LES BASES DE SYMFONY

PLAN

- Introduction à Symfony
- Architecture
- Développer avec Symfony
- Configuration
- Routes et Contrôleurs
- Vue
- Services
- Modèle
- Formulaires

INTRODUCTION À SYMFONY

SYMFONY ?

C'est tout d'abord :

> *Symfony est un ensemble **réutilisables** de composants PHP autonomes, découplés et cohérents qui résolvent des problèmes de développement Web courants.*

Mais aussi, en fonction des composants utilisés :

> *Symfony est également un framework web full-stack.*

Fabien Potencier, http://fabien.potencier.org/article/49/what-is-symfony2.

HISTORIQUE

Première release : 18 octobre 2005

Dernières versions :

- LTS : 2.8.28
- Courante : 3.3.10
- Développement : 3.4.0-BETA3

Support :

Version	Correctifs	Patchs de sécurité
2.8	Novembre 2018	Novembre 2019
3.3	Janvier 2018	Juillet 2018

Prochaine **LTS 3.4** dont la sortie est prévue en **Novembre 2017** Et la version 4 va sortir la semaine prochaine (le 15 ou 16/11).

NOTIONS

- **Components** : Librairies autonomes
- **Bundles** : Packages réutilisables
- **D**ependency **I**njection : principe d'inversion de contrôle
- **Service** : Instance d'un objet PHP qui effectue une tâche

NOTIONS : COMPONENTS

Les composants implémentent des **fonctionnalités communes** nécessaires au développement de sites Web.

Ils sont la base du framework Symfony, mais ils peuvent être utilisés de façon autonome.

Il y a 36 composants :

```
BrowserKit              EventDispatcher     OptionsResolver     Templating
ClassLoader             ExpressionLanguage  Process             Translation
Config                  Filesystem          PropertyAccess      VarDumper
Console                 Finder              PropertyInfo        Yaml
CssSelector             Form                Routing
Debug                   HttpFoundation      Security
DependencyInjection     HttpKernel          Serializer
DomCrawler              Intl                Stopwatch
```

NOTIONS : BUNDLES

Un **bundle** est un dossier contenant un ensemble de fichiers (PHP, CSS, JS, images, ...) qui implémente une **fonctionnalité unique** (un blog, un forum, etc...) et qui doit être **réutilisable**.

Pour utiliser un Bundle dans l'application, on doit le déclarer dans `AppKernel.php` en utilisant la méthode `registerBundles()` :

```
public function registerBundles()
{
    $bundles = array(
        // ...

        new DA2I\DemoBundle\DA2IDemoBundle(),
    );

    // ...
}
```

NOTIONS : DI

L'injection de dépendances implémente le principe d'inversion de contrôle.

Créer dynamiquement les dépendances entre les différentes classes via une description (fichier de configuration).

NOTIONS : SERVICE

Un **service** est un terme générique pour définir un objet PHP qui effectue une tâche spécifique.

Un service est généralement utilisé de façon **global** (connexion à une base de données, envoi d'email)

Dans Symfony, les services sont souvent configurés et récupérés à partir du **conteneur de service**.

INSTALLATION

Via l'installeur Symfony :

```
sudo mkdir -p /usr/local/bin
sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
sudo chmod a+x /usr/local/bin/symfony
```

On peut créé notre application :

```
symfony new da2i_projet
```

On lance le serveur web :

```
cd da2i_projet
php bin/console server:start
```

Et on accède au site: http://localhost:8000

ARCHITECTURE

UN PROJET SYMFONY (1/2)

Structure d'un projet Symfony 3 :

```
chemin/du/projet/
    app/
        config/
        Resources/
            views/
    bin/
        console
    src/
        AppBundle/
            ...
        ...
    tests/
        ...
    var/
        cache/
        logs/
        sessions/
    vendor/
        ...
    web/
        app.php
```

UN PROJET SYMFONY (2/2)

Chaque dossier à son propre objectif (et son ensemble de fichiers):

- `app/` contient le noyau de l'application, la configuration et les vues;
- `src/` contient vos **bundles**;
- `tests/` contient vos tests;
- `var/` contient les fichiers qui changent souvent (comme sur un système Unix);
- `vendor/` contient les dépendences du projet;
- `web/` contient vos front contrôleurs et les resources (css/js).

