FRAMEWORK SYMPHONY

Qu'est-ce qu'un framework ?

- Un cadre de travail
- Un ensemble de librairies de fonctionnalités
  - validation, authentification,...
- Permet l'amélioration de la productivité des développeurs

- Avec toujours le même but : développer des solutions facile à maintenir et faire évoluer

Les plus

- Une structure imposée :
  - architecture MVC, répertoires
- Profiter d'une expérience
- Une certaine standardisation
  - permet le travail collaboratif
  - engagement plus ciblé
- Une communauté
- Un support, une documentation

Les moins

- Plus complexe à appréhender
  - temps d'apprentissage plus loin
- Temps d'exécution augmenté

Les frameworks

- Symphony a été développé par SensioLab (société française)
  - open source
- Zend Framework développé par Zend (société qui maintient PHP)
- CodeIgniter, CakePHP,...

COMPOSER

Gestion des dépendances

- Le projet dépend de librairies
  - Ces librairies dépendent d'autres librairies
- Composer
  - permet de déclarer les dépendances
  - recherche les versions nécessaires pour les packages et les installe

- Installation
  - https://getcomposer.org/

SYMPHONY

Evolution de Symfony

- Version 3
  - Remplacer dans les commandes
    - app/console par bin/console
- Version 4
  - Changement de dossier
  - Un seul front end
  - Peu de composants par défaut
    - Pour installer un composant
      - Exemple pour le composant Twig
        - composer req twig
    - Composant Flex
      - retrouve le package à installer
      - configure

