

Configuration

Configuration

Anonymous · 10/10/2026

Configuration

Directory Structure

The kernel expects a `config` directory for configurations. Any other structure is free.

The directories come from `Environment`:

Directory	Where it is
Project	The <code>project</code> directory if it is given to <code>Environment</code> ; otherwise the <code>PROJECT_DIR</code> of the context (what the application receives from the runtime); otherwise the root package of Composer, the project that is running. It is not deduced from where a package of the dependencies is installed.
Cache	<code><project>/var/cache/<environment></code> .
Configuration	The <code>config</code> directory of the project.
Logs	<code><project>/var/log</code> .

The configuration files (`services.yaml`, `routes.yaml`...) are read **from the configuration directory only**. A file with the same name in the directory where the process runs is not read, whatever the working directory is.

Service Configuration

Define services in `config/services.php`:

```
use Symfony\Component\DependencyInjection\Loader\Configurator\ContainerConfigurator;

return static function (ContainerConfigurator $configurator) {
    $services = $configurator->services();

    // Auto-configure and autowire services by default.
    $services->defaults()
        ->autowire()
        ->autoconfigure();
};
```

```
// Register a single service.
$services->set('app.service', AppService::class)
    ->public();

// Register multiple services from namespace.
$services->load('App\\Service\\', '../src/Service/*')
    ->public();

};
```

Route Configuration

Define routes in `config/routes.php` or `config/routes.yaml`:

```
// routes.php
return [
    'home' => [
        'path' => '/',
        'controller' => 'App\\Controller\\HomeController::index',
    ],
    'blog_show' => [
        'path' => '/blog/{slug}',
        'controller' => 'App\\Controller\\BlogController::show',
        'parameters' => [
            'requirements' => [
                'slug' => '[a-z0-9-]+',
            ],
        ],
    ],
];
```

Or in YAML:

```
# routes.yaml
home:
  path: /
  controller: App\\Controller\\HomeController::index

blog_show:
  path: /blog/{slug}
  controller: App\\Controller\\BlogController::show
  parameters:
    requirements:
      slug: '[a-z0-9-]+'
```

The routes of `routes.yaml` and the ones of `routes.php` are **added**: each file is loaded after the one before it, so the routes of all of them are kept, in the order the files were loaded. A name that two files define has the route of the later one (it keeps its place in the order). The imports of a YAML file

are added the same way.

The Cache of the Container

The container is built from the configuration and dumped to a PHP file in the cache directory (`var/cache/<environment>/container_<id>.php`, where the id is the hash of the class of the kernel). The next boot loads that file instead of building the container again.

- **In debug mode** the container is built and dumped on every boot, so a change in the configuration is seen right away.
- **Without debug** (production) the file that is there is used as it is. **It is never checked against the configuration:** what the container was built with stays until the file is deleted. That includes `services.yaml` and `routes.yaml` (also the ones that the packages bring, so a `composer update` that changes them), the classes that a resource glob such as `App\Controller\` finds, and the environment variables, because they are dumped as `env.<NAME>` parameters.

The kernel does not clear the cache: it is the deployment that has to start with an empty cache directory. The usual way is to not share `var/cache` between deploys (each release has its own), so every deploy builds the container once, with the configuration and the environment of that release. If the cache directory is shared or kept, delete `var/cache/<environment>` after changing the configuration, the `.env` or the dependencies.

Two kernels of the same class in the same process (for example in tests) do not share a container: the second one that is built gets its own, with the configuration of its moment. The cache file always declares the class with the name of the kernel, so the next process can load it.

Last updated on 10/10/2026