e-application - 2019

4. Plus loin avec Symfony4

Guillaume Kulakowski
Architecte Technique en API Management @CGI

Loïc Cariou
Architecte Technique en solutions open-source @CGI

https://twitter.com/llaumgui

https://www.linkedin.com/in/guillaumekulakowski
https://www.linkedin.com/in/loïc-cariou-93b8418a

guillaume@kulakowski.fr
cariou.loic.um2@gmail.com

https://blog.kulakowski.fr

CGI UNIVERSITÉ DE MONTPELLIER

Agenda

1. Sécurité
2. Performance
3. Formulaires
4. Validation
5. TP n°6
6. Encore plus loin...

Description du projet final

A rendre pour le :
03/01/2020
2 personnes max

Dans le cadre de ce cours et afin de mettre en pratique l'utilisation du framework Symfony4, nous allons réaliser un blog (par exemple celui de la communauté de communes du Groland : groblog).
Celui-ci devra être héberger sur la plateforme Cloud Heroku.

Le blog devra être :

- En HTLM5 (**sémantique** et **valide**). Utilisant un **framework CSS** (Bootstrap, pure, etc).

Le blog comprend 2 contrôleurs (Blog & Crud) et 5 routes (Action) :

- **IndexAction =>** page d'accueil (/) listant les x derniers billets (page à page),
- **PostAction =>** page billet (/post/$id) affichant le contenu d'un bille
- **NewAction =>** Ajout
- **EditAction =>** Edition
- **DeleteAction =>** Suppression

Le blog implémente **FosUser** pour se connecter (la page de login **doit être habillée**) et les routes CRUD doivent être protégées.
Le code source et le blog seront stockés dans le « cloud ».

Quoi rendre ?

Ajouter sur Heroku dans la rubrique « Access », les 2 collaborateurs :

- guillaume@kulakowski.fr
- cariou.loic.um2@gmail.com

- Ajouter dans votre projet un fichier README.md :
  - Nom & Prénom des participants au projet (**2 maximum**),
  - Compte et accès,
  - Commentaires,
  - Appréciation sur le cours pour l'année suivante (facultatif).

Si vous ne réussissez pas à déployer votre blog sur la plateforme Heroku **(Le déploiement sur heroku compte dans la note final)** vous pouvez nous l'envoyer par mail.

- Un fichier um_2019_NOM1_NOM2.tar.gz avec :
  - Sources (sans cache mais avec les vendors)
  - Dump SQL (MySQL) ou fichier SQLite.

1 – Sécurité

- La sécurité dans Sf4
- Les différentes attaques
- OWASP

Sécurité : Injection SQL

- Injection SQL : consiste à altérer une requête SQL pour en modifier l'impact :

```
SELECT uid FROM Users WHERE name = '(nom)' AND
password = '(mot de passe hashé)';
```

Devient :

```
SELECT uid FROM Users WHERE name = 'Dupont' -- ' AND
password = '4e383a1918b432a9bb7702f086c56596e';
```

La réponse de Symfony : utilisation de Doctrine dans les règles de l'art (*setParameter*).

Sécurité : XSS

Le cross-site scripting (abrégé *XSS*), est un type de faille de sécurité des sites web permettant d'injecter du contenu dans une page, permettant ainsi de provoquer des actions sur les navigateurs web visitant la page.

Les possibilités des XSS sont très larges puisque l'attaquant peut utiliser tous les langages pris en charge par le navigateur (JavaScript, Java, Flash...) et de nouvelles possibilités sont régulièrement découvertes notamment avec l'arrivée de nouvelles technologies comme HTML5.

Il est par exemple possible de rediriger vers un autre site pour de l'hameçonnage ou encore de voler la session en récupérant les cookies.

La réponse de Symfony :

- **Dans Twig :** l'échappement est **par défaut**. Pour le désactiver :

```
{{ article.body | raw }}
```

- **En PHP :** il y a une fonction *escape* dans la view :

```
Hello <?php echo $view->escape($name) ?>
```

Ne jamais faire confiance à une donnée de type user input !

Sécurité : CSRF

Les attaques de type Cross-Site Request Forgery (abrégées CSRF prononcées sea-surfing ou parfois XSRF) utilisent l'utilisateur comme déclencheur, celui-ci devient complice sans en être conscient. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés.

La réponse de Symfony : là encore, si Symfony est utilisé dans les règles de l'art en utilisant le composant « *form* », la protection et la génération de token est automatique.

