

Step 1

Checking your Work Environment

Before starting to work on the project, we need to check that you have a good working environment. It is very important. The developers tools we have at our disposal today are very different from the ones we had 10 years ago. They have evolved a lot, for the better. It would be a shame to not leverage them. Good tools can get you a long way.

Please, don't skip this step. Or at least, read the last section about the Symfony CLI.

1.1 A Computer

You need a computer. The good news is that it can run on any popular OS: macOS, Windows, or Linux. Symfony and all the tools we are going to use are compatible with each of these.

1.2 Opinionated Choices

I want to move fast with the best options out there. I made opinionated choices for this book.

PostgreSQL is going to be our choice for everything: from database to queues, from cache to session storage. For most projects, *PostgreSQL* is the best solution, scale well, and allows to simplify the infrastructure with only one service to manage.

At the end of the book, we will learn how to use *RabbitMQ* for queues and *Redis* for sessions.

1.3 IDE

You can use Notepad if you want to. I would not recommend it though.

I used to work with Textmate. Not anymore. The comfort of using a “real” IDE is priceless. Auto-completion, use statements added and sorted automatically, jumping from one file to another are a few features that will boost your productivity.

I would recommend using *Visual Studio Code* or *PhpStorm*. The former is free, the latter is not but has a better integration with Symfony (thanks to the *Symfony Support Plugin*). It is up to you. I know you want to know which IDE I am using. I am writing this book in Visual Studio Code.

1.4 Terminal

We will switch from the IDE to the command line all the time. You can use your IDE’s built-in terminal, but I prefer to use a real one to have more space.

Linux comes built-in with *Terminal*. Use *iTerm2* on macOS. On Windows, *Hyper* works well.

1.5 Git

For version control, we will use *Git* as everybody is using it now.

On Windows, install *Git bash*.

Be sure you know how to do the common operations like running `git clone`, `git log`, `git show`, `git diff`, `git checkout`, ...

1.6 PHP

We will use Docker for services, but I like to have PHP installed on my local computer for performance, stability, and simplicity reasons. Call me old school if you like, but the combination of a local PHP and Docker services is the perfect combo for me.

Use PHP 8.1 and check that the following PHP extensions are installed or install them now: `intl`, `pdo_pgsql`, `xsl`, `amqp`, `gd`, `openssl`, `sodium`. Optionally install `redis`, `curl`, and `zip` as well.

You can check the extensions currently enabled via `php -m`.

We also need `php-fpm` if your platform supports it, `php-cgi` works as well.

1.7 Composer

Managing dependencies is everything nowadays with a Symfony project. Get the latest version of *Composer*, the package management tool for PHP.

If you are not familiar with Composer, take some time to read about it.



You don't need to type the full command names: `composer req` does the same as `composer require`, use `composer rem` instead of `composer remove`, ...

1.8 NodeJS

We won't write much JavaScript code, but we will use JavaScript/NodeJS tools to manage our assets. Check that you have the *NodeJS* installed.

1.9 Docker and Docker Compose

Services are going to be managed by Docker and Docker Compose. *Install them* and start Docker. If you are a first time user, get familiar with the tool. Don't panic though, our usage will be very straightforward. No fancy configurations, no complex setup.

1.10 Symfony CLI

Last, but not least, we will use the *symfony* CLI to boost our productivity. From the local web server it provides, to full Docker integration and cloud support through Platform.sh, it will be a great time saver.

Install the *Symfony CLI* now.

To use HTTPS locally, we also need to *install a certificate authority (CA)* to enable TLS support. Run the following command:

```
$ symfony server:ca:install
```

Check that your computer has all needed requirements by running the following command:

```
$ symfony book:check-requirements
```

If you want to get fancy, you can also run the *Symfony proxy*. It is optional but it allows you to get a local domain name ending with *.wip* for your project.

When executing a command in a terminal, we will almost always prefix it with *symfony* like in *symfony composer* instead of just *composer*, or *symfony console* instead of *./bin/console*.

The main reason is that the Symfony CLI automatically sets some environment variables based on the services running on your machine via Docker. These environment variables are available for HTTP requests because the local web server injects them automatically. So, using `symfony` on the CLI ensures that you have the same behavior across the board.

Moreover, the Symfony CLI automatically selects the “best” possible PHP version for the project.

Step 2

Introducing the Project

We need to find a project to work on. It is quite a challenge as we need to find a project large enough to cover Symfony thoroughly, but at the same time, it should be small enough; I don't want you to get bored implementing similar features more than once.

2.1 Revealing the Project

It might be nice if the project was somehow related to Symfony and its community. As we organize quite a few online and in-person conferences every year, what about a *guestbook*? A livre d'or as we say in French. I like the old-fashioned and outdated feeling of developing a guestbook in the 21st century!

We have it. The project is all about getting feedback on conferences: a list of conferences on the homepage, a page for each conference, full of nice comments. A comment is composed of some small text and an optional photo taken during the conference. I suppose I have just written down all the specifications we need to get started.

The *project* will contain several *applications*. A traditional web application with an HTML frontend, an API, and an SPA for mobile phones. How does that sound?

