Symfony

The Reference Book

Version: 4.0

generated on May 24, 2018

The Reference Book (4.0)

This work is licensed under the "Attribution-Share Alike 3.0 Unported" license (*http://creativecommons.org/licenses/by-sa/3.0/*).

You are free **to share** (to copy, distribute and transmit the work), and **to remix** (to adapt the work) under the following conditions:

- **Attribution**: You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike**: If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor SensioLabs shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (*http://github.com/symfony/symfony-docs/issues*). Based on tickets and users feedback, this book is continuously updated.

Contents at a Glance

FrameworkBundle Configuration ("framework") 6
DoctrineBundle Configuration ("doctrine") 30
SecurityBundle Configuration ("security") 38
SwiftmailerBundle Configuration ("swiftmailer") 48
TwigBundle Configuration ("twig") 54
MonologBundle Configuration ("monolog") 60
WebProfilerBundle Configuration ("web_profiler") 62
DebugBundle Configuration ("debug") 64
Configuring in the Kernel 66
Form Types Reference 68
TextType Field 70
TextareaType Field 75
EmailType Field 80
IntegerType Field 85
MoneyType Field 91
NumberType Field 97
PasswordType Field 103
PercentType Field 108
SearchType Field 114
UrlType Field 118
RangeType Field 123
TelType Field 128
ColorType Field 133
ChoiceType Field (select drop-downs, radio buttons & checkboxes) 138
EntityType Field 152
CountryType Field 164
LanguageType Field 171
LocaleType Field 178
TimezoneType Field 185
CurrencyType Field 193
DateType Field 200
DateIntervalType Field 208
DateTimeType Field 215
TimeType Field 222
BirthdayType Field 229
CheckboxType Field 234

FileType Field 239
RadioType Field 244
CollectionType Field 249
RepeatedType Field 259
HiddenType Field 264
ButtonType Field 267
ResetType Field 269
SubmitType Field 271
FormType Field 274
Validation Constraints Reference 283
NotBlank 285
Blank 287
NotNull 289
IsNull 291
IsTrue 293
IsFalse 295
Type 297
Email 300
Length 302
Url 305
Regex 308
Ip 311
Uuid 314
Range 316
EqualTo 319
NotEqualTo 321
IdenticalTo 323
NotIdenticalTo 325
LessThan 327
LessThanOrEqual 330
GreaterThan 333
GreaterThanOrEqual 336
Date 339
DateTime 341
Time 343
Choice 345
Collection 349
Count 353
UniqueEntity 355
Language 358
Locale 360
Country 362
File 364
Image 368
CardScheme 374
Currency 376
Luhn 378

Iban 380
Bic 382
Isbn 384
Issn 386
Callback 388
Expression 392
All 395
UserPassword 397
Valid 399
Twig Template Form Function and Variable Reference 402
Symfony Twig Extensions 408
Built-in Symfony Service Tags 420
Built-in Symfony Events 433
Requirements for Running Symfony 437

Chapter 1
FrameworkBundle Configuration ("framework")

The FrameworkBundle defines the main framework configuration, from sessions and translations to forms, validation, routing and more. All these options are configured under the `framework` key in your application configuration.

Listing 1-1
```
# displays the default config values defined by Symfony
$ php bin/console config:dump framework

# displays the actual config values used by your application
$ php bin/console debug:config framework
```

When using XML, you must use the `http://symfony.com/schema/dic/symfony` namespace and the related XSD schema is available at: `http://symfony.com/schema/dic/symfony/symfony-1.0.xsd`

Configuration

- secret
- http_method_override
- ide
- test
- default_locale
- trusted_hosts
- **form**
    - enabled

- **csrf_protection**
    - enabled
- **esi**
    - enabled
- **fragments**
    - enabled
    - path
- **profiler**
    - enabled
    - collect
    - only_exceptions
    - only_master_requests
    - dsn
- **request:**
    - formats
- **router**
    - resource
    - type
    - http_port
    - https_port
    - strict_requirements
- **session**
    - storage_id
    - handler_id
    - name
    - cookie_lifetime
    - cookie_path
    - cookie_domain
    - cookie_secure
    - cookie_httponly
    - gc_divisor
    - gc_probability
    - gc_maxlifetime
    - save_path
    - metadata_update_threshold
- **assets**
    - base_path
    - base_urls
    - packages
    - version_strategy
    - version
    - version_format
    - json_manifest_path

- **templating**
    - hinclude_default_template
    - **form**
        - resources
    - cache
    - engines
    - loaders
- **translator**
    - enabled
    - fallbacks
    - logging
    - paths
- **property_access**
    - magic_call
    - throw_exception_on_invalid_index
- **validation**
    - enabled
    - cache
    - enable_annotations
    - translation_domain
    - strict_email
    - **mapping**
        - paths
- **annotations**
    - cache
    - file_cache_dir
    - debug
- **serializer**
    - enabled
    - enable_annotations
    - name_converter
    - circular_reference_handler
    - **mapping**
        - paths
- **php_errors**
    - log
    - throw

- **cache**
    - app
    - system
    - directory
    - default_doctrine_provider
    - default_psr6_provider
    - default_redis_provider
    - default_memcached_provider
    - **pools**
        - **name**
            - adapter
            - public
            - default_lifetime
            - provider
            - clearer
    - prefix_seed
- lock

secret

type: `string` **required**

This is a string that should be unique to your application and it's commonly used to add more entropy to security related operations. Its value should be a series of characters, numbers and symbols chosen randomly and the recommended length is around 32 characters.

In practice, Symfony uses this value for encrypting the cookies used in the *remember me functionality* and for creating signed URIs when using ESI (Edge Side Includes).

This option becomes the service container parameter named `kernel.secret`, which you can use whenever the application needs an immutable random string to add more entropy.

As with any other security-related parameter, it is a good practice to change this value from time to time. However, keep in mind that changing this value will invalidate all signed URIs and Remember Me cookies. That's why, after changing this value, you should regenerate the application cache and log out all the application users.

http_method_override

type: `boolean` **default**: `true`

This determines whether the `_method` request parameter is used as the intended HTTP method on POST requests. If enabled, the *`Request::enableHttpMethodParameterOverride`*[1] method gets called automatically. It becomes the service container parameter named `kernel.http_method_override`.

For more information, see How to Change the Action and Method of a Form.

1. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html#method_enableHttpMethodParameterOverride

If you're using the HttpCache Reverse Proxy with this option, the kernel will ignore the `_method` parameter, which could lead to errors.

To fix this, invoke the `enableHttpMethodParameterOverride()` method before creating the `Request` object:

Listing 1-2
```php
// public/index.php

// ...
$kernel = new CacheKernel($kernel);

Request::enableHttpMethodParameterOverride(); // <-- add this line
$request = Request::createFromGlobals();
// ...
```

trusted_proxies

The `trusted_proxies` option was removed in Symfony 3.3. See *How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy*.

ide

type: `string` **default**: `null`

Symfony turns file paths seen in variable dumps and exception messages into links that open those files right inside your browser. If you prefer to open those files in your favorite IDE or text editor, set this option to any of the following values: `phpstorm`, `sublime`, `textmate`, `macvim` and `emacs`.

The `phpstorm` option is supported natively by PhpStorm on MacOS, Windows requires *PhpStormProtocol*[2] and Linux requires *phpstorm-url-handler*[3].

If you use another editor, the expected configuration value is a URL template that contains an `%f` placeholder where the file path is expected and `%l` placeholder for the line number (percentage signs (`%`) must be escaped by doubling them to prevent Symfony from interpreting them as container parameters).

Listing 1-3
```yaml
# config/packages/framework.yaml
framework:
    ide: 'myide://open?url=file://%%f&line=%%l'
```

Since every developer uses a different IDE, the recommended way to enable this feature is to configure it on a system level. This can be done by setting the `xdebug.file_link_format` option in your `php.ini` configuration file. The format to use is the same as for the `framework.ide` option, but without the need to escape the percent signs (`%`) by doubling them.

If both `framework.ide` and `xdebug.file_link_format` are defined, Symfony uses the value of the `xdebug.file_link_format` option.

Setting the `xdebug.file_link_format` ini option works even if the Xdebug extension is not enabled.

2. https://github.com/aik099/PhpStormProtocol
3. https://github.com/sanduhrs/phpstorm-url-handler

When running your app in a container or in a virtual machine, you can tell Symfony to map files from the guest to the host by changing their prefix. This map should be specified at the end of the URL template, using `&` and `>` as guest-to-host separators:

Listing 1-4

```
// /path/to/guest/.../file will be opened
// as /path/to/host/.../file on the host
// and /foo/.../file as /bar/.../file also
'myide://%f:%l&/path/to/guest/>/path/to/host/&/foo/>/bar/&...'
```

test

type: `boolean`

If this configuration setting is present (and not `false`), then the services related to testing your application (e.g. `test.client`) are loaded. This setting should be present in your `test` environment (usually via `config/packages/test/framework.yaml`).

For more information, see Testing.

default_locale

type: `string` **default**: `en`

The default locale is used if no `_locale` routing parameter has been set. It is available with the *`Request::getDefaultLocale`*[4] method.

You can read more information about the default locale in Setting a Default Locale.

trusted_hosts

type: `array` | `string` **default**: `array()`

A lot of different attacks have been discovered relying on inconsistencies in handling the `Host` header by various software (web servers, reverse proxies, web frameworks, etc.). Basically, every time the framework is generating an absolute URL (when sending an email to reset a password for instance), the host might have been manipulated by an attacker.

You can read "HTTP Host header attacks[5]" for more information about these kinds of attacks.

The Symfony *`Request::getHost()`*[6] method might be vulnerable to some of these attacks because it depends on the configuration of your web server. One simple solution to avoid these attacks is to whitelist the hosts that your Symfony application can respond to. That's the purpose of this `trusted_hosts` option. If the incoming request's hostname doesn't match one in this list, the application won't respond and the user will receive a 400 response.

Listing 1-5

```
# config/packages/framework.yaml
framework:
    trusted_hosts:  ['example.com', 'example.org']
```

Hosts can also be configured using regular expressions (e.g. `^(.+\.)?example.com$`), which make it easier to respond to any subdomain.

---

4. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html#method_getDefaultLocale
5. *http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html*
6. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html#method_getHost

In addition, you can also set the trusted hosts in the front controller using the `Request::setTrustedHosts()` method:

Listing 1-6
```
// public/index.php
Request::setTrustedHosts(array('^(.+\.)?example.com$', '^(.+\.)?example.org$'));
```

The default value for this option is an empty array, meaning that the application can respond to any given host.

Read more about this in the Security Advisory Blog post[7].

form

enabled

type: `boolean` **default**: `true` or `false` depending on your installation

Whether to enable the form services or not in the service container. If you don't use forms, setting this to `false` may increase your application's performance because less services will be loaded into the container.

This option will automatically be set to `true` when one of the child settings is configured.

This will automatically enable the validation.

For more details, see Forms.

csrf_protection

For more information about CSRF protection, see How to Implement CSRF Protection.

enabled

type: `boolean` **default**: `true` or `false` depending on your installation

This option can be used to disable CSRF protection on *all* forms. But you can also disable CSRF protection on individual forms.

If you're using forms, but want to avoid starting your session (e.g. using forms in an API-only website), `csrf_protection` will need to be set to `false`.

esi

You can read more about Edge Side Includes (ESI) in Working with Edge Side Includes.

enabled

type: `boolean` **default**: `false`

Whether to enable the edge side includes support in the framework.

7. *https://symfony.com/blog/security-releases-symfony-2-0-24-2-1-12-2-2-5-and-2-3-3-released#cve-2013-4752-request-gethost-poisoning*

You can also set `esi` to `true` to enable it:

Listing 1-7
```yaml
# config/packages/framework.yaml
framework:
    esi: true
```

fragments

Learn more about fragments in the HTTP Cache article.

enabled

type: `boolean` **default**: `false`

Whether to enable the fragment listener or not. The fragment listener is used to render ESI fragments independently of the rest of the page.

This setting is automatically set to `true` when one of the child settings is configured.

path

type: `string` **default**: `'/_fragment'`

The path prefix for fragments. The fragment listener will only be executed when the request starts with this path.

profiler

enabled

type: `boolean` **default**: `false`

The profiler can be enabled by setting this option to `true`. When you install it using Symfony Flex, the profiler is enabled in the `dev` and `test` environments.

The profiler works independently from the Web Developer Toolbar, see the *WebProfilerBundle configuration* on how to disable/enable the toolbar.

collect

type: `boolean` **default**: `true`

This option configures the way the profiler behaves when it is enabled. If set to `true`, the profiler collects data for all requests. If you want to only collect information on-demand, you can set the `collect` flag to `false` and activate the data collectors manually:

Listing 1-8
```php
$profiler->enable();
```

only_exceptions

type: `boolean` **default**: `false`

When this is set to `true`, the profiler will only be enabled when an exception is thrown during the handling of the request.

only_master_requests

type: `boolean` **default**: `false`

When this is set to `true`, the profiler will only be enabled on the master requests (and not on the subrequests).

dsn

type: `string` **default**: `'file:%kernel.cache_dir%/profiler'`

The DSN where to store the profiling information.

See Switching the Profiler Storage for more information about the profiler storage.

request

formats

type: `array` **default**: `[]`

This setting is used to associate additional request formats (e.g. `html`) to one or more mime types (e.g. `text/html`), which will allow you to use the format & mime types to call *`Request::getFormat($mimeType)`*[8] or *`Request::getMimeType($format)`*[9].

In practice, this is important because Symfony uses it to automatically set the `Content-Type` header on the `Response` (if you don't explicitly set one). If you pass an array of mime types, the first will be used for the header.

To configure a `jsonp` format:

Listing 1-9
```yaml
# config/packages/framework.yaml
framework:
    request:
        formats:
            jsonp: 'application/javascript'
```

router

resource

type: `string` **required**

The path the main routing resource (e.g. a YAML file) that contains the routes and imports the router should load.

type

type: `string`

The type of the resource to hint the loaders about the format. This isn't needed when you use the default routers with the expected file extensions (`.xml`, `.yml` or `.yaml`, `.php`).

http_port

type: `integer` **default**: `80`

8. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html#method_getFormat
9. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html#method_getMimeType

The port for normal http requests (this is used when matching the scheme).

https_port

type: `integer` **default**: `443`

The port for https requests (this is used when matching the scheme).

strict_requirements

type: `mixed` **default**: `true`

Determines the routing generator behaviour. When generating a route that has specific *requirements*, the generator can behave differently in case the used parameters do not meet these requirements.

The value can be one of:

`true`
: Throw an exception when the requirements are not met;

`false`
: Disable exceptions when the requirements are not met and return `null` instead;

`null`
: Disable checking the requirements (thus, match the route even when the requirements don't match).

`true` is recommended in the development environment, while `false` or `null` might be preferred in production.

session

storage_id

type: `string` **default**: `'session.storage.native'`

The service id used for session storage. The `session.storage` service alias will be set to this service id. This class has to implement *`SessionStorageInterface`*[10].

handler_id

type: `string` **default**: `'session.handler.native_file'`

The service id used for session storage. The `session.handler` service alias will be set to this service id.

You can also set it to `null`, to default to the handler of your PHP installation.

> *You can see an example of the usage of this in How to Use PdoSessionHandler to Store Sessions in the Database.*

name

type: `string` **default**: `null`

This specifies the name of the session cookie. By default it will use the cookie name which is defined in the `php.ini` with the `session.name` directive.

10. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.html

cookie_lifetime

type: `integer` **default**: `null`

This determines the lifetime of the session - in seconds. The default value - `null` - means that the `session.cookie_lifetime` value from `php.ini` will be used. Setting this value to `0` means the cookie is valid for the length of the browser session.

cookie_path

type: `string` **default**: `/`

This determines the path to set in the session cookie. By default it will use `/`.

cookie_domain

type: `string` **default**: `''`

This determines the domain to set in the session cookie. By default it's blank, meaning the host name of the server which generated the cookie according to the cookie specification.

cookie_secure

type: `boolean` **default**: `false`

This determines whether cookies should only be sent over secure connections.

cookie_httponly

type: `boolean` **default**: `true`

This determines whether cookies should only be accessible through the HTTP protocol. This means that the cookie won't be accessible by scripting languages, such as JavaScript. This setting can effectively help to reduce identity theft through XSS attacks.

gc_divisor

type: `integer` **default**: `100`

See gc_probability.

gc_probability

type: `integer` **default**: `1`

This defines the probability that the garbage collector (GC) process is started on every session initialization. The probability is calculated by using `gc_probability` / `gc_divisor`, e.g. 1/100 means there is a 1% chance that the GC process will start on each request.

gc_maxlifetime

type: `integer` **default**: `1440`

This determines the number of seconds after which data will be seen as "garbage" and potentially cleaned up. Garbage collection may occur during session start and depends on gc_divisor and gc_probability.

save_path

type: `string` **default**: `%kernel.cache_dir%/sessions`

This determines the argument to be passed to the save handler. If you choose the default file handler, this is the path where the session files are created. For more information, see *Configuring the Directory where Session Files are Saved*.

You can also set this value to the `save_path` of your `php.ini` by setting the value to `null`:

Listing 1-10
```yaml
# config/packages/framework.yaml
framework:
    session:
        save_path: ~
```

metadata_update_threshold

type: `integer` **default**: `0`

This is how many seconds to wait between updating/writing the session metadata. This can be useful if, for some reason, you want to limit the frequency at which the session persists.

Starting in Symfony 3.4, session data is *only* written when the session data has changed. Previously, you needed to set this option to avoid that behavior.

assets

base_path

type: `string`

This option allows you to define a base path to be used for assets:

Listing 1-11
```yaml
# config/packages/framework.yaml
framework:
    # ...
    assets:
        base_path: '/images'
```

base_urls

type: `array`

This option allows you to define base URLs to be used for assets. If multiple base URLs are provided, Symfony will select one from the collection each time it generates an asset's path:

Listing 1-12
```yaml
# config/packages/framework.yaml
framework:
    # ...
    assets:
        base_urls:
            - 'http://cdn.example.com/'
```

packages

You can group assets into packages, to specify different base URLs for them:

Listing 1-13
```yaml
# config/packages/framework.yaml
framework:
    # ...
    assets:
        packages:
            avatars:
                base_urls: 'http://static_cdn.example.com/avatars'
```

Now you can use the `avatars` package in your templates:

Listing 1-14
```twig
<img src="{{ asset('...', 'avatars') }}">
```

Each package can configure the following options:

- base_path
- base_urls
- version_strategy
- version
- version_format
- json_manifest_path

version

type: `string`

This option is used to *bust* the cache on assets by globally adding a query parameter to all rendered asset paths (e.g. `/images/logo.png?v2`). This applies only to assets rendered via the Twig `asset()` function (or PHP equivalent) as well as assets rendered with Assetic.

For example, suppose you have the following:

Listing 1-15
```twig
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />
```

By default, this will render a path to your image such as `/images/logo.png`. Now, activate the `version` option:

Listing 1-16
```yaml
# config/packages/framework.yaml
framework:
    # ...
    assets:
        version: 'v2'
```

Now, the same asset will be rendered as `/images/logo.png?v2` If you use this feature, you **must** manually increment the `version` value before each deployment so that the query parameters change.

You can also control how the query string works via the version_format option.

This parameter cannot be set at the same time as `version_strategy` or `json_manifest_path`.

As with all settings, you can use a parameter as value for the `version`. This makes it easier to increment the cache on each deployment.

version_format

type: `string` **default**: `%%s?%%s`

This specifies a *sprintf*[11] pattern that will be used with the version option to construct an asset's path. By default, the pattern adds the asset's version as a query string. For example, if `version_format` is set to `%%s?version=%%s` and `version` is set to `5`, the asset's path would be `/images/logo.png?version=5`.

11. https://secure.php.net/manual/en/function.sprintf.php

All percentage signs (%) in the format string must be doubled to escape the character. Without escaping, values might inadvertently be interpreted as Service Parameters.

Some CDN's do not support cache-busting via query strings, so injecting the version into the actual file path is necessary. Thankfully, `version_format` is not limited to producing versioned query strings.

The pattern receives the asset's original path and version as its first and second parameters, respectively. Since the asset's path is one parameter, you cannot modify it in-place (e.g. `/images/logo-v5.png`); however, you can prefix the asset's path using a pattern of `version-%%2$s/%%1$s`, which would result in the path `version-5/images/logo.png`.

URL rewrite rules could then be used to disregard the version prefix before serving the asset. Alternatively, you could copy assets to the appropriate version path as part of your deployment process and forgot any URL rewriting. The latter option is useful if you would like older asset versions to remain accessible at their original URL.

version_strategy

type: `string` **default**: `null`

The service id of the *asset version strategy* applied to the assets. This option can be set globally for all assets and individually for each asset package:

Listing 1-17

```yaml
# config/packages/framework.yaml
framework:
    assets:
        # this strategy is applied to every asset (including packages)
        version_strategy: 'app.asset.my_versioning_strategy'
        packages:
            foo_package:
                # this package removes any versioning (its assets won't be versioned)
                version: ~
            bar_package:
                # this package uses its own strategy (the default strategy is ignored)
                version_strategy: 'app.asset.another_version_strategy'
            baz_package:
                # this package inherits the default strategy
                base_path: '/images'
```

This parameter cannot be set at the same time as `version` or `json_manifest_path`.

json_manifest_path

type: `string` **default**: `null`

The file path to a `manifest.json` file containing an associative array of asset names and their respective compiled names. A common cache-busting technique using a "manifest" file works by writing out assets with a "hash" appended to their file names (e.g. `main.ae433f1cb.css`) during a front-end compilation routine.

Symfony's Webpack Encore supports outputting hashed assets. Moreover, this can be incorporated into many other workflows, including Webpack and Gulp using *webpack-manifest-plugin*[12] and *gulp-rev*[13], respectively.

This option can be set globally for all assets and individually for each asset package:

Listing 1-18
```yaml
# config/packages/framework.yaml
framework:
    assets:
        # this manifest is applied to every asset (including packages)
        json_manifest_path: "%kernel.project_dir%/public/build/manifest.json"
        packages:
            foo_package:
                # this package uses its own manifest (the default file is ignored)
                json_manifest_path: "%kernel.project_dir%/public/build/a_different_manifest.json"
            bar_package:
                # this package uses the global manifest (the default file is used)
                base_path: '/images'
```

This parameter cannot be set at the same time as `version` or `version_strategy`. Additionally, this option cannot be nullified at the package scope if a global manifest file is specified.

If you request an asset that is *not found* in the `manifest.json` file, the original - *unmodified* - asset path will be returned.

templating

hinclude_default_template

type: `string` **default**: `null`

Sets the content shown during the loading of the fragment or when JavaScript is disabled. This can be either a template name or the content itself.

See How to Embed Asynchronous Content with hinclude.js for more information about hinclude.

form

resources

type: `string[]` **default**: `['FrameworkBundle:Form']`

A list of all resources for form theming in PHP. This setting is not required if you're using the Twig format for your templates, in that case refer to the form article.

Assume you have custom global form themes in `templates/form_themes/`, you can configure this like:

Listing 1-19
```yaml
# config/packages/framework.yaml
framework:
    templating:
        form:
            resources:
                - 'form_themes'
```

The default form templates from `FrameworkBundle:Form` will always be included in the form resources.

12. https://www.npmjs.com/package/webpack-manifest-plugin
13. https://www.npmjs.com/package/gulp-rev

See Global Form Theming for more information.

cache

type: `string`

The path to the cache directory for templates. When this is not set, caching is disabled.

When using Twig templating, the caching is already handled by the TwigBundle and doesn't need to be enabled for the FrameworkBundle.

engines

type: `string[]` / `string` **required**

The Templating Engine to use. This can either be a string (when only one engine is configured) or an array of engines.

At least one engine is required.

loaders

type: `string[]`

An array (or a string when configuring just one loader) of service ids for templating loaders. Templating loaders are used to find and load templates from a resource (e.g. a filesystem or database). Templating loaders must implement *`LoaderInterface`*[14].

translator

enabled

type: `boolean` **default**: `true` or `false` depending on your installation

Whether or not to enable the `translator` service in the service container.

fallbacks

type: `string|array` **default**: `array('en')`

This option is used when the translation key for the current locale wasn't found.

For more details, see Translations.

logging

default: `true` when the debug mode is enabled, `false` otherwise.

When `true`, a log entry is made whenever the translator cannot find a translation for a given key. The logs are made to the `translation` channel and at the `debug` for level for keys where there is a translation in the fallback locale and the `warning` level if there is no translation to use at all.

14. https://api.symfony.com/4.0/Symfony/Component/Templating/Loader/LoaderInterface.html

paths

type: `array` **default**: `[]`

This option allows to define an array of paths where the component will look for translation files.

property_access

magic_call

type: `boolean` **default**: `false`

When enabled, the `property_accessor` service uses PHP's magic __call() method when its `getValue()` method is called.

throw_exception_on_invalid_index

type: `boolean` **default**: `false`

When enabled, the `property_accessor` service throws an exception when you try to access an invalid index of an array.

validation

enabled

type: `boolean` **default**: `true` or `false` depending on your installation

Whether or not to enable validation support.

This option will automatically be set to `true` when one of the child settings is configured.

cache

type: `string`

The service that is used to persist class metadata in a cache. The service has to implement the *`CacheInterface`*[15].

Set this option to `validator.mapping.cache.doctrine.apc` to use the APC cache provide from the Doctrine project.

enable_annotations

type: `boolean` **default**: `false`

If this option is enabled, validation constraints can be defined using annotations.

translation_domain

type: `string` **default**: `validators`

The translation domain that is used when translating validation constraint error messages.

strict_email

type: `Boolean` **default**: `false`

15. https://api.symfony.com/4.0/Symfony/Component/Validator/Mapping/Cache/CacheInterface.html

If this option is enabled, the *egulias/email-validator*[16] library will be used by the *Email* constraint validator. Otherwise, the validator uses a simple regular expression to validate email addresses.

mapping

paths

type: `array` **default**: `[]`

This option allows to define an array of paths with files or directories where the component will look for additional validation files.

annotations

cache

type: `string` **default**: `'file'`

This option can be one of the following values:

file
: Use the filesystem to cache annotations

none
: Disable the caching of annotations

a service id
: A service id referencing a *Doctrine Cache*[17] implementation

file_cache_dir

type: `string` **default**: `'%kernel.cache_dir%/annotations'`

The directory to store cache files for annotations, in case `annotations.cache` is set to `'file'`.

debug

type: `boolean` **default**: `%kernel.debug%`

Whether to enable debug mode for caching. If enabled, the cache will automatically update when the original file is changed (both with code and annotation changes). For performance reasons, it is recommended to disable debug mode in production, which will happen automatically if you use the default value.

serializer

enabled

type: `boolean` **default**: `true` or `false` depending on your installation

Whether to enable the `serializer` service or not in the service container.

enable_annotations

type: `boolean` **default**: `false`

If this option is enabled, serialization groups can be defined using annotations.

---

16. https://github.com/egulias/EmailValidator
17. http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/caching.html

For more information, see Using Serialization Groups Annotations.

name_converter

type: `string`

The name converter to use. The *`CamelCaseToSnakeCaseNameConverter`*[18] name converter can enabled by using the `serializer.name_converter.camel_case_to_snake_case` value.

For more information, see Converting Property Names when Serializing and Deserializing.

circular_reference_handler

type `string`

The service id that is used as the circular reference handler of the default serializer. The service has to implement the magic `__invoke($object)` method.

For more information, see Handling Circular References.

mapping

paths

type: `array` **default**: `[]`

This option allows to define an array of paths with files or directories where the component will look for additional serialization files.

php_errors

log

type: `boolean` **default**: `false`

Use the application logger instead of the PHP logger for logging PHP errors.

throw

type: `boolean` **default**: `%kernel.debug%`

Throw PHP errors as `\ErrorException` instances. The parameter `debug.error_handler.throw_at` controls the threshold.

cache

app

type: `string` **default**: `cache.adapter.filesystem`

The cache adapter used by the `cache.app` service. The FrameworkBundle ships with multiple adapters: `cache.adapter.apcu`, `cache.adapter.doctrine`, `cache.adapter.system`, `cache.adapter.filesystem`, `cache.adapter.psr6`, `cache.adapter.redis` and `cache.adapter.memcached`.

18. https://api.symfony.com/4.0/Symfony/Component/Serializer/NameConverter/CamelCaseToSnakeCaseNameConverter.html

It might be tough to understand at the beginning, so to avoid confusion remember that all pools perform the same actions but on different medium given the adapter they are based on. Internally, a pool wraps the definition of an adapter.

system

type: `string` **default**: `cache.adapter.system`

The cache adapter used by the `cache.system` service. It supports the same adapters available for the `cache.app` service.

directory

type: `string` **default**: `%kernel.cache_dir%/pools`

The path to the cache directory used by services inheriting from the `cache.adapter.filesystem` adapter (including `cache.app`).

default_doctrine_provider

type: `string`

The service name to use as your default Doctrine provider. The provider is available as the `cache.doctrine` service.

default_psr6_provider

type: `string`

The service name to use as your default PSR-6 provider. It is available as the `cache.psr6` service.

default_redis_provider

type: `string` **default**: `redis://localhost`

The DSN to use by the Redis provider. The provider is available as the `cache.redis` service.

default_memcached_provider

type: `string` **default**: `memcached://localhost`

The DSN to use by the Memcached provider. The provider is available as the `cache.memcached` service.

pools

type: `array`

A list of cache pools to be created by the framework extension.

For more information about how pools works, see cache pools.

name

type: `prototype`

Name of the pool you want to create.

Your pool name must differ from `cache.app` or `cache.system`.

adapter

type: `string` **default**: `cache.app`

The name of the adapter to use. You could also use your own implementation.

Your service MUST implement the *`CacheItemPoolInterface`*[19] interface.

public

type: `boolean` **default**: `false`

Whether your service should be public or not.

default_lifetime

type: `integer`

Default lifetime of your cache items in seconds.

provider

type: `string`

The service name to use as provider when the specified adapter needs one.

clearer

type: `string`

The cache clearer used to clear your PSR-6 cache.

For more information, see `Psr6CacheClearer`[20].

prefix_seed

type: `string` **default**: `null`

If defined, this value is used as part of the "namespace" generated for the cache item keys. A common practice is to use the unique name of the application (e.g. `symfony.com`) because that prevents naming collisions when deploying multiple applications into the same path (on different servers) that share the same cache backend.

It's also useful when using *blue/green deployment*[21] strategies and more generally, when you need to abstract out the actual deployment directory (for example, when warming caches offline).

lock

type: `string`

The default lock adapter. If not defined, the value is set to `semaphore` when available, or to `flock` otherwise. Store's DSN are also allowed.