Aller plus loin.

Plus loin dans la sécurité : OWASP

- Ne péchez jamais par orgueil ! Vérifiez que vous n'êtes vulnérable à aucune faille !
- Il existe des standards pour le web : OWASP, qui propose un TOP 10 des vulnérabilités.
- Il existe des outils : Zed Attack Proxy (ZAP) qui va se comporter comme un proxy entre vous et votre site afin de le tester. Le programme est en mesure également d'agir seul et de faire du « *brute force* ».
- Une faille de sécurité peut avoir des effets néfastes sur le business ou la réputation :
  - LinkedIn manque de sel et stock les mot de passe presque en clair
  - Parti politique qui autorise le « Directory indexes » sur son répertoire d'upload et laisse fuiter la liste de ses donateurs.

Un outil comme ZAP peut pourrir votre BD. **Pensez à monter un environnement de tests.**

2 – Performance

- L'OPCache
- Le cache utilisateur
- Le cache HTTP

Performance : OPCache / User cache

- Cache de Byte code : APC ou **Zend OpCache**.
- Cache utilisateur : APC ou **APCu** (en attendant YAC).
- Cache distribué pour session / cache : Memcached et le bundle Memcached.

Aller plus loin :
- Configurer les Sessions et les gestionnaires de sauvegarde
- Performance

Le cache HTTP

- Le moyen le plus efficace d'améliorer les performances d'une application est de mettre en cache l'intégralité d'une réponse pour ne plus avoir à rappeler l'application pour les requêtes suivantes. Bien sûr, ce n'est pas toujours possible pour les sites web fortement dynamiques, ou peut être que si...
- Le système de cache de Symfony4 est différent, car il se base sur la simplicité et la puissance du **cache HTTP** tel qu'il est défini dans la spécification HTTP. Au lieu de réinventer un processus de mise en cache, Symfony4 adopte la norme qui définit la communication de base sur le Web. Une fois que vous avez compris les fondamentaux de la validation HTTP et de l'expiration de la mise en cache, vous serez prêts à maîtriser le système de cache de Symfony2.

Le cache HTTP & la passerelle

- Une passerelle de cache, ou reverse proxy, est une couche indépendante qui se trouve devant votre application.
- La passerelle met en cache les réponses telles qu'elles sont retournées par l'application et répond aux requêtes avec les réponses qui sont en cache avant qu'elles n'atteignent l'application.
- Symfony4 possède sa propre passerelle par défaut.
- Mais n'importe quelle autre peut être également utilisée :
  - Varnish,
  - Squid.

Le cache HTTP & les entêtes

- Les entêtes du cache HTTP sont utilisés pour communiquer avec la passerelle de cache et tout autre cache entre votre application et le client. Symfony4 en propose par défaut et fournit une interface puissante pour interagir avec eux.
- HTTP définit quatre entêtes de cache :
  - Cache-Control
  - Expires
  - Etag
  - Last-Modified

```php
// ...

use Symfony\Component\HttpFoundation\Response;

$response = new Response();

// marque la réponse comme publique ou privée
$response->setPublic();
$response->setPrivate();

// définit l'âge max des caches privés ou des caches partagés
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

 // définit une directive personnalisée du Cache-Control
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Le cache HTTP & l'expiration

- L'expiration et la validation HTTP sont les deux modèles utilisés pour déterminer si le contenu d'un cache est valide (peut être réutilisé à partir du cache) ou périmé (doit être régénéré par l'application).
- Avec le modèle d'expiration, on spécifie simplement combien de temps une réponse doit être considérée comme « valide » en incluant un entête Cache-Control et/ou Expires. Les systèmes de cache qui supportent l'expiration enverront la même réponse jusqu'à ce que la version en cache soit expirée et devienne « invalide ».

```php
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

Le cache HTTP & la validation

- Quand une page est dynamique (c-a-d quand son contenu change souvent), le modèle de validation est souvent nécessaire. Avec ce modèle, le système de cache stocke la réponse, mais demande au serveur à chaque requête si la réponse est encore valide. L'application utilise un identifiant unique (l'entête Etag) et/ou un timestamp (l'entête Last-Modified) pour vérifier si la page a changé depuis sa mise en cache.

```php
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
    $response->setETag(md5($response->getContent()));
    $response->setPublic(); // permet de s'assurer que la réponse est publique, e
    $response->isNotModified($this->getRequest());

    return $response;
}
```

ESI et Hinclude

- Les Edge Side Includes (ESI) autorisent le cache HTTP à mettre en cache des fragments de pages (voir des fragments imbriqués) de façon indépendante. Avec les ESI, vous pouvez même mettre en cache une page entière pendant 60 minutes, mais un bloc imbriqué dans cette page uniquement 5 minutes.
- Les contrôleurs peuvent être imbriqués de façon asynchrone avec la bibliothèque javascript hinclude.js. Comme le contenu imbriqué vient d'une autre page (et d'un autre contrôleur), Symfony4 utilise le helper standard render pour configurer les tags hinclude.

