Le framework PHP Symfony V6

Michel Meynard

UM

Univ. Montpellier

Table des matières I

1. Introduction
2. Introduction à Symfony 6
3. Ligne de commande symfony et serveur Web
4. Route, Controller, Response
5. Twig et les Vues
6. Flex
7. Services Symfony

Table des matières II

8 Environnement du projet

9 Profilage et Débogage

10 Doctrine

11 Formulaires Symfony

12 Authentification et sécurité

13 EasyAdmin : CRUD pour les entités

14 Développer un service Web (API REST) avec Symfony

Table des matières III

15 Partager son projet Symfony avec Git

16 Personnalisation des rendus de formulaires Symfony

17 La validation des formulaires et des entités

18 Conclusion

19 Bibliographie

Plan

1 Introduction

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

Prérequis I

Les étudiants doivent avoir des connaissances de base sur :

- le langage HTML ainsi que le protocole HTTP,
- notions de serveur HTTP, par exemple Apache ou Nginx ou Node.js ou ...
- client léger, navigateur Web, comme chrome, chromium, firefox, opera
- connaissance de base de Javascript : langage événementiel et fonctionnel dont un moteur (interpréteur) est incorporé dans le navigateur
- références HTML et JavaScript : [**?**]
- connaissance du langage PHP et du patron d'architecture MVC (Modèle Vue Contrôleur) permettant de structurer un site dynamique

Prérequis II

- En TP, il devra être capable de rechercher les références concernant PHP, HTML, Javascript, les feuilles de style CSS
  `https://www.w3schools.com/` ou
  `https://developer.mozilla.org/fr`
- références biblio. sur Symfony : site [1]

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

Objectifs

Les objectifs de ce cours sont nombreux :

- apprentissage d'un framework PHP ;
- étude de la dynamique d'une application Web : initialisation, saisie, traitement, ...
- technologies : routage, sessions, ORM ...
- réalisation d'un projet.

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

HTTP et architecture du Web I

Quelques définitions de base sont nécessaires :

HTTP `HyperText Transfer Protocol` est un protocole de communication **client-serveur** développé pour le World Wide Web (www)

HTTP inventé par Tim Berners-Lee avec les adresses web (URL) et le langage HTML

port utilisé par défaut le port 80

HTTP est un protocole **non orienté connexion**

client HTTP (navigateur) envoie une requête au serveur qui lui retourne une réponse généralement sous la forme d'un fichier HTML. Le client affiche alors la réponse qui contient généralement des **liens** hypertextes ou des formulaires, qui une fois cliqués ou soumis génèrent une requête au serveur etc.

HTTP et architecture du Web II

serveur HTTP Apache, Nginx, IIS, ... sont des serveurs logiciels installés sur des machines appelées également “serveur” (matériel)

client HTTP appelé aussi client léger ou navigateur Web, les plus connus sont Firefox, Internet Explorer, Opera, ...

URL *Uniform Resource Locator* est une chaîne de caractères codée en ASCII pour adresser les ressources du World Wide Web : document HTML, image, son, forum Usenet, boîte aux lettres électroniques, etc.

exemple d'URL appelée aussi **adresse web** : `http://www.google.fr/`, `mailto:toto@titi.fr`, `http://www.lirmm.fr/~meynard/Ens2/rubrique.php3?id_rubrique=36&x=1`, ...

HTTP et architecture du Web III

URL relative à l'intérieur d'une page HTML, des liens vers d'autres pages du même site peuvent être définis avec une écriture relative au répertoire courant de la page : par exemple, `../images/toto.jpg` est une URL relative.

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

Application Web I

- HTTP n'est pas orienté connexion : le serveur ne mémorise aucune information à propos du client
- entre 2 requêtes du client, il peut se passer 2 secondes ou un temps infini !
- Donc l'écriture des applications côté serveur doit prendre en compte cette absence de mémoire
- une part non négligeable du traitement dans une application peut être effectué côté client afin d'effectuer des vérifications (contrôles) mais aussi des calculs métiers !
- éviter de surcharger le serveur
- des règles de sécurité interdisent aux scripts côté client d'accéder au ressources du poste client (fichiers, imprimante, ...) sauf si l'utilisateur le demande !

Application Web II

- Actuellement, avec les Single Page Application (SPA), l'application est située majoritairement sur le client qui la charge au début puis cette appli. fait appel à des Web Services (API Rest) quand cela est nécessaire.

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

Application Côté Serveur I

L'application côté serveur utilise plusieurs technologies possibles :

cgi script (python, bash) ou binaire (compilé par g++) chargé dans un processus externe au serveur http (fork). La sortie standard de ce processus est redirigé dans la réponse envoyée par le serveur au client.

inconvénient principal des cgi perte de temps nécessitée par la création d'un processus pour chaque nouvelle requête ;

module le serveur intégre des modules interprétant des langages de programmation tels que Perl, Python, PHP. L'interprétation de script est beaucoup plus rapide car dans un Thread

Node.js un serveur Node.js est un démon JavaScript qui tourne sur le serveur et qui sert chaque requête sans créer de processus léger ou lourd en utilisant le caractère asynchrone de Js !

Plan

1. Introduction
   - Prérequis
   - Objectifs
   - HTTP et architecture du Web
   - Application Web
   - Application Côté Serveur
   - Application Côté Client

Application Côté Client I

L'application côté client peut utiliser plusieurs technologies possibles :

JavaScript langage de script interprété par le navigateur et permettant de manipuler l'arborescence du document(DOM) et possédant de nombreuses autres API (la norme s'appelle ECMAScript)

applet Java mini application Java permettant d'utiliser toute la puissance du langage Java (API, structures de données, ...) OBSOLÈTE

ActionScript langage de script compatible avec JavaScript (implémentation d'ECMAScript) et permettant de réaliser des animations vidéos et multimédia en Flash ... OBSOLÈTE

Java Web Start technologie de déploiement d'application Java : un lien HTML vous permet de télécharger une application Java indépendante du navigateur

Application Côté Client II

A noter que les technologies applet et ActionScript pour Flash sont obsolètes parce qu'elles nécessitaient des plugins non sécurisés qui sont aujourd'hui remplacés par des extensions Web

Plan

2 Introduction à Symfony 6

Symfony, qu'est ce que c'est ? I

- cadriciel (framework) libre permettant de développer des applications Web en PHP
- un ensemble de composants (form, httpClient, pdo) indépendants et réutilisables (avec ou sans Symfony) que des Content Management System peuvent utiliser (Drupal, phpBB, eZ Publish)
- une communauté de 600 000 développeurs s'entraidant
- une ligne de commande permettant de créer un nouveau projet, d'installer des `bundles` ...
- des `bundles` Symfony qui sont des abstractions Flex tels que Twig ou Doctrine permettant leur installation, configuration et automatisation même s'ils sont indépendants de Symfony (Doctrine)
- Symfony 6 **nécessite PHP 8 !** Avec PHP 7, on aura un projet Symfony 5.4

Plan

2 Introduction à Symfony 6
- Composer : un gestionnaire de dépendances
- Installation de Composer

Composer : un gestionnaire de dépendances I

- Historiquement `pear` fut un gestionnaire de paquetages PHP centralisé qui permettait de nombreuses fonctionalités dont la gestion des dépendances entre paquetages mais qui était un peu rigide
- les auteurs de paquetages ou de librairies PHP sont maintenant hébergés sur différents gestionnaires de version (`GitHub`, `GoogleCode`, `...`) et proposent leur librairie sous différents formats (zip, tgz, phar, ...)
- Composer est un gestionnaire de dépendances de paquetages
- Il est associé à un gestionnaire de paquetage `Packagist` qui lui maintient en dépôt (git) tous les paquetages utilisables avec Composer
- Quelques paquetages gérés par composer : Doctrine (ORM), API Platform (outil de gestion d'API REST), Maker (outil pour créer des assistants Symfony)

Plan

2 Introduction à Symfony 6
- Composer : un gestionnaire de dépendances
- Installation de Composer

Installation de Composer I

La façon la plus simple est d'aller sur la page Web `https://getcomposer.org/download/` et de lire la doc.
Sur MacOS X :

```
$ cd public_html/
$ mkdir MonProjet
$ cd MonProjet
$ curl -sS https://getcomposer.org/installer | php
```

- La commande curl permet de télécharger l'exécutable php `installer.phar` (PHp ARchive)
- Puis on passe via un tube ce même fichier à l'interprète `php` qui exécute l'installateur
- A partir de là, le fichier `composer.phar` est installé et est disponible en ligne de commande

Installation de Composer II

- Pour installer des paquetages dans un projet, il suffit d'éditer le fichier JSON `composer.json` indiquant ce que l'on veut installer et de lancer la ligne de commande `php composer.phar install`

Plan

3 Ligne de commande symfony et serveur Web

Plan

3 Ligne de commande symfony et serveur Web
- Installation
- Création d'un projet Symfony 6

Installation I

Depuis la version 5, il existe 2 Command Line Interface (CLI) permettant de réaliser des actions dans un projet symfony :

- `php bin/console make:controller`
- `symfony console make:controller`

La première est historique et est utilisée majoritairement. La seconde nous paraît plus simple mais pour utiliser `symfony` qui n'est pas un bundle mais un exécutable (MacOS, Windows ou Linux), il faut l'installer dans un répertoire exécutable (PATH) :

```
$ curl -sS https://get.symfony.com/cli/installer | bash
$ mv ~/.symfony/bin/symfony /usr/local/bin/symfony
```

La façon la plus simple est d'aller sur la page Web `https://getcomposer.org/download/` et de lire la doc.

Ligne de commande symfony I

La commande `symfony` permet de réaliser un grand nombre de tâches dont voici quelques exemples :

- `symfony new --full monprojet` pour créer un répertoire monprojet contenant une application web traditionnelle (twig, doctrine, ...)
- `symfony new monprojet` pour créer un répertoire monprojet contenant un microservice, une application CLI, ou une API RESTful
- `symfony new Sf5 --version=5.4` pour créer un projet en version 5.4 même avec PHP 8
- lancer un serveur web local sur le port 8000 avec journalisation des accès : `symfony server:start`
- pour obtenir de l'aide sur une ligne de commande : `symfony server:start -help`

Ligne de commande symfony II

- on préférera donc lancer le serveur en arrière-plan (`-d`) et sur le port 8080 (`-port=8080` : `$ symfony server:start -d -port=8080`
- on peut aussi lancer le navigateur sur la page d'accueil : `$ symfony open:local`
- on pourra automatiser ces commandes dans un script bash `serveur` afin de les relancer en début de période de développement !
- pour visualiser le journal d'accès au service web (sans ouvrir le fichier) : `symfony server:log` (quitter avec Ctrl-C)
- si on veut utiliser https, il faut créer une authorité locale de certification, enregistrer un certificat, ... mais symfony simplifie cette séquence de tâches : `symfony server:ca:install`
- par la suite, https sera utilisé par défaut avec `symfony open:local`
- `symfony console ...` permet d'effectuer toute commande console (voir ci-après)

Ligne de commande symfony III

- `symfony composer req maker --dev` permet de lancer une commande composer (require, install, ...)

Commande console : php bin/console I

- indispensable pour administrer le projet, la liste des commandes de console est variable selon les bundles installés
- la création d'un projet ne peut être réalisé que par la CLI composer ou la CLI symfony
- le lancement d'un serveur ne peut plus être fait que par la CLI symfony
- toutes les commandes consoles `php bin/console xyz` sont accessibles via `symfony console xyz`
- cependant, beaucoup d'exemples trouvés sur le web utilisent encore `php bin/console xyz` ...

Plan

3 Ligne de commande symfony et serveur Web
- Installation
- Création d'un projet Symfony 6

Création d'un projet Symfony6 (composer) I

On peut créer un projet Symfony soit avec composer, soit avec la ligne de commande symfony. Plutôt qu'un long discours, créons un projet Symfony Sf6Cours initialisé à un squelette minimal pour un site Web classique :

- télécharger `composer` qui est un outil php permettant de gérer les dépendances entre des modules php
- `htdocs$ composer.phar create-project symfony/skeleton ↪ Sf6Cours`
- nous venons de créer un projet symfony nommé `Sf6Cours` dans le répertoire de même nom
- `cd Sf6Cours/public; php -S localhost:8888` nous permet de lancer un serveur Web que nous visualisons sur un navigateur à l'url `http://localhost:8888/` : `public/index.php` est le routeur principal
- on peut également utiliser un autre serveur Web (Apache, Nginx, symfony local web server)

Création d'un projet Symfony (cli symfony) I

- télécharger `symfony` qui est un binaire exécutable depuis `https://symfony.com/download` selon votre OS
- `htdocs$ symfony new --webapp mon-super-projet`

  pour un projet de site complet (twig, doctrine, ... 50 Mio)
- `htdocs$ symfony new mon-mini-projet`

  pour un projet de site minimal (6.7Mio)
- nous créons interactivement un projet symfony dans un répertoire de même nom que le projet
- `cd Sf6Cours/public; symfony server:start` nous permet de lancer un serveur Web que nous visualisons sur un navigateur à l'url `http://localhost:8000/`
- la ligne de commande symfony est un adaptateur (*design pattern adapter, wrapper*) de composer mais pas seulement

Plan

4 Route, Controller, Response

Les fondamentaux : Route, Controller, Response I

- le projet Sf6Cours est un squelette d'une dizaine de fichiers pouvant se transformer en un service Web API REST ou une webapp ou un microservice
- la première notion fondamentale est le **routage** qui va permettre d'indiquer les chemins atteignables et leur localisation
- la seconde concerne le **contrôleur**, élément essentiel du patron d'architecture MVC
- la troisième concerne les classes `Request` mais surtout `Response` qui modélisent le protocole `HTTP`

Routage, contrôleur et réponse I

- le fichier `config/routes.yaml` contient le type d'association entre un **chemin** (path) et une **méthode d'un contrôleur**
- Depuis Symfony 6, le type d'association par défaut est `atttribute` (PHP natif) mais on pourrait définir `annotation` `#[Route(...` ou `yaml`

```
# config/routes/attributes.yaml
controllers:
  resource: ../../src/Controller/
  type: attribute
```

- Les attributs PHP 8(ou bien les annotations) sont utilisés dans différents composants Symfony (routage, doctrine, contraintes de validation ...)
- un attribut PHP 8 a la syntaxe suivante :
  `#[Nom('valeur', prop:'val2', ...)]`

Routage, contrôleur et réponse II

- En Symfony 6, **les attributs de Route permettent d'accoler la méthode déclenchée du contrôleur à la description de la route**
- Les attributs PHP8 sont natifs et utilisent l'API `Reflection` de PHP. Ils évitent l'utilisation d'un parser d'annotations qui était lancé avant l'interprète PHP.

Un exemple simple de Contrôleur I

```php
<?php
namespace App\Controller; // espace de nom
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↪  // superclasse
use Symfony\Component\HttpFoundation\Response; // retournée
use Symfony\Component\Routing\Annotation\Route; // route

class MonController extends AbstractController{ // la classe
  #[Route("/")]                                  // attribut PHP
  public function index(): Response {            // une méthode annotée
    return new Response("Page d'accueil !");
  } // suivent d'autres méthodes annotées
}
```

Le routage en détail I

La doc sur le routage étant volumineuse, on donne quelques pistes ci-dessous :

- le premier paramètre obligatoire de l'attribut `Route()` est son chemin e.g. `/tournoi/creer`
- on définit fréquemment un nom de route afin de pouvoir s'y rendre dynamiquement (PHP) :
  `@Route("/tournoi/liste", name="tournois")`
- on peut utiliser un **paramètre** de route avec son nom entre accolade : `@Route("/tournoi/{id}")`. La valeur de ce paramètre sera passée à la méthode de contrôleur qui doit le nommer :
  `public function tournoi($id): Response { ...`
- lorqu'un paramètre fait partie de la route, on peut vérifier sa **validité** grâce à une exp. rég. : `@Route("/tournoi/{id<\d+>}")`. Dans ce cas, la valeur requise est une suite de chiffres décimaux

Le routage en détail II

- on peut également fournir la valeur par défaut d'un paramètre : `@Route("/hello/{nom<[a-zA-Z]+>?Dupont}")`
- On peut avoir plusieurs paramètres séparés par des slashs
- par défaut toutes les méthodes HTTP (get, post, put, ...) sont concernées par les routes : si on veut limiter les méthodes, on ajoutera à la route une propriété `methods` de valeur une union des méthodes possibles e.g. :

  ```
  #[Route('/api/posts/{id}', methods: ['GET', 'HEAD'])]
  ```

- on peut ajouter une **condition** d'accès à cette route portant sur différents paramètres e.g. :

  ```
  condition: "request.headers.get('User-Agent') matches
  ↪  '/firefox/i'"_
  ```

  Une condition permet d'évaluer une expression PHP quelconque

Le routage en détail III

- si un utilisateur ne fournit pas un paramètre, ou que ce dernier est invalide ou qu'une condition n'est pas respectée, le routage ne fonctionnera pas et on obtiendra une réponse 404 (Page Not Found)
- si plusieurs routes définies sont en conflit, ce sera la première méthode correspondante trouvée qui sera utilisée

Un exemple complet de routage I

On va créer un premier contrôleur gérant plusieurs routes avec paramètres :

```php
<?php // src/Controller/MonController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class MonController extends AbstractController{
  #[Route("/")]
  #[Route("/bonjour/{nom<[a-z]+>?toi}/{id<\d+>?13}")]
  public function bonjour($nom=null, $id=null) {
    if (!$nom){
      return new Response('Bonjour le monde !');
    } else {
      return new Response("Bonjour $nom d'id $id !");
    }
  }
```

Un exemple complet de routage II

```
    #[Route("/mult/{x<\d+>?1}/{y<\d+>?1}")]
    public function mult($x,$y){
      return new Response($x*$y);
    }
}
```

http ://localhost :8888/	Bonjour le monde !
http ://localhost :8888/bonjour	Bonjour toi d'id 13 !
http ://localhost :8888/bonjour/toto	Bonjour toto d'id 13 !
http ://localhost :8888/bonjour/toto/5	Bonjour toto d'id 5 !
http ://localhost :8888/mult	1
http ://localhost :8888/mult/8/7	56
http ://localhost :8888/mult/123	123

TABLE – Quelques routes et leurs réponses

Le routage pour réaliser une API REST I

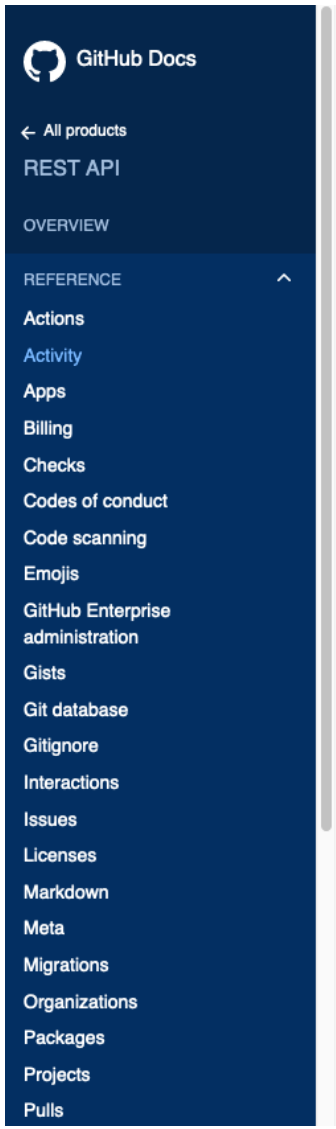
Le routage Symfony est une base indispensable pour réaliser un backend sous la forme d'un service Web utilisant API RESTful :

- Une architecture client-serveur constituée de clients, de serveurs et de ressources, avec des requêtes gérées via HTTP
- Des communications client-serveur **sans état**, c'est-à-dire que les informations du client ne sont jamais stockées entre les requêtes traitées séparément de manière totalement indépendante
- La possibilité de mettre en cache des données afin de rationaliser les interactions client-serveur
- Une interface uniforme entre les composants qui permet un transfert standardisé des informations (souvent en JSON)
- les ressources demandées sont identifiables (BD relationnelle ou NoSQL) et séparées des représentations envoyées au client (JSON)

Le routage pour réaliser une API REST II

- les ressources sont manipulées par le client au moyen de la représentation reçue, qui contient suffisamment d'informations
- les messages autodescriptifs renvoyés au client contiennent assez de détails pour décrire la manière dont celui-ci doit traiter les informations
- l'API possède un hypermédia, qui permet au client d'utiliser des hyperliens pour connaître toutes les autres actions disponibles après avoir accédé à une ressource

Un exemple de doc de l'API REST de GitHub I



`GET` `/notifications`

Parameters

Name	Type	In	Description
`accept`	string	header	Setting to `application/vnd.github.v3+json` is recommended.
`all`	boolean	query	If `true`, show notifications marked as read.
`participating`	boolean	query	If `true`, only shows notifications in which the user is directly participating or mentioned.
`since`	string	query	Only show notifications updated after the given time. This is a timestamp in ISO 8601 format: `YYYY-MM-DDTHH:MM:SSZ`.
`before`	string	query	Only show notifications updated before the given time. This is a timestamp in ISO 8601 format: `YYYY-MM-DDTHH:MM:SSZ`.
`per_page`	integer	query	Results per page (max 100).
`page`	integer	query	Page number of the results to fetch.

Code samples

Shell

```
curl \
  -H "Accept: application/vnd.github.v3+json" \
  https://api.github.com/notifications
```

JavaScript (@octokit/core.js)

Routage dynamique I

- Une route Symfony est une association entre un nom de route et son chemin :

```php
<?php #[Route("/tournoi/liste", name="tournois")]
```

- on peut s'y rendre de multiples façons : saisie du chemin dans la barre d'adresse, clic sur un lien HTML, ...
- Le plan de routage est défini à la conception et toute page HTML peut posséder un lien référençant une route de l'application dans une vue :

```html
<a href="/tournoi/liste">Liste des tournois</a>
```

- Il est également possible de router dynamiquement depuis un contrôleur grâce à une redirection située dans une méthode de contrôleur. Dans ce cas, la redirection est réalisée **via le nom de la route** et non son chemin :

```php
<?php return $this->redirectToRoute('tournois');
```

Plan

5 Twig et les Vues

Twig un moteur de template (gabarit) PHP I

Twig (voir [2]) est destiné aux concepteurs (designers) et aux développeurs. C'est un langage qui étend HTML et c'est aussi un moteur de rendu qui effectue la liaison entre les données de l'application et les constructions twig.

