[Partie 2] - Page de contact : validateurs, formulaires et envoi d'emails.

> Je propose également des formations en petits groupes sur 2 à 3 jours, plus d'infos sur la page dédiée. N'hésitez pas à me contacter (06.62.28.01.87 ou clement [@] keiruaprod.fr) pour en discuter !

Introduction

Maintenant, nous avons mis en place le template HTML de base. Il est désormais temps de créer une des pages fonctionnelles. Nous allons commencer avec une des pages les plus simples, la page de contact. A la fin de ce chapitre, nous aurons une page de contact qui permet aux utilisateurs d'envoyer des messages au webmaster. Ces requêtes lui seront envoyées par email.

Les thèmes suivants seront abordés au cours de ce chapitre :

1. Les validateurs
2. Les formulaires
3. La mise en place de paramètres de configuration pour un bundle

La page de contact

Routage

Comme avec la page `A propos` que nous avons créée dans le chapitre précédent, nous allons commencer par définir la route vers la page de contact. Ouvrez le fichier de routage du `BloggerBlogBundle` situé dans `src/Blogger/BlogBundle/Resources/config/routing.yml` et ajoutez-y la règle de routage suivante :

```
# src/Blogger/BlogBundle/Resources/config/routing.yml
BloggerBlogBundle_contact:
    pattern:  /contact
    defaults: { _controller: BloggerBlogBundle:Page:contact }
    requirements:
        _method:  GET
```

Il n'y a rien de nouveau ici, la règle associe le lien `/contact``pour la méthode HTTP ``GET` et exécute l'action `contact` du contrôleur `Page` dans le `BloggerBlogBundle`.

Contrôleur

Ensuite, ajoutons l'action pour la page de contact dans le contrôleur `Page` du `BloggerBlogBundle` situé dans `src/Blogger/BlogBundle/Controller/PageController.php`.

```
// src/Blogger/BlogBundle/Controller/PageController.php
// ..
public function contactAction()
{
    return $this->render('BloggerBlogBundle:Page:contact.html.twig');
}
// ..
```

Pour le moment cette action est très simple, elle affiche simplement la vue de la page de contact. Nous reviendrons sur ce contrôleur plus tard.

Vue

Créez la vue pour la page de contact dans `src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig` et ajoutez-y le contenu suivant.

```
{# src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig #}
{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block title %}Contact{% endblock%}
```

```
{% block body %}
    <header>
        <h1>Contact symblog</h1>
    </header>

    <p>Want to contact symblog?</p>
{% endblock %}
```

Ce template est également très simple. Il étend la mise en page du template de `BloggerBlogBundle` remplace le bloc de titre pour un titre personnalisé et définit du contenu pour le bloc `body`.

Lien vers la page

Finalement nous devons mettre à jour le lien dans le template de l'application, situé dans `app/Resources/views/base.html.twig` pour créer le lien vers la page de contact.

```
<!-- app/Resources/views/base.html.twig -->
{% block navigation %}
    <nav>
        <ul class="navigation">
            <li><a href="{{ path('BloggerBlogBundle_homepage') }}">Home</a></li>
            <li><a href="{{ path('BloggerBlogBundle_about') }}">About</a></li>
            <li><a href="{{ path('BloggerBlogBundle_contact') }}">Contact</a></li>
        </ul>
    </nav>
{% endblock %}
```

Si vous vous rendez avec votre navigateur à l'adresse `http://symblog.dev/app_dev.php/` et cliquez sur le lien vers la page de contact dans la barre de navigation, vous devriez voir une page de contact très simple. Maintenant que la page est correctement mise en place, il est temps de commencer à travailler sur le formulaire de contact. C'est découpé en 2 parties distinctes: le validateur et le formulaire. Avant de parler de ces deux sujets, il faut réfléchir à la manière dont nous allons gérer les données du formulaire de contact.

L'entité Contact

Commençons par créer une classe qui représente une requête de contact par un utilisateur. Nous voulons récupérer des informations de base telles que le nom, le sujet de la requête ainsi que le message que l'utilisateur souhaite envoyer. Créez un nouveau fichier dans `src/Blogger/BlogBundle/Entity/Enquiry.php` et collez-y le contenu suivant.

```
<?php
// src/Blogger/BlogBundle/Entity/Enquiry.php

namespace Blogger\BlogBundle\Entity;

class Enquiry
{
    protected $name;

    protected $email;

    protected $subject;

    protected $body;

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }

    public function getEmail()
    {
        return $this->email;
    }

    public function setEmail($email)
    {
        $this->email = $email;
    }

    public function getSubject()
    {
        return $this->subject;
    }

    public function setSubject($subject)
    {
```

```
        $this->subject = $subject;
    }

    public function getBody()
    {
        return $this->body;
    }

    public function setBody($body)
    {
        $this->body = $body;
    }
}
```

Comme vous pouvez le voir cette classe définit simplement quelques membres protégés ainsi que leurs accesseurs. Rien ici n'indique comment valider les données, ni comment ces données sont liées aux éléments du formulaire. Nous reviendrons là dessus plus tard.