2.2 Learning is Doing

Learning is doing. Period. Reading a book about Symfony is nice. Coding an application on your personal computer while reading a book about Symfony is even better. This book is very special as everything has been done to let you follow along, code, and be sure to get the same results as I had locally on my machine when I coded it initially.

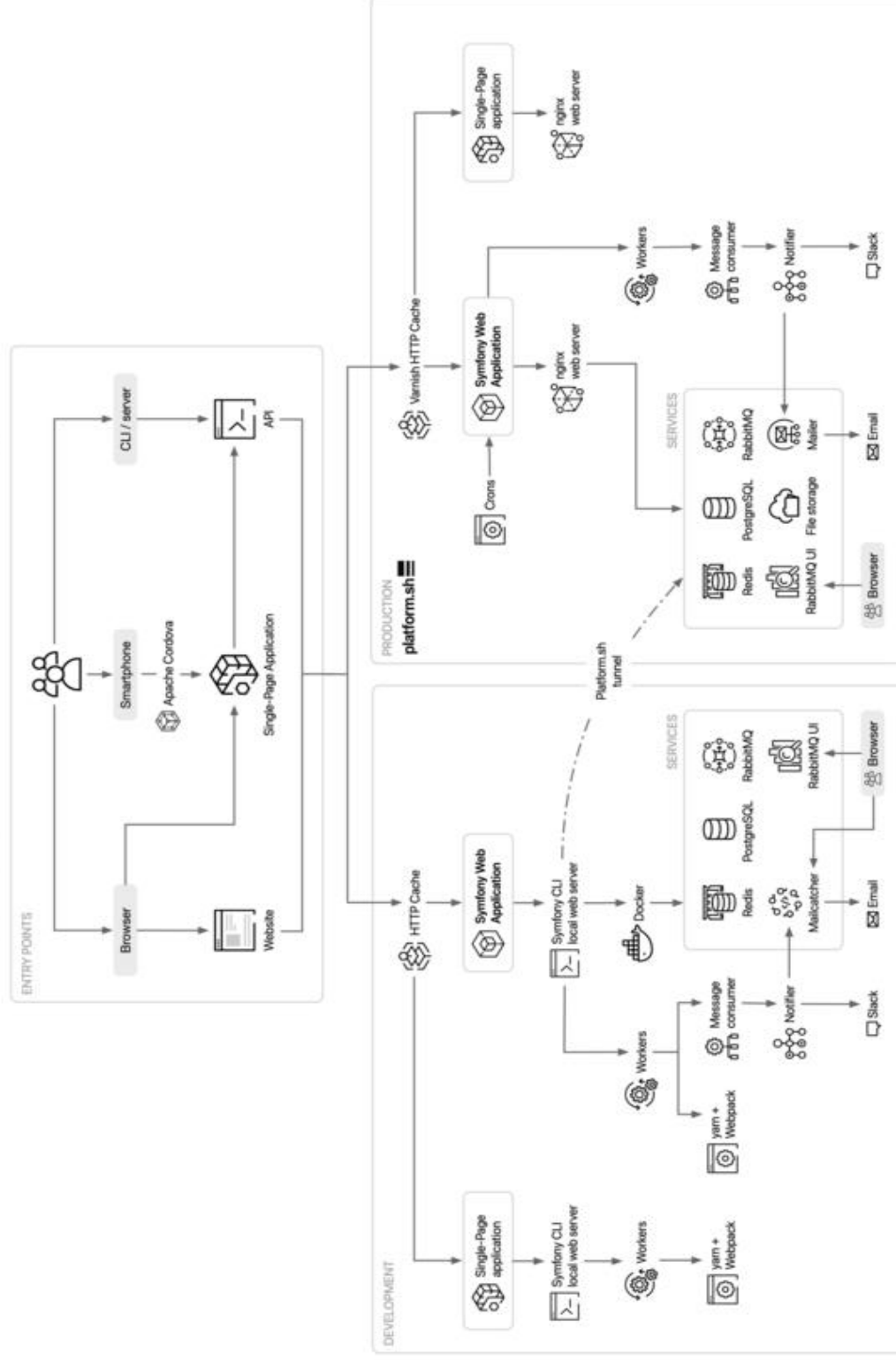
The book contains all the code you need to write and all the commands you need to execute to get the final result. No code is missing. All commands are written down. This is possible because modern Symfony applications have very little boilerplate code. Most of the code we will write together is about the project's *business logic*. Everything else is mostly automated or generated automatically for us.

2.3 Looking at the Final Infrastructure Diagram

Even if the project idea seems simple, we are not going to build an “Hello World”-like project. We won't only use PHP and a database.

The goal is to create a project with some of the complexities you might find in real-life. Want a proof? Have a look at the final infrastructure of the project:

Symphony 5, The Fast Track - Software Architecture Diagram



One of the great benefit of using a framework is the small amount of code needed to develop such a project:

- 20 PHP classes under `src/` for the website;
- 550 PHP Logical Lines of Code (LLOC) as reported by *PHPLOC*;
- 40 lines of configuration tweaks in 3 files (via attributes and YAML), mainly to configure the backend design;
- 20 lines of development infrastructure configuration (Docker);
- 100 lines of production infrastructure configuration (Platform.sh);
- 5 explicit environment variables.

Ready for the challenge?

