Symfony

The Best Practices Book

Version: 4.0

generated on March 3, 2018

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 unneeded for web applications. Much of the time, 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 **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].

Best practice is a noun that means *"a well defined procedure that is known to produce near-optimum results"*. And that's exactly what this guide aims to provide. Even if you don't agree with every recommendation, we believe these will help you build great applications with less complexity.

This guide is **specially suited** for:

- Websites and web applications developed with the full-stack Symfony Framework.

For other situations, this guide might be a good **starting point** that you can then **extend and fit to your specific needs**:

- Bundles shared publicly to the Symfony community;
- Advanced developers or teams who have created their own standards;
- Some complex applications that have highly customized requirements;
- Bundles that may be shared internally within a company.

1. `https://connect.sensiolabs.com/profile/fabpot`

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! and read the *Getting Started guides* first.

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, a sample application called *Symfony Demo*[2] has been developed with all these best practices in mind. Execute this command to download the demo application:

Listing 1-1
```
$ composer create-project symfony/symfony-demo
```

The demo application is a simple blog engine, because that will allow us to focus on the Symfony concepts and features without getting buried in difficult implementation details. Instead of developing the application step by step in this guide, you'll find selected snippets of code through the chapters.

Don't Update Your Existing Applications

After reading this handbook, some of you may be considering refactoring your existing Symfony applications. Our recommendation is sound and clear: you may use these best practices for **new applications** but **you should not refactor your existing applications to comply with these best practices**. The reasons for not doing it are various:

- Your existing applications are not wrong, they just follow another set of guidelines;
- A full codebase refactorization is prone to introduce errors in your applications;
- The amount of work spent on this could be better dedicated to improving your tests or adding features that provide real value to the end users.

Next: *Creating the Project*

2. https://github.com/symfony/demo

Chapter 2
Creating the Project

Installing Symfony

Use Composer and Symfony Flex to create and manage Symfony applications.

Composer[1] is the package manager used by modern PHP applications to manage their dependencies. *Symfony Flex*[2] is a Composer plugin designed to automate some of the most common tasks performed in Symfony applications. Using Flex is optional but recommended because it improves your productivity significantly.

Use the Symfony Skeleton to create new Symfony-based projects.

The *Symfony Skeleton*[3] is a minimal and empty Symfony project which you can base your new projects on. Unlike past Symfony versions, this skeleton installs the absolute bare minimum amount of dependencies to make a fully working Symfony project. Read the *Installing & Setting up the Symfony Framework* article to learn more about installing Symfony.

Creating the Blog Application

In your command console, browse to a directory where you have permission to create files and execute the following commands:

Listing 2-1
```
1 $ cd projects/
2 $ composer create-project symfony/skeleton blog
```

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

1. https://getcomposer.org/
2. https://github.com/symfony/flex
3. https://github.com/symfony/skeleton

The technical requirements to run Symfony are simple. If you want to check if your system meets those requirements, read *Requirements for Running Symfony*.

Structuring the Application

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

Listing 2-2

```
blog/
├─ bin/
│  └─ console
├─ config/
└─ public/
│  └─ index.php
├─ src/
│  └─ Kernel.php
├─ var/
│  ├─ cache/
│  └─ log/
└─ vendor/
```

This file and directory hierarchy is the convention proposed by Symfony to structure your applications. It's recommended to keep this structure because it's easy to navigate and most directory names are self-explanatory, but you can *override the location of any Symfony directory*:

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 used 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, if InvoiceBundle depends on ProductBundle, then there's no advantage to having two separate bundles.

> *Don't create any bundle to organize your application logic.*

Symfony applications can still use third-party bundles (installed in `vendor/`) to add features, but you should use PHP namespaces instead of bundles to organize your own code.

Next: *Configuration*

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

These are the options that change from one machine to another (e.g. from your development machine to the production server) but which don't change the application behavior.

> *Define the infrastructure-related configuration options as environment variables. During development, use the `.env` file at the root of your project to set these.*

By default, Symfony adds these types of options to the `.env` file when installing new dependencies in the app:

Listing 3-1
```
# .env
###> doctrine/doctrine-bundle ###
DATABASE_URL=sqlite:///%kernel.project_dir%/var/data/blog.sqlite
###< doctrine/doctrine-bundle ###

###> symfony/swiftmailer-bundle ###
MAILER_URL=smtp://localhost?encryption=ssl&auth_mode=login&username=&password=
###< symfony/swiftmailer-bundle ###

# ...
```

These options aren't defined inside the `config/services.yaml` 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.

