBundle
```

Faites attention en utilisant la méthode ci-dessus, car **la table sera vidée** avant d'insérer les fixtures !
Pour avoir **des données de test**, il vaut mieux utiliser la méthode ci-dessous.

Pour ajouter automatiquement de fausses données, il vaut mieux utiliser le bundle approprié :

```
$ composer require "willdurand/faker-bundle" "@stable"
```

app/AppKernel.php

```
if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    // …
    $bundles[] = new Bazinga\Bundle\FakerBundle\BazingaFakerBundle();
}
```

app/config/config_dev.yml

```
bazinga_faker:
    seed:       1234
    locale:     fr_FR
    entities:
        AppBundle\Model\User:
            number: 5
```

```
$ php app/console faker:populate
```

Tester le code

Comme indiquer dans les bonnes pratiques, il vaut mieux mettre en place des tests le plus tôt possible. Ceci se fait assez simplement une fois `phpunit` installé.

On écrit un fichier de test par contrôleur.

src/AppBundle/Tests/Controller/DefaultControllerTest.php

```
<?php
```

```php
namespace AppBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    public function urlProvider()
    {
        return array(
            array('/'),
        );
    }

    /**
     * @dataProvider urlProvider
     */
    public function testPageIsSuccessful($url)
    {
        $client = self::createClient();
        $client->request('GET', $url);
        $this->assertTrue($client->getResponse()->isSuccessful());
    }

    public function testPrivate()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/private');
        $this->assertEquals(302,
$client->getResponse()->getStatusCode());
    }
}
```

Une fois le test démarré, toutes les méthodes `test*()` existantes seront appelées.

```
$ phpunit -c app/
```

Démarrer le serveur de test

Le serveur de test permet de s'affranchir des problèmes de permissions que l'on peut avoir quand on utilise Apache|Lighttpd|Nginx...

```
$ php app/console server:run
Server running on http://127.0.0.1:8000

Quit the server with CONTROL-C.
```

On peut maintenant passer au développement.

Ajouter des pages

Dans symfony, les contrôlleurs sont rangés dans des modules nommés *Bundle*. Les modules, s'ils sont bien développés, peuvent être utilisés indépendemment dans d'autres applications.

Un **bundle par défaut** est disponible, mais d'autres peuvent être créés facilement :

```
  $ php app/console generate:bundle --namespace=Bioaster/TestBundle --
format=annotation --dir=src --no-interaction
```

Sauf raison valable, il est conseillé de travailler dans `src/AppBundle`.

La première étape est de **router** l'URL de la page demandée jusqu'au contrôleur. Ceci est fait grâce à 2 variables :

- le préfix où le bundle est ancré : app/config/routing.ymlbioaster_test:
  resource: "@BioasterTestBundle/Controller/"
  type: annotation
  prefix: /
- la route définie pour chaque méthode du contrôleur :
  src/Bioaster/TestBundle/Controller/DefaultController.php class DefaultController extends Controller
  {
  /**
  * @Route("/hello/{name}")
  */
  public function indexAction($name)
  {
  [...]

Dans l'exemple ci-dessus, la méthode `indexAction` est accessible par l'URL `/hello/mon_nom`.

En définissant un préfix a `/test` , la même action aurait été accessible à `/test/hello/mon_nom`.

Les routes disponibles peuvent facilement être retrouvées :

```
  $ php app/console router:debug
  [router] Current routes
   Name                         Method Scheme Host Path
   …
   bioaster_test_default_index  ANY    ANY    ANY  /test/hello/{name}
   homepage                     ANY    ANY    ANY  /app/example
```

Une URL peut aussi être testée pour voir quelle action de quel contrôleur est appelée :

```
$ php app/console router:match /test/hello/test
Route "bioaster_test_default_index" matches

[router] Route "bioaster_test_default_index"
Name         bioaster_test_default_index
Path         /test/hello/{name}
Path Regex   #^/test/hello/(?P<name>[^/]++)$#s
…
Defaults     _controller: BioasterTestBundle:Default:index
```

Il reste maintenant à ajoutre de nouvelles routes, de nouveaux contrôleurs, de nouvelles actions et de nouvelles vues.

src/Bioaster/TestBundle/Controller/TestController.php