2.4 Getting the Project Source Code

To continue on the old-fashioned theme, I could have created a CD containing the source code, right? But what about a Git repository companion instead?

Clone the *guestbook repository* somewhere on your local machine:

```
$ symfony new --version=6.2-1 --book guestbook
```

This repository contains all the code of the book.

Note that we are using `symfony new` instead of `git clone` as the command does more than just cloning the repository (hosted on Github under the `the-fast-track` organization: <https://github.com/the-fast-track/book-6.2-1>). It also starts the web server, the containers, migrates the database, loads fixtures, ... After running the command, the website should be up and running, ready to be used.

The code is 100% guaranteed to be synchronized with the code in the book (use the exact repository URL listed above). Trying to manually synchronize changes from the book with the source code in the repository is almost impossible. I tried in the past. I failed. It is just impossible. Especially for books like the ones I write: books that tells you a story

about developing a website. As each chapter depends on the previous ones, a change might have consequences in all following chapters.

The good news is that the Git repository for this book is *automatically generated* from the book content. You read that right. I like to automate everything, so there is a script whose job is to read the book and create the Git repository. There is a nice side-effect: when updating the book, the script will fail if the changes are inconsistent or if I forget to update some instructions. That's BDD, Book Driven Development!

2.5 Navigating the Source Code

Even better, the repository is not just about the final version of the code on the `main` branch. The script executes each action explained in the book and it commits its work at the end of each section. It also tags each step and substep to ease browsing the code. Nice, isn't it?

If you are lazy, you can get the state of the code at the end of a step by checking out the right tag. For instance, if you'd like to read and test the code at the end of step 10, execute the following:

```
$ symfony book:checkout 10
```

Like for cloning the repository, we are not using `git checkout` but `symfony book:checkout`. The command ensures that whatever the state you are currently in, you end up with a functional website for the step you ask for. **Be warned that all data, code, and containers are removed by this operation.**

You can also check out any substep:

```
$ symfony book:checkout 10.2
```

Again, I highly recommend you code yourself. But if you get stuck, you can always compare what you have with the content of the book.

Not sure that you got everything right in substep 10.2? Get the diff:

```
$ git diff step-10-1...step-10-2
```

```
# And for the very first substep of a step:  
$ git diff step-9...step-10-1
```

Want to know when a file has been created or modified?

```
$ git log -- src/Controller/ConferenceController.php
```

You can also browse diffs, tags, and commits directly on GitHub. This is a great way to copy/paste code if you are reading a paper book!

Step 3

Going from Zero to Production

I like to go fast. I want our little project to be live as fast as possible. Like now. In production. As we haven't developed anything yet, we will start by deploying a nice and simple "Under construction" page. You will love it!

Spend some time trying to find the ideal, old fashioned, and animated "Under construction" GIF on the Internet. Here is *the one* I'm going to use:



I told you, it is going to be a lot of fun.

3.1 Initializing the Project

Create a new Symfony project with the `symfony` CLI tool we have previously installed together:

```
$ symfony new guestbook --version=6.2 --php=8.1 --webapp --docker --cloud
$ cd guestbook
```

This command is a thin wrapper on top of `Composer` that eases the creation of Symfony projects. It uses a *project skeleton* that includes the bare minimum dependencies; the Symfony components that are needed for almost any project: a console tool and the HTTP abstraction needed to create Web applications.

As we are creating a fully-featured web application, we have added a few options that will make our life easier:

- `--webapp`: By default, an application with the fewest possible dependencies is created. For most web projects, it is recommended to use the `webapp` package on top. It contains most of the packages needed for “modern” web applications. The `webapp` package adds a lot of Symfony packages, including Symfony Messenger and PostgreSQL via Doctrine.
- `--docker`: On your local machine, we will use Docker to manage services like PostgreSQL. This option enables Docker so that Symfony will automatically add Docker services based on the required packages (a PostgreSQL service when adding the ORM or a mail catcher when adding Symfony Mailer for instance).
- `--cloud`: If you want to deploy your project on Platform.sh, this option automatically generates a sensible Platform.sh configuration. Platform.sh is the preferred and simplest way to deploy testing, staging, and production Symfony environments in the cloud.

If you have a look at the GitHub repository for the skeleton, you will notice that it is almost empty. Just a `composer.json` file. But the `guestbook` directory is full of files. How is that even possible? The answer lies in the `symfony/flex` package. Symfony Flex is a Composer plugin that hooks into the installation process. When it detects a package for which it has a

recipe, it executes it.

The main entry point of a Symfony Recipe is a manifest file that describes the operations that need to be done to automatically register the package in a Symfony application. You never have to read a README file to install a package with Symfony. Automation is a key feature of Symfony.

As Git is installed on our machine, `symfony new` also created a Git repository for us and it added the very first commit.

Have a look at the directory structure:

```
|— bin/  
|— composer.json  
|— composer.lock  
|— config/  
|— public/  
|— src/  
|— symfony.lock  
|— var/  
|— vendor/
```

The `bin/` directory contains the main CLI entry point: `console`. You will use it all the time.

The `config/` directory is made of a set of default and sensible configuration files. One file per package. You will barely change them, trusting the defaults is almost always a good idea.