> **Note**
>
> Faisons une petite digression pour parler de l'utilisation des espaces de nom dans Symfony2. La classe d'entité que nous avons créée est définie par l'espace de nom `Blogger\BlogBundle\Entity`. Comme le chargement automatique de Symfony2 supporte le standard PSR-0, l'espace de nom reflète directement la structure de répertoires du bundle. La classe d'entité `Enquiry` est située dans `src/Blogger/BlogBundle/Entity/Enquiry.php`, ce qui permet à Symfony2 de charger cette classe automatiquement de manière correcte.
>
> Comment le chargeur automatique de Symfony2 sait que l'espace de nom `Blogger` se trouve dans le répertoire `src` ? C'est grâce à la configuration du chargement automatique dans `app/autoloader.php`
>
> ```
> // app/autoloader.php
> $loader->registerNamespaceFallbacks(array(
>     __DIR__.'/../src',
> ));
> ```
>
> Cette ligne de code enregistre les répertoires à utiliser pour tous les espaces de noms qui ne sont pas enregistrés. Comme l'espace de nom `Blogger` n'est pas enregistré, le chargement automatique des classes de Symfony2 va chercher les fichiers requis dans le répertoire `src`
>
> Le chargement automatique et les espaces de nom sont des concepts très puissant dans Symfony2. Si vous avez des problèmes dans lesquels PHP n'arrivent pas à trouver une ou plusieurs classes, il y a des chances pour qu'il y ait des erreurs dans votre espace de nom ou dans votre structure de répertoires. Vérifiez également que les espaces de nom soient bien enregistrés dans le chargeur comme vu au dessus. Vous ne devriez jamais être tentés de `réparer` celà en utilisant les directives PHP `require` ou `include`.

Formulaires

Ensuite, nous allons créer le formulaire. Symfony2 est livré avec une librairie de formulaires très puissante qui rend très facile le fait de travailler avec les formulaires. Comme avec tous les autres composants de Symfony2, le composant de formulaire peut être utilisé à l'extérieur de Symfony2 pour vos propores projets. La source du composant de formulaire est disponible sur Github. Nous allons commencer par créer une classe `AbstractType` qui représente le formulaire de contact. Nous aurions pu créer le formulaire directement dans le contrôleur et ne pas nous embêter avec cette classe, néanmoins séparer le formulaire dans sa propre classe nous permet de réutiliser le formulaire dans l'application. Cela nous évite également d'encombrer le contrôleur, car après tout, il est censé être simple : son but est de faire le lien entre le modèle et la vue.

EnquiryType

Créez un nouveau fichier dans `src/Blogger/BlogBundle/Form/EnquiryType.php` et collez-y le contenu suivant.

```
<?php
// src/Blogger/BlogBundle/Form/EnquiryType.php

namespace Blogger\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class EnquiryType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
        $builder->add('email', 'email');
        $builder->add('subject');
        $builder->add('body', 'textarea');
    }

    public function getName()
    {
        return 'contact';
    }
}
```

La classe `EnquiryType` nous permet de présenter la classe `FormBuilder`. la classe `FormBuilder` est votre meilleur atout lorsqu'il est question de créer des formulaires. Elle est capable de simplifier le processus de définition des champs à partir des métadonnées qu'un champ possède. Comme notre entité est très simple, nous n'avons défini aucune métadonnée, donc le `FormBuilder` va créer des champs de texte par défaut. C'est adapté à la plupart des champs, sauf pour le corps du message pour lequel nous souhaitons utiliser une `textarea`, et pour l'adresse email pour laquelle nous allons utiliser le nouveau champ d'adresse email proposé par l'HTML5.

> **Note**
>
> Un aspect clé à prendre en compte est le fait que la méthode `getName` doit renvoyer un identifiant unique.

