Official Symfony Best Practices

early preview release

Official Symfony Best Practices

Contents at a Glance

Chapter 1

The Symfony Framework Best Practices

The Symfony framework is well-known for being *really* flexible and is used to build micro-sites, enterprise applications that handle billions of connections and even as the basis for *other* frameworks. Since its release in July 2011, the community has learned a lot about what's possible and how to do things *best*.

These community resources - like blog posts or presentations - have created an unofficial set of recommendations for developing Symfony applications. Unfortunately, a lot of these recommendations are in fact wrong. They unnecessarily overcomplicate things and don't follow the original pragmatic philosophy of Symfony.

What is this Guide About?

This guide aims to fix that by describing the **official best-practices for developing web apps with the Symfony full-stack framework**. These are best- practices that fit the philosophy of the framework as envisioned by its original creator *Fabien Potencier*[1].

We know that old habits die hard and some of you will be shocked by some of these best-practices. But by following these, you'll be able to develop apps faster, with less complexity and with the same or even higher quality. It's also a moving target that will continue to improve.

Keep in mind that these are **optional recommendations** that you and your team may or may not follow to develop Symfony applications. If you want to continue using your own best-practices and methodologies, you can of course do it. Symfony is flexible enough to adapt to your needs. That will never change.

Who this Book Is for (Hint: It's not a Tutorial)

Any Symfony developer, whether you are an expert or a newcomer, can read this guide. But since this isn't a tutorial, you'll need some basic knowledge of Symfony to follow everything. If you are totally new to Symfony, welcome! Start with *The Quick Tour*[2] tutorial first.

1. `https://connect.sensiolabs.com/profile/fabpot`
2. `http://symfony.com/doc/current/quick_tour/the_big_picture.html`

We've deliberately kept this guide short. We won't repeat explanations that you can find in the vast Symfony documentation, like discussions about dependency injection or front controllers. We'll solely focus on explaining how to do what you already know.

The Application

In addition to this guide, you'll find a sample application developed with all these best practices in mind. **The application is a simple blog engine**, because that will allow us to focus on the Symfony concepts and features without getting buried in difficult details.

Instead of developing the application step by step in this guide, you'll find selected snippets of code through the chapters. Please refer to the last chapter of this guide to find more details about this application and the instructions to install it.

Chapter 2
Creating the Project

Installing Symfony

There is only one recommended way to install Symfony:

BEST PRACTICE

Always use *Composer*[1] to install Symfony.

Composer is the dependency manager used by modern PHP applications. Adding or removing requirements for your project and updating the third-party libraries used by your code is a breeze thanks to Composer.

Dependency Management with Composer

Before installing Symfony, you need to make sure that you have Composer installed globally. Open your terminal (also called *command console*) and run the following command:

Listing 2-1
```
$ composer --version
Composer version 1e27ff5e22df81e3cd0cd36e5fdd4a3c5a031f4a 2014-08-11 15:46:48
```

You'll probably see a different version identifier. Never mind because Composer is updated on a continuous basis and its specific version doesn't matter.

Installing Composer Globally

In case you don't have Composer installed globally, execute the following two commands if you use Linux or Mac OS X (the second command will ask for your user password):

1. `https://getcomposer.org/`

Listing 2-2
```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

Depending on your Linux distribution, you may need to execute `su` command instead of `sudo`.

If you use a Windows system, download the executable installer from the *Composer download page*[2] and follow the steps to install it.

Creating the Blog Application

Now that everything is correctly set up, you can create a new project based on Symfony. In your command console, browse to a directory where you have permission to create files and execute the following commands:

Listing 2-3
```
$ cd projects/
$ composer create-project symfony/framework-standard-edition blog/
```

This command will create a new directory called `blog` that will contain a fresh new project based on the most recent stable Symfony version available.

Checking the Symfony Installation

Once the installation is finished, enter the `blog/` directory and check that Symfony is correctly installed by executing the following command:

Listing 2-4
```
$ cd blog/
$ php app/console --version

Symfony version 2.6.* - app/dev/debug
```

If you see the installed Symfony version, everything worked as expected. If not, you can execute the following *script* to check what does prevent your system from correctly executing Symfony applications:

Listing 2-5
```
$ php app/check.php
```

Depending on your system, you can see up to two different lists when executing the *check.php* script. The first one shows the mandatory requirements which your system must meet to execute Symfony applications. The second list shows the optional requirements suggested for an optimal execution of Symfony applications:

Listing 2-6
```
Symfony2 Requirements Checker
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

> PHP is using the following php.ini file:
  /usr/local/zend/etc/php.ini

> Checking Symfony requirements:
```

2. https://getcomposer.org/download/

```
  .....E.......................W.....

[ERROR]
Your system is not ready to run Symfony2 projects

Fix the following mandatory requirements
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 * date.timezone setting must be set
   > Set the "date.timezone" setting in php.ini* (like Europe/Paris).

Optional recommendations to improve your setup
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 * short_open_tag should be disabled in php.ini
   > Set short_open_tag to off in php.ini*.
```

Symfony releases are digitally signed for security reasons. If you want to verify the integrity of your Symfony installation, take a look at the *public checksums repository*[3] and follow *these steps*[4] to verify the signatures.

Structuring the Application

After creating the application, enter the `blog/` directory and you'll see a number of files and directories generated automatically:

Listing 2-7

```
blog/
├── app/
│   ├── console
│   ├── cache/
│   ├── config/
│   ├── logs/
│   └── Resources/
├── src/
│   └── AppBundle/
├── vendor/
└── web/
```

This file and directory hierarchy is the convention proposed by Symfony to structure your applications. The recommended purpose of each directory is the following:

- `app/cache/`, stores all the cache files generated by the application;
- `app/config/`, stores all the configuration defined for any environment;
- `app/logs/`, stores all the log files generated by the application;
- `app/Resources/`, stores all the templates and the translation files for the application;
- `src/AppBundle/`, stores the Symfony specific code (controllers and routes), your domain code (e.g. Doctrine classes) and all your business logic;
- `vendor/`, this is the directory where Composer installs the application's dependencies and you should never modify any of its contents;

3. https://github.com/sensiolabs/checksums
4. http://fabien.potencier.org/article/73/signing-project-releases

- `web/`, stores all the front controller files and all the web assets, such as stylesheets, JavaScript files and images.

Application Bundles

When Symfony 2.0 was released, most developers naturally adopted the symfony 1.x way of dividing applications into logical modules. That's why many Symfony apps use bundles to divide their code into logical features: `UserBundle`, `ProductBundle`, `InvoiceBundle`, etc.

But a bundle is *meant* to be something that can be reused as a stand-alone piece of software. If `UserBundle` cannot be used *"as is"* in other Symfony apps, then it shouldn't be its own bundle. Moreover `InvoiceBundle` depends on `ProductBundle`, then there's no advantage to having two separate bundles.

BEST PRACTICE

Create only one bundle called `AppBundle` for your application logic

Implementing a single `AppBundle` bundle in your projects will make your code more concise and easier to understand. Starting in Symfony 2.6, the official Symfony documentation uses the `AppBundle` name.

There is no need to prefix the `AppBundle` with your own vendor (e.g. `AcmeAppBundle`), because this application bundle is never going to be shared.

All in all, this is the typical directory structure of a Symfony application that follows these best practices:

Listing 2-8