STRUCTURE D'UN BUNDLE

La structure recommandée est la suivante :

```
XXX/...
    DemoBundle/
        XXXDemoBundle.php
        Controller/
        Resources/
            config/
            doc/
                index.rst
            translations/
            views/
            public/
        Tests/
        LICENSE
```

Le(s) dossier(s) XXX correspondent au namespace du projet. Pour reverser un `Bundle` à la communauté, il est nécessaire d'ajouter les fichiers `LICENSE` et `Resources/doc/index.rst`.

Dans le cadre d'un projet, Symfony conseille d'utiliser le Bundle `AppBundle`.

BUNDLE ! OÙ JE METS MON CODE ?

Type	Dossier
Commandes	`Command/`
Contrôleurs	`Controllers/`
Extension du conteneur de service	`DependencyInjection/`
Écouteurs d'événements	`EventListener/`
Configurations	`Resources/config/`
Resources Web	`Resources/public/`
Fichiers de traductions	`Resources/translations/`
Templates	`Resources/views/`
Tests unitaires	`Tests/`

REQUEST

La classe `Symfony\Component\HttpFoundation\Request` permet d'obtenir un objet PHP correspond à la requête :

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

Natif PHP	Objet Request
$_GET['foo']	$request->query->get('foo')
$_POST['foo']	$request->request->get('foo')
$_SERVER['PATH_INFO']	$request->getPathInfo();
$_SERVER['HTTP_HOST']	$request->server->get('HTTP_HOST')
gethostname()	$request->header->get('host')

RESPONSE

La classe `Symfony\Component\HttpFoundation\Response` permet de représenter la réponse :

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response();
```

Natif PHP	Objet Response
echo('ma réponse')	$response->setContent('ma réponse');
header("HTTP/1.0 404 Not Found");	$response->setStatusCode(Response::HTTP_NOT_
header('Content-Type: application/json');	$response->headers->set('Content-Type', 'applicatio

La réponse en Symfony n'est envoyée que lorsqu'on le demande explicitement : `$response->send();`

LET'S PLAY : MON FRONT CONTROLLER

Pré-requis :

- Créer un fichier PHP `front.php` dans le dossier web
- Utiliser les classes `Request` et `Response` de Symfony
- Ajouter l'autoload de composer :

```
require __DIR__.'/../vendor/autoload.php';
```

Ce que je veux :

- Afficher "Page d'accueil" quand on accède à http://127.0.0.1:8000/front.php
- Afficher "Salut les DA2I" quand on accède à http://127.0.0.1:8000/front.php/hello
- Et "Page not found" avec le code de retour 404 dans les autres cas.

DÉVELOPPER AVEC SYMFONY

LES ENVIRONNEMENTS (DEV / PROD)

Deux fonts controllers `app.php` et `app_dev.php`.

Le front controller `app_dev` offre plusieurs avantages :

- Affichage des erreurs
- Debug toolbar

LET'S PLAY : MON 1ER BUNDLE

Ce que je veux :

- Création d'un Bundle `Da2iFoodTruckBundle` avec le namespace `Da2i\Bundle\FoodTruckBundle`
- Déclarer ce nouveau Bundle dans l'application

/!\ Changer l'autoload PSR4 dans le composer.json et relancer un `composer dumpautolaod`

CONFIGURATION

CONFIGURATION DE L'APPLICATION

Les fichiers de configuration, dans Symfony, peuvent être écrit en YAML, XML ou PHP

Par défault, les fichiers de configuration se trouvent dans le dossier `app/config`. Et les fichiers principaux sont :

- `parameters.yml` : pour la configuration liée à l'infrastructure.
- `config.yml` : configuration de l'application.

Chaque *Bundle* aura ses propres fichiers de configuration dans son dossier `Resource/config`.

CONFIGURATION YAML

Exemple :

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
    secret: '%secret%'
    router: { resource: '%kernel.root_dir%/config/routing.yml' }
    # ...

# Twig Configuration
twig:
    debug:            '%kernel.debug%'
    strict_variables: '%kernel.debug%'

# ...
```

ROUTES ET CONTRÔLEURS

CONTROLLER

Un contrôleur est une fonction PHP qui reçoit un requête HTTP et qui renvoie une réponse HTTP.