Créer le formulaire dans le contrôleur

Nous avons désormais défini l'entité `Enquiry` et la classe `EnquiryType`, nous pouvons désormais mettre à jour l'action pour la page de contact afin de s'en servir. Remplacez le contenu de la méthode contactAction dans le fichier `src/Blogger/BlogBundle/Controller/PageController.php` par le suivant :

```
// src/Blogger/BlogBundle/Controller/PageController.php
public function contactAction()
{
    $enquiry = new Enquiry();
    $form = $this->createForm(new EnquiryType(), $enquiry);

    $request = $this->getRequest();
    if ($request->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // Perform some action, such as sending an email

            // Redirect - This is important to prevent users re-posting
            // the form if they refresh the page
            return $this->redirect($this->generateUrl('BloggerBlogBundle_contact'));
        }
    }

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

Nous commençons par créer une instance de l'entité `Enquiry`. Cette entité représente les données d'un message sur la page de contact. Nous créons ensuite le formulaire correspondant: nous spécifions le type `EnquiryType` créé précédemment, et passons en paramètres notre object entité $enquiry. La méthode `createForm` est capable d'utiliser ces 2 patrons pour créer la représentation d'un formulaire.

Comme cette action du contrôleur va maintenant s'occuper d'afficher et traiter le formulaire qui lui est soumis, nous devons faire attention à la méthode HTTP utilisée. Les formulaires soumis sont généralement envoyés via la méthode `POST`, et notre formulaire n'y fera pas exception. Si la requête est de type `POST`, un appel à la méthode `bindRequest` va transformer les données soumises pour les associées à notre objet `$enquiry`. A ce moment-là, l'objet `$enquiry` contiendra une représentation de ce que l'utilisateur aura envoyé.

Nous vérifions ensuite que le formulaire est valide. Comme nous n'avons pas précisé de validateurs pour le moment, le formulaire sera toujours valide.

Enfin, nous précisons le template à utiliser pour l'affichage. Notez que nous passons également à la vue une représentation du formulaire à afficher, ce qui nous permet d'effectuer l'affichage adéquat dans la vue.

Comme nous avons utilisé 2 nouvelles classes dans notre contrôleur, nous devons importer les espaces de nom correspondants. Mettez à jour le début du fichier `src/Blogger/BlogBundle/Controller/PageController.php` avec le contenu suivant.

```
<?php
// src/Blogger/BlogBundle/Controller/PageController.php

namespace Blogger\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
// Import new namespaces
use Blogger\BlogBundle\Entity\Enquiry;
use Blogger\BlogBundle\Form\EnquiryType;

class PageController extends Controller
// ..
```

Affichage du formulaire

Grâce à Twig, l'affichage de formulaires est très simple. Twig propose en effet un système par couches pour l'affichage de formulaires qui permet soit d'afficher un formulaire comme une unique entité, soit comme des éléments individuels, selon le besoin de personnalisation nécessaire.

Afin de démontrer la puissance des méthodes de Twig, nous allons utiliser le bout de code suivante pour afficher le formulaire entier :

```
<form action="{{ path('BloggerBlogBundle_contact') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

Bien que cette méthode soit utile et très simple durant la phase de prototypage, cela s'avère limité lorsque le besoin de personnalisation est important, ce qui est souvent le cas avec les formulaires.

Pour notre formulaire de contacts, nous allons opter pour un compromis. Remplacez le code du template `src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig` par le suivant :

```
{# src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig #}
{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block title %}Contact{% endblock%}

{% block body %}
    <header>
        <h1>Contact symblog</h1>
    </header>

    <p>Want to contact symblog?</p>

    <form action="{{ path('BloggerBlogBundle_contact') }}" method="post" {{ form_enctype(form) }} class="blogger">
        {{ form_errors(form) }}

        {{ form_row(form.name) }}
        {{ form_row(form.email) }}
        {{ form_row(form.subject) }}
        {{ form_row(form.body) }}

        {{ form_rest(form) }}

        <input type="submit" value="Submit" />
    </form>
{% endblock %}
```

Comme vous pouvez le voir, nous utilisons 4 nouvelles fonctions Twig pour afficher notre formulaire.

La première fonction `form_enctype` définit le type de contenu du formulaire. C'est nécessaire lorsqu'un formulaire traite avec des fichiers uploadés. Ce n'est donc pas nécessaire pour le moment, mais c'est une bonne habitude que de l'utiliser pour tous les formulaires au cas où l'upload de fichier soit ajouté dans le futur. Déboguer un formulaire qui traite de l'upload de fichier dans lequel le type de contenu n'est pas spécifié peut être un vrai casse tête !

La seconde fonction `form_errors` affichera les erreurs du formulaire dans le cas où la validation échoie.

La 3ème fonction `form_row` affiche les éléments liés à un champ. Cela comporte toutes les erreurs associées au champ, l'étiquette liée au champ ainsi que l'élément du champ de formulaire lui même.

Enfin, nous utilisons la fonction `form_rest`. C'est toujours une bonne habitude d'utiliser cette fonction à la fin de l'affichage pour afficher les champs qui auraient pû être oubliés, ce qui inclue les champs cachés ainsi que le jeton CSRF de Symfony2.

> **Note**
>
> Les attaques de type Cross-site request forgery (CSRF) sont expliquées en détail dans le chapitre sur les formulaires du livre Symfony2.

Donner du style au formulaire.