```
blog/
├── app/
│   ├── console
│   ├── cache/
│   ├── config/
│   └── logs/
│   └── Resources/
├── src/
│   └── AppBundle/
├── vendor/
└── web/
    ├── app.php
    └── app_dev.php
```

If you are using Symfony 2.6 or a newer version, the `AppBundle` bundle is already generated for you. If you are using an older Symfony version, you can generate it by hand executing this command:

Listing 2-9

```
$ php app/console generate:bundle --namespace=AppBundle --dir=src --format=annotation
--no-interaction
```

Extending the Directory Structure

If your project or infrastructure requires some changes to the default directory structure of Symfony, you can *override the location of the main directories*[5]: `cache/`, `logs/` and `web/`.

In addition, Symfony3 will use a slightly different directory structure when it's released:

Listing 2-10

```
blog-symfony3/
├── app/
│   ├── config/
│   └── Resources/
├── bin/
│   └── console
├── src/
├── var/
│   ├── cache/
│   └── logs/
├── vendor/
└── web/
```

The changes are pretty superficial, but for now, we recommend that you use the Symfony2 directory structure.

5. `http://symfony.com/doc/current/cookbook/configuration/override_dir_structure.html`

Chapter 3
Configuration

Configuration usually involves different application parts (such as infrastructure and security credentials) and different environments (development, production). That's why Symfony recommends that you split the application configuration into three parts.

Infrastructure-Related Configuration

BEST PRACTICE

Define the infrastructure-related configuration options in the `app/config/parameters.yml` file.

The default `parameters.yml` file follows this recommendation and defines the options related to the database and mail server infrastructure:

Listing 3-1

```yaml
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     127.0.0.1
    database_port:     ~
    database_name:     symfony
    database_user:     root
    database_password: ~

    mailer_transport:  smtp
    mailer_host:       127.0.0.1
    mailer_user:       ~
    mailer_password:   ~

    # ...
```

These options aren't defined inside the `app/config/config.yml` file because they have nothing to do with the application's behavior. In other words, your application doesn't care about the location of your database or the credentials to access to it, as long as the database is correctly configured.

Canonical Parameters

BEST PRACTICE

Define all your application's parameters in the `app/config/parameters.dist.yml` file.

Since version 2.3, Symfony includes a configuration file called `parameters.dist.yml`, which stores the canonical list of configuration parameters for the application.

Whenever a new configuration parameter is defined for the application, you should also add it to this file and submit the changes to your version control system. Then, whenever a developer updates the project or deploys it to a server, Symfony will check if there is any difference between the canonical `parameters.dist.yml` file and your local `parameters.yml` file. If there is a difference, Symfony will ask you to provide a value for the new parameter and it will add it to your local `parameters.yml` file.

Application-Related Configuration

BEST PRACTICE

Define the application behavior related configuration options in the `app/config/config.yml` file.

The `config.yml` file contains the options used by the application to modify its behavior, such as the sender of email notifications, or the enabled *feature toggles*[1]. Defining these values in `parameters.yml` file would add an extra layer of configuration that's not needed because you don't need or want these configuration values to change on each server.

The configuration options defined in the `config.yml` file usually vary from one *execution environment*[2] to another. That's why Symfony already includes `app/config/config_dev.yml` and `app/config/config_prod.yml` files so that you can override specific values for each environment.

Constants vs Configuration Options

One of the most common errors when defining application configuration is to create new options for values that never change, such as the number of items for paginated results.

BEST PRACTICE

Use constants to define configuration options that rarely change.

1. http://en.wikipedia.org/wiki/Feature_toggle
2. http://symfony.com/doc/current/cookbook/configuration/environments.html

The traditional approach for defining configuration options has caused many Symfony apps to include an option like the following, which would be used to control the number of posts to display on the blog homepage:

Listing 3-2
```yaml
# app/config/config.yml
parameters:
    homepage.num_items: 10
```

If you ask yourself when the last time was that you changed the value of *any* option like this, odds are that you *never* have. Creating a configuration option for a value that you are never going to configure is silly. Our recommendation is to define these values as constants in your application. You could, for example, define a `NUM_ITEMS` constant in the `Post` entity:

Listing 3-3
```php
// src/AppBundle/Entity/Post.php
namespace AppBundle\Entity;

class Post
{
    const NUM_ITEMS = 10;

    // ...
}
```

The main advantage of defining constants is that you can use their values everywhere in your application. When using parameters, they are only available from places wih access to the Symfony container.

Constants can be used for example in your Twig templates thanks to the `constant()` function:

Listing 3-4
```twig
<p>
    Displaying the {{ constant('NUM_ITEMS', post) }} most recent results.
</p>
```

And Doctrine entities and repositories can now easily access these values, whereas they cannot access the container parameters:

Listing 3-5
```php
namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;
use AppBundle\Entity\Post;