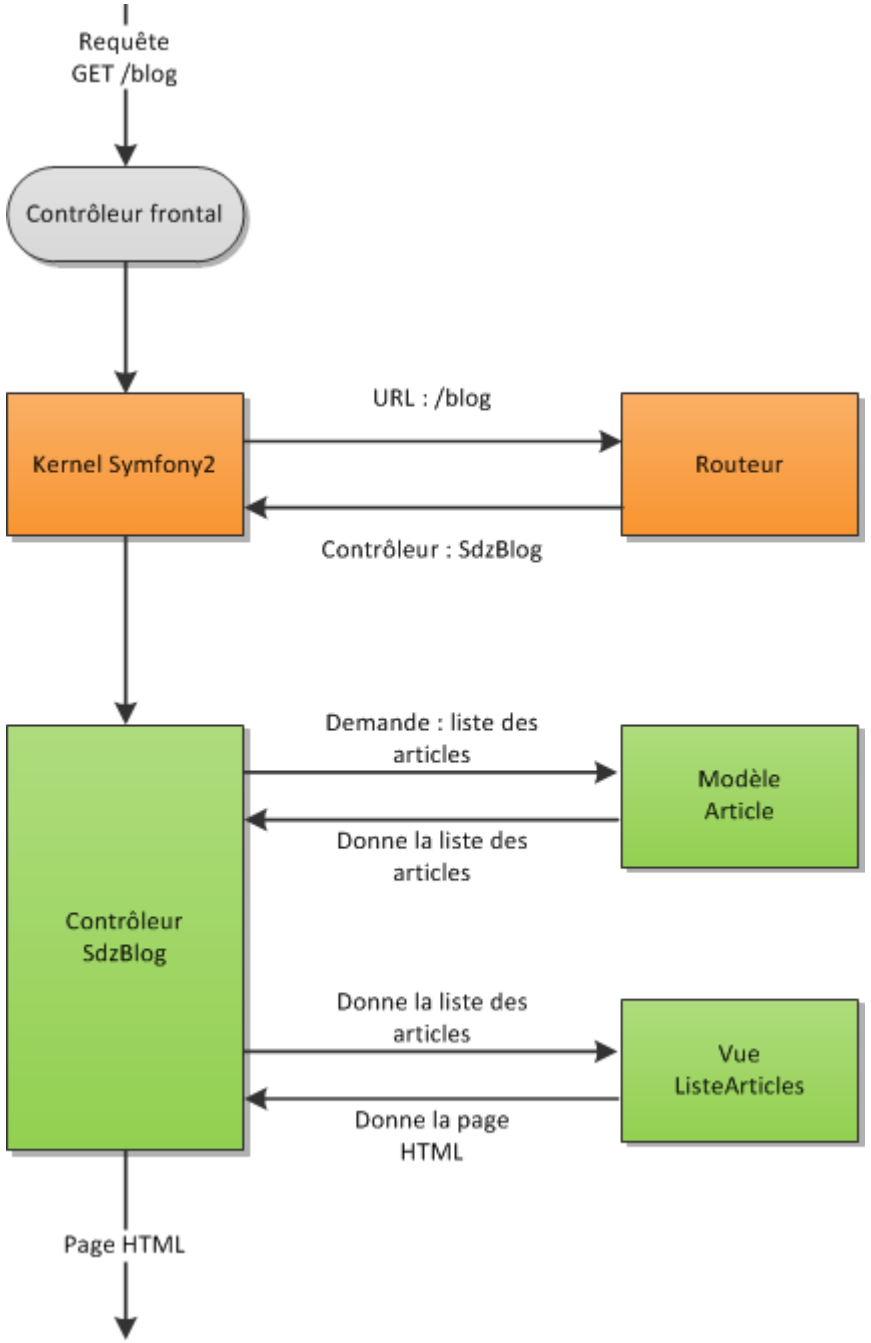
Glossaire de base

- asset
  - concerne tout ce qui n'est pas exécutable (image, css,...)
- bundle
  - dossier contenant les fichiers qui concerne une fonctionnalité
- container (service container)
  - objet de service
- entity
  - objet persisté en base de données
- repository
  - centralise tout ce qui touche à la récupération d'entité

Structure des dossiers

- /bin
  - mode console (ligne de commande)
- /config
  - configuration
- /public
  - image, css, js et le controleur frontal app.php
- /src
  - sources organisées en bundles (briques)
- /var
  - logs et cache
- /vendor
  - bibliothèques externes Twig SwiftMailer,...

MVC



- Seul le dossier public est accessible
- 1 contrôleur frontal
  - index.php

- Dans les versions précédentes 2 contrôleurs frontaux :
  - app.php
    - pour la production
  - app_dev.php
    - pour le développement
    - contient une barre d'outils

Les bundles

- C'est une brique logicielle permettant l'échange
- Regroupe à un endroit tout ce qui concerne une fonctionnalité (blog, utilisateurs, boutique,...)
- Les règles communes de développement permettent de les faire communiquer

Structure d'un bundle

- /Controller
- /DependencyInjection
  - contient des infos concernant les dépendances
- /Entity
  - contient le modèle
- /Form
  - contient les formulaires
- /Resources
  - /config contient les fichiers de configuration (route par exemple)
  - /public contient les fichiers publics
  - /views contient les vues
- /Tests
  - contient les tests unitaires et fonctionnels

Installation et mode console

- Vérifier que php fonctionne en ligne de commande :
  - php –v
  - Attention minimum V7 pour S4
- Installer voir https://symfony.com/doc/current/setup.html
  - composer create-project symfony/website-skeleton my-project
- Tester l'installation localhost/demo/public

- Utilisation de la console
  - Beaucoup de commandes sont accessibles uniquement via la console
    - généré un bundle : php bin/console generate:bundle
    - vider la cache : php bin/console clear:cache

Le contrôleur

- Le rôle du contrôleur est de retourner une réponse
- Exemple

```php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class DemoController extends Controller
{
     /**
      * @Route("/demo/hello")
      */
    public function helloAction()
    {
        $name = 'Peter';

        return new Response(
            '<html><body>Hello '.$name.'</body></html>'
        );
    }
}
```

- Le contrôleur hérite du contrôleur de base
  - on le charge donc préalablement avec le use
- Convention de nommage :
  - La classe a le nom du contrôleur suivi de Controller
  - La méthode a le nom de l'action suivi de Action
- La réponse est renvoyée directement par l'objet Response
  - on charge donc la librairie correspondante