Si vous regardez maintenant notre formulaire via la page `http://symblog.dev/app_dev.php/contact`, vous remarquerez qu'il n'est pas très engageant. Ajoutons lui du style pour améliorer son rendu. Comme les styles sont spécifiques à notre bundle de blog, nous allons les créer dans une feuille de style à l'intérieur même du bundle. Créez un nouveau fichier dans `src/Blogger/BlogBundle/Resources/public/css/blog.css` et copiez-y le contenu suivant :

```
.blogger-notice { text-align: center; padding: 10px; background: #DFF2BF; border: 1px solid; color: #4F8A10; margin-bottom: 10px; }
form.blogger { font-size: 16px; }
form.blogger div { clear: left; margin-bottom: 10px; }
form.blogger label { float: left; margin-right: 10px; text-align: right; width: 100px; font-weight: bold; vertical-align: top; padding-top: 10px; }
form.blogger input[type="text"],
form.blogger input[type="email"]
    { width: 500px; line-height: 26px; font-size: 20px; min-height: 26px; }
form.blogger textarea { width: 500px; height: 150px; line-height: 26px; font-size: 20px; }
form.blogger input[type="submit"] { margin-left: 110px; width: 508px; line-height: 26px; font-size: 20px; min-height: 26px; }
form.blogger ul li { color: #ff0000; margin-bottom: 5px; }
```

Nous devons informer l'application que nous souhaitons utiliser cette feuille de style. Nous pourrions importer la feuille de style dans le template de la page de contact, mais comme d'autres templates pourraient utiliser cette feuille de style par la suite, cela a plus de sens de l'importer dans le `layout` que nous avons créé pour notre `BloggerBlogBundle` du chapitre 1. Ouvrez ce fichier, situé dans `src/Blogger/BlogBundle/Resources/views/layout.html.twig` et mettez-y le contenu suivant :

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}
    <link href="{{ asset('bundles/bloggerblog/css/blog.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{% block sidebar %}
    Sidebar content
{% endblock %}
```

Remarquez que nous avons défini un bloc `stylesheets` qui remplace le bloc défini dans le template parent. Il est important de remarquer l'appel à la fonction `parent`. Cette fonction se charge d'importer le contenu du bloc dans le template parent `app/Resources/base.html.twig`, dans le cas présent celui s'occupant des feuilles de style, cela nous permet d'ajouter nos nouvelles feuilles de style à la suite. Nous ne voulons pas supprimer les feuilles de style existantes.

Afin que la fonction `asset` fasse les bons liens avec les ressources, nous devons copier ou déplacer les ressources du bundle dans le répertoire `web` de l'application. Cela peut être fait par la commande suivante:

```
$ php app/console assets:install web --symlink
```

> **Note**
>
> Si vous utilisez un système d'exploitation qui ne supporte pas les liens symboliques (symlink), tel que Windows, vous devez supprimer l'option symlink comme ceci :
>
> ```
> php app/console assets:install web
> ```
>
> Cette méthode va en fait copier les ressources présentes dans le répertoire `public` des bundles dans le répertoire `web` de l'application. Comme les fichiers sont copiés, il est nécessaire de lancer cette commande à chaque fois que vous faites une modification dans les ressources publiques utilisées par un bundle.

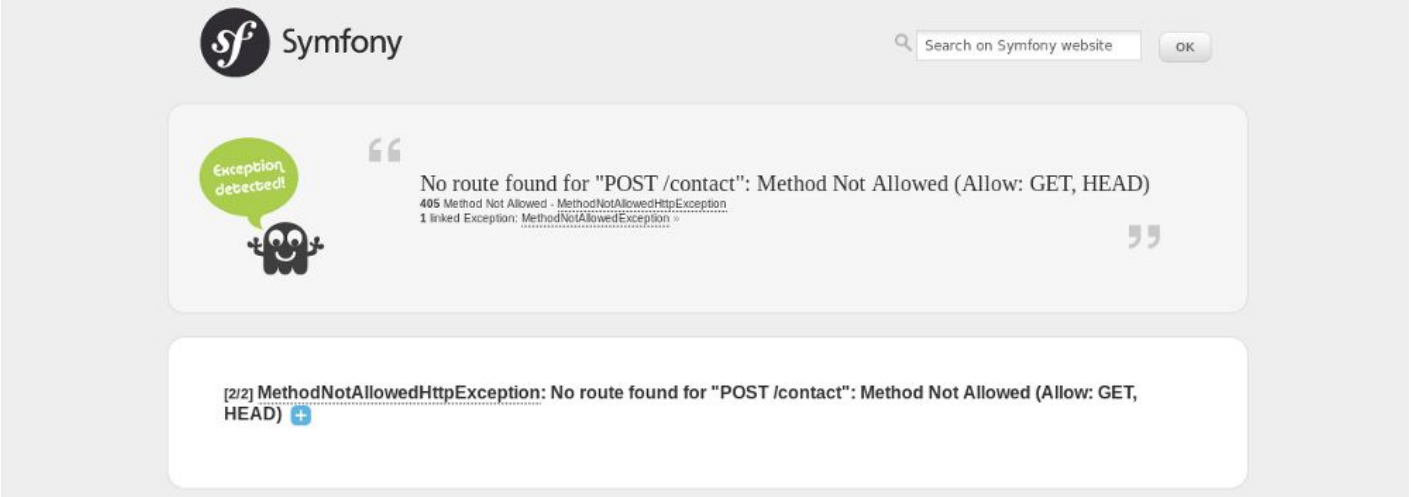
Vous pouvez maintenant rafraîchir la page de contact, dans laquelle le nouveau style s'applique au formulaire. C'est quand même un peu plus sympa comme ça non ?

> **Tip**

Alors que la fonction `asset` nous permet d'utiliser les ressources, il y a une meilleure alternative pour cette opération, il s'agit d'Assectic. Cette librairie, écrite par Kris Wallsmith est fournie par défaut avec la distribution standard de Symfony2. Elle permet une gestion des ressources bien meilleure que celle proposée par Symfony2. Assetic permet de lancer des filtres sur les fichiers pour les fusionner automatiquement, les alléger ou les compresser. Elle permet également de lancer des filtres de compression sur les images. Enfin Assetic nous permet de faire référence à des ressources directement à l'intérieur des répertoires publics des bundles, sans avoir à lancer la commande `assets:install`. Nous reviendrons plus en détail sur ce sujet dans un chapitre ultérieur.

Echec à la soumission

Les plus enthousiastes parmi vous auront probablement déjà essayé de soumettre le formulaire, et seront tombés sur une erreur de Symfony2.



Ce message d'erreur nous indique qu'il n'y a pas de route correspondante à l'adresse `/contact` pour la méthode HTTP POST. La route que nous avons définie n'accepte que les requêtes de type GET et HEAD, car nous l'avons configurée comme cela.

Mettons à jour notre route de contact dans `src/Blogger/BlogBundle/Resources/config/routing.yml` afin de permettre également les requêtes de type POST.