The `public/` directory is the web root directory, and the `index.php` script is the main entry point for all dynamic HTTP resources.

The `src/` directory hosts all the code you will write; that's where you will spend most of your time. By default, all classes under this directory use the `App` PHP namespace. It is your home. Your code. Your domain logic. Symfony has very little to say there.

The `var/` directory contains caches, logs, and files generated at runtime by the application. You can leave it alone. It is the only directory that needs to be writable in production.

The `vendor/` directory contains all packages installed by Composer, including Symfony itself. That's our secret weapon to be more productive. Let's not reinvent the wheel. You will rely on existing libraries to do the hard work. The directory is managed by Composer. Never

touch it.

That's all you need to know for now.

3.2 Creating some Public Resources

Anything under `public/` is accessible via a browser. For instance, if you move your animated GIF file (name it `under-construction.gif`) into a new `public/images/` directory, it will be available at a URL like `https://localhost/images/under-construction.gif`.

Download my GIF image here:

```
$ mkdir public/images/  
$ php -r "copy('http://clipartmag.com/images/website-under-construction-image-6.gif', 'public/images/under-construction.gif');"
```

3.3 Launching a Local Web Server

The `symfony` CLI comes with a Web Server that is optimized for development work. You won't be surprised if I tell you that it works nicely with `Symfony`. Never use it in production though.

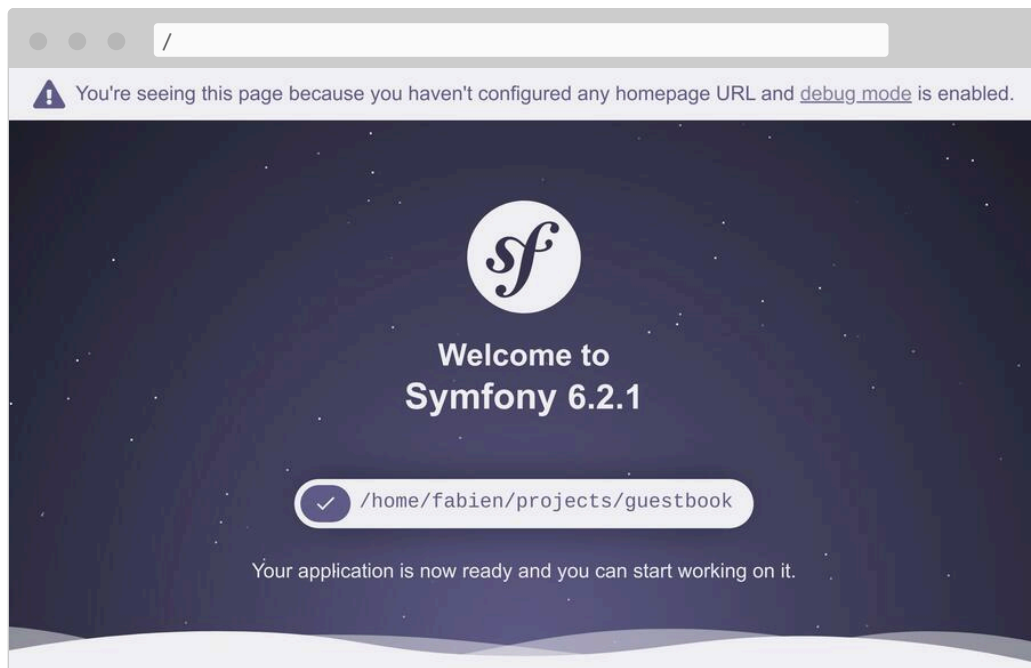
From the project directory, start the web server in the background (`-d` flag):

```
$ symfony server:start -d
```

The server started on the first available port, starting with 8000. As a shortcut, open the website in a browser from the CLI:

```
$ symfony open:local
```

Your favorite browser should take the focus and open a new tab that displays something similar to the following:



To troubleshoot problems, run `symfony server:log`; it tails the logs from the web server, PHP, and your application.

Browse to `/images/under-construction.gif`. Does it look like this?



Satisfied? Let's commit our work:

```
$ git add public/images
$ git commit -m'Add the under construction image'
```

3.4 Preparing for Production

What about deploying our work to production? I know, we don't even have a proper HTML page yet to welcome our users. But being able to see the little “under construction” image on a production server would be a

great step forward. And you know the motto: *deploy early and often*.

You can host this application on any provider supporting PHP... which means almost all hosting providers out there. Check a few things though: we want the latest PHP version and the possibility to host services like a database, a queue, and some more.

I have made my choice, it's going to be *Platform.sh*. It provides everything we need and it helps fund the development of Symfony.

As we used the `--cloud` option when we created the project, Platform.sh has already been initialized with a few files needed by Platform.sh, namely `.platform/services.yaml`, `.platform/routes.yaml`, and `.platform.app.yaml`.

3.5 Going to Production

Deploy time?

Create a new remote Platform.sh project:

```
$ symfony cloud:project:create --title="Guestbook" --plan=development
```

This command does a lot:

- The first time you launch this command, authenticate with your Platform.sh credentials if not done already.
- It provisions a new project on Platform.sh (you get 30 days *for free* on the first project you create).