class PostRepository extends EntityRepository
{
    public function findLatest($limit = Post::NUM_ITEMS)
    {
        // ...
    }
}
```

The only notable disadvantage of using constants for this kind of configuration values is that you cannot redefine them easily in your tests.

Semantic Configuration: Don't Do It

BEST PRACTICE

Don't define a semantic dependency injection configuration for your bundles.

As explained in *How to Expose a semantic Configuration for a Bundle*[3] article, Symfony bundles have two choices on how to handle configuration: normal service configuration through the `services.yml` file and semantic configuration through a special `*Extension` class.

Although semantic configuration is much more powerful and provides nice features such as configuration validation, the amount of work needed to define that configuration isn't worth it for bundles that aren't meant to be shared as third-party bundles.

Moving Sensitive Options Outside of Symfony Entirely

When dealing with sensitive options, like database credentials, we also recommend that you store them outside the Symfony project and make them available through environment variables. Learn how to do it in the following article: *How to Set external Parameters in the Service Container*[4]

3. http://symfony.com/doc/current/cookbook/bundles/extension.html
4. http://symfony.com/doc/current/cookbook/configuration/external_parameters.html

Chapter 4
Organizing Your Business Logic

In computer software, **business logic** or domain logic is "the part of the program that encodes the real-world business rules that determine how data can be created, displayed, stored, and changed" (read *full definition*[1]).

In Symfony applications, business logic is all the custom code you write for your app that's not specific to the framework (e.g. routing and controllers). Domain classes, Doctrine entities and regular PHP classes that are used as services are good examples of business logic stuff.

For most projects, you should store everything inside the `AppBundle`. Inside here, you can create whatever directories you want to organize things:

Listing 4-1
```
symfoy2-project/
├── app/
├── src/
│   └── AppBundle/
│       └── Utils/
│           └── MyClass.php
├── vendor/
└── web/
```

Storing Classes Outside of the Bundle?

But there's no technical reason for putting business logic inside of a bundle. If you like, you can create your own namespace inside the `src/` directory and put things there:

Listing 4-2
```
symfoy2-project/
├── app/
├── src/
│   ├── Acme/
│   │   └── Utils/
│   │       └── MyClass.php
```

1. `http://en.wikipedia.org/wiki/Business_logic`

```
7  │   └── AppBundle/
8  ├── vendor/
9  └── web/
```

The recommended approach of using the `AppBundle` directory is for simplicity. If you're advanced enough to know what needs to live in a bundle and what can live outside of one, then feel free to do that.

Services: Naming and Format

The blog application needs a utility that can transform a post title (e.g. "Hello World") into a slug (e.g. "hello-world"). The slug will be used as part of the post URL.

Let's, create a new `Slugger` class inside `src/AppBundle/Utils/` and add the following `slugify()` method:

Listing 4-3

```php
// src/AppBundle/Utils/Slugger.php
namespace AppBundle\Utils;

class Slugger
{
    public static function slugify($string)
    {
        return preg_replace(
            '/[^a-z0-9]/', '-', strtolower(trim(strip_tags($string)))
        );
    }
}
```

Next, define a new service for that class.

Listing 4-4

```yaml
# app/config/services.yml
services:
    # keep your service names short
    slugger:
        class: AppBundle\Utils\Slugger
```

Traditionally, the naming convention for a service involved following the class name and location to avoid name collisions. Thus, the service *would have been* called `app.utils.slugger`. But by using short service names, your code will be easier to read and use.

BEST PRACTICE

The name of your application's services should be as short as possible, ideally just one simple word.

Now you can use the custom slugger in any controller class, such as the `AdminController`:

Listing 4-5

```php
public function createAction(Request $request)
{
    // ...

    if ($form->isSubmitted() && $form->isValid()) {
        $slug = $this->get('slugger')->slugify($post->getTitle());
        $post->setSlug($slug);

        // ...
    }
}
```

Service Format: YAML

In the previous section, YAML was used to define the service.

BEST PRACTICE

Use the YAML format to define your own services.

This is controversial, and in our experience, YAML and XML usage is evenly distributed among developers, with a slight preference towards YAML. Both formats have the same performance, so this is ultimately a matter of personal taste.

We recommend YAML because it's friendly to newcomers and concise. You can of course use whatever format you like.

Service: No Class Parameter

You may have have noticed that the previous service definition doesn't configure the class namespace as a parameter:

Listing 4-6

```yaml
# app/config/services.yml

# service definition with class namespace as parameter
parameters:
    slugger.class: AppBundle\Utils\Slugger

services:
    slugger:
        class: "%slugger.class%"
```

This practice is cumbersome and completely unnecessary for your own services:

BEST PRACTICE

Don't define parameters for the classes of your services.

This practice is yet another practice wrongly adopted from third-party bundles. If you develop a bundle meant to be shared, then you *may* define parameters But if you're developing a service for your own application, there is no need to make this configurable.

Using a Persistence Layer

Symfony is an HTTP framework that only cares about generating an HTTP response for each HTTP request. That's why Symfony doesn't provide a way to talk to a persistence layer (e.g. database, external API). You can choose whatever library of strategy you want for this.

In practice, many Symfony applications rely on the independent *Doctrine project*[2] to define their model using entities and repositories. Just like with business logic, we recommend storing Doctrine entities in the `AppBundle`

The three entities defined by our sample blog application are a good example:

Listing 4-7

```
symfony2-project/
├─ ...
└─ src/
   └─ AppBundle/
      └─ Entity/
         ├─ Comment.php
         ├─ Post.php
         └─ User.php
```

If you're more advanced, you can of course store them under your own namespace in `src/`.

Doctrine Mapping Information

Doctrine Entities are plain PHP objects that you store in some "database". Doctrine only knows about your entities through the mapping metadata configured for your model classes. Doctrine supports four metadata formats: YAML, XML, PHP and annotations.

BEST PRACTICE

Use annotations to define the mapping information of the Doctrine entities.

Annotations are by far the most convenient and agile way of setting up and looking for mapping information:

Listing 4-8

```
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @ORM\Entity
```

2. http://www.doctrine-project.org/

```
 */
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $title;

    /**
     * @ORM\Column(type="string")
     */
    private $slug;

    /**
     * @ORM\Column(type="text")
     */
    private $content;

    /**
     * @ORM\Column(type="string")
     */
    private $authorEmail;

    /**
     * @ORM\Column(type="datetime")
     */
    private $publishedAt;

    /**
     * @ORM\OneToMany(
     *      targetEntity="Comment",
     *      mappedBy="post",
     *      orphanRemoval=true
     * )
     * @ORM\OrderBy({"publishedAt" = "ASC"})
     */
    private $comments;

    public function __construct()
    {
        $this->publishedAt = new \DateTime();
        $this->comments = new ArrayCollection();
    }