```
# src/Blogger/BlogBundle/Resources/config/routing.yml
BloggerBlogBundle_contact:
    pattern:  /contact
    defaults: { _controller: BloggerBlogBundle:Page:contact }
    requirements:
        _method:  GET|POST
```

> **Tip**
>
> Vous vous demandez probablement pourquoi la route acceptait les requêtes de type HEAD, alors que seul GET avait été spécifié. Et bien parce qu'une requête HEAD est de type GET où seules les en-têtes HTTP sont retournées.

Maintenant vous pouvez soumettre le formulaire qui devrait fonctionner comme attendu, bien qu'il ne fasse rien de particulier pour le moment. La page redirige simplement à nouveau vers le formulaire de contact.

Validateurs.

Les validateurs de Symfony2 permettent de valider les données. La validation est une tâche courante lorsqu'il est question de valider les données de formulaire. Cette tâche doit être réalisée avant que les données ne soient envoyées vers une base de données. Les validateurs Symfony2 nous permettent de séparer notre logique de validation des composants qui pourraient s'en servir, tels que les composants de formulaire ou de base de donnée. Cette approche signifie que nous allons avoir un jeu de règles de validation par objet.

Commençons par mettre à jour l'entité `Enquiry` dans `src/Blogger/BlogBundle/Entity/Enquiry.php` pour spécifier quelques validateurs. N'oubliez pas d'ajouter les 5 nouvelles déclarations avec `use` au début du fichier.

```
<?php
// src/Blogger/BlogBundle/Entity/Enquiry.php

namespace Blogger\BlogBundle\Entity;

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\MaxLength;

class Enquiry
{
    // ..
```

```
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('name', new NotBlank());

        $metadata->addPropertyConstraint('email', new Email());

        $metadata->addPropertyConstraint('subject', new NotBlank());
        $metadata->addPropertyConstraint('subject', new MaxLength(50));

        $metadata->addPropertyConstraint('body', new MinLength(50));
    }

    // ..

}
```

Afin de définir les validateurs, nous devons implémenter la méthode statique `loadValidatorMetadata`, qui nous fournit un objet de type `ClassMetadata`. Nous pouvons utiliser cet objet pour définir des contraintes sur les propriétés de nos entités membres. La première ligne applique la contrainte `NotBlank` à la propriété `name`. Les validateurs sont aussi simples qu'ils y paraissent: celui-ci se contente de renvoyer vrai si la valeur à valider n'est pas vide. Nous mettons ensuite en place la validation pour le champ `email`. Le système de validation nous fournit en effet une règle de validation pour ce type de champ, qui va même vérifier le MX record (équivalent en gros du DNS, mais pour les adresses mail) afin de s'assurer que le domaine est valide. Pour l'attribut `subject`, nous voulons à la fois nous assurer que le champ n'est pas vide et qu'il ne dépasse pas une taille maximale, ce qui est fait avec les contraintes `NotBlank` et `MaxLength`: il est en effet possible d'appliquer autant de règles de validations qu'on le souhaite.

Une liste complète des contraintes de validation est disponible dans les documents de référence de Symfony2. Il est également possible de créer des règles de validation personnalisées.

Vous pouvez maintenant soumettre le formulaire de contact, et les données soumises passent dans les contraintes de validation: essayez de mettre une adresse email invalide. Vous devriez voir un message d'erreur qui vous informe que l'addresse n'est pas valide. Chaque validateur propose un message par défaut qui peut être remplacé si nécessaire. Pour changer le message par défaut du validateur du champ email, vous pouvez par exemple faire :