---

19. https://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html
20. *https://api.symfony.com/4.0/Symfony/Component/HttpKernel/CacheClearer/Psr6CacheClearer.html*
21. http://martinfowler.com/bliki/BlueGreenDeployment.html

Full Default Configuration

Listing 1-20

```yaml
framework:
    secret:               ~
    http_method_override: true
    trusted_proxies:      []
    ide:                  ~
    test:                 ~
    default_locale:       en

    csrf_protection:
        enabled:              false

    # form configuration
    form:
        enabled:              false
        csrf_protection:
            enabled:          true
            field_name:       ~

    # esi configuration
    esi:
        enabled:              false

    # fragments configuration
    fragments:
        enabled:              false
        path:                 /_fragment

    # profiler configuration
    profiler:
        enabled:              false
        collect:              true
        only_exceptions:      false
        only_master_requests: false
        dsn:                  file:%kernel.cache_dir%/profiler

    # router configuration
    router:
        resource:             ~ # Required
        type:                 ~
        http_port:            80
        https_port:           443

        # * set to true to throw an exception when a parameter does not
        #   match the requirements
        # * set to false to disable exceptions when a parameter does not
        #   match the requirements (and return null instead)
        # * set to null to disable parameter checks against requirements
        #
        # 'true' is the preferred configuration in development mode, while
        # 'false' or 'null' might be preferred in production
        strict_requirements:  true

    # session configuration
    session:
        storage_id:           session.storage.native
        handler_id:           session.handler.native_file
        name:                 ~
        cookie_lifetime:      ~
        cookie_path:          ~
        cookie_domain:        ~
        cookie_secure:        ~
        cookie_httponly:      ~
        gc_divisor:           ~
        gc_probability:       ~
        gc_maxlifetime:       ~
        save_path:            '%kernel.cache_dir%/sessions'

    # serializer configuration
```

```yaml
    serializer:
       enabled:                    false
       cache:                       ~
       name_converter:              ~
       circular_reference_handler: ~

    # assets configuration
    assets:
        base_path:          ~
        base_urls:          []
        version:            ~
        version_format:     '%%s?%%s'
        packages:

            # Prototype
            name:
                base_path:            ~
                base_urls:            []
                version:              ~
                version_format:       '%%s?%%s'

    # templating configuration
    templating:
        hinclude_default_template:  ~
        form:
            resources:

                # Default:
                - FrameworkBundle:Form
        cache:                ~
        engines:              # Required

            # Example:
            - twig
        loaders:              []

    # translator configuration
    translator:
        enabled:              false
        fallbacks:            [en]
        logging:              "%kernel.debug%"
        paths:                []

    # validation configuration
    validation:
        enabled:              false
        cache:                ~
        enable_annotations:   false
        translation_domain:   validators
        mapping:
            paths:            []

    # annotation configuration
    annotations:
        cache:                file
        file_cache_dir:       '%kernel.cache_dir%/annotations'
        debug:                '%kernel.debug%'

    # PHP errors handling configuration
    php_errors:
        log:                  false
        throw:                '%kernel.debug%'

    # cache configuration
    cache:
        app: cache.app
        system: cache.system
        directory: '%kernel.cache_dir%/pools'
        default_doctrine_provider: ~
        default_psr6_provider: ~
        default_redis_provider: 'redis://localhost'
```

```yaml
        default_memcached_provider: 'memcached://localhost'
        pools:
            # Prototype
            name:
                adapter: cache.app
                public: false
                default_lifetime: ~
                provider: ~
                clearer: ~

    # lock configuration
    lock:
        invoice: 'redis://localhost'
        report: semaphore
    # lock: ~
    # lock: 'flock'
    # lock: ['semaphore', 'redis://localhost']
```

Chapter 2
DoctrineBundle Configuration ("doctrine")

Full Default Configuration

Listing 2-1

```yaml
# config/packages/doctrine.yaml
doctrine:
    dbal:
        default_connection:   default
        types:
            # A collection of custom types
            # Example
            some_custom_type:
                class:               App\DBAL\MyCustomType
                commented:           true

        connections:
            # A collection of different named connections (e.g. default, conn2, etc)
            default:
                dbname:              ~
                host:                localhost
                port:                ~
                user:                root
                password:            ~
                # charset of the database
                charset:             ~
                # charset and collation of the tables. Not inherited from database
                default_table_options:
                    charset:         ~
                    collate:         ~
                path:                ~
                memory:              ~

                # The unix socket to use for MySQL
                unix_socket:         ~

                # True to use as persistent connection for the ibm_db2 driver
                persistent:          ~

                # The protocol to use for the ibm_db2 driver (default to TCPIP if omitted)
                protocol:            ~

```

```
                # True to use dbname as service name instead of SID for Oracle
                service:              ~

                # The session mode to use for the oci8 driver
                sessionMode:          ~

                # True to use a pooled server with the oci8 driver
                pooled:               ~

                # Configuring MultipleActiveResultSets for the pdo_sqlsrv driver
                MultipleActiveResultSets:  ~
                driver:               pdo_mysql
                platform_service:     ~

                # the version of your database engine
                server_version:       ~

                # when true, queries are logged to a 'doctrine' monolog channel
                logging:              '%kernel.debug%'
                profiling:            '%kernel.debug%'
                driver_class:         ~
                wrapper_class:        ~
                # the DBAL keepSlave option
                keep_slave:           false
                options:
                    # an array of options
                    key:                  []
                mapping_types:
                    # an array of mapping types
                    name:                 []

                # If defined, only the tables whose names match this regular expression are managed
                # by the schema tool (in this example, any table name not starting with `wp_`)
                #schema_filter:               '/^(?!wp_)/'

                slaves:

                    # a collection of named slave connections (e.g. slave1, slave2)
                    slave1:
                        dbname:               ~
                        host:                 localhost
                        port:                 ~
                        user:                 root
                        password:             ~
                        charset:              ~
                        path:                 ~
                        memory:               ~

                        # The unix socket to use for MySQL
                        unix_socket:          ~

                        # True to use as persistent connection for the ibm_db2 driver
                        persistent:           ~

                        # The protocol to use for the ibm_db2 driver (default to TCPIP if omitted)
                        protocol:             ~

                        # True to use dbname as service name instead of SID for Oracle
                        service:              ~

                        # The session mode to use for the oci8 driver
                        sessionMode:          ~

                        # True to use a pooled server with the oci8 driver
                        pooled:               ~

                        # Configuring MultipleActiveResultSets for the pdo_sqlsrv driver
                        MultipleActiveResultSets:  ~

    orm:
        default_entity_manager:  ~
```

```
        auto_generate_proxy_classes:  false
        proxy_dir:            '%kernel.cache_dir%/doctrine/orm/Proxies'
        proxy_namespace:      Proxies
        # search for the "ResolveTargetEntityListener" class for an article about this
        resolve_target_entities: []
        entity_managers:
            # A collection of different named entity managers (e.g. some_em, another_em)
            some_em:
                query_cache_driver:
                    type:                 array # Required
                    host:                 ~
                    port:                 ~
                    instance_class:       ~
                    class:                ~
                    namespace:            ~
                metadata_cache_driver:
                    type:                 array # Required
                    host:                 ~
                    port:                 ~
                    instance_class:       ~
                    class:                ~
                    namespace:            ~
                result_cache_driver:
                    type:                 array # Required
                    host:                 ~
                    port:                 ~
                    instance_class:       ~
                    class:                ~
                    namespace:            ~
                connection:           ~
                class_metadata_factory_name:  Doctrine\ORM\Mapping\ClassMetadataFactory
                default_repository_class:  Doctrine\ORM\EntityRepository
                auto_mapping:         false
                hydrators:

                    # An array of hydrator names
                    hydrator_name:                []
                mappings:
                    # An array of mappings, which may be a bundle name or something else
                    mapping_name:
                        mapping:              true
                        type:                 ~
                        dir:                  ~
                        alias:                ~
                        prefix:               ~
                        is_bundle:            ~
                dql:
                    # a collection of string functions
                    string_functions:
                        # example
                        # test_string: App\DQL\StringFunction

                    # a collection of numeric functions
                    numeric_functions:
                        # example
                        # test_numeric: App\DQL\NumericFunction

                    # a collection of datetime functions
                    datetime_functions:
                        # example
                        # test_datetime: App\DQL\DatetimeFunction

                # Register SQL Filters in the entity manager
                filters:
                    # An array of filters
                    some_filter:
                        class:                ~ # Required
                        enabled:              false
```

Doctrine DBAL Configuration

DoctrineBundle supports all parameters that default Doctrine drivers accept, converted to the XML or YAML naming standards that Symfony enforces. See the Doctrine *DBAL documentation*[1] for more information. The following block shows all possible configuration keys:

Listing 2-2

```yaml
doctrine:
    dbal:
        dbname:               database
        host:                 localhost
        port:                 1234
        user:                 user
        password:             secret
        driver:               pdo_mysql
        # if the url option is specified, it will override the above config
        url:                  mysql://db_user:db_password@127.0.0.1:3306/db_name
        # the DBAL driverClass option
        driver_class:         App\DBAL\MyDatabaseDriver
        # the DBAL driverOptions option
        options:
            foo: bar
        path:                 '%kernel.project_dir%/var/data/data.sqlite'
        memory:               true
        unix_socket:          /tmp/mysql.sock
        # the DBAL wrapperClass option
        wrapper_class:        App\DBAL\MyConnectionWrapper
        charset:              UTF8
        logging:              '%kernel.debug%'
        platform_service:     App\DBAL\MyDatabasePlatformService
        server_version:       '5.6'
        mapping_types:
            enum: string
        types:
            custom: App\DBAL\MyCustomType
```

The `server_version` option was added in Doctrine DBAL 2.5, which is used by DoctrineBundle 1.3. The value of this option should match your database server version (use `postgres -V` or `psql -V` command to find your PostgreSQL version and `mysql -V` to get your MySQL version).

If you are running a MariaDB database, you must prefix the `server_version` value with `mariadb-` (e.g. `server_version: mariadb-10.2.12`).

Always wrap the server version number with quotes to parse it as a string instead of a float number. Otherwise, the floating-point representation issues can make your version be considered a different number (e.g. `5.6` will be rounded as `5.599999999999999644728632119949907064437866210937 5`).

If you don't define this option and you haven't created your database yet, you may get `PDOException` errors because Doctrine will try to guess the database server version automatically and none is available.

If you want to configure multiple connections in YAML, put them under the `connections` key and give them a unique name:

Listing 2-3

```yaml
doctrine:
    dbal:
        default_connection:       default
        connections:
            default:
                dbname:           Symfony
                user:             root
```

1. http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html

```
                password:         null
                host:             localhost
                server_version:   5.6
            customer:
                dbname:           customer
                user:             root
                password:         null
                host:             localhost
                server_version:   5.7
```

The `database_connection` service always refers to the *default* connection, which is the first one defined or the one configured via the `default_connection` parameter.

Each connection is also accessible via the `doctrine.dbal.[name]_connection` service where `[name]` is the name of the connection.

Doctrine ORM Configuration

This following configuration example shows all the configuration defaults that the ORM resolves to:

Listing 2-4

```yaml
doctrine:
    orm:
        auto_mapping: true
        # the standard distribution overrides this to be true in debug, false otherwise
        auto_generate_proxy_classes: false
        proxy_namespace: Proxies
        proxy_dir: '%kernel.cache_dir%/doctrine/orm/Proxies'
        default_entity_manager: default
        metadata_cache_driver: array
        query_cache_driver: array
        result_cache_driver: array
```

There are lots of other configuration options that you can use to overwrite certain classes, but those are for very advanced use-cases only.

Shortened Configuration Syntax

When you are only using one entity manager, all config options available can be placed directly under `doctrine.orm` config level.

Listing 2-5

```yaml
doctrine:
    orm:
        # ...
        query_cache_driver:
           # ...
        metadata_cache_driver:
            # ...
        result_cache_driver:
            # ...
        connection: ~
        class_metadata_factory_name:  Doctrine\ORM\Mapping\ClassMetadataFactory
        default_repository_class:  Doctrine\ORM\EntityRepository
        auto_mapping: false
        hydrators:
            # ...
        mappings:
            # ...
        dql:
            # ...
        filters:
            # ...
```

This shortened version is commonly used in other documentation sections. Keep in mind that you can't use both syntaxes at the same time.

Caching Drivers

The built-in types of caching drivers are: `array`, `apc`, `apcu`, `memcache`, `memcached`, `redis`, `wincache`, `zenddata` and `xcache`. There is a special type called `service` which lets you define the ID of your own caching service.

The following example shows an overview of the caching configurations:

Listing 2-6

```yaml
doctrine:
    orm:
        auto_mapping: true
        # each caching driver type defines its own config options
        metadata_cache_driver: apc
        result_cache_driver:
            type: memcache
            host: localhost
            port: 11211
            instance_class: Memcache
        # the 'service' type requires to define the 'id' option too
        query_cache_driver:
            type: service
            id: App\ORM\MyCacheService
```

Mapping Configuration

Explicit definition of all the mapped entities is the only necessary configuration for the ORM and there are several configuration options that you can control. The following configuration options exist for a mapping:

type

One of `annotation` (the default value), `xml`, `yml`, `php` or `staticphp`. This specifies which type of metadata type your mapping uses.

dir

Absolute path to the mapping or entity files (depending on the driver).

prefix

A common namespace prefix that all entities of this mapping share. This prefix should never conflict with prefixes of other defined mappings otherwise some of your entities cannot be found by Doctrine.

alias

Doctrine offers a way to alias entity namespaces to simpler, shorter names to be used in DQL queries or for Repository access.

is_bundle

This option is `false` by default and it's considered a legacy option. It was only useful in previous Symfony versions, when it was recommended to use bundles to organize the application code.

Custom Mapping Entities in a Bundle

Doctrine's `auto_mapping` feature loads annotation configuration from the `Entity/` directory of each bundle *and* looks for other formats (e.g. YAML, XML) in the `Resources/config/doctrine` directory.

If you store metadata somewhere else in your bundle, you can define your own mappings, where you tell Doctrine exactly *where* to look, along with some other configurations.

If you're using the `auto_mapping` configuration, you just need to overwrite the configurations you want. In this case it's important that the key of the mapping configurations corresponds to the name of the bundle.

For example, suppose you decide to store your `XML` configuration for `AppBundle` entities in the `@AppBundle/SomeResources/config/doctrine` directory instead:

Listing 2-7

```yaml
doctrine:
    # ...
    orm:
        # ...
        auto_mapping: true
        mappings:
            # ...
            AppBundle:
                type: xml
                dir: SomeResources/config/doctrine
```

Mapping Entities Outside of a Bundle

For example, the following looks for entity classes in the `Entity` namespace in the `src/Entity` directory and gives them an `App` alias (so you can say things like `App:Post`):

Listing 2-8

```yaml
doctrine:
        # ...
        orm:
            # ...
            mappings:
                # ...
                SomeEntityNamespace:
                    type: annotation
                    dir: '%kernel.project_dir%/src/Entity'
                    is_bundle: false
                    prefix: App\Entity
                    alias: App
```

Detecting a Mapping Configuration Format

If the `type` on the bundle configuration isn't set, the DoctrineBundle will try to detect the correct mapping configuration format for the bundle.

DoctrineBundle will look for files matching `*.orm.[FORMAT]` (e.g. `Post.orm.yaml`) in the configured `dir` of your mapping (if you're mapping a bundle, then `dir` is relative to the bundle's directory).

The bundle looks for (in this order) XML, YAML and PHP files. Using the `auto_mapping` feature, every bundle can have only one configuration format. The bundle will stop as soon as it locates one.

If it wasn't possible to determine a configuration format for a bundle, the DoctrineBundle will check if there is an `Entity` folder in the bundle's root directory. If the folder exist, Doctrine will fall back to using an annotation driver.

Default Value of Dir

If `dir` is not specified, then its default value depends on which configuration driver is being used. For drivers that rely on the PHP files (annotation, staticphp) it will be `[Bundle]/Entity`. For drivers that are using configuration files (XML, YAML, ...) it will be `[Bundle]/Resources/config/doctrine`.

If the `dir` configuration is set and the `is_bundle` configuration is `true`, the DoctrineBundle will prefix the `dir` configuration with the path of the bundle.

Chapter 3

SecurityBundle Configuration ("security")

The security system is one of the most powerful parts of Symfony and can largely be controlled via its configuration.

Full Default Configuration

The following is the full default configuration for the security system. Each part will be explained in the next section.

Listing 3-1

```yaml
# config/packages/security.yaml
security:
    access_denied_url:    ~ # Example: /foo/error403

    # strategy can be: none, migrate, invalidate
    session_fixation_strategy:  migrate
    hide_user_not_found:  true
    always_authenticate_before_granting:  false
    erase_credentials:    true
    access_decision_manager:
        strategy:            affirmative # One of affirmative, consensus, unanimous
        allow_if_all_abstain:  false
        allow_if_equal_granted_denied:  true

    encoders:
        # Examples:
        App\Entity\User1: sha512
        App\Entity\User2:
            algorithm:           sha512
            encode_as_base64:    true
            iterations:          5000

        # PBKDF2 encoder
        # see the note about PBKDF2 below for details on security and speed
        App\Entity\User3:
            algorithm:            pbkdf2
            hash_algorithm:       sha512
            encode_as_base64:     true
            iterations:           1000
            key_length:           40

        # Example options/values for what a custom encoder might look like
```

```
        App\Entity\User4:
            id:                 App\Security\MyPasswordEncoder

        # BCrypt encoder
        # see the note about bcrypt below for details on specific dependencies
        App\Entity\User5:
            algorithm:          bcrypt
            cost:               13

        # Plaintext encoder
        # it does not do any encoding
        App\Entity\User6:
            algorithm:          plaintext
            ignore_case:        false

        # Argon2i encoder
        Acme\DemoBundle\Entity\User6:
            algorithm:          argon2i

    providers:          # Required
        # Examples:
        my_in_memory_provider:
            memory:
                users:
                    foo:
                        password:       foo
                        roles:          ROLE_USER
                    bar:
                        password:       bar
                        roles:          [ROLE_USER, ROLE_ADMIN]

        my_entity_provider:
            entity:
                class:          App\Entity\User7
                property:       username
                # name of a non-default entity manager
                manager_name:   ~

        my_ldap_provider:
            ldap:
                service:        ~
                base_dn:        ~
                search_dn:      ~
                search_password: ~
                default_roles:  'ROLE_USER'
                uid_key:        'sAMAccountName'
                filter:         '({uid_key}={username})'

        # Example custom provider
        my_some_custom_provider:
            id:                 ~

        # Chain some providers
        my_chain_provider:
            chain:
                providers:      [ my_in_memory_provider, my_entity_provider ]

    firewalls:          # Required
        # Examples:
        somename:
            pattern: .*
            # restrict the firewall to a specific host
            host: admin\.example\.com
            # restrict the firewall to specific HTTP methods
            methods: [GET, POST]
            request_matcher: some.service.id
            access_denied_url: /foo/error403
            access_denied_handler: some.service.id
            entry_point: some.service.id
            provider: some_key_from_above
            # manages where each firewall stores session information
```

```yaml
            # See "Firewall Context" below for more details
            context: context_key
            stateless: false
            x509:
                provider: some_key_from_above
            remote_user:
                provider: some_key_from_above
            http_basic:
                provider: some_key_from_above
            http_basic_ldap:
                provider:     some_key_from_above
                service:      ldap
                dn_string:    '{username}'
                query_string: ~
            http_digest:
                provider: some_key_from_above
            guard:
                # A key from the "providers" section of your security config, in case your user provider is
different than the firewall
                provider:             ~

                # A service id (of one of your authenticators) whose start() method should be called when an
anonymous user hits a page that requires authentication
                entry_point:          null

                # An array of service ids for all of your "authenticators"
                authenticators:       []
            form_login:
                # submit the login form here
                check_path: /login_check

                # the user is redirected here when they need to log in
                login_path: /login

                # if true, forward the user to the login form instead of redirecting
                use_forward: false

                # login success redirecting options (read further below)
                always_use_default_target_path: false
                default_target_path:            /
                target_path_parameter:          _target_path
                use_referer:                    false

                # login failure redirecting options (read further below)
                failure_path:    /foo
                failure_forward: false
                failure_path_parameter: _failure_path
                failure_handler: some.service.id
                success_handler: some.service.id

                # field names for the username and password fields
                username_parameter: _username
                password_parameter: _password

                # csrf token options
                csrf_parameter:       _csrf_token
                csrf_token_id:        authenticate
                csrf_token_generator: my.csrf_token_generator.id

                # by default, the login form *must* be a POST, not a GET
                post_only:      true
                remember_me:    false

                # by default, a session must exist before submitting an authentication request
                # if false, then Request::hasPreviousSession is not called during authentication
                require_previous_session: true

            form_login_ldap:
                # submit the login form here
                check_path: /login_check

```

```yaml
                # the user is redirected here when they need to log in
                login_path: /login

                # if true, forward the user to the login form instead of redirecting
                use_forward: false

                # login success redirecting options (read further below)
                always_use_default_target_path: false
                default_target_path:            /
                target_path_parameter:          _target_path
                use_referer:                    false

                # login failure redirecting options (read further below)
                failure_path:    /foo
                failure_forward: false
                failure_path_parameter: _failure_path
                failure_handler: some.service.id
                success_handler: some.service.id

                # field names for the username and password fields
                username_parameter: _username
                password_parameter: _password

                # csrf token options
                csrf_parameter:       _csrf_token
                csrf_token_id:        authenticate
                csrf_token_generator: my.csrf_token_generator.id

                # by default, the login form *must* be a POST, not a GET
                post_only:      true
                remember_me:    false

                # by default, a session must exist before submitting an authentication request
                # if false, then Request::hasPreviousSession is not called during authentication
                require_previous_session: true

                service:      ~
                dn_string:    '{username}'
                query_string: ~

            remember_me:
                token_provider: name
                secret: "%secret%"
                name: NameOfTheCookie
                lifetime: 3600 # in seconds
                path: /foo
                domain: somedomain.foo
                secure: false
                httponly: true
                always_remember_me: false
                remember_me_parameter: _remember_me
            logout:
                path:   /logout
                target: /
                invalidate_session: false
                delete_cookies:
                    a: { path: null, domain: null }
                    b: { path: null, domain: null }
                handlers: [some.service.id, another.service.id]
                success_handler: some.service.id
            anonymous: ~

        # Default values and options for any firewall
        some_firewall_listener:
            pattern:              ~
            security:             true
            request_matcher:      ~
            access_denied_url:    ~
            access_denied_handler:  ~
            entry_point:          ~
            provider:             ~
```

```
            stateless:            false
            context:              ~
            logout:
                csrf_parameter:       _csrf_token
                csrf_token_generator: ~
                csrf_token_id:        logout
                path:                 /logout
                target:               /
                success_handler:      ~
                invalidate_session:   true
                delete_cookies:

                    # Prototype
                    name:
                        path:                 ~
                        domain:               ~
                handlers:             []
            anonymous:
                secret:               "%secret%"
            switch_user:
                provider:             ~
                parameter:            _switch_user
                role:                 ROLE_ALLOWED_TO_SWITCH

    access_control:
        requires_channel:     ~

        # use the urldecoded format
        path:                 ~ # Example: ^/path to resource/
        host:                 ~
        ips:                  []
        methods:              []
        roles:                []
    role_hierarchy:
        ROLE_ADMIN:     [ROLE_ORGANIZER, ROLE_USER]
        ROLE_SUPERADMIN: [ROLE_ADMIN]
```

Form Login Configuration

When using the `form_login` authentication listener beneath a firewall, there are several common options for configuring the "form login" experience.

For even more details, see *How to Customize Redirect After Form Login*.

The Login Form and Process

login_path

type: `string` **default**: `/login`

This is the route or path that the user will be redirected to (unless `use_forward` is set to `true`) when they try to access a protected resource but isn't fully authenticated.

This path **must** be accessible by a normal, un-authenticated user, else you may create a redirect loop. For details, see "Avoid Common Pitfalls".

check_path

type: `string` **default**: `/login_check`

This is the route or path that your login form must submit to. The firewall will intercept any requests (`POST` requests only, by default) to this URL and process the submitted login credentials.

Be sure that this URL is covered by your main firewall (i.e. don't create a separate firewall just for `check_path` URL).

use_forward

type: `boolean` **default**: `false`

If you'd like the user to be forwarded to the login form instead of being redirected, set this option to `true`.

username_parameter

type: `string` **default**: `_username`

This is the field name that you should give to the username field of your login form. When you submit the form to `check_path`, the security system will look for a POST parameter with this name.

password_parameter

type: `string` **default**: `_password`

This is the field name that you should give to the password field of your login form. When you submit the form to `check_path`, the security system will look for a POST parameter with this name.

post_only

type: `boolean` **default**: `true`

By default, you must submit your login form to the `check_path` URL as a POST request. By setting this option to `false`, you can send a GET request to the `check_path` URL.

Redirecting after Login

always_use_default_target_path

type: `boolean` **default**: `false`

If `true`, users are always redirected to the default target path regardless of the previous URL that was stored in the session.

default_target_path

type: `string` **default**: `/`

The page users are redirected to when there is no previous page stored in the session (for example, when the users browse the login page directly).

target_path_parameter

type: `string` **default**: `_target_path`

When using a login form, if you include an HTML element to set the target path, this option lets you change the name of the HTML element itself.

use_referer

type: `boolean` **default**: `false`

If `true`, the user is redirected to the value stored in the `HTTP_REFERER` header when no previous URL was stored in the session. If the referrer URL is the same as the one generated with the `login_path` route, the user is redirected to the `default_target_path` to avoid a redirection loop.

For historical reasons, and to match the misspelling of the HTTP standard, the option is called `use_referer` instead of `use_referrer`.

Logout Configuration

invalidate_session

type: `boolean` **default**: `true`

By default, when users log out from any firewall, their sessions are invalidated. This means that logging out from one firewall automatically logs them out from all the other firewalls.

The `invalidate_session` option allows to redefine this behavior. Set this option to `false` in every firewall and the user will only be logged out from the current firewall and not the other ones.

LDAP functionality

There are several options for connecting against an LDAP server, using the `form_login_ldap` and `http_basic_ldap` authentication providers or the `ldap` user provider.

For even more details, see *Authenticating against an LDAP server*.

Authentication

You can authenticate to an LDAP server using the LDAP variants of the `form_login` and `http_basic` authentication providers. Simply use `form_login_ldap` and `http_basic_ldap`, which will attempt to `bind` against a LDAP server instead of using password comparison.

Both authentication providers have the same arguments as their normal counterparts, with the addition of two configuration keys:

service

type: `string` **default**: `ldap`

This is the name of your configured LDAP client.

dn_string

type: `string` **default**: `{username}`

This is the string which will be used as the bind DN. The `{username}` placeholder will be replaced with the user-provided value (their login). Depending on your LDAP server's configuration, you may need to override this value.

query_string

type: `string` **default**: `null`

This is the string which will be used to query for the DN. The `{username}` placeholder will be replaced with the user-provided value (their login). Depending on your LDAP server's configuration, you will need to override this value. This setting is only necessary if the user's DN cannot be derived statically using the `dn_string` config option.

User provider

Users will still be fetched from the configured user provider. If you wish to fetch your users from a LDAP server, you will need to use the `ldap` user provider, in addition to one of the two authentication providers (`form_login_ldap` or `http_basic_ldap`).

Listing 3-2
```yaml
# config/packages/security.yaml
security:
    # ...

    providers:
        my_ldap_users:
            ldap:
                service: ldap
                base_dn: 'dc=symfony,dc=com'
                search_dn: '%ldap.search_dn%'
                search_password: '%ldap.search_password%'
                default_roles: ''
                uid_key: 'uid'
                filter: '(&({uid_key}={username})(objectclass=person)(ou=Users))'
```

Using the PBKDF2 Encoder: Security and Speed

The *PBKDF2*[1] encoder provides a high level of Cryptographic security, as recommended by the National Institute of Standards and Technology (NIST).

You can see an example of the `pbkdf2` encoder in the YAML block on this page.

But using PBKDF2 also warrants a warning: using it (with a high number of iterations) slows down the process. Thus, PBKDF2 should be used with caution and care.

A good configuration lies around at least 1000 iterations and sha512 for the hash algorithm.

Using the BCrypt Password Encoder

Listing 3-3
```yaml
# config/packages/security.yaml
security:
    # ...

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm: bcrypt
            cost:      15
```

The `cost` can be in the range of `4-31` and determines how long a password will be encoded. Each increment of `cost` *doubles* the time it takes to encode a password.

If you don't provide the `cost` option, the default cost of `13` is used.

1. https://en.wikipedia.org/wiki/PBKDF2

You can change the cost at any time — even if you already have some passwords encoded using a different cost. New passwords will be encoded using the new cost, while the already encoded ones will be validated using a cost that was used back when they were encoded.

A salt for each new password is generated automatically and need not be persisted. Since an encoded password contains the salt used to encode it, persisting the encoded password alone is enough.

BCrypt encoded passwords are `60` characters long, so make sure to allocate enough space for them to be persisted.

A simple technique to make tests much faster when using BCrypt is to set the cost to `4`, which is the minimum value allowed, in the `test` environment configuration.

Using the Argon2i Password Encoder

To use this encoder, you either need to use PHP version 7.2 or install the *libsodium*[2] extension.

Listing 3-4
```yaml
# app/config/security.yml
security:
    # ...

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm: argon2i
```

A salt for each new password is generated automatically and need not be persisted. Since an encoded password contains the salt used to encode it, persisting the encoded password alone is enough.

Argon2i encoded passwords are `96` characters long, but due to the hashing requirements saved in the resulting hash this may change in the future.

Firewall Context

Most applications will only need one firewall. But if your application *does* use multiple firewalls, you'll notice that if you're authenticated in one firewall, you're not automatically authenticated in another. In other words, the systems don't share a common "context": each firewall acts like a separate security system.

However, each firewall has an optional `context` key (which defaults to the name of the firewall), which is used when storing and retrieving security data to and from the session. If this key were set to the same value across multiple firewalls, the "context" could actually be shared:

Listing 3-5
```yaml
# config/packages/security.yaml
security:
```

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

```
    # ...

    firewalls:
        somename:
            # ...
            context: my_context
        othername:
            # ...
            context: my_context
```

The firewall context key is stored in session, so every firewall using it must set its `stateless` option to `false`. Otherwise, the context is ignored and you won't be able to authenticate on multiple firewalls at the same time.

Chapter 4

SwiftmailerBundle Configuration ("swiftmailer")

This reference document is a work in progress. It should be accurate, but all options are not yet fully covered. For a full list of the default configuration options, see Full Default Configuration

The `swiftmailer` key configures Symfony's integration with Swift Mailer, which is responsible for creating and delivering email messages.

The following section lists all options that are available to configure a mailer. It is also possible to configure several mailers (see Using Multiple Mailers).

Configuration

- url
- transport
- username
- password
- command
- host
- port
- timeout
- source_ip
- local_domain
- encryption
- auth_mode
- **spool**
    - type
    - path

- sender_address
- **antiflood**
    - threshold
    - sleep
- delivery_addresses
- delivery_whitelist
- disable_delivery
- logging

url

type: `string`

The entire SwiftMailer configuration using a DSN-like URL format.

Example: `smtp://user:pass@host:port/?timeout=60&encryption=ssl&auth_mode=login&...`

transport

type: `string` **default**: `smtp`

The exact transport method to use to deliver emails. Valid values are:

- smtp
- gmail (see Using Gmail to Send Emails)
- mail (deprecated in SwiftMailer since version 5.4.5)
- sendmail
- null (same as setting disable_delivery to `true`)

username

type: `string`

The username when using `smtp` as the transport.

password

type: `string`

The password when using `smtp` as the transport.

command

type: `string` **default**: `/usr/sbin/sendmail -bs`

Command to be executed by `sendmail` transport.

host

type: `string` **default**: `localhost`

The host to connect to when using `smtp` as the transport.

port

type: `string` **default**: 25 or 465 (depending on encryption)

The port when using `smtp` as the transport. This defaults to 465 if encryption is `ssl` and 25 otherwise.

timeout

type: `integer`

The timeout in seconds when using `smtp` as the transport.

source_ip

type: `string`

The source IP address when using `smtp` as the transport.

local_domain

type: `string`

New in version 2.4.0: The `local_domain` option was introduced in SwiftMailerBundle 2.4.0.

The domain name to use in `HELO` command.

encryption

type: `string`

The encryption mode to use when using `smtp` as the transport. Valid values are `tls`, `ssl`, or `null` (indicating no encryption).

auth_mode

type: `string`

The authentication mode to use when using `smtp` as the transport. Valid values are `plain`, `login`, `cram-md5`, or `null`.

spool

For details on email spooling, see *How to Spool Emails*.

type

type: `string` **default**: `file`

The method used to store spooled messages. Valid values are `memory` and `file`. A custom spool should be possible by creating a service called `swiftmailer.spool.myspool` and setting this value to `myspool`.

path

type: `string` **default**: `%kernel.cache_dir%/swiftmailer/spool`

When using the `file` spool, this is the path where the spooled messages will be stored.

sender_address

type: `string`

If set, all messages will be delivered with this address as the "return path" address, which is where bounced messages should go. This is handled internally by Swift Mailer's `Swift_Plugins_ImpersonatePlugin` class.

antiflood

threshold

type: `integer` **default**: `99`

Used with `Swift_Plugins_AntiFloodPlugin`. This is the number of emails to send before restarting the transport.

sleep

type: `integer` **default**: `0`

Used with `Swift_Plugins_AntiFloodPlugin`. This is the number of seconds to sleep for during a transport restart.

delivery_addresses

type: `array`

In previous versions, this option was called `delivery_address`.

If set, all email messages will be sent to these addresses instead of being sent to their actual recipients. This is often useful when developing. For example, by setting this in the `config/packages/dev/swiftmailer.yaml` file, you can guarantee that all emails sent during development go to one or more some specific accounts.

This uses `Swift_Plugins_RedirectingPlugin`. Original recipients are available on the `X-Swift-To`, `X-Swift-Cc` and `X-Swift-Bcc` headers.

delivery_whitelist

type: `array`

Used in combination with `delivery_address` or `delivery_addresses`. If set, emails matching any of these patterns will be delivered like normal, as well as being sent to `delivery_address` or `delivery_addresses`. For details, see the How to Work with Emails during Development article.

disable_delivery

type: `boolean` **default**: `false`

If true, the `transport` will automatically be set to `null` and no emails will actually be delivered.

logging

type: `boolean` **default**: `%kernel.debug%`

If true, Symfony's data collector will be activated for Swift Mailer and the information will be available in the profiler.

The following options can be set via environment variables using the `%env()%` syntax: `url`, `transport`, `username`, `password`, `host`, `port`, `timeout`, `source_ip`, `local_domain`, `encryption`, `auth_mode`. For details, see the *How to Set external Parameters in the Service Container* article.

Full Default Configuration

Listing 4-1

```yaml
swiftmailer:
    transport:            smtp
    username:             ~
    password:             ~
    host:                 localhost
    port:                 false
    encryption:           ~
    auth_mode:            ~
    spool:
        type:                 file
        path:                 '%kernel.cache_dir%/swiftmailer/spool'
    sender_address:       ~
    antiflood:
        threshold:            99
        sleep:                0
    delivery_addresses:   []
    disable_delivery:     ~
    logging:              '%kernel.debug%'
```

Using Multiple Mailers

You can configure multiple mailers by grouping them under the `mailers` key (the default mailer is identified by the `default_mailer` option):

Listing 4-2

```yaml
swiftmailer:
    default_mailer: second_mailer
    mailers:
        first_mailer:
            # ...
        second_mailer:
            # ...
```

Each mailer is registered automatically as a service with these IDs:

Listing 4-3

```php
// ...

// returns the first mailer
$container->get('swiftmailer.mailer.first_mailer');

// also returns the second mailer since it is the default mailer
$container->get('swiftmailer.mailer');

// returns the second mailer
$container->get('swiftmailer.mailer.second_mailer');
```

When configuring multiple mailers, options must be placed under the appropriate mailer key of the configuration instead of directly under the `swiftmailer` key.