Beware that dumping the contents of the `$_SERVER` and `$_ENV` variables or outputting the `phpinfo()` contents will display the values of the environment variables, exposing sensitive information such as the database credentials.

Canonical Parameters

Define all your application's env vars in the `.env.dist` file.

Symfony includes a configuration file called `.env.dist` at the project root, which stores the canonical list of environment variables for the application.

Whenever a new env var is defined for the application, you should also add it to this file and submit the changes to your version control system so your workmates can update their `.env` files.

Application-Related Configuration

Define the application behavior related configuration options in the `config/services.yaml` file.

The `services.yaml` 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 `.env` 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 `services.yaml` may vary from one *environment* to another. That's why Symfony supports defining `config/services_dev.yaml` and `config/services_prod.yaml` 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.

Use constants to define configuration options that rarely change.

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
# config/services.yaml
parameters:
    homepage.number_of_items: 10
```

If you've done something like this in the past, it's likely that you've in fact *never* actually needed to change that value. Creating a configuration option for a value that you are never going to configure just isn't necessary. Our recommendation is to define these values as constants in your application. You could, for example, define a `NUMBER_OF_ITEMS` constant in the `Post` entity:

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

class Post
{
    const NUMBER_OF_ITEMS = 10;

    // ...
}
```

1. https://en.wikipedia.org/wiki/Feature_toggle

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 with access to the Symfony container.

Constants can be used for example in your Twig templates thanks to the *constant() function*[2]:

Listing 3-4
```twig
<p>
    Displaying the {{ constant('NUMBER_OF_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 App\Repository;

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

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

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

Parameter Naming

> *The name of your configuration parameters should be as short as possible and should include a common prefix for the entire application.*

Using `app.` as the prefix of your parameters is a common practice to avoid collisions with Symfony and third-party bundles/libraries parameters. Then, use just one or two words to describe the purpose of the parameter:

Listing 3-6
```yaml
# config/services.yaml
parameters:
    # don't do this: 'dir' is too generic and it doesn't convey any meaning
    app.dir: '...'
    # do this: short but easy to understand names
    app.contents_dir: '...'
    # it's OK to use dots, underscores, dashes or nothing, but always
    # be consistent and use the same format for all the parameters
    app.dir.contents: '...'
    app.contents-dir: '...'
```

Next: *Organizing Your Business Logic*

2. http://twig.sensiolabs.org/doc/functions/constant.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.

For most projects, you should store all your code inside the `src/` directory. Inside here, you can create whatever directories you want to organize things:

Listing 4-1
```
symfony-project/
├─ config/
├─ public/
├─ src/
│  └─ Utils/
│     └─ MyClass.php
├─ tests/
├─ var/
└─ vendor/
```

Services: Naming and Configuration

Use autowiring to automate the configuration of application services.

Service autowiring is a feature provided by Symfony's Service Container to manage services with minimal configuration. It reads the type-hints on your constructor (or other methods) and automatically passes the correct services to each method. It can also add *service tags* to the services needed them, such as Twig extensions, event subscribers, etc.

The blog application needs a utility that can transform a post title (e.g. "Hello World") into a slug (e.g. "hello-world") to include it as part of the post URL. Let's create a new `Slugger` class inside `src/Utils/`:

1. https://en.wikipedia.org/wiki/Business_logic

Listing 4-2
```php
// src/Utils/Slugger.php
namespace App\Utils;

class Slugger
{
    public function slugify(string $value): string
    {
        // ...
    }
}
```

If you're using the default services.yaml configuration, this class is auto-registered as a service whose ID is `App\Utils\Slugger` (or simply `Slugger::class` if the class is already imported in your code).

> *The id of your application's services should be equal to their class name, except when you have multiple services configured for the same class (in that case, use a snake case id).*

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

Listing 4-3
```php
use App\Utils\Slugger;

public function create(Request $request, Slugger $slugger)
{
    // ...

