Notes (06/11/2016)

Préparer un conteneur Apache + PHP :

Sur ma machine Mac Book :

1. Démarrer Docker
2. Exécuter les commandes suivantes dans un terminal (1) :

```
$ cd /Volumes/Data/Code/

$ mkdir Udemy && cd Udemy

$ mkdir learn_symfony_3

$ docker run --name symfony -p 8001:80 -p 443:443 -v /Volumes/Data/Code/Udemy/
learn_symfony_3/:/home/disk/ -d eboraas/apache-php

$ docker start mysqlsgbd
```

Ensuite rentrer dans le conteneur “symfony” :

```
$ docker exec -it symfony bash

$ php -v # il faut avoir une version >= 5.6

$ cd /home/disk

$ apt-get update && apt-get install -y vim git

$ a2enmod rewrite

$ vim /etc/apache2/apache2.conf # and put AllowOverride on <Directory /var/www>
conf

$ service apache2 reload
```

Ouvrir un autre terminal (2) et exécuter les commandes suivantes :

```
$ which composer # pour récupérer le chemin de cet exécutable

$ cp /usr/local/bin/composer /Volumes/Data/Code/Udemy/learn_symfony_3/composer
```

Terminal (1) :

```
$ cp composer /usr/local/bin/composer

$ composer -version #tu dois voir la version du composer
```

Récupérer l’adresse IP du serveur MySql “mysqlsgbd “; Terminal (2) :

```
$ docker inspect mysqlsgbd
```

Puis localiser la clé “**Networks.bridge.IPAddress**” : `172.17.0.3`
Les accès au SGBD :

- username : root
- password : root

Création du projet “autotrader” :

In Terminal (2) :
Création d’une base de données avec le nom “autotrader” :

```
$ docker exec -it mysqlsgbd bash
$ mysql -u root -proot
$ mysql > create database autotrader;
$ mysql > exit;
$ exit
```

In Terminal (1) :

```
$ composer create-project symfony/framework-standard-edition autotrader
```

Par la suite répond aux questions suivantes :

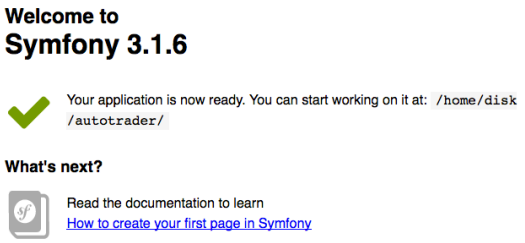
```
database_host (127.0.0.1): 172.17.0.3
database_port (null): 3306
database_name (symfony): autotrader
database_user (root): root
database_password (null): root
mailer_transport (smtp):
mailer_host (127.0.0.1):
mailer_user (null):
mailer_password (null):
secret (ThisTokenIsNotSoSecretChangeIt):
```

Faire pointer le dossier `/var/www/html` vers le dossier `web` du projet.

```
$ rm -r /var/www/html
```

```
$ ln -s /home/disk/autotrader/web /var/www/html
```

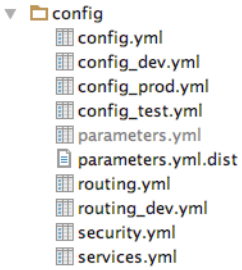
Puis à partir du navigateur internet de ma machine j'accède à l'adresse web suivante : `http://localhost:8001/app.php`
Tu dois voir une écran qui ressemble à cette capture :



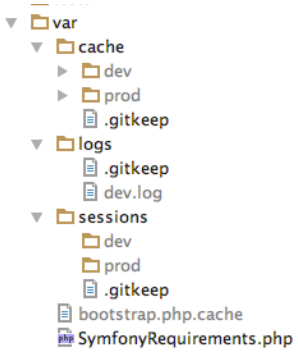
Ensuite ouvrir le projet dans PHPStorm.
Pour activer l'accès à "http://localhost:8001/app_dev.php" à partir de ma machine il faut commenter dans le fichier `web/app_dev.php` le code suivant comme suit :

```
if ( isset( $_SERVER['HTTP_CLIENT_IP'] ) ||

isset( $_SERVER['HTTP_X_FORWARDED_FOR'] ) ||

! ( in_array( @$_SERVER['REMOTE_ADDR'], [ '127.0.0.1', '::1' ] ) ||

php_sapi_name() === 'cli-server' )) {

    header( 'HTTP/1.0 403 Forbidden' );

    exit( 'You are not allowed to access this file. Check ' . basename(
__FILE__ ) . ' for more information.' );

}
```

Environnements :
Dans Symfony il existe trois types d'environnements : dev, prod et test



Le format utiliser est `<filename>_<env_name>.<yml>`
Dans le dossier **var** du projet chaque sous dossier contiendera un sous-dossier par environnement.