- Symphony est livré avec un nombre minimal de bundles installés par défaut
- le développeur doit installer des bundles au fur et à mesure de ses besoins à l'aide de Composer
- `composer require twig` : c'est la commande qui installe le **bundle** (paquet) twig et qui met à jour `composer.json`
- en réalité, le bundle a un autre nom mais c'est le composant flex de Symfony qui écrit la recette pour les dépendances de twig (à oublier)
- L'installation par composer de twig a généré :

Twig un moteur de template (gabarit) PHP II

- un répertoire `templates` contenant un fichier modèle `base.html.twig` : ce fichier définit un modèle de page contenant des blocs qui seront redéfinis par les templates héritant de ce fichier
  `{% extends 'base.html.twig' %}`
- `config/packages/twig.yaml` une config par défaut de twig
- `config/packages/test/twig.yaml` une config par défaut de twig pendant les tests

- Afin de ne pas mélanger notre classe DefaultController qui renvoyait grâce à index et bonjour des réponses HTTP basiques, nous allons créer un nouveau contrôleur retournant des réponses au format HTML grâce à twig

Création d'un autre contrôleur I

- un Bundle Symfony est une extension (plugin) permettant d'ajouter des fonctionnalités à celles installées par défaut.
- Installation du bundle `maker` qui permet de fabriquer en ligne de commande de nombreuses choses dont des contrôleurs :

  ```
  $ symfony composer req maker --dev
  ```

- on aurait pu installer ce bundle directement avec `composer.phar` ... On dispose maintenant d'une console permettant de lancer des commandes et des sous commandes
- `symfony console make:controller tournoiController` crée un nouveau contrôleur qui possède déjà une fonction index :

Création d'un autre contrôleur II

```php
<?php
class TournoiController extends AbstractController{
  #[Route("/tournoi", name="tournoi")]
  public function index(): Response{
    return $this->render('tournoi/index.html.twig', [
      'controller_name' => 'TournoiController',
    ]);
  }
}
```

- la route annotée `/tournoi` provoquera le calcul du rendu (HTML) par la méthode `render()` de la classe parent `AbstractController`
- Ce nouveau contrôleur (en supplément de DefaultController) est situé dans le répertoire `src/Controller` tandis que le template twig est situé dans `templates/tournoi`

Le template généré par défaut I

```
{% extends 'base.html.twig' %}
{# hérite du parent base.html.twig #}

{% block title %}Hello TournoiController!{% endblock %}
{# définition du bloc title utilisé par la base #}

{% block body %} {# définition du bloc body utilisé par la base
↪  #}

<div class="example-wrapper">
  <h1>Hello {{ controller_name }}! </h1>

  This friendly message is coming from:
  <ul>
  <li>Your controller at <code><a href="{{
↪  '/Applications/MAMP/htdocs/ ...
↪  /TournoiController.php'|file_link(0)
↪  }}">src/Controller/TournoiController.php</a></code></li>
```

Le template généré par défaut II

```
  <li>Your template at <code><a href="{{ '/Applications/MAMP/
↪   ... /tournoi/index.html.twig'|file_link(0)
↪   }}">templates/tournoi/index.html.twig</a></code></li>
  </ul>
</div>
{% endblock %}
```

Plan

5 Twig et les Vues
- Syntaxe des template Twig

Syntaxe RAPIDE des templates twig I

Voici les principales constructions :

- `{# mes comment. sur plusieurs lignes #}`
- `{{ mavar }}` affiche une variable twig qui sera associée à une valeur PHP lors du rendu :

```
return $this->render('mapage.html.twig',[ 'mavar' => $x,
```

- `{{ 3+6 }}` **évaluation d'une expression** (9)
- `{{ expression | filtre }}` permet de transformer la valeur de l'expression en lui appliquant un filtre e.g. `{{ 'TOTO'|lower }}` donnera 'toto', `{{ 42.55|round }}` donnera 43
- `{{ "now"|date('d/m/Y H:i', timezone="Europe/Paris") }}` attention now n'est pas une variable mais un mot réservé qui désigne l'heure courante

Syntaxe RAPIDE des templates twig II

- `{{ mavar.prop }}` accès à l'attribut prop de mavar : methode ou propriété d'un objet PHP, ou item d'un tableau PHP par sa clé string 'prop'. Si la propriété est privée, la méthode `getProp()` sera utilisée
- `{{ mavar['prop'] }}` est aussi valide pour les tableaux
- `{% for item in liste %} <li> item {% endfor %}` **exécution itérative d'instructions**. Ici, affiche tous les items de la liste qui doit être un Traversable (array ou classe). Cette variable liste doit être passée par le contrôleur dans le tableau de paramètres de la fonction render() :

```
return $this->render('tournoi/index.html.twig', [
'controller_name' => 'TournoiController',
'liste' => ['Michel', 'Meynard']
```

le template for affichera alors 2 items de liste : Michel puis Meynard

Syntaxe RAPIDE des templates twig III

- `{% for key, user in users %}<li>{{ key }}: {{ user.usernam`
  itération sur les clés et les valeurs
- la structure de controle if avec
  `'product' => ['nom' => 'marteau', 'stock' => "5"]`

```
{% if product.stock > 10 %} Disponible
{% elseif product.stock > 0 %} Plus que {{ product.stock }}
↪  {{ product.nom }} disponibles !
{% else %}  Indisponible
{% endif %}
```

- `{% extends 'parent.twig' %}` un template qui **hérite** (extends) d'un parent en récupèrant la structure du parent et en remplaçant (surchargeant) les blocs définis dans le parent par les siens
- `{% block X %}contenu{% endblock %}` permet de définir le contenu d'un bloc X qui pourra être utilisé une ou plusieurs fois

Syntaxe RAPIDE des templates twig IV

- d'autres balises (tag) twig existent (voir doc [2])

Un exemple de rendu par Twig I

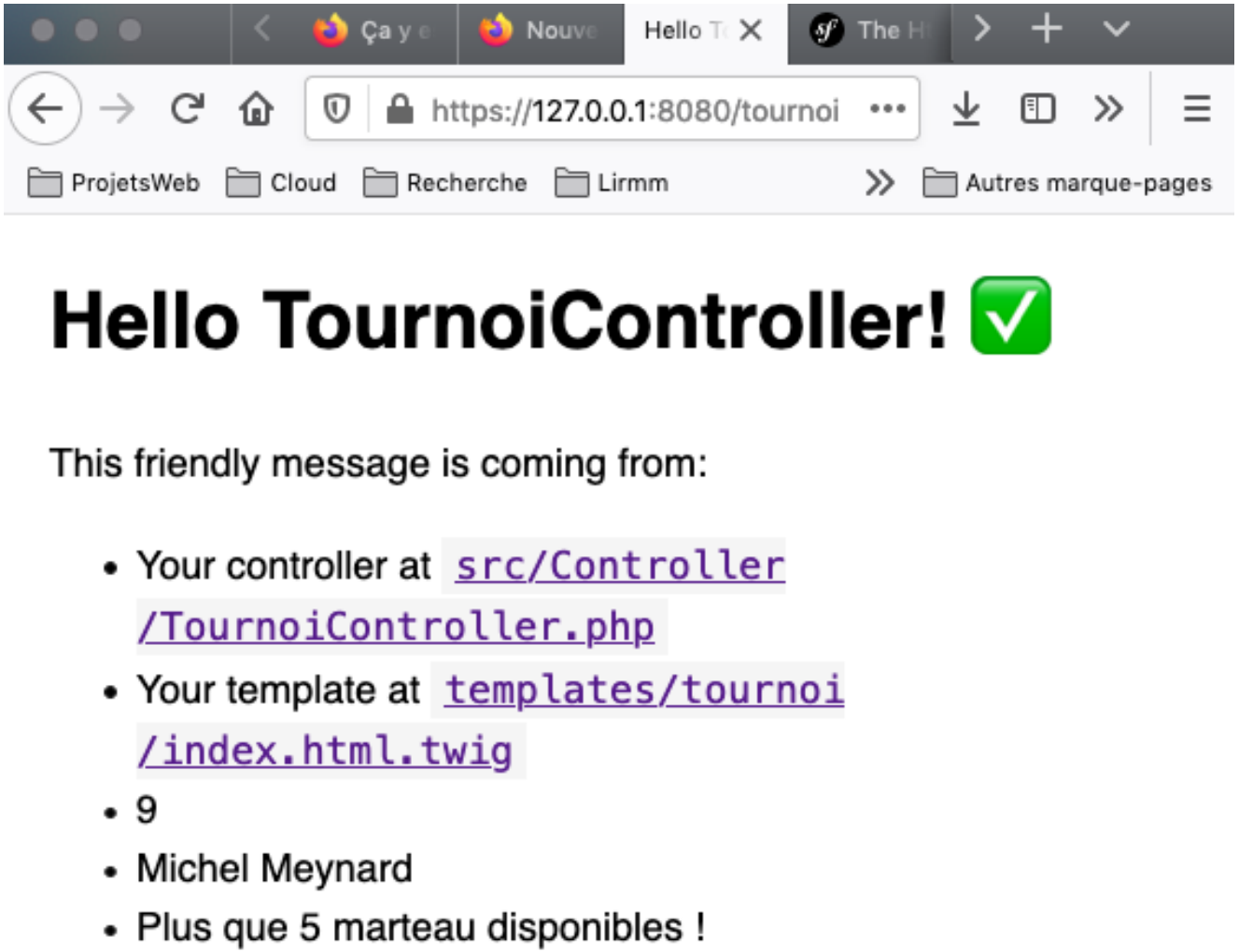
Soit le code suivant :

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

{% block title %}Hello TournoiController!{% endblock %}
{# définition du bloc title utilisé par la base #}

{% block body %} {# définition du bloc body utilisé par la base
↪  #}

<div class="example-wrapper">
  <h1>Hello {{ controller_name }} !</h1>
  This friendly message is coming from:
  <ul>
    <li>Your controller at <code><a href="{{
↪  '/Applications/MAMP/htdocs/Sf6Cours/src/Controller/TournoiContr
↪  }}">src/Controller/TournoiController.php</a></code></li>
```

Un exemple de rendu par Twig II

```
    <li>Your template at <code><a href="{{
↪   '/Applications/MAMP/htdocs/Sf6Cours/templates/tournoi/index.html
↪   }}">templates/tournoi/index.html.twig</a></code></li>
    <li> {{ 3+6 }}
    <li> {% for i in liste %} {{ i }} {% endfor %}
    <li> {% if product.stock > 10 %} {{ product.nom }}
↪   disponible
{% elseif product.stock > 0 %} Plus que {{ product.stock }} {{
↪   product.nom }} disponibles !
{% else %}  Indisponible
{% endif %}
  </ul>
</div>
{% endblock %}
```

Un exemple de rendu par Twig III



on obtient le rendu :

Twig et app I

En plus des paramètres passés par le contrôleur au template Twig, Symfony injecte dans chaque template une variable app représentant l'application qui contient :

- `user` l'utilisateur (`user`) authentifié ou `null` si non authentifié
- `request` les données de la requête
- `session` les données de la session PHP
- `debug` booléen indiquant si on est en mode débogage
- ...

Résumé des épisodes précédents I

Symfony permet de :

- définir différents fichiers contrôleurs, dans le rép. `Controller`, contenant des fonctions PHP, **annotées** par une ou plusieurs routes, et qui retournent un résultat `Response` qui peut contenir :
  - une simple chaîne de caractères (Bonjour le monde)
  - un rendu HTML basé sur un template twig situé dans le rép. `templates`. Les différents templates peuvent utiliser l'héritage pour redéfinir certains blocs.
  - des données JSON si on veut implémenter une API REST
- d'utiliser la commande `symfony` pour lancer un serveur HTTP(S), créer un nouveau projet, ouvrir votre navigateur préféré sur la page en cours de développement, installer via composer des bundles (maker), ...

Plan

6 Flex

Flex et les bundles I

- la philosophie de Symfony est de débuter un projet avec le minimum de composants nécessaires installés grâce à `composer` puis d'installer et configurer des **bundles** supplémentaires au fur et à mesure du développement
- par exemple, ni twig ni doctrine ne sont installés mais `symfony/flex` l'est !
- Flex est un plugin (extension) de composer permettant d'associer une recette (recipe) à un nom de bundle (alias) afin de réaliser plusieurs actions suite à `composer req alias` (avec alias=twig) :
  - installer le paquet composer symfony/twig et ses dépendances correspondantes
  - réaliser la configuration de ce paquet dans le répertoire config/ du projet symfony
  - créer des fichiers de base comme par exemple `base.html.twig` et son répertoire parent : templates

Flex et les bundles II

- quelques alias Flex indispensables :
  - `composer require profiler` : ajoute une barre noire en bas de page permettant de mesurer les performances de l'appli. (cache, temps de chargement, ...). Il requiert automatiquement le bundle twig (2 recettes)
  - `composer require api` : 5 recettes ajoutées Doctrine, Validator, Security ... utilisés pour réaliser des API REST, il suffit de créer une entité Doctrine annotée pour créer automatiquement des méthodes de contrôleurs associées à la route /api/monentite ... qui seront invoquées en fonction de la méthode HTTP utilisée (POST, GET, PUT, DELETE)
- Tout bundle installé peut être simplement désinstallé par `composer remove api`

Plan

7 Services Symfony

Architecture Orientée Service I

- Le patron d'*architecture orientée service* permet de séparer certaines fonctionalités utilisées à plusieurs endroits de l'application dans des services et est présent dans de multiples frameworks (Angular, Symfony, ...)
- chaque service est spécialisé pour un nombre faible de tâches (souvent 1)
- Le service est habituellement une unique instance d'une classe conservée dans l'unique *conteneur* qui regroupe tous les services
- Le conteneur gère les dépendances entre services, instancie un service la première fois qu'on lui demande, puis le conserve afin de le fournir à la demande

Services Symfony I

De nombreux services prédéfinis existent dans Symfony et sont accessibles depuis tout contrôleur :

- `mailer` permet d'envoyer des mails
- `logger` permet de journaliser les actions importantes réalisées sur le site telles que les connexions
- un service d'authentification est toujours utile
- un service de liaison avec une Base de Données permettra d'effectuer des requêtes
- En Symfony, toute classe définie par le développeur peut-être vue comme un service
- Les services sont utilisés grâce :
  - au principe d'**injection de dépendance** (Dependency Injection ou DI) comme dans d'autres cadriciels

Services Symfony II

- ou par l'accès au container qui est un attribut de toute classe Controller :

  ```
  $this->container->get('mailer')}
  ```

Dependency Injection (DI) I

C'est une méthode de programmation qui permet d'écrire un meilleur code plus facilement maintenable en rendant les composants plus indépendants grâce au patron de conception d'**inversion de contrôle** :

- dans une application objet classique, imaginons qu'un objet a :A dans sa méthode f() ait besoin d'un objet b :B exécutant sa méthode g(), qui ait elle même besoin d'un objet c :C et de sa méthode h()
- classiquement, les liens entre les objets sera statiquement inscrit dans le code : f() créera un objet de type B avant d'appeler g()
- avec **l'inversion de contrôle**, on programmera en commençant par créer l'objet c qu'on donnera à b lors de sa création (constructeur) puis on créera a en lui donnant b
- puis a appellera f() qui appellera g() sur l'objet b qui appellera h() sur l'objet c :C

Dependency Injection (DI) II

- cela permettra, par exemple, de changer la classe de l'objet intermédiaire b en une autre sans avoir à réécrire A
- il faut bien entendu que ces différentes classes aient une interface commune qui contienne la méthode g()
- le couplage entre l'utilisateur et le service est léger et sera réalisé lors de la construction de l'utilisateur
- mais on peut encore faire mieux grâce à PHP-DI qui est un paquet Composer permettant de ne pas instancier explicitement lors de l'appel au constructeur mais plutôt en utilisant une directive de configuration ...

PHP-DI Comment ça marche ? I

Principe du fonctionnement de l'injecteur :

1. L'application crée un conteneur (container) de composants (dans le cas d'une application web PHP, en cycles requête/réponse, le conteneur est à créer de préférence au niveau du script répartiteur frontal, vers lequel toutes les requêtes sont acheminées)
2. Le conteneur s'initialise et prend connaissance des directives qui caractérisent son comportement (fichier de configuration)
3. L'application demande au conteneur de lui fournir chaque composant qu'elle souhaite utiliser, au moment où elle en a besoin. Si une instance du composant demandé ne s'y trouve pas encore, le conteneur la crée (assemblée en cascade avec toutes les pièces qui constituent le composant parmi les couches inférieures, pièces elles-mêmes puisées/créées automatiquement depuis le conteneur)

DI en pratique avec Symfony I

- avec Symfony, l'auto-cablage (autowiring) représente la capacité du conteneur de créer (s'il n'existe pas) et d'injecter le service dans la classe
- ceci est possible grâce au type hinting (indice de type) qui doit être fourni au constructeur de classe et doit correspondre à la classe du service attendu
- un exemple d'injection de service de journalisation dans une méthode de contrôleur

```php
<?php #[Route("/products")]
public function list(LoggerInterface $logger): Response {
  $logger->info('/products request');
  // ...
}
```

DI en pratique avec Symfony II

- l'injection de service par cablage automatique peut être réalisée dans n'importe quelle méthode grâce à l'indice de type (type hint) indispensable
- pour voir la liste des services contenus dans le container :
  `$ php bin/console debug:container` (plusieurs centaines)

Plan

7 Services Symfony
- Session
- Le service Request
- Création d'un service spécialisé

Un exemple de service : session PHP I

- Une session PHP permet de mémoriser (dans `$_SESSION`) des couples (clé, valeur) conservées entre plusieurs requêtes grâce à un cookie `PHPSESSID`.
- Symfony utilise cette capacité en fournissant une interface objet (SessionInterface) permettant la régularité du code
- L'utilisation des sessions a évolué depuis Symfony 6 car l'interface `SessionInterface` n'est plus injectable !
- Une session symfony peut être obtenue par la méthode `getSession()` utilisée sur le service `Request` disponible (par injection) dans les contrôleurs, ou sur le service `RequestStack` disponible partout !
- Pour l'exemple, on crée une route `/session/cle/valeur` permettant d'enregistrer dans la session un couple (clé, valeur)

Un exemple de service : session PHP II

- Si le couple est absent de la route, on affichera la liste des couples précédemment enregistrés :

```php
<?php #[Route("/session/{k?}/{v?}")]
function session($k,$v, Request $req) : Response {
  $session=$req->getSession();   $r="";
  if($k===null){
    foreach($session->all() as $c => $w){
      $s=strval($w);
      $r .= strval($c)."=>".$s."<br>";
    }
  }else{
    $session->set($k,$v);
  }
  return new Response($r);
}
```

- l'interface `SessionInterface` contient les méthodes suivantes :

Un exemple de service : session PHP III

- `all()` retourne un tableau associatif de tous les couples (clé, valeur)
- `get('name', 'default_value')` retourne la valeur actuelle de la clé ou si elle n'existe pas, le second paramètre (optionnel)
- `set($key, mixed $value)` affecte une valeur à une clé en remplaçant l'ancienne si elle existait
- `has('cle')` retourne un booléen indiquant si la clé est présente dans la session
- `remove('cle')` supprime le couple dans la session et rertourne la valeur
- `clear()` supprime tous les couples de la session

- les couples que nous avons vus sont stockés dans un `AttributeBag` qui est lui même présent dans la variable superglobale PHP `$_SESSION`. Mais d'autres configuration peuvent exister : FlashMessage, session dans une BD, ...

Plan

7 Services Symfony
- Session
- Le service Request
- Création d'un service spécialisé

Le service Request I

Jusque là, afin de récupérer les paramètres de la chaîne de requête (query string), on fabriquait un objet Request en utilisant un appel :

```
$req = Request::createFromGlobals();
$nom=$req->query->get('nom'); // récup. $_GET['nom']
```

Cette méthode statique (createFromGlobals) permet de récupérer tous les paramètres de la requête qu'on récupère en PHP traditionnel grâce aux tableaux super-globaux `$_POST`, `$_GET`, `$_SERVER`, ...

Cependant, Symfony peut réaliser plus facilement l'injection dans toute méthode de contrôleur grâce à l'indice de type :

```
public function index(Request $request): Response          {
```

Plan

7 Services Symfony
- Session
- Le service Request
- Création d'un service spécialisé

Création d'un service spécialisé I

Dans cet exemple on veut mémoriser dans une session, l'historique des opérations réalisées précédemment :

- on crée une classe `HistoOp` de service dont le constructeur injecte le service de session :

```php
<?php /** @file src/Service/HistoOp.php historique
↪  d'opérations */
namespace App\Service;
use
↪  Symfony\Component\HttpFoundation\Session\SessionInterface;

class HistoOp { // service contruit à chaque fois !!!
  private $session;
  public function __construct(Request $req){
    $this->session = $req->getSession();
  }
```

Création d'un service spécialisé II

- De plus, cette classe possède 2 méthodes, une pour ajouter une opération à l'historique dans un tableau PHP dans la session, et une qui retourne toutes les opérations historisées :

```php
<?php /** ajoute une opération (chaîne de car) d'historique
↪  (tableau) */
public function addOp(string $s){
  $h=$this->session->get('histoop',[]);
  $h[]=$s; // push
  $this->session->set('histoop', $h);
}
public function getOps(){
  return $this->session->get('histoop');
}
```

- Il ne reste plus qu'à créer une méthode de contrôleur pour ajouter et une pour lister l'historique :

Création d'un service spécialisé III

```php
<?php #[Route("/+/{x<\d+>?1}/{y<\d+>?1}")]
function add($x,$y, HistoOp $histoop) : Response { //
↪  injection
  $r=strval(intval($x)+intval($y));
  $histoop->addOp($x.'+'.$y.'='.$r); // sauve
  return new Response($r);
}
```

- l'injection d'un service non prédéfini nécessite que celui-ci ait été sauvé dans /src/Service/ afin que le conteneur le trouve
- visuellement, une addition :

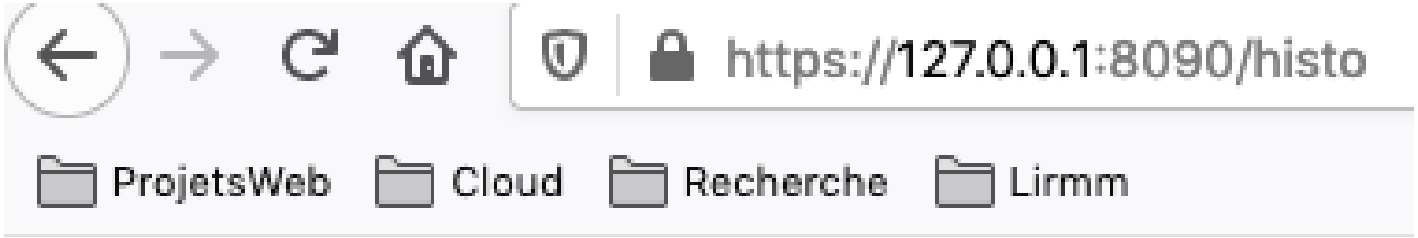
Création d'un service spécialisé IV

- la seconde route pour afficher l'historique sauvegardé en session :