```
$metadata->addPropertyConstraint('email', new Email(array(
    'message' => 'symblog does not like invalid emails. Give me a real one!'
)));
```

> **Tip**
>
> Si vous utilisez un navigateur qui supporte le HTML5 (il y a de grandes chances pour que ce soit le cas), des messages HTML5 vont apparaître pour vous faire respecter certaines contraintes à partir des métadonnées de votre objet `Entity`. Vous pouvez le voir pour l'élément email, dont le code HTML est le suivant :
>
> ```
> <input type="email" value="" required="required" name="contact[email]" id="contact_email">
> ```
>
> Il utilise un des nouveaux champs HTML5, dont l'attribut `required` est défini. La validation côté client est bien dans le sens où elle ne nécessite pas un aller-retour avec le serveur pour valider le formulaire, néanmoins elle ne devrait pas être utilisée seule. Vous devriez toujours valider les données côté serveur, car il est très facile pour un utilisateur de passer outre la validation côté client.

Envoyer l'email

Bien que notre formulaire nous permettre actuellement de soumettre des requêtes, rien ne leur arrive réellement pour le moment. Mettons à jour le contrôleur afin d'envoyer un email au webmaster du blog. Symfony2 est livré avec la librairie d'envoi d'email Swift Mailer. Il s'agit d'une librairie très puissante, nous allons seulement effleurer la surface de ce qu'il est possible de faire avec.

Configurer les paramètres de Swift Mailer

Swift Mailer est déjà configurée de base dans la distribution standard de Symfony2, néanmoins nous devons configurer quelques paramètres concernant la méthode d'envoi, et lui fournir les accréditations nécessaires pour réaliser cette tâche. Ouvrez le fichier de paramètres dans `app/parameters.ini` et trouvez les paramètres préfixés par `mailer_`.

```
mailer_transport="smtp"
mailer_host="localhost"
mailer_user=""
mailer_password=""
```

Swift Mailer propose un certain nombre de méthodes pour envoyer les emails, entre autre l'utilisation d'un serveur SMTP, l'installation locale de sendmail, ou même l'utilisation d'un compte GMail. Par souci de simplicité, c'est cette dernière méthode que nous allons utiliser. Mettez à jour les paramètres comme suit, en remplaçant votre nom d'utilisateur (username) et votre mot de passe (password) lorsque c'est nécessaire.

```
mailer_transport="gmail"
mailer_encryption="ssl"
mailer_auth_mode="login"
mailer_host="smtp.gmail.com"
```

```
mailer_user="your_username"
mailer_password="your_password"
```

> **Warning**
>
> Faites attention si vous utilisez un système de contrôle de version (SCV) tel que Git pour votre projet, en particulier si votre dépôt est accessible publiquement à n'importe qui. Vous devriez vous assurer que les fichiers contenant des informations sensibles, tel que `app/parameters.ini`, sont dans la liste des fichier à ignorer. Une approche courante consiste à suffixer le nom de fichier qui a des informations sensibles, tel que `app/parameters.ini`, avec `.dist`. Vous pouvez alors proposer des valeurs par défaut pour les paramètres sensibles dans ce fichier, et l'ajouter à votre gestionnaire de version, pendant que le vrai fichier, par exemple `app/parameters.ini` est inclus dans la liste de ceux à ignorer. Vous pouvez alors déployer les fichiers `*.dist` avec votre projet et permettre aux développeurs de supprimer l'extension .dist et renseigner les paramètres requis.

Mise à jour du contrôleur

Mettez à jour le contrôleur de `Page` situé dans `src/Blogger/BlogBundle/Controller/PageController.php` avec le contenu suivant :

```
// src/Blogger/BlogBundle/Controller/PageController.php

public function contactAction()
{
    // ..
    if ($form->isValid()) {

        $message = \Swift_Message::newInstance()
            ->setSubject('Contact enquiry from symblog')
            ->setFrom('enquiries@symblog.co.uk')
            ->setTo('email@email.com')
            ->setBody($this->renderView('BloggerBlogBundle:Page:contactEmail.txt.twig', array('enquiry' => $enquiry)));
        $this->get('mailer')->send($message);

        $this->get('session')->setFlash('blogger-notice', 'Your contact enquiry was successfully sent. Thank you!');

