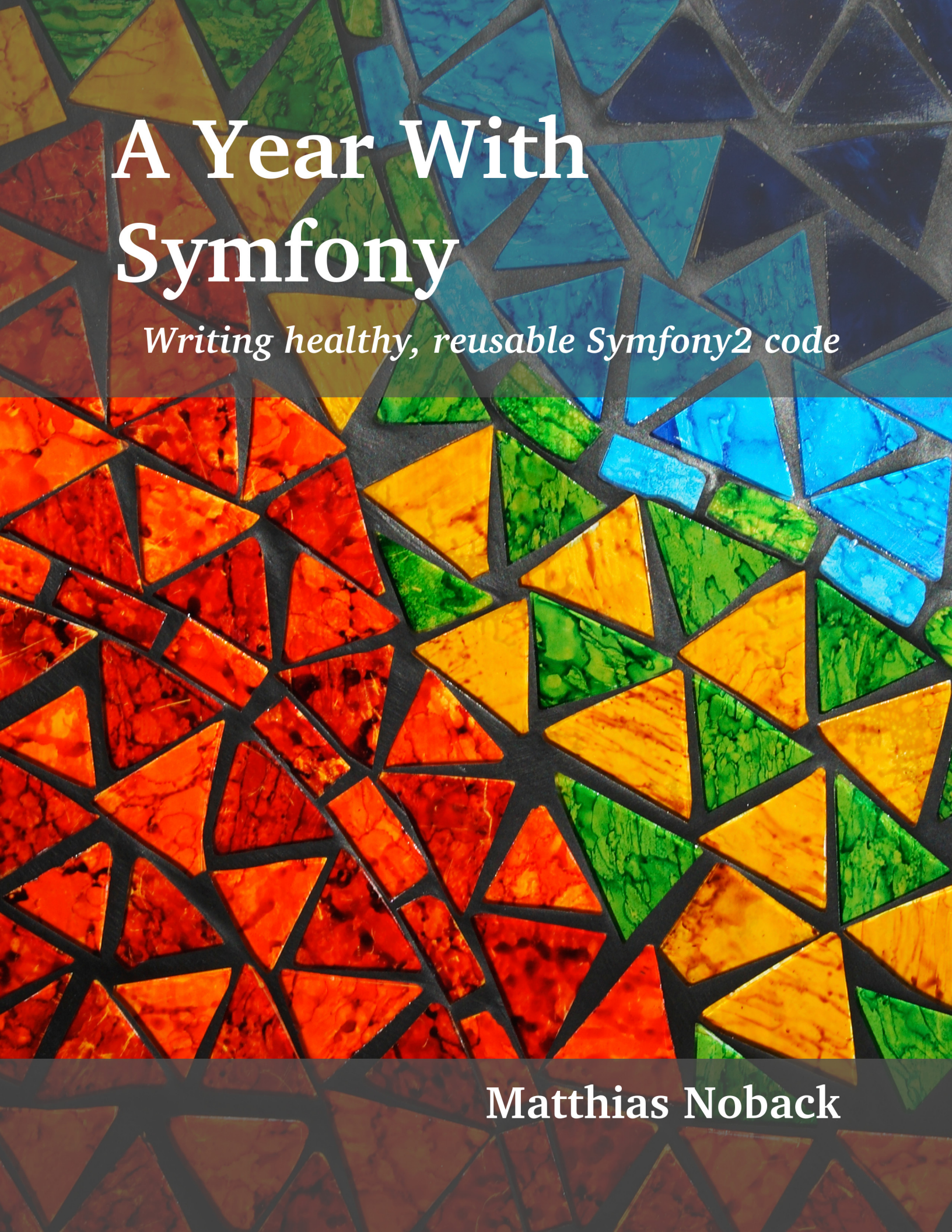
A Year With Symfony
Writing healthy, reusable Symfony2 code
Matthias Noback

A Year With Symfony

Writing healthy, reusable Symfony2 code

Matthias Noback

This book is for sale at http://leanpub.com/a-year-with-symfony

This version was published on 2014-04-16

To Liesbeth and Lucas, because this is... Life

Contents

About this preview

This preview file contains samples that are representative for the book itself. To give you an idea what the entire book looks like, I've added the titles of the parts, chapters and sections to this preview file too.

Foreword

Introduction

Thank you

Who should read this book

Conventions

Overview of the contents

The first part of this book is called The journey from request to response. It will take you along from the first entry point of a Symfony application in the front controller to the last breath it takes before sending a response back to the client. At times I will show you how you can hook into the process and modify the flow or just change the results of the intermediate steps.

The next part is called Patterns of dependency injection. It contains a collection of patterns that are solutions to recurring problems when it comes to creating or modifying service definitions, based on a bundle's configuration. I will show you many very practical examples that you can use to model your own bundle's container extension, configuration class and compiler passes.