```php
<?php #[Route("/histo/")]
function histo(HistoOp $histoop) : Response { // injection
  $r="";
  foreach($histoop->getOps() as $op){
    $r.= $op."<br>";
  }
  return new Response($r);
}
```

Création d'un service spécialisé V

- visuellement, l'historique :



45+55=100
4+8=12
23+57=80

- on peut donc injecter un service prédéfini ou personnalisé) dans une méthode de route ou dans un constructeur de service personnalisé en le sauvegardant dans un attribut

Plan

8 Environnement du projet

Modes de Projet et environnement I

- Un projet peut être exécuté dans différents modes :
  - dev : développement en règle générale sur une machine locale (serveur HTTP et MySQL (XAMP))
  - test : on teste les fonctionalités et la robustesse du projet
  - prod : en production, différents serveurs seront localisés à différents endroits et le projet doit être optimisé afin d'être rapide à l'exécution
- Symfony choisit de paramétrer l'environnement du projet (dont le mode d'exécution) grâce à des variables d'environnement définies dans un fichier caché `/.env` telles que : `APP_ENV=dev`
- Ce fichier `/.env` s'enrichit au cours des installations de bundles tels que doctrine :
  `DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name`

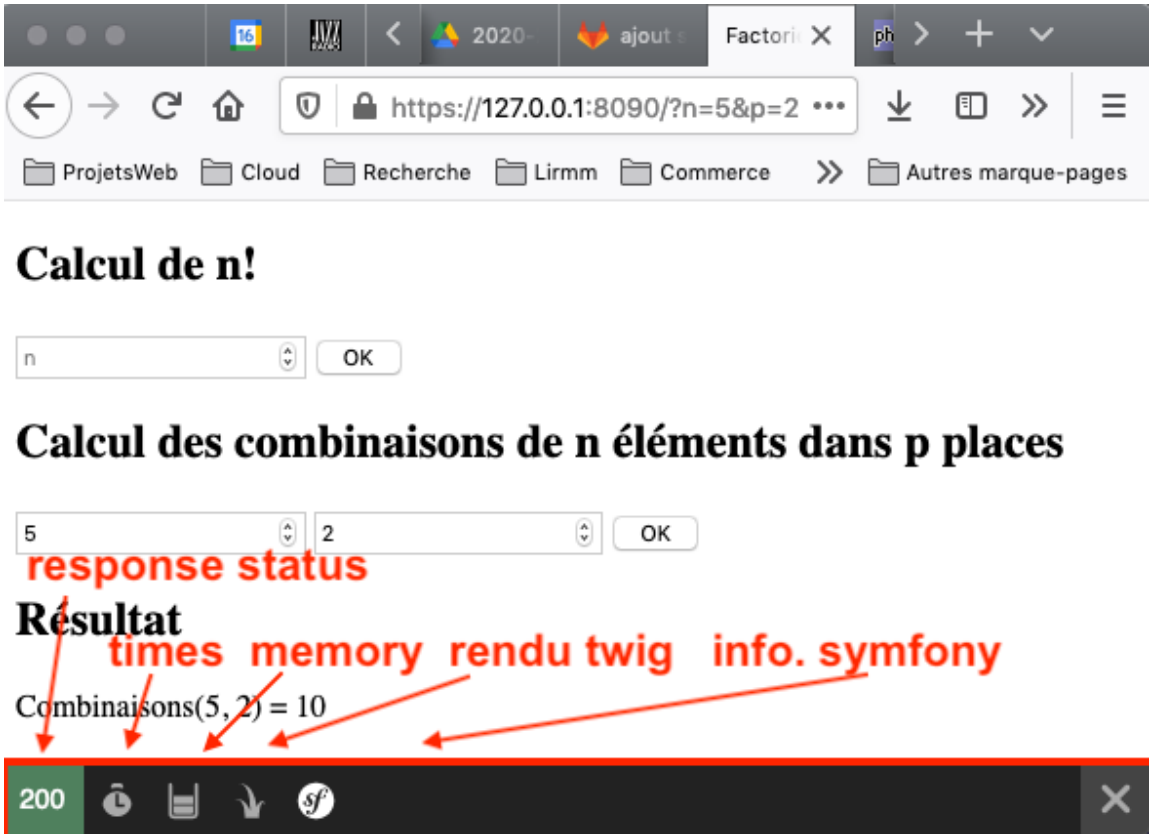
Plan

9 Profilage et Débogage

Profilage I

- Afin de gérer les performances du code, on peut utiliser le bundle `web_profiler` qui insère une barre horizontale noire en bas des pages Webs (twig) fournissant différentes informations de temps et d'espace mémoire consommé
- ce bundle est installé automatiquement avec twig et n'est visible qu'en mode dev et test !
- Son affichage peut-être basculé en cliquant sur l'extrémité droite de la barre :

Profilage II



- un autre bundle plus complet peut-être utilisé quand il n'y a pas de rendu HTML : profiler : `composer require -dev symfony/profiler-pack`

Débogage avec XDebug I

Le débogage depuis VSCode(ium) via l'extension PHP Xdebug est possible comme dans toute application PHP. Pour cela, il faut :

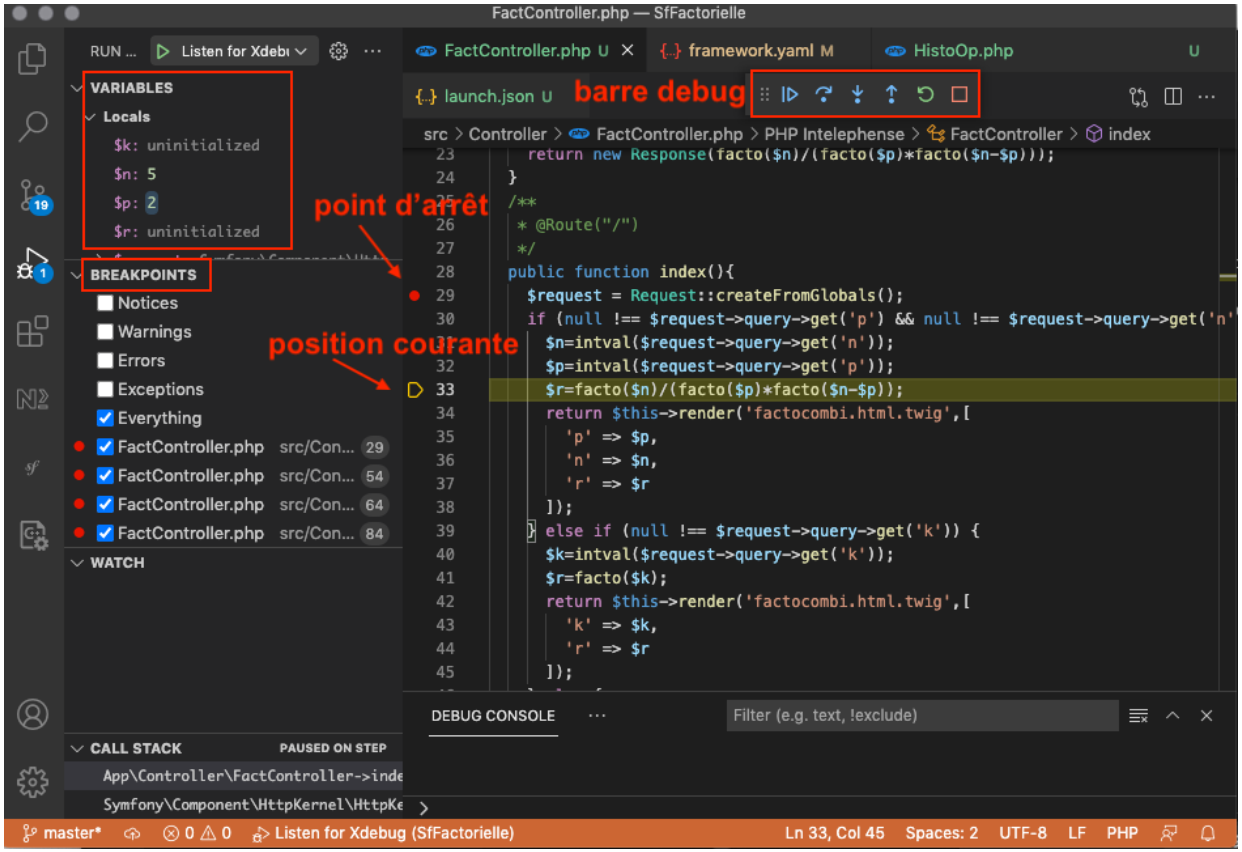
- installer l'extension `Xdebug helper` dans votre navigateur préféré (petit insecte grisé ou vert si actif)
- dans VSCode, passer dans la vue `Run and Debug`
- créer un fichier de config de débogage `launch.json` contenant :

```
"configurations": [
  {
    "name": "Listen for Xdebug",
    "type": "php",
    "request": "launch",
    "port": 9000
  }, ...
```

- en haut à gauche de la vue de Debug, cliquer sur le triangle vert pour lancer le serveur effectuant `Listen for Xdebug`

Débogage avec XDebug II

- dans le navigateur, effectuer une requête sur le contrôleur PHP où vous aurez préalablement posé un point d'arrêt
- une barre de débogage apparaît dans VSCode . . .



Plan

10 Doctrine

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Introduction I

- Doctrine 2 est un ORM depuis PHP 5.4+
- Il est possible de l'utiliser avec les frameworks PHP Symfony, Zend Framework, ...
- Une correspondance objet-relationnel (en anglais object-relational mapper ou ORM) est un patron de conception (design pattern) qui permet de manipuler une base de données relationnelle en utilisant des objets PHP persistants
- Ainsi le code PHP est plus homogène, ce qui facilite sa programmation et son débogage
- Doctrine 2 est une refondation complète de Doctrine 1 et utilise un grand nombre de patron de conception (Design Pattern)
- La manipulation des objets PHP (persistance, consultation) est réalisée grâce à un Gestionnaire d'Entités (EntityManager) qui implémente le "data mapper pattern"

Introduction II

- Une **entité** est un object PHP identifié par un attribut unique qui est lié à la clé primaire de la ligne qui lui correspond
- Une classe d'entités possède des entités, chaque entité correspondant à une ligne d'une table relationnelle
- génération de la BD à partir de classes d'entités métiers ou construction du modèle objet à partir d'une BD existante (rétro-ingéniérie) ;
- pas de schéma XML intermédiaire entre modèle objet et BD (Propel)
- DQL (Doctrine Query Language) inspiré par Hibernate Query Language (ORM Java)
- les entités sont liées entre elles par des associations : une association matérialise sur les objets PHP une contrainte d'intégrité référentielle de la BD ;
- des transactions ACID sont présentes

Introduction III

- l'actualisation des associations (références à d'autre(s) entité(s)) dans les entités est réalisée de manière **fainéante** (lazy) ; c'est-à-dire au fur et à mesure des accès ;
- Plusieurs démarches de conception de projet existent :
  - "code first" consiste à suivre la procédure suivante consistant à écrire le code PHP des entités et leurs métadonnées puis à créer la BD ;
  - "database first" permet de construire automatiquement le modèle objet PHP à partir d'une BD existante (rétro-ingéniérie) ;
  - "model first" devrait permettre de tout construire (entités et BD) à partir d'un modèle UML ...

  Actuellement, seule la démarche "code first" est totalement fonctionnelle. La création programmée d'entités à partir d'une BD existante nécessite de rajouter par la suite certaines associations entre entités oubliées par l'outil.

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Installation de doctrine dans Symfony I

Le paquet qui contient Doctrine s'installe en utilisant `Composer` :

```
Sf6Cours$ composer require symfony/orm-pack
```

Un certain nombre de paquets sont alors installés et configurés par défaut. Une fois Doctrine installé, on peut configurer l'accès au SGBD dans l'`env` du projet en choisissant le driver mysql, l'utilisateur tempo et son mot de passe identique, la base de données `Sf6Cours`, l'hôte localhost et le port 8889 (par défaut 3306) :

```
# utilisation de MySQL (MariaDB) localement
DATABASE_URL="mysql://tempo:tempo@127.0.0.1:8889/Sf6Cours?
↪  serverVersion=5.7"
```

Il ne reste plus qu'à créer la BD en lançant la commande :

```
Sf6Cours$ php bin/console doctrine:database:create
Created database `Sf6Cours` for connection named default
```

De nombreuses commandes doctrine sont disponibles :

Installation de doctrine dans Symfony II

```
Sf6Cours$ php bin/console list doctrine
...
Sf6Cours$ php bin/console doctrine:query:sql "create table essai
↪  (nom varchar(10));"
Sf6Cours$ php bin/console doctrine:query:sql "select * from
↪  essai"
/Applications/MAMP/htdocs/Sf6Cours/vendor/doctrine/dbal/lib/Doctrine
array (size=0)
  empty
```

Dans cet exemple, on a créé une table monocolonne et affiché son contenu (vide)

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Classe d'entité et métadonnées I

Une entité étant une instance PHP persistante, il faut commencer par définir :

- la classe d'entité en PHP : ensemble d'attributs protégés ayant chacun une paire d'accesseur/mutateur (get/set) et un attribut `id` qui correspond à la clé primaire et qui lui ne possède qu'un accesseur (dans le répertoire `src/Entity/`
- chaque classe peut posséder des méthodes **métiers**
- les **métadonnées** de la classe qui représentent la liaison entre classe et table et entre attribut et colonne
- Ces métadonnées peuvent être fournies en XML, en YAML (XML indenté) dans un fichier externe ou **en PHP grâce à des annotations ou des attributs** (`#[Entity ...`) dans le fichier définissant la classe d'entité

Classe d'entité et métadonnées II

- Depuis Symfony 6, il est recommandé d'utiliser les attributs natifs PHP

Une ligne de commande utilisant le bundle maker (maker-bundle $>$ 1.30.1) permet de créer une entité **interactivement** :

```
Sf6Cours$ bin/console make:entity
Class name of the entity to create or update :
> Evenement

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> nom

Field type (enter ? to see all types) [string]:
> string
```

Classe d'entité et métadonnées III

```
Field length [255]:
> 30

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Evenement.php

Add another property? Enter the property name (or press <return>
↪  to stop adding fields):
> dateDeb
...
```

Grâce à cette suite de questions réponses, on construit son entité itérativement en ajoutant des propriétés (attributs) à la classe `Evenement` :

Classe d'entité et métadonnées IV

```php
<?php // src/Entity/Evenement.php
namespace App\Entity;

use App\Repository\EvenementRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass=EvenementRepository::class)]
class Evenement {

#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column(type="integer")]
private $id;

#[ORM\Column(type="string", length=30)]

private $nom;
```

Classe d'entité et métadonnées V

```php
#[ORM\Column(type="datetime")]

private $dateDeb;


#[ORM\Column(type="datetime")]

private $dateFin;

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

public function getNom(): ?string {
  return $this->nom;
}
```

Classe d'entité et métadonnées VI

```php
public function setNom(string $nom): self {
  $this->nom = $nom;
  return $this;
}

public function getDateDeb(): ?\DateTimeInterface {
return $this->dateDeb;
}
...
```

- Ce code généré est bien entendu modifiable à votre convenance
- Dans les classes d'entités, on écrira **toutes les méthodes métiers** de l'entité en PHP objet sans s'intéresser à la persistance de ces objets
- La **persistance sera gérée dans les contrôleurs**

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Génération du schéma de BD et migration I

Afin de générer le schéma de BD en SQL (CREATE TABLE ...) et de l'exécuter, il faut utiliser deux commandes `doctrine` successives depuis un terminal :

- `Sf6Cours$ php bin/console make:migration`

  permet de construire un script PHP de **migration** en évaluant l'ensemble des entités définies et leur différence avec l'état actuel de la BD. Dans notre cas :

```
CREATE TABLE evenement (id INT AUTO_INCREMENT NOT NULL, nom
↪  VARCHAR(30) NOT NULL, date_deb DATETIME NULL ...
```

- `Sf6Cours$ php bin/console doctrine:migrations:migrate`

  effectue la migration depuis les entités vers la BD. On peut vérifier la création de la table `evenement` !

Bien entendu, ce processus est incrémental : si on modifie ou crée une entité, il faudra effectuer à nouveau ces deux étapes !

Associations entre entités I

Afin de représenter les liens qui unissent des entités, on utilise des associations (manyToOne, OneToOne, ...) entre entités qui correspondent à des contraintes d'intégrité référentielles (clés étrangères) du modèle relationnel.
Créons une entité Tournoi qui appartient à l'Entité Evenement : lors d'un événement sportif (Rolland Garros), il y a plusieurs catégories de tournois (simple messieurs, double dames, ...)
Là encore le bundle maker nous permet d'ajouter des propriétés de type `relation` afin d'indiquer le type d'association :

Associations entre entités II

```
Sf6Cours$ bin/console make:entity

Class name of the entity to create or update :
> Tournoi

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> ev

Field type (enter ? to see all types) [string]:
> relation

What class should this entity be related to?:
> Evenement

What type of relationship is this?
```

Associations entre entités III

```
-------------------------------------------------------------
Type         Description
-------------------------------------------------------------
ManyToOne    Each Tournoi relates to (has) one Evenement.
             Each Evenement can relate to (can have) many Tournoi

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne
...
```

Après migration, On s'aperçoit que la contrainte d'intégrité a été ajoutée :

```
ALTER TABLE tournoi ADD CONSTRAINT FK_18AFD9DF40A4EC42 FOREIGN
↪  KEY (ev_id) REFERENCES evenement (id)
```

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Le gestionnaire d'entité et la persistance I

Pour résumer :

- nous avons défini 2 entités Evenement et Tournoi qui sont deux classes PHP
- nous les avons reliées par une association manyToOne (plusieurs tournois font partie d'un événement)
- nous les avons associées grâce à des métadonnées définies par attributs PHP à une BD MySQL que nous avons créée puis modifiée en ligne de commande (migrations successives)
- il nous reste maintenant à instancier des entités et les rendre persistantes grâce au gestionnaire d'entité de Doctrine (EntityManager)
- On peut utiliser plusieurs EntityManager dans un projet Symfony et dans ce cas on utilisera le service `ManagerRegistry`

Le gestionnaire d'entité et la persistance II

- Mais dans les cas simples, pour l'obtenir depuis un contrôleur, il suffit d'injecter la dépendance sur l'interface `EntityManagerInterface` pour obtenit ce service
- nous allons créer une route dans le contrôleur TournoiControleur qui va permettre d'insérer dans la BD un nouvel événement saisi comme paramètre de route :

```php
<?php use Doctrine\ORM\EntityManagerInterface;
class TournoiController extends AbstractController {
#[Route("/tournoi/newEvt/{nom<[0-9a-zA-Z ]+>}",
↪  name="newevt")]
public function newevt($nom, EntityManagerInterface
↪  $entityManager): Response {
  $ev=new Evenement(); // constructeur par défaut tjrs
  $ev->setNom($nom);
  $entityManager->persist($ev); // en tampon
  $entityManager->flush(); // en BD
```

Le gestionnaire d'entité et la persistance III

```
        return new Response("Événement '$nom' créé avec l'id :
        ↪  ".$ev->getId().' !');
    }
```

- remarquons que l'id est généré automatiquement par MySQL au moment du flush()

Cet objet PHP `$entityManager` nous permet d'enregistrer les entités devant persister en deux phases :

- `$entityManager->persist($product);` permet de tamponner l'entité `product` dans le gestionnaire d'entités

Le gestionnaire d'entité et la persistance IV

- cette méthode `persist` sera utilisée plusieurs fois avant de lancer la transaction de BD qui effectuera l'ensemble des requêtes (insert ou update ou delete).
- `$entityManager->flush();` lance la transaction sur la BD
- Remarquons que toute entité enregistrée (tamponnée) par `persist()` peut être modifiée par la suite, l'état final de l'entité sera enregistré lors du `flush()`

Création d'entité associée à une autre I

On souhaite ajouter un nouveau tournoi dans un événement existant :

- on va créer une nouvelle route dans notre contrôleur `newTnoi`
- il faut la paramétrer avec l'identifiant de l'événement, le nom et la description du tournoi
- avant d'attacher le tournoi à un événement, il faut récupérer l'entité Evenement depuis la BD : on utilise la méthode d'EntityManger `find(string $classe, int $id)`

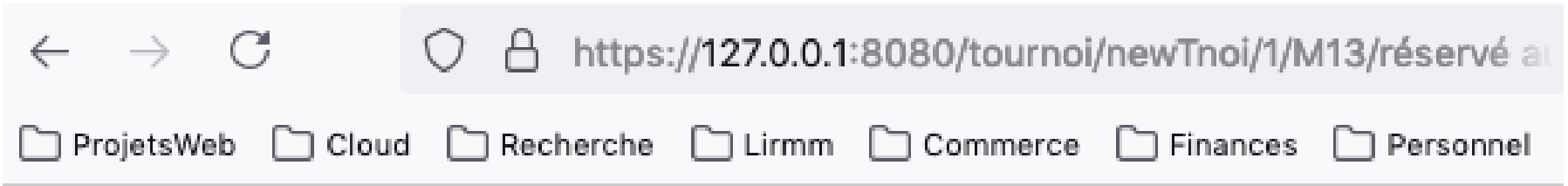
```
<?php #[Route("/tournoi/newTnoi/{evtid<[0-9]+>}/{nom<[0-9a-zA-Z
↪  ]+>}/{desc?}", name="newTnoi")]
public function newTnoi($evtid, $nom, $desc,
↪  EntityManagerInterface $em): Response {
  $tnoi=new Tournoi(); // constructeur par défaut tjrs
  $tnoi->setNom($nom);
  $tnoi->setDescription($desc);
  $evt = $em->find("App\Entity\Evenement",(int)$evtid);
```

Création d'entité associée à une autre II

```
    // remarque: on peut utiliser 'App:' qui est un alias de
    ↪  'App\Entity\'
    if($evt === null){
      return new Response("L'événement $evtid n'existe pas ! Le
      ↪  tournoi n'a pas été créé !");
    } else {
      $tnoi->setEv($evt);
      $em->persist($tnoi); // en tampon
      $em->flush();
      return new Response("Le tournoi {$tnoi->getNom()} a été
      ↪  enregistré dans l'événement {$evt->getNom()} !");
    }
}
```

Création d'entité associée à une autre III

Une exécution :



Les méthodes de l'entityManager I

- pour supprimer une entité de la BD : `$em->remove($tnoi);`. A faire suivre d'un `flush()` afin d'effectuer la requête SQL `DELETE`
- pour mettre à jour une entité :
  - si celle-ci a été chargée (`find`) depuis la BD, elles est gérée par Doctrine :**déjà dans le tampon de persistence**. Il ne reste plus qu'à flusher !
  - sinon (`new`), persist puis flusher
- pour sortir une entité du tampon de persistence, il faut utiliser la méthode `detach()` : `$em->detach($etud)`
- pour retrouver (R de CRUD) une ou plusieurs entités, plusieurs méthodes (find, findAll, findBy, DQL) pourront être utilisées sur le dépôt d'entité (Repository) associé :
  `$em->getRepository(Product::class)->findAll()`

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

EntityRepository (dépôt d'entités) I

- pour manipuler l'ensemble des entités d'une classe `Evenement`, on utilise un dépôt (repository) récupéré depuis l'entityManager
- puis, on lui applique une requête `findAll` ou autre
- on crèe une nouvelle vue Twig afin de visualiser l'ensemble des événements et de leurs tournois créés grâce aux routes précédentes (newEvt, newTnoi)

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

{% block title %}Liste des tournois{% endblock %}
{# définition du bloc title utilisé par le parent base #}

{% block body %} {# définition du bloc body utilisé parent #}
  <h1>Liste des événements et des tournois</h1>
  <ul>
  {% for e in evts %}
```

EntityRepository (dépôt d'entités) II

```
    <li> <h2>{{ e.nom }}</h2> {# utilise getNom() #}
    <ul>
    {% for t in e.tournois %}
    <li> <em>{{ t.nom }}</em> ( {{ t.description }} )
    {% endfor %}
    </ul>
  {% endfor %}
  </ul>
{% endblock %}
```

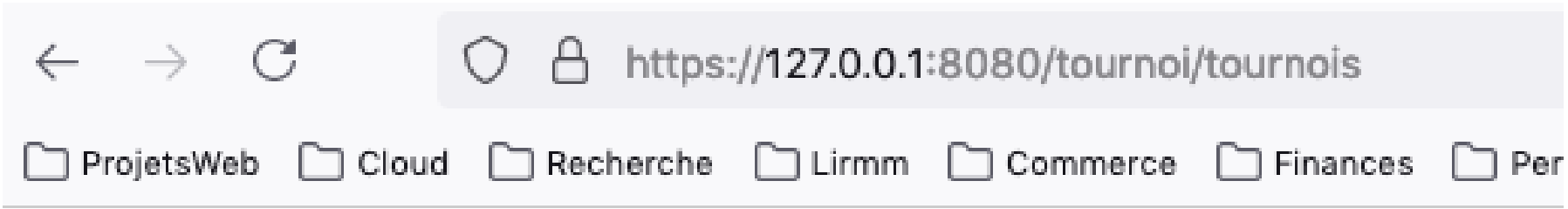
Cette vue est utilisé par le contrôleur `tournois` suivant :

EntityRepository (dépôt d'entités) III

```php
<?php #[Route("/tournoi/tournois",name="tournois")]
/* liste des évts et des tournois */
public function tournois(EntityManagerInterface $entityManager):
↪  Response {
  $evts =
  ↪  $entityManager->getRepository("App\Entity\Evenement")->findAl
  return $this->render('tournoi/tournois.html.twig', [
    'evts' => $evts,
  ]);
}
```

EntityRepository (dépôt d'entités) IV

le rendu graphique :



Liste des événements et des tournois

- **challenge 18 Juillet 21**
  - *M13* (réservé aux moins de 13 ans)
  - *MasterM* (Master Masculin)
  - *MasterF* (Master Féminin)
  - *Loisir* (Tournoi Loisir ouvert à tous)
- **challenge 15 Aout 21**

Autres consultation d'entités I

On a vu deux méthodes de consultation (query) :

- `$em->find(string $classe, int $id)` récupère une entité par son Id
- `$repo->findAll()` qui récupère toutes les entités (la table)

mais il en existe plein d'autres dont les quelques suivantes :

- `$repo->findBy(array('nom'=>'Dupont','prenom'=>'Paul'))` (conjonctions d'égalité). On peut utiliser `findBy()` ou `findOneBy()` sur un dépôt en fournissant un tableau de (clé, valeur) ou chaque clé est un nom d'attribut et chaque valeur l'unique valeur d'attribut accepté.
- La méthode `findBy()` permet également :
  - de fournir un ensemble de valeurs possibles pour un attribut (id IN (1,2,3)) en indiquant un tableau de valeur (array()) ;
  - de trier selon une colonne (ORDER BY) (2nd paramètre) ;

Autres consultation d'entités II

- de limiter le nb de résultats (LIMIT x) (3eme paramètre) ;
- de définir un offset (résultats sauf les n premiers) (4eme paramètre) ;

- L'exemple suivant parcourt tous les produits dont le nom est 'un' ou 'deux' ou 'trois' en ordre décroissant sur les noms, croissant sur les id, en en prenant que 10 au maximum et en ne prenant pas les 2 premiers !

```php
<?php ...
$ps=$entityManager->getRepository('Product')->findBy(
  array('name' => array("un","deux", "trois")), // IN
  array('name' => 'DESC', 'id' => 'ASC'),       // ORDER BY
  10,               // LIMIT
  2                 // OFFSET
);
if (count($ps)){                    // plusieurs produits
  foreach ($ps as $p) {
    ...
```

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Doctrine Query Language (DQL) I

Le langage DQL ressemble au SQL mais n'est pas du SQL ! Il permet de créer une requête complexe (createQuery()) puis de l'exécuter (getResult()) en récupérant un tableau d'entités.

```php
<?php                    // dql.php
...
$q=$entityManager->createQuery("select p from Product p where
  p.id>8 or p.name in ('un', 'deux', 'trois') order by
  p.name asc");
$ps=$q->getResult();
if (count($ps)){                    // plusieurs produits
  foreach ($ps as $p) {
    echo sprintf("%d %s\n",$p->getId(), $p->getName());
  }
} else {
  echo "Aucun Produit !\n";
}
```

Doctrine Query Language (DQL) II

On remarquera que l'on récupère une collection (sorte de tableau) d'entités selon des conditions diverses puis on la parcourt.