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

        // ...
    }
}
```

Services can also be public or private. If you use the default services.yaml configuration, all services are private by default.

> *Services should be `private` whenever possible. This will prevent you from accessing that service via `$container->get()`. Instead, you will need to use dependency injection.*

Service Format: YAML

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

> *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.

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 or 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 `src/Entity/` directory.

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

Listing 4-4
```
symfony-project/
├─ ...
└─ src/
   └─ Entity/
      ├─ Comment.php
      ├─ Post.php
      └─ User.php
```

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.

> *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 4-5
```
namespace App\Entity;

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

/**
 * @ORM\Entity
 */
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")
```

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

```
     */
    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 4-6

```
$ composer require "doctrine/doctrine-fixtures-bundle"
```

Then, this bundle is enabled automatically, but only for the `dev` and `test` environments:

Listing 4-7

```
// config/bundles.php

return [
    // ...
    Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle::class => ['dev' => true, 'test' => true],
];
```

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 4-8

```
$ php bin/console doctrine:fixtures:load

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

3. https://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html#writing-simple-fixtures

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 Coding standards* and even use the *PHP-CS-Fixer*[6], which is a command-line utility that can fix the coding standards of an entire codebase in a matter of seconds.

Next: *Controllers*

4. https://www.php-fig.org/psr/psr-1/
5. https://www.php-fig.org/psr/psr-2/
6. https://github.com/FriendsOfPHP/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.

Your controller methods should just call to other services, trigger some events if needed and then return a response, but they should not contain any actual business logic. If they do, refactor it out of the controller and into a service.

> *Make your controller extend the `AbstractController` base controller provided by Symfony 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.

Controller Action Naming

> *Don't add the `Action` suffix to the methods of the controller actions.*

The first Symfony versions required that controller method names ended in `Action` (e.g. `newAction()`, `showAction()`). This suffix became optional when annotations were introduced for controllers. In modern Symfony applications this suffix is neither required nor recommended, so you can safely remove 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
# config/routes.yaml
controllers:
    resource: '../src/Controller/'
    type:     annotation
```

This configuration will load annotations from any controller stored inside the `src/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/
   ├─ ...
   └─ Controller/
      ├─ DefaultController.php
      ├─ ...
      ├─ Api/
      │  ├─ ...
      │  └─ ...
      └─ Backend/
         ├─ ...
         └─ ...
```

Template Configuration

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

The `@Template` annotation is useful, but also involves some magic. We don't think its benefit is worth the magic, and so 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).

What does the Controller look like

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

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

use App\Entity\Post;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class DefaultController extends AbstractController
{
    /**
     * @Route("/", name="homepage")
     */
    public function index()
    {
```

```
        $posts = $this->getDoctrine()
            ->getRepository(Post::class)
            ->findLatest();

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

Fetching Services

If you extend the base `AbstractController` class, you can't access services directly from the container via `$this->container->get()` or `$this->get()`. Instead, you must use dependency injection to fetch services: most easily done by type-hinting action method arguments:

> *Don't use `$this->get()` or `$this->container->get()` to fetch services from the container. Instead, use dependency injection.*

By not fetching services directly from the container, you can make your services *private*, which has several advantages.

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.

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

For example:

Listing 5-4

```
use App\Entity\Post;
use Symfony\Component\Routing\Annotation\Route;

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

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

Normally, you'd expect a `$id` argument to `show()`. Instead, by creating a new argument (`$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. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

When Things Get More Advanced

The above example 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 new(Request $request, $postSlug)
{
    $post = $this->getDoctrine()
        ->getRepository(Post::class)
        ->findOneBy(['slug' => $postSlug]);

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

    // ...
}
```

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

Listing 5-6

```
use App\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

/**
 * @Route("/comment/{postSlug}/new", name="comment_new")
 * @ParamConverter("post", options={"mapping"={"postSlug"="slug"}})
 */
public function new(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 and after filters*.

Next: *Templates*

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.

Use Twig templating format for your templates.

Generally speaking, PHP templates are more verbose than Twig templates 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).

Template Locations

Store the application templates in the `templates/` directory at the root of your project.

Centralizing your templates in a single location simplifies the work of your designers. In addition, using this directory simplifies the notation used when referring to templates (e.g. `$this->render('admin/post/show.html.twig')` instead of `$this->render('@SomeTwigNamespace/Admin/Posts/show.html.twig')`).

Use lowercased snake_case for directory and template names.