Le contrôleur

- Créer le contrôleur en ligne de commande
  - php bin/console make:controller

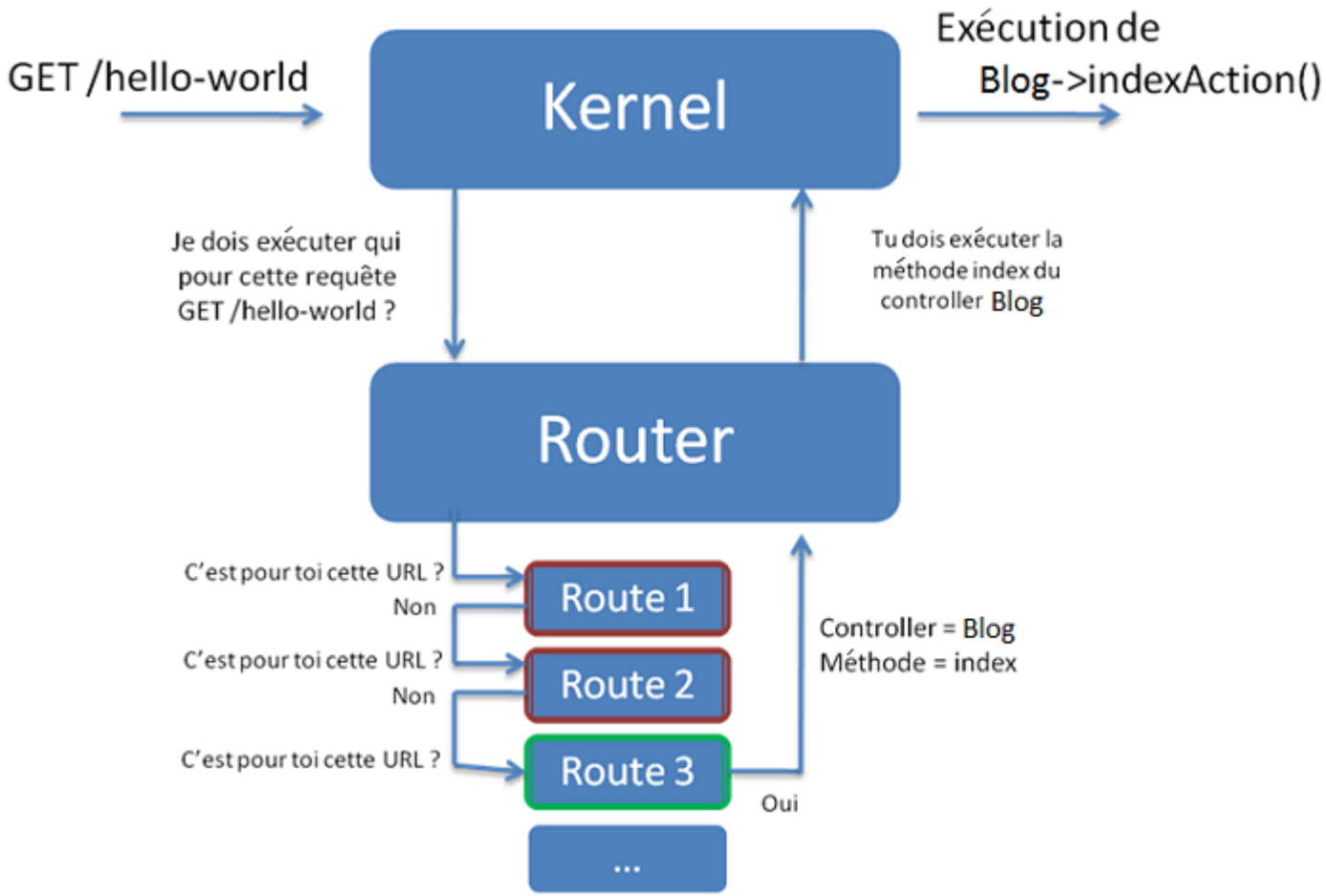
Récupération des GET et POST

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

public function indexAction(Request $request)
{
    // retrieve GET and POST variables respectively
    $request->query->get('page');
    $request->request->get('page');
}
```

- Inclusion du composant Request
- Déclaration du paramètre $request en tant qu'objet Request
- L'objet Request permet de récupérer
  - le GET
    - $genre = $request->query->get('genre');
  - le POST
    - $genre = $request->request->get('genre');

Les routes


GET /hello-world
Kernel
Exécution de
Blog->indexAction()
Je dois exécuter qui
pour cette requête
GET /hello-world ?
Tu dois exécuter la
méthode index du
controller Blog
Router
C'est pour toi cette URL ?
Non
Route 1
C'est pour toi cette URL ?
Non
Route 2
C'est pour toi cette URL ?
Route 3
Oui
Controller = Blog
Méthode = index
...

Les routes

- Il est possible de définir des routes avec un fichier yml ou par annotation dans les contrôleurs
- Installer le composant `composer require annotations`
- Par annotation, la route est définie par le commentaire qui précède la méthode Action
  - Route pour la racine

```
/**
 * @Route("/",name="home")
 */
```

  - Route pour une liste d'utilisateurs

```
/**
 * @Route("/user/list", name="user_list")
 */
```

  - Route vers un utilisateur

```
/**
 * @Route("/user/user/show/{id}", name="user_show")
 */
  public function showAction($id)
{
            ...
}
```

  - Vérifier les routes

`php bin/console debug:router`

Les vues

- La vue permet de mettre en forme les données que le contrôleur lui à envoyer afin de former une page html, un flux json, xml, rss, un email,...
  - L'objet Response est remplacée par $this->render
- Exemple de réponse à partir du contrôleur :

```php
public function helloAction()
{
    $name = 'Peter';

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

  - La réponse est envoyée à la vue avec le paramètre associé
- Exemple de template Twig

```twig
<h1>Hello {{ name }}</h1>
```

- Attention ! il faut installer le composant twig

Les vues avec Twig

- Dans le template Twig
  - {{ … }} affiche quelque chose ;
  - {% … %} fait quelque chose ;