```php
<?php

namespace Bioaster\TestBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class TestController extends Controller
{
    /**
     * @Route("/hello2/{name}", name = "test2")
     */
    public function indexAction($name)
    {
        return $this->render('default/index.html.twig', array(
            'name' => $name
        ));
    }
}
```

Il est important de garder chaque action du contrôleur assez courte (10-20 lignes) et d'implémenter la logique de manipulation de données dans le modèle pour respecter la philosophie de symfony : *“thin controllers and fat models”*.

Avec Propel, le code s'ajoute par exemple à `src/AppBundle/Model/User.php`.

Ajouter un formulaire

- http://symfony.com/doc/current/best_practices/forms.html
- http://propelorm.org/Propel/cookbook/symfony2/mastering-symfony2-forms-with-propel.html

Authentifier les utilisateurs

C'est facile, il suffit d'utiliser le bundle `BioasterCertLogin`... enfin, dès qu'il sera écrit !

- Créer un UserProvider pour récupérer les données de la BD ; c'est facile avec Propel !
- Créer un certificateAuthenticator pour récupérer les données de certificat passé par le serveur web.

Créer une interface d'administration

Mais est-ce bien nécessaire ?

- https://github.com/symfony2admingenerator/AdmingeneratorGeneratorBundle
- https://github.com/smirik/SmirikPropelAdminBundle/blob/master/Resources/doc/index.md

Bonne pratiques

Cette liste de bonnes pratiques est tirée de la page "best practices" de Symfony.

- Introduction
  - you should not refactor your existing applications to comply with these best practices.
- Creating the project
  - Create only one bundle called AppBundle for your application logic
  - There is no need to prefix the AppBundle with your own vendor (e.g. AcmeAppBundle), because this application bundle is never going to be shared.
  - `$ php app/console generate:bundle –namespace=AppBundle –dir=src –format=annotation –no-interaction`
  - app/Resources/, stores all the templates and the translation files
- Configuration
  - Define the infrastructure-related configuration options in the app/config/parameters.yml file.
  - Define all your application's parameters in the app/config/parameters.yml.dist file.
  - Define the application behavior related configuration options in the app/config/config.yml file.
  - Use constants to define configuration options that rarely change.
  - Moving Sensitive Options Outside of Symfony Entirely
- Business Logic
  - “Services: Naming and Format”
- Controllers
  - Symfony follows the philosophy of “thin controllers and fat models”
  - Use the ParamConverter trick to automatically query for Doctrine|Propel entities when it's simple

and convenient.
    - Pre and Post Hooks

- Templates
    - Store all your application's templates in app/Resources/views/ directory.
    - Define your Twig extensions in the AppBundle/Twig/ directory and configure them using the app/config/services.yml file.
    - ex: a custom md2html Twig filter

- Forms
    - Define your forms as PHP classes... EntityType and createForm()
    - http://propelorm.org/Propel/cookbook/symfony2/mastering-symfony2-forms-with-propel.html
    - Add buttons in the templates, not in the form classes or the controllers.
    - Handling Form Submits [...] we recommend using $form→isSubmitted() in the if statement for clarity.
    - (Always redirect after post!)

- I18n
    - Use the XLIFF format for your translation files.
    - Always use keys for translations instead of content strings.

- Security
- Use the bcrypt encoder for encoding your users' passwords.
- Whenever possible, use the @Security annotation
- If your security logic is complex and can't be centralized into a method like isAuthor(), you should leverage custom voters.
- with Propel http://propelorm.org/documentation/cookbook/symfony2/the-symfony2-security-component-and-propel.html

- Web-assets
    - Store your assets in the web/ directory.
    - Use Assetic to compile, combine and minimize web assets

- Tests
    - Define a functional test that at least checks if your application pages are successfully loading.

Divers

- Routing
    - @Route(“/blog/{page}”, defaults={“page”: 1}, requirements={“page”: “\d+”})
    - @Method(“POST”)

- JPEG optimize

- Testing

- Faker
    - Populating Entities Using an ORM