Le cycle de vie d'une requête :

- Le **front controller** qui amorce l'application.
- Le **routeur** qui cherche une **route** correspondante et le contrôleur associé.
- Exécution du **contrôleur** qui renvoie une response.

LET'S PLAY : MON CONTROLLER

Créer un controller `Default` qui a une action `index` pour afficher "Hello DA2I" en homepage.

=> problème, comment le routeur trouve notre contôleur ?

ROUTE

Pour définir une route, on utilise le fichier `app/config/routing.yml`.

Exemple:

```
# app/config/routing.yml
homepage:
    path:  /
    defaults: { _controller: AppBundle:Hello:index }
```

Chaque route a un *controller* associé qui a pour format : `bundle:controller:action`

Par exemple, un _controller qui a comme valeur AcmeBlogBundle:Blog:show veut dire :

- Dans le Bundle : AcmeBlogBundle
- Dans la classe controller : BlogController
- Utiliser la méthode : showAction

LET'S PLAY : JE TRACE MA ROUTE

Ajouter la configuration de la route vers le controller.

```
homepage:
    path:  /
    defaults: { _controller: AppFoodTruckBundle:Default:index }
```

LET'S PLAY : UN PEU DE CONFIGURATION

Ce que je veux :
Que le texte "Da2i" de mon Controller `AppFoodTruckBundle:Default:index` soit configurable dans `config.xml`.

```
# app/config/config.yml
...
parameters:
    ...
    site_name: Da2i project
```

```
$this->getParameter('site_name')
```

LET'S PLAY : CONTROLLER++

Ce que je veux : Une page qui affiche "Bienvenue, Maxime" quand j'accède à http://localhost:8000/hello/maxime

```
hello:
    path: /hello/{name}
    defaults: { _controller: AppFoodTruckBundle:Default:hello }

hello_bis:
    path: /hello_bis
    defaults: { _controller: AppFoodTruckBundle:Default:helloBis }
```

VUE

LES TEMPLATES

Un template est un fichier texte qui génére n'importe quel format texte (HTML, XML, CSV, ...).

Dans Symfony, le moteur de template est Twig :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

POURQUOI TWIG ?

- Fichier de template concis et lisible.
- Plus facile à prendre en main pour les webdesigners.
- Permet de séparer la présentation et la logique de l'application
- Plus puissant que le templating en PHP (contrôle des espaces, sandboxing, échappement HTML, ...)
- Gestion de cache

LA SYNTAXE TWIG

Trois type de syntaxe spéciale :

`{{ ... }}`

"Dit moi quelque chose" : affiche le contenu d'une variable ou un résultat d'une expression du template

`{% ... %}`

"Fait quelque chose" : syntaxe qui contrôle la logique du template. Ex: condition, boucle, ...

`{# ... #}`

"Commente quelque chose": Ajout un commentaire qui n'est pas afficher dans le rendu final.

Mais aussi des filtres et des fonctions. Auxquels, on peut venir ajouter nos propres filtres.

NOMMAGE ET EMPLACEMENTS

Deux emplacements possible :

- `app/Resources/views/` : contient les templates de base de l'application ou les surcharges de template d'un Bundle tiers.
- `src/AppBundles/Resources/views/` : Quand on veut partager son Bundle avec la communauté.

Pour le nommage, deux possibilités :

- Si le template ce trouve dans `app/Resources/views/`, c'est le chemin relatif à partir de ce dossier. Ex: pour `app/Resources/views/montemplate.html.twig`, le nom sera `montemplate.html.twig`
- Si il est dans le dossier du Bundle, on va utiliser la syntaxe suivante : `@BundleName/directory/filename.html.twig`

LET'S PLAY : TEMPLATISONS !

Ce que je veux :
Remplacer les retours de nos 2 actions par des templates `default/index.html.twig` et `default/hello.html.twig`

```
    return $this->render('default/index.html.twig');

    return $this->render('default/hello.html.twig', array(
        'name' => $name,
    ));
```

```
{# default/index.html.twig #}
<html>
    <body>
        <h1>Hello Da2i !</h1>
    </body>
</html>
{# default/hello.html.twig #}
<html>
    <body>
        <h1>Bienvenue {{ name }}</h1>
    </body>
</html>
```

HÉRITAGE ET MISE EN PAGE (1/2)

Un certain nombre d'élements d'une page sont communs (header, footer, menu, ...). On a donc la notion d'héritage pour répondre à ce besoin.