When using autowiring only the default mailer is injected when type-hinting some argument with the `\Swift_Mailer` class. If you need to inject a different mailer in some service, use any of these alternatives based on the service binding feature:

Listing 4-4

```yaml
# config/services.yaml
services:
    _defaults:
        bind:
            # this injects the second mailer when type-hinting constructor arguments with \Swift_Mailer
            \Swift_Mailer: '@swiftmailer.mailer.second_mailer'
            # this injects the second mailer when a service constructor argument is called $specialMailer
            $specialMailer: '@swiftmailer.mailer.second_mailer'

    App\Some\Service:
        # this injects the second mailer only for this argument of this service
        $differentMailer: '@swiftmailer.mailer.second_mailer'

    # ...
```

Chapter 5
TwigBundle Configuration ("twig")

Listing 5-1

```yaml
# config/packages/twig.yaml
twig:
    exception_controller:  twig.controller.exception:showAction

    form_themes:

        # Default:
        - form_div_layout.html.twig

        # Bootstrap:
        - bootstrap_3_layout.html.twig
        - bootstrap_3_horizontal_layout.html.twig
        - bootstrap_4_layout.html.twig
        - bootstrap_4_horizontal_layout.html.twig

        # Foundation
        - foundation_5_layout.html.twig

        # Example:
        - form.html.twig

    globals:

        # Examples:
        foo:                '@bar'
        pi:                 3.14

        # Example options, but the easiest use is as seen above
        some_variable_name:
            # a service id that should be the value
            id:                 ~
            # set to service or leave blank
            type:               ~
            value:              ~
    autoescape:             ~

    # See http://twig.sensiolabs.org/doc/
recipes.html#using-the-template-name-to-set-the-default-escaping-strategy
    autoescape_service:        ~ # Example: 'my_service'
    autoescape_service_method: ~ # use in combination with autoescape_service option
    base_template_class:       ~ # Example: Twig_Template
    cache:                     '%kernel.cache_dir%/twig'
    charset:                   '%kernel.charset%'
```

```
44      debug:                '%kernel.debug%'
45      strict_variables:     ~
46      auto_reload:          ~
47      optimizations:        ~
48      default_path: '%kernel.project_dir%/templates'
49      paths:
50          '%kernel.project_dir%/vendor/acme/foo-bar/templates': foo_bar
51
52      date:
53          format: d.m.Y, H:i:s
54          interval_format: '%%d days'
55          timezone: Asia/Tokyo
56      number_format:
57          decimals: 2
58          decimal_point: ','
            thousands_separator: '.'
```

Configuration

auto_reload

type: `boolean` **default**: `'%kernel.debug%'`

If `true`, whenever a template is rendered, Symfony checks first if its source code has changed since it was compiled. If it has changed, the template is compiled again automatically.

autoescape

type: `boolean` or `string` **default**: `'name'`

If set to `false`, automatic escaping is disabled (you can still escape each content individually in the templates).

Setting this option to `false` is dangerous and it will make your application vulnerable to XSS exploits because most third-party bundles assume that auto-escaping is enabled and they don't escape contents themselves.

If set to a string, the template contents are escaped using the strategy with that name. Allowed values are `html`, `js`, `css`, `url`, `html_attr` and `name`. The default value is `name`. This strategy escapes contents according to the template name extension (e.g. it uses `html` for `*.html.twig` templates and `js` for `*.js.html` templates).

See autoescape_service and autoescape_service_method to define your own escaping strategy.

autoescape_service

type: `string` **default**: `null`

As of Twig 1.17, the escaping strategy applied by default to the template is determined during compilation time based on the filename of the template. This means for example that the contents of a `*.html.twig` template are escaped for HTML and the contents of `*.js.twig` are escaped for JavaScript.

This option allows to define the Symfony service which will be used to determine the default escaping applied to the template.

autoescape_service_method

type: `string` **default**: `null`

If `autoescape_service` option is defined, then this option defines the method called to determine the default escaping applied to the template.

base_template_class

type: `string` **default**: `'Twig_Template'`

Twig templates are compiled into PHP classes before using them to render contents. This option defines the base class from which all the template classes extend. Using a custom base template is discouraged because it will make your application harder to maintain.

cache

type: `string` **default**: `'%kernel.cache_dir%/twig'`

Before using the Twig templates to render some contents, they are compiled into regular PHP code. Compilation is a costly process, so the result is cached in the directory defined by this configuration option.

Set this option to `null` to disable Twig template compilation. However, this is not recommended; not even in the `dev` environment, because the `auto_reload` option ensures that cached templates which have changed get compiled again.

charset

type: `string` **default**: `'%kernel.charset%'`

The charset used by the template files. By default it's the same as the value of the `kernel.charset` container parameter, which is `UTF-8` by default in Symfony applications.

date

These options define the default values used by the `date` filter to format date and time values. They are useful to avoid passing the same arguments on every `date` filter call.

format

type: `string` **default**: `F j, Y H:i`

The format used by the `date` filter to display values when no specific format is passed as argument.

internal_format

type: `string` **default**: `%d days`

The format used by the `date` filter to display `DateInterval` instances when no specific format is passed as argument.

timezone

type: `string` **default**: (the value returned by `date_default_timezone_get()`)

The timezone used when formatting date values with the `date` filter and no specific timezone is passed as argument.

debug

type: `boolean` **default**: `'%kernel.debug%'`

If `true`, the compiled templates include a `__toString()` method that can be used to display their nodes.

exception_controller

type: `string` **default**: `twig.controller.exception:showAction`

This is the controller that is activated after an exception is thrown anywhere in your application. The default controller (*`ExceptionController`*[1]) is what's responsible for rendering specific templates under different error conditions (see *How to Customize Error Pages*). Modifying this option is advanced. If you need to customize an error page you should use the previous link. If you need to perform some behavior on an exception, you should add a listener to the `kernel.exception` event (see kernel.event_listener).

number_format

These options define the default values used by the `number_format` filter to format numeric values. They are useful to avoid passing the same arguments on every `number_format` filter call.

decimals

type: `integer` **default**: `0`

The number of decimals used to format numeric values when no specific number is passed as argument to the `number_format` filter.

decimal_point

type: `string` **default**: `.`

The character used to separate the decimals from the integer part of numeric values when no specific character is passed as argument to the `number_format` filter.

thousands_separator

type: `string` **default**: `,`

The character used to separate every group of thousands in numeric values when no specific character is passed as argument to the `number_format` filter.

optimizations

type: `int` **default**: `-1`

1. https://api.symfony.com/4.0/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html

Twig includes an extension called `optimizer` which is enabled by default in Symfony applications. This extension analyzes the templates to optimize them when being compiled. For example, if your template doesn't use the special `loop` variable inside a `for` tag, this extension removes the initialization of that unused variable.

By default, this option is `-1`, which means that all optimizations are turned on. Set it to `0` to disable all the optimizations. You can even enable or disable these optimizations selectively, as explained in the Twig documentation about *the optimizer extension*[2].

default_path

type: `string` **default**: `'%kernel.project_dir%/templates'`

The default directory where Symfony will look for Twig templates.

paths

type: `array` **default**: `null`

This option defines the directories where Symfony will look for Twig templates in addition to the default locations. Symfony looks for the templates in the following order:

1. The directories defined in this option;
2. The `Resources/views/` directories of the bundles used in the application;
3. The `src/Resources/views/` directory of the application;
4. The directory defined in the `default_path` option.

The values of the `paths` option are defined as `key: value` pairs where the `value` part can be `null`. For example:

Listing 5-2
```yaml
# config/packages/twig.yaml
twig:
    # ...
    paths:
        '%kernel.project_dir%/vendor/acme/foo-bar/templates': ~
```

The directories defined in the `paths` option have more priority than the default directories defined by Symfony. In the above example, if the template exists in the `acme/foo-bar/templates/` directory inside your application's `vendor/`, it will be used by Symfony.

If you provide a value for any path, Symfony will consider it the Twig namespace for that directory:

Listing 5-3
```yaml
# config/packages/twig.yaml
twig:
    # ...
    paths:
        '%kernel.project_dir%/vendor/acme/foo-bar/templates': 'foo_bar'
```

This option is useful to not mess with the default template directories defined by Symfony. Besides, it simplifies how you refer to those templates:

Listing 5-4
```
@foo_bar/template_name.html.twig
```

strict_variables

type: `boolean` **default**: `false`

2. http://twig.sensiolabs.org/doc/api.html#optimizer-extension

If set to `true`, Symfony shows an exception whenever a Twig variable, attribute or method doesn't exist. If set to `false` these errors are ignored and the non-existing values are replaced by `null`.

Chapter 6
MonologBundle Configuration ("monolog")

For a full list of handler types and related configuration options, see *Monolog Configuration*[1].

Full Default Configuration

Listing 6-1

```yaml
# config/packages/prod/monolog.yaml
monolog:
    handlers:

        # Examples:
        syslog:
            type:               stream
            path:               /var/log/symfony.log
            level:              ERROR
            bubble:             false
            formatter:          my_formatter
        main:
            type:               fingers_crossed
            action_level:       WARNING
            # By default, buffer_size is unlimited (0), which could
            # generate huge logs.
            buffer_size:        0
            handler:            custom
        console:
            type:               console
            verbosity_levels:
                VERBOSITY_NORMAL:       WARNING
                VERBOSITY_VERBOSE:      NOTICE
                VERBOSITY_VERY_VERBOSE: INFO
                VERBOSITY_DEBUG:        DEBUG
        custom:
            type:               service
            id:                 my_handler

        # Default options and values for some "my_custom_handler"
        # Note: many of these options are specific to the "type".
        # For example, the 'service' type doesn't use any options
        # except id and channels
```

1. https://github.com/symfony/monolog-bundle/blob/master/DependencyInjection/Configuration.php

```
        my_custom_handler:
            type:                 ~ # Required
            id:                   ~
            priority:             0
            level:                DEBUG
            bubble:               true
            path:                 '%kernel.logs_dir%/%kernel.environment%.log'
            ident:                false
            facility:             user
            max_files:            0
            action_level:         WARNING
            activation_strategy:  ~
            stop_buffering:       true
            buffer_size:          0
            handler:              ~
            members:              []
            channels:
                type:     ~
                elements: ~
            from_email:           ~
            to_email:             ~
            subject:              ~
            mailer:               ~
            email_prototype:
                id:                   ~ # Required (when the email_prototype is used)
                method:               ~
            formatter:            ~
    # Set to false to use seconds (instead of microseconds) in
    # the logs (gives a small performance boost).
    use_microseconds:             true
```

When the profiler is enabled, a handler is added to store the logs' messages in the profiler. The profiler uses the name "debug" so it is reserved and cannot be used in the configuration.

Chapter 7
WebProfilerBundle Configuration ("web_profiler")

The WebProfilerBundle provides detailed technical information about each request execution and displays it in both the web debug toolbar and the profiler.

The web debug toolbar is not available for responses of type `StreamedResponse`.

Configuration

- toolbar
- intercept_redirects
- excluded_ajax_paths

toolbar

type: `boolean` **default**: `false`

It enables and disables the toolbar entirely. Usually you set this to `true` in the `dev` and `test` environments and to `false` in the `prod` environment.

intercept_redirects

type: `boolean` **default**: `false`

If a redirect occurs during an HTTP response, the browser follows it automatically and you won't see the toolbar or the profiler of the original URL, only the redirected URL.

When setting this option to `true`, the browser *stops* before making any redirection and shows you the URL which is going to redirect to, its toolbar, and its profiler. Once you've inspected the toolbar/profiler data, you can click on the given link to perform the redirect.

excluded_ajax_paths

type: `string` **default**: `'^/(app(_[\\w]+)?\\.php/)?_wdt'`

When the toolbar logs Ajax requests, it matches their URLs against this regular expression. If the URL matches, the request is not displayed in the toolbar. This is useful when the application makes lots of Ajax requests or they are heavy and you want to exclude some of them.

Full Default Configuration

Listing 7-1
```yaml
# config/packages/dev/web_profiler.yaml
web_profiler:
    toolbar:              false
    intercept_redirects:  false
    excluded_ajax_paths:  ^/(app(_[\\w]+)?\\.php/)?_wdt
```

Chapter 8
DebugBundle Configuration ("debug")

The DebugBundle allows greater integration of the *VarDumper component* in the Symfony full-stack framework and can be configured under the `debug` key in your application configuration. When using XML, you must use the `http://symfony.com/schema/dic/debug` namespace.

The XSD schema is available at `http://symfony.com/schema/dic/debug/debug-1.0.xsd`.

Configuration

- max_items
- min_depth
- max_string_length
- dump_destination

max_items

type: `integer` **default**: `2500`

This is the maximum number of items to dump. Setting this option to `-1` disables the limit.

min_depth

type: `integer` **default**: `1`

Configures the minimum tree depth until which all items are guaranteed to be cloned. After this depth is reached, only `max_items` items will be cloned. The default value is `1`, which is consistent with older Symfony versions.

max_string_length

type: `integer` **default**: `-1`

This option configures the maximum string length before truncating the string. The default value (`-1`) means that strings are never truncated.

dump_destination

type: `string` **default**: `null`

Configures the output destination of the dumps.

By default, the dumps are shown in the toolbar. Since this is not always possible (e.g. when working on a JSON API), you can have an alternate output destination for dumps. Typically, you would set this to `php://stderr`:

Listing 8-1
```yaml
# config/packages/debug.yaml
debug:
    dump_destination: php://stderr
```

Chapter 9
Configuring in the Kernel

Some configuration can be done on the kernel class itself (located by default at `src/Kernel.php`). You can do this by overriding specific methods in the parent *`Kernel`*[1] class.

Configuration

- Charset
- Kernel Name
- Project Directory
- Cache Directory
- Log Directory

Charset

type: `string` **default**: `UTF-8`

This returns the charset that is used in the application. To change it, override the *`getCharset()`*[2] method and return another charset, for instance:

Listing 9-1
```php
// src/Kernel.php
use Symfony\Component\HttpKernel\Kernel as BaseKernel;
// ...

class Kernel extends BaseKernel
{
    public function getCharset()
    {
        return 'ISO-8859-1';
    }
}
```

1. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html
2. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html#method_getCharset

Kernel Name

type: `string` **default**: `src` (i.e. the directory name holding the kernel class)

To change this setting, override the *`getName()`*[3] method. Alternatively, move your kernel into a different directory. For example, if you moved the kernel into a `foo/` directory (instead of `src/`), the kernel name will be `foo`.

The name of the kernel isn't usually directly important - it's used in the generation of cache files - and you probably will only change it when *using applications with multiple kernels*.

Project Directory

type: `string` **default**: the directory of the project `composer.json`

This returns the root directory of your Symfony project. It's calculated as the directory where the main `composer.json` file is stored.

If for some reason the `composer.json` file is not stored at the root of your project, you can override the *`getProjectDir()`*[4] method to return the right project directory:

Listing 9-2
```php
// src/Kernel.php
use Symfony\Component\HttpKernel\Kernel as BaseKernel;
// ...

class Kernel extends BaseKernel
{
    // ...

    public function getProjectDir()
    {
        return realpath(__DIR__.'/../');
    }
}
```

Cache Directory

type: `string` **default**: `$this->rootDir/cache/$this->environment`

This returns the path to the cache directory. To change it, override the *`getCacheDir()`*[5] method. Read "Override the cache Directory" for more information.

Log Directory

type: `string` **default**: `$this->rootDir/log`

This returns the path to the log directory. To change it, override the *`getLogDir()`*[6] method. Read "Override the logs Directory" for more information.

3. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html#method_getName
4. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html#method_getProjectDir
5. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html#method_getCacheDir
6. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Kernel.html#method_getLogDir

Chapter 10
Form Types Reference

A form is composed of *fields*, each of which are built with the help of a field *type* (e.g. `TextType`, `ChoiceType`, etc). Symfony comes standard with a large list of field types that can be used in your application.

Supported Field Types

The following field types are natively available in Symfony:

Text Fields

- *TextType*
- *TextareaType*
- *EmailType*
- *IntegerType*
- *MoneyType*
- *NumberType*
- *PasswordType*
- *PercentType*
- *SearchType*
- *UrlType*
- *RangeType*
- *TelType*
- *ColorType*

Choice Fields

- *ChoiceType*
- *EntityType*
- *CountryType*
- *LanguageType*
- *LocaleType*
- *TimezoneType*

- *CurrencyType*

Date and Time Fields

- *DateType*
- *DateIntervalType*
- *DateTimeType*
- *TimeType*
- *BirthdayType*

Other Fields

- *CheckboxType*
- *FileType*
- *RadioType*

Field Groups

- *CollectionType*
- *RepeatedType*

Hidden Fields

- *HiddenType*

Buttons

- *ButtonType*
- *ResetType*
- *SubmitType*

Base Fields

- *FormType*

Chapter 11
TextType Field

The TextType field represents the most basic input text field.

Rendered as	`input text` field
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Overridden options	• compound
Parent type	*FormType*
Class	*TextType*[1]

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/TextType.html

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 11-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 11-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 11-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 11-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 11-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 11-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 11-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

2. http://diveintohtml5.info/forms.html

3. https://secure.php.net/manual/en/function.trim.php

Chapter 12
TextareaType Field

Renders a `textarea` HTML element.

Rendered as	`textarea` tag
Inherited options	• attr • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*TextareaType*[1]

If you prefer to use an **advanced WYSIWYG editor** instead of a plain textarea, consider using the IvoryCKEditorBundle community bundle. Read *its documentation*[2] to learn how to integrate it in your Symfony application.

Inherited Options

These options inherit from the *FormType*:

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/TextareaType.html
2. https://symfony.com/doc/current/bundles/IvoryCKEditorBundle/index.html

attr

type: `array` **default**: `array()`

If you want to add extra attributes to an HTML field representation you can use the `attr` option. It's an associative array with HTML attributes as keys. This can be useful when you need to set a custom class for some widget:

Listing 12-1
```
$builder->add('body', TextareaType::class, array(
    'attr' => array('class' => 'tinymce'),
));
```

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 12-2
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 12-3
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 12-4
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 12-5
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 12-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 12-7
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 12-8
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[4] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

3. http://diveintohtml5.info/forms.html

4. https://secure.php.net/manual/en/function.trim.php

Chapter 13
EmailType Field

The `EmailType` field is a text field that is rendered using the HTML5 `<input type="email" />` tag.

Rendered as	`input email` field (a text box)
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*EmailType*[1]

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/EmailType.html

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 13-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 13-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 13-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 13-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 13-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 13-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 13-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

2. http://diveintohtml5.info/forms.html
3. https://secure.php.net/manual/en/function.trim.php

Chapter 14
IntegerType Field

Renders an input "number" field. Basically, this is a text field that's good at handling data that's in an integer form. The input `number` field looks like a text box, except that - if the user's browser supports HTML5 - it will have some extra front-end functionality.

This field has different options on how to handle input values that aren't integers. By default, all non-integer values (e.g. 6.78) will round down (e.g. 6).

Rendered as	`input number` field
Options	• grouping • rounding_mode
Overridden options	• compound • scale
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*IntegerType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/IntegerType.html

Field Options

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

rounding_mode

type: `integer` **default**: `IntegerToLocalizedStringTransformer::ROUND_DOWN`

By default, if the user enters a non-integer number, it will be rounded down. There are several other rounding methods and each is a constant on the *`IntegerToLocalizedStringTransformer`*[2]:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` Round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up.

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

scale

type: `integer` **default**: `0`

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). This option inherits from *number* type and is overriden to `0` for `IntegerType`.

Inherited Options

These options inherit from the *FormType*:

2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.html

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 14-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 14-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 14-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 14-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 14-5
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 14-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 14-7
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 14-8
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. `http://diveintohtml5.info/forms.html`

Chapter 15
MoneyType Field

Renders an input text field and specializes in handling submitted "money" data.

This field type allows you to specify a currency, whose symbol is rendered next to the text field. There are also several other options for customizing how the input and output of the data is handled.

Rendered as	`input text` field
Options	• currency • divisor • grouping • scale
Overridden options	• compound
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*MoneyType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/MoneyType.html

Field Options

currency

type: `string` **default**: EUR

Specifies the currency that the money is being specified in. This determines the currency symbol that should be shown by the text box. Depending on the currency - the currency symbol may be shown before or after the input text field.

This can be any *3 letter ISO 4217 code*[2]. You can also set this to false to hide the currency symbol.

divisor

type: `integer` **default**: `1`

If, for some reason, you need to divide your starting value by a number before rendering it to the user, you can use the `divisor` option. For example:

Listing 15-1
```
use Symfony\Component\Form\Extension\Core\Type\MoneyType;
// ...

$builder->add('price', MoneyType::class, array(
    'divisor' => 100,
));
```

In this case, if the `price` field is set to `9900`, then the value `99` will actually be rendered to the user. When the user submits the value `99`, it will be multiplied by `100` and `9900` will ultimately be set back on your object.

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

scale

type: `integer` **default**: `2`

If, for some reason, you need some scale other than 2 decimal places, you can modify this value. You probably won't need to do this unless, for example, you want to round to the nearest dollar (set the scale to `0`).

Overridden Options

compound

type: `boolean` **default**: `false`

2. https://en.wikipedia.org/wiki/ISO_4217

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 15-2
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 15-3
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 15-4
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 15-5
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 15-6
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 15-7
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 15-8
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 15-9
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Form Variables

Variable	Type	Usage
money_pattern	`string`	The format to use to display the money, including the currency.

3. `http://diveintohtml5.info/forms.html`

Chapter 16
NumberType Field

Renders an input text field and specializes in handling number input. This type offers different options for the scale, rounding and grouping that you want to use for your number.

Rendered as	`input text` field
Options	• grouping • scale • rounding_mode
Overridden options	• compound
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*NumberType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/NumberType.html

Field Options

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

scale

New in version 2.7: The `scale` option was introduced in Symfony 2.7. Prior to Symfony 2.7, it was known as `precision`.

type: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `scale` is set to `2`, a submitted value of `20.123` will be rounded to, for example, `20.12` (depending on your rounding_mode).

rounding_mode

type: `integer` **default**: `NumberToLocalizedStringTransformer::ROUND_HALF_UP`

If a submitted number needs to be rounded (based on the scale option), you have several configurable options for that rounding. Each option is a constant on the *`NumberToLocalizedStringTransformer`*[2]:

- `NumberToLocalizedStringTransformer::ROUND_DOWN` Round towards zero.
- `NumberToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity.
- `NumberToLocalizedStringTransformer::ROUND_UP` Round away from zero.
- `NumberToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity.
- `NumberToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down.
- `NumberToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor.
- `NumberToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up.

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/DataTransformer/NumberToLocalizedStringTransformer.html

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 16-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 16-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 16-3
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 16-4
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 16-5
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 16-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 16-7
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 16-8
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
```

```
7       'label_format' => 'form.address.%name%',
8   ));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. http://diveintohtml5.info/forms.html

Chapter 17
PasswordType Field

The `PasswordType` field renders an input password text box.

Rendered as	`input password` field
Options	• always_empty
Overridden options	• trim
Inherited options	• disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*TextType*
Class	*PasswordType*[1]

Field Options

always_empty

type: `boolean` **default**: `true`

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/PasswordType.html

If set to true, the field will *always* render blank, even if the corresponding field has a value. When set to false, the password field will be rendered with the `value` attribute set to its true value only upon submission.

Put simply, if for some reason you want to render your password field *with* the password value already entered into the box, set this to false and submit the form.

Overridden Options

trim

type: `boolean` **default**: `false`

Unlike the rest of form types, the `PasswordType` doesn't apply the *trim*[2] function to the value submitted by the user. This ensures that the password is merged back onto the underlying object exactly as it was typed by the user.

Inherited Options

These options inherit from the *FormType*:

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 17-1
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

2. https://secure.php.net/manual/en/function.trim.php

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 17-2
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 17-3
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 17-4
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 17-5
```twig
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 17-6
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. `http://diveintohtml5.info/forms.html`

Chapter 18
PercentType Field

The `PercentType` renders an input text field and specializes in handling percentage data. If your percentage data is stored as a decimal (e.g. `.95`), you can use this field out-of-the-box. If you store your data as a number (e.g. `95`), you should set the `type` option to `integer`.

This field adds a percentage sign "`%`" after the input box.

Rendered as	`input text` field
Options	• scale • type
Overridden options	• compound
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*PercentType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/PercentType.html

Field Options

scale

type: `integer` **default**: `0`

By default, the input numbers are rounded. To allow for more decimal places, use this option.

type

type: `string` **default**: `fractional`

This controls how your data is stored on your object. For example, a percentage corresponding to "55%", might be stored as `.55` or `55` on your object. The two "types" handle these two cases:

- `fractional` If your data is stored as a decimal (e.g. `.55`), use this type. The data will be multiplied by `100` before being shown to the user (e.g. `55`). The submitted data will be divided by `100` on form submit so that the decimal value is stored (`.55`);
- `integer` If your data is stored as an integer (e.g. 55), then use this option. The raw value (`55`) is shown to the user and stored on your object. Note that this only works for integer values.

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 18-1

```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 18-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 18-3

```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 18-4

```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 18-5

```php
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 18-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 18-7
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 18-8
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

2. `http://diveintohtml5.info/forms.html`

Chapter 19
SearchType Field

This renders an `<input type="search" />` field, which is a text box with special functionality supported by some browsers.

Read about the input search field at *DiveIntoHTML5.info*[1]

Rendered as	`input search` field
Inherited options	• disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*SearchType*[2]

Inherited Options

These options inherit from the *FormType*:

disabled

type: `boolean` **default**: `false`

1. http://diveintohtml5.info/forms.html#type-search
2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/SearchType.html

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 19-1
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 19-2
```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults(array(
4         'error_mapping' => array(
5             'matchingCityAndZipCode' => 'city',
```

```
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 19-3
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 19-4
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 19-5
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 19-6
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *trim*[4] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

3. http://diveintohtml5.info/forms.html
4. https://secure.php.net/manual/en/function.trim.php

Chapter 20
UrlType Field

The `UrlType` field is a text field that prepends the submitted value with a given protocol (e.g. `http://`) if the submitted value doesn't already have a protocol.

Rendered as	`input url` field
Options	• default_protocol
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*UrlType*[1]

Field Options

default_protocol

type: `string` **default**: `http`

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/UrlType.html

If a value is submitted that doesn't begin with some protocol (e.g. `http://`, `ftp://`, etc), this protocol will be prepended to the string when the data is submitted to the form.

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 20-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 20-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 20-3
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 20-4
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 20-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 20-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 20-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

2. http://diveintohtml5.info/forms.html

3. https://secure.php.net/manual/en/function.trim.php

Chapter 21
RangeType Field

The `RangeType` field is a slider that is rendered using the HTML5 `<input type="range" />` tag.

Rendered as	`input range` field (slider in HTML5 supported browser)
Inherited options	• attr • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • required • trim
Parent type	*TextType*
Class	*RangeType*[1]

Basic Usage

Listing 21-1

```
use Symfony\Component\Form\Extension\Core\Type\RangeType;
// ...

$builder->add('name', RangeType::class, array(
    'attr' => array(
        'min' => 5,
        'max' => 50
    )
));
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/RangeType.html

Inherited Options

These options inherit from the *FormType*:

attr

type: `array` **default**: `array()`

If you want to add extra attributes to an HTML field representation you can use the `attr` option. It's an associative array with HTML attributes as keys. This can be useful when you need to set a custom class for some widget:

Listing 21-2
```
$builder->add('body', TextareaType::class, array(
    'attr' => array('class' => 'tinymce'),
));
```

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 21-3
```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, array(
5     'data' => 'abcdef',
6 ));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 21-4
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 21-5

```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 21-6

```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 21-7
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 21-8
```
{{ form_label(form.name, 'Your name', {
    'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

2. http://diveintohtml5.info/forms.html

If true, the whitespace of the submitted string value will be stripped via the *trim*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

3. https://secure.php.net/manual/en/function.trim.php

Chapter 22

TelType Field

The `TelType` field is a text field that is rendered using the HTML5 `<input type="tel">` tag. Following the recommended HTML5 behavior, the value of this type is not validated in any way, because formats for telephone numbers vary too much depending on each country.

Nevertheless it may be useful to use this type in web applications because some browsers (e.g. smartphone browsers) adapt the input keyboard to make it easier to input phone numbers.

Rendered as	`input tel` field (a text box)
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*TelType*[1]

Inherited Options

These options inherit from the *FormType*:

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/TelType.html

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 22-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 22-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 22-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 22-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 22-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 22-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 22-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

2. http://diveintohtml5.info/forms.html

3. https://secure.php.net/manual/en/function.trim.php

Chapter 23

ColorType Field

The `ColorType` field is a text field that is rendered using the HTML5 `<input type="color">` tag. Depending on each browser, the behavior of this form field can vary substantially. Some browsers display it as a simple text field, while others display a native color picker.

The value of the underlying `<input type="color">` field is always a 7-character string specifying an RGB color in lower case hexadecimal notation. That's why it's not possible to select semi-transparent colors with this element.

Rendered as	`input color` field (a text box)
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required • trim
Parent type	*TextType*
Class	*ColorType*[1]

Inherited Options

These options inherit from the *FormType*:

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/ColorType.html

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 23-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The default value is `''` (the empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 23-2
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 23-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 23-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 23-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 23-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 23-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[3] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

2. http://diveintohtml5.info/forms.html
3. https://secure.php.net/manual/en/function.trim.php

Chapter 24

ChoiceType Field (select drop-downs, radio buttons & checkboxes)

A multi-purpose field used to allow the user to "choose" one or more options. It can be rendered as a `select` tag, radio buttons, or checkboxes.

To use this field, you must specify *either* `choices` or `choice_loader` option.

Rendered as	can be various tags (see below)
Options	• choices • choice_attr • choice_label • choice_loader • choice_name • choice_translation_domain • choice_value • expanded • group_by • multiple • placeholder • preferred_choices
Overridden options	• compound • empty_data • error_bubbling • trim
Inherited options	• attr • by_reference • data

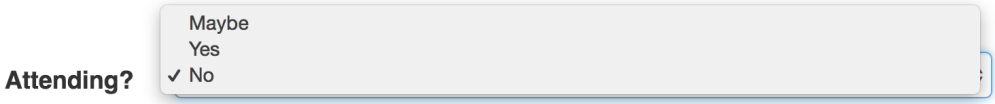
	• disabled • error_mapping • inherit_data • label • label_attr • label_format • mapped • required • translation_domain
Parent type	*FormType*
Class	*ChoiceType*[1]

Example Usage

The easiest way to use this field is to specify the choices directly via the `choices` option:

Listing 24-1
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('isAttending', ChoiceType::class, array(
    'choices'  => array(
        'Maybe' => null,
        'Yes' => true,
        'No' => false,
    ),
));
```

This will create a `select` drop-down like this:



If the user selects `No`, the form will return `false` for this field. Similarly, if the starting data for this field is `true`, then `Yes` will be auto-selected. In other words, the **value** of each item is the value you want to get/set in PHP code, while the **key** is what will be shown to the user.

Advanced Example (with Objects!)

This field has a *lot* of options and most control how the field is displayed. In this example, the underlying data is some `Category` object that has a `getName()` method:

Listing 24-2
```
use App\Entity\Category;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('category', ChoiceType::class, [
    'choices' => [
        new Category('Cat1'),
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/ChoiceType.html

```
        new Category('Cat2'),
        new Category('Cat3'),
        new Category('Cat4'),
    ],
    'choice_label' => function($category, $key, $index) {
        /** @var Category $category */
        return strtoupper($category->getName());
    },
    'choice_attr' => function($category, $key, $index) {
        return ['class' => 'category_'.strtolower($category->getName())];
    },

    'group_by' => function($category, $key, $index) {
        // randomly assign things into 2 groups
        return rand(0, 1) == 1 ? 'Group A' : 'Group B';
    },
    'preferred_choices' => function($category, $key, $index) {
        return $category->getName() == 'Cat2' || $category->getName() == 'Cat3';
    },
]);
```

You can also customize the choice_name and choice_value of each choice if you need further HTML customization.

Select Tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

Element Type	Expanded	Multiple
select tag	false	false
select tag (with `multiple` attribute)	false	true
radio buttons	true	false
checkboxes	true	true

Customizing each Option's Text (Label)

Normally, the array key of each item in the `choices` option is used as the text that's shown to the user. But that can be completely customized via the choice_label option. Check it out for more details.

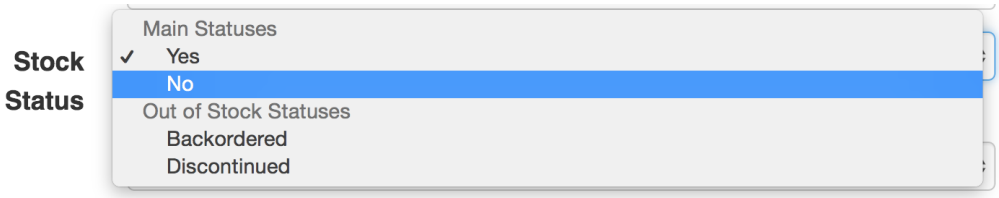
Grouping Options

You can easily "group" options in a select by passing a multi-dimensional choices array:

Listing 24-3

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('stockStatus', ChoiceType::class, array(
    'choices' => array(
        'Main Statuses' => array(
            'Yes' => 'stock_yes',
            'No' => 'stock_no',
        ),
        'Out of Stock Statuses' => array(
```

```
            'Backordered' => 'stock_backordered',
            'Discontinued' => 'stock_discontinued',
        ),
    ),
));
```



To get fancier, use the group_by option.

Field Options

choices

type: `array` **default**: `array()`

This is the most basic way to specify the choices that should be used by this field. The `choices` option is an array, where the array key is the item's label and the array value is the item's value:

Listing 24-4

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('inStock', ChoiceType::class, array(
    'choices' => array('In Stock' => true, 'Out of Stock' => false),
));
```

choice_attr

type: `array`, `callable` or `string` **default**: `array()`

Use this to add additional HTML attributes to each choice. This can be an array of attributes (if they are the same for each choice), a callable or a property path (just like choice_label).

If an array, the keys of the `choices` array must be used as keys:

Listing 24-5

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('attending', ChoiceType::class, array(
    'choices' => array(
        'Yes' => true,
        'No' => false,
        'Maybe' => null,
    ),
    'choice_attr' => function($val, $key, $index) {
        // adds a class like attending_yes, attending_no, etc
        return ['class' => 'attending_'.strtolower($key)];
    },
));
```

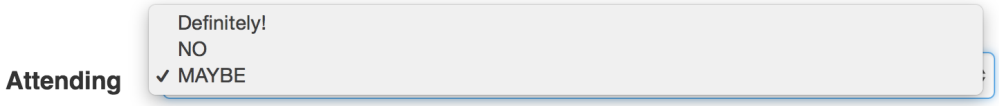
choice_label

type: `string`, `callable` or `false` **default**: `null`

Normally, the array key of each item in the `choices` option is used as the text that's shown to the user. The `choice_label` option allows you to take more control:

Listing 24-6
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('attending', ChoiceType::class, array(
    'choices' => array(
        'yes' => true,
        'no' => false,
        'maybe' => null,
    ),
    'choice_label' => function ($value, $key, $index) {
        if ($value == true) {
            return 'Definitely!';
        }
        return strtoupper($key);

        // or if you want to translate some key
        //return 'form.choice.'.$key;
    },
));
```

This method is called for *each* choice, passing you the choice `$value` and the `$key` from the choices array (`$index` is related to choice_value). This will give you:



If your choice values are objects, then `choice_label` can also be a property path. Imagine you have some `Status` class with a `getDisplayName()` method:

Listing 24-7
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('attending', ChoiceType::class, array(
    'choices' => array(
        new Status(Status::YES),
        new Status(Status::NO),
        new Status(Status::MAYBE),
    ),
    'choice_label' => 'displayName',
));
```

If set to `false`, all the tag labels will be discarded for radio or checkbox inputs. You can also return `false` from the callable to discard certain labels.

choice_loader

type: *`ChoiceLoaderInterface`*[2]

The `choice_loader` can be used to only partially load the choices in cases where a fully-loaded list is not necessary. This is only needed in advanced cases and would replace the `choices` option.