Then, deploy:

```
$ symfony cloud:deploy
```

The code is deployed by pushing the Git repository. At the end of the command, the project will have a specific domain name you can use to access it.

Check that everything worked fine:

```
$ symfony cloud:url -1
```

You should get a 404, but browsing to `/images/under-construction.gif` should reveal our work.

Note that you don't get the beautiful default Symfony page on Platform.sh. Why? You will learn soon that Symfony supports several environments and Platform.sh automatically deployed the code in the production environment.



If you want to delete the project on Platform.sh, use the `cloud:project:delete` command.



Going Further

- The repositories for the *official Symfony recipes* and for the *recipes contributed by the community*, where you can submit your own recipes;
- The *Symfony Local Web Server*;
- The *Platform.sh documentation*.

Step 4

Adopting a Methodology

Teaching is about repeating the same thing again and again. I won't do that. I promise. At the end of each step, you should do a little dance and save your work. It is like `Ctrl+S` but for a website.

4.1 Implementing a Git Strategy

At the end of each step, don't forget to commit your changes:

```
$ git add .  
$ git commit -m 'Add some new feature'
```

You can safely add “everything” as Symfony manages a `.gitignore` file for you. And each package can add more configuration. Have a look at the current content:

```
.gitignore  
###> symfony/framework-bundle ###  
/.env.local  
/.env.local.php
```

```
/.env.*.local  
/config/secrets/prod/prod.decrypt.private.php  
/public/bundles/  
/var/  
/vendor/  
###< symfony/framework-bundle ###
```

The funny strings are markers added by Symfony Flex so that it knows what to remove if you decide to uninstall a dependency. I told you, all the tedious work is done by Symfony, not you.

It could be nice to push your repository to a server somewhere. GitHub, GitLab, or Bitbucket are good choices.



If you are deploying on Platform.sh, you already have a copy of the Git repository as Platform.sh uses Git behind the scenes when you are using `cloud:deploy`. But you should not rely on the Platform.sh Git repository. It is only for deployment usage. It is not a backup.

4.2 Deploying to Production Continuously

Another good habit is to deploy frequently. Deploying at the end of each step is a good pace:

```
$ symfony cloud:deploy
```

Step 5

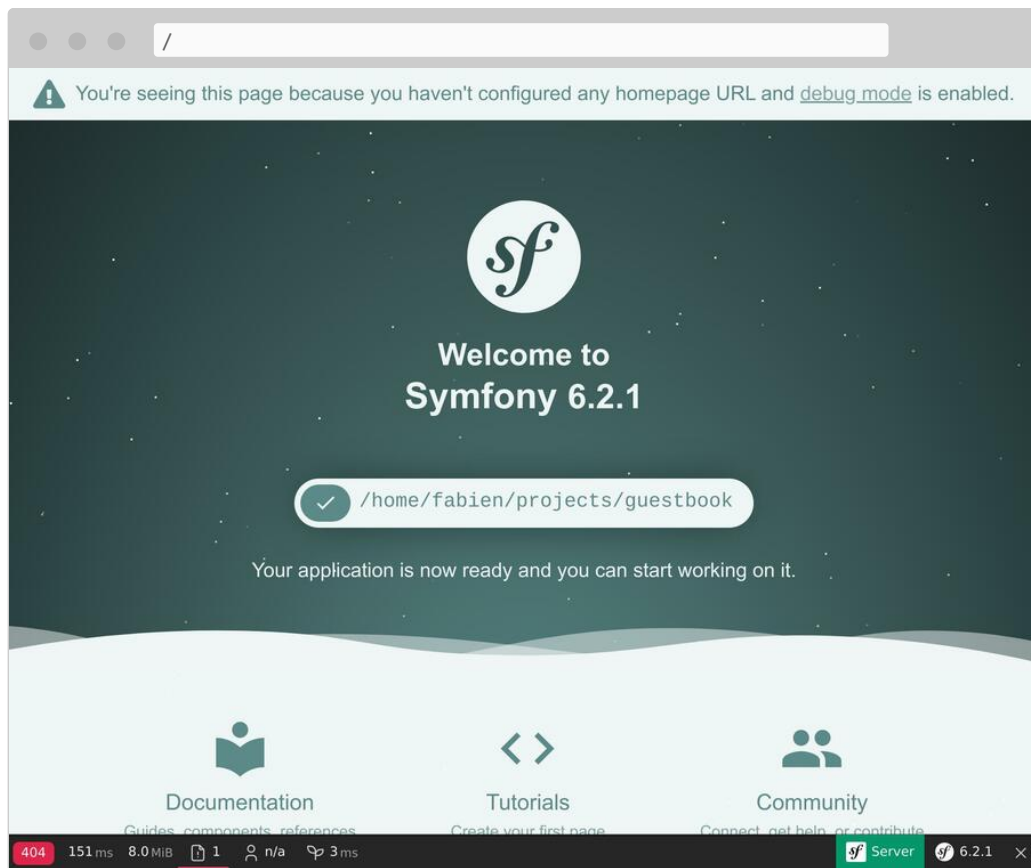
Troubleshooting Problems

Setting up a project is also about having the right tools to debug problems. Fortunately, many nice helpers are already included as part of the `webapp` package.