  - {# … #} n'affiche rien et ne fait rien : c'est la syntaxe pour les commentaires, qui peuvent être sur plusieurs lignes.
- Exemple d'utilisation de paramètres envoyés par le contrôleur
  - Hello {{ name }}!
- Ajout d'une URL dans la réponse
  - retour vers world

Les entités

- Pour persister les données en base de données on peut utiliser un ORM (Object-Relation Mapper)
- L'ORM intégré à Symfony est Doctrine
- Un objet dont l'enregistrement est confié à un ORM est une entité
- Plus besoin de programmer des requêtes l'ORM s'en chargera pour nous
- L'utilisation de l'ORM garantit l'utilisation d'autres gestionnaires de base de données

Créer la base de données

- Pour rappel les paramètres sont dans
  - /.env

```
# customize this line!
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"
```

- Créer la base de données dans Mysql

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

Une entité

```php
namespace App\Entity;

class etudiant
{
    private $id;
    private $nom;
    private $prenom;
    private $dn;
    private $matricule;

    public function getId()
    {
        return $this->id;
    }

    public function setNom($nom)
    {
        $this->nom = $nom;
        return $this;
    }

    public function getNom()
    {
        return $this->nom;
    }
    public function setDn($dn)
    {
        $this->dn = $dn;
        return $this;
    }

    public function getDn()
    {
        return $this->dn;
    }

    public function setMatricule($matricule)
    {
        $this->matricule = $matricule;

        return $this;
    }
}
```

Les annotations

- Doctrine utilise les commentaires définis dans la classe entité pour gérer la DB :

```php
use Doctrine\ORM\Mapping as ORM;
/**
 * etudiant
 *
 * @ORM\Table(name="hello_etudiant")
 * @ORM\Entity(repositoryClass="demo\Bundle\helloBundle\Entity\etudiantRepository")
 */
class etudiant
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

Les annotations

- La génération de l'entité (classe) peut se faire à l'aide de la console :

```
php bin/console make:entity Product
```

- On peut aussi ajouter des attributs calculés à l'entité en ajoutant des méthodes

Les relations

- Différents types de relations peuvent être gérées directement par Doctrine :
  - One-To-One