On construit un fichier de mise en page :

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
                <ul>
                    <li><a href="/">Home</a></li>
                    <li><a href="/blog">Blog</a></li>
                </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
```

On définit des zones via l'instruction `{% block ... %}`. Ci-dessous on a trois zones : `title`, `sidebar` et `content`.

HÉRITAGE ET MISE EN PAGE (2/2)

Chacunes des zones peut être surchargées par un template enfant.

Par exemple :

```
{# app/Resources/views/blog/index.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

LET'S PLAY : HÉRITONS !

Ce que je veux :

- un titre de page
- un menu
- un contenu
- un footer avec un lien vers le site da2i

LES SERVICES

QU'EST CE QU'UN SERVICE ?

Un **service** est un terme générique pour définir un objet PHP qui effectue une tâche spécifique.

Un service est généralement utilisé de façon **global** (connexion à une base de données, envoi d'email)

Dans Symfony, les services sont souvent configurés et récupérés à partir du **conteneur de service**.

QU'EST CE QU'UN CONTENEUR DE SERVICES ?

Un **conteneur de service**, également appelé conteneur d'**injection de dépendance** (DIC, Dependency Injection Container), est un objet spécial qui **gère l'instanciation des services** dans une application.

Le conteneur de service prend soin d'instancier et d'injecter des services dépendants.

COMMENT CRÉER UN SERVICE ? (1/2)

On crée une classe PHP :

```php
namespace AppBundle\Service;
class MonService
{
    public function getMessage()
    {
        return 'Mon message';
    }
}
```

Et on l'utilise :

```php
use AppBundle\Service\MonService;

public function newAction(MonService $monService) {
    $this->addFlash('success', $monService->getMessage());
}
```

Magique ?

COMMENT CRÉER UN SERVICE ? (2/2)

Non !

Les services sont automatiquement chargés via le fichier `services.yml` :

```
## app/config/services.yml
services:
    ## default configuration for services in *this* file
    _defaults:
        autowire: true
        autoconfigure: true
        public: false

    ## makes classes in src/AppBundle available to be used as services
    AppBundle\:
        resource: '../../src/AppBundle/*'
        ## you can exclude directories or files
        ## but if a service is unused, it's removed anyway
        exclude: '../../src/AppBundle/{Entity,Repository}'
```

INJECTER DES SERVICES DANS UN SERVICE (1/2)

Si notre Service `MonService` à besoin du service `logger`

Rien de plus simple !

```php
// ...

use Psr\Log\LoggerInterface;

class MonService
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function getMessage()
    {
        $this->logger->info('Appel de ma méthode getMessage');
        // ...
    }
}
```

Et voilà !

INJECTER DES SERVICES DANS UN SERVICE (2/2)

Comment je sais que je peux utiliser le type `LoggerInterface` ?

```
php bin/console debug:container --types
```

```
Symfony Container Public and Private Services
=============================================

 ------------------------------------------------------ --------------------------------
  Service ID
 ------------------------------------------------------ --------------------------------
  AppBundle\AppBundle                                    AppBundle\AppBundle
  AppBundle\Controller\DefaultController                 AppBundle\Controller\DefaultCo
  AppBundle\Controller\DemoController                    AppBundle\Controller\DemoContr
  Doctrine\Common\Annotations\Reader                     alias for "annotations.cached_
  Doctrine\Common\Persistence\ManagerRegistry            alias for "doctrine"
  Doctrine\Common\Persistence\ObjectManager              alias for "doctrine.orm.defaul
  Doctrine\DBAL\Connection                               alias for "doctrine.dbal.defau
  Doctrine\DBAL\Driver\Connection                        alias for "doctrine.dbal.defau
  Doctrine\ORM\EntityManagerInterface                    alias for "doctrine.orm.defaul
  Psr\Cache\CacheItemPoolInterface                       alias for "cache.app"
  Psr\Container\ContainerInterface                       alias for "service_container"
  Psr\Log\LoggerInterface                                alias for "monolog.logger"
  SessionHandlerInterface                                alias for "session.handler.nat
```

(et lire la doc, RTFM !)