Symfony offre une console avec des commandes :

```
$ php bin/console # pour lister toutes les commandes possible
```

Pour comprendre le fonctionnement d'une commande Symfony :

```
$ php bin/console <cmd_name> --help
```

Bundles :

Dans Symfony 3 il faut mettre par défaut les vues Twig dans le dossier `app/Resources/views` et pas dans les bundles dans `src`.

GÉNÉRER UN BUNDLE :

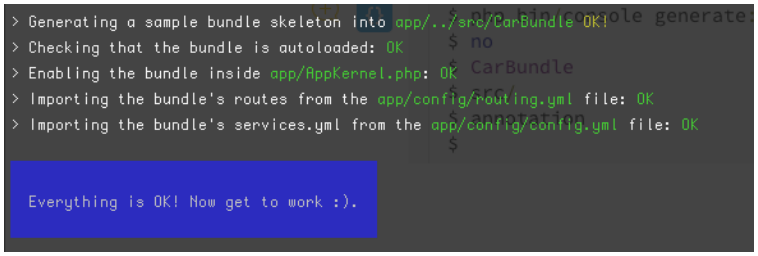
Il existe deux types de bundle dans Symfony :

- Bundle partagé avec plusieurs application
- Bundle propre au projet : application bundle

La commande pour créer un nouveau bundle est :

```
$ php bin/console generate:bundle #puis répondre par
$ no
$ CarBundle
$ src/
$ annotation
```

Et voilà le bundle est bien généré :



NB :
Dans le fichier `config/routing.yml` les deux bundles "AppBundle" and "CarBundle" on le même préfix. Pour qu'ils fonctionnent bien par la suite il faut changer le préfixe du nouveau bundle en `/car`

INSTALLER ET CONFIGURER DES BUNDLES EXTERNES :

Sources :

- Github, bitbucket, ...
- Local machine

Ex : KnpMenuBundle

- Composer :

```
$ composer require knplabs/knp-menu-bundle dev-master
```

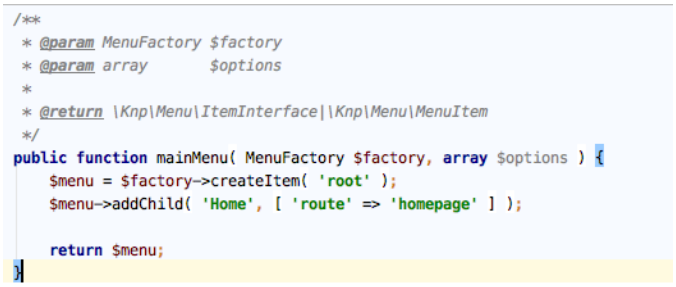
- AppKernel :

Ajouter `new \Knp\Bundle\MenuBundle\KnpMenuBundle()` au tableau `$bundles` dans le fichier `AppKernel.php`

- Build menu :

Vu qu'il sera partagé dans toute notre application, il faut le créer dans le bundle **AppBundle**.

1. Dans le dossier src/AppBundle créer un sous-dossier **Menu**
2. Dans ce dossier (Menu) tu crée une classe **Builder**. Et ajouter dedans la méthode suivante :



```php
/**
 * @param MenuFactory $factory
 * @param array       $options
 *
 * @return \Knp\Menu\ItemInterface|\Knp\Menu\MenuItem
 */
public function mainMenu( MenuFactory $factory, array $options ) {
    $menu = $factory->createItem( 'root' );
    $menu->addChild( 'Home', [ 'route' => 'homepage' ] );

    return $menu;
}
```

3. **NB :** le nom de la route **homepage** est déclarer dans le controleur DefaultController du bundle AppBundle.