The third part is about Project structure. I suggest various ways to get your controllers much cleaner, by delegating actions to form handlers, domain managers and event listeners. We also take a look at state and how to avoid it in the service layer of your application.

A quick intermezzo follows then about Configuration conventions. This part should help you with setting up the configuration for your application. It encourages you and your team to settle on some kind of a configuration convention.

The fifth part is very important as it concerns every serious application, with user sessions and sensitive data. It is about Security. This would seem to be covered completely by all the Symfony components (after all the framework itself has been audited for security issues) and Twig, but unfortunately such a thing would not be possible. You always have to keep thinking about security yourself. This part of the book contains various suggestions on how to deal with security, where to keep an eye on, when you can rely on the framework and when you need to take care of things yourself.

The sixth part is about annotations. When Symfony2 was first released in 2011 it introduced annotations as a revolutionary way to configure an application from within the doc blocks of classes, methods and properties. The first chapter of this part explains how annotations work. After that, you will learn how to create your own annotations and how you can use annotations to influence the response that is generated for a request.

The final part covers all ways of Being a Symfony developer although in fact this part is one big encouragement to *not* be a Symfony developer and to write things as loosely coupled to the Symfony framework as possible. This means separating code into reusable and project-specific code, then splitting the reusable code into library and bundle code. I will discuss different other ideas that make your bundles nice, clean and friendly for other projects.

Enjoy!

I The journey from request to response

1 The `HttpKernelInterface`

Symfony is famous for its `HttpKernelInterface`:

```
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

interface HttpKernelInterface
{
    const MASTER_REQUEST = 1;
    const SUB_REQUEST = 2;

    /**
      * @return Response
      */
    public function handle(
        Request $request,
        $type = self::MASTER_REQUEST,
        $catch = true
    );
}
```

An implementation of this interface would only have to implement one method and thereby declare itself capable of converting in some way a given `Request` into a `Response`. When you take a look at any of the front controllers in the `/web` directory of a Symfony project, you can see that this `handle()` method plays a central role in processing web requests - as you might expect:

```
// in /web/app.php
$kernel = new AppKernel('prod', false);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
```

First, `AppKernel` gets instantiated. This is a class specific to your project, and you can find it in `/app/AppKernel.php`. It allows you to register your bundles, and to change some major settings, like the location of the cache directory or the configuration file that should be loaded. Its constructor arguments are the name of the environment and whether or not the kernel should run in debug mode.

Environment

The environment can be any string. It is mainly a way to determine which configuration file should be loaded (e.g. `config_dev.yml` or `config_prod.yml`). This is made explicit in `AppKernel`:

```
1  public function registerContainerConfiguration(LoaderInterface $loader)
2  {
3      $loader
4          ->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
5  }
```

Debug mode

In debug mode you will have:

- A pretty, verbose exception page, showing all the required information for debugging problems.
- Verbose error messages in case the pretty exception page could not be rendered.
- Elaborate information about the time required to run different parts of the application (bootstrapping, database calls, template rendering, etc.).
- Extensive information about requests (using the web profiler and the accompanying toolbar).
- Automatic cache invalidation: this makes sure that changes to `config.yml`, `routing.yml` and the likes will be taken into account without recompiling the entire service container or routing matcher for each request (which would take a lot of time).

Next a `Request` object is created based on the existing PHP superglobals (`$_GET`, `$_POST`, `$_COOKIE`, `$_FILES` and `$_SERVER`). The `Request` class together with other classes from the `HttpFoundation` component provide object-oriented ways to wrap the superglobals. These classes also cover many corner cases you may experience with different versions of PHP or on different platforms. It is wise (in a Symfony context) to always use `Request` to retrieve any data you would normally have taken directly from the superglobals.

Then the `handle()` method of the `AppKernel` instance gets called. Its only argument is the current `Request` object. The default arguments for the type of the request ("master") and whether or not to catch and handle exceptions (yes) will be added automatically.

The result of this `handle()` method is guaranteed to be an instance of `Response` (also from the `HttpFoundation` component). Finally the response will be sent back to the client that made the request - for instance a browser.

1.1 Booting the kernel

Of course, the magic happens inside the `handle()` method of the kernel. You will find this method implemented in the `Kernel` class, which is the parent class of `AppKernel`:

```
// in Symfony\Component\HttpKernel\Kernel

public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    if (false === $this->booted) {
        $this->boot();
    }

    return $this->getHttpKernel()->handle($request, $type, $catch);
}
```

First of all, it is made sure that the `Kernel` is booted, before the `HttpKernel` is asked to do the rest. The process of booting includes:

- Initializing all the registered bundles
- Initializing the service container

Bundles as container extensions

Bundles are known amongst Symfony developers as the place to put your own code. Each bundle should have a name that reflects what kind of things you could do with the code inside it. For instance you may have a `BlogBundle`, a `CommunityBundle`, a `CommentBundle`, etc. You register your bundles in `AppKernel.php`, by adding them to the existing list of bundles:

```
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            ...,
            new Matthias\BlogBundle()
        );

        return $bundles;
    }
}
```

This is definitely a good idea - it allows you to plug functionality into and out of your project with a single line of code. However, when looking at the `Kernel` and how it deals with all bundles, including yours, it becomes apparent that bundles are mainly treated as ways to extend the service container, not as libraries of code. This is why you find a `DependencyInjection` folder inside many bundles, accompanied by a `{nameOfTheBundle}Extension` class. During the process of initializing the service container, each bundle is allowed to register some services of its own to the service container, maybe add some parameters too, and possibly modify some service definitions before the container gets compiled and dumped to the cache directory:

```
1  namespace Matthias\BlogBundle\DependencyInjection;
2
3  use Symfony\Component\HttpKernel\DependencyInjection\Extension;
4  use Symfony\Component\Config\FileLocator;
5  use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
6
7  class MatthiasBlogExtension extends Extension
8  {
9      public function load(array $configs, ContainerBuilder $container)
10     {
11         $loader = new XmlFileLoader($container,
12             new FileLocator(__DIR__.'/../Resources/config'));
13
14         // add service definitions to the container
15         $loader->load('services.xml');
16
17         $processedConfig = $this->processConfiguration(
18             new Configuration(),
19             $configs
20         );
21
22         // set a parameter
23         $container->setParameter(
24             'matthias_blog.comments_enabled',
25             $processedConfig['enable_comments']
26         );
27     }
28
29     public function getAlias()
30     {
31         return 'matthias_blog';
32     }
33 }
```

The name returned by the `getAlias()` method of a container extension is actually the key under which you can set configuration values (for instance in `config.yml`):

```
matthias_blog:
    enable_comments: true
```

You will read more about bundle configuration in Patterns of dependency injection.

Every configuration key corresponds to a bundle

In the example above you saw that `matthias_blog` is the configuration key for settings related to the `MatthiasBlogBundle`. It may now not be such a big surprise that this is true for all keys you may know from `config.yml` and the likes: values under `framework` are related to the `FrameworkBundle` and values under `security` (even though they are defined in a separate file called `security.yml`) are related to the `SecurityBundle`. Simple as that!

Creating the service container

After all the bundles have been enabled to add their services and parameters, the container is finalized in a process that is called "compilation". During this process it is still possible to make some last-minute changes to service definitions or parameters. It is also the right moment to validate and optimize service definitions. Afterwards, the container is in its final form, and it gets dumped into two different formats: an XML file of all resolved definitions and parameters and a PHP file ready to be used as the one and only service container in your application.

Both files can be found in the cache directory corresponding to the environment of the kernel, for instance `/app/cache/dev/appDevDebugProjectContainer.xml`. The XML file looks like any regular XML service definition file, only a lot bigger:

```
<service id="event_dispatcher" class="...\ContainerAwareEventDispatcher">
  <argument type="service" id="service_container"/>
  <call method="addListenerService">
    <argument>kernel.controller</argument>
    ...
  </call>
  ...
</service>
```

The PHP file contains a method for each service that can be requested. Any creation logic, like controller arguments or method calls after instantiation can be found in this file, and it is therefore the perfect place to debug your service definitions in case anything appears to be wrong with them:

```
class appDevDebugProjectContainer extends Container
{
    ...

    protected function getEventDispatcherService()
    {
        $this->services['event_dispatcher'] =
            $instance = new ContainerAwareEventDispatcher($this);

        $instance->addListenerService('kernel.controller', ...);

        ...

        return $instance;
    }

    ...
}
```

1.2 From the `Kernel` to the `HttpKernel`

Now that the kernel is booted (i.e. all bundles are initialized, their extensions are registered, and the service container is finalized), the real handling of the request is delegated to an instance of `HttpKernel`:

```
// in Symfony\Component\HttpKernel\Kernel

public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    if (false === $this->booted) {
        $this->boot();
    }

    return $this->getHttpKernel()->handle($request, $type, $catch);
}
```

The `HttpKernel` implements `HttpKernelInterface` and it truly knows how to convert a request to a response. The `handle()` method looks like this:

```php
public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    try {
        return $this->handleRaw($request, $type);
    } catch (\Exception $e) {
        if (false === $catch) {
            throw $e;
        }

        return $this->handleException($e, $request, $type);
    }
}
```

As you can see, most of the work is done in the private `handleRaw()` method, and the try/catch block is here to capture any exceptions. When the initial argument `$catch` was `true` (which is the default value for "master" requests), every exception will be handled nicely. The `HttpKernel` will try to find someone who can still create a decent `Response` object for it (see also Exception handling).