    // getters and setters ...
}
```

All formats have the same performance, so this is once again ultimately a matter of taste.

Data Fixtures

As fixtures support is not enabled by default in Symfony, you should execute the following command to install the Doctrine fixtures bundle:

Listing 4-9
```
$ composer require "doctrine/doctrine-fixtures-bundle":"~2"
```

Then, enable the bundle in `AppKernel.php`, but only for the `dev` and `test` environments:

Listing 4-10
```
use Symfony\Component\HttpKernel\Kernel;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            // ...
        );

        if (in_array($this->getEnvironment(), array('dev', 'test'))) {
            // ...
            $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
        }

        return $bundles;
    }

    // ...
}
```

We recommend creating just *one fixture class*[3] for simplicity, though you're welcome to have more if that class gets quite large.

Assuming you have at least one fixtures class and that the database access is configured properly, you can load your fixtures by executing the following command:

Listing 4-11
```
$ php app/console doctrine:fixtures:load

Careful, database will be purged. Do you want to continue Y/N ? Y
  > purging database
  > loading AppBundle\DataFixtures\ORM\LoadFixtures
```

Coding Standards

The Symfony source code follows the *PSR-1*[4] and *PSR-2*[5] coding standards that were defined by the PHP community. You can learn more about *the Symfony Code Standards*[6] and even use the *PHP-CS-Fixer*[7], which is a command-line utility that can fix the coding standards of an entire codebase in a matter of seconds.

3. http://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html#writing-simple-fixtures
4. http://www.php-fig.org/psr/psr-1/
5. http://www.php-fig.org/psr/psr-2/
6. http://symfony.com/doc/current/contributing/code/standards.html
7. https://github.com/fabpot/PHP-CS-Fixer

Chapter 5
Controllers

Symfony follows the philosophy of *"thin controllers and fat models"*. This means that controllers should hold just the thin layer of *glue-code* needed to coordinate the different parts of the application.

As a rule of thumb, you should follow the 5-10-20 rule, where controllers should only define 5 variables or less, contain 10 actions or less and include 20 lines of code or less in each action. This isn't an exact science, but it should help you realize when code should be refactored out of the controller and into a service.

BEST PRACTICE

Make your controller extend the `FrameworkBundle` base Controller and use annotations to configure routing, caching and security whenever possible.

Coupling the controllers to the underlying framework allows you to leverage all of its features and increases your productivity.

And since your controllers should be thin and contain nothing more than a few lines of *glue-code*, spending hours trying to decouple them from your framework doesn't benefit you in the long run. The amount of time *wasted* isn't worth the benefit.

In addition, using annotations for routing, caching and security simplifies configuration. You don't need to browse tens of files created with different formats (YAML, XML, PHP): all the configuration is just where you need it and it only uses one format.

Overall, this means you should aggressively decouple your business logic from the framework while, at the same time, aggressively coupling your controllers and routing *to* the framework in order to get the most out of it.

Routing Configuration

To load routes defined as annotations in your controllers, add the following configuration to the main routing configuration file:

Listing 5-1
```yaml
# app/config/routing.yml
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
```

This configuration will load annotations from any controller stored inside the `src/AppBundle/Controller/` directory and even from its subdirectories. So if your application defines lots of controllers, it's perfectly ok to reorganize them into subdirectories:

Listing 5-2
```
<your-project>/
├─ ...
└─ src/
   └─ AppBundle/
      ├─ ...
      └─ Controller/
         ├─ DefaultController.php
         ├─ ...
         ├─ Api/
         │  ├─ ...
         │  └─ ...
         └─ Backend/
            ├─ ...
            └─ ...
```

Template Configuration

BEST PRACTICE

Don't use the `@Template()` annotation to configure the template used by the controller.

The `@Template` annotation is useful, but also involves some magic. For that reason, we recommend against using it.

Most of the time, `@Template` is used without any parameters, which makes it more difficult to know which template is being rendered. It also makes it less obvious to beginners that a controller should always return a Response object (unless you're using a view layer).

Lastly, the `@Template` annotation uses a `TemplateListener` class that hooks into the `kernel.view` event dispatched by the framework. This listener introduces a measurable performance impact. In the sample blog application, rendering the homepage took 5 milliseconds using the `$this->render()` method and 26 milliseconds using the `@Template` annotation.

How the Controller Looks

Considering all this, here is an example of how the controller should look for the homepage of our app:

Listing 5-3
```php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        $em = $this->getDoctrine()->getManager();
        $posts = $em->getRepository('App:Post')->findLatest();

        return $this->render('default/index.html.twig', array(
            'posts' => $posts
        ));
    }
}
```

Using the ParamConverter

If you're using Doctrine, then you can *optionally* use the *ParamConverter*[1] to automatically query for an entity and pass it as an argument to your controller.

BEST PRACTICE

Use the ParamConverter trick to automatically query for Doctrine entities when it's simple and convenient.

For example:

Listing 5-4

```php
/**
 * @Route("/{id}", name="admin_post_show")
 */
public function showAction(Post $post)
{
    $deleteForm = $this->createDeleteForm($post);

    return $this->render('admin/post/show.html.twig', array(
        'post'      => $post,
        'delete_form' => $deleteForm->createView(),
    ));
}
```

Normally, you'd expect a `$id` argument to `showAction`. Instead, by creating a new argument (`$post`) and type-hinting it with the `Post` class (which is a Doctrine entity), the ParamConverter automatically queries for an object whose `$id` property matches the `{id}` value. It will also show a 404 page if no `Post` can be found.

1. http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

When Things Get More Advanced

This works without any configuration because the wildcard name `{id}` matches the name of the property on the entity. If this isn't true, or if you have even more complex logic, the easiest thing to do is just query for the entity manually. In our application, we have this situation in `CommentController`:

Listing 5-5

```
/**
 * @Route("/comment/{postSlug}/new", name = "comment_new")
 */
public function newAction(Request $request, $postSlug)
{
    $post = $this->getDoctrine()
        ->getRepository('AppBundle:Post')
        ->findOneBy(array('slug' => $slug));

    if (!$post) {
        throw $this->createNotFoundException();
    }

    // ...
}
```

You can also use the `@ParamConverter` configuration, which is infinitely flexible:

Listing 5-6

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @Route("/comment/{postSlug}/new", name = "comment_new")
 * @ParamConverter("post", options={"mapping": {"postSlug": "slug"}})
 */
public function newAction(Request $request, Post $post)
{
    // ...
}
```

The point is this: the ParamConverter shortcut is great for simple situations. But you shouldn't forget that querying for entities directly is still very easy.

Pre and Post Hooks

If you need to execute some code before or after the execution of your controllers, you can use the EventDispatcher component to *set up before/after filters*[2].