Et si je veux que mon message soit configurable ?

```
// ...
class MonService
{
    // ...
    private $message;

    public function __construct(LoggerInterface $logger, $message)
    {
        // ...
        $this->message = $message;
    }

    public function getMessage()
    {
        // ...
        return $this->message;
    }
}
```

Cannot autowire service "MonService": argument "$message" of method "__construct()" must have a type-hint or be given a value explicitly.

DÉFINIR MANUELLEMENT UN ARGUMENT (2/2)

Normal, on veut le définir explicitement :

```
# app/config/services.yml
services:
    # ...

    # Configurer explicement le service
    AppBundle\Service\MonService:
        arguments:
            $message: 'Mon message de façon explicte'
```

Maintenant, le conteneur de services connait le 2ième argument de mon service.

LET'S PLAY : À VOTRE SERVICE

Ce que je veux :

- Créer un service `ApplicationConfig` dans le Bundle `FoodTruckBundle` (pensez au fichier `service.yml`)
- Ce service doit avoir une méthode `getSiteName` qui :
  - Log l'appel à la méthode
  - Retourne le nom du site
- Utiliser ce service dans l'action `helloAction` de notre `DefaultController`

MODÈLES

CONFIGURATION DES ACCÈS

Pré-requis :

- un serveur de base de donnée (MySQL, ...)

Configuration des accès à la base de données dans le fichier `app/config/parameters.yml` :

```
# app/config/parameters.yml
parameters:
    database_host:     localhost
    database_name:     da2i_project
    database_user:     da2i_user
    database_password: da2i_password

# ...
```

Et on créé la base de données :

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

CRÉATION D'UNE ENTITÉ

Notre application doit manipuler des utilisateurs :

```php
// src/AppBundle/Entity/Customer.php
namespace AppBundle\Entity;

class Customer
{
    private $fistname;
    private $lastname;
    private $age;
}
```

LE MAPPING (1/2)

Doctrine offre la possibilité de manipuler des objets, on doit donc lui définir comment sont mappé les champs de la table avec les propriétés de notre classe :

LE MAPPING (2/2)

On donne des méta-données qui indique à Doctrine comment la classe Customer et ses propriétés doivent être mappées à une table de la base de données :

```
# src/AppBundle/Resources/config/doctrine/Customer.orm.yml
AppBundle\Entity\Customer:
    type: entity
    table: customer
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        firstname:
            type: string
            length: 100
        lastname:
            type: string
            length: 100
        age:
            type: integer
```

La documentation de doctrine fournit les informations sur les types de champs.

Et on valide les mapping :

```
php bin/console doctrine:schema:validate
```

GETTERS/SETTERS

On ajoute les getters et setters à notre Entité `Customer` :

```
// src/AppBundle/Entity/Customer.php
namespace AppBundle\Entity;

class Customer
{
    // ...
    /**
     * @return string
     */
    public function getFirstname() {
        return $this->firstname;
    }

    public function setFirstname($firstname) {
        $this->firstname = (string) $firstname;
        return $this;
    }
    // ...
```

CRÉATION DE LA TABLE

On visualise les requêtes SQL :

```
bin/console doctrine:schema:update --dump-sql
```

On exécute les requêtes :

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

CRÉER UN OBJET

Utiliser le manager d'entité (`EntityManager`) :

```
// src/AppBundle/Controller/DefaultController.php
// ...
use AppBundle\Entity\Customer;
use Symfony\Component\HttpFoundation\Response;
use Doctrine\ORM\EntityManagerInterface;
// ...
public function createAction(EntityManagerInterface $entityManager)
{
    $customer = new Customer();
    $customer->setFirstname('Maxime');
    $customer->setLastname('Leclercq');
    $customer->setAge(30);

    // préviens Doctrine que l'on souhaite enregistrer un client
    $entityManager->persist($customer);
    // exécute réellement les requêtes (c'est-à-dire la requête INSERT)
    $entityManager->flush();

    return new Response('Nouveau client enregistré avec l`id '.$customer->getId());
}
```

On oublie pas de définir notre route dans le fichier `app/config/routing.yml`.