- Show it :

Open the `app/Resources/views/base.html.twig` and declare the menu using `knp_menu_render` twig helper comme suit :

```twig
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
        {{ knp_menu_render('AppBundle:Builder:mainMenu') }}
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

- Check it :

  Pour voir le menu allez sur l'adresse http://localhost:8001/app_dev.php/

TIPS ON MENU:

Pour garder le menu sélectionné pour certaines routes il faut ajouter une autre option en plus de 'route' comme suit :

```
'extras' => [ 'routes' => [  ['route'=>''], ['route'=>''], ...]]
```

Ex : Pour garder le menu 'Offre' sélectionné aussi lorsqu'on visite la page "Show details" on mets :

```php
$menu->addChild( 'Offer',
    [
        'route'  => 'offer',
        'extras' => [
            'routes' => [
                [ 'route' => 'show_car' ],
            ],
        ],
    ] );
```

On peut passer des paramètres à notre route principale et nos extras routes :

```php
$menu->addChild('Category', [
    'route' => 'category_show',
    'routeParameters' => ['slug' => $category->getSlug()],
    'extras' => [
        'routes' => [
            [
                'route' => 'thread_show',
                'parameters' => ['categorySlug' => $category->getSlug()]
            ],
        ],
    ],
]);
```

Le fait de passer les paramètres à des routes paramétrables, c'est seulement pour choisir un url au lieu de sélectionner le menu pour tous les urls issu de cette route.

Par exemple ce code :

```
$menu->addChild( 'Manage Cars',
    [
        'route'  => 'car_index',
        'extras' => [
            'routes' => [
                [ 'route' => 'car_new' ],
                [ 'route' => 'car_show' ],
                [ 'route' => 'car_edit', 'parameters' => [ 'id' => '2' ] ],
            ],
        ],
    ] );
```

Permet seulement de sélectionner le menu “Manage Cars” lorsqu’on est sur la page d’édition de la voiture avec l’id 2.

NB : Ici $category est une instance d’une entity.

Styling : Twitter bootstrap

Récupérer les liens CDN (css, js) du framework bootstrap du site officiel : http://getbootstrap.com/getting-started/#download-cdn
Puis les ajouter respectivement dans les bloques “`stylesheets`” et “`javascripts`” du fichier app/Resources/views/base.html.twig.
Ensuite supprimer le bloque “stylesheets” dans le fichier `app/Resources/views/default/index.html.twig`
Le fichier deviendera comme suit :

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

{% block body %}
    <h1>Welcome to AutoTrader</h1>
{% endblock %}
```

Puis ajouter le marque du menu Bootstrap dans le fichier base.html.twig :

```
<nav class="navbar navbar-default">
<div class="container-fluid">
<!-- Brand and toggle get grouped for better mobile display -->
<div class="navbar-header">
<button type="button" class="navbar-toggle collapsed"
        data-toggle="collapse" data-target="#bs-example-navbar-collapse-1"
        aria-expanded="false">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        </button><a class="navbar-brand" href="#">Brand</a>
```

```
 </div>
        <!-- Collect the nav links, forms, and other content for toggling -->
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
         {{ knp_menu_render('AppBundle:Builder:mainMenu', { 'currentClass':
'active' }) }}
        </div>
  </div>
</nav>
```

Ensuite il faut ajouter une classe css dans le Builder du menu comme suit :

```php
/**
 * @param MenuFactory $factory
 * @param array       $options
 *
 * @return \Knp\Menu\ItemInterface|\Knp\Menu\MenuItem
 */
public function mainMenu( MenuFactory $factory, array $options ) {
    $menu = $factory->createItem( 'root' );
    $menu->setChildrenAttribute( 'class', 'nav navbar-nav' );
    $menu->addChild( 'Home', [ 'route' => 'homepage' ] );

    return $menu;
}
```

Controllers and templating :

Un contrôleur est un callable. La classe des contrôlleurs sert à regrouper plusieurs contrôleurs.

1. Il faut changer la route vers l'action index du contrôleur CarController vers "/our-cars" avec le nom "offer". Puis ajouter une nouvel élément de menu dans le builder.
2. Il faut aussi extends le template du controleur Car comme c'est le cas avec le fichier index.html.twig dans le dossier Resources/views/default. Sinon cette vue n'aura pas le même layout que notre application.