Aller plus loin :
- Le Cache HTTP
- Contenu asynchrone avec hinclude.js
- Comment utiliser Varnish pour accélérer mon site Web

3 – Formulaires

- Création de formulaire
- Lien avec les entités
- Passage au templates

Le form FormBuilder en relation avec l'entité

- Symfony4 propose un « *FormBuilder* » qui, relié à une entité, va facilement vous permettre de construire un formulaire de création ou d'édition (CRUD).
- Prenons l'exemple de l'entité suivante :

```php
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Utiliser la méthode createFormBuilder

- Une fois l'entité correctement écrite, il ne reste plus qu'à invoquer la méthode *createFormBuilder()*,
- Puis de passer le formulaire à la vue,
- Et pour finir afficher le formulaire dans la vue via une fonction Twig :

```
{{ form(form) }}
```

```php
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // crée une tâche et lui donne quelques données par défaut pour cet exem
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->add('save', 'submit')
            ->getForm();

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

Aller plus loin

- Sf4 intègre nativement un grand nombre de types de champs.
- Le layout général des formulaires est personnalisable.
- Il est possible de faire des formulaires imbriqués.
- Le composant Form est intimement lié à Doctrine.
- La fonction Twig *form* vue précédemment peut également être découpée en plusieurs fonctions :

```
1  {# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}
2  {{ form_start(form) }}
3      {{ form_errors(form) }}
4
5      {{ form_row(form.task) }}
6      {{ form_row(form.dueDate) }}
7
8      <input type="submit" />
9  {{ form_end(form) }}
```

Aller plus loin : Formulaires.

4 – Validation

- La validation au sein des entités
- Validation au sein des formulaires

Le lien entité / validation

- Lors de la sauvegarde d'une entité en base, il est possible de contrôler que les informations renseignées sont correctes. C'est ce procédé que Symfony4 appelle la validation.
- Il existe plusieurs formats de validation (YAML, Annotations, XML & PHP). La validation étant liée à l'entité, je vous conseille fortement d'utiliser les annotation au sein de vos entités.

```php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Valider une entité

- Une fois l'entité créé, il devient simple de la vérifier à partir d'un contrôleur.
- Pour cela on utilise le service *validator*.

```php
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... do something to the $author object

    $validator = $this->get('validator');
    $errorList = $validator->validate($author);

    if (count($errorList) > 0) {
        return new Response(print_r($errorList, true));
    } else {
        return new Response('The author is valid! Yes!');
    }
}
```

Rediriger les erreurs dans l'IHM

- Il est également possible de rediriger les erreurs dans la vue pour les afficher :

```php
if (count($errorList) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errorList' => $errorList,
    ));
} else {
    // ...
}
```