        // Redirect - This is important to prevent users re-posting
        // the form if they refresh the page
        return $this->redirect($this->generateUrl('BloggerBlogBundle_contact'));
    }
    // ..
}
```

Une fois que la librairie Swift Mailer nous a permis de créer une instance d'un objet `Swift_Message`, il est utilisé pour envoyer l'email.

> **Note**
>
> Comme la librairie Swift Mailer n'utilise pas les espaces de nom, nous devons préfixer la classe avec avec un `\`. Cela indique à PHP de réaliser l'échappement vers l' espace global. Vous devrez préfixer tous les classes et fonctions qui ne sont pas dans un espace de nom avec un `\`. Si vous ne placiez pas préfixe avant la classe `Swift_Message`, PHP chercherait alors la classe dans l'espace de nom actuel, dans cet exemple `Blogger\BlogBundle\Controller`, conduisant à l'apparition d'une erreur, car la classe ne peut légitimement pas être trouvée.

Nous avons également émis un message `flash` sur la session. Les messages flash sont des messages qui sont affichés seulement après une requête, ensuite ils sont supprimés par Symfony2. Le message flash sera affiché dans le template actuel pour informer l'utilisateur que le message a été envoyé. Comme les messages flash ne sont affichés qu'après une requête unique, ils sont parfaits pour notifier l'utilisateur de la réussite (ou l'échec) des actions précédentes.

Pour afficher les messages flash nous devons mettre à jour le template de la page de contact situé dans `src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig`. Mettez à jour le contenu du template avec ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig #}

{# rest of template ... #}
<header>
    <h1>Contact symblog</h1>
</header>

{% if app.session.hasFlash('blogger-notice') %}
    <div class="blogger-notice">
        {{ app.session.flash('blogger-notice') }}
    </div>
{% endif %}

<p>Want to contact symblog?</p>

{# rest of template ... #}
```

Cela vérifie qu'il y a un message flash à afficher avec pour identificateur 'blogger-notice', et si c'est le cas on l'affiche.

Enregistrer l'email du webmaster

Symfony2 propose un système de configuration que nous pouvons utiliser pour définir nos propres paramètres. Nous allons utiliser cette méthode afin de définir une adresse email pour le webmaster, plutôt que de la coder en dur dans le contrôleur comme nous l'avons fait au dessus. De cette manière, nous pourrons facilement réutiliser cette variable à d'autres endroits sans dupliquer le code. De plus, lorsque votre blog aura généré tellement de traffic que les requêtes seront trop pénibles à gérer par le webmaster, il sera peut être temps de les déléguer à votre assistant. Créez un nouveau fichier dans `src/Blogger/BlogBundle/Resources/config/config.yml` et collez-y le code suivant :

```
# src/Blogger/BlogBundle/Resources/config/config.yml
parameters:
    # Blogger contact email address
    blogger_blog.emails.contact_email: contact@email.com
```

Lorsque l'on définit des paramètres, c'est une bonne habitude de découper le nom de la variable en un certain nombre de composants. La première partie devrait être le nom du bundle en minuscule, en utilisant des underscore _ pour séparer les mots. Dans cet exemple, nous avons remplacé `BloggerBlogBundle` par `blogger_blog`. Le reste du nom de paramètres peut contenir n'importe quel nombre de parties, séparées par des points .. Cela nous permet de regrouper les paramètres logiquement.

Afin que l'application Symfony2 puisse utiliser ces nouveaux paramètres, nous devons importer le nouveau fichier de configuration situé dans `app/config/config.yml`. Pour réaliser cela, mettez à jour les `imports` au début du fichier par ce qui suit :

```
# app/config/config.yml
imports:
    # .. existing import here
    - { resource: @BloggerBlogBundle/Resources/config/config.yml }
```

Le chemin d'import est le chemin physique du fichier sur le disque. La directive `@BloggerBlogBundle` va va se rendre au chemin du `BloggerBlogBundle`, qui est `src/Blogger/BlogBundle`.

Nous pouvons enfin mettre à jour l'action de contact, afin de nous servir de ce nouveau paramètre.

```
// src/Blogger/BlogBundle/Controller/PageController.php

public function contactAction()
{
    // ..
    if ($form->isValid()) {

        $message = \Swift_Message::newInstance()
            ->setSubject('Contact enquiry from symblog')
            ->setFrom('enquiries@symblog.co.uk')
            ->setTo($this->container->getParameter('blogger_blog.emails.contact_email'))
            ->setBody($this->renderView('BloggerBlogBundle:Page:contactEmail.txt.twig', array('enquiry' => $enquiry)));
        $this->get('mailer')->send($message);

        // ..
    }
    // ..
}
```

> **Tip**
>
> Comme le fichier de configuration est importé au début du fichier de configuration de l'application, il est facile de remplacer n'importe lequel des paramètres importés. Par exemple, en ajoutant ce qui suit à la fin du fichier `app/config/config.yml`, nous pourrions remplacer les valeurs des paramètres du bundle.
>
> ```
> # app/config/config.yml
> parameters:
>     # Blogger contact email address
>     blogger_blog.emails.contact_email: assistant@email.com
> ```
>
> Cette personnalisation permet au bundle de proposer des valeurs par défaut que l'application peut par la suite remplacer.