RÉCUPÉRER UN OBJET - LE REPOSITORY (1/3)

On souhaite récupérer un client via son ID :

```
// src/AppBundle/Controller/DefaultController.php
// ...
public function showAction($customerId)
{
    $customer = $this->getDoctrine()
        ->getRepository(Customer::class)
        ->find($customerId);

    if (!$customer) {
        throw $this->createNotFoundException(
            'Aucun client trouvé pour l'id '.$customerId
        );
    }

    // ...
}
```

Pour récupérer un object, on passe toujours par le `Repository`.

RÉCUPÉRER UN OBJET - LE REPOSITORY (2/3)

```
$repository = $this->getDoctrine()->getRepository(Customer::class);

// requête pour récupérer un client unique via sa clé primaire
$customer = $repository->find($customerId);

// Méthode dynamique pour trouver un seul client basé sur une valeur de colonne
$customer = $repository->findOneById($customerId);

// Méthodes dynamiques pour tourver un ensemble de client basé sur une valeur de colonn
$clients = $repository->findByLastname('Leclercq');
$clients = $repository->findByLastname(30);

// Récupère tous les clients
$clients = $repository->findAll();
```

On peut exécuter aussi des requêtes plus complexes, cf Demander des objets

RÉCUPÉRER UN OBJET - LE REPOSITORY (3/3)

Deux autres méthodes trés utiles `findBy()` et `findOneBy()` :

```
$repository = $this->getDoctrine()->getRepository(Customer::class);

// requête pour récupérer un client à partir de son prénom et nom
$client = $repository->findOneBy(
    array('firstname' => 'Maxime', 'lastname' => 'Leclercq')
);

// requête pour récupérer plusieurs clients qui ont 30 ans et triés par nom.
$clients = $repository->findBy(
    array('age' => 30),
    array('lastname' => 'ASC')
);
```

METTRE À JOUR UN OBJET

```
// src/AppBundle/Controller/DefaultController.php
public function updateAction(EntityManagerInterface $entityManager, $customerId)
{
    $customer = $this->getDoctrine()
        ->getRepository(Customer::class)
        ->find($customerId);
    $customer->setLastname('Modification');

    if (!$customer) {
        throw $this->createNotFoundException(
            'Aucun client trouvé pour l'id '.$customerId
        );
    }

    // exécute réellement les requêtes (c'est-à-dire la requête UPDATE)
    $entityManager->flush();

    return new Response('Le client avec l`id '.$customer->getId().' a été modifié');
}
```

SUPPRIMER UN OBJECT

```
// src/AppBundle/Controller/DefaultController.php
public function deleteAction(EntityManagerInterface $entityManager, $customerId)
{
    $customer = $this->getDoctrine()
        ->getRepository(Customer::class)
        ->find($customerId);
    $customer->setLastname('Modification');

    if (!$customer) {
        throw $this->createNotFoundException(
            'Aucun client trouvé pour l'id '.$customerId
        );
    }

    // On prévient Doctrine que l'on veut supprimer un objet
    $entityManager->remove($customer);
    // exécute réellement les requêtes (c'est-à-dire la requête DELETE)
    $entityManager->flush();

    return new Response('Le client a été supprimé');
}
```

LET'S PLAY: CRÉER MON ENTITY

Ce que je veux :

- Une entité `Ingredient` avec 2 colonnes :
  - id
  - name
- Les opérations de CRUD suivante :
  - une page qui liste l'ensemble de mes ingrédients
  - un bouton pour créer un Ingédient "tomate"
  - un bouton qui me modifie mon Ingrédient "tomate" en "cheddar"
  - un bouton qui me supprimer mon Ingrédient

CRÉATION D'UNE CLASSE "REPOSITORY"

Quand on souhaite créer un "Repository" personnalisé :

```
# src/AppBundle/Resources/config/doctrine/Customer.orm.yml
AppBundle\Entity\Customer:
    type: entity
    repositoryClass: AppBundle\Repository\CustomerRepository
    # ...

Puis on créé la classe :

```php
// src/AppBundle/Repository/CustomerRepository.php
namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;

class CustomerRepository extends EntityRepository
{
    public function findAllOrderedByLastname()
    {
        return $this->findBy([], ['lastname' => 'ASC']);
    }
}
```
```

LET'S PLAY: CRÉER MON REPOSITORY

Ce que je veux :

- Un repository `IngredientRepository` avec la méthode `findAllOrderedByName()`
- Utiliser la nouvelle classe pour la page "liste".