This recommendation aligns with Twig best practices, where variables and template names use lowercased snake_case too (e.g. `user_profile` instead of `userProfile` and `edit_form.html.twig` instead of `EditForm.html.twig`).

Use a prefixed underscore for partial templates in template names.

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

You often want to reuse template code using the `include` function to avoid redundant code. To determine those partials easily in the filesystem you should prefix partials and any other template without HTML body or `extends` tag with a single underscore.

Twig Extensions

> *Define your Twig extensions in the `src/Twig/` directory. Your application will automatically detect them and configure them.*

Our application needs a custom `md2html` Twig filter so that we can transform the Markdown contents of each post into HTML. To do this, create a new `Markdown` class that will be used later by the Twig extension. It just needs to define one single method to transform Markdown content into HTML:

Listing 6-1

```php
namespace App\Utils;

class Markdown
{
    // ...

    public function toHtml(string $text): string
    {
        return $this->parser->text($text);
    }
}
```

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

Listing 6-2

```php
namespace App\Twig;

use App\Utils\Markdown;
use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class AppExtension extends AbstractExtension
{
    private $parser;

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

    public function getFilters()
    {
        return [
            new TwigFilter('md2html', [$this, 'markdownToHtml'], [
                'is_safe' => ['html'],
                'pre_escape' => 'html',
            ]),
        ];
    }

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

And that's it!

If you're using the default services.yaml configuration, you're done! Symfony will automatically know about your new service and tag it to be used as a Twig extension.

Next: *Forms*

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

Define your forms as PHP classes.

The Form component allows you to build forms right inside your controller code. This is perfectly fine if you don't need to reuse the form somewhere else. But for organization and reuse, we recommend that you define each form in its own PHP class:

Listing 7-1
```php
namespace App\Form;

use App\Entity\Post;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\DateTimeType;

class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('summary', TextareaType::class)
            ->add('content', TextareaType::class)
            ->add('authorEmail', EmailType::class)
            ->add('publishedAt', DateTimeType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
```

```
        $resolver->setDefaults([
            'data_class' => Post::class,
        ]);
    }
}
```

Put the form type classes in the `App\Form` namespace, unless you use other custom form classes like data transformers.

To use the class, use `createForm()` and pass the fully qualified class name:

Listing 7-2

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

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

    // ...
}
```

Form Button Configuration

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

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

The Symfony Form component allows you to 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

```
class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('save', SubmitType::class, ['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

```
namespace App\Controller\Admin;

use App\Entity\Post;
use App\Form\PostType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class PostController extends Controller
{
    // ...

```

```
    public function new(Request $request)
    {
        $post = new Post();
        $form = $this->createForm(PostType::class, $post);
        $form->add('submit', SubmitType::class, [
            'label' => 'Create',
            'attr' => ['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

```
{{ form_start(form) }}
    {{ form_widget(form) }}

    <input type="submit" class="btn" value="Create" />
{{ form_end(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.

One of the simplest ways - which is especially useful during development - is to render the form tags and use the `form_widget()` function to render all of the fields:

Listing 7-6

```
{{ form_start(form, {attr: {class: 'my-form-class'} }) }}
    {{ form_widget(form) }}
{{ form_end(form) }}
```

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* 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 new(Request $request)
{
    // build the form ...

    $form->handleRequest($request);

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

        return $this->redirectToRoute('admin_post_show', [
            'id' => $post->getId()
        ]);
```

```
15      }
16
17      // render the template
18  }
```

We recommend that you use a single action for both rendering the form and handling the form submit. For example, you *could* have a `new()` action that *only* renders the form and a `create()` action that *only* processes the form submit. Both those actions will be almost identical. So it's much simpler to let `new()` handle everything.

Next: *Internationalization*

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 installed before using it (`composer require translation`).

Translation Source File Location

Store the translation files in the `translations/` directory at the root of your project.

Your translators' lives will be much easier if all the application translations are in one central location.

Translation Source File Format

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

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 supports notes in 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 *PHP Translation Bundle*[1] includes advanced extractors that can read your project and automatically update the XLIFF files.