2. https://api.symfony.com/4.0/Symfony/Component/Form/ChoiceList/Loader/ChoiceLoaderInterface.html

You can use an instance of *CallbackChoiceLoader*[3] if you want to take advantage of lazy loading:

Listing 24-8
```
use Symfony\Component\Form\ChoiceList\Loader\CallbackChoiceLoader;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('constants', ChoiceType::class, array(
    'choice_loader' => new CallbackChoiceLoader(function() {
        return StaticClass::getConstants();
    }),
));
```

This will cause the call of `StaticClass::getConstants()` to not happen if the request is redirected and if there is no pre set or submitted data. Otherwise the choice options would need to be resolved thus triggering the callback.

choice_name

type: `callable` or `string` **default**: `null`

Controls the internal field name of the choice. You normally don't care about this, but in some advanced cases, you might. For example, this "name" becomes the index of the choice views in the template.

This can be a callable or a property path. See choice_label for similar usage. If `null` is used, an incrementing integer is used as the name.

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

choice_value

type: `callable` or `string` **default**: `null`

Returns the string "value" for each choice, which must be unique across all choices. This is used in the `value` attribute in HTML and submitted in the POST/PUT requests. You don't normally need to worry about this, but it might be handy when processing an API request (since you can configure the value that will be sent in the API request).

This can be a callable or a property path. If `null` is given, an incrementing integer is used as the value.

If you pass a callable, it will receive one argument: the choice itself. When using the *EntityType Field*, the argument will be the entity object for each choice or `null` in some cases, which you need to handle:

Listing 24-9
```
'choice_value' => function (MyOptionEntity $entity = null) {
    return $entity ? $entity->getId() : '';
},
```

expanded

type: `boolean` **default**: `false`

3. https://api.symfony.com/4.0/Symfony/Component/Form/ChoiceList/Loader/CallbackChoiceLoader.html

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

group_by

type: `array`, `callable` or `string` **default**: `null`

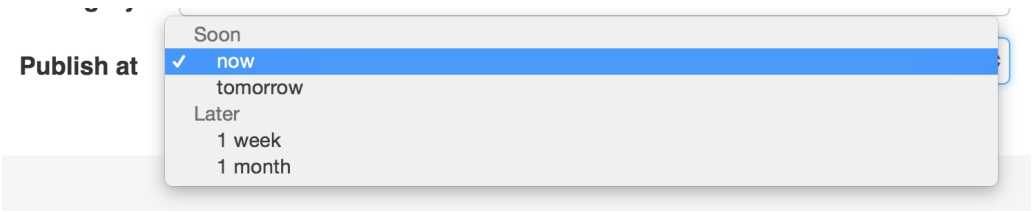
You can easily "group" options in a select simply by passing a multi-dimensional array to `choices`. See the Grouping Options section about that.

The `group_by` option is an alternative way to group choices, which gives you a bit more flexibility.

Take the following example:

Listing 24-10
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'group_by' => function($value, $key, $index) {
        if ($value <= new \DateTime('+3 days')) {
            return 'Soon';
        } else {
            return 'Later';
        }
    },
));
```

This groups the dates that are within 3 days into "Soon" and everything else into a "Later" group:



If you return `null`, the option won't be grouped. You can also pass a string "property path" that will be called to get the group. See the choice_label for details about using a property path.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 24-11
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 24-12
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 24-13
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

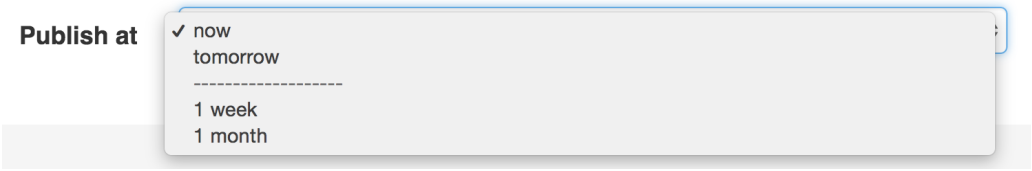
Listing 24-14
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 24-15
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
```

```
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 24-16
```
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

Overridden Options

compound

type: `boolean` **default**: same value as `expanded` option

This option specifies if a form is compound. The value is by default overridden by the value of the `expanded` option.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 24-17
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false`

Set that error on this field must be attached to the field instead of the parent field (the form in most cases).

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

Inherited Options

These options inherit from the *FormType*:

attr

type: `array` **default**: `array()`

If you want to add extra attributes to an HTML field representation you can use the `attr` option. It's an associative array with HTML attributes as keys. This can be useful when you need to set a custom class for some widget:

Listing 24-18
```
$builder->add('body', TextareaType::class, array(
    'attr' => array('class' => 'tinymce'),
));
```

by_reference

type: `boolean` **default**: `true`

In most cases, if you have an `author` field, then you expect `setAuthor()` to be called on the underlying object. In some cases, however, `setAuthor()` may *not* be called. Setting `by_reference` to `false` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 24-19
```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\FormType;
// ...

$builder = $this->createFormBuilder($article);
```

```
7  $builder
8      ->add('title', TextType::class)
9      ->add(
10         $builder->create('author', FormType::class, array('by_reference' => ?))
11             ->add('name', TextType::class)
12             ->add('email', EmailType::class)
13     )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 24-20
```
$article->setTitle('...');
$article->getAuthor()->setName('...');
$article->getAuthor()->setEmail('...');
```

Notice that `setAuthor()` is not called. The author is modified by reference.

If you set `by_reference` to false, submitting looks like this:

Listing 24-21
```
1  $article->setTitle('...');
2  $author = clone $article->getAuthor();
3  $author->setName('...');
4  $author->setEmail('...');
5  $article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *CollectionType* field where your underlying collection data is an object (like with Doctrine's `ArrayCollection`), then `by_reference` must be set to `false` if you need the adder and remover (e.g. `addAuthor()` and `removeAuthor()`) to be called.

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 24-22
```
1  use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2  // ...
3
4  $builder->add('token', HiddenType::class, array(
5      'data' => 'abcdef',
6  ));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 24-23
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 24-24
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 24-25
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 24-26
```twig
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 24-27
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

translation_domain

type: `string` **default**: `messages`

In case choice_translation_domain is set to `true` or `null`, this configures the exact translation domain that will be used for any labels or options that are rendered for this field

Field Variables

Variable	Type	Usage
multiple	`boolean`	The value of the multiple option.
expanded	`boolean`	The value of the expanded option.
preferred_choices	`array`	A nested array containing the `ChoiceView` objects of choices which should be presented to the user with priority.
choices	`array`	A nested array containing the `ChoiceView` objects of the remaining choices.
separator	`string`	The separator to use between choice groups.
placeholder	`mixed`	The empty value if not already in the list, otherwise `null`.
choice_translation_domain	`mixed`	`boolean`, `null` or `string` to determine if the value should be translated.
is_selected	`callable`	A callable which takes a `ChoiceView` and the selected value(s) and returns whether the choice is in the selected value(s).
placeholder_in_choices	`boolean`	Whether the empty value is in the choice list.

It's significantly faster to use the selectedchoice(selected_value) test instead when using Twig.

4. `http://diveintohtml5.info/forms.html`

Chapter 25
EntityType Field

A special `ChoiceType` field that's designed to load options from a Doctrine entity. For example, if you have a `Category` entity, you could use this field to display a `select` field of all, or some, of the `Category` objects from the database.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Options	• choice_label • class • em • query_builder
Overridden options	• choice_name • choice_value • choices • data_class
Inherited options	from the *ChoiceType*: • choice_attr • choice_translation_domain • expanded • group_by • multiple • placeholder • preferred_choices • translation_domain • trim from the *FormType*: • data • disabled

	• empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*ChoiceType*
Class	*EntityType*[1]

Basic Usage

The `entity` type has just one required option: the entity which should be listed inside the choice field:

Listing 25-1
```
use App\Entity\User;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

$builder->add('users', EntityType::class, array(
    // looks for choices from this entity
    'class' => User::class,

    // uses the User.username property as the visible option string
    'choice_label' => 'username',

    // used to render a select box, check boxes or radios
    // 'multiple' => true,
    // 'expanded' => true,
));
```

This will build a `select` drop-down containing *all* of the `User` objects in the database. To render radio buttons or checkboxes instead, change the multiple and expanded options.

Using a Custom Query for the Entities

If you want to create a custom query to use when fetching the entities (e.g. you only want to return some entities, or need to order them), use the query_builder option:

Listing 25-2
```
use App\Entity\User;
use Doctrine\ORM\EntityRepository;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

$builder->add('users', EntityType::class, array(
    'class' => User::class,
    'query_builder' => function (EntityRepository $er) {
        return $er->createQueryBuilder('u')
            ->orderBy('u.username', 'ASC');
    },
    'choice_label' => 'username',
));
```

1. https://api.symfony.com/4.0/Symfony/Bridge/Doctrine/Form/Type/EntityType.html

Using Choices

If you already have the exact collection of entities that you want to include in the choice element, just pass them via the `choices` key.

For example, if you have a `$group` variable (passed into your form perhaps as a form option) and `getUsers()` returns a collection of `User` entities, then you can supply the `choices` option directly:

Listing 25-3
```
use App\Entity\User;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

$builder->add('users', EntityType::class, array(
    'class' => User::class,
    'choices' => $group->getUsers(),
));
```

Select Tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

Element Type	Expanded	Multiple
select tag	false	false
select tag (with multiple attribute)	false	true
radio buttons	true	false
checkboxes	true	true

Field Options

choice_label

type: `string`, `callable` or *PropertyPath*[2]

This is the property that should be used for displaying the entities as text in the HTML element:

Listing 25-4
```
use App\Entity\Category;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

$builder->add('category', EntityType::class, array(
    'class' => Category::class,
    'choice_label' => 'displayName',
));
```

If left blank, the entity object will be cast to a string and so must have a `__toString()` method. You can also pass a callback function for more control:

Listing 25-5
```
use App\Entity\Category;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

```

2. https://api.symfony.com/4.0/Symfony/Component/PropertyAccess/PropertyPath.html

```
$builder->add('category', EntityType::class, array(
    'class' => Category::class,
    'choice_label' => function ($category) {
        return $category->getDisplayName();
    }
));
```

The method is called for each entity in the list and passed to the function. For more details, see the main choice_label documentation.

When passing a string, the `choice_label` option is a property path. So you can use anything supported by the *PropertyAccessor component*

For example, if the translations property is actually an associative array of objects, each with a name property, then you could do this:

Listing 25-6
```
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
// ...

$builder->add('genre', EntityType::class, array(
    'class' => 'App\Entity\Genre',
    'choice_label' => 'translations[en].name',
));
```

class

type: `string` **required**

The class of your entity (e.g. `App:Category`). This can be a fully-qualified class name (e.g. `App\Entity\Category`) or the short alias name (as shown prior).

em

type: `string` | `Doctrine\Common\Persistence\ObjectManager` **default**: the default entity manager

If specified, this entity manager will be used to load the choices instead of the `default` entity manager.

query_builder

type: `Doctrine\ORM\QueryBuilder` or a `callable` **default**: `null`

Allows you to create a custom query for your choices. See Using a Custom Query for the Entities for an example.

The value of this option can either be a `QueryBuilder` object, a callable or `null` (which will load all entities). When using a callable, you will be passed the `EntityRepository` of the entity as the only argument and should return a `QueryBuilder`. Returning `null` in the Closure will result in loading all entities.

The entity used in the `FROM` clause of the query_builder option will always be validated against the class which you have specified with the form's class option. If you return another entity instead of the one used in your `FROM` clause (for instance if you return an entity from a joined table), it will break validation.

Overridden Options

choice_name

type: `callable` or `string` **default**: `null`

Controls the internal field name of the choice. You normally don't care about this, but in some advanced cases, you might. For example, this "name" becomes the index of the choice views in the template.

This can be a callable or a property path. See choice_label for similar usage. If `null` is used, an incrementing integer is used as the name.

In the `EntityType`, this defaults to the `id` of the entity, if it can be read. Otherwise, it falls back to using auto-incrementing integers.

choice_value

type: `callable` or `string` **default**: `null`

Returns the string "value" for each choice, which must be unique across all choices. This is used in the `value` attribute in HTML and submitted in the POST/PUT requests. You don't normally need to worry about this, but it might be handy when processing an API request (since you can configure the value that will be sent in the API request).

This can be a callable or a property path. If `null` is given, an incrementing integer is used as the value.

If you pass a callable, it will receive one argument: the choice itself. When using the *EntityType Field*, the argument will be the entity object for each choice or `null` in some cases, which you need to handle:

Listing 25-7
```
'choice_value' => function (MyOptionEntity $entity = null) {
    return $entity ? $entity->getId() : '';
},
```

In the `EntityType`, this is overridden to use the `id` by default. When the `id` is used, Doctrine only queries for the objects for the ids that were actually submitted.

choices

type: `array` | `\Traversable` **default**: `null`

Instead of allowing the class and query_builder options to fetch the entities to include for you, you can pass the `choices` option directly. See Using Choices.

data_class

type: `string` **default**: `null`

This option is not used in favor of the `class` option which is required to query the entities.

Inherited Options

These options inherit from the *ChoiceType*:

choice_attr

type: `array`, `callable` or `string` **default**: `array()`

Use this to add additional HTML attributes to each choice. This can be an array of attributes (if they are the same for each choice), a callable or a property path (just like choice_label).

If an array, the keys of the `choices` array must be used as keys:

Listing 25-8
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('attending', ChoiceType::class, array(
    'choices' => array(
        'Yes' => true,
        'No' => false,
        'Maybe' => null,
    ),
    'choice_attr' => function($val, $key, $index) {
        // adds a class like attending_yes, attending_no, etc
        return ['class' => 'attending_'.strtolower($key)];
    },
));
```

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

group_by

type: `array`, `callable` or `string` **default**: `null`

You can easily "group" options in a select simply by passing a multi-dimensional array to `choices`. See the Grouping Options section about that.

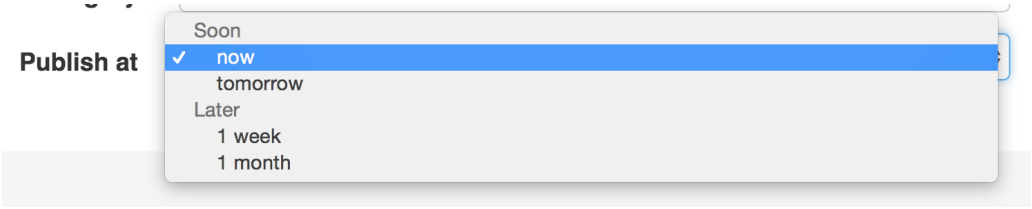
The `group_by` option is an alternative way to group choices, which gives you a bit more flexibility.

Take the following example:

Listing 25-9
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'group_by' => function($value, $key, $index) {
        if ($value <= new \DateTime('+3 days')) {
            return 'Soon';
        } else {
            return 'Later';
        }
```

```
17      },
18  ));
```

This groups the dates that are within 3 days into "Soon" and everything else into a "Later" group:



If you return `null`, the option won't be grouped. You can also pass a string "property path" that will be called to get the group. See the choice_label for details about using a property path.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

If you are working with a collection of Doctrine entities, it will be helpful to read the documentation for the *CollectionType Field* as well. In addition, there is a complete example in the *How to Embed a Collection of Forms* article.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 25-10
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 25-11
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 25-12

```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

Listing 25-13

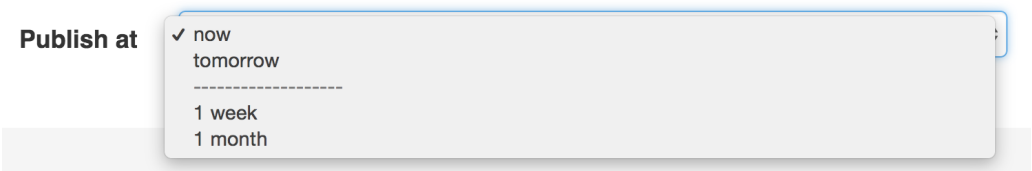
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 25-14

```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 25-15
```
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

This option expects an array of entity objects (that's actually the same as with the `ChoiceType` field, which requires an array of the preferred "values").

translation_domain

type: `string` **default**: `messages`

In case choice_translation_domain is set to `true` or `null`, this configures the exact translation domain that will be used for any labels or options that are rendered for this field

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 25-16
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 25-17
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 25-18
```
1  public function configureOptions(OptionsResolver $resolver)
2  {
3      $resolver->setDefaults(array(
4          'error_mapping' => array(
5              'matchingCityAndZipCode' => 'city',
6          ),
7      ));
8  }
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 25-19
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 25-20
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 25-21
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 25-22
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
```

```
7       'label_format' => 'form.address.%name%',
8   ));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. http://diveintohtml5.info/forms.html

Chapter 26
CountryType Field

The `CountryType` is a subset of the `ChoiceType` that displays countries of the world. As an added bonus, the country names are displayed in the language of the user.

The "value" for each country is the two-letter country code.

The locale of your user is guessed using *`Locale::getDefault()`*[1]

Unlike the `ChoiceType`, you don't need to specify a `choices` option as the field type automatically uses all of the countries of the world. You *can* specify the option manually, but then you should just use the `ChoiceType` directly.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Overridden options	• choices
Inherited options	from the *ChoiceType* • error_bubbling • error_mapping • expanded • multiple • placeholder • preferred_choices • trim from the *FormType* • data • disabled

1. https://secure.php.net/manual/en/locale.getdefault.php

	• empty_data • label • label_attr • label_format • mapped • required
Parent type	*ChoiceType*
Class	*CountryType*[2]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getRegionBundle()->getCountryNames()`

The country type defaults the `choices` option to the whole list of countries. The locale is used to translate the countries names.

If you want to override the built-in choices of the country type, you will also have to set the `choice_loader` option to `null`.

Inherited Options

These options inherit from the *ChoiceType*:

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 26-1
```
1  public function configureOptions(OptionsResolver $resolver)
2  {
3      $resolver->setDefaults(array(
```

2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/CountryType.html

```
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 26-2
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 26-3
```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 26-4
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 26-5
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

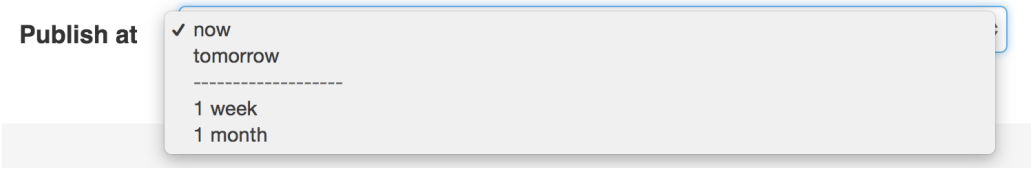
This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

Listing 26-6
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 26-7
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 26-8
```
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 26-9
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 26-10
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 26-11
```
1  {{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 26-12
```
1  {{ form_label(form.name, 'Your name', {
2          'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3  }) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 26-13
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. `http://diveintohtml5.info/forms.html`

Chapter 27
LanguageType Field

The `LanguageType` is a subset of the `ChoiceType` that allows the user to select from a large list of languages. As an added bonus, the language names are displayed in the language of the user.

The "value" for each language is the *Unicode language identifier* used in the *International Components for Unicode*[1] (e.g. `fr` or `zh_Hant`).

The locale of your user is guessed using *`Locale::getDefault()`*[2], which requires the `intl` PHP extension to be installed and enabled.

Unlike the `ChoiceType`, you don't need to specify a `choices` option as the field type automatically uses a large list of languages. You *can* specify the option manually, but then you should just use the `ChoiceType` directly.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Overridden options	• choices
Inherited options	from the *ChoiceType* • error_bubbling • error_mapping • expanded • multiple • placeholder • preferred_choices • trim from the *FormType*

1. http://site.icu-project.org
2. https://secure.php.net/manual/en/locale.getdefault.php

	• data • disabled • empty_data • label • label_attr • label_format • mapped • required
Parent type	*ChoiceType*
Class	*LanguageType*[3]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLanguageBundle()->getLanguageNames()`.

The choices option defaults to all languages. The default locale is used to translate the languages names.

If you want to override the built-in choices of the language type, you will also have to set the `choice_loader` option to `null`.

Inherited Options

These options inherit from the *ChoiceType*:

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 27-1
```
1  public function configureOptions(OptionsResolver $resolver)
2  {
```

3. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/LanguageType.html

```
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 27-2

```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 27-3

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 27-4
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 27-5
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

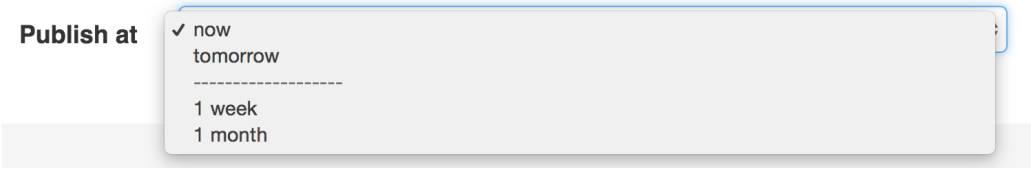
This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

Listing 27-6
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 27-7
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 27-8
```
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 27-9
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 27-10
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 27-11
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 27-12
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 27-13
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

4. `http://diveintohtml5.info/forms.html`

Chapter 28
LocaleType Field

The `LocaleType` is a subset of the `ChoiceType` that allows the user to select from a large list of locales (language+country). As an added bonus, the locale names are displayed in the language of the user.

The "value" for each locale is either the two letter *ISO 639-1*[1] *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the *ISO 3166-1 alpha-2*[2] *country* code (e.g. `fr_FR` for French/France).

The locale of your user is guessed using *`Locale::getDefault()`*[3]

Unlike the `ChoiceType`, you don't need to specify a `choices` option as the field type automatically uses a large list of locales. You *can* specify these options manually, but then you should just use the `ChoiceType` directly.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Overridden options	• choices
Inherited options	from the *ChoiceType* • error_bubbling • error_mapping • expanded • multiple • placeholder • preferred_choices • trim from the *FormType*

1. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
2. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
3. https://secure.php.net/manual/en/locale.getdefault.php

	• data • disabled • empty_data • label • label_attr • label_format • mapped • required
Parent type	*ChoiceType*
Class	*LocaleType*[4]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLocaleBundle()->getLocaleNames()`

The choices option defaults to all locales. It uses the default locale to specify the language.

If you want to override the built-in choices of the locale type, you will also have to set the `choice_loader` option to `null`.

Inherited Options

These options inherit from the *ChoiceType*:

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 28-1
```
1  public function configureOptions(OptionsResolver $resolver)
2  {
```

4. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/LocaleType.html

```
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 28-2

```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 28-3

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 28-4
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 28-5
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

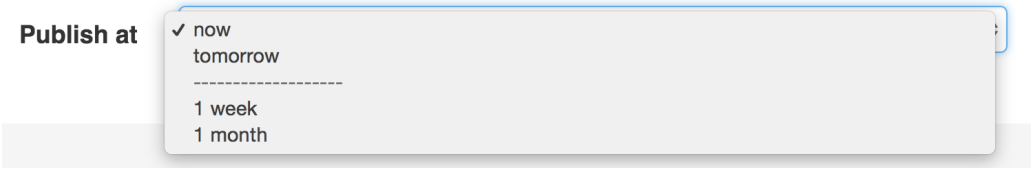
This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

Listing 28-6
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 28-7
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 28-8
```
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 28-9
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 28-10
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 28-11
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 28-12
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 28-13
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

%id%
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

%name%
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[5] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

5. http://diveintohtml5.info/forms.html

Chapter 29
TimezoneType Field

The `TimezoneType` is a subset of the `ChoiceType` that allows the user to select from all possible timezones.

The "value" for each timezone is the full timezone name, such as `America/Chicago` or `Europe/Istanbul`.

Unlike the `ChoiceType`, you don't need to specify a `choices` option as the field type automatically uses a large list of timezones. You *can* specify the option manually, but then you should just use the `ChoiceType` directly.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Options	• input • regions
Overridden options	• choices
Inherited options	from the *ChoiceType* • expanded • multiple • placeholder • preferred_choices • trim from the *FormType* • data • disabled • empty_data • error_bubbling • error_mapping • label

	• label_attr • label_format • mapped • required
Parent type	*ChoiceType*
Class	*TimezoneType*[1]

Field Options

input

type: `string` **default**: `string`

The format of the *input* data - i.e. the format that the timezone is stored on your underlying object. Valid values are:

- `string` (e.g. `America/New_York`)
- `datetimezone` (a `DateTimeZone` object)

regions

type: `int` **default**: `\DateTimeZone::ALL`

The available regions in the timezone choice list. For example: `DateTimeZone::AMERICA | DateTimeZone::EUROPE`

Overridden Options

choices

default: An array of timezones.

The Timezone type defaults the choices to all timezones returned by *`DateTimeZone::listIdentifiers()`*[2], broken down by continent.

If you want to override the built-in choices of the timezone type, you will also have to set the `choice_loader` option to `null`.

Inherited Options

These options inherit from the *ChoiceType*:

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/TimezoneType.html
2. https://secure.php.net/manual/en/datetimezone.listidentifiers.php

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 29-1
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

Listing 29-2
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('states', ChoiceType::class, array(
    'placeholder' => false,
));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 29-3
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

// a blank (with no text) option will be added
$builder->add('states', ChoiceType::class, array(
    'required' => false,
));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

Listing 29-4

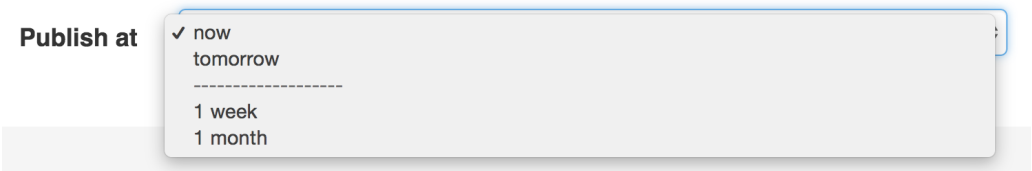
```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('language', ChoiceType::class, array(
    'choices' => array(
        'English' => 'en',
        'Spanish' => 'es',
        'Bork'   => 'muppets',
        'Pirate' => 'arr',
    ),
    'preferred_choices' => array('muppets', 'arr'),
));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 29-5

```php
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('publishAt', ChoiceType::class, array(
    'choices' => array(
        'now' => new \DateTime('now'),
        'tomorrow' => new \DateTime('+1 day'),
        '1 week' => new \DateTime('+1 week'),
        '1 month' => new \DateTime('+1 month'),
    ),
    'preferred_choices' => function ($value, $key) {
        // prefer options within 3 days
        return $value <= new \DateTime('+3 days');
    },
));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 29-6

```twig
{{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 29-7
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 29-8
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 29-9
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 29-10
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 29-11
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 29-12
```twig
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 29-13
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

3. `http://diveintohtml5.info/forms.html`

Chapter 30
CurrencyType Field

The `CurrencyType` is a subset of the *ChoiceType* that allows the user to select from a large list of *3-letter ISO 4217*[1] currencies.

Unlike the `ChoiceType`, you don't need to specify a `choices` option as the field type automatically uses a large list of currencies. You *can* specify the option manually, but then you should just use the `ChoiceType` directly.

Rendered as	can be various tags (see Select Tag, Checkboxes or Radio Buttons)
Overridden options	• choices
Inherited options	from the *ChoiceType* • error_bubbling • expanded • multiple • placeholder • preferred_choices • trim from the *FormType* type • data • disabled • empty_data • label • label_attr • label_format • mapped • required
Parent type	*ChoiceType*

1. `https://en.wikipedia.org/wiki/ISO_4217`

Class	*CurrencyType*[2]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getCurrencyBundle()->getCurrencyNames()`

The choices option defaults to all currencies.

If you want to override the built-in choices of the currency type, you will also have to set the `choice_loader` option to `null`.

Inherited Options

These options inherit from the *ChoiceType*:

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

expanded

type: `boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

multiple

type: `boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

placeholder

type: `string` or `boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if the `multiple` option is set to false.

- Add an empty value with "Choose an option" as the text:

Listing 30-1
```
1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
```

2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/CurrencyType.html

```
3
4  $builder->add('states', ChoiceType::class, array(
5      'placeholder' => 'Choose an option',
6  ));
```

- Guarantee that no "empty" value option is displayed:

Listing 30-2
```
1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  $builder->add('states', ChoiceType::class, array(
5      'placeholder' => false,
6  ));
```

If you leave the `placeholder` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 30-3
```
1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  // a blank (with no text) option will be added
5  $builder->add('states', ChoiceType::class, array(
6      'required' => false,
7  ));
```

preferred_choices

type: `array`, `callable` or `string` **default**: `array()`

This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

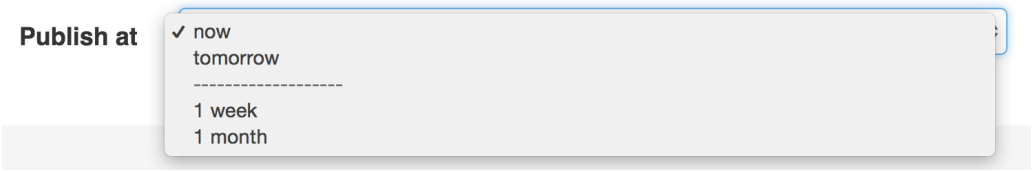
Listing 30-4
```
1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  $builder->add('language', ChoiceType::class, array(
5      'choices' => array(
6          'English' => 'en',
7          'Spanish' => 'es',
8          'Bork'   => 'muppets',
9          'Pirate' => 'arr',
10     ),
11     'preferred_choices' => array('muppets', 'arr'),
12 ));
```

This options can also be a callback function to give you more flexibility. This might be especially useful if your values are objects:

Listing 30-5
```
1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  $builder->add('publishAt', ChoiceType::class, array(
5      'choices' => array(
6          'now' => new \DateTime('now'),
7          'tomorrow' => new \DateTime('+1 day'),
8          '1 week' => new \DateTime('+1 week'),
9          '1 month' => new \DateTime('+1 month'),
10     ),
11     'preferred_choices' => function ($value, $key) {
12         // prefer options within 3 days
13         return $value <= new \DateTime('+3 days');
```

```
14      },
15  ));
```

This will "prefer" the "now" and "tomorrow" choices only:



Finally, if your values are objects, you can also specify a property path string on the object that will return true or false.

The preferred choices are only meaningful when rendering a `select` element (i.e. `expanded` false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

Listing 30-6
```
1  {{ form_widget(form.publishAt, { 'separator': '=====' }) }}
```

trim

type: `boolean` **default**: `false`

Trimming is disabled by default because the selected value or values must match the given choice values exactly (and they could contain whitespaces).

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 30-7
```
1  use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2  // ...
3
4  $builder->add('token', HiddenType::class, array(
5      'data' => 'abcdef',
6  ));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 30-8
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 30-9
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 30-10
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 30-11
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[3] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

3. `http://diveintohtml5.info/forms.html`

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Chapter 31
DateType Field

A field that allows the user to modify date information via a variety of different HTML elements.

This field can be rendered in a variety of different ways via the widget option and can understand a number of different input formats via the input option.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• choice_translation_domain • days • placeholder • format • html5 • input • model_timezone • months • view_timezone • widget • years
Overridden options	• by_reference • compound • data_class • error_bubbling
Inherited options	• data • disabled • error_mapping • inherit_data • invalid_message

	• invalid_message_parameters • mapped
Parent type	*FormType*
Class	*DateType*[1]

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `publishedAt` field whose underlying date is a `DateTime` object. The following configures the `date` type for that field as **three different choice fields**:

Listing 31-1
```
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...

$builder->add('publishedAt', DateType::class, array(
    'widget' => 'choice',
));
```

If your underlying date is *not* a `DateTime` object (e.g. it's a unix timestamp), configure the input option.

Rendering a single HTML5 Textbox

For a better user experience, you may want to render a single text field and use some kind of "date picker" to help your user fill in the right format. To do that, use the `single_text` widget:

Listing 31-2
```
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...

$builder->add('publishedAt', DateType::class, array(
    // renders it as a single text box
    'widget' => 'single_text',
));
```

This will render as an `input type="date"` HTML5 field, which means that **some - but not all - browsers will add nice date picker functionality to the field**. If you want to be absolutely sure that *every* user has a consistent date picker, use an external JavaScript library.