2 Events leading to a response

The `handleRaw()` method of the `HttpKernel` is a beautiful piece of code, in which it becomes clear that handling a request is not deterministic *per se*. There are all kinds of ways in which you can hook into the process, and completely replace or just modify any intermediate result.

2.1 Early response

The first moment you can take control of handling the request, is right at the beginning. Usually the `HttpKernel` will try to generate a response by executing a controller. But any event listener that listens to the `KernelEvents::REQUEST` (`kernel.request`) event is allowed to generate a completely custom response:

```
1  use Symfony\Component\HttpKernel\Event\GetResponseEvent;
2
3  private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
4  {
5      $event = new GetResponseEvent($this, $request, $type);
6      $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);
7
8      if ($event->hasResponse()) {
9          return $this->filterResponse(
10             $event->getResponse(),
11             $request,
12             $type
13         );
14     }
15
16     ...
17 }
```

As you can see, the event object that gets created is an instance of `GetResponseEvent` and it allows listeners to set a custom `Response` object using its `setResponse()` method, for example:

```php
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class MaintenanceModeListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $response = new Response(
            'This site is temporarily unavailable',
            503
        );

        $event->setResponse($response);
    }
}
```

Registering event listeners

The event dispatcher used by the `HttpKernel` is the one that is also available as the `event_dispatcher` service. When you want to automatically register some class as an event listener, you can create a service definition for it and add the tag `kernel.event_listener` or `kernel.event_subscriber` (in case you choose to implement `EventSubscriberInterface`).

```
<service id="..." class="...">
  <tag name="kernel.event_listener"
       event="kernel.request"
       method="onKernelRequest" />
</service>
```

Or:

```
<service id="..." class="...">
  <tag name="kernel.event_subscriber" />
</service>
```

You can optionally give your event listener a priority, so that it will take precedence over other listeners:

```
<service id="..." class="...">
  <tag name="kernel.event_listener"
       event="kernel.request"
       method="onKernelRequest"
       priority="100" />
</service>
```

The higher the number, the earlier it will be notified.

Some notable `kernel.request` event listeners

The framework itself has many listeners for the `kernel.request` event. These are mostly listeners to set some things straight, before letting the kernel call any controller. For instance one listener makes sure the application has a locale (either the default locale, or the `_locale` part from the URI), another one processes requests for fragments of pages.

There are however two main players in the early request stage: the `RouterListener` and the `Firewall`. The `RouterListener` takes the path info from the `Request` object and tries to match it to some known route. It stores the result of the matching process in the `Request` object as attributes, for instance the name of the controller that corresponds to the route that was matched:

```
namespace Symfony\Component\HttpKernel\EventListener;

class RouterListener implements EventSubscriberInterface
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();

        $parameters = $this->matcher->match($request->getPathInfo());

        ...

        $request->attributes->add($parameters);
    }
}
```

When for example the matcher is asked to match `/demo/hello/World`, and the routing configuration looks like this:

```
_demo_hello:
    path: /demo/hello/{name}
    defaults:
        _controller: AcmeDemoBundle:Demo:hello
```

The parameters returned by the `match()` call are a combination of the values defined under `defaults:` and the dynamic values matched by placeholders in the path (like `{name}`):

```
array(
  '_route' => '_demo_hello',
  '_controller' => 'AcmeDemoBundle:Demo:hello',
  'name' => 'World'
);
```

These end up in the `Request` parameter bag called "attributes". As you would be able to guess: `HttpKernel` will later examine the request attributes and execute the given controller.

Another important event listener is the `Firewall`. As we saw above, the `RouterListener` does not provide the `HttpKernel` with a `Response` object, it just does some work at the beginning of a request. On the contrary, the `Firewall` sometimes forces a certain `Response` object, for instance when a user is not authenticated when he should have been, since he has requested a protected page. The `Firewall` (through quite a complex process) then forces a redirect to for instance a login page, or sets some headers requiring the user to enter his credentials using HTTP authentication.

2.2 Resolving the controller

2.3 Allow replacement of the controller

Some notable `kernel.controller` listeners

2.4 Collect arguments for executing the controller

2.5 Execute the controller

2.6 Enter the view layer

A notable `kernel.view` listener

2.7 Filter the response

Notable `kernel.response` listeners

3 Exception handling

3.1 Notable `kernel.exception` listeners

4 Sub-requests

4.1 When are sub-requests used?

II Patterns of dependency injection

5 What is a bundle?

As we saw in the previous chapter: running a Symfony application means booting the kernel and handling a request or executing a command, where booting the kernel means: loading all bundles and registering their service container extensions (which can be found in the `DependencyInjection` folder of a bundle). The container extension usually loads a `services.xml` file (but this can be anything) and the bundle's configuration, defined in a separate class, usually in the same namespace, called `Configuration`. These things together (*bundle*, *container extension* and *configuration*) can be used to wire up your bundle: you can define parameters and services so that the functionality you provide inside your bundle is available to other parts of the application. You can even go one step further and register *compiler passes* to further modify the service container before it gets its final form.