TWIG :

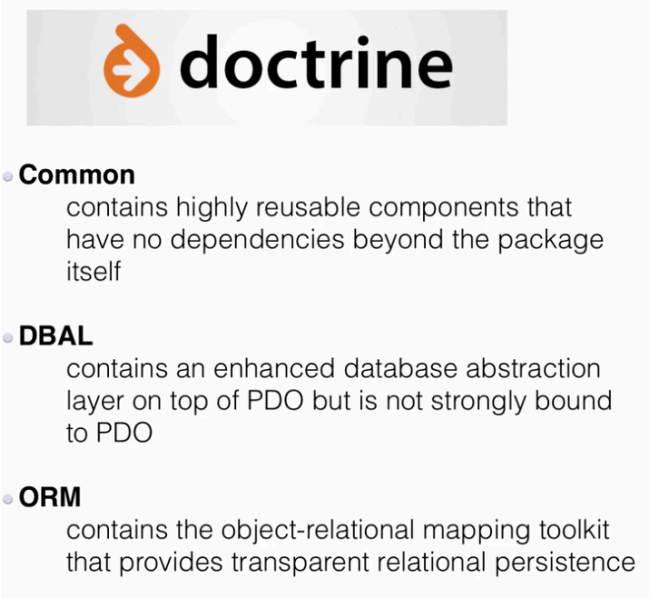
Dans twig il exist trois type de tags :

- {% logic %} : blocks, flow structure, ...
- {{ value or expression }} : pour afficher une valeur ou la valeur d'une expression
- {# comment #} : pour les commentaires

ENVOYER DES DONNÉES À LA VUE :

En passe à la fonction render un tableau (key, value) comme deuxième paramètre. Au niveau de la vue on accède à ces données en utilisant les key.

Doctrine :



Doctrine utilise trois concepts :

- `EntityManager`
- `Entity`
- `Repository`

Les paramètres et la configuration de la base de données se trouvent dans le fichier `app/config/parameters.yml` et `app/config/config.yml`

CREATE DATABASE :

Pour créer la base de données configurée dans les paramètres de l'application (si elle n'existe pas) :

```
$ php bin/console doctrine:database:create
```

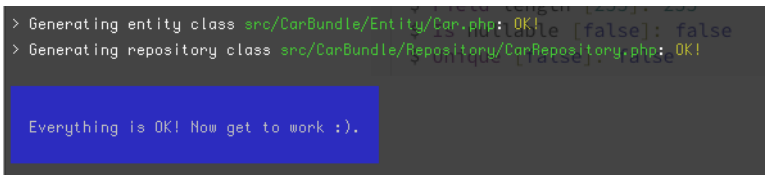
Pour supprimer la base de données configurée :

```
$ php bin/console doctrine:database:drop --force
```

ENTITY :

C'est un classe PHP Plain old object avec une identité.
C'est une classe doctrine qui représente nos données.
Pour générer une entity via la ligne de commande en utilise : `$ php bin/console doctrine:generate:entity`
Ex :

```
$ New field name : name
$ Field type [string]: string
$ Field length [255]: 255
$ Is nullable [false]: false
$ Unique [false]: false

$ New field name : make
$ Field type [string]: string
$ Field length [255]: 255
$ Is nullable [false]: false
$ Unique [false]: false $ php bin/console doctrine:generate:entity
$ The Entity shortcut name: CarBundle:Car
$ Configuration format (yml, xml, php, or annotation) [annotation]:annotation

$ New field name : name
$ Field type [string]: string
$ Field length [255]: 255
$ Is nullable [false]: false
$ Unique [false]: false

$ New field name : make
$ Field type [string]: string
$ Field length [255]: 255
$ Is nullable [false]: false
$ Unique [false]: false
```



Pour mettre à jour la schéma de la base de données :

```
$ php bin/console doctrine:schema:update --force
```

REPOSITORIES :

```php
$rep = $this->getDoctrine()
            ->getEntityManager()
            ->getRepository( Car::class );

$cars = $rep->findAll();
```

POUR AJOUTER UN NOUVEL ATTRIBUT À NOTRE ENTITÉ :

1. Ajouter un attribut `private` à notre classe
2. Ajouter les annotations Doctrine
3. Générer les `Getters/Setters`
4. Mettre à jour le schéma de la base de données.

RELATIONS ENTRE LES ENTITÉS :

```
Model ← Car → Make
```

Ex :
Dans l'entité `Car` :

```php
/**
 * @var Model
 *
 * @ORM\ManyToOne(targetEntity="CarBundle\Entity\Model", inversedBy="cars")
 */
private $model;
```

Dans l'entité `Model` :

```php
/**
 * @var ArrayCollection
 * @ORM\OneToMany(targetEntity="CarBundle\Entity\Car", mappedBy="model")
 */
private $cars;
```

Pour générer les Getters/Setters via la ligne de commande on fait :

```
$ php bin/console doctrine:generate:entities CarBundle
```

Ensuite il faut mettre à jour le schéma de la base de données.