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Associations entre entités I

- Les associations entre entités permettent de modéliser les relations existant entre les objets métiers
- Elles sont, pour certaines, liées à des contraintes d'intégrité référentielles dans la BD mais **pas toujours**
- e.g. l'héritage entre classe d'entités ne donnera pas forcément lieu à une relation dans la BD
- certaines BD n'implémentent pas certaines contraintes d'ntégrité

Quelques règles :

- une association unidirectionnelle ne possède qu'un côté propriétaire (owning side)
- une association bidirectionnelle possède un côté propriétaire (owning side) et un côté inverse

Associations entre entités II

- l'entité du côté inverse utilise l'attribut `mappedBy` pour désigner l'attribut de l'entité propriétaire
- l'entité du côté propriétaire utilise l'attribut `inversedBy` pour désigner l'attribut de l'entité inverse
- seuls les changements (ajout, suppression) côté propriétaire seront pris en compte par `doctrine` lors du `flush` : effectuer les changements des deux côtés ou bien seulement du côté propriétaire

Association unidirectionnelle "one to one" I

Cas d'utilisation : un étudiant est un utilisateur, certains utilisateurs ne sont pas étudiant

code PHP annoté (méta-données) suivi du code SQL associé :

```php
<?php /** @file Etudiant.php */
#[Entity **/
class Etudiant{
...
/**
#[OneToOne(targetEntity="User")
#[JoinColumn(name="user_id", referencedColumnName="id")
**/
private $user;
...
```

Association unidirectionnelle "one to one" II

```
CREATE TABLE Etudiant (        # SQL
...,
user_id INT DEFAULT NULL,
UNIQUE INDEX UNIQ_6FBC94267FE4B2B (user_id),
PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE User (
id INT AUTO_INCREMENT NOT NULL,
PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Etudiant ADD FOREIGN KEY (user_id) REFERENCES
User(id);
```

Remarquons l'index **unique** créé sur la colonne `user_id`

Association bidirectionnelle "one to one" I

- Cas d'utilisation : un caddy appartient à un unique client qui ne peut en avoir qu'un au maximum
- On veut conserver du côté de l'entité client une référence au caddy et réciproquement (bidirectionnel)
- ci-dessous le code PHP annoté (méta-données) ainsi que le code SQL associé :

```php
<?php /** @file Customer.php */
#[Entity]
class Customer{
// ...
#[OneToOne(targetEntity="Cart", mappedBy="customer")]

  private $cart;
}
#[Entity]
```

Association bidirectionnelle "one to one" II

```
class Cart{
// ...
#[OneToOne(targetEntity="Customer", inversedBy="cart")]
#[JoinColumn(name="customer_id", referencedColumnName="id")]

  private $customer;
}
...

CREATE TABLE Cart (                    # SQL
  id INT AUTO_INCREMENT NOT NULL,
  customer_id INT DEFAULT NULL,
  PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Customer (
  id INT AUTO_INCREMENT NOT NULL,
  PRIMARY KEY(id)
```

Association bidirectionnelle "one to one" III

```
) ENGINE = InnoDB;
ALTER TABLE Cart ADD FOREIGN KEY (customer_id) REFERENCES
  Customer(id);
```

- la table propriétaire de clé étrangère (`customer_id`) est le caddy
- Au niveau des entités, elles sont symétriques en possédant chacune une référence à l'autre
- Au niveau des méta-données, c'est l'attribut inversedBy qui désigne le propriétaire de clé étrangère tandis que le référencé possède un attribut "mappedBy"
- L'entité propriétaire est importante car au moment du **flush, la sauvegarde en BD se basera sur les attributs des propriétaires**
- Dans le cas des associations bidirectionnelles (A(b) - B(a)), la métadonnée `mappedBy` associée à b indique l'attribut de l'autre classe d'entité associée qui est source de l'association inverse (a)

Association bidirectionnelle "one to one" IV

- Réciproquement, la métadonnée `inversedBy` associée à a indique l'attribut (b) de l'autre classe d'entité associée qui est source de l'association inverse
- Comme dans le modèle relationnel de la BD, une seule contrainte d'intégrité référentielle existe (`Bt(a_id)->At(id)`), c'est dans les métadonnées de l'entité inversedBy (B) qu'on implantera la liaison avec la BD grâce à la métadonnées JoinColumn(name) qui désigne la colonne clé étrangère de la table (`a_id`) et JoinColumn(referencedColumnName) qui désigne la colonne cible de la référence (id)

Association unidirectionnelle many-to-one I

Plusieurs utilisateurs (many) peuvent partager la même adresse. On indique ci-dessous le code PHP annoté (méta-données) ainsi que le code SQL associé :

```php
<?php /** @file User.php */
class User{
...

#[ManyToOne(targetEntity="Address")]
 #[JoinColumn(name="address_id", referencedColumnName="id")]

private $address;
}
...
```

Association unidirectionnelle many-to-one II

```
CREATE TABLE User (          # SQL
  ...,
  address_id INT DEFAULT NULL,
  PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Address (
  id INT AUTO_INCREMENT NOT NULL,
  PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE User ADD FOREIGN KEY (address_id) REFERENCES
  Address(id);
```

Remarquons que ce cas étant extrêmement fréquent, la ligne `@JoinColumn` n'est pas nécessaire si le nommage des attributs et des colonnes suit la convention. Remarquons encore que l'adresse est optionnelle (NULL).

Association bidirectionnelle one-to-many I

- Un produit possédant plusieurs caractéristiques aura une propriété oneToMany "features" qui sera forcément en correspondance (mappedBy) par une propriété "product" manyToOne dans l'entité Feature
- Remarquons que la contrainte d'intégrité réfrentielle sera dirigée depuis feature(product_id) vers product(id)
- A l'inverse, la propriété "features" de l'entité Product sera une `collection` de caractéristiques (`Doctrine\Common\Collections\Collection`)

Association bidirectionnelle one-to-many II

```php
<?php /** @file Product.php, Feature.php */
class Product{
  // ...
  /**
 #[OneToMany(targetEntity="Feature", mappedBy="product")]
  **/
  private $features;
  // ...
public function __construct() {
  $this->features = new ArrayCollection();
  ...
class Feature{
  // ...
  /**
 #[ManyToOne(targetEntity="Product", inversedBy="features")
 #[JoinColumn(name="product_id", referencedColumnName="id")
  **/
```

Association bidirectionnelle one-to-many III

```php
private $product;
// ...
```

Association unidirectionnelle many-to-many I

- Des utilisateurs sont affiliés à différents groupes d'utilisateurs
- Dans la BD, une table de jointure (non modélisée par une entité !) permettra de représenter les associations multiples de groupes et d'utilisateurs

```php
<?php /** @file app/Entity/User.php */
class User{
// ...
#[ManyToMany(targetEntity="Group")]
#[JoinTable(name="users_groups")]
#[joinColumns={@JoinColumn(name="user_id", referencedColumnName="id
#[inverseJoinColumns={@JoinColumn(name="group_id", referencedColumn

  private $groups;
// ...
public function __construct() {
```

Association unidirectionnelle many-to-many II

```
    $this->groups = new \Doctrine\Common\Collections\
        ArrayCollection();
}
...
class Group{
    // ...
}

CREATE TABLE User (                    # SQL
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE users_groups (
    user_id INT NOT NULL,
    group_id INT NOT NULL,
    PRIMARY KEY(user_id, group_id)
) ENGINE = InnoDB;
```

Association unidirectionnelle many-to-many III

```
CREATE TABLE Group (
  id INT AUTO_INCREMENT NOT NULL,
  PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE users_groups ADD FOREIGN KEY (user_id)
  REFERENCES User(id);
ALTER TABLE users_groups ADD FOREIGN KEY (group_id)
  REFERENCES Group(id);
```

Héritages I

Plusieurs héritages sont techniquement possibles grâce aux associations Doctrine :

- Mapped Superclass : permet de définir les attributs et associations communs dans une superclasse qui ne donnera pas lieu à création de table mais chaque entité fille donnera une table contenant les champs de sa superclasse
- Single Table Inheritance : dans ce patron, toute la hiérarchie est représentée dans une unique table qui contient un champ discriminant la classe. Les champs spécifiques à chaque sous-classe doivent absolument être NULL puisqu'ils existeront pour tout le monde !
- Class Table Inheritance : chaque classe fille correspondra à une table la représentant et autant de tables que nécessaire pour représenter ses ancêtres avec des clés étrangères les reliant ;

Héritages II

- Overrides : permet d'indiquer qu'un attribut ou une association de la classe fille prend le pas sur un attribut ou une association de la super-classe (valable dans le cadre de Mapped Superclass).

Remarquons encore que l'association OneToOne uni ou bi-directionnelle permet de modéliser une sorte d'héritage : cependant les attributs communs seront dupliqués dans chaque table ...

Plan

10 Doctrine
- Introduction
- Installation de doctrine dans Symfony
- Classe d'entité et métadonnées
- Génération du schéma de BD et la ligne de commande doctrine
- Le gestionnaire d'entité et la persistance
- Consultation des entités (query)
- Doctrine Query Language (DQL)
- Associations entre entités
- Autres possibilités de Doctrine

Rétro-ingéniérie : génération des Métadonnées I

Afin de fabriquer les métadonnées associées à une BD préexistante, on peut utiliser la commande doctrine afin de générer un fichier de métadonnées yml, xml ou même php (à éviter) :

```
$ vendor/bin/doctrine orm:convert-mapping --from-database yml .
```

Cela produira un fichier `Product.dcm.yml` dans le répertoire courant (.) dont le contenu suit :

```yaml
Products:
  type: entity
  table: products
  id:
    id:
      type: integer
      nullable: false
      unsigned: false
      id: true
```

Rétro-ingéniérie : génération des Métadonnées II

```
    generator:
    strategy: IDENTITY
  fields:
    name:
      type: string
      nullable: false
      length: 255
      fixed: false
  lifecycleCallbacks: {  }
```

Attention, cette technique ne permet pas de récupérer :

- les associations inverses (mapped by) ;
- les héritages ;
- les clés étrangères qui sont aussi clés primaires ;
- les cascades ;

Rétro-ingéniérie : génération des Métadonnées III

De plus, cette technique créera des métadonnées d'entités à partir des tables de jointures (users-groups) et ne définira que des associations ManyToOne (clé étrangères).
Par conséquent, il faudra ensuite reprendre les entités créées afin qu'elles correspondent aux fonctionalités souhaitées par l'application !

Rétro-ingéniérie : génération des entités I

Attention, le “convert-mapping” ne permet que de créer les métadonnées : afin de créer les fichiers php définissant les classes d'entités, il faut utiliser la commande “generate-entities” qui utilise les méta-données (quelque soit leur format (xml, yml, php)) et quelle que soit la façon dont elles ont été produites, manuellement ou automatiquement (convert-mapping) :

```
$ vendor/bin/doctrine orm:generate-entities src/
```

Cette commande va créer le fichier Product.php ou le modifier afin d'ajouter des attributs **privés** pour les références aux objets liés, accesseurs et mutateurs.
Encore une fois, il est préférable de créer les entités à la main si l'on veut spécifier de l'héritage entre classes ou définir des méthodes spécifiques métier.

Transactions ACID I

Afin de réaliser des transactions assurant la cohérence de la BD, Doctrine 2 fournit DBAL qui est la couche entre l'ORM et PDO. DBAL permet notamment de contrôler les limites d'une transaction à l'aide des méthodes `beginTransaction()`, `commit()`, `rollback()`. Par défaut, les requêtes sont effectuées en mode auto-commit.

```php
<?php          // transaction.php
// $em instanceof EntityManager
$em->getConnection()->beginTransaction(); // suspend auto-commit
try {
  $etud = new Etudiant('Master Info');
  $em->persist($etud);
  $user = new User('Michel');
  $em->persist($user);
  $etud->setUser($user);
  $em->flush();
  $em->getConnection()->commit();
```

Transactions ACID II

```
} catch (Exception $e) {
  $em->getConnection()->rollback();
  throw $e;
}
```

Remarquons que dans cet exemple, c'est le `beginTransaction()` qui supprime le mode auto-commit par défaut de PDO. Par conséquent, ensuite le `flush` ne suffira pas pour valider et il faudra le `commit`. Enfin, DBAL permet également de manipuler des verrous (lock) sur les tables.

Autres attributs Doctrine sur les entités I

Un grand nombre d'attributs existent dans la documentation Doctrine, dont celles-là :

- @Table et @Column contiennent de nombreuses options. Un attribut non persistant ne sera pas annoté par @Column
- trier une collection i.e. les événements d'un gestionnaire

```
/** entité User : hydrater les événements par ordre
↪  anti-chronologique */
#[ORM\OneToMany(targetEntity=Evenement::class,
↪  mappedBy="gestionnaire")]
#[ORM\OrderBy({"dateDeb" = "DESC"})]

private $evenements;
```

- @HasLifecycleCallback permet des méthodes hook (@Pre|@Post)(Persist|Remove|Update)

Autres attributs Doctrine sur les entités II

```
/** entité ayant des callbacks
#[Entity
#[HasLifecycleCallbacks
*/
class User{
#[PostPersist]
  public function createdAt() {$this->createdAt=time();}
...
```

Conclusion I

- L'utilisation d'un ORM tel que Doctrine ne peut être envisagé que dans le cadre d'un développement à plein temps d'un projet
- En effet, le nombre de concepts employés ainsi que le vocabulaire utilisé nécessite un long temps d'apprentissage
- Intégré dans Symfony, Doctrine 2 semble avoir pris le pas sur les autres ORMs PHP
- De plus, ses concepts proviennent d'ORM leaders dans d'autres langages (Hibernatus en Java) et son apprentissage restera utile

Plan

11 Formulaires Symfony

Formulaires Symfony I

Les formulaires Symfony ont les caractéristiques suivantes :

- bundle non indispensable à installer par :

  ```
  $ composer require symfony/form
  ```

- ils peuvent être facilement associés aux entités Doctrine afin de créer,mettre à jour, supprimer des entités
- leur utilisation demande 3 étapes :
  1. construction du formulaire soit dans un contrôleur, soit dans une classe dédiée
  2. rendu visuel du formulaire dans un template twig
  3. traitement des données (validation) et transformation en objets PHP éventuellement persistants (entités)
- Un grand nombre de `Type` de formulaire sont prédéfinis :
  - TextType pour les champs de type text
  - ChoiceType pour les champs de type select
  - UrlType pour la saisie d'URL

Formulaires Symfony II

  - FormType pour un ensemble de champs constituant un formulaire
  - PersonnaliseType pour un groupe de champs (adresse postale) ou extension d'un type préexistant (customisation)
- Tous ces Type héritent d'`AbstractType` qui est la classe de base
- **Attention, ces classes PHP modélisent des intermédiaires entre le formulaire HTML (Vue) et l'entité Doctrine (Modèle). Ils permettent de transférer des données du modèle vers la vue lorsqu'on affiche un formulaire HTML de mise à jour d'entité. Ils permettent surtout de transférer de la vue vers le modèle lors de la validation duformulaire HTML.**

Création d'un formulaire à partir d'un contrôleur I

AbstractController possède une méthode createFormBuilder() permettant de créer le formulaire à partir d'une entité :

```
<?php #[Route("/tournoi/saisieTnoi/{evtid<[0-9]+>}",
↪  name="saisieTnoi")]
public function saisieTnoi($evtid): Response {
  $tnoi=new Tournoi();
  $tnoi->setNom("");
  $tnoi->setDescription(""); // saisie donc vide
  $form = $this->createFormBuilder($tnoi)
    ->add('nom', TextType::class)
    ->add('description', TextType::class)
    ->add('sauver', SubmitType::class, ['label' => 'Créer le
    ↪  tournoi !'])
    ->getForm(); // le formulaire est créé
  return $this->render('tournoi/saisieTnoi.html.twig', [
    'form' => $form->createView()
```

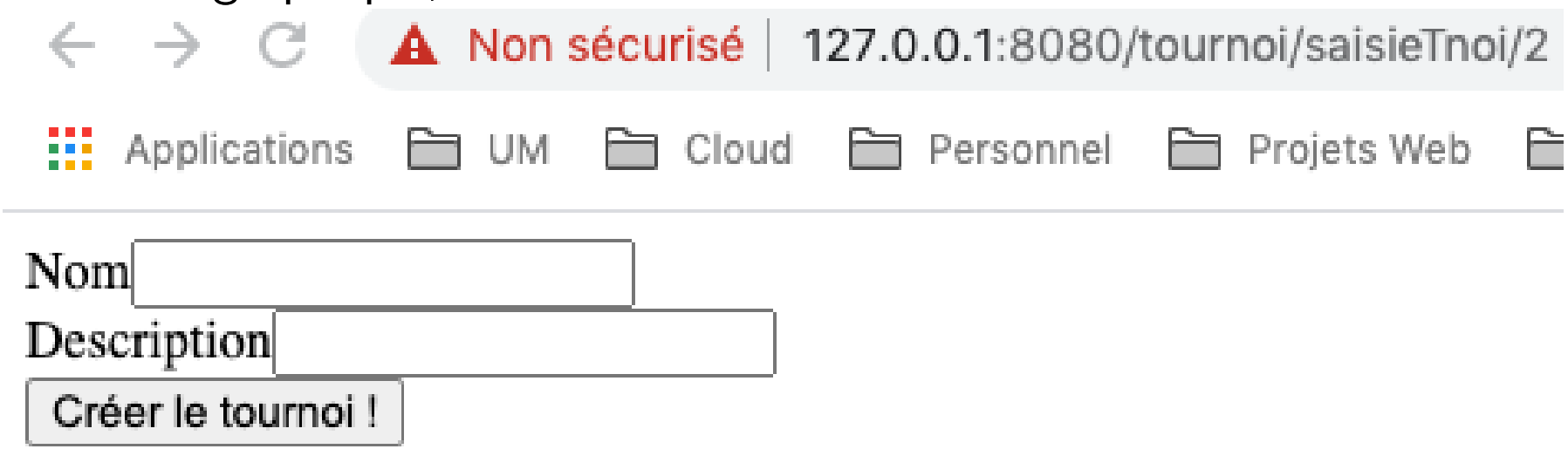
Création d'un formulaire à partir d'un contrôleur II

```
    ]);
}
```

Le template Twig est trivial et utilise la fonction Twig `form` sur la vue créée à partir du formulaire Symfony (createView) :

```
{# saisieTnoi.html.twig #}
{{ form(form) }}
```

Le rendu graphique, sans CSS :



Création d'un formulaire à partir d'un contrôleur III

Dans cet exemple simple :

- le traitement de la soumission n'est pas effectué
- la validation de la saisie n'est pas réalisée
- l'id de l'événement est un paramètre de route, il serait préférable qu'il soit un `select` !

On pourrait ajouter ce code en augmentant la taille de la fonction de contrôleur `saisieTnoi` mais ce n'est pas une bonne pratique (design pattern contrôleur doit être simple et **la logique métier doit être dans les objets métiers**) : on va créer une classe de formulaire pour un tournoi qui pourra être réutilisé indépendamment du contrôleur !

Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Création d'une classe de formulaire I

- on crée un nouveau Type (TournoiType) de formulaire
- cette classe hérite de `AbstractType` qui est une classe de base implémentant l'interface `FormTypeInterface`
- elle doit posséder une méthode `buildForm` qui possède un paramètre `FormBuilder` auto-cablé auquel on va ajouter des champs de formulaire

```php
<?php // src/Form/TournoiType.php
namespace App\Form;

use Symfony\Component\Form\AbstractType;
...
class TournoiType extends AbstractType{
  public function buildForm(FormBuilderInterface $builder, array
  ↪  $options):void{
    $builder
```

Création d'une classe de formulaire II

```
        ->add('nom', TextType::class)
        ->add('description', TextType::class)
        ->add('sauver', SubmitType::class,
            ['label' => 'Créer le tournoi !'])
    ;
  }
}
```

Dans le contrôleur, on définit une nouvelle route `saisirTournoi` :

```
<?php #[Route("/tournoi/saisirTnoi", name="saisirTnoi")]
/* AVEC type de formulaire */
public function saisirTnoi(): Response {
  $tnoi=new Tournoi();
  $form = $this->createForm(TournoiType::class, $tnoi);
  return $this->render('tournoi/saisieTnoi.html.twig', [
    'form' => $form->createView()
```

Création d'une classe de formulaire III

```
    ]);
}
```

- Remarquons qu'on a conservé le même fichier de rendu Twig !
- la méthode `createForm` utilise un premier argument qui est la classe de TournoiType et un second qui est une entité contenant les données
- quand on va emboîter des formulaires, il est préférable que chaque Type de formulaire connaisse l'entité qui lui correspond. Pour cela, on ajoute une méthode configureOptions au type :

```php
<?php
public function configureOptions(OptionsResolver $resolver):
↪  void{
  $resolver->setDefaults([
    'data_class' => Tournoi::class,
  ]);
}
```

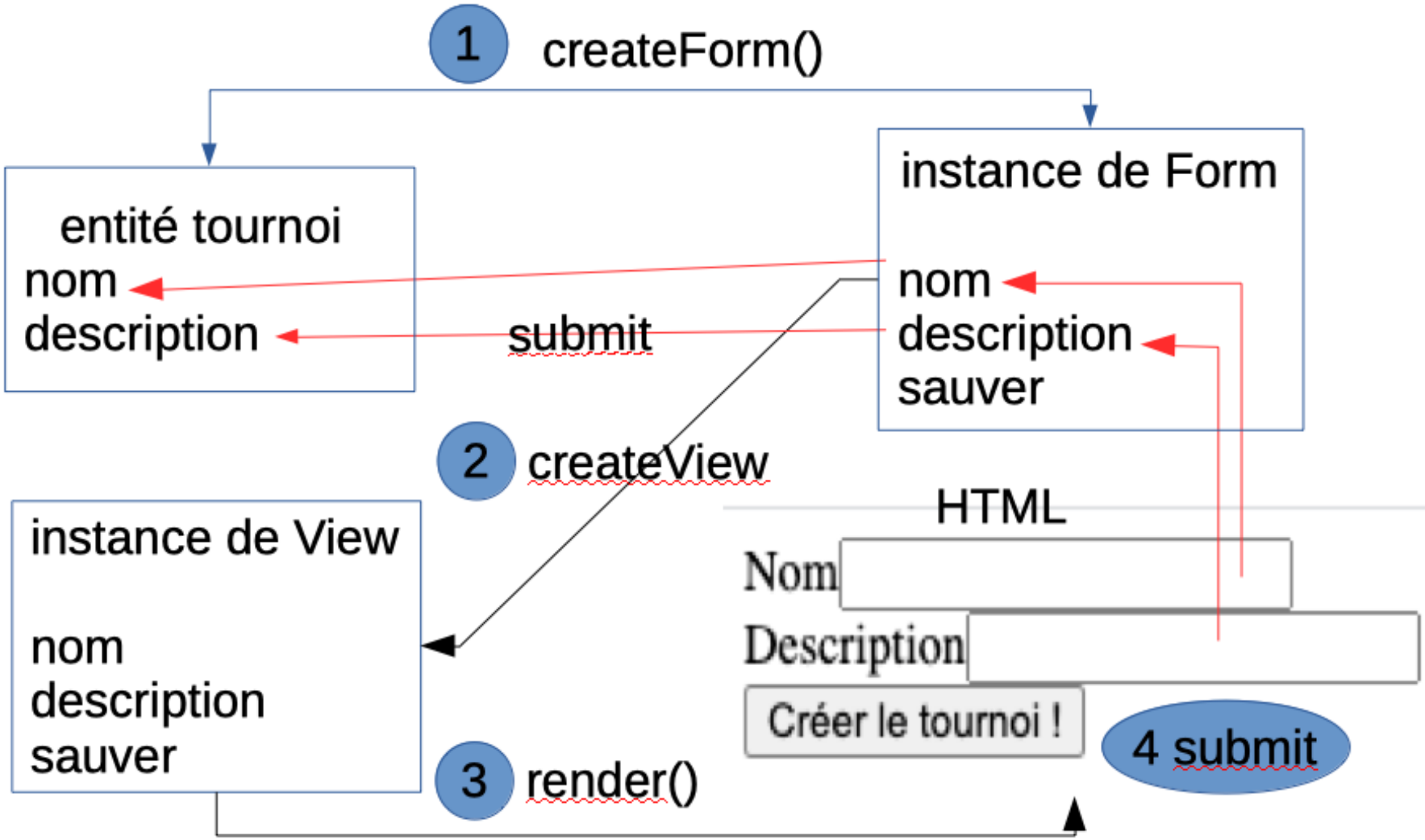
Création d'une classe de formulaire IV

- le `resolver` pourra ainsi connaître l'entité correspondante

Remarquons que tout ce processus de création de formulaire est simplifié, comme souvant dans Sf, grâce au bundle `maker` qui offre une interface CLI :

```
Sf6Cours$ php bin/console make:form
The name of the form class (e.g. VictoriousElephantType):
> EvenementType

The name of Entity or fully qualified model class name that the
↪  new form will be bound to (empty for none):
> Evenement

created: src/Form/EvenementType.php
  Success!

Next: Add fields to your form and start using it.
```

Dynamique des formulaires et de leur soumission I



Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Gestion des associations entre entités I

Afin de créer un tournoi, il nous faut lui associer un événement existant. Ceci peut se faire de différentes manières plus ou moins complexes (voir embed form) ! la façon la plus simple est la suivante :

- Editer le TournoiType.php et ajouter à la construction un champ de type EvenementType (aggrégation de champs de formulaire défini par le développeur) :

```
<?php public function buildForm(FormBuilderInterface
↪  $builder, array $options):void{
  $builder
    ->add('ev', EvenementType::class) // ICI
    ->add('nom', TextType::class)
    ->add('description', TextType::class)
    ->add('sauver', SubmitType::class,
        ['label' => 'Créer le tournoi !'])
  ;
}
```

Gestion des associations entre entités II

- on obtient graphiquement :

Ev
Nom monev
Date deb
Jan 2 2022
08 : 30
Date fin
Jan 3 2022
17 : 30
Nom M6
Description destiné à ...
Créer le tournoi !