After creating many bundles, I concluded that much of my work as a developer which is specific for a Symfony application consists of writing code for exactly these things: bundle, extension and configuration classes and compiler passes. When you know how to write good code, you still need to know how to create good bundles, and this basically means that you need to know how to create good service definitions. There are many ways to do this, and in this chapter I will describe most of them. Knowing your options enables you to make better choices when looking for a dependency injection pattern.

Don't use generation commands

When you start using Symfony, you may be tempted to use commands provided by the `SensioGeneratorBundle` to generate bundles, controllers, entities and form types. I do not recommend this. These generated classes can be nice to keep as a reference for creating these classes manually, but in general they contain too much code you don't need, or that you don't need to begin with. So use these generate commands once, take a look how things should be done, and then learn yourself to do things like that, instead of relying on these commands. It will really make you a faster developer, who understands the framework well.

6 Service patterns

6.1 Required dependencies

Required constructor arguments

Abstract definitions for extra arguments

Required setter calls

Method calls in abstract definitions

6.2 Optional dependencies

Optional constructor arguments

Optional setter calls

6.3 A collection of services

Multiple method calls

The best of both worlds

Service tags

Single method call

Replacing a single argument

Service ids instead of references

6.4 Delegated creation

Not so useful

Sometimes useful

6.5 Manually creating services

Definition

Arguments

Tags

Aliases

6.6 The `Configuration` class

6.7 Dynamically add tags

manager for Doctrine ORM and one for MongoDB. To make the choice for a specific storage manager configurable, create a `Configuration` class like this:

```
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('browser');

        $rootNode
            ->children()
                ->scalarNode('storage_manager')
                    ->validate()
                        ->ifNotInArray(array('doctrine_orm', 'mongo_db')
                        ->thenInvalid('Invalid storage manager')
                    ->end()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Then given there are two service definition files for each of the storage managers, like `doctrine_orm.xml`:

```
<services>
  <service id="mailbox.doctrine_orm.storage_manager" class="...">
  </service>
</services>
```

And `mongo_db.xml`:

```
<services>
  <service id="mailbox.mongo_db.storage_manager" class="...">
  </service>
</services>
```

You could then load either of these files by doing something like this in your container extension:

```php
class MailboxExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );

        // create an XmlLoader
        $loader = ...;

        // load only the services for the given storage manager
        $storageManager = $processedConfig['storage_manager'];
        $loader->load($storageManager.'.xml');

        // make the specific storage manager available as the general one
        $container->setAlias(
            'mailbox.storage_manager',
            'mailbox.'.$storageManager.'.storage_manager'
        );
    }
}
```

A convenient alias is created in the end to allow other parts of the application to just request the `mailbox.storage_manager` service, instead of worrying about the storage-specific service that should be used. However, the way this is done is too rigid: the id of each specific storage manager service should conform to the pattern `mailbox.{storageManagerName}.storage_manager`. It would be better to define the alias inside the service definition files themselves:

```xml
<services>
  <service id="mailbox.doctrine_orm.storage_manager" class="...">
  </service>

  <service id="storage_manager"
    alias="mailbox.doctrine_orm.storage_manager">
  </service>
</services>
```

Using the strategy pattern for loading service definitions has many advantages:

- Only the services that are useful in the current application will be loaded. When you don't have a MongoDB server up and running, there will be no services that accidentally refer to it.

- The setup is open for extension, since you can add the name of another storage manager to the list in the `Configuration` class and then add a service definition file with the necessary services and an alias.

6.9 Loading and configuring additional services

Say you have a bundle dedicated to input filtering. Probably you offer several different services, like services for filtering form data, and services for filtering data stored using Doctrine ORM. It should be possible to enable or disable any of these services or collections of services at any time because they may not all be applicable to your specific situation. There is a handy shortcut for configuration definitions to accomplish a thing like this:

```
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('input_filter');

        $rootNode
            ->children()
                ->arrayNode('form_integration')
                    // will be enabled by default
                    ->canBeDisabled()
                ->end()
                ->arrayNode('doctrine_orm_integration')
                    // will be disabled by default
                    ->canBeEnabled()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

With a configuration tree like this, you can enable or disable specific parts of the bundle in `config.yml`:

```
input_filter:
    form_integration:
        enabled: false
    doctrine_orm_integration:
        enabled: true
```

Inside your container extension you can then load the appropriate services:

```
class InputFilterExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );

        if ($processedConfig['doctrine_orm_integration']['enabled']) {
            $this->loadDoctrineORMIntegration(
                $container,
                $processedConfig['doctrine_orm_integration']
            );
        }

        if ($processedConfig['form_integration']['enabled']) {
            $this->loadFormIntegration(
                $container,
                $processedConfig['form_integration']
            );
        }

        ...
    }

    private function loadDoctrineORMIntegration(
        ContainerBuilder $container,
        array $configuration
    ) {
        // load services, etc.
        ...
    }

    private function loadFormIntegration(
```

```
36          ContainerBuilder $container,
37          array $configuration
38      ) {
39          ...
40      }
41  }
```

Each of the stand-alone parts of the bundle can be loaded separately like this.