For example, suppose you want to use the *Bootstrap Datepicker*[2] library. First, make the following changes:

Listing 31-3
```
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...

$builder->add('publishedAt', DateType::class, array(
    'widget' => 'single_text',

    // prevents rendering it as type="date", to avoid HTML5 date pickers
    'html5' => false,

    // adds a class that can be selected in JavaScript
    'attr' => ['class' => 'js-datepicker'],
));
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/DateType.html
2. https://github.com/eternicode/bootstrap-datepicker

Then, add the following JavaScript code in your template to initialize the date picker:

Listing 31-4
```
<script>
    $(document).ready(function() {
        // you may need to change this code if you are not using Bootstrap Datepicker
        $('.js-datepicker').datepicker({
            format: 'yyyy-mm-dd'
        });
    });
</script>
```

This `format` key tells the date picker to use the date format that Symfony expects. This can be tricky: if the date picker is misconfigured, Symfony won't understand the format and will throw a validation error. You can also configure the format that Symfony should expect via the format option.

The string used by a JavaScript date picker to describe its format (e.g. `yyyy-mm-dd`) may not match the string that Symfony uses (e.g. `yyyy-MM-dd`). This is because different libraries use different formatting rules to describe the date format. Be aware of this - it can be tricky to make the formats truly match!

Field Options

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

days

type: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 31-5
```
'days' => range(1,31)
```

placeholder

type: `string` | `array`

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. When the placeholder value is a string, it will be used as the **blank value** of all select boxes:

Listing 31-6
```
$builder->add('dueDate', DateType::class, array(
    'placeholder' => 'Select a value',
));
```

Alternatively, you can use an array that configures different placeholder values for the year, month and day fields:

Listing 31-7
```
$builder->add('dueDate', DateType::class, array(
    'placeholder' => array(
```

```
        'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
    )
));
```

format

type: `integer` or `string` **default**: *IntlDateFormatter::MEDIUM*[3] (or `yyyy-MM-dd` if widget is `single_text`)

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the widget option is set to `single_text` and will define how the user will input the data. By default, the format is determined based on the current user locale: meaning that *the expected format will be different for different users*. You can override it by passing the format as a string.

For more information on valid formats, see *Date/Time Format Syntax*[4]:

Listing 31-8

```
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...

$builder->add('date_created', DateType::class, array(
    'widget' => 'single_text',
    // this is actually the default format for single_text
    'format' => 'yyyy-MM-dd',
));
```

If you want your field to be rendered as an HTML5 "date" field, you have to use a `single_text` widget with the `yyyy-MM-dd` format (the *RFC 3339*[5] format) which is the default value if you use the `single_text` widget.

html5

type: `boolean` **default**: `true`

If this is set to `true` (the default), it'll use the HTML5 type (date, time or datetime) to render the field. When set to `false`, it'll use the text type.

This is useful when you want to use a custom JavaScript datapicker, which often requires a text type instead of an HTML5 type.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

3. https://php.net/manual/en/class.intldateformatter.php#intl.intldateformatter-constants
4. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax
5. https://tools.ietf.org/html/rfc3339

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[6].

model_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[7].

months

type: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[8].

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

years

type: `array` **default**: five years before to five years after the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

Overridden Options

by_reference

default: `false`

The `DateTime` classes are treated as immutable objects.

6. https://php.net/manual/en/function.date.php#refsect1-function.date-changelog
7. https://php.net/manual/en/timezones.php
8. https://php.net/manual/en/timezones.php

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

data_class

type: `string` **default**: `null`

The internal normalized representation of this type is an array, not a `\DateTime` object. Therefore, the `data_class` option is initialized to `null` to avoid the `FormType` object from initializing it to `\DateTime`.

error_bubbling

default: `false`

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 31-9
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 31-10
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 31-11
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 31-12
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Field Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).
date_pattern	`string`	A string with the date format to use.

Chapter 32
DateIntervalType Field

This field allows the user to select an *interval* of time. For example, if you want to allow the user to choose *how often* they receive a status email, they could use this field to choose intervals like every "10 minutes" or "3 days".

The field can be rendered in a variety of different ways (see widget) and can be configured to give you a `DateInterval` object, an *ISO 8601*[1] duration string (e.g. `P1DT12H`) or an array (see input).

Underlying Data Type	can be `DateInterval`, string or array (see the `input` option)
Rendered as	single text box, multiple text boxes or select fields - see the widget option
Options	• days • hours • minutes • months • seconds • weeks • input • labels • placeholder • widget • with_days • with_hours • with_invert • with_minutes • with_months • with_seconds • with_weeks • with_years • years

1. `https://en.wikipedia.org/wiki/ISO_8601`

Inherited options	• data • disabled • inherit_data • invalid_message • invalid_message_parameters • mapped
Parent type	*FormType*
Class	*DateIntervalType*[2]

Basic Usage

This field type is highly configurable, but easy to use. The most important options are input and widget.

You can configure *a lot* of different options, including exactly *which* range options to show (e.g. don't show "months", but *do* show "days"):

Listing 32-1
```php
$builder->add('remindEvery', DateIntervalType::class, array(
    'widget'      => 'integer', // render a text field for each part
    // 'input'    => 'string',  // if you want the field to return a ISO 8601 string back to you

    // customize which text boxes are shown
    'with_years'  => false,
    'with_months' => false,
    'with_days'   => true,
    'with_hours'  => true,
));
```

Field Options

days

type: `array` **default**: 0 to 31

List of days available to the days field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 32-2
```php
'days' => range(1, 31)
```

placeholder

type: `string` or `array`

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. The `placeholder` option can be used to add a "blank" entry to the top of each select box:

Listing 32-3
```php
$builder->add('remindEvery', DateIntervalType::class, array(
    'placeholder' => '',
));
```

Alternatively, you can specify a string to be displayed for the "blank" value:

Listing 32-4

2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/DateIntervalType.html

```
$builder->add('remindEvery', DateIntervalType::class, array(
    'placeholder' => array('years' => 'Years', 'months' => 'Months', 'days' => 'Days')
));
```

hours

type: `array` **default**: 0 to 24

List of hours available to the hours field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 32-5
```
'hours' => range(1, 24)
```

input

type: `string` **default**: `dateinterval`

The format of the *input* data - i.e. the format that the interval is stored on your underlying object. Valid values are:

- `string` (a string formatted with *ISO 8601*[3] standard, e.g. `P7Y6M5DT12H15M30S`)
- `dateinterval` (a `DateInterval` object)
- `array` (e.g. `array('days' => '1', 'hours' => '12',)`)

The value that comes back from the form will also be normalized back into this format.

labels

type: `array` **default**: (see below)

The labels displayed for each of the elements of this type. The default values are `null`, so they display the "humanized version" of the child names (`Invert`, `Years`, etc.):

Listing 32-6
```
'labels' => array(
    'invert' => null,
    'years' => null,
    'months' => null,
    'days' => null,
    'hours' => null,
    'minutes' => null,
    'seconds' => null,
)
```

minutes

type: `array` **default**: 0 to 60

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 32-7
```
'minutes' => range(1, 60)
```

months

type: `array` **default**: 0 to 12

List of months available to the months field type. This option is only relevant when the `widget` option is set to `choice`:

3. https://en.wikipedia.org/wiki/ISO_8601

```
'months' => range(1, 12)
```

Listing 32-8

seconds

type: `array` **default**: 0 to 60

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`:

```
'seconds' => range(1, 60)
```

Listing 32-9

weeks

type: `array` **default**: 0 to 52

List of weeks available to the weeks field type. This option is only relevant when the `widget` option is set to `choice`:

```
'weeks' => range(1, 52)
```

Listing 32-10

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders one to six select inputs for years, months, weeks, days, hours, minutes and/or seconds, depending on the with_years, with_months, with_weeks, with_days, with_hours, with_minutes and with_seconds options. Default: Three fields for years, months and days.
- `text`: renders one to six text inputs for years, months, weeks, days, hours, minutes and/or seconds, depending on the with_years, with_months, with_weeks, with_days, with_hours, with_minutes and with_seconds options. Default: Three fields for years, months and days.
- `integer`: renders one to six integer inputs for years, months, weeks, days, hours, minutes and/or seconds, depending on the with_years, with_months, with_weeks, with_days, with_hours, with_minutes and with_seconds options. Default: Three fields for years, months and days.
- `single_text`: renders a single input of type `text`. User's input will be validated against the form `PnYnMnDTnHnMnS` (or `PnW` if using only weeks).

with_days

type: `Boolean` **default**: `true`

Whether or not to include days in the input. This will result in an additional input to capture days.

This can not be used when with_weeks is enabled.

with_hours

type: `Boolean` **default**: `false`

Whether or not to include hours in the input. This will result in an additional input to capture hours.

with_invert

type: `Boolean` **default**: `false`

Whether or not to include invert in the input. This will result in an additional checkbox. This can not be used when the widget option is set to `single_text`.

with_minutes

type: `Boolean` **default**: `false`

Whether or not to include minutes in the input. This will result in an additional input to capture minutes.

with_months

type: `Boolean` **default**: `true`

Whether or not to include months in the input. This will result in an additional input to capture months.

with_seconds

type: `Boolean` **default**: `false`

Whether or not to include seconds in the input. This will result in an additional input to capture seconds.

with_weeks

type: `Boolean` **default**: `false`

Whether or not to include weeks in the input. This will result in an additional input to capture weeks.

This can not be used when with_days is enabled.

with_years

type: `Boolean` **default**: `true`

Whether or not to include years in the input. This will result in an additional input to capture years.

years

type: `array` **default**: 0 to 100

List of years available to the years field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 32-11
```
'years' => range(1, 100)
```

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 32-12
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 32-13
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
```

```
4       'invalid_message_parameters' => array('%num%' => 6),
5   ));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Field Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
with_days	`Boolean`	The value of the with_days option.
with_invert	`Boolean`	The value of the with_invert option.
with_hours	`Boolean`	The value of the with_hours option.
with_minutes	`Boolean`	The value of the with_minutes option.
with_months	`Boolean`	The value of the with_months option.
with_seconds	`Boolean`	The value of the with_seconds option.
with_weeks	`Boolean`	The value of the with_weeks option.
with_years	`Boolean`	The value of the with_years option.

Chapter 33
DateTimeType Field

This field type allows the user to modify data that represents a specific date and time (e.g. `1984-06-05 12:15:30`).

Can be rendered as a text input or select tags. The underlying format of the data can be a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• choice_translation_domain • date_format • date_widget • days • placeholder • format • hours • html5 • input • minutes • model_timezone • months • seconds • time_widget • view_timezone • widget • with_minutes • with_seconds • years
Overridden options	• by_reference

	• compound • data_class • error_bubbling
Inherited options	• data • disabled • inherit_data • invalid_message • invalid_message_parameters • mapped
Parent type	*FormType*
Class	*DateTimeType*[1]

Field Options

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

date_format

type: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

Defines the `format` option that will be passed down to the date field. See the DateType's format option for more details.

date_widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

days

type: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 33-1
```
'days' => range(1,31)
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/DateTimeType.html

placeholder

type: `string` | `array`

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. When the placeholder value is a string, it will be used as the **blank value** of all select boxes:

Listing 33-2
```
use Symfony\Component\Form\Extension\Core\Type\DateTimeType;

$builder->add('startDateTime', DateTimeType::class, array(
    'placeholder' => 'Select a value',
));
```

Alternatively, you can use an array that configures different placeholder values for the year, month, day, hour, minute and second fields:

Listing 33-3
```
use Symfony\Component\Form\Extension\Core\Type\DateTimeType;

$builder->add('startDateTime', DateTimeType::class, array(
    'placeholder' => array(
        'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
        'hour' => 'Hour', 'minute' => 'Minute', 'second' => 'Second',
    )
));
```

format

type: `string` **default**: `Symfony\Component\Form\Extension\Core\Type\DateTimeType::HTML5_FORMAT`

If the `widget` option is set to `single_text`, this option specifies the format of the input, i.e. how Symfony will interpret the given input as a datetime string. It defaults to the *RFC 3339*[2] format which is used by the HTML5 `datetime` field. Keeping the default value will cause the field to be rendered as an `input` field with `type="datetime"`. For more information on valid formats, see *Date/Time Format Syntax*[3].

hours

type: `array` **default**: 0 to 23

List of hours available to the hours field type. This option is only relevant when the `widget` option is set to `choice`.

html5

type: `boolean` **default**: `true`

If this is set to `true` (the default), it'll use the HTML5 type (date, time or datetime) to render the field. When set to `false`, it'll use the text type.

This is useful when you want to use a custom JavaScript datapicker, which often requires a text type instead of an HTML5 type.

input

type: `string` **default**: `datetime`

2. https://tools.ietf.org/html/rfc3339
3. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05 12:15:00`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array(2011, 06, 05, 12, 15, 0)`)
- `timestamp` (e.g. `1307276100`)

The value that comes back from the form will also be normalized back into this format.

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[4].

minutes

type: `array` **default**: 0 to 59

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`.

model_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[5].

months

type: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

seconds

type: `array` **default**: 0 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

time_widget

type: `string` **default**: `choice`

Defines the `widget` option for the *TimeType*.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[6].

4. https://php.net/manual/en/function.date.php#refsect1-function.date-changelog
5. https://php.net/manual/en/timezones.php
6. https://php.net/manual/en/timezones.php

widget

type: `string` **default**: `null`

Defines the `widget` option for both the *DateType* and *TimeType*. This can be overridden with the date_widget and time_widget options.

with_minutes

type: `boolean` **default**: `true`

Whether or not to include minutes in the input. This will result in an additional input to capture minutes.

with_seconds

type: `boolean` **default**: `false`

Whether or not to include seconds in the input. This will result in an additional input to capture seconds.

years

type: `array` **default**: five years before to five years after the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

Overridden Options

by_reference

default: `false`

The `DateTime` classes are treated as immutable objects.

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

data_class

type: `string` **default**: `null`

The internal normalized representation of this type is an array, not a `\DateTime` object. Therefore, the `data_class` option is initialized to `null` to avoid the `FormType` object from initializing it to `\DateTime`.

error_bubbling

default: `false`

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 33-4
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 33-5
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Field Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).

Chapter 34
TimeType Field

A field to capture time input.

This can be rendered as a text field, a series of text fields (e.g. hour, minute, second) or a series of select fields. The underlying data can be stored as a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	can be various tags (see below)
Options	• choice_translation_domain • placeholder • hours • html5 • input • minutes • model_timezone • seconds • view_timezone • widget • with_minutes • with_seconds
Overridden options	• by_reference • compound • data_class • error_bubbling
Inherited Options	• data • disabled • error_mapping • inherit_data

	• invalid_message • invalid_message_parameters • mapped
Parent type	FormType
Class	*TimeType*[1]

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `startTime` field whose underlying time data is a `DateTime` object. The following configures the `TimeType` for that field as two different choice fields:

Listing 34-1
```
use Symfony\Component\Form\Extension\Core\Type\TimeType;
// ...

$builder->add('startTime', TimeType::class, array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

The `input` option *must* be changed to match the type of the underlying date data. For example, if the `startTime` field's data were a unix timestamp, you'd need to set `input` to `timestamp`:

Listing 34-2
```
use Symfony\Component\Form\Extension\Core\Type\TimeType;
// ...

$builder->add('startTime', TimeType::class, array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

placeholder

type: `string` | `array`

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/TimeType.html

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. When the placeholder value is a string, it will be used as the **blank value** of all select boxes:

Listing 34-3
```
$builder->add('startTime', 'time', array(
    'placeholder' => 'Select a value',
));
```

Alternatively, you can use an array that configures different placeholder values for the hour, minute and second fields:

Listing 34-4
```
1  $builder->add('startTime', 'time', array(
2      'placeholder' => array(
3          'hour' => 'Hour', 'minute' => 'Minute', 'second' => 'Second',
4      )
5  ));
```

hours

type: `array` **default**: 0 to 23

List of hours available to the hours field type. This option is only relevant when the `widget` option is set to `choice`.

html5

type: `boolean` **default**: `true`

If this is set to `true` (the default), it'll use the HTML5 type (date, time or datetime) to render the field. When set to `false`, it'll use the text type.

This is useful when you want to use a custom JavaScript datapicker, which often requires a text type instead of an HTML5 type.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `12:17:26`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('hour' => 12, 'minute' => 17, 'second' => 26)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

minutes

type: `array` **default**: 0 to 59

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`.

model_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[2].

seconds

type: `array` **default**: 0 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[3].

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders one, two (default) or three select inputs (hour, minute, second), depending on the with_minutes and with_seconds options.
- `text`: renders one, two (default) or three text inputs (hour, minute, second), depending on the with_minutes and with_seconds options.
- `single_text`: renders a single input of type `time`. User's input will be validated against the form `hh:mm` (or `hh:mm:ss` if using seconds).

Combining the widget type `single_text` and the with_minutes option set to `false` can cause unexpected behavior in the client as the input type `time` might not support selecting an hour only.

with_minutes

type: `boolean` **default**: `true`

Whether or not to include minutes in the input. This will result in an additional input to capture minutes.

with_seconds

type: `boolean` **default**: `false`

Whether or not to include seconds in the input. This will result in an additional input to capture seconds.

Overridden Options

by_reference

default: `false`

The `DateTime` classes are treated as immutable objects.

2. `https://php.net/manual/en/timezones.php`
3. `https://php.net/manual/en/timezones.php`

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

data_class

type: `string` **default**: `null`

The internal normalized representation of this type is an array, not a `\DateTime` object. Therefore, the `data_class` option is initialized to `null` to avoid the `FormType` object from initializing it to `\DateTime`.

error_bubbling

default: `false`

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 34-5
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 34-6
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 34-7
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 34-8
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Form Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
with_minutes	`boolean`	The value of the with_minutes option.
with_seconds	`boolean`	The value of the with_seconds option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).

Chapter 35
BirthdayType Field

A *DateType* field that specializes in handling birthdate data.

Can be rendered as a single text box, three text boxes (month, day and year), or three select boxes.

This type is essentially the same as the *DateType* type, but with a more appropriate default for the years option. The years option defaults to 120 years ago to the current year.

Underlying Data Type	can be `DateTime`, `string`, `timestamp`, or `array` (see the input option)
Rendered as	can be three select boxes or 1 or 3 text boxes, based on the widget option
Overridden options	• years
Inherited options	from the *DateType*: • choice_translation_domain • days • placeholder • format • input • model_timezone • months • view_timezone • widget from the *FormType*: • data • disabled • inherit_data • invalid_message • invalid_message_parameters • mapped

Parent type	*DateType*
Class	*BirthdayType*[1]

Overridden Options

years

type: `array` **default**: 120 years ago to the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

Inherited Options

These options inherit from the *DateType*:

choice_translation_domain

type: `string`, `boolean` or `null`

This option determines if the choice values should be translated and in which translation domain.

The values of the `choice_translation_domain` option can be `true` (reuse the current translation domain), `false` (disable translation), `null` (uses the parent translation domain or the default domain) or a string which represents the exact translation domain to use.

days

type: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 35-1
```
'days' => range(1,31)
```

placeholder

type: `string` | `array`

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. When the placeholder value is a string, it will be used as the **blank value** of all select boxes:

Listing 35-2
```
$builder->add('birthdate', BirthdayType::class, array(
    'placeholder' => 'Select a value',
));
```

Alternatively, you can use an array that configures different placeholder values for the year, month and day fields:

Listing 35-3
```
1 $builder->add('birthdate', BirthdayType::class, array(
2     'placeholder' => array(
3         'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/BirthdayType.html

```
    )
));
```

format

type: `integer` or `string` **default**: *IntlDateFormatter::MEDIUM*[2] (or `yyyy-MM-dd` if widget is `single_text`)

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the widget option is set to `single_text` and will define how the user will input the data. By default, the format is determined based on the current user locale: meaning that *the expected format will be different for different users*. You can override it by passing the format as a string.

For more information on valid formats, see *Date/Time Format Syntax*[3]:

Listing 35-4
```php
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...

$builder->add('date_created', DateType::class, array(
    'widget' => 'single_text',
    // this is actually the default format for single_text
    'format' => 'yyyy-MM-dd',
));
```

If you want your field to be rendered as an HTML5 "date" field, you have to use a `single_text` widget with the `yyyy-MM-dd` format (the *RFC 3339*[4] format) which is the default value if you use the `single_text` widget.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[5].

model_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[6].

2. https://php.net/manual/en/class.intldateformatter.php#intl.intldateformatter-constants
3. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax
4. https://tools.ietf.org/html/rfc3339
5. https://php.net/manual/en/function.date.php#refsect1-function.date-changelog

months

type: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[7].

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 35-5

```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

inherit_data

type: `boolean` **default**: `false`

6. https://php.net/manual/en/timezones.php
7. https://php.net/manual/en/timezones.php

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 35-6
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Chapter 36
CheckboxType Field

Creates a single input checkbox. This should always be used for a field that has a boolean value: if the box is checked, the field will be set to true, if the box is unchecked, the value will be set to false.

Rendered as	`input checkbox` field
Options	• value
Overridden options	• compound • empty_data
Inherited options	• data • disabled • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*CheckboxType*[1]

Example Usage

Listing 36-1
```
1 use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
2 // ...
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/CheckboxType.html

```
3
4 $builder->add('public', CheckboxType::class, array(
5     'label'    => 'Show this entry publicly?',
6     'required' => false,
7 ));
```

Field Options

value

type: `mixed` **default**: `1`

The value that's actually used as the value for the checkbox or radio button. This does not affect the value that's set on your object.

To make a checkbox or radio button checked by default, use the data option.

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies if a form is compound. As it's not the case for checkbox, by default the value is overridden with the `false` value.

empty_data

type: `string` **default**: `mixed`

This option determines what value the field will return when the `placeholder` choice is selected. In the checkbox and the radio type, the value of `empty_data` is overriden by the value returned by the data transformer (see *How to Use Data Transformers*).

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 36-2
```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
```

```
$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 36-3
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 36-4
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 36-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 36-6
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 36-7
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Form Variables

Variable	Type	Usage
checked	`boolean`	Whether or not the current input is checked.

2. `http://diveintohtml5.info/forms.html`

Chapter 37
FileType Field

The `FileType` represents a file input in your form.

Rendered as	`input file` field
Options	• multiple
Overridden options	• compound • data_class • empty_data
Inherited options	• disabled • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*FileType*[1]

Basic Usage

Say you have this form definition:

Listing 37-1
```
use Symfony\Component\Form\Extension\Core\Type\FileType;
// ...
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/FileType.html

```
$builder->add('attachment', FileType::class);
```

When the form is submitted, the `attachment` field will be an instance of *`UploadedFile`*[2]. It can be used to move the `attachment` file to a permanent location:

Listing 37-2
```
use Symfony\Component\HttpFoundation\File\UploadedFile;

public function upload()
{
    // ...

    if ($form->isSubmitted() && $form->isValid()) {
        $someNewFilename = ...

        $file = $form['attachment']->getData();
        $file->move($directory, $someNewFilename);

        // ...
    }

    // ...
}
```

The `move()` method takes a directory and a file name as its arguments. You might calculate the filename in one of the following ways:

Listing 37-3
```
// use the original file name
$file->move($directory, $file->getClientOriginalName());

// compute a random name and try to guess the extension (more secure)
$extension = $file->guessExtension();
if (!$extension) {
    // extension cannot be guessed
    $extension = 'bin';
}
$file->move($directory, rand(1, 99999).'.'.$extension);
```

Using the original name via `getClientOriginalName()` is not safe as it could have been manipulated by the end-user. Moreover, it can contain characters that are not allowed in file names. You should sanitize the name before using it directly.

Read *How to Upload Files* for an example of how to manage a file upload associated with a Doctrine entity.

Field Options

multiple

type: `Boolean` **default**: `false`

When set to true, the user will be able to upload multiple files at the same time.

Overridden Options

compound

type: `boolean` **default**: `false`

2. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/File/UploadedFile.html

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

data_class

type: `string` **default**: *`File`*[3]

This option sets the appropriate file-related data mapper to be used by the type.

empty_data

type: `mixed` **default**: `null`

This option determines what value the field will return when the submitted value is empty.

Inherited Options

These options inherit from the *FormType*:

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 37-4
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

3. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/File/File.html

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 37-5
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 37-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 37-7
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 37-8
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
```

```
7       'label_format' => 'form.address.%name%',
8   ));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Form Variables

Variable	Type	Usage
type	`string`	The type variable is set to `file`, in order to render as a file input field.

4. `http://diveintohtml5.info/forms.html`

Chapter 38
RadioType Field

Creates a single radio button. If the radio button is selected, the field will be set to the specified value. Radio buttons cannot be unchecked - the value only changes when another radio button with the same name gets checked.

The `RadioType` isn't usually used directly. More commonly it's used internally by other types such as *ChoiceType*. If you want to have a boolean field, use *CheckboxType*.

Rendered as	`input radio` field
Inherited options	from the *CheckboxType*: • value from the *FormType*: • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*CheckboxType*
Class	*RadioType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/RadioType.html

Inherited Options

These options inherit from the *CheckboxType*:

value

type: `mixed` **default**: `1`

The value that's actually used as the value for the checkbox or radio button. This does not affect the value that's set on your object.

To make a checkbox or radio button checked by default, use the data option.

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 38-1
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

empty_data

type: `string` **default**: `mixed`

This option determines what value the field will return when the `placeholder` choice is selected. In the checkbox and the radio type, the value of `empty_data` is overriden by the value returned by the data transformer (see *How to Use Data Transformers*).

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 38-2
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 38-3
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 38-4
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 38-5
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 38-6
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Form Variables

Variable	Type	Usage
checked	`boolean`	Whether or not the current input is checked.

2. `http://diveintohtml5.info/forms.html`

Chapter 39
CollectionType Field

This field type is used to render a "collection" of some field or form. In the easiest sense, it could be an array of `TextType` fields that populate an array `emails` values. In more complex examples, you can embed entire forms, which is useful when creating forms that expose one-to-many relationships (e.g. a product from where you can manage many related product photos).

Rendered as	depends on the entry_type option
Options	• allow_add • allow_delete • delete_empty • entry_options • entry_type • prototype • prototype_data • prototype_name
Inherited options	• by_reference • empty_data • error_bubbling • error_mapping • label • label_attr • label_format • mapped • required
Parent type	*FormType*
Class	*CollectionType*[1]

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/CollectionType.html

If you are working with a collection of Doctrine entities, pay special attention to the allow_add, allow_delete and by_reference options. You can also see a complete example in the *How to Embed a Collection of Forms* article.

Basic Usage

This type is used when you want to manage a collection of similar items in a form. For example, suppose you have an `emails` field that corresponds to an array of email addresses. In the form, you want to expose each email address as its own input text box:

Listing 39-1
```php
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
// ...

$builder->add('emails', CollectionType::class, array(
    // each entry in the array will be an "email" field
    'entry_type' => EmailType::class,
    // these options are passed to each "email" type
    'entry_options' => array(
        'attr' => array('class' => 'email-box'),
    ),
));
```

The simplest way to render this is all at once:

Listing 39-2
```
{{ form_row(form.emails) }}
```

A much more flexible method would look like this:

Listing 39-3
```
{{ form_label(form.emails) }}
{{ form_errors(form.emails) }}

<ul>
{% for emailField in form.emails %}
    <li>
        {{ form_errors(emailField) }}
        {{ form_widget(emailField) }}
    </li>
{% endfor %}
</ul>
```

In both cases, no input fields would render unless your `emails` data array already contained some emails.

In this simple example, it's still impossible to add new addresses or remove existing addresses. Adding new addresses is possible by using the allow_add option (and optionally the prototype option) (see example below). Removing emails from the `emails` array is possible with the allow_delete option.

Adding and Removing Items

If allow_add is set to `true`, then if any unrecognized items are submitted, they'll be added seamlessly to the array of items. This is great in theory, but takes a little bit more effort in practice to get the client-side JavaScript correct.

Following along with the previous example, suppose you start with two emails in the `emails` data array. In that case, two input fields will be rendered that will look something like this (depending on the name of your form):

Listing 39-4

```
<input type="email" id="form_emails_0" name="form[emails][0]" value="foo@foo.com" />
<input type="email" id="form_emails_1" name="form[emails][1]" value="bar@bar.com" />
```

To allow your user to add another email, just set allow_add to `true` and - via JavaScript - render another field with the name `form[emails][2]` (and so on for more and more fields).

To help make this easier, setting the prototype option to `true` allows you to render a "template" field, which you can then use in your JavaScript to help you dynamically create these new fields. A rendered prototype field will look like this:

Listing 39-5
```
<input type="email"
    id="form_emails___name__"
    name="form[emails][__name__]"
    value=""
/>
```

By replacing `__name__` with some unique value (e.g. `2`), you can build and insert new HTML fields into your form.

Using jQuery, a simple example might look like this. If you're rendering your collection fields all at once (e.g. `form_row(form.emails)`), then things are even easier because the `data-prototype` attribute is rendered automatically for you (with a slight difference - see note below) and all you need is this JavaScript code:

Listing 39-6
```
// add-collection-widget.js
jQuery(document).ready(function () {
    jQuery('.add-another-collection-widget').click(function (e) {
        e.preventDefault();
        var list = jQuery(jQuery(this).attr('data-list'));
        // Try to find the counter of the list
        var counter = list.data('widget-counter') | list.children().length;
        // If the counter does not exist, use the length of the list
        if (!counter) { counter = list.children().length; }

        // grab the prototype template
        var newWidget = list.attr('data-prototype');
        // replace the "__name__" used in the id and name of the prototype
        // with a number that's unique to your emails
        // end name attribute looks like name="contact[emails][2]"
        newWidget = newWidget.replace(/__name__/g, counter);
        // Increase the counter
        counter++;
        // And store it, the length cannot be used if deleting widgets is allowed
        list.data(' widget-counter', counter);

        // create a new list element and add it to the list
        var newElem = jQuery(list.attr('data-widget-tags')).html(newWidget);
        newElem.appendTo(list);
    });
});
```

And update the template as follows:

Listing 39-7
```
{{ form_start(form) }}
    {# ... #}

    {# store the prototype on the data-prototype attribute #}
    <ul id="email-fields-list"
        data-prototype="{{ form_widget(form.emails.vars.prototype)|e }}"
        data-widget-tags="{{ '<li></li>'|e }}">
    {% for emailField in form.emails %}
        <li>
            {{ form_errors(emailField) }}
            {{ form_widget(emailField) }}
        </li>
```

```
13      {% endfor %}
14      </ul>
15
16      <a href="#"
17          class="add-another-collection-widget"
18          data-list="#email-fields-list">Add another email</a>
19
20      {# ... #}
21  {{ form_end(form) }}
22
23  <script src="add-collection-widget.js"></script>
```

If you're rendering the entire collection at once, then the prototype is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection. The only difference is that the entire "form row" is rendered for you, meaning you wouldn't have to wrap it in any container element as it was done above.

Field Options

allow_add

type: `boolean` **default**: `false`

If set to `true`, then if unrecognized items are submitted to the collection, they will be added as new items. The ending array will contain the existing items as well as the new item that was in the submitted data. See the above example for more details.

The prototype option can be used to help render a prototype item that can be used - with JavaScript - to create new form items dynamically on the client side. For more information, see the above example and Allowing "new" Tags with the "Prototype".

If you're embedding entire other forms to reflect a one-to-many database relationship, you may need to manually ensure that the foreign key of these new objects is set correctly. If you're using Doctrine, this won't happen automatically. See the above link for more details.

allow_delete

type: `boolean` **default**: `false`

If set to `true`, then if an existing item is not contained in the submitted data, it will be correctly absent from the final array of items. This means that you can implement a "delete" button via JavaScript which simply removes a form element from the DOM. When the user submits the form, its absence from the submitted data will mean that it's removed from the final array.

For more information, see Allowing Tags to be Removed.

Be careful when using this option when you're embedding a collection of objects. In this case, if any embedded forms are removed, they *will* correctly be missing from the final array of objects. However, depending on your application logic, when one of those objects is removed, you may want to delete it or at least remove its foreign key reference to the main object. None of this is handled automatically. For more information, see Allowing Tags to be Removed.

delete_empty

type: `Boolean` or `callable` **default**: `false`

If you want to explicitly remove entirely empty collection entries from your form you have to set this option to `true`. However, existing collection entries will only be deleted if you have the allow_delete option enabled. Otherwise the empty values will be kept.

The `delete_empty` option only removes items when the normalized value is `null`. If the nested entry_type is a compound form type, you must either set the `required` option to `false` or set the `empty_data` option to `null`. Both of these options can be set inside entry_options. Read about the form's empty_data option to learn why this is necessary.

A value is deleted from the collection only if the normalized value is `null`. However, you can also set the option value to a callable, which will be executed for each value in the submitted collection. If the callable returns `true`, the value is removed from the collection. For example:

Listing 39-8

```
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
// ...

$builder->add('users', CollectionType::class, array(
    // ...
    'delete_empty' => function (User $user = null) {
        return null === $user || empty($user->getFirstName());
    },
));
```

Using a callable is particularly useful in case of compound form types, which may define complex conditions for considering them empty.

entry_options

type: `array` **default**: `array()`

This is the array that's passed to the form type specified in the entry_type option. For example, if you used the *ChoiceType* as your entry_type option (e.g. for a collection of drop-down menus), then you'd need to at least pass the `choices` option to the underlying type:

Listing 39-9

```
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('favorite_cities', CollectionType::class, array(
    'entry_type'   => ChoiceType::class,
    'entry_options'  => array(
        'choices'  => array(
            'Nashville' => 'nashville',
            'Paris'     => 'paris',
            'Berlin'    => 'berlin',
            'London'    => 'london',
        ),
    ),
));
```

entry_type

type: `string` **default**: `Symfony\\Component\\Form\\Extension\\Core\\Type\\TextType`

This is the field type for each item in this collection (e.g. `TextType`, `ChoiceType`, etc). For example, if you have an array of email addresses, you'd use the *EmailType*. If you want to embed a collection of some other form, pass the form type class as this option (e.g. `MyFormType::class`).

prototype

type: `boolean` **default**: `true`

This option is useful when using the allow_add option. If `true` (and if allow_add is also `true`), a special "prototype" attribute will be available so that you can render a "template" example on your page of what a new element should look like. The `name` attribute given to this element is `__name__`. This allows you to add a "add another" button via JavaScript which reads the prototype, replaces `__name__` with some unique name or number and render it inside your form. When submitted, it will be added to your underlying array due to the allow_add option.

The prototype field can be rendered via the `prototype` variable in the collection field:

Listing 39-10
```
{{ form_row(form.emails.vars.prototype) }}
```

Note that all you really need is the "widget", but depending on how you're rendering your form, having the entire "form row" may be easier for you.

If you're rendering the entire collection field at once, then the prototype form row is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection.

For details on how to actually use this option, see the above example as well as Allowing "new" Tags with the "Prototype".

prototype_data

type: `mixed` **default**: `null`

Allows you to define specific data for the prototype. Each new row added will initially contain the data set by this option. By default, the data configured for all entries with the entry_options option will be used.

Listing 39-11
```
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
// ...