```php
/**
 * @ORM\OneToOne(targetEntity="BlogBundle\Entity\Image", cascade={"persist"})
 */
private $image;
```

  - One-To-Many
  - Many-To-One

- Pour tout savoir :
- http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html
- Attention ajouter @ORM à chaque @

Générer la DB

- Dans la console
  - pour visualiser les requêtes avant exécution

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

  - pour exécuter les requêtes

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

Récupérer des entités

- find($id)
  - récupère l'entité sur base de son id
  - Exemple :
    - $product = $this->getDoctrine()
    - ->getRepository('AcmeStoreBundle:Product')->find($id);
- findAll()
  - récupère toutes les entités et retourne un Array
- findBy(array $critères, array $order, $limite, $offset)
  - récupère une liste d'entité sur base de critères
- findByX($valeur)
  - identique à findBy, X étant la propriété de l'entité recherchée
- findOneByX($valeur)

Récupérer des entités

- A partir du contrôleur

```php
public function showAction($id)
{
    $em = $this->getDoctrine()->getManager();

    $product = $em->getRepository('AcmeStoreBundle:Product')->find($id);

    if (!$product) {
        throw $this->createNotFoundException(
            'Aucun produit trouvé pour cet id : '.$id
        );
    }

    // ... faire quelque chose comme envoyer l'objet $product à un template
}
```

Persister les entités

- Dans le contrôleur

```php
public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice('19.99');
    $product->setDescription('Lorem ipsum dolor');

    $em = $this->getDoctrine()->getManager();
    $em->persist($product);
    $em->flush();

    return new Response('Id du produit créé : '.$product->getId());
}
```

Message flash

- Permet d'envoyer un message à l'utilisateur de la réussite ou de l'échec de l'insertion
- Le message est stocké en session
- Dans le contrôleur

```php
try {
    $em = $this->getDoctrine()->getManager();
    $em->persist($category);
    $em->flush();
    $this->addFlash('notice', 'La categorie bien enregistrée.');
} catch (Exception $e) {
    $this->addFlash('notice', 'La categorie n\'est pas enregistrée.');
}
```

- Dans le template twig

```twig
{% for flashMessage in app.session.flashBag.get('notice') %}
    <div class="flash-notice">
        {{ flashMessage }}
    </div>
{% endfor %}
```

Les formulaires

- Installer le composant form
  - composer req form
- Configurer la route
- Créer manuellement une page formulaire avec Twig
  - Formulaire cycle
  -
    - {{ form(form) }}
    -

Utiliser un formulaire dans le contrôleur

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', TextType::class, array('label' => 'Tache'))
        ->add('dueDate', DateType::class, array('label' => 'A faire pour le '))
        ->add('save', SubmitType::class, array('label' => 'Sauvegarder'))
        ->getForm();

    $form->handleRequest($request);
    $task = $form->getData();

    if ($form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($task);
        $em->flush();
        return new Response('La tâche ajoutée avec succès !');    }

    return $this->render('AcmeAccountBundle:Account:register.html.twig', array('form' =>
$form->createView()));}
```

Validation

- Installer le composant
  - composer require validator
- Ajouter les contraintes de validation comme annotation dans la classe de l'entité

```php
// src/AppBundle/Entity/Task.php
 namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;

class Task
 {
        /**
        * @Assert\NotBlank() *
        /
        public $task;

        /**
        * @Assert\NotBlank()
        * @Assert\Type("\DateTime")
        */
        protected $dueDate;
}
```

Générer un formulaire basé sur une entité

- En ligne de commande

```
php bin/console make:form userType
```

- Générera le formulaire dans le dossier Form

```
class userType extends AbstractType
{
        /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nom')
            ->add('prenom')
        ;
    }
    ...
```

Utiliser un formulaire existant

- Définir l'action dans le contrôleur

```
public function addAction()
{
    $form = $this->createForm(userType::class, new User());
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // fait quelque chose comme sauvegarder la tâche dans la bdd

        return $this->redirect($this->generateUrl('add_success'));
    }

    return $this->render('add.html.twig',   array('form' => $form-
>createView()));
}
```