Gestion des associations entre entités III

- Lors de la soumission, une erreur apparaît car le contrôleur qu'on n'a pas modifié tente de rendre persistant le tournoi alors que l'événement qu'on crée simultanément n'existe pas dans la BD (pas de clé étrangère `ev_id`) !
- pour la résoudre, on ajoute une règle de *cascade sur l'association reliant tournoi à événement afin que l'événement soit rendu persistant avant le tournoi qui en dépend (dans l'entité Tournoi.php)*

```php
<?php /**
 #[ORM\ManyToOne(targetEntity=Evenement::class,
↪   inversedBy="tournois", cascade={"persist"})]
* ...
*/
private $ev;
```

Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Validation des formulaires I

Les règles de validation seront des attributs PHP `Assert` qui seront vérifiées, côté serveur, avant le traitement de la requête

- Avant tout, il faut **installer le composant Validator** :

  ```
  composer require symfony/validator
  ```

- il faut ensuite vérifier dans config/packages/frameyork.yaml que les attributs PHP définissant les règles de validation sont activées :

  ```yaml
  framework:
    secret: '%env(APP_SECRET)%'
    validation: { enable_annotations: true }
  ```

- ensuite, on modifie la propriété nom de l'entité Tournoi pour la rendre obligatoire :

  ```php
  <?php #[ORM\Column(type="string", length=30)
  #[Assert\NotBlank]
  private $nom;
  ```

Validation des formulaires II

- De même, on veut également que l'événement associé soit valide :

```php
<?php /**
#[ORM\ManyToOne(targetEntity=Evenement::class,
↪  inversedBy="tournois", cascade={"persist"})]
#[ORM\JoinColumn(nullable=false)]
#[Assert\Type(type="App\Entity\Evenement")]
 #[Assert\Valid]
private $ev;
```

- Un grand nombre de **contraintes** de validation existent ...
- Ces contraintes peuvent être associées aux propriétés des entités (`Assert`) ou aux champs d'un formulaire

Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Traitement des formulaires I

Le traitement de la soumission d'un formulaire consiste à migrer les données depuis la requête HTTP (get ou post) vers les propriétés du formulaire puis vers l'entité. Il est préconisé de **grouper la création du formulaire et la gestion de la requête** dans la même fonction de contrôleur :

```php
<?php #[Route("/tournoi/saisirTnoi", name="saisirTnoi")
/* AVEC type de formulaire*/
public function saisirTnoi(Request $request,
↪  EntityManagerInterface $entityManager): Response {
  $tnoi=new Tournoi();
  $form = $this->createForm(TournoiType::class, $tnoi, []);
  $form->handleRequest($request); // SuperGlobales -> form
  if ($form->isSubmitted() && $form->isValid()) {
      $entityManager->persist($tnoi);
      $entityManager->flush();
      return $this->redirectToRoute('tournois');
```

Traitement des formulaires II

```
    }
    return $this->render('tournoi/saisieTnoi.html.twig', [
        'form' => $form->createView()
    ]);
}
```

Les messages flash de rétroaction (feedback) I

- Les messages flash sont destinés aux utilisateurs afin de les informer de la réussite ou non de leurs actions
- ils sont implémentés dans la session PHP et sont conçus pour être effacés dès qu'ils ont été utilisé une fois (par Twig)
- la méthode `addFlash()` d'`AbstractController` permet d'enregistrer un message dans une des catégories traditionnelles suivantes : info, warning, danger
- une catégorie est une chaîne de caractères quelconque
- les messages flashs sont traditionellement affichés dans un template Twig utilisant le tableau `app.flashes`

Les messages flash de rétroaction (feedback) II

Exemple contrôleur

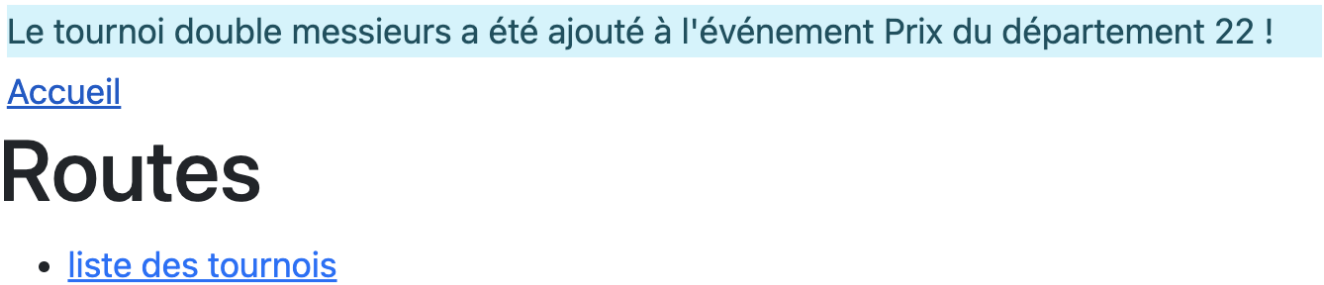
```php
<?php if ($form->isSubmitted() && $form->isValid()) {
  $em->persist($tnoi);
  $em->flush();
  $this->addFlash(
    'info',
    "Le tournoi {$tnoi->getNom()} a été ajouté à l'événement
    ↪  {$tnoi->getEv()->getNom()} !"
  );
  return $this->redirectToRoute('tournoi');
  }
```

Les messages flash de rétroaction (feedback) III

Exemple template avec colorisation par bootstrap

```
{# read and display all flash messages with bootstrap #}
{% for label, messages in app.flashes %}
    {% for message in messages %}
        <div class="alert-{{ label }} m=1">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

Exemple rendu visuel



Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Le type de champ EntityType I

Une fois créé le premier événement et tournoi, grâce à notre formulaire précédent, on souhaite fournir un nouveau formulaire d'ajout de tournoi à un événement **déjà créé** : on va utiliser un champ EntityType :

- un champ EntityType est représenté par un `select HTML` qui contiendra des options correspondant à des instances d'entités (événements)
- il hérite du champ ChoiceType qui visuellement est aussi un select mais dont les valeurs sont scalaires
- il a un grand nombre d'options possibles (tableau en 3ème position de la méthode `builder->add()`)
- Dans un premier temps, nous souhaitons sélectionner un événement par son nom, avant de saisir les champs du tournoi

Un exemple simple :

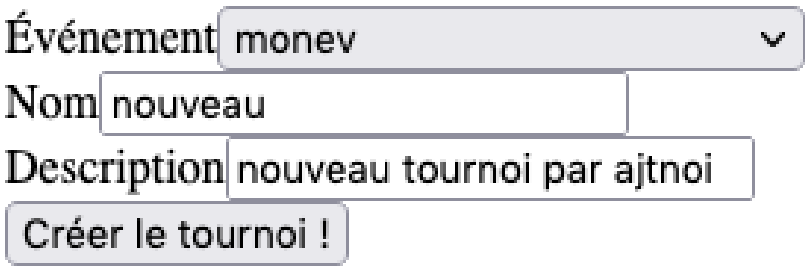
Le type de champ EntityType II

```php
<?php
class AjoutTournoiType extends AbstractType{
  public function buildForm(FormBuilderInterface $builder, array
  ↪  $options):void{
    $builder
    $builder
      ->add('ev', EntityType::class, [  // nom de l'attribut dans
      ↪  Tournoi
        'class' => Evenement::class, // quelles entités
        'choice_label' => 'nom', // label des options du select
        'label' => 'Événement' // label avant le select
      ])
      ->add('nom', TextType::class)
      ->add('description', TextType::class)
      ->add('sauver', SubmitType::class,
          ['label' => 'Créer le tournoi !'])
  ;
```

Le type de champ EntityType III

```
    }
    public function configureOptions(OptionsResolver $resolver):
    ↪  void{
        $resolver->setDefaults([
            'data_class' => Tournoi::class,
        ]);
    }
}
```

rendu visuel



le contrôleur quasi inchangé !

Le type de champ EntityType IV

```php
<?php   #[Route("/tournoi/ajTnoi", name="ajTnoi")]
  /* AVEC type de formulaire  */
 public function ajTnoi(Request $request, EntityManagerInterface
 ↪  $entityManager): Response {
   $tnoi=new Tournoi();
   $form = $this->createForm(AjoutTournoiType::class, $tnoi);
   $form->handleRequest($request); // SuperGlobales -> form
   if ($form->isSubmitted() && $form->isValid()) {
       // the original `$tnoi` variable has also been updated
       // $tnoi = $form->getData(); // inutile
       $entityManager->persist($tnoi);
       $entityManager->flush();
       return $this->redirectToRoute('tournois');
   }
   return $this->render('tournoi/saisieTnoi.html.twig', [
     'form' => $form->createView()
```

Le type de champ EntityType V

```
    ]);
}
```

le HTML généré :

```
<form name="ajout_tournoi" method="post">
  <div id="ajout_tournoi">
    <div><label for="ajout_tournoi_ev"
    ↪  class="required">Événement</label>
      <select id="ajout_tournoi_ev" name="ajout_tournoi[ev]">
        <option value="1">challenge 18 Juillet 21</option>
        <option value="2">challenge 15 Aout 21</option>
        ...
```

Quelques remarques :

Le type de champ EntityType VI

- lors de la création du formulaire (createForm()) une requête est effectuée auprès du dépôt d'Evenement (EvenementRepository) pour récupérer tous les événements et les représenter dans des options de la balise select
- on peut modifier la requête par défaut (findAll()) en utilisant le QueryBuilder de Doctrine dans une option :

```
<?php public function buildForm(FormBuilderInterface
↪  $builder, array $options):void{
  $builder
    ->add('ev', EntityType::class, [  // nom de l'attribut
    ↪  dans Tournoi
      'class' => Evenement::class, // quelles entités
      'choice_label' => 'nom', // label des options du
      ↪  select
      'label' => 'Événement', // label avant le select
      'query_builder' => function (EntityRepository $er) { //
      ↪  er = EvenementRepo.
```

Le type de champ EntityType VII

```
        $qb=$er->createQueryBuilder('e'); // e = Evenement
        // return $qb->where('e.id <= 5'); // ok DQL
        return $qb->where('e.nom like
        ↪  :exp')->setParameter('exp','chal%'); //ok
    }
    , 'expanded' => true // <select> devient checkboxes
])
->add('nom', TextType::class)
...
```

- dans le tableau d'options situé en 3ème paramètre de `builder->add()`, les deux options booléennes `multiple`, `expanded` permettent de définir les types de balises HTML parmi : `select unique`, `select multiple`, `radio`, `checkbox`. Ex. avec

Le type de champ EntityType VIII

`'expanded' => true` :

Événement

challenge 18 Juillet 21 challenge 15 Aout 21 challenge 15 Septembre 22

Nom m18

Description moind de 18 ans

Créer le tournoi !

Plan

11 Formulaires Symfony
- Création de classe de formulaire
- Gestion des associations entre entités
- Validation des formulaires
- Traitement des formulaires
- EntityType pour les champs référençant une autre entité
- Conclusion

Conclusion sur les formulaires Symfony I

Les formulaires Symfony permettent :

- en créant des Types de modéliser des champs ou aggrégations de champs
- de réaliser une double liaison (binding) entre la valeur de l'entité et celle dans le champ (value HTML) puis lors de la soumission, la valeur saisie ou modifiée est liée à l'entité qui est modifiée
- de créer des formulaires sophistiqués emboîtés avec des sous-formulaires multiples (Collection) en utilisant du JavaScript
- d'effectuer un rendu graphique personnalisable (CSS) grâce à Twig `{{form_row}}` ...
- d'utiliser des contraintes de validation, le service `Validator`, de gérer les erreurs de validation ...

Plan

12 Authentification et sécurité

Plan

12 Authentification et sécurité
- Le bundle security et l'authentification
- Contrôle d'accès

Authentification et sécurité I

Afin de permettre aux utilisateurs de se connecter avant d'atteindre certaines zones de l'application, il faut assurer un nombre important de conditions :

- des utilisateurs doivent être définis avec un (`login, password`) d'une certaine façon : dans le fichier de config en dur, dans une entité `User`, dans un fichier ... Symfony utilise la notion de `providers` pour ce "fournisseur" d'utilisateurs
- une méthode d'authentification doit être choisie e.g. par un formulaire de connexion. Symfony utilise la notion de `firewall` pour configurer la méthode d'authentification.
- le contrôle d'accès permet enfin de limiter les routes accessibles et les actions possibles pour les utilisateurs en fonction de leur rôle (`access_control`)
- beaucoup de changements entre version 5 et 6 de Symfony ! Attention aux codes trouvés sur le Web ...

Installation et configuration des utilisateurs I

Les étapes nécessaires sont nombreuses :

- Afin d'utiliser les différents paquets relatifs à la sécurité, on doit installer le bundle `security-bundle` :

  ```
  $ composer require symfony/security-bundle
  ```

- Le fichier de configuration `config/packages/security.yaml` est le le coeur du système de sécurité
- On va ensuite créer la classe User grâce au bundle Maker. Attention, User sera une entité Doctrine mais pas seulement . . . La commande `make:user` fait bien plus que `make:entity User` puisqu'elle modifie la configuration `security.yaml` !

  ```
  $ php bin/console make -help
  $ php bin/console make:user
  ```

- En ne modifiant pas les réponses par défaut, vous obtenez une entité Doctrine User ayant les propriétés suivantes :

Installation et configuration des utilisateurs II

- un id auto-généré par le SGBD
- un email UNIQUE qui est également le login de connexion de l'utilisateur (getUserName())
- un tableau de rôles qui sera typé en JSON côté BD
- un password hashé

- Bien entendu, on peut donner des réponses différentes aux questions posées par `make`. Seule obligation : le fichier `User.php` doit implémenter `UserInterface`.
- Pour ajouter des attributs à l'entité User :

```
Sf6Cours$ php bin/console make:entity User
Your entity already exists! So let's add some new fields!
New property name (press <return> to stop adding fields):
```

- il ne faut pas oublier la migration Doctrine afin de créer la table vide user associée :

Installation et configuration des utilisateurs III

```
$php bin/console make:migration
$php bin/console doctrine:migrations:migrate
```

- le fichier config/packages/security.yaml a été modifié :

```
security:
  password_hashers:

↪   Symfony\Component\Security\Core\User\PasswordAuthenticatedU
↪   'auto'
  providers:
    app_user_provider:
      entity:
        class: App\Entity\User
        property: email
```

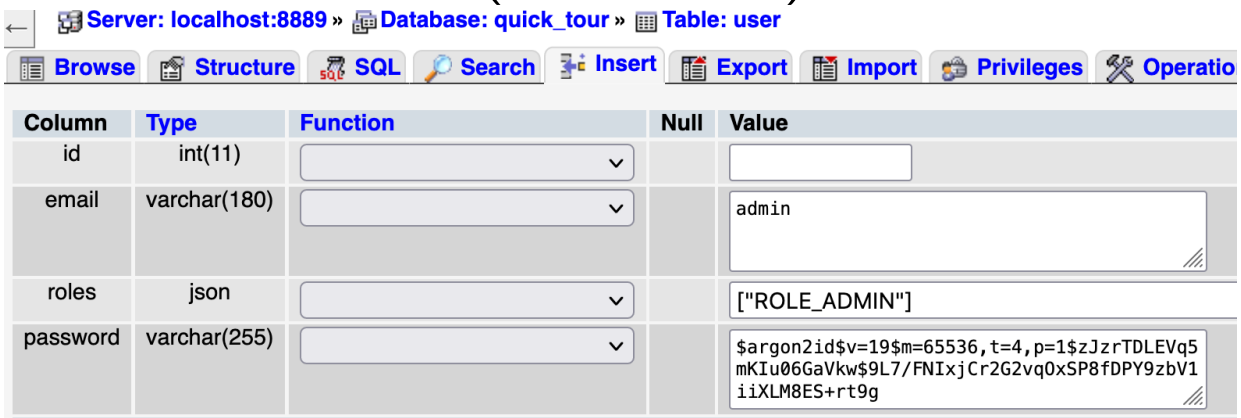
- l'algo. de hashage est automatique, ce qui signifie que c'est symfony qui l'a défini

Installation et configuration des utilisateurs IV

- le fournisseur d'utilisateur (provider) est une entité dont la propriété de login est `email`
- afin de créer des comptes `admin`, il faut générer un mot de passe hashé pour chacun :

```
$ php bin/console security:hash-password
Password hash
↪   $2y$13$1xMZjxZUPqDZ3dsKG1xYLePaUlr4UjewUUBHC8EyLZjhUNociXGd
```

- puis insérer une ligne dans la table user de mysql, soit en ligne de commande (mysql), soit en utilisant PHPMyAdmin et en créant l'admin via la GUI (notre choix)



Le pare-feu (firewall) de l'application I

- La section firewalls de la config est la section la plus importante
- Chaque entrée utilise une expression régulière (pattern) de route pour définir le mode d'authentication propre à cette zone
- Nous simplifions en utilisant le firewall `main` sans pattern qui sera valable pour toutes les routes et nous le dirigeons vers le fournisseur d'utilisateur (provider) défini auparavant :

```yaml
security:
  # ...
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      lazy: true
      # provider that you set earlier inside providers
      provider: app_user_provider
```

Le pare-feu (firewall) de l'application II

- le mode `lazy` évite de démarrer une session dans une zone sans contrôle d'accès

Authentification par formulaire de login I

Il existe d'autres méthodes d'authentification mais celle-ci est la plus simple :

- on crée un contrôleur pour gérer la route `/login` du formulaire de login :

```
$ php bin/console make:controller Login
created: src/Controller/LoginController.php
created: templates/login/index.html.twig
```

- on modifie le contrôleur :

```php
<?php class LoginController extends AbstractController{
  #[Route('/login', name: 'app_login')]
  public function index(): Response{
    return $this->render('login/index.html.twig', [
        'controller_name' => 'LoginController',
    ]);
  }
}
```

Authentification par formulaire de login II

- on modifie le firewall main afin de définir la route du formulaire de login :

```
form_login:
  # "app_login" is the name of the route created previously
  login_path: app_login
  check_path: app_login
```

- enfin on modifie le contrôleur créé précédemment afin qu'il procède à l'authentification lors de la soumission du formulaire :

```
<?php #[Route('/login', name: 'app_login')]
public function index(AuthenticationUtils
↪  $authenticationUtils): Response{
  // get the login error if there is one
  $error =
  ↪  $authenticationUtils->getLastAuthenticationError();
  // last username entered by the user
  $lastUsername = $authenticationUtils->getLastUsername();
```

Authentification par formulaire de login III

```
    return $this->render('login/index.html.twig', [
        'controller_name' => 'LoginController',
        'last_username' => $lastUsername,
        'error'         => $error,
        ]);
}
```

- Attention, par défaut, après un login réussi, une redirection a lieu vers la route racine (/) qui doit exister ! (sinon erreur 404)

```
form_login:
    # "app_login" is the name of the route created previously
    login_path: app_login
    check_path: app_login
    default_target_path: afterLogin # pour éviter / par défaut
```

Authentification par formulaire de login IV

- La route de nom afterLogin est à choisir pour les utilisateurs connectés
- Le formulaire de login (`index.html.twig`) doit être modifié :

```
{% block body %}
{% if error %}
  <div>{{ error.messageKey|trans(error.messageData,
↪  'security') }}</div>
{% endif %}

<form action="{{ path('app_login') }}" method="post">
    <label for="username">Email:</label>
    <input type="text" id="username" name="_username"
↪   value="{{ last_username }}"/>
    <label for="password">Password:</label>
    <input type="password" id="password" name="_password"/>
```

Authentification par formulaire de login V

```
    {# If you want to control the URL the user is redirected
↪   to on success
    <input type="hidden" name="_target_path"
↪   value="/account"/> #}
    <button type="submit">login</button>
</form>
{% endblock %}
```

- les noms de champs `_username` et `_password` sont indispensables puisqu'ils seront traités par le service `authenticationUtils`

Déconnexion (logout) I

- La déconnexion a besoin d'une route mais pas de contrôleur !
- En effet, une requête sur `/logout` générera une redirection qu'il faut prévoir (target)
- La configuration du `firewall` est modifiée :

```yaml
form_login:
    ...
logout:
    path:   /logout
    target: /   # redirigé vers la racine par défaut
```

- Dans le contrôleur, il faut définir la route associée à une méthode qui ne sera jamais appelée !

Déconnexion (logout) II

```php
<?php #[Route('/logout', name: 'app_logout', methods:
↪  ['GET'])]
public function logout(){ // controller INUTILE car jamais
↪  appelé
  throw new \Exception('Don\'t forget to activate logout in
  ↪  security.yaml');
} // exception jamais levée !
```

- Ainsi toute requête sur la route `/logout` sera redirigée vers la page d'accueil `/`

Plan

12 Authentification et sécurité
- Le bundle security et l'authentification
- Contrôle d'accès

Rôles et contrôle d'accès I

Nous avons vu auparavant :

- comment créer une authentification basée sur les **sessions** PHP,
- où les utilisateurs sont enregistrés dans une classe d'entités User qui contient la propriété `roles` qui est une chaîne JSON contenant un tableau de rôles élémentaires tels que `ROLE_ADMIN, ROLE_USER, ...`
- il nous reste à définir le contrôle d'accès aux différentes routes du site afin de ne permettre l'accès au back-office (administration) qu'aux utilisateurs possédant le rôle `ROLE_ADMIN`

Nous allons modifier le fichier de configuration `config/packages/security.yaml` :

Rôles et contrôle d'accès II

```yaml
security:
  firewalls:
    # ...

  access_control:
    # require ROLE_ADMIN for /admin*
    - { path: '^/admin', roles: ROLE_ADMIN }

    # the 'path' value can be any valid regular expression
    # (this one will match URLs like /api/post/7298 and
↪   /api/comment/528491)
    - { path: ^/api/(post|comment)/\d+$, roles: ROLE_USER }
```

Quelques remarques :

Rôles et contrôle d'accès III

- le contrôle d'accès contient une suite de règles où chacune est une liste (raison du tiret (-)) de `cle:valeur` ou un objet JSON contenant un patron de chemin (path) sous forme d'expression régulière, un rôle (roles) et des options ...
- l'algorithme de contrôle consiste à rechercher la première règle qui correspond à la route requise
- il faut donc indiquer les patrons plus spécifiques avant ceux qui sont plus généraux
- les chemins non spécifiés sont libres d'accès
- les options peuvent préciser :
  - des méthodes HTTP (`methods: [POST, PUT]`)
  - des adrs IP (`ip: 127.0.0.1`)
  - des domaines Internet (`host: symfony\.com$`)
  - un port (`port:80`)

Rôles et contrôle d'accès IV

- En cas d'accès non autorisé, si l'utilisateur est authentifié, une exception `AccessDeniedHttpException` est levée !
- Si l'utilisateur n'est pas authentifié, il est redirigé sur la route de `login` afin qu'il s'authentifie
- en mode `dev` les exceptions de routes ou de contrôle d'accès sont affichées avec la trace de pile mais en mode `prod` elles peuvent être personnalisées ...

Comment accéder aux informations sur l'utilisateur I

Pour accéder aux informations concernant l'utilisateur authentifié, on distingue deux cas :

- dans un template Twig, on utilisera la propriété `app.user` :