$builder->add('tags', CollectionType::class, array(
    'entry_type' => TextType::class,
    'allow_add' => true,
    'prototype' => true,
    'prototype_data' => 'New Tag Placeholder',
));
```

prototype_name

type: `string` **default**: `__name__`

If you have several collections in your form, or worse, nested collections you may want to change the placeholder so that unrelated placeholders are not replaced with the same value.

Inherited Options

These options inherit from the *FormType*. Not all options are listed here - only the most applicable to this type:

by_reference

type: `boolean` **default**: `true`

In most cases, if you have an `author` field, then you expect `setAuthor()` to be called on the underlying object. In some cases, however, `setAuthor()` may *not* be called. Setting `by_reference` to `false` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 39-12
```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\FormType;
// ...

$builder = $this->createFormBuilder($article);
$builder
    ->add('title', TextType::class)
    ->add(
        $builder->create('author', FormType::class, array('by_reference' => ?))
            ->add('name', TextType::class)
            ->add('email', EmailType::class)
    )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 39-13
```
$article->setTitle('...');
$article->getAuthor()->setName('...');
$article->getAuthor()->setEmail('...');
```

Notice that `setAuthor()` is not called. The author is modified by reference.

If you set `by_reference` to false, submitting looks like this:

Listing 39-14
```
$article->setTitle('...');
$author = clone $article->getAuthor();
$author->setName('...');
$author->setEmail('...');
$article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *CollectionType* field where your underlying collection data is an object (like with Doctrine's `ArrayCollection`), then `by_reference` must be set to `false` if you need the adder and remover (e.g. `addAuthor()` and `removeAuthor()`) to be called.

empty_data

type: `mixed`

The default value is `array()` (empty array).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 39-15
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `true`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 39-16
```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults(array(
4         'error_mapping' => array(
5             'matchingCityAndZipCode' => 'city',
6         ),
7     ));
8 }
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;

- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 39-17
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 39-18
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 39-19
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 39-20
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[2] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

Field Variables

Variable	Type	Usage
allow_add	`boolean`	The value of the allow_add option.
allow_delete	`boolean`	The value of the allow_delete option.

2. `http://diveintohtml5.info/forms.html`

Chapter 40
RepeatedType Field

This is a special field "group", that creates two identical fields whose values must match (or a validation error is thrown). The most common use is when you need the user to repeat their password or email to verify accuracy.

Rendered as	input `text` field by default, but see type option
Options	• first_name • first_options • options • second_name • second_options • type
Overridden options	• error_bubbling
Inherited options	• data • error_mapping • invalid_message • invalid_message_parameters • mapped
Parent type	*FormType*
Class	*RepeatedType*[1]

Example Usage

Listing 40-1

1. `https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/RepeatedType.html`

```php
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
// ...

$builder->add('password', RepeatedType::class, array(
    'type' => PasswordType::class,
    'invalid_message' => 'The password fields must match.',
    'options' => array('attr' => array('class' => 'password-field')),
    'required' => true,
    'first_options'  => array('label' => 'Password'),
    'second_options' => array('label' => 'Repeat Password'),
));
```

Upon a successful form submit, the value entered into both of the "password" fields becomes the data of the `password` key. In other words, even though two fields are actually rendered, the end data from the form is just the single value (usually a string) that you need.

The most important option is `type`, which can be any field type and determines the actual type of the two underlying fields. The `options` option is passed to each of those individual fields, meaning - in this example - any option supported by the `PasswordType` can be passed in this array.

Rendering

The repeated field type is actually two underlying fields, which you can render all at once, or individually. To render all at once, use something like:

Listing 40-2
```twig
{{ form_row(form.password) }}
```

To render each field individually, use something like this:

Listing 40-3
```twig
{# .first and .second may vary in your use - see the note below #}
{{ form_row(form.password.first) }}
{{ form_row(form.password.second) }}
```

The names `first` and `second` are the default names for the two sub-fields. However, these names can be controlled via the first_name and second_name options. If you've set these options, then use those values instead of `first` and `second` when rendering.

Validation

One of the key features of the `repeated` field is internal validation (you don't need to do anything to set this up) that forces the two fields to have a matching value. If the two fields don't match, an error will be shown to the user.

The `invalid_message` is used to customize the error that will be displayed when the two fields do not match each other.

Field Options

first_name

type: `string` **default**: `first`

This is the actual field name to be used for the first field. This is mostly meaningless, however, as the actual data entered into both of the fields will be available under the key assigned to the `RepeatedType`

field itself (e.g. `password`). However, if you don't specify a label, this field name is used to "guess" the label for you.

first_options

type: `array` **default**: `array()`

Additional options (will be merged into options below) that should be passed *only* to the first field. This is especially useful for customizing the label:

Listing 40-4
```php
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
// ...

$builder->add('password', RepeatedType::class, array(
    'first_options'  => array('label' => 'Password'),
    'second_options' => array('label' => 'Repeat Password'),
));
```

options

type: `array` **default**: `array()`

This options array will be passed to each of the two underlying fields. In other words, these are the options that customize the individual field types. For example, if the `type` option is set to `password`, this array might contain the options `always_empty` or `required` - both options that are supported by the `PasswordType` field.

second_name

type: `string` **default**: `second`

The same as `first_name`, but for the second field.

second_options

type: `array` **default**: `array()`

Additional options (will be merged into options above) that should be passed *only* to the second field. This is especially useful for customizing the label (see first_options).

type

type: `string` **default**: `Symfony\Component\Form\Extension\Core\Type\TextType`

The two underlying fields will be of this field type. For example, passing `PasswordType::class` will render two password fields.

Overridden Options

error_bubbling

default: `false`

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 40-5
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 40-6
```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 40-7
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 40-8
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

Chapter 41

HiddenType Field

The hidden type represents a hidden input field.

Rendered as	`input hidden` field
Overriden options	• compound • error_bubbling • required
Inherited options	• data • error_mapping • mapped • property_path
Parent type	*FormType*
Class	*HiddenType*[1]

Overridden Options

compound

type: `boolean` **default**: `false`

This option specifies whether the type contains child types or not. This option is managed internally for built-in types, so there is no need to configure it explicitly.

error_bubbling

default: `true`

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/HiddenType.html

Pass errors to the root form, otherwise they will not be visible.

required

default: `false`

Hidden fields cannot have a required attribute.

Inherited Options

These options inherit from the *FormType*:

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 41-1
```php
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 41-2
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;

- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 41-3
```
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

property_path

type: `PropertyPathInterface|string|null` **default**: `null`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value (`null`) will use the field's name as the property.

Chapter 42

ButtonType Field

A simple, non-responsive button.

Rendered as	`button` tag
Inherited options	• attr • disabled • label • translation_domain
Parent type	none
Class	*ButtonType*[1]

Inherited Options

The following options are defined in the *BaseType*[2] class. The `BaseType` class is the parent class for both the `button` type and the *FormType*, but it is not part of the form type tree (i.e. it cannot be used as a form type on its own).

attr

type: `array` **default**: `array()`

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 42-1

```
use Symfony\Component\Form\Extension\Core\Type\ButtonType;
// ...

```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/ButtonType.html
2. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/BaseType.html

```
4  $builder->add('save', ButtonType::class, array(
5      'attr' => array('class' => 'save'),
6  ));
```

disabled

type: `boolean` **default**: `false`

If you don't want a user to be able to click a button, you can set the disabled option to true. It will not be possible to submit the form with this button, not even when bypassing the browser and sending a request manually, for example with cURL.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be displayed on the button. The label can also be directly set inside the template:

Listing 42-2
```
1  {{ form_widget(form.save, { 'label': 'Click me' }) }}
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

Chapter 43
ResetType Field

A button that resets all fields to their original values.

Rendered as	`input reset` tag
Inherited options	• attr • disabled • label • label_attr • translation_domain
Parent type	*ButtonType*
Class	*ResetType*[1]

Inherited Options

attr

type: `array` **default**: `array()`

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 43-1
```
use Symfony\Component\Form\Extension\Core\Type\ResetType;
// ...

$builder->add('save', ResetType::class, array(
    'attr' => array('class' => 'save'),
));
```

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/ResetType.html

disabled

type: `boolean` **default**: `false`

If you don't want a user to be able to click a button, you can set the disabled option to true. It will not be possible to submit the form with this button, not even when bypassing the browser and sending a request manually, for example with cURL.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be displayed on the button. The label can also be directly set inside the template:

Listing 43-2
```
{{ form_widget(form.save, { 'label': 'Click me' }) }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 43-3
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

Chapter 44
SubmitType Field

A submit button.

Rendered as	`button submit` tag
Inherited options	• attr • disabled • label • label_attr • label_format • translation_domain • validation_groups
Parent type	*ButtonType*
Class	*SubmitType*[1]

The Submit button has an additional method *`isClicked()`*[2] that lets you check whether this button was used to submit the form. This is especially useful when *a form has multiple submit buttons*:

Listing 44-1
```
if ($form->get('save')->isClicked()) {
    // ...
}
```

Inherited Options

attr

type: `array` **default**: `array()`

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/SubmitType.html
2. https://api.symfony.com/4.0/Symfony/Component/Form/ClickableInterface.html#method_isClicked

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 44-2
```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
// ...

$builder->add('save', SubmitType::class, array(
    'attr' => array('class' => 'save'),
));
```

disabled

type: `boolean` **default**: `false`

If you don't want a user to be able to click a button, you can set the disabled option to true. It will not be possible to submit the form with this button, not even when bypassing the browser and sending a request manually, for example with cURL.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be displayed on the button. The label can also be directly set inside the template:

Listing 44-3
```
{{ form_widget(form.save, { 'label': 'Click me' }) }}
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 44-4
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 44-5
```
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
```

```
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

%id%
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

%name%
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

validation_groups

type: `array` **default**: `null`

When your form contains multiple submit buttons, you can change the validation group based on the button which was used to submit the form. Imagine a registration form wizard with buttons to go to the previous or the next step:

Listing 44-6
```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
// ...

$form = $this->createFormBuilder($user)
    ->add('previousStep', SubmitType::class, array(
        'validation_groups' => false,
    ))
    ->add('nextStep', SubmitType::class, array(
        'validation_groups' => array('Registration'),
    ))
    ->getForm();
```

The special `false` ensures that no validation is performed when the previous step button is clicked. When the second button is clicked, all constraints from the "Registration" are validated.

You can read more about this in How to Choose Validation Groups Based on the Submitted Data.

Form Variables

Variable	Type	Usage
clicked	`boolean`	Whether the button is clicked or not.

Chapter 45
FormType Field

The `FormType` predefines a couple of options that are then available on all types for which `FormType` is the parent.

Options	• action • allow_extra_fields • by_reference • compound • constraints • data • data_class • empty_data • error_bubbling • error_mapping • extra_fields_message • inherit_data • invalid_message • invalid_message_parameters • label_attr • label_format • mapped • method • post_max_size_message • property_path • required • trim
Inherited options	• attr • auto_initialize • block_name • disabled • label

	• translation_domain
Parent	none
Class	*FormType*[1]

Field Options

action

type: `string` **default**: empty string

This option specifies where to send the form's data on submission (usually a URI). Its value is rendered as the `action` attribute of the `form` element. An empty value is considered a same-document reference, i.e. the form will be submitted to the same URI that rendered the form.

allow_extra_fields

type: `boolean` **default**: `false`

Usually, if you submit extra fields that aren't configured in your form, you'll get a "This form should not contain extra fields." validation error.

You can silence this validation error by enabling the `allow_extra_fields` option on the form.

by_reference

type: `boolean` **default**: `true`

In most cases, if you have an `author` field, then you expect `setAuthor()` to be called on the underlying object. In some cases, however, `setAuthor()` may *not* be called. Setting `by_reference` to `false` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 45-1
```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\FormType;
// ...

$builder = $this->createFormBuilder($article);
$builder
    ->add('title', TextType::class)
    ->add(
        $builder->create('author', FormType::class, array('by_reference' => ?))
            ->add('name', TextType::class)
            ->add('email', EmailType::class)
    )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 45-2
```
$article->setTitle('...');
$article->getAuthor()->setName('...');
$article->getAuthor()->setEmail('...');
```

Notice that `setAuthor()` is not called. The author is modified by reference.

1. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/FormType.html

If you set `by_reference` to false, submitting looks like this:

Listing 45-3
```
$article->setTitle('...');
$author = clone $article->getAuthor();
$author->setName('...');
$author->setEmail('...');
$article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *CollectionType* field where your underlying collection data is an object (like with Doctrine's `ArrayCollection`), then `by_reference` must be set to `false` if you need the adder and remover (e.g. `addAuthor()` and `removeAuthor()`) to be called.

compound

type: `boolean` **default**: `true`

If `true` this option creates the form as "compound", meaning that it can contain children and be a parent of other forms.

Most of the time you won't need to override this option. You might want to control for it when creating a custom form type with advanced rendering logic.

In a view a compound form is rendered as a `<div>` container or a `<form>` element (the whole form is obviously a compound form).

Non-compound forms are always leaves in a form tree, they cannot have children.

A non-compound form is rendered as one of the html form elements: `<input>` (`TextType`, `FileType`, `HiddenType`), `<textarea>` (`TextareaType`) or `<select>` (`ChoiceType`).

An interesting case is the `ChoiceType`. With `expanded=false` it is a non-compound form and is rendered as a `<select>` tag. With `expanded=true` the `ChoiceType` becomes a compound form and is rendered as a set of radios or checkboxes.

constraints

type: `array` or *`Constraint`*[2] **default**: `null`

Allows you to attach one or more validation constraints to a specific field. For more information, see Adding Validation. This option is added in the *`FormTypeValidatorExtension`*[3] form extension.

data

type: `mixed` **default**: Defaults to field of the underlying structure.

When you create a form, each field initially displays the value of the corresponding property of the form's domain data (e.g. if you bind an object to the form). If you want to override this initial value for the form or an individual field, you can set it in the data option:

Listing 45-4
```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
// ...

$builder->add('token', HiddenType::class, array(
    'data' => 'abcdef',
));
```

2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraint.html
3. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Validator/Type/FormTypeValidatorExtension.html

The `data` option *always* overrides the value taken from the domain data (object) when rendering. This means the object value is also overriden when the form edits an already persisted object, causing it to lose its persisted value when the form is submitted.

data_class

type: `string`

This option is used to set the appropriate data mapper to be used by the form, so you can use it for any form field type which requires an object:

Listing 45-5
```
use App\Entity\Media;
use App\Form\MediaType;
// ...

$builder->add('media', MediaType::class, array(
    'data_class' => Media::class,
));
```

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `data_class` is set and `required` is `true`, then `new $data_class()`;
- If `data_class` is set and `required` is `false`, then `null`;
- If `data_class` is not set and `compound` is `true`, then `array()` (empty array);
- If `data_class` is not set and `compound` is `false`, then `''` (empty string).

This option determines what value the field will *return* when the submitted value is empty (or missing). It does not set an initial value if none is provided when the form is rendered in a view.

This means it helps you handling form submission with blank fields. For example, if you want the `name` field to be explicitly set to `John Doe` when no value is selected, you can do it like this:

Listing 45-6
```
$builder->add('name', null, array(
    'required'   => false,
    'empty_data' => 'John Doe',
));
```

This will still render an empty text box, but upon submission the `John Doe` value will be set. Use the `data` or `placeholder` options to show this initial value in the rendered form.

If a form is compound, you can set `empty_data` as an array, object or closure. See the *How to Configure empty Data for a Form Class* article for more details about these options.

If you want to set the `empty_data` option for your entire form class, see the *How to Configure empty Data for a Form Class* article.

Form data transformers will still be applied to the `empty_data` value. This means that an empty string will be cast to `null`. Use a custom data transformer if you explicitly want to return the empty string.

error_bubbling

type: `boolean` **default**: `false` unless the form is `compound`

If `true`, any errors for this field will be passed to the parent field or form. For example, if set to `true` on a normal field, any errors for that field will be attached to the main form, not to the specific field.

error_mapping

type: `array` **default**: `array()`

This option allows you to modify the target of a validation error.

Imagine you have a custom method named `matchingCityAndZipCode()` that validates whether the city and zip code match. Unfortunately, there is no "matchingCityAndZipCode" field in your form, so all that Symfony can do is display the error on top of the form.

With customized error mapping, you can do better: map the error to the city field so that it displays above it:

Listing 45-7
```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

Here are the rules for the left and the right side of the mapping:

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The right side contains simply the names of fields in the form.

By default, errors for any property that is not mapped will bubble up to the parent form. You can use the dot (`.`) on the left side to map errors of all unmapped properties to a particular field. For instance, to map all these errors to the `city` field, use:

Listing 45-8
```php
$resolver->setDefaults(array(
    'error_mapping' => array(
        '.' => 'city',
    ),
));
```

extra_fields_message

type: `string` **default**: `This form should not contain extra fields.`

This is the validation error message that's used if the submitted form data contains one or more fields that are not part of the form definition. The placeholder `{{ extra_fields }}` can be used to display a comma separated list of the submitted extra field names.

inherit_data

type: `boolean` **default**: `false`

This option determines if the form will inherit data from its parent form. This can be useful if you have a set of fields that are duplicated across multiple forms. See *How to Reduce Code Duplication with "inherit_data"*.

When a field has the `inherit_data` option set, it uses the data of the parent form as is. This means that *Data Transformers* won't be applied to that field.

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a *TimeType* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (reference).

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 45-9
```
$builder->add('some_field', SomeFormType::class, array(
    // ...
    'invalid_message' => 'You entered an invalid value, it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label_attr

type: `array` **default**: `array()`

Sets the HTML attributes for the `<label>` element, which will be used when rendering the label for the field. It's an associative array with HTML attribute as a key. This attributes can also be directly set inside the template:

Listing 45-10
```
{{ form_label(form.name, 'Your name', {
        'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}
}) }}
```

label_format

type: `string` **default**: `null`

Configures the string used as the label of the field, in case the `label` option was not set. This is useful when using keyword translation messages.

If you're using keyword translation messages as labels, you often end up having multiple keyword messages for the same label (e.g. `profile_address_street`, `invoice_address_street`). This is because the label is build for each "path" to a field. To avoid duplicated keyword messages, you can configure the label format to a static value, like:

Listing 45-11
```php
// ...
$profileFormBuilder->add('address', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));

$invoiceFormBuilder->add('invoice', AddressType::class, array(
    'label_format' => 'form.address.%name%',
));
```

This option is inherited by the child types. With the code above, the label of the `street` field of both forms will use the `form.address.street` keyword message.

Two variables are available in the label format:

`%id%`
: A unique identifier for the field, consisting of the complete path to the field and the field name (e.g. `profile_address_street`);

`%name%`
: The field name (e.g. `street`).

The default value (`null`) results in a "humanized" version of the field name.

The `label_format` option is evaluated in the form theme. Make sure to update your templates in case you *customized form theming*.

mapped

type: `boolean` **default**: `true`

If you wish the field to be ignored when reading or writing to the object, you can set the `mapped` option to `false`.

method

type: `string` **default**: `POST`

This option specifies the HTTP method used to submit the form's data. Its value is rendered as the `method` attribute of the `form` element and is used to decide whether to process the form submission in the `handleRequest()` method after submission. Possible values are:

- POST
- GET
- PUT
- DELETE
- PATCH

When the method is PUT, PATCH, or DELETE, Symfony will automatically render a `_method` hidden field in your form. This is used to "fake" these HTTP methods, as they're not supported on standard browsers. This can be useful when using method routing requirements.

The PATCH method allows submitting partial data. In other words, if the submitted form data is missing certain fields, those will be ignored and the default values (if any) will be used. With all other HTTP methods, if the submitted form data is missing some fields, those fields are set to `null`.

post_max_size_message

type: `string` **default**: `The uploaded file was too large. Please try to upload a smaller file.`

This is the validation error message that's used if submitted POST form data exceeds `php.ini`'s `post_max_size` directive. The `{{ max }}` placeholder can be used to display the allowed size.

Validating the `post_max_size` only happens on the root form.

property_path

type: `PropertyPathInterface|string|null` **default**: `null`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value (`null`) will use the field's name as the property.

required

type: `boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

The required option also affects how empty data for each field is handled. For more details, see the empty_data option.

trim

type: `boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[5] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

Inherited Options

The following options are defined in the *`BaseType`*[6] class. The `BaseType` class is the parent class for both the `form` type and the *ButtonType*, but it is not part of the form type tree (i.e. it cannot be used as a form type on its own).

4. http://diveintohtml5.info/forms.html
5. https://secure.php.net/manual/en/function.trim.php
6. https://api.symfony.com/4.0/Symfony/Component/Form/Extension/Core/Type/BaseType.html

attr

type: `array` **default**: `array()`

If you want to add extra attributes to an HTML field representation you can use the `attr` option. It's an associative array with HTML attributes as keys. This can be useful when you need to set a custom class for some widget:

Listing 45-12
```
$builder->add('body', TextareaType::class, array(
    'attr' => array('class' => 'tinymce'),
));
```

auto_initialize

type: `boolean` **default**: `true`

An internal option: sets whether the form should be initialized automatically. For all fields, this option should only be `true` for root forms. You won't need to change this option and probably won't need to worry about it.

block_name

type: `string` **default**: the form's name (see Knowing which block to customize)

Allows you to override the block name used to render the form type. Useful for example if you have multiple instances of the same form and you need to personalize the rendering of the forms individually.

disabled

type: `boolean` **default**: `false`

If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

label

type: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. Setting to false will suppress the label. The label can also be directly set inside the template:

Listing 45-13
```
1  {{ form_label(form.name, 'Your name') }}
```

translation_domain

type: `string`, `null` or `false` **default**: `null`

This is the translation domain that will be used for any label or option that is rendered for this field. Use `null` to reuse the translation domain of the parent form (or the default domain of the translator for the root form). Use `false` to disable translations.

Chapter 46

Validation Constraints Reference

The Validator is designed to validate objects against *constraints*. In real life, a constraint could be: "The cake must not be burned". In Symfony, constraints are similar: They are assertions that a condition is true.

Supported Constraints

The following constraints are natively available in Symfony:

Basic Constraints

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

- *NotBlank*
- *Blank*
- *NotNull*
- *IsNull*
- *IsTrue*
- *IsFalse*
- *Type*

String Constraints

- *Email*
- *Length*
- *Url*
- *Regex*
- *Ip*
- *Uuid*

Number Constraints

- *Range*

Comparison Constraints

- *EqualTo*
- *NotEqualTo*
- *IdenticalTo*
- *NotIdenticalTo*
- *LessThan*
- *LessThanOrEqual*
- *GreaterThan*
- *GreaterThanOrEqual*

Date Constraints

- *Date*
- *DateTime*
- *Time*

Collection Constraints

- *Choice*
- *Collection*
- *Count*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Financial and other Number Constraints

- *Bic*
- *CardScheme*
- *Currency*
- *Luhn*
- *Iban*
- *Isbn*
- *Issn*

Other Constraints

- *Callback*
- *Expression*
- *All*
- *UserPassword*
- *Valid*

Chapter 47
NotBlank

Validates that a value is not blank - meaning not equal to a blank string, a blank array or `null`:

Listing 47-1
```php
if (false === $value || (empty($value) && '0' != $value)) {
    // validation will fail
}
```

To force that a value is simply not equal to `null`, see the *NotNull* constraint.

Applies to	property or method
Options	• message • payload
Class	*NotBlank*[1]
Validator	*NotBlankValidator*[2]

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not blank, you could do the following:

Listing 47-2
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotBlank.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotBlankValidator.html

```
11      protected $firstName;
12  }
```

Options

message

type: `string` **default**: `This value should not be blank.`

This is the message that will be shown if the value is blank.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 48
Blank

Validates that a value is blank - meaning equal to an empty string or `null`:

Listing 48-1
```
if ('' !== $value && null !== $value) {
    // validation will fail
}
```

To force that a value strictly be equal to `null`, see the *IsNull* constraint.

To force that a value is *not* blank, see *NotBlank*. But be careful as `NotBlank` is *not* strictly the opposite of `Blank`.

Applies to	property or method
Options	• message • payload
Class	*Blank*[1]
Validator	*BlankValidator*[2]

Basic Usage

If, for some reason, you wanted to ensure that the `firstName` property of an `Author` class were blank, you could do the following:

Listing 48-2
```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Blank.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/BlankValidator.html

```
8       /**
9        * @Assert\Blank()
10       */
11      protected $firstName;
12  }
```

Options

message

type: `string` **default**: `This value should be blank.`

This is the message that will be shown if the value is not blank.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 49
NotNull

Validates that a value is not strictly equal to `null`. To ensure that a value is simply not blank (not a blank string), see the *NotBlank* constraint.

Applies to	property or method
Options	• message • payload
Class	*NotNull*[1]
Validator	*NotNullValidator*[2]

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not strictly equal to `null`, you would:

Listing 49-1
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotNull()
     */
    protected $firstName;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotNull.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotNullValidator.html

Options

message

type: `string` **default**: `This value should not be null.`

This is the message that will be shown if the value is `null`.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 50
IsNull

Validates that a value is exactly equal to `null`. To force that a property is simply blank (blank string or `null`), see the *Blank* constraint. To ensure that a property is not null, see *NotNull*.

Also see *NotNull*.

Applies to	property or method
Options	• message • payload
Class	*IsNull*[1]
Validator	*IsNullValidator*[2]

Basic Usage

If, for some reason, you wanted to ensure that the `firstName` property of an `Author` class exactly equal to `null`, you could do the following:

Listing 50-1

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\IsNull()
     */
    protected $firstName;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IsNull.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IsNullValidator.html

Options

message

type: `string` **default**: `This value should be null.`

This is the message that will be shown if the value is not `null`.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 51
IsTrue

Validates that a value is `true`. Specifically, this checks to see if the value is exactly `true`, exactly the integer `1`, or exactly the string "`1`".

Also see *IsFalse*.

Applies to	property or method
Options	• message • payload
Class	*True*[1]
Validator	*TrueValidator*[2]

Basic Usage

This constraint can be applied to properties (e.g. a `termsAccepted` property on a registration model) or to a "getter" method. It's most powerful in the latter case, where you can assert that a method returns a true value. For example, suppose you have the following method:

Listing 51-1
```php
// src/Entity/Author.php
namespace App\Entity;

class Author
{
    protected $token;

    public function isTokenValid()
    {
        return $this->token == $this->generateToken();
    }
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/True.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/TrueValidator.html

Then you can constrain this method with `IsTrue`.

Listing 51-2

```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    protected $token;

    /**
     * @Assert\IsTrue(message="The token is invalid")
     */
    public function isTokenValid()
    {
        return $this->token == $this->generateToken();
    }
}
```

If the `isTokenValid()` returns false, the validation will fail.

Options

message

type: `string` **default**: `This value should be true.`

This message is shown if the underlying data is not true.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 52
IsFalse

Validates that a value is `false`. Specifically, this checks to see if the value is exactly `false`, exactly the integer `0`, or exactly the string "`0`".

Also see *IsTrue*.

Applies to	property or method
Options	• message • payload
Class	*IsFalse*[1]
Validator	*IsFalseValidator*[2]

Basic Usage

The `IsFalse` constraint can be applied to a property or a "getter" method, but is most commonly useful in the latter case. For example, suppose that you want to guarantee that some `state` property is *not* in a dynamic `invalidStates` array. First, you'd create a "getter" method:

Listing 52-1

```
protected $state;

protected $invalidStates = array();

public function isStateInvalid()
{
    return in_array($this->state, $this->invalidStates);
}
```

In this case, the underlying object is only valid if the `isStateInvalid()` method returns **false**:

Listing 52-2

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IsFalse.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IsFalseValidator.html

```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\IsFalse(
     *     message = "You've entered an invalid state."
     * )
     */
     public function isStateInvalid()
     {
        // ...
     }
}
```

Options

message

type: `string` **default**: `This value should be false.`

This message is shown if the underlying data is not false.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 53
Type

Validates that a value is of a specific data type. For example, if a variable should be an array, you can use this constraint with the `array` type option to validate this.

Applies to	property or method
Options	• type • message • payload
Class	*Type*[1]
Validator	*TypeValidator*[2]

Basic Usage

This will check if `firstName` is of type `string` and that `age` is an `integer`.

Listing 53-1

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Type("string")
     */
    protected $firstName;

    /**
     * @Assert\Type(
     *     type="integer",
     *     message="The value {{ value }} is not a valid {{ type }}."
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Type.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/TypeValidator.html

```
17          * )
18          */
19         protected $age;
20     }
```

Options

type

type: `string` [default option]

This required option is the fully qualified class name or one of the PHP datatypes as determined by PHP's `is_()` functions.

- *array*[3]
- *bool*[4]
- *callable*[5]
- *float*[6]
- *double*[7]
- *int*[8]
- *integer*[9]
- *long*[10]
- *null*[11]
- *numeric*[12]
- *object*[13]
- *real*[14]
- *resource*[15]
- *scalar*[16]
- *string*[17]

Also, you can use `ctype_()` functions from corresponding *built-in PHP extension*[18]. Consider *a list of ctype functions*[19]:

- *alnum*[20]
- *alpha*[21]
- *cntrl*[22]
- *digit*[23]
- *graph*[24]
- *lower*[25]

---

3. https://secure.php.net/manual/en/function.is-array.php
4. https://secure.php.net/manual/en/function.is-bool.php
5. https://secure.php.net/manual/en/function.is-callable.php
6. https://secure.php.net/manual/en/function.is-float.php
7. https://secure.php.net/manual/en/function.is-double.php
8. https://secure.php.net/manual/en/function.is-int.php
9. https://secure.php.net/manual/en/function.is-integer.php
10. https://secure.php.net/manual/en/function.is-long.php
11. https://secure.php.net/manual/en/function.is-null.php
12. https://secure.php.net/manual/en/function.is-numeric.php
13. https://secure.php.net/manual/en/function.is-object.php
14. https://secure.php.net/manual/en/function.is-real.php
15. https://secure.php.net/manual/en/function.is-resource.php
16. https://secure.php.net/manual/en/function.is-scalar.php
17. https://secure.php.net/manual/en/function.is-string.php
18. https://php.net/book.ctype.php
19. https://php.net/ref.ctype.php
20. https://secure.php.net/manual/en/function.ctype-alnum.php
21. https://secure.php.net/manual/en/function.ctype-alpha.php
22. https://secure.php.net/manual/en/function.ctype-cntrl.php
23. https://secure.php.net/manual/en/function.ctype-digit.php
24. https://secure.php.net/manual/en/function.ctype-graph.php

- *print*[26]
- *punct*[27]
- *space*[28]
- *upper*[29]
- *xdigit*[30]

Make sure that the proper *locale*[31] is set before using one of these.

message

type: `string` **default**: `This value should be of type {{ type }}.`

The message if the underlying data is not of the given type.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

25. https://secure.php.net/manual/en/function.ctype-lower.php
26. https://secure.php.net/manual/en/function.ctype-print.php
27. https://secure.php.net/manual/en/function.ctype-punct.php
28. https://secure.php.net/manual/en/function.ctype-space.php
29. https://secure.php.net/manual/en/function.ctype-upper.php
30. https://secure.php.net/manual/en/function.ctype-xdigit.php
31. https://secure.php.net/manual/en/function.setlocale.php

Chapter 54
Email

Validates that a value is a valid email address. The underlying value is cast to a string before being validated.

Applies to	property or method
Options	• strict • message • checkMX • checkHost • payload
Class	*Email*[1]
Validator	*EmailValidator*[2]

Basic Usage

Listing 54-1

```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Email(
     *     message = "The email '{{ value }}' is not a valid email.",
     *     checkMX = true
     * )
     */
     protected $email;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Email.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/EmailValidator.html

Options

strict

type: `boolean` **default**: `false`

When false, the email will be validated against a simple regular expression. If true, then the *egulias/email-validator*[3] library is required to perform an RFC compliant validation.

message

type: `string` **default**: `This value is not a valid email address.`

This message is shown if the underlying data is not a valid email address.

checkMX

type: `boolean` **default**: `false`

If true, then the *checkdnsrr*[4] PHP function will be used to check the validity of the MX record of the host of the given email.

checkHost

type: `boolean` **default**: `false`

If true, then the *checkdnsrr*[5] PHP function will be used to check the validity of the MX *or* the A *or* the AAAA record of the host of the given email.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

3. https://packagist.org/packages/egulias/email-validator
4. https://secure.php.net/manual/en/function.checkdnsrr.php
5. https://secure.php.net/manual/en/function.checkdnsrr.php

Chapter 55
Length

Validates that a given string length is *between* some minimum and maximum value.

`null` and empty strings are not handled by this constraint. You need to also add the *NotBlank* or *NotNull* constraints to validate against these.