A cleaner configuration class

6.10 Configure which services to use

6.11 Completely dynamic service definitions

7 Parameter patterns

7.1 `Parameters.yml`

7.2 Parameter resolving

Parameters for class names

Manually resolving parameters

7.3 Define parameters in a container extension

7.4 Override parameters with a compiler pass

III Project structure

8 Organizing application layers

8.1 Slim controllers

8.2 Form handlers

8.3 Domain managers

8.4 Events

Persistence events

9 State and context

9.1 The security context

9.2 The request

Avoiding a dependency on the current request

Use an event listener

Providing the request object at runtime

Using specific values only

IV Configuration conventions

10 Application configuration setup

10.1 Use local configuration files

Keep `parameters.yml`

Add a `default_parameters.yml`

11 Configuration conventions

11.1 Routing

Choosing Route Names

11.2 Services

11.3 Mapping metadata

V Security

12 Introduction

12.1 Symfony and security

12.2 Goals: prevention and confinement

Minimize impact

Reflection

Before diving in...

13 Authentication and sessions

13.1 Invalidating sessions

Session hijacking

Long-running sessions

14 Controller design

14.1 Secure actions

14.2 Putting controllers behind the firewall

15 Input validation

15.1 Safe forms

HTML5 validation

Validation constraints

Forms without an entity

15.2 Validate values from `Request`

Request attributes

Route parameters

Query or request parameters

Use the ParamFetcher

15.3 Sanitizing HTML

Automatic sanitizing

16 Output escaping

16.1 Twig

Know your escaping context

Escaping function output

Escaping function arguments

Be wary of the `raw` filter

17 Being secretive

17.1 Mask authentication errors

17.2 Prevent exceptions from showing up

17.3 Customize error pages

17.4 Be vague about user data

VI Using annotations

18 Introduction

19 An annotation is a simple value object

19.1 Adding attributes to your annotation

Passing the attributes via the constructor

Populating public properties with the provided attributes

Validation using `@Attributes`

Validation using `@var` and `@Required`

19.2 Limiting the use of an annotation

20 Valid use cases for annotations

20.1 Loading configuration

20.2 Controlling application flow

21 Using annotations in your Symfony application

21.1 Responding to `Request` attributes: the `@Referrer` annotation

Browsers have the useful habit to add a `Referer` header to a request (unless it is a direct request, like when you type a URL by hand or select one from your bookmarks). The name of the header actually contains a spelling error (it should have one extra "r": `Referrer`), but there's nothing we can do about that, except for making no spelling mistakes ourselves.

The `Referer` header contains the full URL of the page that was visited by the client before they requested the current URL. While handling a request you can use it to redirect a client back to where they came from. Or you could store the previous URL to later analyze which sites refer to your site.

In this particular example I'd like to use the `Referer` header to apply certain rules. For instance, some actions inside my application's controllers should only be executed when the user comes from a certain other page, and some of them are only available when the previous URL has the same domain as the current URL. Though this would not be a good security measure, it can be something that's nice to have. More importantly it is a good example of how you can influence the application flow based on an attribute of the current `Request` object.

Be aware of security issues

Whenever you make your application logic depend on request attributes, think about any security issues that may arise from that. Some particular issues with request attributes are described in the chapter about [Security]{#request-attributes}.

Based on the scenario I described above, I should be able to write something like this above my actions:

```
class SomeController
{
    /**
     * @Referrer(pattern="^/demo", sameDomain=true)
     */
    public function specialAction()
    {
        ...
    }
}
```

This would trigger some kind of validation mechanism which takes the `Referer` header and checks if its path matches the given pattern and if the referring URL has the same domain as the current URL.

The corresponding annotation class could look something like this:

```
1  namespace Matthias\ReferrerBundle\Annotation;
2
3  /**
4   * @Annotation
5   * @Attributes({
6   *   @Attribute("pattern", type="string"),
7   *   @Attribute("sameDomain", type="boolean")
8   * })
9   */
10 class Referrer
11 {
12     public $pattern = '.*';
13     public $sameDomain = false;
14 }
```

None of the attributes are marked as required. Instead I chose to add default values for the class properties `$pattern` and `$sameDomain`.

