Table of Contents

You are browsing the **Symfony 4 documentation**, which changes significantly from Symfony 3.x. If your app doesn't use Symfony 4 yet, browse the Symfony 3.4 documentation.

Configuring Symfony (and Environments)

4.2 version   edit this page

Symfony applications can install third-party packages (bundles, libraries, etc.) to bring in new features (*services*) to your project. Each package can be customized via configuration files that live - by default - in the `config/` directory.

Configuration: config/packages/ ¶

The configuration for each package can be found in `config/packages/`. For instance, the framework bundle is configured in `config/packages/framework.yaml`:

YAML   XML   PHP

```yaml
# config/packages/framework.yaml
framework:
    secret: '%env(APP_SECRET)%'
    #default_locale: en
    #csrf_protection: true
    #http_method_override: true

    # Enables session support. Note that the session will ONLY be started if you read or wri
    # Remove or comment this section to explicitly disable session support.
    session:
        handler_id: ~

    #esi: true
    #fragments: true
    php_errors:
        log: true
```

The top-level key (here `framework`) references configuration for a specific bundle (*FrameworkBundle* in this case).

Configuration Formats

Throughout the documentation, all configuration examples will be shown in three formats (YAML, XML and PHP). YAML is used by default, but you can choose whatever you like best. There is no performance difference:

- The YAML Format: Simple, clean and readable;
- *XML*: More powerful than YAML at times & supports IDE autocompletion;
- *PHP*: Very powerful but less readable than standard configuration formats.

Configuration Reference & Dumping ¶

There are *two* ways to know *what* keys you can configure:

1. Use the Reference Section;
2. Use the `config:dump-reference` command.

For example, if you want to configure something related to the framework bundle, you can see an example dump of all available configuration options by running:

```
$ php bin/console config:dump-reference framework
```

The parameters Key: Parameters (Variables) ¶

The configuration has some special top-level keys. One of them is called `parameters`: it's used to define *variables* that can be referenced in *any* other configuration file. For example, when you install the *translation* package, a `locale` parameter is added to `config/services.yaml`:

YAML XML PHP

```
# config/services.yaml
parameters:
    locale: en

# ...
```

This parameter is then referenced in the framework config in `config/packages/translation.yaml`:

YAML XML PHP

```yaml
# config/packages/translation.yaml
framework:
    # any string surrounded by two % is replaced by that parameter value
    default_locale: '%locale%'

    # ...
```

You can define whatever parameter names you want under the `parameters` key of any configuration file. To reference a parameter, surround its name with two percent signs - e.g. `%locale%`.

You can also set parameters dynamically, like from environment variables. See *How to Set external Parameters in the Service Container*.

For more information about parameters - including how to reference them from inside a controller - see Service Parameters.

The .env File & Environment Variables ¶

There is also a `.env` file which is loaded and its contents become environment variables. This is useful during development, or if setting environment variables is difficult for your deployment.

When you install packages, more environment variables are added to this file. But you can also add your own.

Environment variables can be referenced in any other configuration files by using a special syntax. For example, if you install the `doctrine` package, then you will have an environment variable called `DATABASE_URL` in your `.env` file. This is referenced inside `config/packages/doctrine.yaml`:

```yaml
# config/packages/doctrine.yaml
doctrine:
    dbal:
        url: '%env(DATABASE_URL)%'

        # The `resolve:` prefix replaces container params by their values inside the env var
        # url: '%env(resolve:DATABASE_URL)%'
```

For more details about environment variables, see Environment Variables.

Caution

Applications created before November 2018 had a slightly different system, involving a `.env.dist` file. For information about upgrading, see: *Nov 2018 Changes to .env & How to Update*.

The `.env` file is special, because it defines the values that usually change on each server. For example, the database credentials on your local development machine might be different from your workmates. The `.env` file should contain sensible, non-secret *default* values for all of your environment variables and *should* be commited to your repository.

To override these variables with machine-specific or sensitive values, create a `.env.local` file. This file is **not committed to the shared repository** and is only stored on your machine. In fact, the `.gitignore` file that comes with Symfony prevents it from being committed.

You can also create a few other `.env` files that will be loaded:

- `.env.{environment}`: e.g. `.env.test` will be loaded in the `test` environment and committed to your repository.
- `.env.{environment}.local`: e.g. `.env.prod.local` will be loaded in the `prod` environment but will *not* be committed to your repository.

If you decide to set real environment variables on production, the `.env` files *are* still loaded, but your real environment variables will override those values.

Environments & the Other Config Files ¶

You have just *one* app, but whether you realize it or not, you need it to behave *differently* at different times:

- While **developing**, you want your app to log everything and expose nice debugging tools;
- After deploying to **production**, you want that *same* app to be optimized for speed and only log errors.

How can you make *one* application behave in two different ways? With *environments*.

You've probably already been using the `dev` environment without even knowing it. After you deploy, you'll use the `prod` environment.

To learn more about *how* to execute and control each environment, see *How to Master and Create new Environments*.

Keep Going! ¶

Congratulations! You've tackled the basics in Symfony. Next, learn about *each* part of Symfony individually by following the guides. Check out:

- Forms
- Databases and the Doctrine ORM
- Service Container
- Security
- How to Send an Email
- Logging

And the many other topics.

Learn more ¶

- How to Organize Configuration Files
- Nov 2018 Changes to .env & How to Update
- How to Master and Create new Environments
- How to Set external Parameters in the Service Container
- Understanding how the Front Controller, Kernel and Environments Work together
- Building your own Framework with the MicroKernelTrait
- How To Create Symfony Applications with Multiple Kernels

- How to Override Symfony's default Directory Structure
- Using Parameters within a Dependency Injection Class

Latest from the Symfony Blog

Symfony 4.2.4 released
March 3, 2019

Symfony 3.4.23 released
March 3, 2019

They Help Us Make Symfony

Thanks **Jordan Hoff** for being a Symfony contributor.
1 commit · **2** lines

Get Involved in the Community

A passionate group of over 600,000 developers from more than 120 countries, all committed to helping PHP surpass the impossible.

Getting involved →

What is Symfony?
Symfony at a Glance
Symfony Components
Case Studies
Symfony Roadmap
Security Policy
Logo & Screenshots
Trademark & Licenses
symfony1 Legacy

Learn Symfony
Getting Started
Components
Best Practices
Bundles
Reference
Training
Certification

Screencasts
Learn Symfony
Learn PHP
Learn JavaScript
Learn Drupal
Learn RESTful APIs

Community
SymfonyConnect
Support
How to be Involved
Events & Meetups
Projects using Symfony
Downloads Stats