LAZY LOADING AND PROXIES OBJECTS :

C'est une technique de lecture des données par Doctrine.
Dans ce cas les entités relation ne sont pas récupérées avec l'entité principale. Mais au moment de l'accès à ces relations dans le code que Doctrine exécute les requêtes SQL vers la base de données. Son inconvénient est qu'il génère plusieurs requêtes vers la base de données au lieu de récupérer toutes les données dans une seule requête.

Pour celà on peut utiliser les requêtes personnalisées.

LES REQUÊTES PERSONNALISÉES (JOINTURES) :

Les requêtes personnalisées sont ajoutées dans les classes repository de chaque entité.
Généralement on utilise le `QueryBuilder` pour créer notre requête.
Ex :

```php
class CarRepository extends \Doctrine\ORM\EntityRepository {
    /**
     * @return array
     */
    public function findCarsWithDetails() {
        $qb    = $this->createQueryBuilder( 'c' );
        $query = $qb->select( 'c, mk, ml' )
                    ->join( 'c.make', 'mk' )
                    ->join( 'c.model', 'ml' )
                    ->getQuery();

        return $query->getResult();
    }
```

```php
    public function findCarWithDetailsById( $id ) {
        $qb = $this->createQueryBuilder( 'c' );

        $query = $qb->select( 'c, mk, ml' )
                    ->join( 'c.make', 'mk' )
                    ->join( 'c.model', 'ml' )
                    ->where( 'c.id = :id' )
                    ->setParameter( 'id', $id )
                    ->getQuery();

        return $query->getSingleResult();
    }
```

Forms :

On utilise généralement le `FormBuilder` au niveau du contrôleur.
On passe à notre vue le résultat de `$form→createView()`.
Ensuite dans la vue on affiche notre formulaire :

- `form_start`
- `form_widget`
- `form_end`

```php
$form = $this->createFormBuilder()
             ->setMethod( 'GET' )
             ->add( 'search', TextType::class )
             ->getForm();
```

CHANGER LE STYLE D'AFFICHAGE DES FORMULAIRES :

1. Ouvrir le fichier `app/config/config.yml`

2. Localiser la section “**twig**”
3. Ajouter une nouvelle clé au dessous :

```yaml
# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
    form_themes:
        - 'bootstrap_3_layout.html.twig'
```

Le reste c’est au niveau de la vue :

```twig
<h1>Our offer</h1>
<div class="pull-right">
    {{ form_start(form, { attr: { class: 'form-inline' }}) }}
    {{ form_widget(form.search, { attr: { placeholder: 'Search a car ...' }}) }}
    <button type="submit" class="btn">Search</button>
    {{ form_end(form) }}
    <br/>
</div>
```

ENVOI ET VALIDATION DU FORMULAIRE :

Pour traiter le formulaire au niveau du contrôleur :

1. On récupère la requête via les paramètres du contrôleur.
2. On fait `$form->handleRequest($request);`
3. Pour tester est-ce que le formulaire a été envoyé ou non : `if($form->isSubmitted())`
4. Pour valider un formulaire contre les contraintes définies pour chaque champs : `if($form->isValid())`

Pour afficher les erreurs au niveau de la vue on utilise : `form_errors`
Ex :

```php
$form = $this->createFormBuilder()
            ->setMethod( 'GET' )
            ->add( 'search',
                TextType::class,
                [
                    'constraints' => [
                        new NotBlank( [ 'message' => 'Vous devez saisir un mot clé' ] ),
                        new Length( [ 'min' => 2, 'minMessage' => 'Le mot clé doit contenir au moins {{ limit }} caractères' ] ),
                    ],
                ] )
            ->getForm();

$form->handleRequest( $request );
if ( $form->isSubmitted() && $form->isValid() ) {
    //TODO
}
```

GÉNÉRATION D’UN FORMULAIRE POUR UNE ENTITÉ :

```
$ php bin/console generate:doctrine:form <entity_shortcut_name>
```

Ex :

```
$ php bin/console generate:doctrine:form CarBundle:Make
```

Opération CRUD (Backoffice) :

COMMANDE :

```
php bin/console generate:doctrine:crud -entity=CarBundle:Car -format=annotation
-with-write -no-interaction
```

