Part 2] - Contact Page: Validators, Forms and Emailing

Overview

Now we have the basic HTML templates in places, its time to make one of the pages functional. We will begin with one of the simplest pages; The Contact page. At the end of this chapter you will have a Contact page that allows users to send the webmaster contact enquiries. These enquiries will be emailed to the webmaster.

The following areas will be demonstrated in this chapter:

1. Validators
2. Forms
3. Setting bundle configuration values

Contact Page

Routing

As with the about page we created in the last chapter, we will start by defining the contact page route. Open up the`BloggerBlogBundle` routing file located at `src/Blogger/BlogBundle/Resources/config/routing.yml` and append the following routing rule.

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

There is nothing new here, the rule matches on the pattern `/contact`, for the HTTP `GET` method and executes the `contact` action of the `Page` controller in the `BloggerBlogBundle`.

Controller

Next lets add the action for the contact page to the `Page` Controller in the `BloggerBlogBundle` located at`src/Blogger/BlogBundle/Controller/PageController.php`.

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

For now the action is very simple, it just renders the contact page view. We will come back to the controller later.

View

Create the contact page view at `src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig` and add the following content.

```
{# 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 %}
```

This template is also quite simple. It extends the `BloggerBlogBundle` layout template, overrides the title block to set a custom title and defines some content for the body block.

Linking to the page

Lastly we need to update the link in the application template located at `app/Resources/views/base.html.twig` to link to the contact page.

```
<!-- 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 %}
```

If you point your browser to `http://symblog.dev/app_dev.php/` and click the contact link in the navigation bar, you should see a very basic contact page displayed. Now we have the page correctly setup, its time to start working on the Contact form. This is split into 2 distinct parts; The Validators and The Form. Before we can address the concept of Validators and the Form we need to think about how we will handle the data of the contact enquiry.

Contact Entity

Lets begin by creating a class that represents a contact enquiry from a user. We want to trap some basic information such as name, subject and enquiry body. Create a new file located at `src/Blogger/BlogBundle/Entity/Enquiry.php` and paste in the following content.

```php
<?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;
    }
}
```

As you can see this class just defines some protected members and the accessors for them. There is nothing here that defines how we validate the members, or how the members relate to form elements. We will come back to that later.

Note

Lets take a quick aside to talk about the use of namespaces in Symfony2. The entity class we have created sets the namespace to `Blogger\BlogBundle\Entity`. As Symfony2 autoloading supports the **PSR-0 standard** the namespace directly maps to the Bundle folder structure. The `Enquiry` entity class is located at`src/Blogger/BlogBundle/Entity/Enquiry.php` which ensures Symfony2 is able to correctly autoload the class.

How does the Symfony2 autoloader know the `Blogger` namespace can be found in the `src` directory? This is thanks to the configurations in the autoloader at `app/autoloader.php`

```php
// app/autoloader.php
$loader->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
```

This statement registers a fallback for any namespaces not already registered. As the `Blogger` namespace is not registered, the Symfony2 autoloader will look for the required files in the `src` directory.

Autoloading and namespaces are a very powerful concept in Symfony2. If you are getting errors where PHP is unable to find classes, its likely you have a mistake in your namespace or folder structure. Also check the namespace has been registered with the autoloader as shown above. You should not be tempted to `fix` this by using the PHP `require` or `include` directives.

Forms

Next we will create the form. Symfony2 comes bundled with a very powerful form framework that makes the tedious task of dealing with forms easy. As with all Symfony2 components, it can be used outside of Symfony2 in your own projects. The **Form Component Source** is available on Github. We will begin by creating an `AbstractType` class that represents the enquiry form. We could have created the form directly in the controller and not bothered with this class, however separating the form into its own class allows us to reuse the form throughout the application. It also prevents us cluttering up the controller. After all, the controller is supposed to be simple. It purpose is to provide the glue between the Model and the View.

EnquiryType

Create a new file located at `src/Blogger/BlogBundle/Form/EnquiryType.php` and paste in the following content.

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

namespace Blogger\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

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

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

The `EnquiryType` class introduces the `FormBuilderInterface` interface. This interface is used by the `FormBuilder` class. The `FormBuilder`class is your best friend when it comes to creating forms. It is able to simplify the process of defining fields based on the metadata the field has. As our Enquiry entity is so simple we haven't defined any metadata yet so the `FormBuilder` will default the field type to text input. This is suitable for most of the fields except body where we want a `textarea`, and email where we want to take advantage of the new email input type in HTML5.