```
Bonjour {{ app.user.email ?? "Anonyme" }} {# ou bien {% if
↪  app.user %} #}
```

- dans du code PHP, on utilisera la méthode `getUser()` pour récupérer l'entité User authentifié ou null :
  - depuis un contrôleur : `$this->getUser()`
  - depuis une classe quelconque, il faut utiliser le **service** `security` :

Comment accéder aux informations sur l'utilisateur II

```php
<?php use Symfony\Component\Security\Core\Security;
class ExampleService{
  private $security;
  public function __construct(Security $security){
    // Avoid calling getUser() in the constructor: auth may
    ↪  not be complete yet
    $this->security = $security;
  }
  public function someMethod(){
    // returns User object or null if not authenticated
    $user = $this->security->getUser();
    ...
```

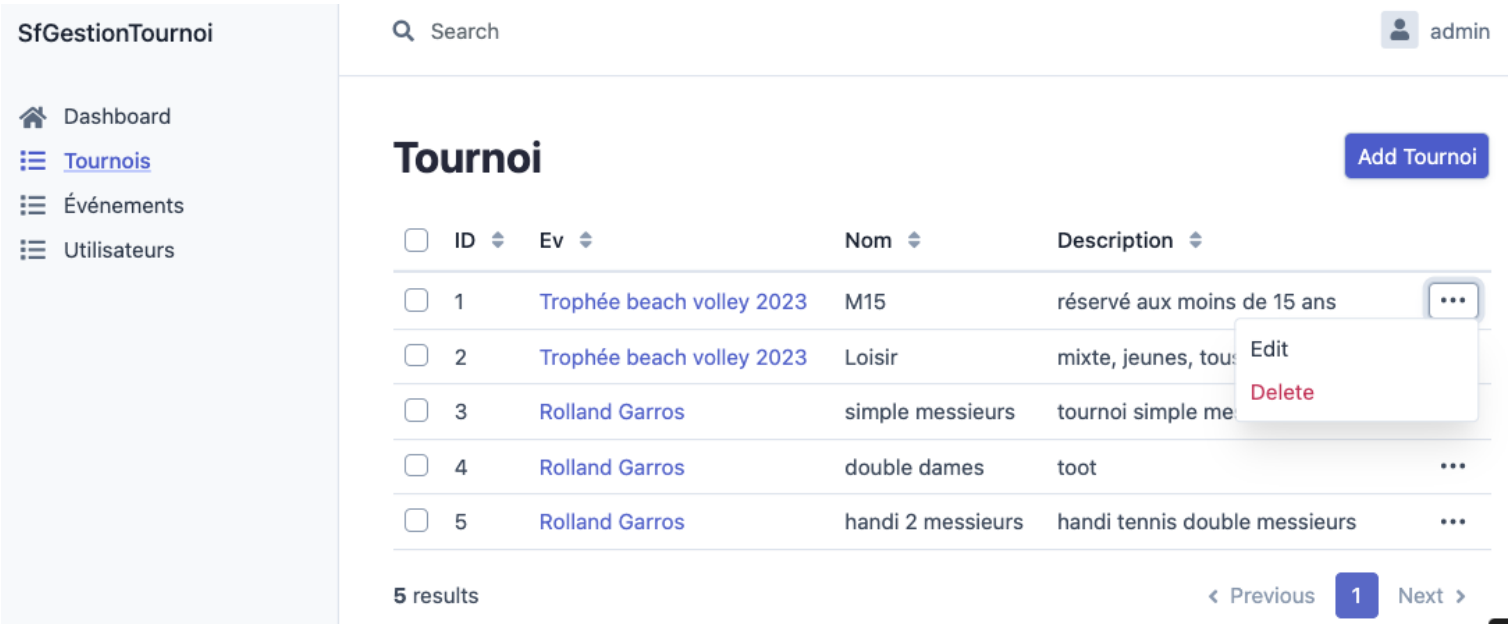
Plan

13 EasyAdmin : CRUD pour les entités

EasyAdmin : CRUD pour les entités I

Le bundle EasyAdmin permet de créer des pages accessibles uniquement par l'administrateur qui vont lui permettre les opérations CRUD (en back-office)

FIGURE – EasyAdmin



EasyAdmin : CRUD pour les entités II

- Installation du bundle EasyAdmin

  `$ symfony composer req "admin"`

- Création d'un tableau de bord (Dashboard) :

  `$ symfony console make:admin:dashboard`

  Conserver les réponses par défaut provoque la création d'un nouveau contrôleur `DashboardController` situé dans le répertoire `Controller/Admin` et d'une méthode gérant la route `/admin`

- Dans cette méthode, il faut commenter l'unique ligne courante (`return parent::index();`) et décommenter l'option 1 qui permet de rediriger vers le rendu visuel d'un des `XCrudController` qui vont gérer les opérations d'éditition sur les événements, les tournois, les inscriptions ...

```php
<?php $adminUrlGenerator = $this->container->get(
↪  AdminUrlGenerator::class);
return $this->redirect($adminUrlGenerator->setController(
↪  TournoiCrudController::class)->generateUrl());
```

EasyAdmin : CRUD pour les entités III

- Il suffit maintenant de créer un nouveau contrôleur par entité afin de permettre les opérations Create Retrieve Update Delete sur celle-ci :

  `$ symfony console make:admin:crud`

  Il faut **choisir** l'entité à administrer e.g. Tournoi
- Un nouveau contrôleur est créé : `TournoiCrudController`. Il faut maintenant ajouter une option de menu dans le tableau de bord afin de pouvoir éditer des tournois. Pour cela, ajouter les 2 lignes dans `DashboardController` :

```php
<?php
use App\Entity\Tournoi; // avant Class ...
...
configureMenuItems(): iterable {
  ...
  yield MenuItem::linkToCrud('Tournois', 'fas fa-list',
  ↪  Tournoi::class); // crud
```

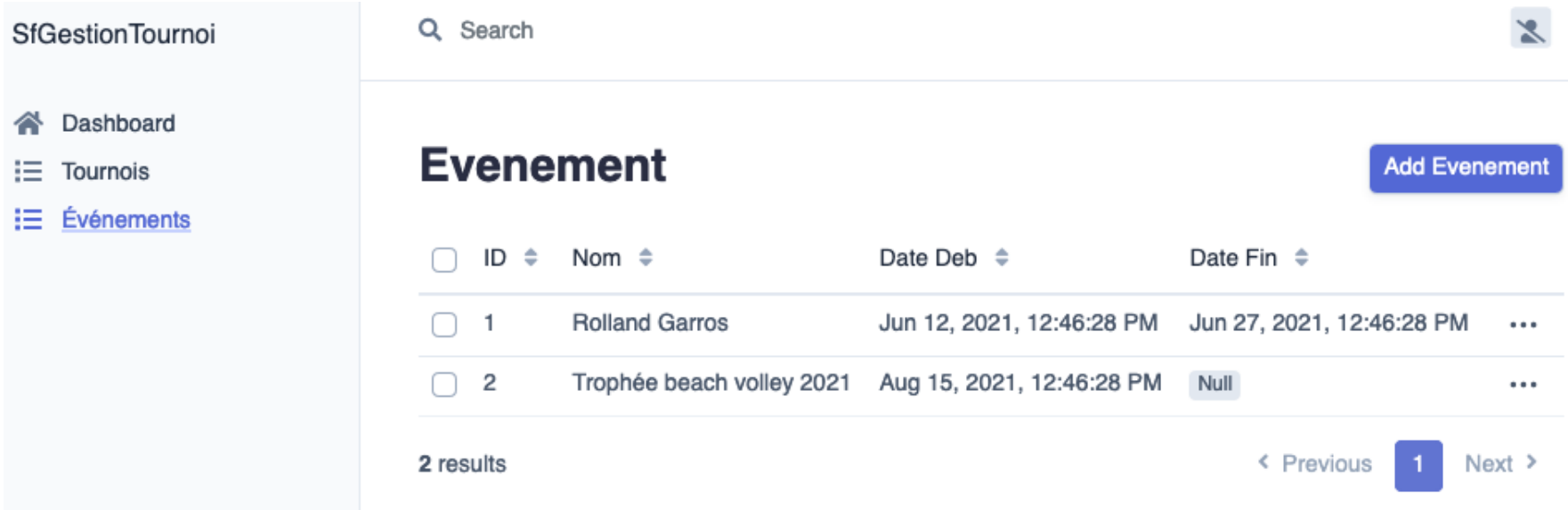
EasyAdmin : CRUD pour les entités IV

Cette dernière ligne génère un item de menu "Tournois"

- une fois cliqué un lien du menu, on chargera la page d'édition de tournoi :
- Remarquons que `yield` est une sorte de return qui ne termine pas la fonction mais permet de retourner d'autres objets par des `yield` suivants (sorte d'Observable js) : il retourne donc des `iterable`.

EasyAdmin : CRUD pour les entités V

- En effectuant le même travail sur l'entité Evenement on obtient le visuel :



- Malheureusement, par défaut, on ne peut éditer que les attributs scalaires des tournois et on ne visualise même pas l'association entre Tournoi et Evenement !
- Afin de matérialiser les associations entre entités, il suffit de :
  - ajouter une méthode `__toString()` à l'entité événement

EasyAdmin : CRUD pour les entités VI

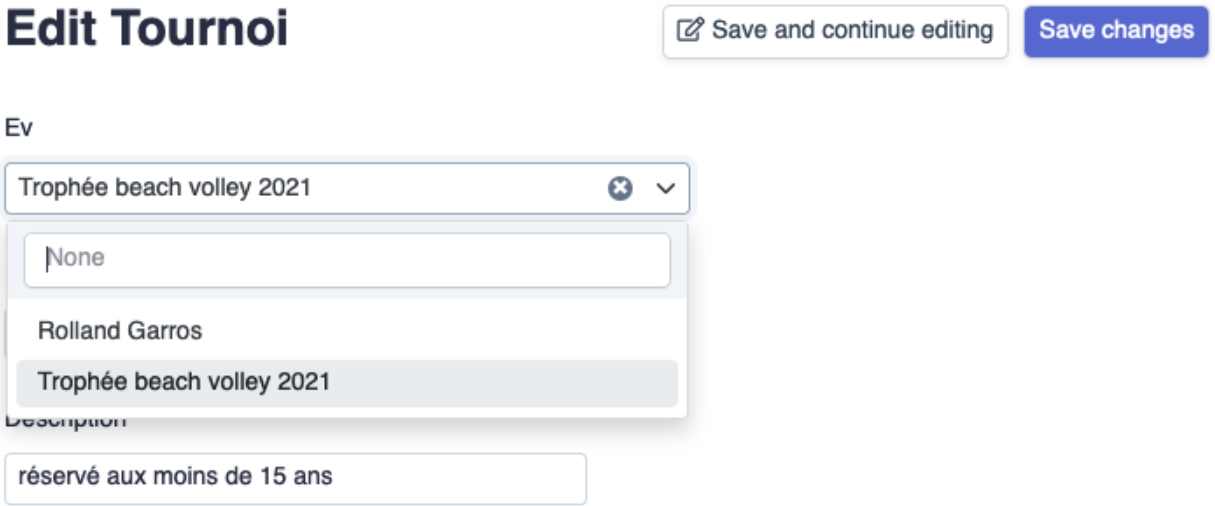
- personnaliser les champs à afficher dans le contrôleur crud de l'entité Tournoi comme suit (AssociationField)

```
class TournoiCrudController extends AbstractCrudController{
  ...
  public function configureFields(string $pageName): iterable
↪   {
    yeld IdField::new('id')->hideWhenUpdating();
    yield AssociationField::new('ev');
    yield TextField::new('nom');
    yield TextField::new('description');
  }
}
```

EasyAdmin : CRUD pour les entités VII

EasyAdmin : CRUD pour les entités VIII

- En cliquant sur `Edit` du menu d'un tournoi, on obtient le rendu suivant :



- Il existe bien d'autres personnalisations ...

Plan

14 Développer un service Web (API REST) avec Symfony

Plan

14 Développer un service Web (API REST) avec Symfony
- Service Web et API RESTful
- Architecture SPA et backend
- Bundle API Platform
- Gestion des relations entre entités
- Stratégie d'API RESTful

Introduction I

- Les “ressources web” ont été définies pour la première fois sur le World Wide Web comme des documents ou des fichiers identifiés par leur URL
- Cependant, elles ont aujourd'hui une définition beaucoup plus générique et abstraite qui inclut toute chose ou entité pouvant être identifiée, nommée, adressée ou gérée d'une façon quelconque sur le web
- Dans un service web REST, les requêtes effectuées sur l'URI d'une ressource produisent une réponse qui peut être en **HTML, XML, JSON** ou un autre format
- La réponse confirme que la ressource stockée a été altérée et elle peut fournir des liens hypertextes vers d'autres ressources ou collection de ressources liées

Introduction II

- Lorsque le protocole HTTP est utilisé, comme c'est souvent le cas, les opérations disponibles sont POST, GET, PUT, DELETE ou d'autres méthodes HTTP
- REST (REpresentational State Transfer) est un style d'architecture définissant un ensemble de contraintes et de propriétés basées sur le protocole HTTP
- Les services web conformes au style d'architecture REST, aussi appelés services web RESTful, établissent une interopérabilité entre les ordinateurs sur Internet
- Les services web REST permettent aux systèmes effectuant des requêtes d'accéder et de manipuler des représentations textuelles de ressources web à travers un jeu d'opérations uniformes et prédéfinies **sans état**

Introduction III

- D'autres types de services web tels que les services web SOAP exposent leurs propres jeux d'opérations arbitraires.
- Avec l'utilisation d'un protocole sans état et d'opérations standards, les systèmes REST visent la réactivité, la fiabilité et l'extensibilité, par la réutilisation de composants pouvant être gérés et mis à jour sans affecter le système global, même pendant son fonctionnement
- Le terme REpresentational State Transfer (REST) a été défini pour la première fois en 2000 par Roy Fielding dans le chapitre 5 de sa thèse de doctorat
- La thèse de Fielding a expliqué les principes de REST auparavant connus comme le "modèle objet de HTTP" depuis 1994 et qui ont été utilisés dans l'élaboration des standards HTTP 1.1 et URI

Introduction IV

- Le terme est censé évoquer comment une application web bien conçue se comporte : c'est un réseau de ressources (une machine à états virtuelle) au sein duquel l'utilisateur évolue en sélectionnant des liens, tels que /utilisateur/tom, et des opérations telles que GET ou DELETE (des transitions d'état), provoquant le transfert de la ressource (représentant le nouvel état de l'application) vers l'utilisateur pour être utilisée

Les opérations CRUD (Create, Retrieve, Update, Delete) sur les ressources du service sont matérialisées par les méthodes HTTP :

- Créer (create) : POST
- Rechercher (retrieve) : GET
- Mettre à jour (update) : PUT
- Supprimer (delete) : DELETE

Plan

14 Développer un service Web (API REST) avec Symfony
- Service Web et API RESTful
- Architecture SPA et backend
- Bundle API Platform
- Gestion des relations entre entités
- Stratégie d'API RESTful

Architecture SPA et backend I

Une Single Page Application est une application Web divisée en 2 parties :

- une partie fontale (frontend) écrite en JavaScript chargée une unique fois lors de la première et unique requête au site Web. Par la suite, toute la logique de l'application résidant dans le js, il n'y aura pas de rechargement de la page. De nombreux frameworks permettent ce fonctionnement (Angular2, React.js, Vue.js, ...). L'application nécessitant des transferts de données (uniquement des données) utilise des services Webs pour ces échanges dans le backend (partie arrière).
- une partie arrière qui assure :
  - l'émission de données vers le fontend (Retrieve)
  - la sauvegarde des données ajoutées, modifiées ou supprimées (Create, Update, Delete)

Architecture SPA et backend II

La façon dont le backend assure la conservation des données (SGBD SQL ou NoSQL, fichiers, ...) est totalement indépendante du frontend et doit lui être inconnue. Le backend "expose son API" qui est le mode d'emploi du service.

Les métiers du Web les plus fréquents sont actuellement :

- développeur frontend : connaissance d'un framework json
- développeur backend : connaissance SGBD, API REST, équilibrage de charge (balancing)
- développeur fullstack : frontend et backend

Et Symfony ? Symfony peut être utilisé avec l'architecture classique de rechargement de page à chaque requête ou bien comme backend. On peut aussi utiliser AJAX pour ne recharger que des parties de pages !

Plan

14 Développer un service Web (API REST) avec Symfony
- Service Web et API RESTful
- Architecture SPA et backend
- Bundle API Platform
- Gestion des relations entre entités
- Stratégie d'API RESTful

Bundle API Platform I

- Ce bundle est destiné à créer une API RESTful facilement
- On peut créer manuellement une API sans bundle mais ce sera beaucoup plus long et très répétitif
- API Platform est un outil libre indépendant de Symfony permettant de développer des API REST
- Nous utiliserons en plus du bundle API Platform, les bundle Doctrine et Maker ...

Installation I

- Nous allons créer un nouveau projet Symfony limité au squelette (pas un projet de site Web)
- Puis nous installons les bundles indispensables : api, ...

```
$ symfony new SfApiTournoi
$ cd SfApiTournoi
$ ../composer.phar require api
$ symfony server:start -d -port=8080;symfony open:local
```

Si nous visitons la page `https://127.0.0.1:8080/api`, nous obtenons le message : `No operations defined in spec!` Modifions le fichier de configuration du package afin de lui donner un titre et une description :

Installation II

```yaml
# config/packages/api_platform.yaml
api_platform:
    title: 'Symfony REST API de tournoi'
    description: 'A Symfony API to manage tournaments.'
    version: '1.0.0'
    mapping:
        paths: ['%kernel.project_dir%/src/Entity']
    patch_formats:
        json: ['application/merge-patch+json']
    swagger:
        versions: [3]
```

Il nous faut maintenant définir nos ressources (entités) qui vont être manipulées par notre API REST !

Création d'entités accessibles par l'api I

- Nous utiliserons le bundle Maker afin de nous simplifier la tâche dans la création d'entités accessibles. Les entités seront des livres d'une bibliothèque.

```
$ composer require symfony/maker-bundle --dev
$ php bin/console make:entity
Class name of the entity to create or update (e.g.
↪  VictoriousPuppy):
> Book
Mark this class as an API Platform resource (expose a CRUD
↪  API for it) (yes/no) [no]:
> yes
created: src/Entity/Book.php
created: src/Repository/BookRepository.php
Entity generated! Now let's add some fields!
New property name (press <return> to stop adding fields):
> title
```

Création d'entités accessibles par l'api II

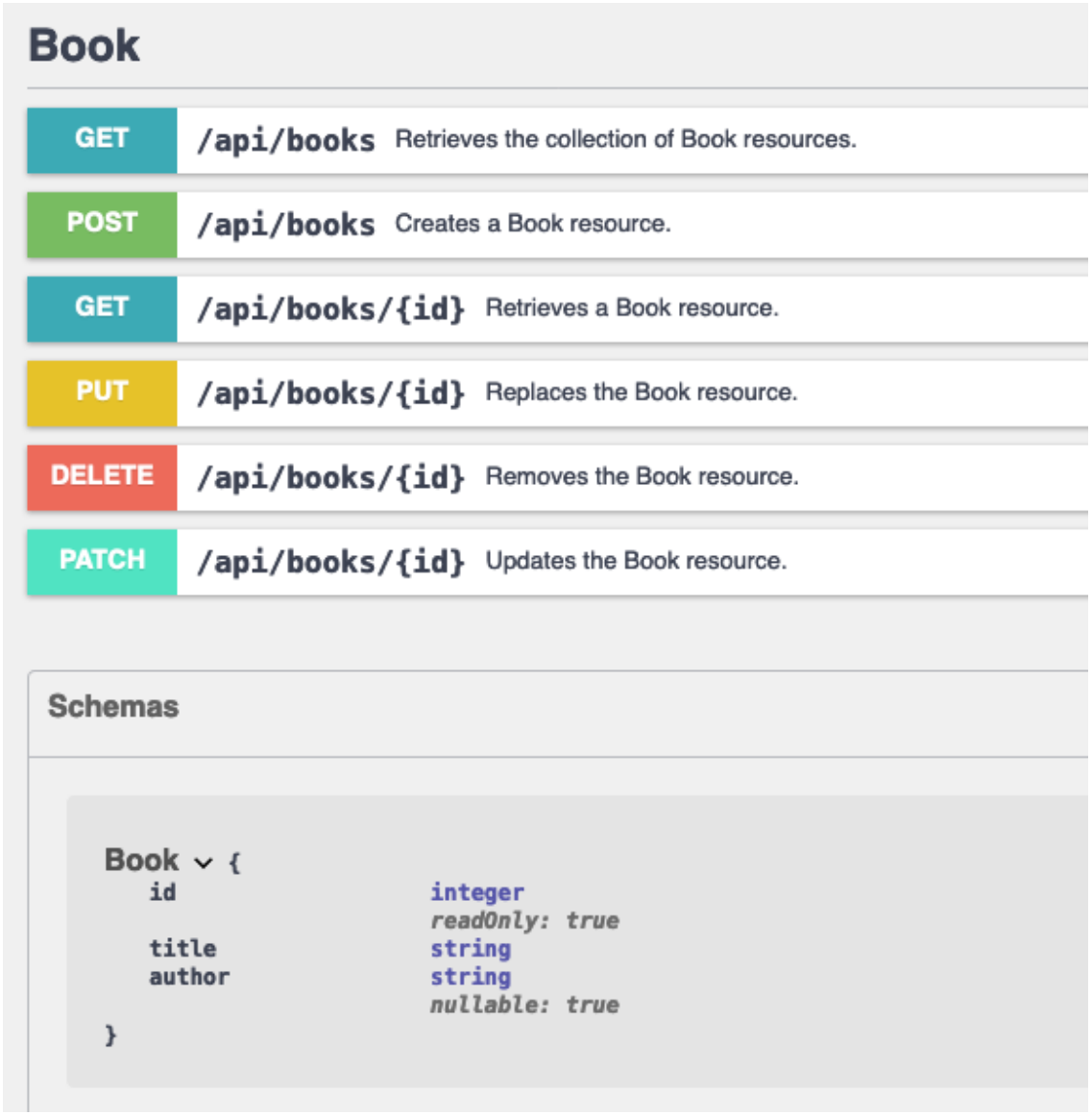
```
...
>author
```

- Une fois créée l'entité ressource, si nous observons `src/Entity/Book`, nous apercevons une annotation supplémentaire `@ApiResource()` à la classe Book (en plus de `@Entity`) !
- Après avoir défini la configuration d'accès à la BD dans `.env` : `DATABASE_URL="mysql://tempo:tempo@127.0.0.1:8889/tempo?serverVersion=5.7&charset=utf8"`, il faut ensuite générer la migration puis l'exécuter :

```
$ php bin/console make:migration
$ php bin/console doctrine:migrations:migrate
```

- La table `book` ayant bien été créée, on peut voir l'exposition de l'api dans l'image suivante :
- On y retrouve les requêtes CRUD classiques ainsi qu'un PATCH permettant de mettre à jour une entité (sans REPLACE)

Création d'entités accessibles par l'api III



Requêtes et réponses I

- Après avoir rempli la table grâce à PhpMyAdmin, on peut exécuter en cliquant une requête (Retrieve) simple : GET api/books
- Le **Web des données** adjoint des **méta-données** aux données fournies afin d'ajouter de la sémantique aux données échangées
- Ces méta-données peuvent être exprimées avec Ressource Description Framework (RDF) en utilisant des balises HTML ou avec JSON-LD en utilisant une notation JSON
- Hydra est un composant d'API Platform qui permet d'utiliser JSON-LD avec le vocabulaire `hydra:`
- Ainsi la réponse fournie (liste de livres dans `hydra:member`) est encapsulée dans un objet décrivant cette réponse (`hydra:collection`)

Requêtes et réponses II

- Enfin Hydra permet également de ne récupérer qu'un nombre maximum de ressources (30 par défaut) grâce au mécanisme de page fourni dans la chaîne de requête
- Suit l'exécution de la requête :

```
Curl

curl -X 'GET' \
'https://127.0.0.1:8080/api/books?page=1' \
-H 'accept: application/ld+json'