> **Note**
>
> Bien qu'il soit facile de créer des paramètres de configuration par cette méthode, Symfony2 propose également une méthode dans laquelle on expose une configuration sémantique pour le bundle. Nous détaillerons cette méthode plus loin dans le tutoriel.

Créer un template pour les email

Le corps de l'email est décrit dans un template. Créez ce template dans `src/Blogger/BlogBundle/Resources/view/Page/contactEmail.txt.twig` et mettez y ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/view/Page/contactEmail.txt.twig #}
A contact enquiry was made by {{ enquiry.name }} at {{ "now" | date("Y-m-d H:i") }}.

Reply-To: {{ enquiry.email }}
Subject: {{ enquiry.subject }}
Body:
{{ enquiry.body }}
```

Le contenu de l'email est celui que l'utilisateur vient de soumettre.

Vous avez peut-être remarqué que l'extension de ce template est différente de celle des autres templates que nous avons créés jusque là. Il utilise l'extension `.txt.twig`. La première partie de l'extension, `.txt`, spécifie le format du fichier à générer. Parmi les formats courants, on peut noter .txt, .html, .css, .js, .xml et .json. La seconde partie de l'extension définit quel moteur de template utiliser. Dans le cas présent Twig. Une extension en `.php` signifierait l'utilisation de PHP pour le rendu du template.

Lorsque vous soumettez une requête un email va être envoyé à l'addresse définie dans le paramètre `blogger_blog.emails.contact_email`.

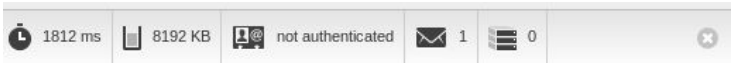
Tip

Symfony2 nous permet de configurer le comportement de la librairie Swift Mailer lorsque l'on travaille dans des environnements de développement Symfony2 différents. Nous pouvons dès à présent voir cela dans l'environnement `test`. Par défaut, la distribution standard de Symfony2 est configurée de telle sorte que Swift Mailer n'envoie pas d'emails lorsque l'application est dans l'environnement `test`. C'est défini dans le fichier `app/config/config_test.yml`.

```
# app/config/config_test.yml
swiftmailer:
    disable_delivery: true
```

Il pourrait être utile de dupliquer cette fonctionnalité pour l'environnement `dev`. Après tout, vous ne voulez pas envoyer accidentellement un email à une adresse incorrecte au cours du développement. Pour cela, ajoutez la configuration ci dessus au fichier de configuration de l'environnement `dev`, dans `app/config/config_dev.yml`.

Vous vous demandez peut-être comment il est possible que s'assurer que les emails sont envoyés, et plus précisément quel est leur contenu, étant donné qu'ils ne sont plus envoyés à une adresse email. Symfony2 propose une solution à cela via la barre d'outil de développement. Lorsqu'un email est envoyé, une notification par email va apparaître dans la barre d'outils, qui contient toutes les informations à propos de l'email que Swift Mailer aurait délivrée.



Si vous réalisez une redirection après l'envoi d'un email, comme nous le faisons avec le formulaire de contact, vous devrez paramètrer la valeur de `intercept_redirects` dans `app/config/config_dev.yml` à `true` afin de voir les notifications d'envoi d'email dans la barre d'outils.

Nous aurions pu configurer Swift Mailer pour envoyer tous les emails à une adresse spécifique dans l'environnement `dev` en plaçant le code suivant dans le fichier de configuration de correspondant, `app/config/config_dev.yml`.

```
# app/config/config_dev.yml
swiftmailer:
    delivery_address:  development@symblog.dev
```

Conclusion

Nous avons présenté les concepts derrière un des aspects les plus fondamentaux de n'importe quel site web: les formulaires. Symfony2 propose une excellente librairie de formulaires et de validateurs qui nous permet de séparer la logique de validation du formulaire de telle sorte qu'elle puisse être utilisée ailleurs dans l'application (dans le modèle par exemple). Nous avons également vu comment mettre en place des paramètres de configuration personnalisés qui peuvent être utilisés dans l'application. Enfin nous avons vu comment envoyer des emails grâce à la librairie Swift Mailer.

Dans la prochaine partie, nous allons aborder un des grands axes de ce tutoriel, le modèle. Nous allons présenter Doctrine 2 et nous en servir pour construire notre modèle pour les articles, construire la page d'affichage des articles, et explorer le thème des données factices.