Note

One key point to mention here is that the `getName` method should return a unique identifier.

Creating the form in the controller

Now we have defined the `Enquiry` entity and `EnquiryType`, we can update the contact action to use them. Replace the content of the contact action located at `src/Blogger/BlogBundle/Controller/PageController.php` with the following.

```php
// 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()
    ));
}
```

We begin by creating an instance of the `Enquiry` entity. This entity represent the data of a contact enquiry. Next we create the actual form. We specify the `EnquiryType` we created earlier, and pass in our enquiry entity object. The `createForm` method is able to use these 2 blueprints to create a form representation.

As this controller action will deal with displaying and processing the submitted form, we need to check the HTTP method. Submitted forms are usually sent via `POST`, and our form will be no exception. If the request method is `POST`, a call to`bindRequest` will transform the submitted data back to the members of our `$enquiry` object. At this point the `$enquiry` object now holds a representation of what the user submitted.

Next we make a check to see if the form is valid. As we have specified no validators at the point, the form will always be valid.

Finally we specify the template to render. Note that we are now also passing over a view representation of the form to the template. This object allow us to render the form in the view.

As we have used 2 new classes in our controller we need to import the namespaces. Update the controller file located at`src/Blogger/BlogBundle/Controller/PageController.php` with the following. The statements should be placed under the existing `use`statement.

```
<?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
// ..
```

Rendering the form

Thanks to Twig's methods rendering forms is very simple. Twig provides a layered system for form rendering that allows you to render the form as one entire entity, or as individual errors and elements, depending on the level of customisation you require.

To demonstrate the power of Twig's methods we can use the following snippet to render the entire form.

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

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

While this is very useful for prototyping and simple forms it has its limitations when extended customisations are needed, which is often the case with forms.

For our contact form, we will opt for the middle ground. Replace the template code located at`src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig` with the following.

```
{# 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 %}
```

As you can see, we use 4 new Twig methods to render the form.

The first method `form_enctype` sets the form content type. This must be set when your form deals with file uploads. Our form doesn't have any use for this method but its always good practice to use this on all forms in the event that you may add file uploads in the future. Debugging a form that handles file uploads that has no content type set can be a real head scratcher!

The second method `form_errors` will render any errors the form has in the event that validation failed.

The third method `form_row` outputs the entire elements related to each form field. This include any errors for the field, the field label and the actual field element.

Finally we use the `form_rest` method. Its always a safe bet to use the method at the end of the form to render any fields you may have forgotten, including hidden fields and the Symfony2 Form CSRF token.

Note

Cross-site request forgery (CSRF) is explained in details in the **Forms chapter** of the Symfony2 book.

Styling the form

If you view the contact form now via `http://symblog.dev/app_dev.php/contact` you will notice it doesn't look that appealing. Lets add some styles to improve this look. As the styles are specific to the form within our Blog bundle we will create the styles in a new stylesheet within the bundle itself. Create a new file located at `src/Blogger/BlogBundle/Resources/public/css/blog.css` and paste in the following content.

```
.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; }
```

We need to let the application know we want to use this stylesheet. We could import the stylesheet into the contact template but as other templates will also use this stylesheet later, it makes sense to import it into the `BloggerBlogBundle`layout we created in chapter 1. Open up the `BloggerBlogBundle` layout located at `src/Blogger/BlogBundle/Resources/views/layout.html.twig`and replace with the following content.

```
{# 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 %}
```

You can see we have defined a stylesheet block to override the stylesheet block defined in the parent template. However, its important to notice the call to the `parent` method. This will import the content from the stylesheets block in the parent template located at `app/Resources/base.html.twig`, allowing us to append our new stylesheet. After all, we don't want to replace the existing stylesheets.