5.1 Discovering the Symfony Debugging Tools

To begin with, the Symfony Profiler is a time saver when you need to find the root cause of a problem.

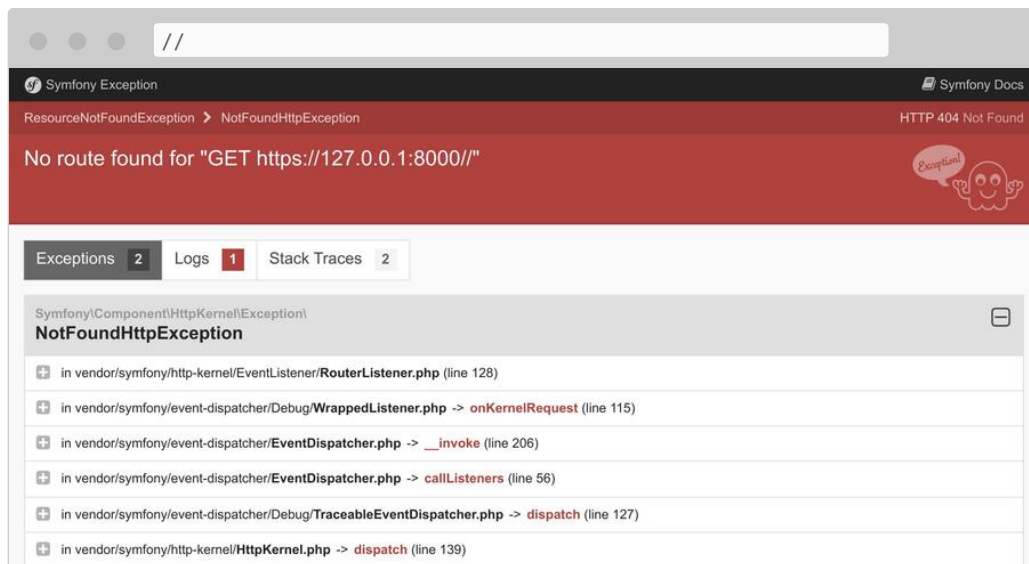
If you have a look at the homepage, you should see a toolbar at the bottom of the screen:



The first thing you might notice is the **404** in red. Remember that this page is a placeholder as we have not defined a homepage yet. Even if the default page that welcomes you is beautiful, it is still an error page. So the correct HTTP status code is 404, not 200. Thanks to the web debug toolbar, you have the information right away.

If you click on the small exclamation point, you get the “real” exception message as part of the logs in the Symfony profiler. If you want to see the stack trace, click on the “Exception” link on the left menu.

Whenever there is an issue with your code, you will see an exception page like the following that gives you everything you need to understand the issue and where it comes from:



Take some time to explore the information inside the Symfony profiler by clicking around.

Logs are also quite useful in debugging sessions. Symfony has a convenient command to tail all the logs (from the web server, PHP, and your application):

```
$ symfony server:log
```

Let's do a small experiment. Open `public/index.php` and break the PHP code there (add foobar in the middle of the code for instance). Refresh the page in the browser and observe the log stream:

```
Dec 21 10:04:59 |DEBUG| PHP    PHP Parse error:  syntax error, unexpected 'use'
(T_USE) in public/index.php on line 5 path="/usr/bin/php7.42" php="7.42.0"
Dec 21 10:04:59 |ERROR| SERVER GET  (500) / ip="127.0.0.1"
```

The output is beautifully colored to get your attention on errors.

5.2 Understanding Symfony Environments

As the Symfony Profiler is only useful during development, we want to avoid it being installed in production. By default, Symfony automatically installed it only for the `dev` and `test` environments.

Symfony supports the notion of *environments*. By default, it has built-in

support for three, but you can add as many as you like: `dev`, `prod`, and `test`. All environments share the same code, but they represent different *configurations*.

For instance, all debugging tools are enabled in the `dev` environment. In the `prod` one, the application is optimized for performance.

Switching from one environment to another can be done by changing the `APP_ENV` environment variable.

When you deployed to Platform.sh, the environment (stored in `APP_ENV`) was automatically switched to `prod`.

5.3 Managing Environment Configurations

`APP_ENV` can be set by using “real” environment variables in your terminal:

```
$ export APP_ENV=dev
```

Using real environment variables is the preferred way to set values like `APP_ENV` on production servers. But on development machines, having to define many environment variables can be cumbersome. Instead, define them in a `.env` file.

A sensible `.env` file was generated automatically for you when the project was created:

```
.env
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=c2927f273163f7225a358e3a1bbbed8a
#TRUSTED_PROXIES=127.0.0.1,127.0.0.2
#TRUSTED_HOSTS='^localhost|example\.com$'
###< symfony/framework-bundle ###
```



Any package can add more environment variables to this file thanks to their recipe used by Symfony Flex.

The `.env` file is committed to the repository and describes the *default* values from production. You can override these values by creating a

`.env.local` file. This file should not be committed and that's why the `.gitignore` file is already ignoring it.

Never store secret or sensitive values in these files. We will see how to manage secrets in another step.