FORMULAIRES

CONSTUIRE UN FORMULAIRE (1/2)

Dans le controller, via la méthode `createFormBuilder` :

```php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use AppBundle\Entity\Customer;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
// ...

public function newAction(Request $request)
{
    // On créé un object Customer et on insert quelques valeurs
    $customer = new Customer();
    $customer->setFirstname('Maxime');
    $customer->setLastname('Leclercq');
    $customer->setAge(30);

    $form = $this->createFormBuilder($customer)
        ->add('firstname', TextType::class)
        ->add('lastname', TextType::class)
        ->add('age', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Créer un client'))
```

CONSTUIRE UN FORMULAIRE (2/2)

Dans notre template :

```
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form) }}
{{ form_widget(form) }}
{{ form_end(form) }}
```

Juste 3 méthodes pour afficher un formulaire :

- form_start(form) : Affiche la balise de début `<form>` (y compris `enctype`)
- form_widget(form) : Afficher tous les champs, à savoir : le champs, le label et les messages d'erreur de validation
- form_end(form) : Affiche la balise de fin `</form>`.

SOUMETTRE UN FORMULAIRE (1/2)

Par défaut, le formulaire effectue une requête de type POST sur le même contrôleur.

Dans notre cas, si on souhaite gérer la soumission du formulaire :

```
// ...
public function newAction(Request $request)
{
    $customer = new Customer();
    $form = $this->createFormBuilder($task)
        ->add('firstname', TextType::class)
        ->add('lastname', TextType::class)
        ->add('age', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Créer un client'))
        ->getForm();

    // on gère la requête
    $form->handleRequest($request);

    // Si le formulaire est soumis et les données sont valides
    if ($form->isSubmitted() && $form->isValid()) {
        // ... enregistrer les données

        return $this->redirectToRoute('task_success');
    }
```

SOUMETTRE UN FORMULAIRE (2/2)

Le contrôleur à trois comportements possible :

1. Au 1er chargement, le formulaire est créé et rendu : La méthode `handleRequest()` ne fait rien et la méthode `isSubmitted()` retourne `false`.
2. L'utilisateur soumet le formulaire : La méthode `handleRequest()` récupère les valeurs et écrit les données dans l'objet :
    1. L'objet n'est pas valide : La méthode `isValid()` renvoie `false` et le formulaire est affiché avec les erreurs.
    2. Si les données sont valides : La méthode `isValid()` renvoie `true` et on peut effectué le traitement.

LA VALIDATION

On ajoute des règles sur nos entités via le fichier `validation.yml` dans notre Bundle :

```
# src/AppBundle/Resources/config/validation.yml
AppBundle\Entity\Customer:
    properties:
        lastname:
            - NotBlank: ~
        firstname:
            - NotBlank: ~
        age:
            - GreaterThanOrEqual: 18
```

Et voilà !

CRÉATION DE CLASSES DE FORMULAIRE (1/2)

Pour les formulaires, la bonne pratique consiste à créer une classe dédiée qui est séparée et autonome du contrôleur dans le dossier `Form/` du Bundle et suffixée de `Type` :

```
// src/AppBundle/Form/CustomerType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class CustomerType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('lastname')
            ->add('firstname')
            ->add('age')
            ->add('save', SubmitType::class)
        ;
    }
}
```

CRÉATION DE CLASSES DE FORMULAIRE (2/2)

Et on utilise notre nouvelle classe dans le contrôleur :

```
// src/AppBundle/Controller/DefaultController.php
use AppBundle\Form\TaskType;

public function newAction()
{
    $customer = ...;
    $form = $this->createForm(CustomerType::class, $customer);

    // ...
}
```

LET'S PLAY: FORMULARISONS

Ce que je veux :

- Créer un formulaire `IngredientType`
- Utiliser ce formulaire pour la création et la modification d'un ingrédient `IngredientController`

FIN !