In order for the `asset` function to correctly link up the the resource we need to copy over or link the bundle resources into the applications `web` folder. This can be done with the following

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

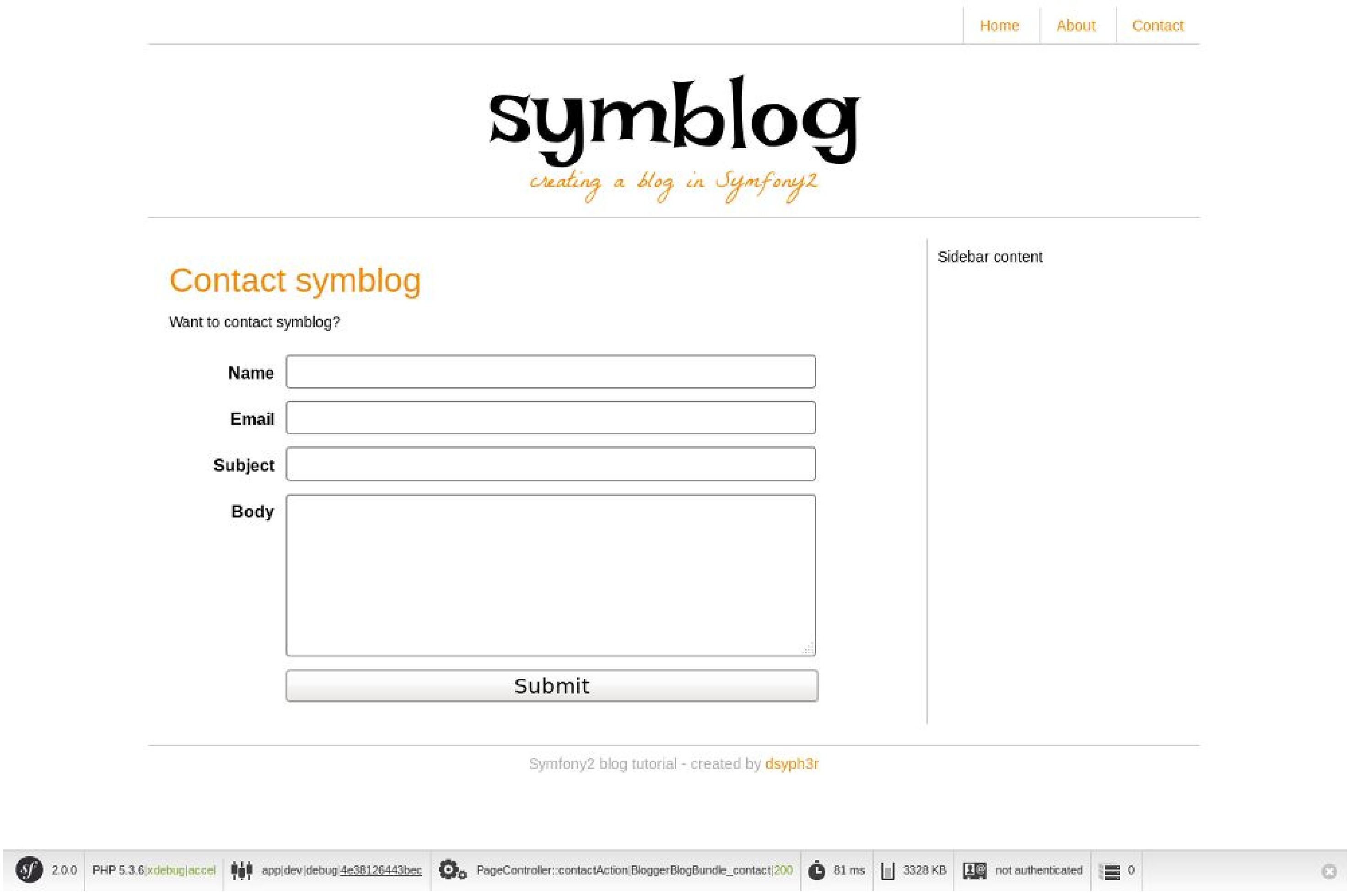
Note

If you are using an Operating System that doesn't support symlinks such as Windows you will need to drop the symlink option as follows.

```
php app/console assets:install web
```

This method will actually copy the resources from the bundles `public` folder into the applications `web` folder. As the files are actually copied, you will need to run this task every time you make a change to a bundles public resource.

Now if you refresh the contact page the form will be beautifully styled.



Home
About
Contact
symblog
creating a blog in Symfony2

Contact symblog

Want to contact symblog?