Applies to	property or method
Options	• min • max • charset • minMessage • maxMessage • exactMessage • payload
Class	*Length*[1]
Validator	*LengthValidator*[2]

Basic Usage

To verify that the `firstName` field length of a class is between "2" and "50", you might add the following:

Listing 55-1
```
// src/Entity/Participant.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Length.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LengthValidator.html

```
6  class Participant
7  {
8      /**
9       * @Assert\Length(
10      *      min = 2,
11      *      max = 50,
12      *      minMessage = "Your first name must be at least {{ limit }} characters long",
13      *      maxMessage = "Your first name cannot be longer than {{ limit }} characters"
14      * )
15      */
16      protected $firstName;
17 }
```

Options

min

type: `integer`

This required option is the "min" length value. Validation will fail if the given value's length is **less** than this min value.

It is important to notice that NULL values and empty strings are considered valid no matter if the constraint required a minimum length. Validators are triggered only if the value is not blank.

max

type: `integer`

This required option is the "max" length value. Validation will fail if the given value's length is **greater** than this max value.

charset

type: `string` **default**: `UTF-8`

The charset to be used when computing value's length. The *`grapheme_strlen`*[3] PHP function is used if available. If not, the *`mb_strlen`*[4] PHP function is used if available. If neither are available, the *`strlen`*[5] PHP function is used.

minMessage

type: `string` **default**: `This value is too short. It should have {{ limit }} characters or more.`

The message that will be shown if the underlying value's length is less than the min option.

maxMessage

type: `string` **default**: `This value is too long. It should have {{ limit }} characters or less.`

The message that will be shown if the underlying value's length is more than the max option.

---

3. https://secure.php.net/manual/en/function.grapheme-strlen.php
4. https://secure.php.net/manual/en/function.mb-strlen.php
5. https://secure.php.net/manual/en/function.strlen.php

exactMessage

type: `string` **default**: `This value should have exactly {{ limit }} characters.`

The message that will be shown if min and max values are equal and the underlying value's length is not exactly this value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 56
Url

Validates that a value is a valid URL string.

Applies to	property or method
Options	• message • protocols • payload • checkDNS • dnsMessage
Class	*Url*[1]
Validator	*UrlValidator*[2]

Basic Usage

Listing 56-1

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url()
     */
     protected $bioUrl;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Url.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/UrlValidator.html

Options

message

type: `string` **default**: `This value is not a valid URL.`

This message is shown if the URL is invalid.

Listing 56-2
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url(
     *    message = "The url '{{ value }}' is not a valid url",
     * )
     */
     protected $bioUrl;
}
```

protocols

type: `array` **default**: `array('http', 'https')`

The protocols considered to be valid for the URL. For example, if you also consider the `ftp://` type URLs to be valid, redefine the `protocols` array, listing `http`, `https`, and also `ftp`.

Listing 56-3
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url(
     *    protocols = {"http", "https", "ftp"}
     * )
     */
     protected $bioUrl;
}
```

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

checkDNS

type: `boolean` **default**: `false`

By default, this constraint just validates the syntax of the given URL. If you also need to check whether the associated host exists, set the `checkDNS` option to the value of any of the `CHECK_DNS_TYPE_*` constants in the *`Url`*[3] class:

Listing 56-4
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url(
     *    checkDNS = "ANY"
     * )
     */
     protected $bioUrl;
}
```

This option uses the *`checkdnsrr`*[4] PHP function to check the validity of the DNS record corresponding to the host associated with the given URL.

dnsMessage

type: `string` **default**: `The host could not be resolved.`

This message is shown when the `checkDNS` option is set to `true` and the DNS check failed.

Listing 56-5
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url(
     *    dnsMessage = "The host '{{ value }}' could not be resolved."
     * )
     */
     protected $bioUrl;
}
```

3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Url.html
4. https://secure.php.net/manual/en/function.checkdnsrr.php

Chapter 57
Regex

Validates that a value matches a regular expression.

Applies to	property or method
Options	• pattern • htmlPattern • match • message • payload
Class	*Regex*[1]
Validator	*RegexValidator*[2]

Basic Usage

Suppose you have a `description` field and you want to verify that it begins with a valid word character. The regular expression to test for this would be `/^\w+/`, indicating that you're looking for at least one or more word characters at the beginning of your string:

Listing 57-1
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex("/^\w+/")
     */
    protected $description;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Regex.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/RegexValidator.html

Alternatively, you can set the match option to `false` in order to assert that a given string does *not* match. In the following example, you'll assert that the `firstName` field does not contain any numbers and give it a custom message:

Listing 57-2
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex(
     *     pattern="/\d/",
     *     match=false,
     *     message="Your name cannot contain a number"
     * )
     */
    protected $firstName;
}
```

Options

pattern

type: `string` [default option]

This required option is the regular expression pattern that the input will be matched against. By default, this validator will fail if the input string does *not* match this regular expression (via the *preg_match*[3] PHP function). However, if match is set to false, then validation will fail if the input string *does* match this pattern.

htmlPattern

type: `string|boolean` **default**: null

This option specifies the pattern to use in the HTML5 `pattern` attribute. You usually don't need to specify this option because by default, the constraint will convert the pattern given in the pattern option into an HTML5 compatible pattern. This means that the delimiters are removed (e.g. `/[a-z]+/` becomes `[a-z]+`).

However, there are some other incompatibilities between both patterns which cannot be fixed by the constraint. For instance, the HTML5 `pattern` attribute does not support flags. If you have a pattern like `/[a-z]+/i`, you need to specify the HTML5 compatible pattern in the `htmlPattern` option:

Listing 57-3
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex(
     *     pattern     = "/^[a-z]+$/i",
     *     htmlPattern = "^[a-zA-Z]+$"
     * )
     */
```

3. https://secure.php.net/manual/en/function.preg-match.php

```
14      protected $name;
15  }
```

Setting `htmlPattern` to false will disable client side validation.

match

type: `boolean` default: `true`

If `true` (or not set), this validator will pass if the given string matches the given pattern regular expression. However, when this option is set to `false`, the opposite will occur: validation will pass only if the given string does **not** match the pattern regular expression.

message

type: `string` **default**: `This value is not valid.`

This is the message that will be shown if this validator fails.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 58

Ip

Validates that a value is a valid IP address. By default, this will validate the value as IPv4, but a number of different options exist to validate as IPv6 and many other combinations.

Applies to	property or method
Options	• version • message • payload
Class	*Ip*[1]
Validator	*IpValidator*[2]

Basic Usage

Listing 58-1

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Ip
     */
     protected $ipAddress;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Ip.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IpValidator.html

Options

version

type: `string` **default**: `4`

This determines exactly *how* the IP address is validated and can take one of a variety of different values:

All ranges

`4`
: Validates for IPv4 addresses

`6`
: Validates for IPv6 addresses

`all`
: Validates all IP formats

No private ranges

`4_no_priv`
: Validates for IPv4 but without private IP ranges

`6_no_priv`
: Validates for IPv6 but without private IP ranges

`all_no_priv`
: Validates for all IP formats but without private IP ranges

No reserved ranges

`4_no_res`
: Validates for IPv4 but without reserved IP ranges

`6_no_res`
: Validates for IPv6 but without reserved IP ranges

`all_no_res`
: Validates for all IP formats but without reserved IP ranges

Only public ranges

`4_public`
: Validates for IPv4 but without private and reserved ranges

`6_public`
: Validates for IPv6 but without private and reserved ranges

`all_public`
: Validates for all IP formats but without private and reserved ranges

message

type: `string` **default**: `This is not a valid IP address.`

This message is shown if the string is not a valid IP address.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 59
Uuid

Validates that a value is a valid *Universally unique identifier (UUID)*[1] per *RFC 4122*[2]. By default, this will validate the format according to the RFC's guidelines, but this can be relaxed to accept non-standard UUIDs that other systems (like PostgreSQL) accept. UUID versions can also be restricted using a whitelist.

Applies to	property or method
Options	• message • strict • versions • payload
Class	*Uuid*[3]
Validator	*UuidValidator*[4]

Basic Usage

Listing 59-1
```yaml
# config/validator/validation.yaml
App\Entity\File:
    properties:
        identifier:
            - Uuid: ~
```

1. http://en.wikipedia.org/wiki/Universally_unique_identifier
2. http://tools.ietf.org/html/rfc4122
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Uuid.html
4. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/UuidValidator.html

Options

message

type: `string` **default**: `This is not a valid UUID.`

This message is shown if the string is not a valid UUID.

strict

type: `boolean` **default**: `true`

If this option is set to `true` the constraint will check if the UUID is formatted per the RFC's input format rules: `216fff40-98d9-11e3-a5e2-0800200c9a66`. Setting this to `false` will allow alternate input formats like:

- `216f-ff40-98d9-11e3-a5e2-0800-200c-9a66`
- `{216fff40-98d9-11e3-a5e2-0800200c9a66}`
- `216fff4098d911e3a5e20800200c9a66`

versions

type: `int[]` **default**: `[1,2,3,4,5]`

This option can be used to only allow specific *UUID versions*[5]. Valid versions are 1 - 5. The following PHP constants can also be used:

- `Uuid::V1_MAC`
- `Uuid::V2_DCE`
- `Uuid::V3_MD5`
- `Uuid::V4_RANDOM`
- `Uuid::V5_SHA1`

All five versions are allowed by default.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

5. `http://en.wikipedia.org/wiki/Universally_unique_identifier#Variants_and_versions`

Chapter 60
Range

Validates that a given number is *between* some minimum and maximum number.

Applies to	property or method
Options	• min • max • minMessage • maxMessage • invalidMessage • payload
Class	*Range*[1]
Validator	*RangeValidator*[2]

Basic Usage

To verify that the "height" field of a class is between "120" and "180", you might add the following:

Listing 60-1

```php
// src/Entity/Participant.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Range(
     *      min = 120,
     *      max = 180,
     *      minMessage = "You must be at least {{ limit }}cm tall to enter",
     *      maxMessage = "You cannot be taller than {{ limit }}cm to enter"
     * )
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Range.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/RangeValidator.html

```
15          */
16         protected $height;
17     }
```

Date Ranges

This constraint can be used to compare `DateTime` objects against date ranges. The minimum and maximum date of the range should be given as any date string *accepted by the DateTime constructor*[3]. For example, you could check that a date must lie within the current year like this:

Listing 60-2

```
1  // src/Entity/Event.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Event
7  {
8      /**
9       * @Assert\Range(
10      *      min = "first day of January",
11      *      max = "first day of January next year"
12      * )
13      */
14     protected $startDate;
15 }
```

Be aware that PHP will use the server's configured timezone to interpret these dates. If you want to fix the timezone, append it to the date string:

Listing 60-3

```
1  // src/Entity/Event.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Event
7  {
8      /**
9       * @Assert\Range(
10      *      min = "first day of January UTC",
11      *      max = "first day of January next year UTC"
12      * )
13      */
14     protected $startDate;
15 }
```

The `DateTime` class also accepts relative dates or times. For example, you can check that a delivery date starts within the next five hours like this:

Listing 60-4

```
1  // src/Entity/Order.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Order
7  {
8      /**
9       * @Assert\Range(
10      *      min = "now",
11      *      max = "+5 hours"
12      * )
13      */
```

3. https://php.net/manual/en/datetime.formats.php

```
14      protected $deliveryDate;
15  }
```

Options

min

type: `integer`

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

max

type: `integer`

This required option is the "max" value. Validation will fail if the given value is **greater** than this max value.

minMessage

type: `string` **default**: `This value should be {{ limit }} or more.`

The message that will be shown if the underlying value is less than the min option.

maxMessage

type: `string` **default**: `This value should be {{ limit }} or less.`

The message that will be shown if the underlying value is more than the max option.

invalidMessage

type: `string` **default**: `This value should be a valid number.`

The message that will be shown if the underlying value is not a number (per the *is_numeric*[4] PHP function).

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

4. https://php.net/manual/en/function.is-numeric.php

Chapter 61
EqualTo

Validates that a value is equal to another value, defined in the options. To force that a value is *not* equal, see *NotEqualTo*.

This constraint compares using `==`, so `3` and `"3"` are considered equal. Use *IdenticalTo* to compare with `===`.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*EqualTo*[1]
Validator	*EqualToValidator*[2]

Basic Usage

If you want to ensure that the `firstName` of a `Person` class is equal to `Mary` and that the `age` is `20`, you could do the following:

Listing 61-1

```
// src/Entity/Person.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/EqualTo.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/EqualToValidator.html

```
9         * @Assert\EqualTo("Mary")
10        */
11       protected $firstName;
12
13       /**
14        * @Assert\EqualTo(
15        *     value = 20
16        * )
17        */
18       protected $age;
19   }
```

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be equal to {{ compared_value }}.`

This is the message that will be shown if the value is not equal.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 62
NotEqualTo

Validates that a value is **not** equal to another value, defined in the options. To force that a value is equal, see *EqualTo*.

This constraint compares using `!=`, so `3` and `"3"` are considered equal. Use *NotIdenticalTo* to compare with `!==`.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*NotEqualTo*[1]
Validator	*NotEqualToValidator*[2]

Basic Usage

If you want to ensure that the `firstName` of a `Person` is not equal to `Mary` and that the `age` of a `Person` class is not `15`, you could do the following:

Listing 62-1

```
// src/Entity/Person.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotEqualTo.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotEqualToValidator.html

```
8       /**
9        * @Assert\NotEqualTo("Mary")
10       */
11      protected $firstName;
12
13      /**
14       * @Assert\NotEqualTo(
15       *     value = 15
16       * )
17       */
18      protected $age;
19  }
```

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should not be equal to {{ compared_value }}.`

This is the message that will be shown if the value is equal.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 63
IdenticalTo

Validates that a value is identical to another value, defined in the options. To force that a value is *not* identical, see *NotIdenticalTo*.

This constraint compares using `===`, so `3` and `"3"` are *not* considered equal. Use *EqualTo* to compare with `==`.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*IdenticalTo*[1]
Validator	*IdenticalToValidator*[2]

Basic Usage

The following constraints ensure that:

- `firstName` of `Person` class is equal to `Mary` *and* is a string
- `age` is equal to``20`` *and* is of type integer

Listing 63-1
```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IdenticalTo.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IdenticalToValidator.html

```
6  class Person
7  {
8      /**
9       * @Assert\IdenticalTo("Mary")
10      */
11     protected $firstName;
12
13     /**
14      * @Assert\IdenticalTo(
15      *     value = 20
16      * )
17      */
18     protected $age;
19 }
```

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be identical to {{ compared_value_type }} {{ compared_value }}.`

This is the message that will be shown if the value is not identical.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 64
NotIdenticalTo

Validates that a value is **not** identical to another value, defined in the options. To force that a value is identical, see *IdenticalTo*.

This constraint compares using `!==`, so `3` and `"3"` are considered not equal. Use *NotEqualTo* to compare with `!=`.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*NotIdenticalTo*[1]
Validator	*NotIdenticalToValidator*[2]

Basic Usage

The following constraints ensure that:

- `firstName` of `Person` is not equal to `Mary` *or* not of the same type
- `age` of `Person` class is not equal to `15` *or* not of the same type

Listing 64-1

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotIdenticalTo.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/NotIdenticalToValidator.html

```
6  class Person
7  {
8      /**
9       * @Assert\NotIdenticalTo("Mary")
10      */
11     protected $firstName;
12
13     /**
14      * @Assert\NotIdenticalTo(
15      *     value = 15
16      * )
17      */
18     protected $age;
19 }
```

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should not be identical to {{ compared_value_type }} {{ compared_value }}.`

This is the message that will be shown if the value is identical.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 65
LessThan

Validates that a value is less than another value, defined in the options. To force that a value is less than or equal to another value, see *LessThanOrEqual*. To force a value is greater than another value, see *GreaterThan*.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*LessThan*[1]
Validator	*LessThanValidator*[2]

Basic Usage

The following constraints ensure that:

- the number of `siblings` of a `Person` is less than `5`
- `age` is less than `80`

Listing 65-1
```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8
9      /**
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LessThan.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LessThanValidator.html

```
10          * @Assert\LessThan(5)
11          */
12         protected $siblings;
13
14         /**
15          * @Assert\LessThan(
16          *     value = 80
17          * )
18          */
19         protected $age;
20     }
```

Comparing Dates

This constraint can be used to compare `DateTime` objects against any date string *accepted by the DateTime constructor*[3]. For example, you could check that a date must be in the past like this:

Listing 65-2

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThan("today")
10      */
11     protected $dateOfBirth;
12 }
```

Be aware that PHP will use the server's configured timezone to interpret these dates. If you want to fix the timezone, append it to the date string:

Listing 65-3

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThan("today UTC")
10      */
11     protected $dateOfBirth;
12 }
```

The `DateTime` class also accepts relative dates or times. For example, you can check that a person must be at least 18 years old like this:

Listing 65-4

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThan("-18 years")
10      */
11     protected $dateOfBirth;
12 }
```

3. https://php.net/manual/en/datetime.formats.php

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be less than {{ compared_value }}.`

This is the message that will be shown if the value is not less than the comparison value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 66
LessThanOrEqual

Validates that a value is less than or equal to another value, defined in the options. To force that a value is less than another value, see *LessThan*.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*LessThanOrEqual*[1]
Validator	*LessThanOrEqualValidator*[2]

Basic Usage

The following constraints ensure that:

- the number of `siblings` of a `Person` is less than or equal to `5`
- the `age` is less than or equal to `80`

Listing 66-1

```
// src/Entity/Person.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\LessThanOrEqual(5)
     */
    protected $siblings;
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LessThanOrEqual.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LessThanOrEqualValidator.html

```
12
13      /**
14       * @Assert\LessThanOrEqual(
15       *     value = 80
16       * )
17       */
18      protected $age;
19  }
```

Comparing Dates

This constraint can be used to compare `DateTime` objects against any date string *accepted by the DateTime constructor*[3]. For example, you could check that a date must be today or in the past like this:

Listing 66-2

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThanOrEqual("today")
10      */
11     protected $age;
12 }
```

Be aware that PHP will use the server's configured timezone to interpret these dates. If you want to fix the timezone, append it to the date string:

Listing 66-3

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThanOrEqual("today UTC")
10      */
11     protected $age;
12 }
```

The `DateTime` class also accepts relative dates or times. For example, you can check that a person must be at least 18 years old like this:

Listing 66-4

```
1  // src/Entity/Person.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
8      /**
9       * @Assert\LessThanOrEqual("-18 years")
10      */
11     protected $age;
12 }
```

3. https://php.net/manual/en/datetime.formats.php

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be less than or equal to {{ compared_value }}.`

This is the message that will be shown if the value is not less than or equal to the comparison value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 67
GreaterThan

Validates that a value is greater than another value, defined in the options. To force that a value is greater than or equal to another value, see *GreaterThanOrEqual*. To force a value is less than another value, see *LessThan*.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*GreaterThan*[1]
Validator	*GreaterThanValidator*[2]

Basic Usage

The following constraints ensure that:

- the number of `siblings` of a `Person` is greater than `5`
- the `age` of a `Person` class is greater than `18`

Listing 67-1

```
// src/Entity/Person.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{

    /**
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/GreaterThan.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/GreaterThanValidator.html

```
     * @Assert\GreaterThan(5)
     */
    protected $siblings;

    /**
     * @Assert\GreaterThan(
     *     value = 18
     * )
     */
    protected $age;
}
```

Comparing Dates

This constraint can be used to compare `DateTime` objects against any date string *accepted by the DateTime constructor*[3]. For example, you could check that a date must at least be the next day:

Listing 67-2

```
// src/Entity/Order.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Order
{
    /**
     * @Assert\GreaterThan("today")
     */
    protected $deliveryDate;
}
```

Be aware that PHP will use the server's configured timezone to interpret these dates. If you want to fix the timezone, append it to the date string:

Listing 67-3

```
// src/Entity/Order.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Order
{
    /**
     * @Assert\GreaterThan("today UTC")
     */
    protected $deliveryDate;
}
```

The `DateTime` class also accepts relative dates or times. For example, you can check that the above delivery date starts at least five hours after the current time:

Listing 67-4

```
// src/Entity/Order.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Order
{
    /**
     * @Assert\GreaterThan("+5 hours")
     */
    protected $deliveryDate;
}
```

3. https://php.net/manual/en/datetime.formats.php

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be greater than {{ compared_value }}.`

This is the message that will be shown if the value is not greater than the comparison value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 68
GreaterThanOrEqual

Validates that a value is greater than or equal to another value, defined in the options. To force that a value is greater than another value, see *GreaterThan*.

Applies to	property or method
Options	• value • message • payload • propertyPath
Class	*GreaterThanOrEqual*[1]
Validator	*GreaterThanOrEqualValidator*[2]

Basic Usage

The following constraints ensure that:

- the number of `siblings` of a `Person` is greater than or equal to `5`
- the `age` of a `Person` class is greater than or equal to `18`

Listing 68-1

```
// src/Entity/Person.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\GreaterThanOrEqual(5)
     */
    protected $siblings;
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/GreaterThanOrEqual.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/GreaterThanOrEqualValidator.html

```
12
13      /**
14       * @Assert\GreaterThanOrEqual(
15       *     value = 18
16       * )
17       */
18      protected $age;
19  }
```

Comparing Dates

This constraint can be used to compare `DateTime` objects against any date string *accepted by the DateTime constructor*[3]. For example, you could check that a date must at least be the current day:

Listing 68-2

```
1  // src/Entity/Order.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Order
7  {
8      /**
9       * @Assert\GreaterThanOrEqual("today")
10      */
11     protected $deliveryDate;
12 }
```

Be aware that PHP will use the server's configured timezone to interpret these dates. If you want to fix the timezone, append it to the date string:

Listing 68-3

```
1  // src/Entity/Order.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Order
7  {
8      /**
9       * @Assert\GreaterThanOrEqual("today UTC")
10      */
11     protected $deliveryDate;
12 }
```

The `DateTime` class also accepts relative dates or times. For example, you can check that the above delivery date starts at least five hours after the current time:

Listing 68-4

```
1  // src/Entity/Order.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Order
7  {
8      /**
9       * @Assert\GreaterThanOrEqual("+5 hours")
10      */
11     protected $deliveryDate;
12 }
```

3. https://php.net/manual/en/datetime.formats.php

Options

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be greater than or equal to {{ compared_value }}.`

This is the message that will be shown if the value is not greater than or equal to the comparison value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

propertyPath

type: `string`

It defines the object property whose value is used to make the comparison.

For example, if you want to compare the `$endDate` property of some object with regard to the `$startDate` property of the same object, use `propertyPath="startDate"` in the comparison constraint of `$endDate`.

Chapter 69
Date

Validates that a value is a valid date, meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid YYYY-MM-DD format.

Applies to	property or method
Options	• message • payload
Class	*Date*[1]
Validator	*DateValidator*[2]

Basic Usage

Listing 69-1

```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Date()
     */
     protected $birthday;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Date.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/DateValidator.html

Options

message

type: `string` **default**: `This value is not a valid date.`

This message is shown if the underlying data is not a valid date.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 70
DateTime

Validates that a value is a valid "datetime", meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a specific format.

Applies to	property or method
Options	• format • message • payload
Class	*DateTime*[1]
Validator	*DateTimeValidator*[2]

Basic Usage

Listing 70-1

```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\DateTime()
     */
     protected $createdAt;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/DateTime.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/DateTimeValidator.html

Options

format

type: `string` **default**: `Y-m-d H:i:s`

This option allows to validate a custom date format. See *`DateTime::createFromFormat()`*[3] for formatting options.

message

type: `string` **default**: `This value is not a valid datetime.`

This message is shown if the underlying data is not a valid datetime.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

3. https://secure.php.net/manual/en/datetime.createfromformat.php

Chapter 71
Time

Validates that a value is a valid time, meaning an object implementing `DateTimeInterface` or a string (or an object that can be cast into a string) that follows a valid `HH:MM:SS` format.

Applies to	property or method
Options	• message • payload
Class	*Time*[1]
Validator	*TimeValidator*[2]

Basic Usage

Suppose you have an Event class, with a `startAt` field that is the time of the day when the event starts:

Listing 71-1

```
// src/Entity/Event.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Event
{
    /**
     * @Assert\Time()
     */
     protected $startsAt;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Time.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/TimeValidator.html

Options

message

type: `string` **default**: `This value is not a valid time.`

This message is shown if the underlying data is not a valid time.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 72
Choice

This constraint is used to ensure that the given value is one of a given set of *valid* choices. It can also be used to validate that each item in an array of items is one of those valid choices.

Applies to	property or method
Options	• choices • callback • multiple • min • max • message • multipleMessage • minMessage • maxMessage • strict • payload
Class	*Choice*[1]
Validator	*ChoiceValidator*[2]

Basic Usage

The basic idea of this constraint is that you supply it with an array of valid values (this can be done in several ways) and it validates that the value of the given property exists in that array.

If your valid choice list is simple, you can pass them in directly via the choices option:

Listing 72-1
```
1  // src/Entity/Author.php
2  namespace App\Entity;
3
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Choice.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/ChoiceValidator.html

```php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice({"New York", "Berlin", "Tokyo"})
     */
    protected $city;

    /**
     * @Assert\Choice(choices={"fiction", "non-fiction"}, message="Choose a valid genre.")
     */
    protected $genre;
}
```

Supplying the Choices with a Callback Function

You can also use a callback function to specify your options. This is useful if you want to keep your choices in some central location so that, for example, you can easily access those choices for validation or for building a select form element:

Listing 72-2
```php
// src/Entity/Author.php
namespace App\Entity;

class Author
{
    public static function getGenres()
    {
        return array('fiction', 'non-fiction');
    }
}
```

You can pass the name of this method to the callback option of the `Choice` constraint.

Listing 72-3
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(callback="getGenres")
     */
    protected $genre;
}
```

If the callback is stored in a different class and is static, for example `Util`, you can pass the class name and the method as an array.

Listing 72-4
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(callback={"Util", "getGenres"})
     */
    protected $genre;
}
```

Available Options

choices

type: `array` [default option]

A required option (unless callback is specified) - this is the array of options that should be considered in the valid set. The input value will be matched against this array.

callback

type: `string|array|Closure`

This is a callback method that can be used instead of the choices option to return the choices array. See Supplying the Choices with a Callback Function for details on its usage.

multiple

type: `boolean` **default**: `false`

If this option is true, the input value is expected to be an array instead of a single, scalar value. The constraint will check that each value of the input array can be found in the array of valid choices. If even one of the input values cannot be found, the validation will fail.

min

type: `integer`

If the `multiple` option is true, then you can use the `min` option to force at least XX number of values to be selected. For example, if `min` is 3, but the input array only contains 2 valid items, the validation will fail.

max

type: `integer`

If the `multiple` option is true, then you can use the `max` option to force no more than XX number of values to be selected. For example, if `max` is 3, but the input array contains 4 valid items, the validation will fail.

message

type: `string` **default**: `The value you selected is not a valid choice.`

This is the message that you will receive if the `multiple` option is set to `false` and the underlying value is not in the valid array of choices.

multipleMessage

type: `string` **default**: `One or more of the given values is invalid.`

This is the message that you will receive if the `multiple` option is set to `true` and one of the values on the underlying array being checked is not in the array of valid choices.

minMessage

type: `string` **default**: `You must select at least {{ limit }} choices.`

This is the validation error message that's displayed when the user chooses too few choices per the min option.

maxMessage

type: `string` **default**: `You must select at most {{ limit }} choices.`

This is the validation error message that's displayed when the user chooses too many options per the max option.

strict

type: `boolean` **default**: `true`

The validator will also check the type of the input value. Specifically, this value is passed to as the third argument to the PHP *in_array*[3] method when checking to see if a value is in the valid choices array.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

3. https://secure.php.net/manual/en/function.in-array.php

Chapter 73
Collection

This constraint is used when the underlying data is a collection (i.e. an array or an object that implements `Traversable` and `ArrayAccess`), but you'd like to validate different keys of that collection in different ways. For example, you might validate the `email` key using the `Email` constraint and the `inventory` key of the collection with the `Range` constraint.

This constraint can also make sure that certain collection keys are present and that extra keys are not present.

Applies to	property or method
Options	• fields • allowExtraFields • extraFieldsMessage • allowMissingFields • missingFieldsMessage • payload
Class	*Collection*[1]
Validator	*CollectionValidator*[2]

Basic Usage

The `Collection` constraint allows you to validate the different keys of a collection individually. Take the following example:

Listing 73-1
```
1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  class Author
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Collection.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CollectionValidator.html

```
{
    protected $profileData = array(
        'personal_email' => '...',
        'short_bio' => '...',
    );

    public function setProfileData($key, $value)
    {
        $this->profileData[$key] = $value;
    }
}
```

To validate that the `personal_email` element of the `profileData` array property is a valid email address and that the `short_bio` element is not blank but is no longer than 100 characters in length, you would do the following:

Listing 73-2

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Collection(
     *     fields = {
     *         "personal_email" = @Assert\Email,
     *         "short_bio" = {
     *             @Assert\NotBlank(),
     *             @Assert\Length(
     *                 max = 100,
     *                 maxMessage = "Your short bio is too long!"
     *             )
     *         }
     *     },
     *     allowMissingFields = true
     * )
     */
     protected $profileData = array(
         'personal_email' => '...',
         'short_bio' => '...',
     );
}
```

Presence and Absence of Fields

By default, this constraint validates more than simply whether or not the individual fields in the collection pass their assigned constraints. In fact, if any keys of a collection are missing or if there are any unrecognized keys in the collection, validation errors will be thrown.

If you would like to allow for keys to be absent from the collection or if you would like "extra" keys to be allowed in the collection, you can modify the allowMissingFields and allowExtraFields options respectively. In the above example, the `allowMissingFields` option was set to true, meaning that if either of the `personal_email` or `short_bio` elements were missing from the `$personalData` property, no validation error would occur.

Required and Optional Field Constraints

Constraints for fields within a collection can be wrapped in the `Required` or `Optional` constraint to control whether they should always be applied (`Required`) or only applied when the field is present (`Optional`).

For instance, if you want to require that the `personal_email` field of the `profileData` array is not blank and is a valid email but the `alternate_email` field is optional but must be a valid email if supplied, you can do the following:

Listing 73-3

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Collection(
     *     fields={
     *         "personal_email"  = @Assert\Required({@Assert\NotBlank, @Assert\Email}),
     *         "alternate_email" = @Assert\Optional(@Assert\Email)
     *     }
     * )
     */
     protected $profileData = array('personal_email');
}
```

Even without `allowMissingFields` set to true, you can now omit the `alternate_email` property completely from the `profileData` array, since it is `Optional`. However, if the `personal_email` field does not exist in the array, the `NotBlank` constraint will still be applied (since it is wrapped in `Required`) and you will receive a constraint violation.

Options

fields

type: `array` [default option]

This option is required and is an associative array defining all of the keys in the collection and, for each key, exactly which validator(s) should be executed against that element of the collection.

allowExtraFields

type: `boolean` **default**: false

If this option is set to `false` and the underlying collection contains one or more elements that are not included in the fields option, a validation error will be returned. If set to `true`, extra fields are ok.

extraFieldsMessage

type: `string` **default**: `This field was not expected.`

The message shown if allowExtraFields is false and an extra field is detected.

allowMissingFields

type: `boolean` **default**: false

If this option is set to `false` and one or more fields from the fields option are not present in the underlying collection, a validation error will be returned. If set to `true`, it's ok if some fields in the fields option are not present in the underlying collection.

missingFieldsMessage

type: `string` **default**: `This field is missing.`

The message shown if allowMissingFields is false and one or more fields are missing from the underlying collection.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 74
Count

Validates that a given collection's (i.e. an array or an object that implements Countable) element count is *between* some minimum and maximum value.

Applies to	property or method
Options	• min • max • minMessage • maxMessage • exactMessage • payload
Class	*Count*[1]
Validator	*CountValidator*[2]

Basic Usage

To verify that the `emails` array field contains between 1 and 5 elements you might add the following:

Listing 74-1
```php
// src/Entity/Participant.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Count(
     *      min = 1,
     *      max = 5,
     *      minMessage = "You must specify at least one email",
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Count.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CountValidator.html

```
13          *      maxMessage = "You cannot specify more than {{ limit }} emails"
14          * )
15          */
16         protected $emails = array();
17     }
```

Options

min

type: `integer`

This required option is the "min" count value. Validation will fail if the given collection elements count is **less** than this min value.

max

type: `integer`

This required option is the "max" count value. Validation will fail if the given collection elements count is **greater** than this max value.

minMessage

type: `string` **default**: `This collection should contain {{ limit }} elements or more.`

The message that will be shown if the underlying collection elements count is less than the min option.

maxMessage

type: `string` **default**: `This collection should contain {{ limit }} elements or less.`

The message that will be shown if the underlying collection elements count is more than the max option.

exactMessage

type: `string` **default**: `This collection should contain exactly {{ limit }} elements.`

The message that will be shown if min and max values are equal and the underlying collection elements count is not exactly this value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 75
UniqueEntity

Validates that a particular field (or fields) in a Doctrine entity is (are) unique. This is commonly used, for example, to prevent a new user to register using an email address that already exists in the system.

Applies to	class
Options	• fields • message • em • repositoryMethod • entityClass • errorPath • ignoreNull • payload
Class	*UniqueEntity*[1]
Validator	*UniqueEntityValidator*[2]

Basic Usage

Suppose you have a `User` entity that has an `email` field. You can use the `UniqueEntity` constraint to guarantee that the `email` field remains unique between all of the constraints in your user table:

Listing 75-1

```
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\ORM\Mapping as ORM;

// DON'T forget this use statement!!!
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

1. https://api.symfony.com/4.0/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntity.html
2. https://api.symfony.com/4.0/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntityValidator.html

```php

/**
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class Author
{
    /**
     * @var string $email
     *
     * @ORM\Column(name="email", type="string", length=255, unique=true)
     * @Assert\Email()
     */
    protected $email;

    // ...
}
```

Options

fields

type: `array` | `string` [default option]

This required option is the field (or list of fields) on which this entity should be unique. For example, if you specified both the `email` and `name` field in a single `UniqueEntity` constraint, then it would enforce that the combination value is unique (e.g. two users could have the same email, as long as they don't have the same name also).

If you need to require two fields to be individually unique (e.g. a unique `email` *and* a unique `username`), you use two `UniqueEntity` entries, each with a single field.

message

type: `string` **default**: `This value is already used.`

The message that's displayed when this constraint fails. This message is always mapped to the first field causing the violation, even when using multiple fields in the constraint.

Messages can include the `{{ value }}` placeholder to display a string representation of the invalid entity. If the entity doesn't define the `__toString()` method, the following generic value will be used: *"Object of class __CLASS__ identified by <comma separated IDs>"*

em

type: `string`

The name of the entity manager to use for making the query to determine the uniqueness. If it's left blank, the correct entity manager will be determined for this class. For that reason, this option should probably not need to be used.

repositoryMethod

type: `string` **default**: `findBy()`

The name of the repository method to use for making the query to determine the uniqueness. If it's left blank, the `findBy()` method will be used. This method should return a countable result.

entityClass

type: `string`

By default, the query performed to ensure the uniqueness uses the repository of the current class instance. However, in some cases, such as when using Doctrine inheritance mapping, you need to execute the query in a different repository. Use this option to define the fully-qualified class name (FQCN) of the Doctrine entity associated with the repository you want to use.

errorPath

type: `string` **default**: The name of the first field in fields

If the entity violates the constraint the error message is bound to the first field in fields. If there is more than one field, you may want to map the error message to another field.

Consider this example:

Listing 75-2

```
// src/Entity/Service.php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity(
 *     fields={"host", "port"},
 *     errorPath="port",
 *     message="This port is already in use on that host."
 * )
 */
class Service
{
    /**
     * @ORM\ManyToOne(targetEntity="Host")
     */
    public $host;

    /**
     * @ORM\Column(type="integer")
     */
    public $port;
}
```

Now, the message would be bound to the `port` field with this configuration.

ignoreNull

type: `boolean` **default**: `true`

If this option is set to `true`, then the constraint will allow multiple entities to have a `null` value for a field without failing validation. If set to `false`, only one `null` value is allowed - if a second entity also has a `null` value, validation would fail.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 76
Language

Validates that a value is a valid language *Unicode language identifier* (e.g. `fr` or `zh-Hant`).

Applies to	property or method
Options	• message • payload
Class	*Language*[1]
Validator	*LanguageValidator*[2]

Basic Usage

Listing 76-1

```
// src/Entity/User.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Language()
     */
     protected $preferredLanguage;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Language.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LanguageValidator.html

Options

message

type: `string` **default**: `This value is not a valid language.`

This message is shown if the string is not a valid language code.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 77
Locale

Validates that a value is a valid locale.