Les vues générées sont déposées par défaut dans le dossier app/Resources/views/car. Si on les déplace vers le bundle il ne faut pas oublier de changer modifier les contrôleurs aussi.
Cet utilitaires permet de générer :

- Un contrôleur pour l'entité
- Quatre vue (index, edit, show, new)
- Une classe formulaire

@TEMPLATE

Cette annotation permet de :

- Retourner directement le tableau des données à partir du contrôleur, sans passer par la fonction `$this→render`
- De localiser le fichier de la vue : `BundleName:ControlerClassName:ContolerName.html.twig` (without Controller and Action suffixes)

Error : Model could not be converted to string
Solution : implémenter la fonction `__toString` dans les entités

NB : Ce Scaffolding Crud est bon début pour ce type d'opération.

PERSONNALISATION DES FORMULAIRES GÉNÉRÉS :

Ajouter le type des champs et spécifier les champs obligatoires, contraintes, ...

- TextType, ...
- EntityType : il faut ajouter une options `class`

STYLE DES VUES :

Il faut aussi ajouter les classes CSS nécessaires pour styliser les vues générées.

MENU :

Ajouter une nouvelle entrée dans le menu de l'application

Messages flash :

Ajouter un message dans le Bag à partir du contrôleur :

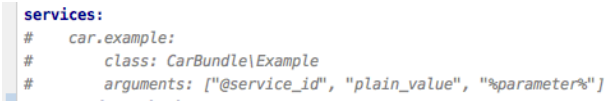
```
$this→addFlash('type', 'ton message')
//type : success, error, warning, info, ...
```

Pour afficher ces messages dans le Layout ou une autre vue :

```
{% for type, flashes in app.session.flashbag.all %}
   {% for flash in flashes %}
     <div class="alert alert-{{ type }} fade in"> {{ flash }} </div>
   {% endfo %}
{% endfor %}
```

Service Container :

Un service est une classe PHP Ordinaire.
On déclare les services propres au bundle dans le fichier `services.yml` de ce même bundle.



Il est recommander de nommer les services avec le format :

```
<bundle_name>.<service_name>
```

Pour utiliser un service à partir d'un contrôleur on utilise :

```
$serv = $this->get('car.example');
```

PASSER DES ARGUMENTS AU SERVICE :

- Un autre service via son id : `"@service_id"`
- Une valeur scalaire : `"string", "int", ...`
- Un paramètre : `"%parameter_name%"`

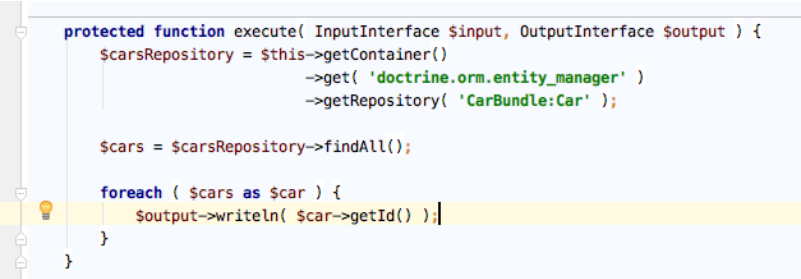
Commandes console :

```
$ php bin/console generate:command
$ Bundle name :  CarBundle
$ Command name :  abc:check-cars
```

Pour l'exécuter :

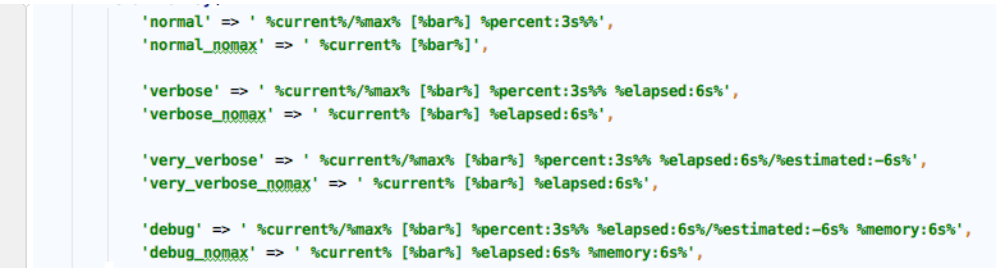
```
$ php bin/console abc:check-cars
```

Pour récupérer les services à partir du Container dans la classe de la nouvelle commande console :



Afficher un état d'avancement de l'exécution de votre commande :

Vous avez la possibilité de spécifier plus d'informations au formatage du `ProgressBar` :
Les formats possible :




```
$bar->setFormat('verbose');
```