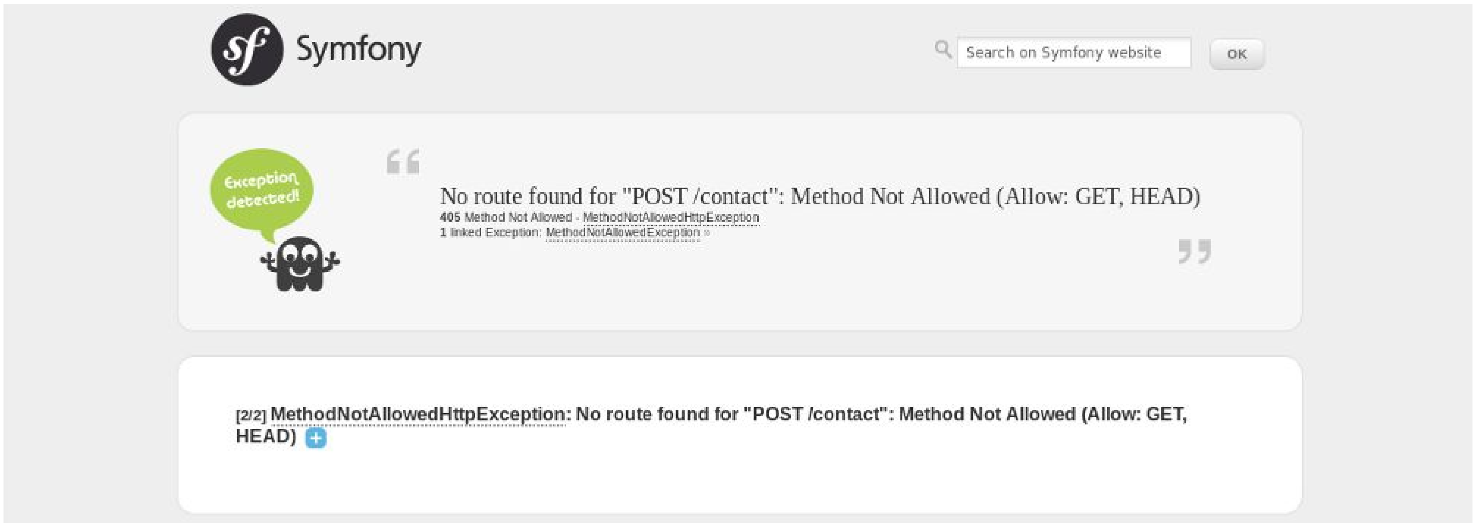
Name
Email
Subject
Body
Submit
Sidebar content
Symfony2 blog tutorial - created by dsyph3r
2.0.0
PHP 5.3.6|xdebug|accel
app|dev|debug|4e38126443bec
PageController::contactAction|BloggerBlogBundle_contact|200
81 ms
3328 KB
not authenticated
0

Tip

While the `asset` function provides the functionality we require to use resources, there is a better alternative for this. The **Assetic** library by **Kris Wallsmith** is bundled with the Symfony2 standard distribution by default. This library provides asset management far beyond the standard Symfony2 capabilities. Assetic allows us to run filters on assets to automatically combine, minify and gzip them. It can also run compression filters on images. Assetic further allows us to reference resources directly within the bundles public folder without having to run the `assets:install` task. We will explore the use of Assetic in later chapters.

Failure to submit

The eager ones among you may have already tried to submit the form to be greeted with a Symfony2 error.



No route found for "POST /contact": Method Not Allowed (Allow: GET, HEAD)
405 Method Not Allowed - MethodNotAllowedHttpException
1 linked Exception: MethodNotAllowedException »

[2/2] MethodNotAllowedHttpException: No route found for "POST /contact": Method Not Allowed (Allow: GET, HEAD)

This error is telling us that there is no route to match `/contact` for the HTTP method POST. The route only accepts GET and HEAD requests. This is because we configured our route with the method requirement of GET.

Lets update the contact route located at `src/Blogger/BlogBundle/Resources/config/routing.yml` to also allow POST requests.

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

Tip

You maybe wondering why the route would allow the HEAD method when only GET was specified. This is because HEAD is a GET request but only the HTTP Headers are returned.

Now when you submit the form it should function as expected, although expected doesn't actually do much yet. The page will just redirect you back to the contact form.

Validators

The Symfony2 validator allows us to perform the task of data validation. Validation is a common task when dealing with data from forms. Validation also needs to be performed on data before it is submitted to a database. The Symfony2 validator allows us to separate our validation logic away from the components that may use it, such as the Form component or the Database component. This approach means we have one set of validation rules for an object.