We want the referrer validation process to be triggered by an annotation that belongs to an action method of a controller class. What would be a good moment to do so? Well, as you may remember from the first part of this book: after the controller has been fully determined and resolved, the kernel dispatches a `kernel.controller` event. Event listeners are then allowed to do anything that's needed, given the current controller. This seems like a perfect fit. So in this case we should create an event listener that listens to this particular `kernel.controller` event. When the listener is notified of such an event, it can inspect the current controller and see if it has a `@Referrer` annotation.

The `ReferrerListener` below will do the trick. We'll make sure that its `onKernelController()` method will be triggered when a `kernel.controller` event is dispatched by the kernel. I assume that you know how to register this event listener.

```
1  namespace Matthias\ReferrerBundle\EventListener;
2
3  use Doctrine\Common\Annotations\Reader;
4  use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
5  use Matthias\ReferrerBundle\Annotation\Referrer;
6  use Symfony\Component\HttpFoundation\Request;
7
8  class ReferrerListener
9  {
10     private $annotationReader;
```

```
11
12      public function __construct(Reader $annotationReader)
13      {
14          $this->annotationReader = $annotationReader;
15      }
16
17      public function onKernelController(FilterControllerEvent $event)
18      {
19          // the current controller callable
20          $controller = $event->getController();
21
22          if (!is_array($controller)) {
23              // we only know how to handle a callable that looks like
24              // array($controllerObject, 'nameOfActionMethod')
25              return;
26          }
27
28          // the annotation reader needs a reflection object like this
29          $action = new \ReflectionMethod($controller[0], $controller[1]);
30
31          $referrerAnnotation = $this
32              ->annotationReader
33              ->getMethodAnnotation(
34                  $action,
35                  'Matthias\ReferrerBundle\Annotation\Referrer'
36              );
37
38          // $referrerAnnotation is either an instance of Referrer or null
39          if (!($referrerAnnotation instanceof Referrer)) {
40              return;
41          }
42
43          $this->validateReferrer($event->getRequest(), $referrerAnnotation);
44      }
45
46      private function validateReferrer(
47          Request $request,
48          Referrer $configuration
49      ) {
50          $actualReferrer = $request->headers->get('referer');
51
52          $pattern = $configuration->pattern;
```

```
53          $sameDomain = $configuration->sameDomain;
54
55          // do anything you like
56          // maybe throw an exception if you don't like it
57      }
58  }
```

Make sure that the service you create for this event listener will receive the `annotation_reader` service as its first constructor argument.

21.2 Prevent controller execution: the `@RequiresCredits` annotation

21.3 Modify the response: the `@DownloadAs` annotation

22 Designing for reusability

23 Conclusion

VII Being a Symfony developer

Many of the developers that I know call themselves “PHP developer”. Some of them might even say that they are a “Symfony developer”. Others will say, “I am a developer”, without revealing their favorite programming language or the only programming language they master.

When I started to work with Symfony, I almost immediately fell somewhat in love with this great framework. Its first version was already so much better than what I was used to. But the second version was just the right thing for my mind - I started learning a lot about it, digging deep into its code, writing documentation for parts that were still undocumented, writing articles about how to accomplish certain things with Symfony, and speaking in public about Symfony and Symfony-related PHP libraries.

After all of this, I would call myself a Symfony developer now. But the best part of the story is: everything that I learned from and while working with Symfony, is generally applicable to software written “for” any other PHP framework. Even when I work with a non-Symfony or a “legacy” PHP application (with no reused code at all), it still pays off to think about ways in which I can use code from the “Symfony ecosystem”, or in fact, from any library that I can pull in using Composer, and make it a better application.

In this part I will demonstrate that being a good Symfony developer is about knowing the framework well, but then writing code that would be beneficent for any PHP project out there and finishing with a tiny layer between this code and the Symfony framework so that most of your code will be reusable even if it is being transferred to a non-Symfony project.

24 Reusable code has low coupling

24.1 Separate company and product code

Your situation as a Symfony developer is most likely:

- You are working for a company.
- You have customers (internal or external) for whom you create web applications.

When you start working on a new application, you will put your project's code in the `/src` directory. But before you start, add two new directories inside this `/src` directory: `/src/NameOfYourCompany` and `/src/NameOfTheProduct`. Of course, these directory names should also be reflected in the namespaces of the classes you create for your project.

Whenever you start working on a new feature for the application - think of which part you could *in theory* reuse, and which part is unique for the application you are working on. Even when this reusable code will in practice *never* be reused, its quality will benefit from this mindset. Start developing classes in the company namespace. Only when you really feel the need to use something that is specific for the project, move it over to the product namespace, or use some kind of extension principle (a subclass, configuration, event listeners, etc.).