1. `https://github.com/php-translation/symfony-bundle`

Translation Keys

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 8-1
```xml
<!-- translations/messages.en.xlf -->
<?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" original="file.ext">
        <body>
            <trans-unit id="title_post_list">
                <source>title.post_list</source>
                <target>Post List</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Next: *Security*

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 guide* has a lot of information about this.

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

> *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.

> *Use the `bcrypt` encoder for hashing your users' passwords.*

If your users have a password, then we recommend hashing 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.

Argon2i is the hashing algorithm as recommended by industry standards, but this won't be available to you unless you are using PHP 7.2+ or have the *libsodium*[1] extension installed. `bcrypt` is sufficient for most applications.

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
# config/packages/security.yaml
security:
    encoders:
        App\Entity\User: bcrypt

    providers:
        database_users:
            entity: { class: App\Entity\User, property: username }

    firewalls:
        secured_area:
            pattern: ^/
            anonymous: true
            form_login:
                check_path: login
                login_path: login

            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.yaml*, the @Security annotation and using isGranted on the `security.authorization_checker` service directly.

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

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

> *Define a custom security voter to implement fine-grained restrictions.*

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

1. `https://pecl.php.net/package/libsodium`

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
use Symfony\Component\Routing\Annotation\Route;
// ...

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

Using Expressions for Complex Security Restrictions

If your security logic is a little bit more complex, you can use an *expression* 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

```php
use App\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
use Symfony\Component\Routing\Annotation\Route;

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

Notice that this requires the use of the *ParamConverter*[2], 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.

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

```twig
{% 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

```php
// src/Entity/Post.php
// ...

class Post
{
    // ...

    /**
     * Is the given User the author of this Post?
     *
     * @return bool
     */
```

2. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

```
13      public function isAuthor(User $user = null)
14      {
15          return $user && $user->getEmail() == $this->getAuthorEmail();
16      }
17  }
```

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

Listing 9-6

```
1  use App\Entity\Post;
2  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
3  use Symfony\Component\Routing\Annotation\Route;
4
5  /**
6   * @Route("/{id}/edit", name="admin_post_edit")
7   * @Security("post.isAuthor(user)")
8   */
9  public function edit(Post $post)
10 {
11     // ...
12 }
```

Listing 9-7

```
1  {% if post.isAuthor(app.user) %}
2      <a href=""> ... </a>
3  {% 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 `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

```
1  /**
2   * @Route("/{id}/edit", name="admin_post_edit")
3   */
4  public function edit($id)
5  {
6      $post = $this->getDoctrine()
7          ->getRepository(Post::class)
8          ->find($id);
9
10     if (!$post) {
11         throw $this->createNotFoundException();
12     }
13
14     if (!$post->isAuthor($this->getUser())) {
15         $this->denyAccessUnlessGranted('edit', $post);
16     }
17     // equivalent code without using the "denyAccessUnlessGranted()" shortcut:
18     //
19     // use Symfony\Component\Security\Core\Exception\AccessDeniedException;
20     // use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface
21     //
22     // ...
23     //
24     // public function __construct(AuthorizationCheckerInterface $authorizationChecker) {
25     //      $this->authorizationChecker = $authorizationChecker;
26     // }
27     //
28     // ...
29     //
30     // if (!$this->authorizationChecker->isGranted('edit', $post)) {
31     //     throw $this->createAccessDeniedException();
32     // }
```

```
33      //
34      // ...
35  }
```

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 much easier than *ACLs* 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

```
1   namespace App\Security;
2
3   use App\Entity\Post;
4   use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
5   use Symfony\Component\Security\Core\Authorization\AccessDecisionManagerInterface;
6   use Symfony\Component\Security\Core\Authorization\Voter\Voter;
7   use Symfony\Component\Security\Core\User\UserInterface;
8
9   class PostVoter extends Voter
10  {
11      const CREATE = 'create';
12      const EDIT   = 'edit';
13
14      private $decisionManager;
15
16      public function __construct(AccessDecisionManagerInterface $decisionManager)
17      {
18          $this->decisionManager = $decisionManager;
19      }
20
21      protected function supports($attribute, $subject)
22      {
23          if (!in_array($attribute, [self::CREATE, self::EDIT])) {
24              return false;
25          }
26
27          if (!$subject instanceof Post) {
28              return false;
29          }
30
31          return true;
32      }
33
34      protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
35      {
36          $user = $token->getUser();
37          /** @var Post */
38          $post = $subject; // $subject must be a Post instance, thanks to the supports method
39
40          if (!$user instanceof UserInterface) {
41              return false;
42          }
43
44          switch ($attribute) {
45              // if the user is an admin, allow them to create new posts
46              case self::CREATE:
47                  if ($this->decisionManager->decide($token, ['ROLE_ADMIN'])) {
48                      return true;
49                  }
50
51                  break;
52
53              // if the user is the author of the post, allow them to edit the posts
```

```
            case self::EDIT:
                if ($user->getEmail() === $post->getAuthorEmail()) {
                    return true;
                }

                break;
        }

        return false;
    }
}
```