Let's begin by updating the `Enquiry` entity located at `src/Blogger/BlogBundle/Entity/Enquiry.php` to specify some Validators. Ensure you add the 5 new `use` statements at the top of the file.

```
<?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));
    }

    // ..

}
```

To define the validators we must implement the static method `loadValidatorMetadata`. This provides us with an object of`ClassMetadata`. We can use this object to set property constraints on our entity members. The first statement applies the`NotBlank` constraint to the `name` member. The `NotBlank` validator is as simple as it sounds, it will only return `true` if the value it is validating is not empty. Next we setup validation for the `email` member. The Symfony2 Validator service provides a validator for **emails** that will even check MX records to ensure the domain is valid. On the `subject` member we want to set a`NotBlank` and a `MaxLength` constraint. You can apply as many validators to a member as you wish.

A full list of **validator constrains** is available in the Symfony2 reference documents. It is also possible to **create custom validators**.

Now when you submit the contact form, your submitted data will be passed through the validation constraints. Try typing in an invalid email address. You should see an error message informing you that the email address is invalid. Each validator provides a default message that can be overridden if required. To change the message output by the email validator you would do the following.

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

Tip

If you are using a browser that supports HTML5 (it is likely you are) you will be prompted with HTML5 messages enforcing certain constraints. This is client side validation and Symfony2 will set suitable HTML5 constraints based on your `Entity` metadata. You can see this on the email element. The output HTML is

```
<input type="email" value="" required="required" name="contact[email]" id="contact_email">
```

It has used one of the new HTML5 Input type fields, email and has set the required attribute. Client side validation is great in that it doesn't require a round trip to the server to validate the form. However, client side validation should not be used alone. You should always validate submitted data server side as it's quite easy for a user to by-pass the client side validation.

Sending the email

While our contact form will allow users to submit enquiries, nothing actually happens with them yet. Let's update the controller to send an email to the blog webmaster. Symfony2 comes complete with the **Swift Mailer** library for sending emails. Swift Mailer is a very powerful library, we will only scratch the surface of what this library can perform.

Configure Swift Mailer settings

Swift Mailer is already configured out of the box to work in the Symfony2 Standard Distribution, however we need to configure some settings regarding sending methods, and credentials. Open up the parameters file located at`app/config/parameters.yml` and find the settings prefixed with `mailer_`.

```
mailer_transport: smtp
mailer_host: 127.0.0.1
mailer_user: null
mailer_password: null
```

Swift Mailer provides a number of methods for sending emails, including using an SMTP server, using a local install of sendmail, or even using a GMail account. For simplicity we will use a GMail account. Update the parameters with the following, substituting your username and password where necessary.

```
mailer_transport: gmail
mailer_encryption: ssl
mailer_auth_mode: login
mailer_host: smtp.gmail.com
mailer_user: your_username
mailer_password: your_password
```

Warning

Be careful if you are using a Version Control System (VCS) like Git for your project, especially if your repository is publicly accessible as your GMail username and password will be committed to the repository and will be available for anybody to see. You should make sure the file `app/config/parameters.ini` is added to the ignore list of your VCS. A common approach to this problem is to suffix the file name of the file that has sensitive information such as `app/config/parameters.ini` with `.dist`. You then provide sensible defaults for the settings in this file and add the actual file, i.e. `app/config/parameters.ini` to you VCS ignore list. You can then deploy the `*.dist` file with your project and allow the developer to remove the `.dist` extension and fill in the required settings.

Update the controller

Update the `Page` controller located at `src/Blogger/BlogBundle/Controller/PageController.php` with the content below.

```
// 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'));
    }
    // ..
}
```

When you have used the Swift Mailer library to create a `Swift_Message` instance, that can be sent as an email.

Note

As the Swift Mailer library does not use namespaces, we need to prefix the Swift Mailer class with a \. This tells PHP to escape back to the **global space**. You will need to prefix all classes and functions that are not namespaced with a \. If you did not place this prefix before the `Swift_Message` class PHP would look for the class in the current namespace, which in this example is `Blogger\BlogBundle\Controller`, causing an error to be thrown.

We have also set a `flash` message on the session. Flash messages are messages that persist for exactly one request. After that they are automatically cleaned up by Symfony2. The `flash` message will be displayed in the contact template to inform the user the enquiry has been sent. As `flash` message only persist for exactly one request, they are perfect for notifying the user of the success of the previous actions.

To display the `flash` message we need to update the contact template located at`src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig`. Update the content of the template with the following.

```
{# 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 ... #}
```

This checks to see if a `flash` message with the identifier 'blogger-notice' is set and outputs it.