5.4 Configuring your IDE

In the development environment, when an exception is thrown, Symfony displays a page with the exception message and its stack trace. When displaying a file path, it adds a link that opens the file at the right line in your favorite IDE. To benefit from this feature, you need to configure your IDE. Symfony supports many IDEs out of the box; I'm using Visual Studio Code for this project:

```
--- a/php.ini
+++ b/php.ini
@@ -6,3 +6,4 @@ max_execution_time=30
 session.use_strict_mode=0n
 realpath_cache_ttl=3600
 zend.detect_unicode=Off
+xddebug.file_link_format=vscode://file/%f:%l
```

Linked files are not limited to exceptions. For instance, the controller in the web debug toolbar becomes clickable after configuring the IDE.

5.5 Debugging Production

Debugging production servers is always trickier. You don't have access to the Symfony profiler for instance. Logs are less verbose. But tailing the logs is possible:

```
$ symfony cloud:logs --tail
```

You can even connect via SSH on the web container:

```
$ symfony cloud:ssh
```

Don't worry, you cannot break anything easily. Most of the filesystem is read-only. You won't be able to do a hot fix in production. But you will learn a much better way later in the book.



Going Further

- *SymfonyCasts Environments and Config Files tutorial*;
- *SymfonyCasts Environment Variables tutorial*;
- *SymfonyCasts Web Debug Toolbar and Profiler tutorial*;
- *Managing multiple .env files* in Symfony applications.

Step 6

Creating a Controller

Our guestbook project is already live on production servers but we cheated a little bit. The project doesn't have any web pages yet. The homepage is served as a boring 404 error page. Let's fix that.

When an HTTP request comes in, like for the homepage (<http://localhost:8000/>), Symfony tries to find a *route* that matches the *request path* (`/` here). A *route* is the link between the request path and a *PHP callable*, a function that creates the HTTP *response* for that request.

These callables are called “controllers”. In Symfony, most controllers are implemented as PHP classes. You can create such a class manually, but because we like to go fast, let's see how Symfony can help us.

6.1 Being Lazy with the Maker Bundle

To generate controllers effortlessly, we can use the `symfony/maker-bundle` package, which has been installed as part of the `webapp` package.

The maker bundle helps you generate a lot of different classes. We will use it all the time in this book. Each “generator” is defined in a command

and all commands are part of the `make` command namespace.

The Symfony Console built-in `list` command lists all commands available under a given namespace; use it to discover all generators provided by the maker bundle:

```
$ symfony console list make
```

6.2 Choosing a Configuration Format

Before creating the first controller of the project, we need to decide on the configuration formats we want to use. Symfony supports YAML, XML, PHP, and PHP attributes out of the box.

For *configuration related to packages*, YAML is the best choice. This is the format used in the `config/` directory. Often, when you install a new package, that package's recipe will add a new file ending in `.yaml` to that directory.

For *configuration related to PHP code*, *attributes* are a better choice as they are defined next to the code. Let me explain with an example. When a request comes in, some configuration needs to tell Symfony that the request path should be handled by a specific controller (a PHP class). When using YAML, XML or PHP configuration formats, two files are involved (the configuration file and the PHP controller file). When using attributes, the configuration is done directly in the controller class.

You might wonder how you can guess the package name you need to install for a feature? Most of the time, you don't need to know. In many cases, Symfony contains the package to install in its error messages. Running `symfony console make:message` without the `messenger` package for instance would have ended with an exception containing a hint about installing the right package.

6.3 Generating a Controller

Create your first *Controller* via the `make:controller` command:

```
$ symfony console make:controller ConferenceController
```

The command creates a `ConferenceController` class under the `src/Controller/` directory. The generated class consists of some boilerplate code ready to be fine-tuned:

```
src/Controller/ConferenceController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
    #[Route('/conference', name: 'conference')]
    public function index(): Response
    {
        return $this->render('conference/index.html.twig', [
            'controller_name' => 'ConferenceController',
        ]);
    }
}
```

The `#[Route('/conference', name: 'conference')]` attribute is what makes the `index()` method a controller (the configuration is next to the code that it configures).

When you hit `/conference` in a browser, the controller is executed and a response is returned.

Tweak the route to make it match the homepage:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -8,7 +8,7 @@ use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
-    #[Route('/conference', name: 'app_conference')]
+    #[Route('/', name: 'homepage')]
    public function index(): Response
    {
        return $this->render('conference/index.html.twig', [
```

The route name will be useful when we want to reference the homepage in the code. Instead of hard-coding the / path, we will use the route name. Instead of the default rendered page, let's return a simple HTML one:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -11,8 +11,13 @@ class ConferenceController extends AbstractController
    #[Route('/', name: 'homepage')]
    public function index(): Response
    {
-       return $this->render('conference/index.html.twig', [
-           'controller_name' => 'ConferenceController',
-       ]);
+       return new Response(<<<EOF
+           <html>
+               <body>
+                   
+               </body>
+           </html>
+           EOF
+       );
    }
}
```

Refresh the browser:



The main responsibility of a controller is to return an HTTP Response for the request.