2. `http://symfony.com/doc/current/cookbook/event_dispatcher/before_after_filters.html`

Chapter 6
Templates

When PHP was created 20 years ago, developers loved its simplicity and how well it blended HTML and dynamic code. But as time passed, other template languages - like *Twig*[1] - were created to make templating even better.

BEST PRACTICE

Use Twig templating format for your templates.

Generally speaking, PHP templates are much more verbose than in Twig because they lack native support for lots of modern features needed by templates, like inheritance, automatic escaping and named arguments for filters and functions.

Twig is the default templating format in Symfony and has the largest community support of all non-PHP template engines (it's used in high profile projects such as Drupal 8).

In addition, Twig is the only template format with guaranteed support in Symfony 3.0. As a matter of fact, PHP may be removed from the officially supported template engines.

Template Locations

BEST PRACTICE

Store all your application's templates in `app/Resources/views/` directory.

Traditionally, Symfony developers stored the application templates in the `Resources/views/` directory of each bundle. Then they used the logical name to refer to them (e.g. `AcmeDemoBundle:Default:index.html.twig`).

1. `http://twig.sensiolabs.org/`

But for the templates used in your application, it's much more convenient to store them in the `app/Resources/views/` directory. For starters, this drastically simplifies their logical names:

Templates stored inside bundles	Templates stored in app/
`AcmeDemoBunde:Default:index.html.twig`	`default/index.html.twig`
`::layout.html.twig`	`layout.html.twig`
`AcmeDemoBundle::index.html.twig`	`index.html.twig`
`AcmeDemoBundle:Default:subdir/index.html.twig`	`default/subdir/index.html.twig`
`AcmeDemoBundle:Default/subdir:index.html.twig`	`default/subdir/index.html.twig`

Another advantage is that centralizing your templates simplifies the work of your designers. They don't need to look for templates in lots of directories scattered through lots of bundles.

Twig Extensions

BEST PRACTICE

Define your Twig extensions in the `AppBundle/Twig/` directory and configure them using the `app/config/services.yml` file.

Our application needs a custom `md2html` Twig filter so that we can transform the Markdown contents of each post into HTML.

To do this, first, install the excellent *Parsedown*[2] Markdown parser as a new dependency of the project:

Listing 6-1
```
$ composer require erusev/parsedown
```

Then, create a new `Markdown` service that will be used later by the Twig extension. The service definition only requires the path to the class:

Listing 6-2
```
# app/config/services.yml
services:
    # ...
    markdown:
        class: AppBundle\Utils\Markdown
```

And the `Markdown` class just needs to define one single method to transform Markdown content into HTML:

Listing 6-3
```
namespace AppBundle\Utils;

class Markdown
{
    private $parser;

```

2. `http://parsedown.org/`

```
    public function __construct()
    {
        $this->parser = new \Parsedown();
    }

    public function toHtml($text)
    {
        $html = $this->parser->text($text);

        return $html;
    }
}
```

Next, create a new Twig extension and define a new filter called `md2html` using the `Twig_SimpleFilter` class. Inject the newly defined `markdown` service in the constructor of the Twig extension:

Listing 6-4

```
namespace AppBundle\Twig;

use AppBundle\Utils\Markdown;

class AppExtension extends \Twig_Extension
{
    private $parser;

    public function __construct(Markdown $parser)
    {
        $this->parser = $parser;
    }

    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter(
                'md2html',
                array($this, 'markdownToHtml'),
                array('is_safe' => array('html'))
            ),
        );
    }

    public function markdownToHtml($content)
    {
        return $this->parser->toHtml($content);
    }

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

Lastly define a new service to enable this Twig extension in the app (the service name is irrelevant because you never use it in your own code):

Listing 6-5

```
# app/config/services.yml
services:
    app.twig.app_extension:
```

```
4           class:     AppBundle\Twig\AppExtension
5           arguments: ["@markdown"]
6           tags:
7               - { name: twig.extension }
```

Chapter 7
Forms

Forms are one of the most misused Symfony components due to its vast scope and endless list of features. In this chapter we'll show you some of the best practices so you can leverage forms but get work done quickly.

Building Forms

BEST PRACTICE

Define your forms as PHP classes.

The Form component allows you to build forms right inside your controller code. Honestly, unless you need to reuse the form somewhere else, that's totally fine. But for organize and reuse, we recommend that you define each form in its own PHP class:

Listing 7-1

```php
namespace AppBundle\Form;

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

class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('summary', 'textarea')
            ->add('content', 'textarea')
            ->add('authorEmail', 'email')
            ->add('publishedAt', 'datetime')
        ;
```

```
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Post'
        ));
    }

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

To use the class, use `createForm` and instantiate the new class:

Listing 7-2

```
use AppBundle\Form\PostType;
// ...

public function newAction(Request $request)
{
    $post = new Post();
    $form = $this->createForm(new PostType(), $post);

    // ...
}
```

Registering Forms as Services

You can also *register your form type as a service*[1]. But this is *not* recommended unless you plan to reuse the new form type in many places or embed it in other forms directly or via the *collection type*[2].

For most forms that are used only to edit or create something, registering the form as a service is over-kill, and makes it more difficult to figure out exactly which form class is being used in a controller.

Form Button Configuration

Form classes should try to be agnostic to *where* they will be used. This makes them easier to re-use later.

BEST PRACTICE

Add buttons in the templates, not in the form classes or the controllers.

Since Symfony 2.5, you can add buttons as fields on your form. This is a nice way to simplify the template that renders your form. But if you add the buttons directly in your form class, this would effectively limit the scope of that form:

Listing 7-3

1. http://symfony.com/doc/current/cookbook/form/create_custom_field_type.html#creating-your-field-type-as-a-service
2. http://symfony.com/doc/current/reference/forms/types/collection.html

```php
class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('save', 'submit', array('label' => 'Create Post'))
        ;
    }

    // ...
}
```

This form *may* have been designed for creating posts, but if you wanted to reuse it for editing posts, the button label would be wrong. Instead, some developers configure form buttons in the controller:

Listing 7-4

```php
namespace AppBundle\Controller\Admin;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use AppBundle\Entity\Post;
use AppBundle\Form\PostType;

class PostController extends Controller
{
    // ...

    public function newAction(Request $request)
    {
        $post = new Post();
        $form = $this->createForm(new PostType(), $post);
        $form->add('submit', 'submit', array(
            'label' => 'Create',
            'attr'  => array('class' => 'btn btn-default pull-right')
        ));

        // ...
    }
}
```

This is also an important error, because you are mixing presentation markup (labels, CSS classes, etc.) with pure PHP code. Separation of concerns is always a good practice to follow, so put all the view-related things in the view layer:

Listing 7-5

```twig
<form method="POST" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Create"
           class="btn btn-default pull-right" />
</form>
```

Rendering the Form

There are a lot of ways to render your form, ranging from rendering the entire thing in one line to rendering each part of each field independently. The best way depends on how much customization you need.