Request URL

https://127.0.0.1:8080/api/books?page=1

Server response
Code        Details
```

Requêtes et réponses III

200
Response body
Download


```
{
"@context": "/api/contexts/Book",
"@id": "/api/books",
"@type": "hydra:Collection",
"hydra:member": [
  {
    "@id": "/api/books/1",
    "@type": "Book",
    "id": 1,
    "title": "Les misérables",
    "author": "Victor Hugo"
  },
  {
```

Requêtes et réponses IV

```
    "@id": "/api/books/2",
    "@type": "Book",
    "id": 2,
    "title": "Regain",
    "author": "Jean Giono"
  }
],
"hydra:totalItems": 2
}
```

Remarques :

- la réponse à `GET /api/books` est une chaîne JSON contenant des métadonnées (`@id`, `@type`, `hydra: ...`) et des données de la ressource (`id`, `title`, `author`)

Requêtes et réponses V

- la métadonnée `@id` est une IRI (généralisation internationale d'URI) permettant d'identifier la ressource avec l'API REST : `GET /api/books/2` récupérera Regain de Giono
- par défaut, toutes les propriétés de la ressource sont "exposées" : on peut les lire (GET) et les écrire (POST, PUT, PATCH)
- Dans le cas d'entités possédant des relations vers d'autres entités, i.e. un événement contient des tournois, la propriété ev de tournoi qui désigne l'événement associé sera représenté par son IRI (@id) :

```
{
  "nom": "Simple Messieurs",
  "description": "réservé aux joueurs",
  "ev": "/api/evenements/1"
}
```

- symétriquement, la propriété `tournois` de l'entité Evenement sera un tableau des IRI des tournois :

Requêtes et réponses VI


```
{
  "@context": "/api/contexts/Evenement",
  "@id": "/api/evenements/1",
  "@type": "Evenement",
  "id": 1,
  "nom": "Rolland-Garros 2022",
  "dateDeb": "2022-06-06T08:00:00+00:00",
  "dateFin": "2022-06-20T22:00:00+00:00",
  "tournois": [
    "/api/tournois/1",
    "/api/tournois/2"
  ]
}
```

Test d'un POST (Create) I

- En utilisant l'interface graphique de l'api, on clique sur POST puis "test it out"
- Puis on remplit le titre et l'auteur du nouveau livre que l'on veut ajouter (CREATE)
- On obtient alors une réponse 201 indiquant que l'ajout de la ressource a été effectué
- On récupère également l'id de l'entité ajouté à la BD (5)

Curl

```
curl -X 'POST' \
  'https://127.0.0.1:8080/api/books' \
  -H 'accept: application/ld+json' \
  -H 'Content-Type: application/ld+json' \
  -d '{
```

Test d'un POST (Create) II

```
  "title": "Les Pensées",
  "author": "Blaise Pascal"
}'

Request URL

https://127.0.0.1:8080/api/books

Server response
Code        Details
201
Response body
Download

{
  "@context": "/api/contexts/Book",
  "@id": "/api/books/5",
```

Test d'un POST (Create) III

```
  "@type": "Book",
  "id": 5,
  "title": "Les Pensées",
  "author": "Blaise Pascal"
}
```

Test d'un PATCH (Update) I

- On souhaite modifier le dernier livre ajouté en supprimant le prénom de l'auteur
- En utilisant l'interface graphique de l'api, on clique sur PATCH puis "test it out"
- On modifie les paramètres : id = 5 et author = Pascal puis on exécute
- Le résultat de code 200 (Book Resource updated) suit :

Curl

```
curl -X 'PATCH' \
  'https://127.0.0.1:8080/api/books/5' \
  -H 'accept: application/ld+json' \
  -H 'Content-Type: application/merge-patch+json' \
  -d '{
  "author": "Pascal"
```

Test d'un PATCH (Update) II

```
}'

Request URL

https://127.0.0.1:8080/api/books/5

Server response
Code        Details
200
Response body
Download

{
  "@context": "/api/contexts/Book",
  "@id": "/api/books/5",
  "@type": "Book",
  "id": 5,
```

Test d'un PATCH (Update) III

```
  "title": "Les Pensées",
  "author": "Pascal"
}
```

Test d'un DELETE et autres requêtes I

- En réalisant le même type de manipulation avec la méthode DELETE 5, on obtient la réponse 204 (Book resource deleted)
- En réalisant, un GET api/books, on vérifie que le livre n'existe plus
- En testant des requêtes impossibles (suppression d'un id inexistant, ajout d'un livre sans titre, ...), on obtient un code 404 (Resource not found) ou 400 (Invalid input)
- Attention, un attribut d'entité obligatoire de type chaîne acceptera une chaîne vide ("" is not null)
- Après quelques manipulations de l'interface graphique de l'api, on s'aperçoit que l'historique des dernières opérations est conservé et nous permet donc d'administrer la BD d'une façon assez agréable !
- la requête PUT permet de remplacer une ressource existante comme PATCH

Activation et désactivation d'opérations I

Si on souhaite désactiver certaines opérations de notre API, par exemple pour ne permettre que la consultation, il faut modifier l'annotation de l'entité :

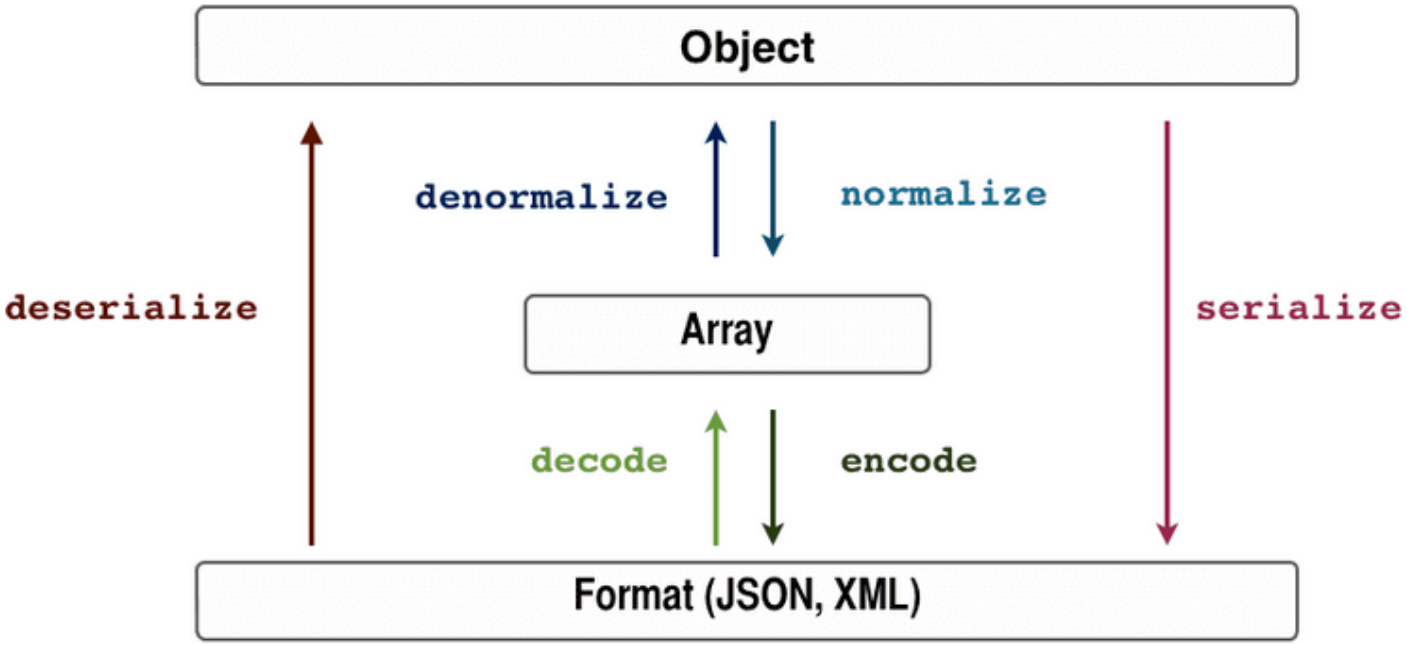
```
#[ApiResource(
  collectionOperations={"GET"},
  itemOperations={"GET","patch"}
)]
```

Après rechargement de la page, seule la requête GET sera possible sur la collection de livres et pour un id de livre, on ne pourra que le lire où le mettre à jour (pas de PUT ni DELETE)

Autres spécialisations de l'API Platform I

On peut ajouter d'autres annotations pour spécialiser l'api :

- La **sérialisation** d'un objet PHP (entité) en JSON passe par une étape de **normalisation** vers un tableau PHP puis d'**encodage** vers une chaîne JSON.



Autres spécialisations de l'API Platform II

- Le contexte de normalisation (et de dénormalisation) permet de définir des groupes d'attributs de l'entité afin de sélectionner les propriétés d'entités qui seront normalisées (donc lisibles) et dénormalisées (donc inscriptible). Par défaut, il existe un groupe par défaut auquel toutes les propriétés appartiennent (tout est exposé)
- De plus, on peut spécialiser des groupes de (dé)normalisation pour chaque type d'opération : itemOperations(GET, POST, ...), collectionOperation(GET)
- La définition d'un groupe de normalisation permettra pour tous les attributs qui y participeront d'être sérialisés donc **lus** par l'api (get)
- La définition d'un groupe de dénormalisation permettra pour tous les attributs qui y participeront d'être désérialisés donc **écrits** par l'api (post put patch ...) dans la BD

Autres spécialisations de l'API Platform III

L'exemple d'annotation de classe suivant spécialise les deux seules opérations possibles :

```
#[ApiResource(
    collectionOperations={
      "get"={
         "normalization_context"={
           "groups"={"book:readable"}
         }
      }
    },
    itemOperations={
      "patch"={
         "denormalization_context"={
           "groups"={"book:writable"}
         }
      }
    },
```

Autres spécialisations de l'API Platform IV

```
    attributes={"order"={"title"="ASC"},
    ↪  "paginationEnabled"=false}
  )]
#[ORM\Entity(repositoryClass=BookRepository::class)]
```

Il faut maintenant annoter chaque attribut de l'entité par un ou plusieurs des groupes afin de permettre sa lecture/modification :

- `id` sera uniquement lisible
- `title` sera lisible et modifiable
- `author` sera uniquement modifiable

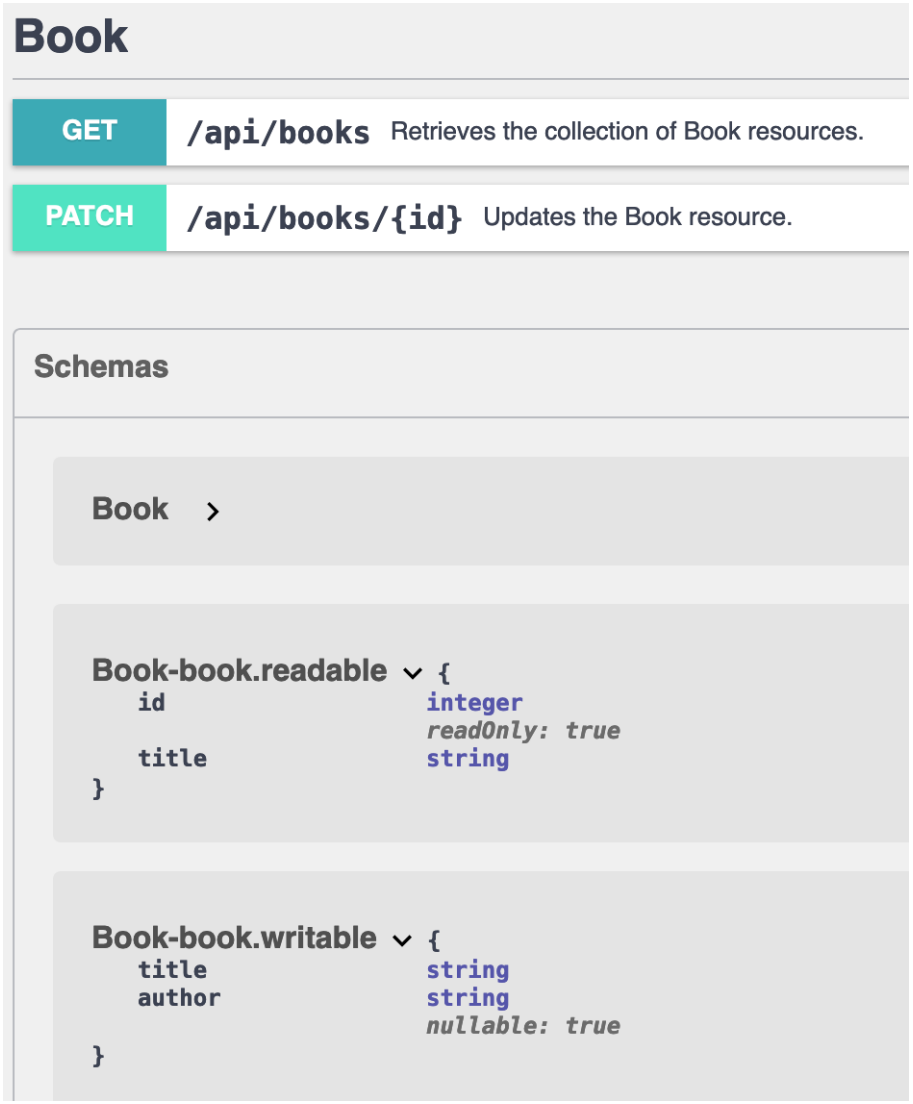
Autres spécialisations de l'API Platform V

```
 #[Groups({"book:readable"})]
private $id;
...
 #[Groups({"book:readable", "book:writable"})]

 private $title;
...
 #[Groups({"book:writable"})]

private $author;
```

Autres spécialisations de l'API Platform VI

Cet exemple fournit une API ayant les schémas suivants :

Autres spécialisations de l'API Platform VII



Autres spécialisations de l'API Platform VIII

Remarques

- Les modifications des annotations d'entité ne sont pas toujours prises en compte sauf si on nettoie le cache :
  `$ php bin/console cache:clear`
- l'opération get retournera les livres ordonnés par leur titre (attributes)
- la pagination ne sera pas effective (par défaut 30 lignes à chaque get)

Plan

14 Développer un service Web (API REST) avec Symfony
- Service Web et API RESTful
- Architecture SPA et backend
- Bundle API Platform
- Gestion des relations entre entités
- Stratégie d'API RESTful

Gestion des relations entre entités I

- Par défaut, les propriétés d'entités qui sont des relations vers d'autres entités sont lues ou écrites sous forme d'une IRI référençant l'API (i.e. /api/tournois/1).
- Pour inclure des propriétés de sous-entités dans le JSON retourné par un GET, il suffit de les annoter par le groupe de la super-entité. Ainsi la normalisation de la sous-entité sera effectuée
- Attention à ne pas créer de références cycliques, sinon la requête échouera pour cause de jointures trop nombreuses ! Si on veut voir dans les événements, leurs tournois et les équipes inscrites, il ne faudra pas annoter les propriétés Tournoi(ev) et Equipe(tournoi)

Un exemple pour inclure à 2 niveaux I

Un événement contient des tournois qui contiennent des équipes :

```php
<?php #[ApiResource(normalizationContext={"groups"={"ev"}})]
/* Attention : guillemets et pas de quotes dans les annotations !
↪  */
#[ORM\Entity(repositoryClass=EvenementRepository::class)]
class Evenement {
 #[ORM\Id]
 #[ORM\GeneratedValue]
 #[ORM\Column(type="integer")]
 #[Groups({"ev"})]

  private $id;

 #[ORM\Column(type="string", length=255)]
 #[Groups({"ev"})]
```

Un exemple pour inclure à 2 niveaux II

```
 private $nom;
 ...

#[ORM\OneToMany(targetEntity=Tournoi::class, mappedBy="ev")]
#[Groups({"ev"})]

 private $tournois;
```

Un tournoi contient des équipes inscrites et appartient à un évt. :

```
<?php #[ApiResource()]
#[ORM\Entity(repositoryClass=TournoiRepository::class)]
class Tournoi{
 #[ORM\Id]
 #[ORM\GeneratedValue]
 #[ORM\Column(type="integer")]
 #[Groups({"ev"})]
```

Un exemple pour inclure à 2 niveaux III

```
private $id;
#[ORM\Column(type="string", length=255)]
#[Groups({"ev"})]

private $nom;
...

#[ORM\ManyToOne(targetEntity=Evenement::class,
↪  inversedBy="tournois")]
#[ORM\JoinColumn(nullable=false)]
 /* ne pas mettre le group "ev" sinon jointures infinies ! */
private $ev;

#[ORM\OneToMany(targetEntity=Equipe::class,
↪  mappedBy="tournoi")
#[Groups({"ev"})]
```

Un exemple pour inclure à 2 niveaux IV

```
  private $equipes;
```

Une équipe est inscrite à un tournoi : même principe, toutes les propriétés d'équipes **SAUF tournoi** sont annotées avec le groupe "ev".

Résultat de GET /api/evenements/1


```
{
  "@context": "/api/contexts/Evenement",
  "@id": "/api/evenements/1",
  "@type": "Evenement",
  "id": 1,
  "nom": "Rolland-Garros 2022",
  "dateDeb": "2022-06-06T08:00:00+00:00",
  "dateFin": "2022-06-20T22:00:00+00:00",
  "tournois": [
    {
      "@id": "/api/tournois/1",
```

Un exemple pour inclure à 2 niveaux V


```
"@type": "Tournoi",
"id": 1,
"nom": "Simple Messieurs",
"description": "réservé aux joueurs",
"equipes": [
  {
    "@id": "/api/equipes/1",
    "@type": "Equipe",
    "id": 1,
    "nom": "Raphael Nadal",
    "description": "",
    "niveau": 1
  },
  {
    "@id": "/api/equipes/2",
    "@type": "Equipe",
    "id": 2,
```

Un exemple pour inclure à 2 niveaux VI


```
      "nom": "Novak Djokovic",
      "description": "",
      "niveau": 1
    },
    {
      "@id": "/api/equipes/3",
      "@type": "Equipe",
      "id": 3,
      "nom": "Michel",
      "description": "",
      "niveau": 5
    }
  ]
},
{
  "@id": "/api/tournois/2",
  "@type": "Tournoi",
```

Un exemple pour inclure à 2 niveaux VII


```
"id": 2,
"nom": "Double Dames",
"description": "réservé aux joueuses",
"equipes": []    } ]  }
```

Plan

14 Développer un service Web (API REST) avec Symfony
- Service Web et API RESTful
- Architecture SPA et backend
- Bundle API Platform
- Gestion des relations entre entités
- Stratégie d'API RESTful

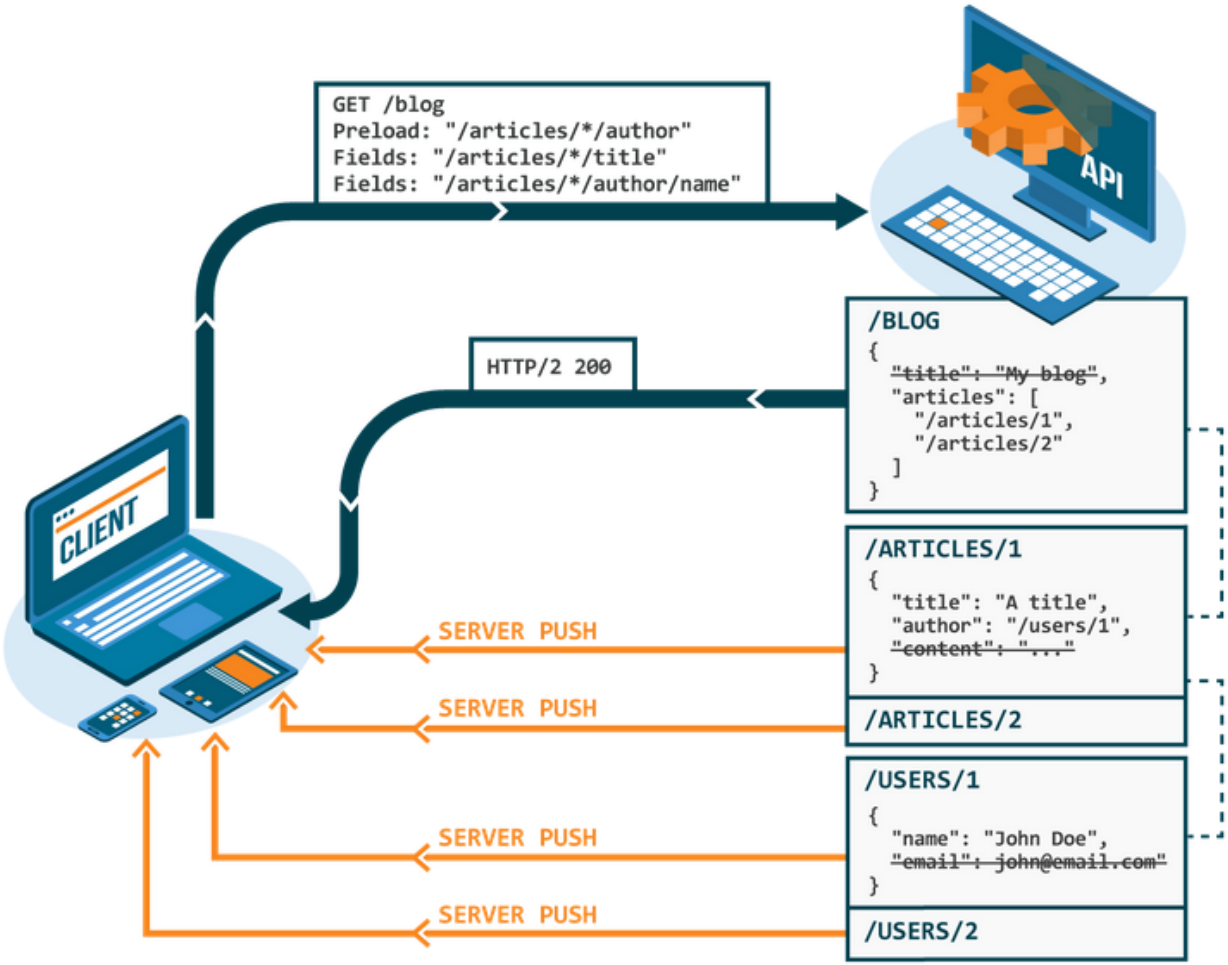
Stratégie d'API RESTful I

- La section précédente montrant la possibilité de construire des objets JSON complexes à partir d'entités du monde relationnel (SQL) ne constitue pas une solution efficace (jointures automatiques pouvant être coûteuses)
- Au contraire, il est préférable de récupérer la super-entité et ses IRI puis de requêter ces IRI de manière asynchrone par la suite : cela permettra à la SPA de commencer à afficher les propriétés propres tandis que les autres requêtes dont déjà lancées

Stratégie d'API RESTful II

- Une meilleure approche consiste à utiliser HTTP/2 et les requêtes PUSH permettant au web service d'envoyer automatiquement les sous-objets. Cela peut être réalisé avec l'annotation `ApiProperty(push: true)` sur la propriété référençant l'objet lié ou en utilisant Vulcain ...

Stratégie d'API RESTful III



Stratégie d'API RESTful IV

- Une autre stratégie consiste à n'utiliser qu'une table pour la super-entité (Evenement) et de stocker les sous-entités directement dans une colonne JSON nommée tournois ...
- On peut aussi utiliser MongoDb pour stocker une collection de documents Evenements ...

Plan

15 Partager son projet Symfony avec Git

Partager son projet Symfony avec Git I

- Git est un système de gestion de versions décentralisé universellement utilisé
- Des dépôts distants tels que GitHub et GitLab permettent de travailler collaborativement
- Dès la création d'un projet symfony, un certains nombre de fichiers `.gitignore` sont créés afin, par exemple, de ne pas suivre les versions des logiciels tiers (`/vendor`)
- L'échange de projet Symfony ne doit être effectué qu'à travers Git afin d'éviter des archives gigantesques
- Lors de l'importation git d'un projet Symfony, il faudra avant de l'utiliser recharger tous les logiciels tiers à l'aide de composer

Git : mode d'emploi création, exportation I

Suivent les commandes commentées permettant la création et l'exportation d'un projet symfony depuis le répertoire racine du projet :

- `$ git init`

  initialisation du projet git (création de `.git`)
- `$ git add .`
  `$ git commit -m "initialisation"`

  ajout récursif du répertoire courant sauf les fichiers ignorés puis validation avec un message de la première version du projet
- dans un navigateur Web, se rendre sur GitHub ou GitLab et créer un nouveau dépôt avec un nom proche de celui du projet Symfony !
- récupérer l'url du dépôt distant i.e. `git@github.com:mmeynard/SfProjetTournoi.git`
- Retourner sur la ligne de commande de votre projet et tapez :

Git : mode d'emploi création, exportation II