If you're using the default services.yaml configuration, your application will autoconfigure your security voter and inject an `AccessDecisionManagerInterface` instance into it thanks to *autowiring*.

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

Listing 9-10

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

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

Listing 9-11

```
/**
 * @Route("/{id}/edit", name="admin_post_edit")
 */
public function edit($id)
{
    $post = ...; // query for the post

    $this->denyAccessUnlessGranted('edit', $post);

    // use Symfony\Component\Security\Core\Exception\AccessDeniedException;
    // use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface
    //
    // ...
    //
    // public function __construct(AuthorizationCheckerInterface $authorizationChecker) {
    //      $this->authorizationChecker = $authorizationChecker;
    // }
    //
    // ...
    //
    // if (!$this->authorizationChecker->isGranted('edit', $post)) {
    //    throw $this->createAccessDeniedException();
    // }
    //
    // ...
}
```

Learn More

The *FOSUserBundle*[3], developed by the Symfony community, adds support for a database-backed user system in Symfony. It also handles common tasks like user registration and forgotten password functionality.

3. https://github.com/FriendsOfSymfony/FOSUserBundle

Enable the *Remember Me feature* 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*.

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

Next: *Web Assets*

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.

Store your assets in the `assets/` directory at the root of your project.

Your designers' and front-end developers' lives will be much easier if all the application assets are in one central location.

Use Webpack Encore[1] to compile, combine and minimize web assets.

Webpack[2] is the leading JavaScript module bundler that compiles, transforms and packages assets for usage in a browser. Webpack Encore is a JavaScript library that gets rid of most of Webpack complexity without hiding any of its features or distorting its usage and philosophy.

Webpack Encore was designed to bridge the gap between Symfony applications and the JavaScript-based tools used in modern web applications. Check out the *official Webpack Encore documentation*[3] to learn more about all the available features.

Next: *Tests*

1. *https://github.com/symfony/webpack-encore*
2. https://webpack.js.org/
3. https://symfony.com/doc/current/frontend.html

Chapter 11
Tests

Of all the different types of test available, these best practices focus solely on unit and functional tests. 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:

> *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
```php
// tests/ApplicationAvailabilityFunctionalTest.php
namespace App\Tests;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ApplicationAvailabilityFunctionalTest extends WebTestCase
{
    /**
     * @dataProvider urlProvider
```

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

```
     */
    public function testPageIsSuccessful($url)
    {
        $client = self::createClient();
        $client->request('GET', $url);

        $this->assertTrue($client->getResponse()->isSuccessful());
    }

    public function urlProvider()
    {
        yield ['/'];
        yield ['/posts'];
        yield ['/post/fixture-post-1'];
        yield ['/blog/category/fixture-category'];
        yield ['/archives'];
        // ...
    }
}
```

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.

In computer software, this kind of test is called *smoke testing*[3] and consists of *"preliminary testing to reveal simple failures severe enough to reject a prospective software release"*.

Hardcode URLs in a Functional Test

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

> *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
```
// ...
private $router; // consider that this holds the Symfony router service

public function testBlogArchives()
{
    $client = self::createClient();
    $url = $this->router->generate('blog_archives');
    $client->request('GET', $url);

    // ...
}
```

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*[4] library from within PHPUnit.

3. https://en.wikipedia.org/wiki/Smoke_testing_(software)
4. http://mink.behat.org

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

Learn More about Functional Tests

Consider using the *HautelookAliceBundle*[5] to generate real-looking data for your test fixtures using *Faker*[6] and *Alice*[7].

5. https://github.com/hautelook/AliceBundle
6. https://github.com/fzaninotto/Faker
7. https://github.com/nelmio/alice