The simplest way - which is especially useful during development - is to render the form tags manually and then use `form_widget()` to render all of the fields:

Listing 7-6
```
<form method="POST" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
</form>
```

BEST PRACTICE

Don't use the `form()` or `form_start()` functions to render the starting and ending form tags.

Experienced Symfony developers will recognize that we're rendering the `<form>` tags manually instead of using the `form_start()` or `form()` functions. While those are convenient, they take away from some clarity with little benefit.

The exception is a delete form because it's really just one button and so benefits from some of these extra shortcuts.

If you need more control over how your fields are rendered, then you should remove the `form_widget(form)` function and render your fields individually. See *How to Customize Form Rendering*[3] for more information on this and how you can control *how* the form renders at a global level using form theming.

Handling Form Submits

Handling a form submit usually follows a similar template:

Listing 7-7
```
public function newAction(Request $request)
{
    // build the form ...

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($post);
        $em->flush();

        return $this->redirect($this->generateUrl(
            'admin_post_show',
            array('id' => $post->getId())
```

3. http://symfony.com/doc/current/cookbook/form/form_customization.html

```
15          ));
16      }
17
18      // render the template
19  }
```

There are really only two notable things here. First, we recommend that you use a single action for both rendering the form and handling the form submit. For example, you *could* have a `newAction` that *only* renders the form and a `createAction` that *only* processes the form submit. Both those actions will be almost identical. So it's much simpler to let `newAction` handle everything.

Second, we recommend using `$form->isSubmitted()` in the `if` statement for clarity. This isn't technically needed, since `isValid()` first calls `isSubmitted()`. But without this, the flow doesn't read well as it *looks* like the form is *always* processed (even on the GET request).

Chapter 8
Internationalization

Internationalization and localization adapt the applications and their contents to the specific region or language of the users. In Symfony this is an opt-in feature that needs to be enabled before using it. To do this, uncomment the following `translator` configuration option and set your application locale:

Listing 8-1
```yaml
# app/config/config.yml
framework:
    # ...
    translator: { fallback: "%locale%" }

# app/config/parameters.yml
parameters:
    # ...
    locale:     en
```

Translation Source File Format

The Symfony Translation component supports lots of different translation formats: PHP, Qt, `.po`, `.mo`, JSON, CSV, INI, etc.

BEST PRACTICE

Use the XLIFF format for your translation files.

Of all the available translation formats, only XLIFF and gettext have broad support in the tools used by professional translators. And since it's based on XML, you can validate XLIFF file contents as you write them.

Symfony 2.6 added support for notes inside XLIFF files, making them more user-friendly for translators. At the end, good translations are all about context, and these XLIFF notes allow you to define that context.

The Apache-licensed *JMSTranslationBundle*[1] offers you a web interface for viewing and editing these translation files. It also has advanced extractors that can read your project and automatically update the XLIFF files.

Translation Source File Location

BEST PRACTICE

Store the translation files in the `app/Resources/translations/` directory.

Traditionally, Symfony developers have created these files in the `Resources/translations/` directory of each bundle.

But since the `app/Resources/` directory is considered the global location for the application's resources, storing translations in `app/Resources/translations/` centralizes them *and* gives them priority over any other translation file. This lets you override translations defined in third-party bundles.

Translation Keys

BEST PRACTICE

Always use keys for translations instead of content strings.

Using keys simplifies the management of the translation files because you can change the original contents without having to update all of the translation files.

Keys should always describe their *purpose* and *not* their location. For example, if a form has a field with the label "Username", then a nice key would be `label.username`, *not* `edit_form.label.username`.

Example Translation File

Applying all the previous best-practices, the sample translation file for English in the application would be:

Listing 8-2

```
<!-- app/Resources/translations/messages.en.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" target-language="en" datatype="plaintext">
        <body>
            <trans-unit id="1">
                <source>title.post_list</source>
                <target>Post List</target>
            </trans-unit>
        </body>
```

1. https://github.com/schmittjoh/JMSTranslationBundle

```
11      </file>
12  </xliff>
```

Chapter 9
Security

Authentication and Firewalls (i.e. Getting the User's Credentials)

You can configure Symfony to authenticate your users using any method you want and to load user information from any source. This is a complex topic, but the *Security Cookbook Section*[1] has a lot of information about this.

Regardless of your needs, authentication is configured in `security.yml`, primarily under the `firewalls` key.

BEST PRACTICE

Unless you have two legitimately different authentication systems and users (e.g. form login for the main site and a token system for your API only), we recommend having only *one* firewall entry with the `anonymous` key enabled.

Most applications only have one authentication system and one set of users. For this reason, you only need *one* firewall entry. There are exceptions of course, especially if you have separated web and API sections on your site. But the point is to keep things simple.

Additionally, you should use the `anonymous` key under your firewall. If you need to require users to be logged in for different sections of your site (or maybe nearly *all* sections), use the `access_control` area.

BEST PRACTICE

Use the `bcrypt` encoder for encoding your users' passwords.

1. `http://symfony.com/doc/current/cookbook/security/index.html`

If your users have a password, then we recommend encoding it using the `bcrypt` encoder, instead of the traditional SHA-512 hashing encoder. The main advantages of `bcrypt` are the inclusion of a *salt* value to protect against rainbow table attacks, and its adaptive nature, which allows to make it slower to remain resistant to brute-force search attacks.

With this in mind, here is the authentication setup from our application, which uses a login form to load users from the database:

Listing 9-1

```yaml
security:
    encoders:
        AppBundle\Entity\User: bcrypt

    providers:
        database_users:
            entity: { class: AppBundle:User, property: username }

    firewalls:
        secured_area:
            pattern: ^/
            anonymous: true
            form_login:
                check_path: security_login_check
                login_path: security_login_form

            logout:
                path: security_logout
                target: homepage

# ... access_control exists, but is not shown here
```

The source code for our project contains comments that explain each part.