Register webmaster email

Symfony2 provides a configuration system that we can use to define our own settings. We will use this system to set the webmaster email address rather than hard coding the address in the controller above. That way we can easily reuse this value in other places without code duplication. Further, when your blog has generated so much traffic the enquiries become too much for you to deal with, you can easily update the email address to pass the emails onto your assistant. Create a new file at `src/Blogger/BlogBundle/Resources/config/config.yml` and paste in the following.

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

When defining parameters it is good practice to break the parameter name into a number of components. The first part should be a lower cased version of the bundle name using underscores to separate words. In our example we have transformed the bundle `BloggerBlogBundle` into `blogger_blog`. The remaining part of the parameter name can contain any number of parts separated by the . (period) character. This allows us to logically group parameters together.

In order for the Symfony2 application to use the new parameters, we need to import the config into the main application config file located at `app/config/config.yml`. To achieve this, update the `imports` directive at the top of the file to the following.

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

The import path is the physical location of the file on disk. The `@BloggerBlogBundle` directive will resolve to the path of the`BloggerBlogBundle` which is `src/Blogger/BlogBundle`.

Finally let's update the contact action to use the parameter.

```
// 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

As the config file is imported at the top of the application configuration file we can easily override any of the imported parameters in the application. For example, adding the following to the bottom of`app/config/config.yml` would override the bundle set value for the parameter.

```
# app/config/config.yml
parameters:
    # Blogger contact email address
    blogger_blog.emails.contact_email: assistant@email.com
```

These customisation allow for the bundle to provide sensible defaults for values where the application can override them.

Note

While its easy to create bundle configuration parameters using this method Symfony2 also provides a method where you **expose a Semantic Configuration** for a bundle. We will explore this method later in the tutorial.

Create the Email template

The body of the email is set to render a template. Create this template at`src/Blogger/BlogBundle/Resources/views/Page/contactEmail.txt.twig` and add the following.

```
{# src/Blogger/BlogBundle/Resources/views/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 }}
```

The content of the email is just the enquiry the user submitted.

You may have also noticed the extension of this template is different to the other templates we have created. It uses the extension `.txt.twig`. The first part of the extension, `.txt` specifies the format of the file to generate. Common formats here include, .txt, .html, .css, .js, .xml and .json. The last part of the extension specifies which templating engine to use, in this case Twig. An extension of `.php` would use PHP to render the template instead.

When you now submit an enquiry, an email will be sent to the address set in the `blogger_blog.emails.contact_email` parameter.

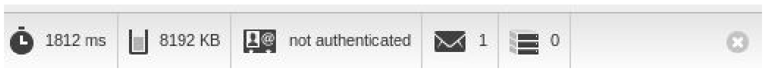
Tip

Symfony2 allows us to configure the behavior of the Swift Mailer library while operating in different Symfony2 environments. We can already see this in use for the `test` environment. By default, the Symfony 2 Standard Distribution configures Swift Mailer to not send emails when running in the `test` environment. This is set in the test configuration file located at `app/config/config_test.yml`.

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

It could be useful to duplicate this functionality for the `dev` environment. After all, you don't want to accidentally send an email to the wrong email address while developing. To achieve this, add the above configuration to the `dev` configuration file located at `app/config/config_dev.yml`.

You may be wondering how you can now test that the emails are being sent, and more specifically the content of them, seeing as they will no longer be delivered to an actual email address. Symfony2 has a solution for this via the developer toolbar. When an email is sent an email notification icon will appear in the toolbar that has all the information about the email that Swift Mailer would have delivered.



If you perform a redirect after sending an email, like we do for the contact form, you will need to set the`intercept_redirects` setting in `app/config/config_dev.yml` to true in order to see the email notification in the toolbar.

We could have instead configured Swift Mailer to send all emails to a specific email address in the `dev`environment by placing the following setting in the `dev` configuration file located at `app/config/config_dev.yml`.

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

Conclusion

We have demonstrated the concepts behind creating one of the most fundamental part of any website: forms. Symfony2 comes complete with an excellent Validator and Form library that allows us to separate validation logic out of the form so it can be used by other parts of the application (such as the Model). We were also introduced to setting custom configuration settings that can be read into our application.

Next we will look at a big part of this tutorial, The Model. We will introduce Doctrine 2 and use it to define the blog Model. We will also build the show blog page and explore the concept of Data fixtures .