As the rest of the chapter is about code that we won't keep, let's commit our changes now:

```
$ git add .
$ git commit -m'Add the index controller'
```


6.4 Adding an Easter Egg

To demonstrate how a response can leverage information from the request, let's add a small *Easter egg*. Whenever the homepage contains a query string like `?hello=Fabien`, let's add some text to greet the person:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -3,17 +3,24 @@
 namespace App\Controller;

 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
+use Symfony\Component\HttpFoundation\Request;
 use Symfony\Component\HttpFoundation\Response;
 use Symfony\Component\Routing\Annotation\Route;

 class ConferenceController extends AbstractController
 {
     #[Route('/', name: 'homepage')]
-    public function index(): Response
+    public function index(Request $request): Response
     {
+        $greet = '';
+        if ($name = $request->query->get('hello')) {
+            $greet = sprintf('<h1>Hello %s!</h1>', htmlspecialchars($name));
+        }
+
         return new Response(<<<EOF
             <html>
                 <body>
+                    $greet
                     
                 </body>
             </html>
         >>>);
     }
 }
```

Symfony exposes the request data through a `Request` object. When Symfony sees a controller argument with this type-hint, it automatically knows to pass it to you. We can use it to get the `name` item from the query string and add an `<h1>` title.

Try hitting `/` then `/?hello=Fabien` in a browser to see the difference.



Notice the call to `htmlspecialchars()` to avoid XSS issues. This is something that will be done automatically for us when we switch to a proper template engine.

We could also have made the name part of the URL:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -9,11 +9,11 @@ use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
-    #[Route('/', name: 'homepage')]
-    public function index(Request $request): Response
+    #[Route('/hello/{name}', name: 'homepage')]
+    public function index(string $name = ''): Response
    {
        $greet = '';
-        if ($name = $request->query->get('hello')) {
+        if ($name) {
            $greet = sprintf('<h1>Hello %s!</h1>', htmlspecialchars($name));
        }
    }
}
```

The {name} part of the route is a dynamic *route parameter* - it works like a wildcard. You can now hit /hello then /hello/Fabien in a browser to get the same results as before. You can get the *value* of the {name} parameter by adding a controller argument with the same *name*. So, \$name.

Revert the changes we have just made:

```
$ git checkout src/Controller/ConferenceController.php
```

6.5 Debugging Variables

A great debug helper is the Symfony `dump()` function. It is always available and allows you to dump complex variables in a nice and interactive format.

Temporarily change `src/Controller/ConferenceController.php` to dump the Request object:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -3,14 +3,17 @@
namespace App\Controller;

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

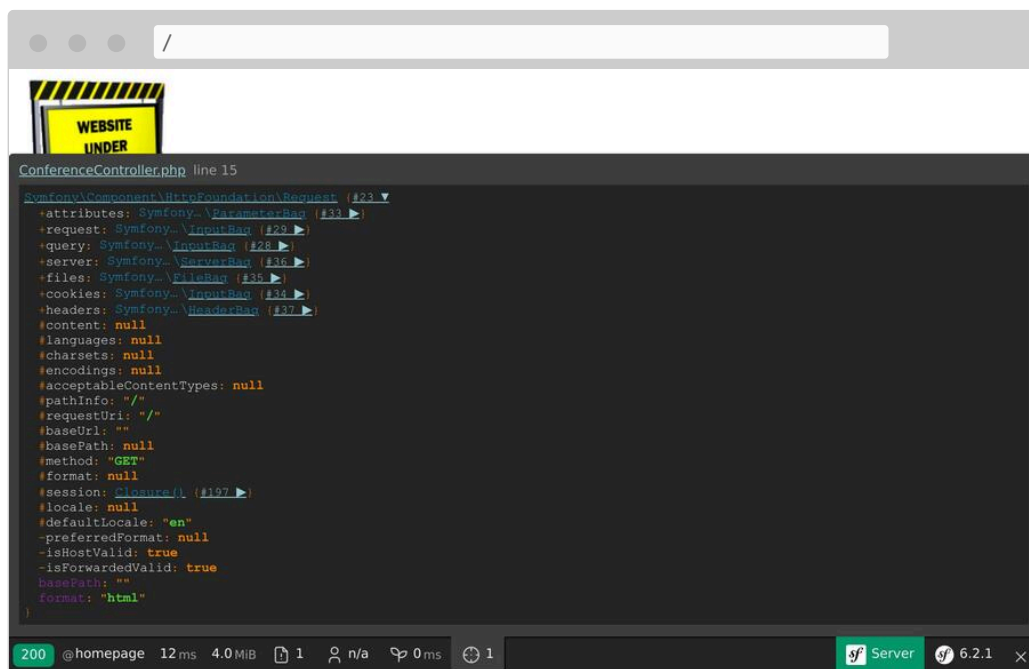
```

+use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
-    public function index(): Response
+    public function index(Request $request): Response
    {
+        dump($request);
+
        return new Response(<<<<EOF
            <html>
            <body>

```

When refreshing the page, notice the new “target” icon in the toolbar; it lets you inspect the dump. Click on it to access a full page where navigating is made simpler:



Revert the changes we have just made:

```
$ git checkout src/Controller/ConferenceController.php
```



Going Further

- The *Symfony Routing* system;
- *SymfonyCasts Routes, Controllers & Pages tutorial*;
- *PHP attributes*;
- The *HttpFoundation* component;
- *XSS (Cross-Site Scripting)* security attacks;
- The *Symfony Routing Cheat Sheet*.