Authorization (i.e. Denying Access)

Symfony gives you several ways to enforce authorization, including the `access_control` configuration in *security.yml*[2], the *@Security annotation* and using *isGranted* on the `security.context` service directly.

BEST PRACTICE

- For protecting broad URL patterns, use `access_control`;
- Whenever possible, use the `@Security` annotation;
- Check security directly on the `security.context` service whenever you have a more complex situation.

There are also different ways to centralize your authorization logic, like with a custom security voter or with ACL.

2. `http://symfony.com/doc/current/reference/configuration/security.html`

BEST PRACTICE

- For fine-grained restrictions, define a custom security voter;
- For restricting access to *any* object by *any* user via an admin interface, use the Symfony ACL.

The @Security Annotation

For controlling access on a controller-by-controller basis, use the `@Security` annotation whenever possible. It's easy to read and is placed consistently above each action.

In our application, you need the `ROLE_ADMIN` in order to create a new post. Using `@Security`, this looks like:

Listing 9-2

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
// ...

/**
 * Displays a form to create a new Post entity.
 *
 * @Route("/new", name="admin_post_new")
 * @Security("has_role('ROLE_ADMIN')")
 */
public function newAction()
{
    // ...
}
```

Using Expressions for Complex Security Restrictions

If your security logic is a little bit more complex, you can use an *expression*[3] inside `@Security`. In the following example, a user can only access the controller if their email matches the value returned by the `getAuthorEmail` method on the `Post` object:

Listing 9-3

```
use AppBundle\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Route("/{id}/edit", name="admin_post_edit")
 * @Security("user.getEmail() == post.getAuthorEmail()")
 */
public function editAction(Post $post)
{
    // ...
}
```

Notice that this requires the use of the *ParamConverter*[4], which automatically queries for the `Post` object and puts it on the `$post` argument. This is what makes it possible to use the `post` variable in the expression.

3. http://symfony.com/doc/current/components/expression_language/introduction.html
4. http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

This has one major drawback: an expression in an annotation cannot easily be reused in other parts of the application. Imagine that you want to add a link in a template that will only be seen by authors. Right now you'll need to repeat the expression code using Twig syntax:

Listing 9-4
```
{% if app.user and app.user.email == post.authorEmail %}
    <a href=""> ... </a>
{% endif %}
```

The easiest solution - if your logic is simple enough - is to add a new method to the `Post` entity that checks if a given user is its author:

Listing 9-5
```
// src/AppBundle/Entity/Post.php
// ...

class Post
{
    // ...

    /**
     * Is the given User the author of this Post?
     *
     * @return bool
     */
    public function isAuthor(User $user = null)
    {
        return $user && $user->getEmail() == $this->getAuthorEmail();
    }
}
```

Now you can reuse this method both in the template and in the security expression:

Listing 9-6
```
use AppBundle\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Route("/{id}/edit", name="admin_post_edit")
 * @Security("post.isAuthor(user)")
 */
public function editAction(Post $post)
{
    // ...
}
```

Listing 9-7
```
{% if post.isAuthor(app.user) %}
    <a href=""> ... </a>
{% endif %}
```

Checking Permissions without @Security

The above example with `@Security` only works because we're using the *ParamConverter*, which gives the expression access to the a `post` variable. If you don't use this, or have some other more advanced use-case, you can always do the same security check in PHP:

Listing 9-8

```php
/**
 * @Route("/{id}/edit", name="admin_post_edit")
 */
public function editAction($id)
{
    $post = $this->getDoctrine()->getRepository('AppBundle:Post')
        ->find($id);

    if (!$post) {
        throw $this->createNotFoundException();
    }

    if (!$post->isAuthor($this->getUser())) {
        throw $this->createAccessDeniedException();
    }

    // ...
}
```

Security Voters

If your security logic is complex and can't be centralized into a method like `isAuthor()`, you should leverage custom voters. These are an order of magnitude easier than *ACL's*[5] and will give you the flexibility you need in almost all cases.

First, create a voter class. The following example shows a voter that implements the same `getAuthorEmail` logic you used above:

Listing 9-9

```php
namespace AppBundle\Security;

use Symfony\Component\Security\Core\Authorization\Voter\AbstractVoter;
use Symfony\Component\Security\Core\User\UserInterface;

// AbstractVoter class requires Symfony 2.6 or higher version
class PostVoter extends AbstractVoter
{
    const CREATE = 'create';
    const EDIT   = 'edit';

    protected function getSupportedAttributes()
    {
        return array(self::CREATE, self::EDIT);
    }

    protected function getSupportedClasses()
    {
        return array('AppBundle\Entity\Post');
    }

    protected function isGranted($attribute, $post, $user = null)
    {
        if (!$user instanceof UserInterface) {
            return false;
        }
```

5. http://symfony.com/doc/current/cookbook/security/acl.html

```
27
28          if ($attribute == self::CREATE && in_array(ROLE_ADMIN, $user->getRoles())) {
29              return true;
30          }
31
32          if ($attribute == self::EDIT && $user->getEmail() === $post->getAuthorEmail()) {
33              return true;
34          }
35
36          return false;
37      }
38  }
```

To enable the security voter in the application, define a new service:

Listing 9-10

```
1  # app/config/services.yml
2  services:
3      # ...
4      post_voter:
5          class:      AppBundle\Security\PostVoter
6          public:     false
7          tags:
8             - { name: security.voter }
```

Now, you can use the voter with the `@Security` annotation:

Listing 9-11

```
1  /**
2   * @Route("/{id}/edit", name="admin_post_edit")
3   * @Security("is_granted('edit', post)")
4   */
5  public function editAction(Post $post)
6  {
7      // ...
8  }
```

You can also use this directly with the `security.context` service, or via the even easier shortcut in a controller:

Listing 9-12