Validation d'une entité

- Toute entrée formulaire doit être validée

```php
use Symfony\Component\Validator\Constraints as Assert;

class user
{
    /**
     * @Assert\IsTrue(message = "This content is not valid")
     */
    public function isValid()
    {
        try {

          //  Load XMl with loadXML method
          //  XML Schema validation with schemaValidate method

        } catch ( \ErrorException $e)
        {
            return false;
        }
        return false;
    }
```

API REST

- Faire une copie du contrôleur
- Modifier les routes
- Ajouter la méthode

```
/**
 * @Route("/api/notes", name = "api_note_list")
 * @Method("GET")
 */
```

- GET envoie du JSON
- POST et PUT reçoivent du JSON

Réponse JSON

- Installer le composant serializer
  - composer require validator
- Etapes :
  - Sérialiser l'objet
  - Envoyer le json

```php
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;
```

```php
public function index()
{
    $encoders = array( new JsonEncoder());
    $normalizers = array(new ObjectNormalizer());
    $serializer = new Serializer($normalizers, $encoders);
    $em = $this->getDoctrine()->getManager();
    $products = $em->getRepository('App:Product')->findAll();
    $jsonContent = $serializer->serialize($products,'json');

    $response = new JsonResponse();
    $response->setContent($jsonContent);

    return $response;
}
```

Intégrer Bootstrap

- Ajouter le dossier css dans web
- Dans le layout twig général, ajouter dans le head

```
<link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}" />
<link rel="stylesheet" href="{{ asset('css/main.css') }}" />
```

- Pour les formulaires, ajouter dans le twig

```
{% form_theme form 'bootstrap_3_layout.html.twig' %}
```