```
$ git remote add origin
↪  git@github.com:mmeynard/SfProjetTournoi.git
```

Définit le nom `origin` comme dépôt distant

- `$ git push --set-upstream origin master`

  exportation sur le dépôt distant de tous les fichiers appartenant à la validation initiale dans la branche `master`. Attention sous GitHub, renommer la branche principale (par défaut `main`) en `master` avant de pousser (push)
- Le contenu du dépôt est très peu volumineux (250Kio)

Git : mode d'emploi importation et chargement des paquetages tiers (/vendor) I

- Depuis un répertoire quelconque d'une machine quelconque :

  ```
  $ git clone git@github.com:mmeynard/SfProjetTournoi.git
  ```

  création d'un répertoire SfProjetTournoi contenant l'importation du tournoi (250 Kio)
- Pour charger les paquetages tiers :

  ```
  $ cd SfProjetTournoi/
  $ composer install
  ...
  ```

  Les bundles sont installés les uns après les autres grâce aux fichiers `composer.json`, `composer.lock` qui détienne la liste des bundles et de leurs versions (16 Mio)
- Il ne reste plus qu'à lancer le serveur :

  ```
  $ symfony server:start -d -port=8080; symfony open:local
  ```

Git : mode d'emploi importation et chargement des paquetages tiers (/vendor) II

- Par la suite, les 2 dépôts locaux peuvent évoluer en parallèle, chaque collaborateur effectuant des modifications sur sa partie
- Lorsque une nouvelle validation (commit) semble nécessaire, il suffira d'effectuer :

```
$ git commit -m "changement du contrôleur de tournoi"
$ git push
```

  la nouvelle version est disponible sur Github
- l'autre collaborateur devra "tirer" cette version avant de "pousser" ses propres changements

```
$ git commit -m "modif du template de tournoi"
$ git pull
$ git push
```

- Il est préférable de réaliser un git pull au début de chaque nouvelle session de travail !

Plan

16 Personnalisation des rendus de formulaires Symfony

Personnalisation des rendus de formulaires Symfony I

- Un formulaire Symfony `FormType` est un objet PHP intermédiaire entre une entité Doctrine et le formulaire `twig` qui va le visualiser
- On le crée généralement dans un contrôleur qui va ensuite appeler le moteur de rendu de twig
- Il est construit à l'aide d'une entité puis est constitué de propriétés ayant un nom correspondant à un attribut de l'entité et un type qui est une classe FooType correspondant au champ de formulaire HTML (TextType, SubmitType)

Personnalisation des rendus de formulaires Symfony II

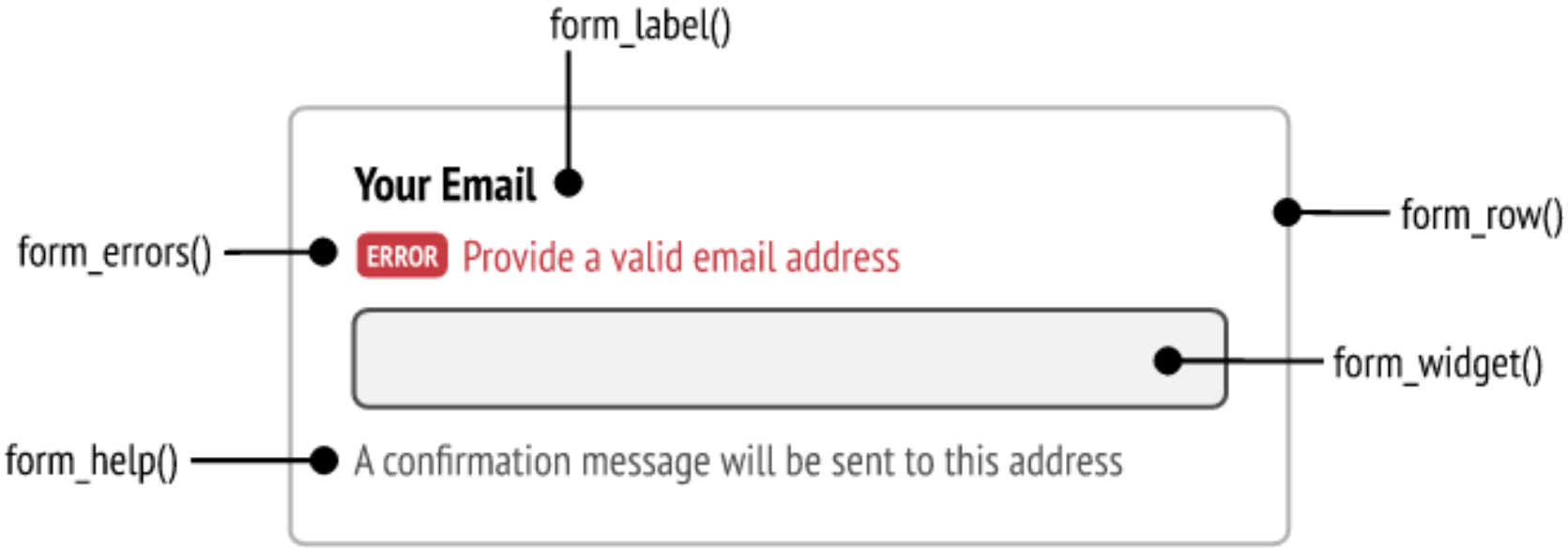
```php
<?php #[Route("/tournoi/saisieTnoi/")]
public function saisieTnoi($evtid): Response {
  $tnoi=new Tournoi();
  $tnoi->setNom("");
  $tnoi->setDescription("");
  $form = $this->createFormBuilder($tnoi)
    ->add('nom', TextType::class)
    ->add('description', TextType::class)
    ->add('sauver', SubmitType::class, ['label' => 'Sauvegarder
    ↪  !'])
    ->getForm(); // le formulaire est créé
  return $this->render('tournoi/saisieTnoi.html.twig', [
    'form' => $form->createView()
  ]);
}
```

Rendu personnalisé I

- Le template Twig le plus simple consiste à appeler une fonction Twig `form()` `{{ form(form) }}`
- Les champs de formulaire HTML seront disposés simplement les uns sous les autres
- Mais on peut(doit) utiliser d'autres fonctions Twig pour mettre en page le rendu des formulaires :
  `form_start()`, `form_end()`, `form_errors()`, `form_row()`
- `form_start()` génère la balise `form` et ses attributs (method, action)

Rendu personnalisé II

- `form_row(form.nom)` correspond à un regroupement de plusieurs fonctions concernant une propriété d'entité (`form_label`, `form_errors`, `form_help`, `form_widget`,)



- On peut utiliser la fonction `form_row()` ou bien aller dans le détail comme dans l'exemple suivant

Utiliser Bootstrap avec Symfony I

- Il suffit d'importer les fichiers css et js dans le fichier `templates/base.html.twig`
- Puis, chaque template Twig étendra ce template de base en redéfinissant les blocs `title` et `body`

```
<!DOCTYPE html><html><head> <!-- templates/base.html.twig -->
  <meta charset="UTF-8"><meta name="viewport"
↪   content="width=device-width, initial-scale=1">
  <title>{% block title %}Gestion de Tournoi{% endblock
↪   %}</title>
  {% block stylesheets %}
    <link
↪   href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/boo
↪   rel="stylesheet"
↪   integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3
↪   crossorigin="anonymous">
```

Utiliser Bootstrap avec Symfony II

```
  {% endblock %}
  {% block javascripts %}
      <script
↪   src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/boots
↪   integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuN
↪   crossorigin="anonymous"></script>
  {% endblock %}
</head>
<body>
  {% block body %}{% endblock %}
</body>
</html>
```

Suit une exemple de formulaire personnalisé :

Utiliser Bootstrap avec Symfony III

```
{# saisieTnoi.html.twig #}
{#    {{ form(form) }} rendu basique #}

{% extends 'base.html.twig' %}
{% block body %}
{{ form_start(form) }}
<div class="my-custom-class-for-errors">
  {{ form_errors(form) }}
</div>

<div class="row">
  <div class="col-md-6 text-end">
    {{ form_label(form.ev) }}
  </div>
  <div class="col-md-6">
    {{ form_widget(form.ev) }}
  </div>
```

Utiliser Bootstrap avec Symfony IV

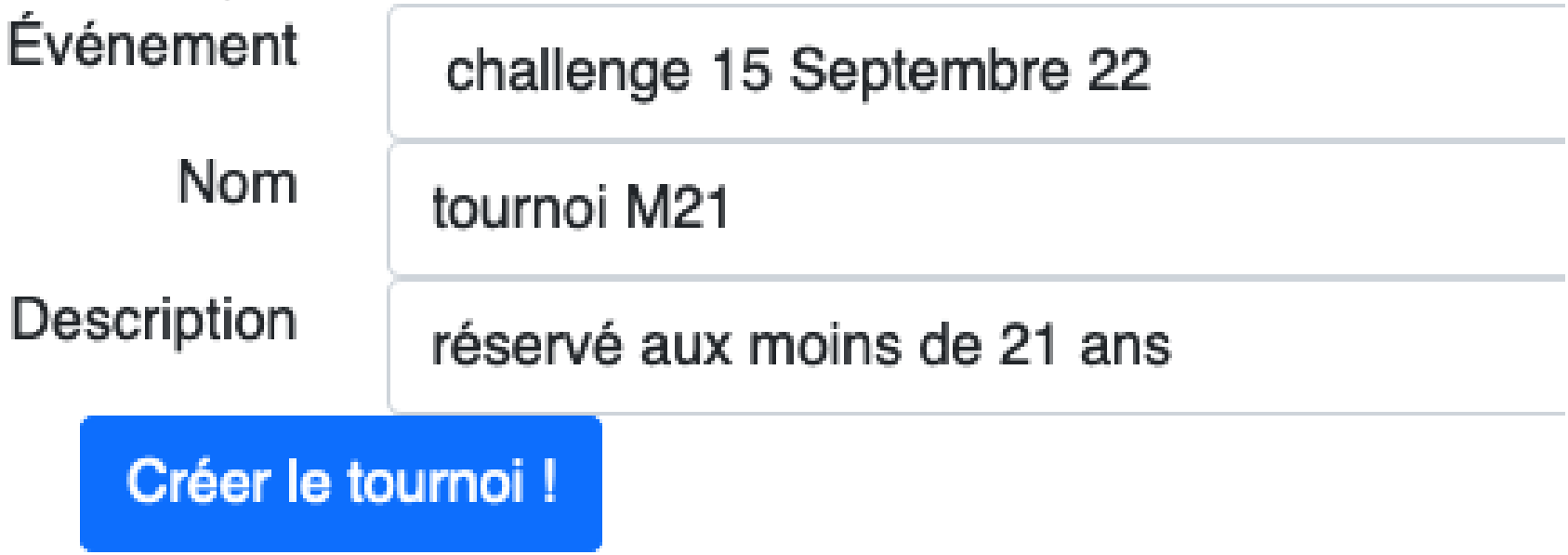
```
</div>

<div class="row">
  <div class="col-md-6 text-end">
    {{ form_label(form.nom) }}
  </div>
  <div class="col-md-6">
    {{ form_widget(form.nom) }}
  </div>
</div>

<div class="row">
  <div class="col-md-6 text-end">
    {{ form_label(form.description) }}
  </div>
  <div class="col-md-6">
    {{ form_widget(form.description) }}
```

Utiliser Bootstrap avec Symfony V

```
    </div>
</div>
<div class="row">
    <div class="col text-center">
        {{ form_widget(form.sauver) }}
    </div>
</div>
{{ form_end(form) }}

{% endblock %}
```

Utiliser Bootstrap avec Symfony VI

Le rendu graphique :



Les thèmes Twig I

- Le choix d'un thème global pour la mise en page est permise en utilisant un framework CSS tel que bootstrap
- Pour cela, il suffit de modifier le fichier de configuration : `config/packages/twig.yaml`

```
twig:
    ...
    form_themes: ['bootstrap_5_layout.html.twig']
```

Plan

17 La validation des formulaires et des entités

La validation des formulaires et des entités I

- La validation des données est réalisée en ajoutant des **contraintes** à des propriétés d'une classe
- Puis on appelle une méthode (`validate`) du service injecté grâce à `ValidatorInterface $validator` :
  `$errors = $validator->validate($tournoi);`
- Cette méthode retourne un tableau des erreurs qui est vide si l'objet est valide
- les contraintes peuvent être ajoutées sur l'entité (Assert) ou sur le formulaire dans l'entrée `"constraints"` du tableau qui est le 3ème paramètre de la méthode `add` du FormBuilder :

La validation des formulaires et des entités II

```php
<?php ...
$builder->add('pseudo', TextType::class, [
  'required' => true,
  'constraints' => [
    new Length(['min' => 3]),
    new Regex(['pattern' => '/^[a-z]+$/i',
      'message' => 'pseudo composé de minuscules'])
    ],
  ])
;
```

Dans cet exemple, l'entrée 'message' définit le message d'erreur personnalisé qui sera affiché dans le formulaire après non validation

Stratégies de liaison formulaire entité I

- La documentation propose de toujours associer un formulaire à une entité mais cela peut poser problème car une entité est toujours valide tandis qu'un formulaire pas forcément
- Certains auteurs préfèrent ajouter une couche intermédiaire (encore une) avec un objet PHP sur lequel repose le formulaire ...
- On choisira la première solution en s'autorisant à ajouter dans l'entité des propriétés nécessaires du formulaire (i.e. plainPassword) mais pas nécessaires dans la base de données !
- Ces propriétés ne seront pas persistantes : pas de `@ORM\Column(`
- Certaines propriétés de l'entité ne seront pas forcément utilisées dans le formulaire (i.e. password qui est crypté après la validation)

Stratégies de liaison formulaire entité II

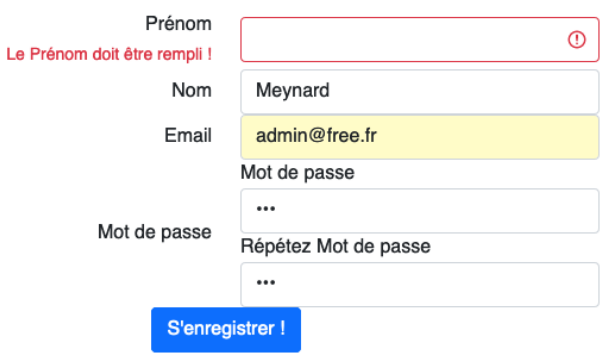
- Les contraintes de validité (NotBlank, ...) peuvent porter sur l'entité (`Assert\NotBlank`) **et/ou** sur le formulaire (clé `constraints` du 3e paramètre de `add(nom, classe, ['constraints' => [new Length(['min' => 3])]])`)
- Si c'est possible, on fera porter les contraintes sur les propriétés de l'entités et ces contraintes seront automatiquement transmises aux propriétés du formulaire

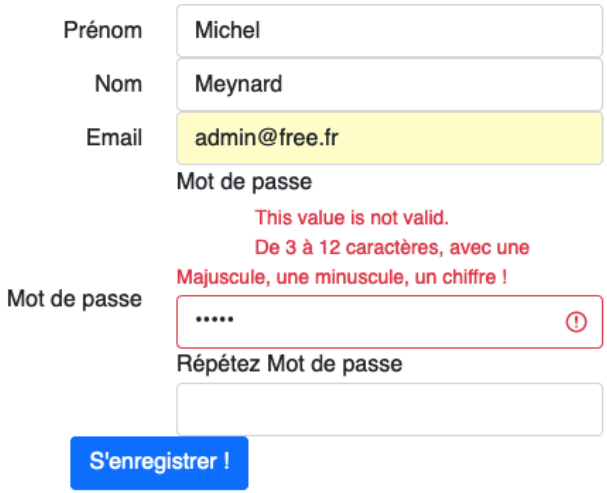
Un exemple de formulaire d'enregistrement (sign up) I

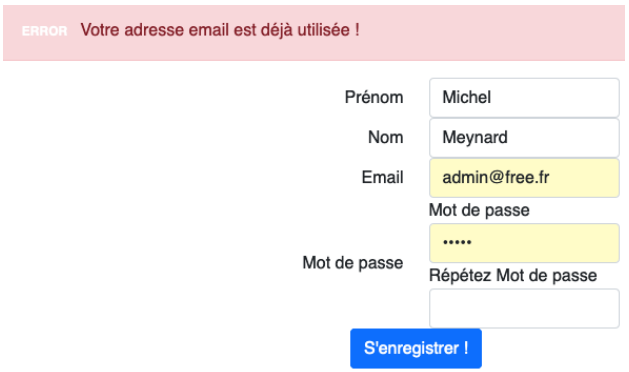
- On va modifier l'entité User afin de pouvoir accueillir de nouveaux gestionnaires de tournoi
- On va créer une nouvelle route (`/tournoi/register`) associée à une méthode `registe()` du contrôleur Tournoi
- On va créer un type de formulaire pour les utilisateurs souhaitant s'enregistrer et qui devront saisir 2 fois leur mot de passe
- La mise en page du template twig devra prendre en compte l'affichage des erreurs

Un exemple de formulaire d'enregistrement (sign up) II

3 cas d'invalidité : Prénom vide ; Passwd 1 Maj, 1 min, 1 chiffre ; email unique en BD







L'entité User I

```php
<?php /** @file App/Entity/User.php */
class User implements UserInterface{
  ...
  #[ORM\Column(type="string", length=180, unique=true)]
  #[Assert\Email(message = "L'adresse '{{ value }}' n'est pas une
  ↪  adresse email valide.")]

  private $email;

  #[var string The hashed password]
  #[ORM\Column(type="string")]
  #[Assert\NotBlank]

  private $password;

  #[Assert\Length(min=3,max=12 )]
```

L'entité User II

```
#[Assert\Regex("/[a-z]/")]
#[Assert\Regex("/[A-Z]/")]
#[Assert\Regex("/\d/", message = "De 3 à 12 caractères, avec
↪  une Majuscule, une minuscule, un chiffre !")]
/* Attention, propriété non liée à une colonne : mot de passe
↪  non crypté
 saisi utilisé par le form  */

private $plainPassword;

#[ORM\Column(type="string", length=30)]
#[Assert\NotBlank(message = "Le {{ label }} doit être rempli
↪  !")]

private $prenom;

#[ORM\Column(type="string", length=30)]
```

L'entité User III

```php
#[Assert\NotNull]

private $nom;
```

Le template twig userRegister.html.twig I

```
{% extends 'base.html.twig' %} {#
↪  Templates/tournoi/userRegister.html.twig #}
{% block body %}
{{ form_start(form) }}
<div class="my-custom-class-for-errors">
  {{ form_errors(form) }}
</div>

<div class="row d-flex align-items-center">
<div class="col-md-3"></div>
  <div class="col-md-3 text-end">
    {{ form_label(form.prenom) }}
  </div>
  <div class="col-md-3">
    {{ form_widget(form.prenom) }}
  </div>
```

Le template twig userRegister.html.twig II

```
</div>

<div class="row d-flex align-items-center">
  <div class="col-md-3"></div>
  <div class="col-md-3 text-end">
    {{ form_label(form.nom) }}
  </div>
  <div class="col-md-3">
    {{ form_widget(form.nom) }}
  </div>
</div>
<div class="row d-flex align-items-center">
  <div class="col-md-3"></div>
  <div class="col-md-3 text-end">
    {{ form_label(form.email) }}
  </div>
  <div class="col-md-3">
```

Le template twig userRegister.html.twig III

```
    {{ form_widget(form.email) }}
  </div>
</div>
<div class="row d-flex align-items-center">
  <div class="col-md-3"></div>
  <div class="col-md-3 text-end">
    {{ form_label(form.plainPassword) }}
  </div>
  <div class="col-md-3">
    {{ form_widget(form.plainPassword) }}
  </div>
</div>
<div class="row d-flex align-items-center">
  <div class="col text-center">
    {{ form_widget(form.sauver) }}
  </div>
```

Le template twig userRegister.html.twig IV

```
</div>
{{ form_end(form) }} {% endblock %}
```

La méthode du Contrôleur function register() I

```php
<?php /** Contrôleur App/Controller/TournoiController.php
#[Route("/tournoi/register", name="register")]
function register(Request $request, UserPasswordHasherInterface
↪  $passwordHasher, EntityManagerInterface $entityManager) :
↪  Response {
  $user=new User();
  $user->setPassword("bidon"); // Sinon la validation plante
  $form = $this->createForm(UserRegisterType::class, $user);
  $form->handleRequest($request);
  if ($form->isSubmitted() && $form->isValid()) { // persister
    $password = $passwordHasher->hashPassword($user,
    ↪  $user->getPlainPassword());
    $user->setPassword($password); // positionnement du passwd
    ↪  crypté
    $user->setRoles(["ROLE_GEST"]);
    $entityManager->persist($user);
```

La méthode du Contrôleur function register() II

```
    try {
      $entityManager->flush();
      return $this->redirectToRoute('tournois'); // REDIRECTION
    } catch(\Exception $e){ // Exception native de PHP
      $form->addError(new FormError("Votre adresse email est déjà
      ↪  utilisée !  " )); // . $e->getMessage()
    }
  }
  return $this->render('tournoi/userRegister.html.twig', [
    'form' => $form->createView()
  ]); // retour au formulaire !
}
```

Le Type de Formulaire UserRegisterType.php I

```php
<?php // App/Form/UserRegisterType.php
class UserRegisterType extends AbstractType{
  public function buildForm(FormBuilderInterface $builder, array
  ↪  $options){
    $builder
      ->add('email', EmailType::class)
      ->add('plainPassword', RepeatedType::class, array(
        'label' => 'Mot de passe',
        'type' => PasswordType::class,
        'first_options'  => array('label' => 'Mot de passe'),
        'second_options' => array('label' => 'Répétez Mot de
        ↪  passe')))
      ->add('prenom', TextType::class, array('label' =>
      ↪  'Prénom'))
      ->add('nom', TextType::class)
      ->add('sauver', SubmitType::class,
```

Le Type de Formulaire UserRegisterType.php II

```
            ['label' => "S'enregistrer !"])
        ;
    }
    public function configureOptions(OptionsResolver $resolver){
        $resolver->setDefaults([
            'data_class' => User::class,
        ]);
    }
}
```

Plan

18 Conclusion

Conclusion I

Objectifs de ce cours :

- introduction au framework Symfony
- utilisation du patron d'architecture Model View Controller
  - View réalisée grâce à Twig, un moteur de template HTML (`templates/Tournoi/listeTournois.twig`)
  - Model implémenté par les entités Doctrine (patron ORM) (instance de classe PHP associée à une ligne de table (`src/Entity/Tournoi.php`)) et les dépôts (repository d'entités associée à une table) (`src/Repository/TournoiRepository.php`)
  - Controller : il y a un contrôleur principal `/public/index.php`) qui prend en compte l'environnement du projet (`.env`), crée un noyau (kernel), construit l'objet Request puis le confie au noyau. Ce dernier, en fonction de la `route` utilisé, appelle la méthode correspondante de la classe contrôleur (`src/Controller/TournoiController.php`)
  - cette méthode retournera une Response sous une forme HTML via Twig ou bien du JSON

Conclusion II

Manques :

- nous n'avons regardé que les scripts côté serveur `backend` et rien sur les applications frontend SPA
- utiliser MongoDb (NoSQL) plutôt qu'un SGBD Relationnel
- des bundles supplémentaires permettant le test, le remplissage (fixtures) . . .)

Un avis personnel :

- documentation avec un seul exemple à chaque fois : vente de formation, certification
- hébergement Symfony avec SymfonyCloud sinon pas si simple ...

Plan

19 Bibliographie

Bibliographie I


[1] *Site de documentation de Symfony*, `https://symfony.com/doc/`, “Tout Symfony !”


[2] *Documentation officielle de Twig*, `https://twig.symfony.com/doc/3.x/tags`


[3] *Le manuel PHP en français*, `https://www.php.net/manual/fr/`, “La référence PHP ”


[4] *Un site de documentation sur les technologies du web*, `http://www.w3schools.com/`, “Très complet et en anglais (XHTML, CSS, JavaScript) ”


[5] *Documentation EasyAdmin bundle*, `https://symfony.com/bundles/EasyAdminBundle`, “Le site officiel pour CRUD”

Bibliographie II

 [6]*API Platform*, `https://api-platform.com/`, “Documentation du framework”