```
1  /**
2   * @Route("/{id}/edit", name="admin_post_edit")
3   */
4  public function editAction($id)
5  {
6      $post = // query for the post ...
7
8      if (!$this->get('security.context')->isGranted('edit', $post)) {
9          throw $this->createAccessDeniedException();
10     }
11 }
```

Learn More

The *FOSUserBundle*[6], developed by the Symfony community, adds support for a database-backed user system in Symfony2. It also handles common tasks like user registration and forgotten password functionality.

Enable the *Remember Me feature*[7] to allow your users to stay logged in for a long period of time.

When providing customer support, sometimes it's necessary to access the application as some *other* user so that you can reproduce the problem. Symfony provides the ability to *impersonate users*[8].

If your company uses a user login method not supported by Symfony, you can develop *your own user provider*[9] and *your own authentication provider*[10].

6. https://github.com/FriendsOfSymfony/FOSUserBundle
7. http://symfony.com/doc/current/cookbook/security/remember_me.html
8. http://symfony.com/doc/current/cookbook/security/impersonating_user.html
9. http://symfony.com/doc/current/cookbook/security/custom_provider.html
10. http://symfony.com/doc/current/cookbook/security/custom_authentication_provider.html

Chapter 10
Web Assets

Web assets are things like CSS, JavaScript and image files that make the frontend of your site look and work great. Symfony developers have traditionally stored these assets in the `Resources/public/` directory of each bundle.

BEST PRACTICE

Store your assets in the `web/` directory.

Scattering your web assets across tens of different bundles makes it more difficult to manage them. Your designers' lives will be much easier if all the application assets are in one location.

Templates also benefit from centralizing your assets, because the links are much more concise:

Listing 10-1
```
<link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}" />
<link rel="stylesheet" href="{{ asset('css/main.css') }}" />

{# ... #}

<script src="{{ asset('js/jquery.min.js') }}"></script>
<script src="{{ asset('js/bootstrap.min.js') }}"></script>
```

Keep in mind that `web/` is a public directory and that anything stored here will be publicly accessible. For that reason, you should put your compiled web assets here, but not their source files (e.g. SASS files).

Using Assetic

These days, you probably can't simply create static CSS and JavaScript files and include them in your template. Instead, you'll probably want to combine and minify these to improve client-side performance.

You may also want to use LESS or Sass (for example), which means you'll need some way to process these into CSS files.

A lot of tools exist to solve these problems, including pure-frontend (non-PHP) tools like GruntJS.

BEST PRACTICE

Use Assetic to compile, combine and minimize web assets, unless you're comfortable with frontend tools like GruntJS.

Assetic[1] is an asset manager capable of compiling assets developed with a lot of different frontend technologies like LESS, Sass and CoffeScript. Combining all your assets with Assetic is a matter of wrapping all the assets with a single Twig tag:

Listing 10-2

```
{% stylesheets
    'css/bootstrap.min.css'
    'css/main.css'
    filter='cssrewrite' output='css/compiled/all.css' %}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}

{# ... #}

{% javascripts
    'js/jquery.min.js'
    'js/bootstrap.min.js'
    output='js/compiled/all.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Frontend-Based Applications

Recently, frontend technologies like AngularJS have become pretty popular for developing frontend web applications that talk to an API.

If you are developing an application like this, you should use the tools that are recommended by the technology, such as Bower and GruntJS. You should develop your frontend application separately from your Symfony backend (even separating the repositories if you want).

Learn More about Assetic

Assetic can also minimize CSS and JavaScript assets *using UglifyCSS/UglifyJS*[2] to speed up your websites. You can even *compress images*[3] with Assetic to reduce their size before serving them to the user. Check out the *official Assetic documentation*[4] to learn more about all the available features.

1. http://symfony.com/doc/current/cookbook/assetic/asset_management.html
2. http://symfony.com/doc/current/cookbook/assetic/uglifyjs.html
3. http://symfony.com/doc/current/cookbook/assetic/jpeg_optimize.html
4. https://github.com/kriswallsmith/assetic

Chapter 11
Tests

Roughly speaking, there are two types of test. Unit testing allows you to test the input and output of specific functions. Functional testing allows you to command a "browser" where you browse to pages on your site, click links, fill out forms and assert that you see certain things on the page.

Unit Tests

Unit tests are used to test your "business logic", which should live in classes that are independent of Symfony. For that reason, Symfony doesn't really have an opinion on what tools you use for unit testing. However, the most popular tools are *PhpUnit*[1] and *PhpSpec*[2].

Functional Tests

Creating really good functional tests can be tough so some developers skip these completely. Don't skip the functional tests! By defining some *simple* functional tests, you can quickly spot any big errors before you deploy them:

BEST PRACTICE

Define a functional test that at least checks if your application pages are successfully loading.

A functional test can be as easy as this:

Listing 11-1
```
1  /** @dataProvider provideUrls */
2  public function testPageIsSuccessful($url)
3  {
```

1. https://phpunit.de/
2. http://www.phpspec.net/

```
4       $client = self::createClient();
5       $client->request('GET', $url);
6
7       $this->assertTrue($client->getResponse()->isSuccessful());
8   }
9
10  public function provideUrls()
11  {
12      return array(
13          array('/'),
14          array('/posts'),
15          array('/post/fixture-post-1'),
16          array('/blog/category/fixture-category'),
17          array('/archives'),
18          // ...
19      );
20  }
```

This code checks that all the given URLs load successfully, which means that their HTTP response status code is between `200` and `299`. This may not look that useful, but given how little effort this took, it's worth having it in your application.

Hardcode URLs in a Functional Test

Some of you may be asking why the previous functional test doesn't use the URL generator service:

BEST PRACTICE

Hardcode the URLs used in the functional tests instead of using the URL generator.

Consider the following functional test that uses the `router` service to generate the URL of the tested page:

Listing 11-2
```
1  public function testBlogArchives()
2  {
3      $client = self::createClient();
4      $url = $client->getContainer()->get('router')->generate('blog_archives');
5      $client->request('GET', $url);
6
7      // ...
8  }
```

This will work, but it has one *huge* drawback. If a developer mistakenly changes the path of the `blog_archives` route, the test will still pass, but the original (old) URL won't work! This means that any bookmarks for that URL will be broken and you'll lose any search engine page ranking.

Testing JavaScript Functionality

The built-in functional testing client is great, but it can't be used to test any JavaScript behavior on your pages. If you need to test this, consider using the *Mink*[3] library from within PHPUnit.

3. `http://mink.behat.org`

Of course, if you have a heavy JavaScript frontend, you should consider using pure JavaScript-based testing tools.

Learn More about Functional Tests

Consider using *Faker*[4] and *Alice*[5] libraries to generate real-looking data for your test fixtures.

4. `https://github.com/fzaninotto/Faker`
5. `https://github.com/nelmio/alice`

This is an early preview release of the

Official Symfony Best Practices

Visit **symfony.com/best-practices** to get the latest version of this document.

Visit **bit.ly/best-practices-workshop** to check out the next dates of the Official Symfony Best Practices workshop.