DÉFINIR LA COMMANDE EN TANT QUE SERVICE :

Pratique dans le cas ou nous voulons pas utiliser directement le conteneur des services dans notre commandes. Donc il faut injecter les services dont dépend notre commande via son constructeur.

```php
class AbcCheckCarsCommand extends ContainerAwareCommand {
    /** @var  EntityManager */
    protected $entityManager;

    /** @var  DataChecker */
    protected $carChecker;

    /**
     * AbcCheckCarsCommand constructor.
     *
     * @param EntityManager $entityManager
     * @param DataChecker   $carChecker
     */
    public function __construct( EntityManager $entityManager, DataChecker $carChecker ) {
        $this->entityManager = $entityManager;
        $this->carChecker    = $carChecker;

        parent::__construct();
    }
```

Et ne faut oublié de le déclarer dans le fichier `CarBundle/Resources/config/services.yml` :

```yaml
car.command.data_check:
    class: CarBundle\Command\AbcCheckCarsCommand
    arguments:
        - "@car.data_checker"
        - "@doctrine.orm.entity_manager"
    tags:
        - { name: console.command }
```

Tester l'application :

PHPUNIT :

https://phpunit.de/
Installation via composer : Ajouter `"phpunit/phpunit":"~4.8"` au niveau de la section `"require-dev"` dans le fichier composer.json du projet.
Puis :

```
$ composer update

$ vendor/bin/phpunit --version

PHPUnit 4.8.27 by Sebastian Bergmann and contributors.
```

CONFIGURATION :

```
phpunit.xml.dist
```

RUN :

```
$ vendor/bin/phpunit
```

CODE COVERAGE :

Il faut ajouter les informations suivantes à la fin du fichier phpunit.xml.dist :

```
    <logging>
        <log type="coverage-html" target="./build/coverage" title="Autotrader coverage report"
             charset="UTF-8" jui="true" highlight="true"/>
        <log type="coverage-clover" target="./build/logs/clover.xml"/>
        <log type="junit" target="./build/logs/junit.xml"/>
        <log type="testdox-html" target="./build/logs/testdox.html"/>
    </logging>
</phpunit>
```

NB : cette partie dépends de XDebug qu'il soit activé.
Après tu exécute à nouveau le test et tu dois voir les lignes suivantes :

```
Generating code coverage report in Clover XML format ... done

Generating code coverage report in HTML format ... done
```

Tu peux ouvrir le fichier `build/couverage/index.html` dans le navigateur pour voir les résultats.

WRITE UNIT TESTS :

Une calsse de test est une classe PHP créé dans le dossier test du projet et qui `extends` la classe `\PHPUnit_Framework_TestCase`.

Configuration :
La configuration sert aussi à définir des Mock object pour les services qui ne sont pas le sujet du test.
Pour configurer la classe de test il existe plusieurs méthode à redéfinir dans la classe :

- `setUp` : elle s'exécute chaque fois avant l'exécution toutes les fonctions de test dans cette classe.
- `tearDown` : elle s'exécute chaque seule fois après l'exécution de toutes les fonctions de test.

Les fonctions de test doivent être préfixée par le mot "test".
Ex :

```php
class DataCheckerTest extends \PHPUnit_Framework_TestCase {
    /**
     * @var EntityManager|\PHPUnit_Framework_MockObject_MockObject
     */
    protected $entityManager;

    protected function setUp() {
        $this->entityManager = $this->getMockBuilder( EntityManager::class )->disableOriginalConstructor()->getMock();
    }

    public function testCheckCarWithRequiredPhotosWillReturnFalse() {
        $dataChecker    = new \CarBundle\Service\DataChecker( $this->entityManager, true );
        $expectedResult = false;

        $car = $this->getMock( Car::class );
        $car->expects( $this->once() )
            ->method( 'setPromoted' )
            ->with( $expectedResult );

        $this->entityManager->expects( $this->once() )
                            ->method( 'persist' )
                            ->with( $car );
        $this->entityManager->expects( $this->once() )
                            ->method( 'flush' );
        $result = $dataChecker->checkCar( $car );

        $this->assertEquals( $expectedResult, $result );
    }
```

FUNCTIONAL TESTING :

Pour tester nos contrôleur il faut `extends` la classe `WebTestCase`.

- Client Http
- Crawler
- Assertions

Fixtures de doctrine (Seeds) :