```twig
{# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

<h3>The author has the following errors</h3>
<ul>
{% for error in errorList %}
    <li>{{ error.message }}</li>
{% endfor %}
</ul>
```

Validation au sein des formulaires

- Lorsque vous utilisez le composant formulaire pour générer la couche CRUD de votre entité, la validation de l'entité est automatiquement prise en compte.
- Vous n'avez rien de plus à faire !

Aller plus loin

- Sf4 intègre nativement un grand nombre de contraintes.
- Les messages d'erreurs sont traduisibles.
- Il est possible de configurer des groupes de validation.

Aller plus loin : Validation.

5 – TD n°6

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

Mon blog !

Maintenant que nous sommes identifiés au système (*FOSUserBundle*), et que le composant *Forms* n'a plus de secret pour nous, nous allons mettre en place la couche CRUD.

6 – Encore plus loin !

- Les tests dans Sf4
- La console
- Envoyer des mails
- Loguer avec monolog
- 42 bonnes pratiques pour Symfony2 (et 3)

Les tests

- L'une des force de Symfony est sa couverture de tests et le fait de pouvoir facilement tester nos applications via l'encapsulation du framework de tests PHP Unit.
- Sans surprise Sf4 permet la réalisation de tests unitaires.
- Mais la force de Symfony est d'embarquer un client permettant de faire des requêtes et par là des tests fonctionnels !

Aller plus loin : Les Tests.

La console

- Nous avons vu précédemment que des scripts PHP exécutés au travers d'Apache ou au travers d'un terminal (php-cli) bénéficient de limites différentes principalement sur :
  - max_execution_time
  - memory_limit
- Pour les traitements dit longs ou répétés (cron et tache planifiée), Symfony4 dispose d'une solution appelée Console.

Aller plus loin : Comment créer une commande pour la Console.

Envoyer des mails avec SwiftMailer

- Pour ce qui est de l'envoi de mail, Symfony4 utilise la librairie SwiftMailer.
- Cette librairie permet de s'interfacer avec un serveur SMTP.
- Elle permet d'envoyer des mails complexes avec pièces jointes très facilement.

Aller plus loin : Comment envoyer un Email.

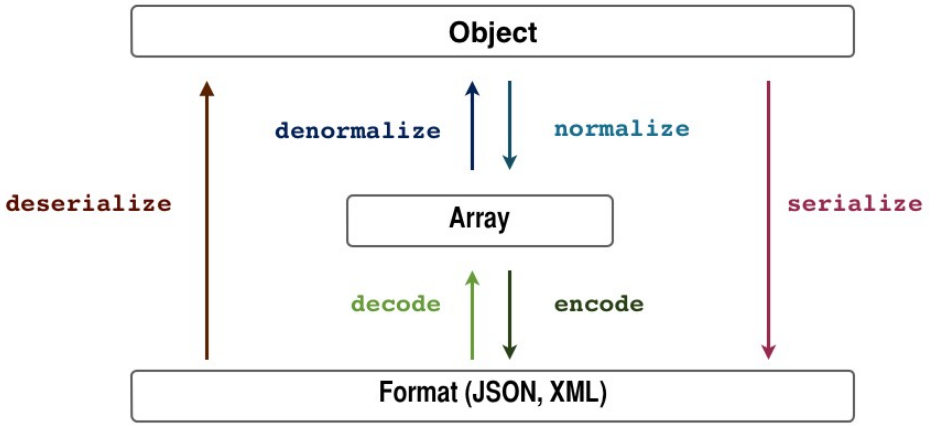
Monolog et les logs

- Symfony utilise la librairie monolog pour la gestion de ses logs.
- Cette solution est complètement paramétrable.
- Elle peut être configurée tant au niveau de la verbosité que de la sévérité.
- La rotation des logs est possible.
- L'envoi des informations de debug à FirePHP ou ChromePHP est également possible.

Aller plus loin : Comment utiliser Monolog pour écrire des Logs.

Serializer

- Un service (désactivé par défaut) pour *serialiser*.
- Possibilité de normaliser.
- Possibilité de convertir (ex : camel_case_to_snake_case).
- Possibilité de créer ses propres encodeurs.
- Possibilité de mettre en cache (APCu).



Aller plus loin : Serializer.

Les événements

- Symfony offre la possibilité d'écouter (*listener*) ou de souscrire (*subscriber*) à des évènements (*event*).
- Symfony défini des événements dans le kernel : http://api.symfony.com/3.1/Symfony/Component/HttpKernel/KernelEvents.html
- Mais il est également possible de créer ces propres évènements.

Aller plus loin : Event dispatcher.

Bonne pratiques Sf4

Pour finir, je vous conseille cette lecture :

Bonnes pratiques pour Symfony 4

Avec un focus sur

- Ne créez pas de multiple bundle pour structurer la logique de votre application,
- Utilisez les paramètres d'environnements,
- N'oubliez pas le fichier .dist,
- Stockez vos assets dans le répertoire web.

Annexes

Guide de survit de bin/console

- Vider le cache :

```
bin/console cache:clear
```

- Générer mes getter et setter de mes entities :

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

Webographie

- Site officiel de Symfony : http://symfony.com/
- Le manuel PHP : http://php.net/manual/fr/index.php
- La doc : http://symfony.com/doc/current/book/index.html

Bibliographie

- Toutes la documentation et les « books » officiels sont librement téléchargeables
  http://symfony.com/doc/current/index.html

Licence

La documentation Symfony4 étant bien faite et sous Licence libre, des illustrations tout comme des bouts de textes de ce cours en sont extraits.