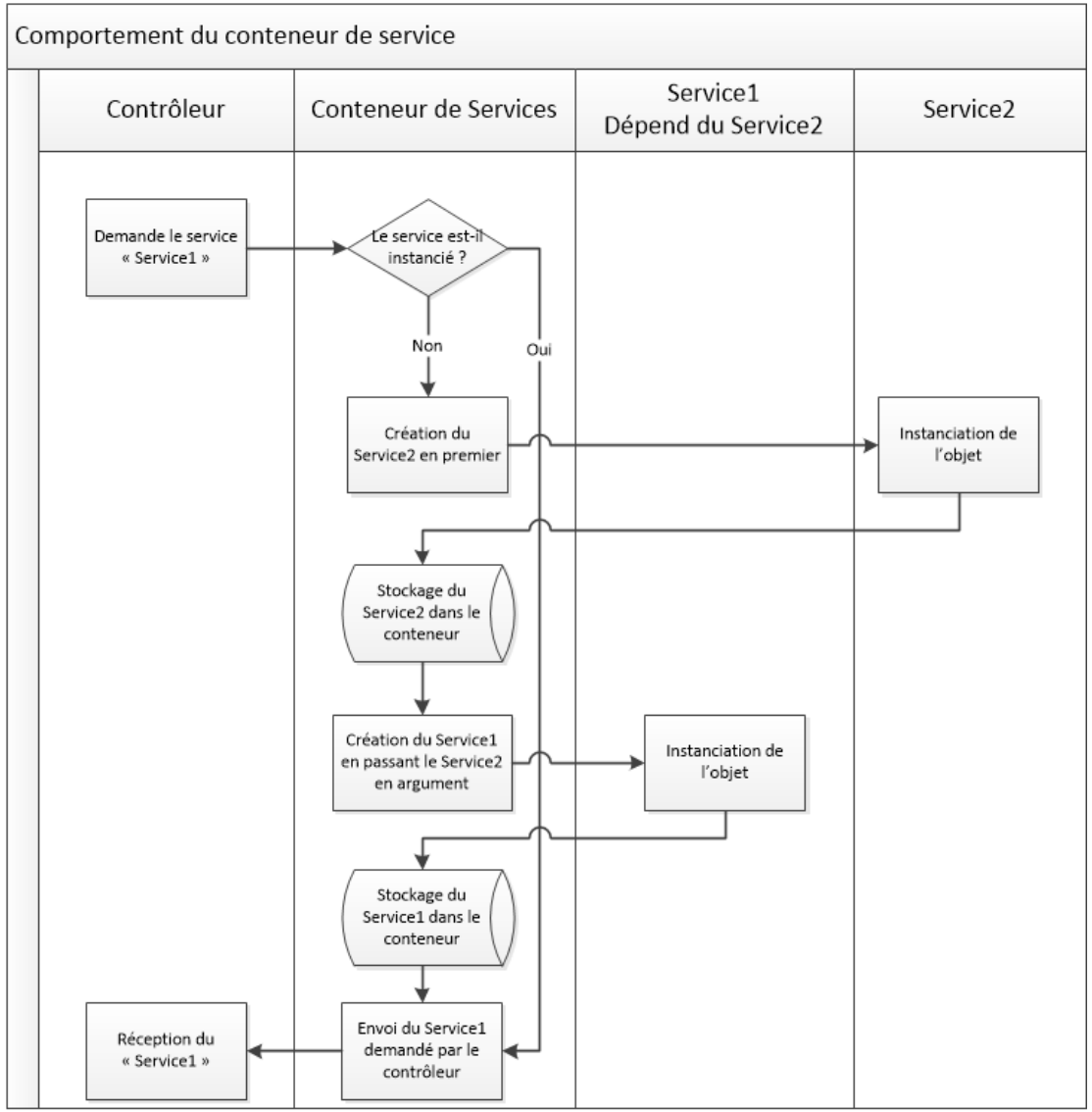
Les services

- Un service remplit une fonction
- Il est réutilisable partout dans le code
- Il nécessite parfois d'autres services
  - Exemple :
    - un service antispam a besoin d'un service de mail
- Un service est une classe associée à une configuration
- Des services sont préinstallés dans S2
- Toutes les classes S2 héritent de ContainerAware donc le contrôleur également
  - Exemple :
    - $mailer = $this->container->get('mailer');
    - ou en condensé
    - $mailer = $container->get('mailer');

Le conteneur de services

- Le conteneur de services organise et instancie les services grâce à leur configuration
- Lorsqu'un service à besoin d'un autre service c'est le conteneur de services qui le sait et qui se charge de la créer.
- Ce concept est appelé "injection de dépendance"

Le conteneur de services



Les services - création

- Création de la classe service dans le fichier decor.php
  - namespace demo\Bundle\helloBundle;

  - class decor
  - {
  - public function star($text)
  - {
  - return '*' . $text . '*';
  - }
  - }
- Déclaration dans DependencyInjection\services.yml
  - services:
  - hello.decor:
  - class: demo\Bundle\helloBundle\decor
- hello.decor devient le nom du service
- class définit le namespace et la classe

Les services - utilisation

- Dans le contrôleur :
  - // appel sur service
  - $decor = $this->container->get('hello.decor');

  - // utilisation du service
  - $text = $decor->star($name);

  - return $this->render('helloBundle:Default:hello.html.twig', array('name' => $text));

Injection de dépendance

- Exemple :
  - parameters:
  - hello.example.class: demo\Bundle\helloBundle\Example

  - services:
  - hello.example:
  - class: %hello.example.class%
  - arguments: [@service_id, "plain_value", %parameter%]

- Les arguments sont récupérés dans le construct du service
  - @service_id est le service qui sera lancé préalablement
  - "plain_value" est une valeur fixe
  - %parameters% paramètre de la classe

Résume

- php bin/console make:entity product
- php bin/console doctrine:database:create
- php bin/console doctrine:schema:update --force
- php bin/console make:controller product
- php bin/console make:form product
- créer add.html.twig

```
/**
 * @ORM\Column(type="string")
 */
public $description;
```

```
/**
 * @Route("/product/add", name="add_product")
 */
public function addAction(Request $request) {
    $product = new Product();
    $form = $this-
>createForm(ProductType::class,$product);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid())
    {
        try {
            $em = $this->getDoctrine()->getManager();
            $em->persist($product);
            $em->flush();
            return new Response('sucess');
        } catch (Exception $e) {
            return new Response('no sucess');
        }
    }
    return $this->render('product/add.html.twig',
['form'=>$form->createView()]);
}
```

```
{{ form_start(form) }}
{{ form_widget(form) }}
<input type="submit" value="Sauvegarder" />
{{ form_end(form) }}
```