Writing reusable code for your company does not mean that it should be open sourced. It just means that you should write it *as if* it will be open sourced. You don't have to create side-projects for all the company code right away. You can develop it inside the project you're currently working on and maybe later make it ready for reuse in another project. Read more about the practical side of this in Dependency management and version control

Coupling between company and product code

When you follow a strict separation between company and product code, this is a set of rules that help you make the separation really useful (when you don't follow these rules, there is no point in keeping separate namespaces actually):

1. *Company code* may know about or depend upon other *company code*.
2. *Company code* may *not* know about or depend upon *product-specific code*.
3. *Product-specific code* may know or depend upon other *product-specific* code.

The first two rules should be taken strictly into account: only when you do so, the code will ever be reusable or ready to be open sourced. The third rule on the contrary *adds* a lot of freedom: since you will by definition not reuse any product-specific code you are allowed to make it highly coupled to other product-specific code.

24.2 Separate library and bundle code

When you notice that you are writing classes and interfaces that have no relation with the entire Symfony framework, or only with some parts of the framework, you should separate your code into "library" code and "bundle" code. The library code is the part of the code that is more or less stand-alone (although it could have some external dependencies). Library code could be reused by a developer who works with the Zend framework, or with the Silex micro-framework (to name just a few). Bundle code replaces or extends some classes from the library code, adding extra Symfony-specific features. It defines a bundle configuration, and it has service definitions, to make instances of library classes available in the application. A bundle in this way just makes the library code available, while requiring just a minimum effort from developers who want to use the library in their Symfony project.

These are the things that belong inside a bundle:

- Controllers
- Container extensions
- Service definitions
- Compiler passes
- Event subscribers
- Container-aware classes that extend more generic classes
- Form types
- Routing configuration
- Other metadata
- …

The list could be much longer. But it could also be much shorter. When you think about it, only the first couple of things on the list are really specific for bundles (i.e. used only in the context of a standard Symfony project). All of the other things could also be used in projects that only make use of specific Symfony Components. For instance form types could also be used in any PHP project with only the Form Component installed.

Examples of library and bundle code

There are many good examples of this separation of bundle and library code. When you take a look at the code of Symfony itself, you can see this clearly: the directory `Component` contains all the Symfony components (the "library" code), and the directory `Bundle` contains the bundles which tie all the classes together, and provide configuration options which are passed as constructor arguments to all these classes. Many great examples of this strategy can be found in the `FrameworkBundle` (the "core" bundle: when it's present in a project, we call it a "Symfony project").

24.3 Reduce coupling to the framework

You can go quite far into reducing coupling to the Symfony framework or to one of its components, and thereby making your code even more reusable.

Event listeners over event subscribers

For example you should prefer event listeners over event subscribers. Event subscribers are special event listeners, that implement `EventSubscriberInterface`:

```
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class SendConfirmationMailListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            AccountEvents::NEW_ACCOUNT_CREATED => 'onNewAccount'
        );
    }

    public function onNewAccount(AccountEvent $event)
    {
        ...
    }
}
```

You can register event subscribers like this in a very clean way using the service tag `kernel.event_subscriber`:

```
<service id="send_confirmation_mail_listener"
  class="SendConfirmationMailListener">
  <tag name="kernel.event_subscriber" />
</service>
```

There are some problems with this approach:

1. An event subscriber becomes totally useless when the Symfony EventDispatcher Component is not available, even though there is nothing Symfony-specific about this event listener.
2. `onNewAccount()` receives an `AccountEvent` object, but it is nowhere determined that such an object can only arise from an event with the name `AccountEvents::NEW_ACCOUNT_CREATED`.

Therefore an event listener which is actually an event *subscriber* is not good for reusable code. It couples the code to the EventDispatcher Component. It is better to remove the interface, remove the method required by the interface and register the listener methods manually using the service tag `kernel.event_listener`:

```
<service id="send_confirmation_mail_listener"
  class="SendConfirmationMailListener">
  <tag name="kernel.event_listener"
    event="new_account_created" method="onNewAccount" />
</service>
```

Constructor arguments over fetching container parameters

Constructor arguments over fetching container services

The performance issue

Framework-agnostic controllers

Thin commands

The environment

25 Reusable code should be mobile

25.1 Dependency management and version control

Package repositories

25.2 Hard-coded storage layer

Auto-mapped entities

Storage-agnostic models

Object managers

25.3 Hard-coded filesystem references

Using the filesystem

26 Reusable code should be open for extension

26.1 Configurable behavior

26.2 Everything should be replaceable

Use lots of interfaces

Use the bundle configuration to replace services

26.3 Add extension points

Service tags

Events

27 Reusable code should be easy to use

27.1 Add documentation

27.2 Throw helpful exceptions

Use specific exception classes

Set detailed and friendly messages

28 Reusable code should be reliable

28.1 Add enough tests

Test your bundle extension and configuration

29 Conclusion

End of file

You have reached EOF. I hope you liked this sample. If you did, then I'm sure the book won't disappoint you. You can buy it on Leanpub[1].

[1] https://leanpub.com/a-year-with-symfony/