The "value" for each locale is either the two letter *ISO 639-1*[1] *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the *ISO 3166-1 alpha-2*[2] *country* code (e.g. `fr_FR` for French/France).

Applies to	property or method
Options	• message • payload
Class	*Locale*[3]
Validator	*LocaleValidator*[4]

Basic Usage

Listing 77-1

```
// src/Entity/User.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Locale()
     */
     protected $locale;
}
```

1. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
2. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Locale.html
4. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LocaleValidator.html

Options

message

type: `string` **default**: `This value is not a valid locale.`

This message is shown if the string is not a valid locale.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 78
Country

Validates that a value is a valid *ISO 3166-1 alpha-2*[1] country code.

Applies to	property or method
Options	• message • payload
Class	*Country*[2]
Validator	*CountryValidator*[3]

Basic Usage

Listing 78-1

```
// src/Entity/User.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Country()
     */
     protected $country;
}
```

1. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Country.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CountryValidator.html

Options

message

type: `string` **default**: `This value is not a valid country.`

This message is shown if the string is not a valid country code.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 79
File

Validates that a value is a valid "file", which can be one of the following:

- A string (or object with a `__toString()` method) path to an existing file;
- A valid *`File`*[1] object (including objects of class *`UploadedFile`*[2]).

This constraint is commonly used in forms with the *FileType* form field.

If the file you're validating is an image, try the *Image* constraint.

Applies to	property or method
Options	• maxSize • binaryFormat • mimeTypes • maxSizeMessage • mimeTypesMessage • disallowEmptyMessage • notFoundMessage • notReadableMessage • uploadIniSizeErrorMessage • uploadFormSizeErrorMessage • uploadErrorMessage • payload
Class	*`File`*[3]
Validator	*`FileValidator`*[4]

1. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/File/File.html
2. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/File/UploadedFile.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/File.html
4. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/FileValidator.html

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a *FileType* field. For example, suppose you're creating an author form where you can upload a "bio" PDF for the author. In your form, the `bioFile` property would be a `file` type. The `Author` class might look as follows:

Listing 79-1
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $bioFile;

    public function setBioFile(File $file = null)
    {
        $this->bioFile = $file;
    }

    public function getBioFile()
    {
        return $this->bioFile;
    }
}
```

To guarantee that the `bioFile` `File` object is valid and that it is below a certain file size and a valid PDF, add the following:

Listing 79-2
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\File(
     *     maxSize = "1024k",
     *     mimeTypes = {"application/pdf", "application/x-pdf"},
     *     mimeTypesMessage = "Please upload a valid PDF"
     * )
     */
    protected $bioFile;
}
```

The `bioFile` property is validated to guarantee that it is a real file. Its size and mime type are also validated because the appropriate options have been specified.

Options

maxSize

type: `mixed`

If set, the size of the underlying file must be below this file size in order to be valid. The size of the file can be given in one of the following formats:

Suffix	Unit Name	value	e.g.
	byte	1 byte	4096

Suffix	Unit Name	value	e.g.
k	kilobyte	1,000 bytes	200k
M	megabyte	1,000,000 bytes	2M
Ki	kibibyte	1,024 bytes	32Ki
Mi	mebibyte	1,048,576 bytes	8Mi

For more information about the difference between binary and SI prefixes, see *Wikipedia: Binary prefix*[5].

binaryFormat

type: `boolean` **default**: `null`

When `true`, the sizes will be displayed in messages with binary-prefixed units (KiB, MiB). When `false`, the sizes will be displayed with SI-prefixed units (kB, MB). When `null`, then the binaryFormat will be guessed from the value defined in the `maxSize` option.

For more information about the difference between binary and SI prefixes, see *Wikipedia: Binary prefix*[6].

mimeTypes

type: `array` or `string`

If set, the validator will check that the mime type of the underlying file is equal to the given mime type (if a string) or exists in the collection of given mime types (if an array).

You can find a list of existing mime types on the *IANA website*[7].

maxSizeMessage

type: `string` **default**: `The file is too large ({{ size }} {{ suffix }}). Allowed maximum size is {{ limit }} {{ suffix }}.`

The message displayed if the file is larger than the maxSize option.

mimeTypesMessage

type: `string` **default**: `The mime type of the file is invalid ({{ type }}). Allowed mime types are {{ types }}.`

The message displayed if the mime type of the file is not a valid mime type per the mimeTypes option.

disallowEmptyMessage

type: `string` **default**: `An empty file is not allowed.`

This constraint checks if the uploaded file is empty (i.e. 0 bytes). If it is, this message is displayed.

notFoundMessage

type: `string` **default**: `The file could not be found.`

5. http://en.wikipedia.org/wiki/Binary_prefix
6. http://en.wikipedia.org/wiki/Binary_prefix
7. http://www.iana.org/assignments/media-types/index.html

The message displayed if no file can be found at the given path. This error is only likely if the underlying value is a string path, as a `File` object cannot be constructed with an invalid file path.

notReadableMessage

type: `string` **default**: `The file is not readable.`

The message displayed if the file exists, but the PHP `is_readable()` function fails when passed the path to the file.

uploadIniSizeErrorMessage

type: `string` **default**: `The file is too large. Allowed maximum size is {{ limit }} {{ suffix }}.`

The message that is displayed if the uploaded file is larger than the `upload_max_filesize` `php.ini` setting.

uploadFormSizeErrorMessage

type: `string` **default**: `The file is too large.`

The message that is displayed if the uploaded file is larger than allowed by the HTML file input field.

uploadErrorMessage

type: `string` **default**: `The file could not be uploaded.`

The message that is displayed if the uploaded file could not be uploaded for some unknown reason, such as the file upload failed or it couldn't be written to disk.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 80
Image

The Image constraint works exactly like the *File* constraint, except that its mimeTypes and mimeTypesMessage options are automatically setup to work for image files specifically.

Additionally it has options so you can validate against the width and height of the image.

See the *File* constraint for the bulk of the documentation on this constraint.

Applies to	property or method
Options	• mimeTypes • minWidth • maxWidth • maxHeight • minHeight • maxRatio • minRatio • allowSquare • allowLandscape • allowPortrait • detectCorrupted • mimeTypesMessage • sizeNotDetectedMessage • maxWidthMessage • minWidthMessage • maxHeightMessage • minHeightMessage • maxRatioMessage • minRatioMessage • allowSquareMessage • allowLandscapeMessage • allowPortraitMessage • corruptedMessage • See *File* for inherited options

Class	*Image*[1]
Validator	*ImageValidator*[2]

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a *FileType* field. For example, suppose you're creating an author form where you can upload a "headshot" image for the author. In your form, the `headshot` property would be a `file` type. The `Author` class might look as follows:

Listing 80-1
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $headshot;

    public function setHeadshot(File $file = null)
    {
        $this->headshot = $file;
    }

    public function getHeadshot()
    {
        return $this->headshot;
    }
}
```

To guarantee that the `headshot` `File` object is a valid image and that it is between a certain size, add the following:

Listing 80-2
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Image(
     *     minWidth = 200,
     *     maxWidth = 400,
     *     minHeight = 200,
     *     maxHeight = 400
     * )
     */
    protected $headshot;
}
```

The `headshot` property is validated to guarantee that it is a real image and that it is between a certain width and height.

You may also want to guarantee the `headshot` image to be square. In this case you can disable portrait and landscape orientations as shown in the following code:

Listing 80-3
```php
// src/Entity/Author.php
namespace App\Entity;
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Image.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/ImageValidator.html

```
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      /**
9       * @Assert\Image(
10      *     allowLandscape = false,
11      *     allowPortrait = false
12      * )
13      */
14     protected $headshot;
15 }
```

You can mix all the constraint options to create powerful validation rules.

Options

This constraint shares all of its options with the *File* constraint. It does, however, modify two of the default option values and add several other options.

mimeTypes

type: `array` or `string` **default**: `image/*`

You can find a list of existing image mime types on the *IANA website*[3].

mimeTypesMessage

type: `string` **default**: `This file is not a valid image.`

minWidth

type: `integer`

If set, the width of the image file must be greater than or equal to this value in pixels.

maxWidth

type: `integer`

If set, the width of the image file must be less than or equal to this value in pixels.

minHeight

type: `integer`

If set, the height of the image file must be greater than or equal to this value in pixels.

maxHeight

type: `integer`

If set, the height of the image file must be less than or equal to this value in pixels.

3. http://www.iana.org/assignments/media-types/image/index.html

minPixels

type: `integer`

If set, the amount of pixels of the image file must be greater than or equal to this value.

maxPixels

type: `integer`

If set, the amount of pixels of the image file must be less than or equal to this value.

maxRatio

type: `float`

If set, the aspect ratio (`width / height`) of the image file must be less than or equal to this value.

minRatio

type: `float`

If set, the aspect ratio (`width / height`) of the image file must be greater than or equal to this value.

allowSquare

type: `Boolean` **default**: `true`

If this option is false, the image cannot be a square. If you want to force a square image, then set leave this option as its default `true` value and set allowLandscape and allowPortrait both to `false`.

allowLandscape

type: `Boolean` **default**: `true`

If this option is false, the image cannot be landscape oriented.

allowPortrait

type: `Boolean` **default**: `true`

If this option is false, the image cannot be portrait oriented.

detectCorrupted

type: `boolean` **default**: `false`

If this option is true, the image contents are validated to ensure that the image is not corrupted. This validation is done with PHP's *`imagecreatefromstring`*[4] function, which requires the *PHP GD extension*[5] to be enabled.

sizeNotDetectedMessage

type: `string` **default**: `The size of the image could not be detected.`

4. https://secure.php.net/manual/en/function.imagecreatefromstring.php
5. http://php.net/manual/en/book.image.php

If the system is unable to determine the size of the image, this error will be displayed. This will only occur when at least one of the size constraint options has been set.

maxWidthMessage

type: `string` **default**: `The image width is too big ({{ width }}px). Allowed maximum width is {{ max_width }}px.`

The error message if the width of the image exceeds maxWidth.

minWidthMessage

type: `string` **default**: `The image width is too small ({{ width }}px). Minimum width expected is {{ min_width }}px.`

The error message if the width of the image is less than minWidth.

maxHeightMessage

type: `string` **default**: `The image height is too big ({{ height }}px). Allowed maximum height is {{ max_height }}px.`

The error message if the height of the image exceeds maxHeight.

minHeightMessage

type: `string` **default**: `The image height is too small ({{ height }}px). Minimum height expected is {{ min_height }}px.`

The error message if the height of the image is less than minHeight.

maxPixelsMessage

type: `string` **default**: `The image has to many pixels ({{ pixels }} pixels). Maximum amount expected is {{ max_pixels }} pixels.`

The error message if the amount of pixels of the image exceeds maxPixels.

minPixelsMessage

type: `string` **default**: `The image has to few pixels ({{ pixels }} pixels). Minimum amount expected is {{ min_pixels }} pixels.`

The error message if the amount of pixels of the image is less than minPixels.

maxRatioMessage

type: `string` **default**: `The image ratio is too big ({{ ratio }}). Allowed maximum ratio is {{ max_ratio }}`

The error message if the aspect ratio of the image exceeds maxRatio.

minRatioMessage

type: `string` **default**: `The image ratio is too small ({{ ratio }}). Minimum ratio expected is {{ min_ratio }}`

The error message if the aspect ratio of the image is less than minRatio.

allowSquareMessage

type: `string` **default**: `The image is square ({{ width }}x{{ height }}px). Square images are not allowed`

The error message if the image is square and you set allowSquare to `false`.

allowLandscapeMessage

type: `string` **default**: `The image is landscape oriented ({{ width }}x{{ height }}px). Landscape oriented images are not allowed`

The error message if the image is landscape oriented and you set allowLandscape to `false`.

allowPortraitMessage

type: `string` **default**: `The image is portrait oriented ({{ width }}x{{ height }}px). Portrait oriented images are not allowed`

The error message if the image is portrait oriented and you set allowPortrait to `false`.

corruptedMessage

type: `string` **default**: `The image file is corrupted.`

The error message when the detectCorrupted option is enabled and the image is corrupted.

Chapter 81
CardScheme

This constraint ensures that a credit card number is valid for a given credit card company. It can be used to validate the number before trying to initiate a payment through a payment gateway.

Applies to	property or method
Options	• schemes • message • payload
Class	*CardScheme*[1]
Validator	*CardSchemeValidator*[2]

Basic Usage

To use the `CardScheme` validator, simply apply it to a property or method on an object that will contain a credit card number.

Listing 81-1
```php
// src/Entity/Transaction.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\CardScheme(
     *     schemes={"VISA"},
     *     message="Your credit card number is invalid."
     * )
     */
    protected $cardNumber;
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CardScheme.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CardSchemeValidator.html

Available Options

schemes

type: `mixed` [default option]

This option is required and represents the name of the number scheme used to validate the credit card number, it can either be a string or an array. Valid values are:

- `AMEX`
- `CHINA_UNIONPAY`
- `DINERS`
- `DISCOVER`
- `INSTAPAYMENT`
- `JCB`
- `LASER`
- `MAESTRO`
- `MASTERCARD`
- `VISA`

For more information about the used schemes, see *Wikipedia: Issuer identification number (IIN)*[3].

message

type: `string` **default**: `Unsupported card type or invalid card number.`

The message shown when the value does not pass the `CardScheme` check.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

3. `https://en.wikipedia.org/wiki/Bank_card_number#Issuer_identification_number_.28IIN.29`

Chapter 82
Currency

Validates that a value is a valid *3-letter ISO 4217*[1] currency name.

Applies to	property or method
Options	• message • payload
Class	*Currency*[2]
Validator	*CurrencyValidator*[3]

Basic Usage

If you want to ensure that the `currency` property of an `Order` is a valid currency, you could do the following:

Listing 82-1
```php
// src/Entity/Order.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Order
{
    /**
     * @Assert\Currency
     */
    protected $currency;
}
```

1. https://en.wikipedia.org/wiki/ISO_4217
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Currency.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CurrencyValidator.html

Options

message

type: `string` **default**: `This value is not a valid currency.`

This is the message that will be shown if the value is not a valid currency.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 83
Luhn

This constraint is used to ensure that a credit card number passes the *Luhn algorithm*[1]. It is useful as a first step to validating a credit card: before communicating with a payment gateway.

Applies to	property or method
Options	• message • payload
Class	*Luhn*[2]
Validator	*LuhnValidator*[3]

Basic Usage

To use the Luhn validator, simply apply it to a property on an object that will contain a credit card number.

Listing 83-1
```php
// src/Entity/Transaction.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\Luhn(message="Please check your credit card number.")
     */
    protected $cardNumber;
}
```

1. https://en.wikipedia.org/wiki/Luhn_algorithm
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Luhn.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/LuhnValidator.html

Available Options

message

type: `string` **default**: `Invalid card number.`

The default message supplied when the value does not pass the Luhn check.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 84

Iban

This constraint is used to ensure that a bank account number has the proper format of an *International Bank Account Number (IBAN)*[1]. IBAN is an internationally agreed means of identifying bank accounts across national borders with a reduced risk of propagating transcription errors.

Applies to	property or method
Options	• message • payload
Class	*Iban*[2]
Validator	*IbanValidator*[3]

Basic Usage

To use the Iban validator, simply apply it to a property on an object that will contain an International Bank Account Number.

Listing 84-1

```
// src/Entity/Transaction.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\Iban(
     *     message="This is not a valid International Bank Account Number (IBAN)."
     * )
     */
    protected $bankAccountNumber;
}
```

1. https://en.wikipedia.org/wiki/International_Bank_Account_Number
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Iban.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IbanValidator.html

Available Options

message

type: `string` **default**: `This is not a valid International Bank Account Number (IBAN).`

The default message supplied when the value does not pass the Iban check.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 85
Bic

This constraint is used to ensure that a value has the proper format of a *Business Identifier Code (BIC)*[1]. BIC is an internationally agreed means to uniquely identify both financial and non-financial institutions.

Applies to	property or method
Options	• message • payload
Class	*Bic*[2]
Validator	*BicValidator*[3]

Basic Usage

To use the Bic validator, simply apply it to a property on an object that will contain a Business Identifier Code (BIC).

Listing 85-1
```php
// src/Entity/Transaction.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\Bic()
     */
    protected $businessIdentifierCode;
}
```

1. https://en.wikipedia.org/wiki/Business_Identifier_Code
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Bic.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/BicValidator.html

Available Options

message

type: `string` **default**: `This is not a valid Business Identifier Code (BIC).`

The default message supplied when the value does not pass the BIC check.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 86

Isbn

This constraint validates that an *International Standard Book Number (ISBN)*[1] is either a valid ISBN-10 or a valid ISBN-13.

Applies to	property or method
Options	• type • message • isbn10Message • isbn13Message • bothIsbnMessage • payload
Class	*Isbn*[2]
Validator	*IsbnValidator*[3]

Basic Usage

To use the `Isbn` validator, simply apply it to a property or method on an object that will contain an ISBN.

Listing 86-1

```
// src/Entity/Book.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Book
{
    /**
     * @Assert\Isbn(
     *     type = "isbn10",
```

1. https://en.wikipedia.org/wiki/Isbn
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Isbn.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IsbnValidator.html

```
11          *     message = "This value is not  valid."
12          * )
13          */
14         protected $isbn;
15     }
```

Available Options

type

type: `string` **default**: `null`

The type of ISBN to validate against. Valid values are `isbn10`, `isbn13` and `null` to accept any kind of ISBN.

message

type: `string` **default**: `null`

The message that will be shown if the value is not valid. If not `null`, this message has priority over all the other messages.

isbn10Message

type: `string` **default**: `This value is not a valid ISBN-10.`

The message that will be shown if the type option is `isbn10` and the given value does not pass the ISBN-10 check.

isbn13Message

type: `string` **default**: `This value is not a valid ISBN-13.`

The message that will be shown if the type option is `isbn13` and the given value does not pass the ISBN-13 check.

bothIsbnMessage

type: `string` **default**: `This value is neither a valid ISBN-10 nor a valid ISBN-13.`

The message that will be shown if the type option is `null` and the given value does not pass any of the ISBN checks.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 87

Issn

Validates that a value is a valid *International Standard Serial Number (ISSN)*[1].

Applies to	property or method
Options	• message • caseSensitive • requireHyphen • payload
Class	*Issn*[2]
Validator	*IssnValidator*[3]

Basic Usage

Listing 87-1

```
// src/Entity/Journal.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Journal
{
    /**
     * @Assert\Issn
     */
     protected $issn;
}
```

1. https://en.wikipedia.org/wiki/Issn
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Issn.html
3. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/IssnValidator.html

Options

message

type: `String` default: `This value is not a valid ISSN.`

The message shown if the given value is not a valid ISSN.

caseSensitive

type: `boolean` default: `false`

The validator will allow ISSN values to end with a lower case 'x' by default. When switching this to `true`, the validator requires an upper case 'X'.

requireHyphen

type: `boolean` default: `false`

The validator will allow non hyphenated ISSN values by default. When switching this to `true`, the validator requires a hyphenated ISSN value.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 88
Callback

The purpose of the Callback constraint is to create completely custom validation rules and to assign any validation errors to specific fields on your object. If you're using validation with forms, this means that you can make these custom errors display next to a specific field, instead of simply at the top of your form.

This process works by specifying one or more *callback* methods, each of which will be called during the validation process. Each of those methods can do anything, including creating and assigning validation errors.

A callback method itself doesn't *fail* or return any value. Instead, as you'll see in the example, a callback method has the ability to directly add validator "violations".

Applies to	class
Options	• callback • payload
Class	*Callback*[1]
Validator	*CallbackValidator*[2]

Configuration

Listing 88-1

```
1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5  use Symfony\Component\Validator\Context\ExecutionContextInterface;
6
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Callback.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/CallbackValidator.html

```php
class Author
{
    /**
     * @Assert\Callback
     */
    public function validate(ExecutionContextInterface $context, $payload)
    {
        // ...
    }
}
```

The Callback Method

The callback method is passed a special `ExecutionContextInterface` object. You can set "violations" directly on this object and determine to which field those errors should be attributed:

Listing 88-2

```php
// ...
use Symfony\Component\Validator\Context\ExecutionContextInterface;

class Author
{
    // ...
    private $firstName;

    public function validate(ExecutionContextInterface $context, $payload)
    {
        // somehow you have an array of "fake names"
        $fakeNames = array(/* ... */);

        // check if the name is actually a fake name
        if (in_array($this->getFirstName(), $fakeNames)) {
            $context->buildViolation('This name sounds totally fake!')
                ->atPath('firstName')
                ->addViolation();
        }
    }
}
```

Static Callbacks

You can also use the constraint with static methods. Since static methods don't have access to the object instance, they receive the object as the first argument:

Listing 88-3

```php
public static function validate($object, ExecutionContextInterface $context, $payload)
{
    // somehow you have an array of "fake names"
    $fakeNames = array(/* ... */);

    // check if the name is actually a fake name
    if (in_array($object->getFirstName(), $fakeNames)) {
        $context->buildViolation('This name sounds totally fake!')
            ->atPath('firstName')
            ->addViolation()
        ;
    }
}
```

External Callbacks and Closures

If you want to execute a static callback method that is not located in the class of the validated object, you can configure the constraint to invoke an array callable as supported by PHP's *call_user_func*[3] function. Suppose your validation function is `Acme\Validator::validate()`:

Listing 88-4
```php
namespace Acme;

use Symfony\Component\Validator\Context\ExecutionContextInterface;

class Validator
{
    public static function validate($object, ExecutionContextInterface $context, $payload)
    {
        // ...
    }
}
```

You can then use the following configuration to invoke this validator:

Listing 88-5
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Callback({"Acme\Validator", "validate"})
 */
class Author
{
}
```

The Callback constraint does *not* support global callback functions nor is it possible to specify a global function or a service method as callback. To validate using a service, you should *create a custom validation constraint* and add that new constraint to your class.

When configuring the constraint via PHP, you can also pass a closure to the constructor of the Callback constraint:

Listing 88-6
```php
// src/Entity/Author.php
namespace App\Entity;

use Symfony\Component\Validator\Context\ExecutionContextInterface;

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $callback = function ($object, ExecutionContextInterface $context, $payload) {
            // ...
        };

        $metadata->addConstraint(new Assert\Callback($callback));
    }
}
```

3. https://secure.php.net/manual/en/function.call-user-func.php

Options

callback

type: `string`, `array` or `Closure` [default option]

The callback option accepts three different formats for specifying the callback method:

- A **string** containing the name of a concrete or static method;
- An array callable with the format `array('<Class>', '<method>')`;
- A closure.

Concrete callbacks receive an *`ExecutionContextInterface`*[4] instance as only argument.

Static or closure callbacks receive the validated object as the first argument and the *`ExecutionContextInterface`*[5] instance as the second argument.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

4. https://api.symfony.com/4.0/Symfony/Component/Validator/Context/ExecutionContextInterface.html
5. https://api.symfony.com/4.0/Symfony/Component/Validator/ExecutionContextInterface.html

Chapter 89
Expression

This constraint allows you to use an expression for more complex, dynamic validation. See Basic Usage for an example. See *Callback* for a different constraint that gives you similar flexibility.

Applies to	class or property/method
Options	• expression • message • payload
Class	*Expression*[1]
Validator	*ExpressionValidator*[2]

Basic Usage

Imagine you have a class `BlogPost` with `category` and `isTechnicalPost` properties:

Listing 89-1

```
// src/Model/BlogPost.php
namespace App\Model;

use Symfony\Component\Validator\Constraints as Assert;

class BlogPost
{
    private $category;

    private $isTechnicalPost;

    // ...

    public function getCategory()
    {
        return $this->category;
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Expression.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/ExpressionValidator.html

```
    }

    public function setIsTechnicalPost($isTechnicalPost)
    {
        $this->isTechnicalPost = $isTechnicalPost;
    }

    // ...
}
```

To validate the object, you have some special requirements:

1. If `isTechnicalPost` is true, then `category` must be either `php` or `symfony`;
2. If `isTechnicalPost` is false, then `category` can be anything.

One way to accomplish this is with the Expression constraint:

Listing 89-2

```
// src/Model/BlogPost.php
namespace App\Model;

use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Expression(
 *     "this.getCategory() in ['php', 'symfony'] or !this.isTechnicalPost()",
 *     message="If this is a tech post, the category should be either php or symfony!"
 * )
 */
class BlogPost
{
    // ...
}
```

The expression option is the expression that must return true in order for validation to pass. To learn more about the expression language syntax, see *The Expression Syntax*.

Mapping the Error to a Specific Field

You can also attach the constraint to a specific property and still validate based on the values of the entire entity. This is handy if you want to attach the error to a specific field. In this context, `value` represents the value of `isTechnicalPost`.

Listing 89-3

```
// src/Model/BlogPost.php
namespace App\Model;

use Symfony\Component\Validator\Constraints as Assert;

class BlogPost
{
    // ...

    /**
     * @Assert\Expression(
     *     "this.getCategory() in ['php', 'symfony'] or value == false",
     *     message="If this is a tech post, the category should be either php or symfony!"
     * )
     */
    private $isTechnicalPost;

    // ...
}
```

For more information about the expression and what variables are available to you, see the expression option details below.

Available Options

expression

type: `string` [default option]

The expression that will be evaluated. If the expression evaluates to a false value (using `==`, not `===`), validation will fail.

To learn more about the expression language syntax, see *The Expression Syntax*.

Inside of the expression, you have access to up to 2 variables:

Depending on how you use the constraint, you have access to 1 or 2 variables in your expression:

- `this`: The object being validated (e.g. an instance of BlogPost);
- `value`: The value of the property being validated (only available when the constraint is applied directly to a property);

message

type: `string` **default**: `This value is not valid.`

The default message supplied when the expression evaluates to false.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 90

All

When applied to an array (or Traversable object), this constraint allows you to apply a collection of constraints to each element of the array.

Applies to	property or method
Options	• constraints • payload
Class	*All*[1]
Validator	*AllValidator*[2]

Basic Usage

Suppose that you have an array of strings and you want to validate each entry in that array:

Listing 90-1

```
// src/Entity/User.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\All({
     *     @Assert\NotBlank,
     *     @Assert\Length(min=5)
     * })
     */
     protected $favoriteColors = array();
}
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/All.html
2. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/AllValidator.html

Now, each entry in the `favoriteColors` array will be validated to not be blank and to be at least 5 characters long.

Options

constraints

type: `array` [default option]

This required option is the array of validation constraints that you want to apply to each element of the underlying array.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 91
UserPassword

This validates that an input value is equal to the current authenticated user's password. This is useful in a form where a user can change their password, but needs to enter their old password for security.

This should **not** be used to validate a login form, since this is done automatically by the security system.

Applies to	property or method
Options	• message • payload
Class	*UserPassword*[1]
Validator	*UserPasswordValidator*[2]

Basic Usage

Suppose you have a `ChangePassword` class, that's used in a form where the user can change their password by entering their old password and a new password. This constraint will validate that the old password matches the user's current password:

Listing 91-1
```
1  // src/Form/Model/ChangePassword.php
2  namespace App\Form\Model;
3
4  use Symfony\Component\Security\Core\Validator\Constraints as SecurityAssert;
5
6  class ChangePassword
7  {
8      /**
9       * @SecurityAssert\UserPassword(
```

1. https://api.symfony.com/4.0/Symfony/Component/Security/Core/Validator/Constraints/UserPassword.html
2. https://api.symfony.com/4.0/Symfony/Component/Security/Core/Validator/Constraints/UserPasswordValidator.html

```
10          *     message = "Wrong value for your current password"
11          * )
12          */
13         protected $oldPassword;
14     }
```

Options

message

type: `message` **default**: `This value should be the user current password.`

This is the message that's displayed when the underlying string does *not* match the current user's password.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 92
Valid

This constraint is used to enable validation on objects that are embedded as properties on an object being validated. This allows you to validate an object and all sub-objects associated with it.

Applies to	property or method
Options	• traverse • payload
Class	*Valid*[1]

By default the `error_bubbling` option is enabled for the *collection Field Type*, which passes the errors to the parent form. If you want to attach the errors to the locations where they actually occur you have to set `error_bubbling` to `false`.

Basic Usage

In the following example, create two classes `Author` and `Address` that both have constraints on their properties. Furthermore, `Author` stores an `Address` instance in the `$address` property:

Listing 92-1
```php
// src/Entity/Address.php
namespace App\Entity;

class Address
{
    protected $street;
    protected $zipCode;
}
```

Listing 92-2
```php
// src/Entity/Author.php
namespace App\Entity;
```

1. https://api.symfony.com/4.0/Symfony/Component/Validator/Constraints/Valid.html

```
3
4  class Author
5  {
6      protected $firstName;
7      protected $lastName;
8      protected $address;
9  }
```

Listing 92-3

```
1  // src/Entity/Address.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Address
7  {
8      /**
9       * @Assert\NotBlank()
10      */
11     protected $street;
12
13     /**
14      * @Assert\NotBlank
15      * @Assert\Length(max=5)
16      */
17     protected $zipCode;
18 }
19
20 // src/Entity/Author.php
21 namespace App\Entity;
22
23 use Symfony\Component\Validator\Constraints as Assert;
24
25 class Author
26 {
27     /**
28      * @Assert\NotBlank
29      * @Assert\Length(min=4)
30      */
31     protected $firstName;
32
33     /**
34      * @Assert\NotBlank
35      */
36     protected $lastName;
37
38     protected $address;
39 }
```

With this mapping, it is possible to successfully validate an author with an invalid address. To prevent that, add the `Valid` constraint to the `$address` property.

Listing 92-4

```
1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      /**
9       * @Assert\Valid
10      */
11     protected $address;
12 }
```

If you validate an author with an invalid address now, you can see that the validation of the `Address` fields failed.

Listing 92-5

```
App\Entity\Author.address.zipCode:
    This value is too long. It should have 5 characters or less.
```

Options

traverse

type: `boolean` **default**: `true`

If this constraint is applied to a property that holds an array of objects, then each object in that array will be validated only if this option is set to `true`.

payload

type: `mixed` **default**: `null`

This option can be used to attach arbitrary domain-specific data to a constraint. The configured payload is not used by the Validator component, but its processing is completely up to you.

For example, you may want to use *several error levels* to present failed constraints differently in the front-end depending on the severity of the error.

Chapter 93

Twig Template Form Function and Variable Reference

When working with forms in a template, there are two powerful things at your disposal:

- Functions for rendering each part of a form;
- Variables for getting *any* information about any field.

You'll use functions often to render your fields. Variables, on the other hand, are less commonly-used, but infinitely powerful since you can access a fields label, id attribute, errors and anything else about the field.

Form Rendering Functions

This reference manual covers all the possible Twig functions available for rendering forms. There are several different functions available and each is responsible for rendering a different part of a form (e.g. labels, errors, widgets, etc).

form(view, variables)

Renders the HTML of a complete form.

Listing 93-1
```
{# render the form and change the submission method #}
{{ form(form, {'method': 'GET'}) }}
```

You will mostly use this helper for prototyping or if you use custom form themes. If you need more flexibility in rendering the form, you should use the other helpers to render individual parts of the form instead:

Listing 93-2
```
{{ form_start(form) }}
    {{ form_errors(form) }}

```

```
    {{ form_row(form.name) }}
    {{ form_row(form.dueDate) }}

    {{ form_row(form.submit, { 'label': 'Submit me' }) }}
{{ form_end(form) }}
```

form_start(view, variables)

Renders the start tag of a form. This helper takes care of printing the configured method and target action of the form. It will also include the correct `enctype` property if the form contains upload fields.

Listing 93-3
```
{# render the start tag and change the submission method #}
{{ form_start(form, {'method': 'GET'}) }}
```

form_end(view, variables)

Renders the end tag of a form.

Listing 93-4
```
{{ form_end(form) }}
```

This helper also outputs `form_rest()` unless you set `render_rest` to false:

Listing 93-5
```
{# don't render unrendered fields #}
{{ form_end(form, {'render_rest': false}) }}
```

form_label(view, label, variables)

Renders the label for the given field. You can optionally pass the specific label you want to display as the second argument.

Listing 93-6
```
{{ form_label(form.name) }}

{# The two following syntaxes are equivalent #}
{{ form_label(form.name, 'Your Name', {'label_attr': {'class': 'foo'}}) }}

{{ form_label(form.name, null, {
    'label': 'Your name',
    'label_attr': {'class': 'foo'}
}) }}
```

See "More about Form Variables" to learn about the `variables` argument.

form_errors(view)

Renders any errors for the given field.

Listing 93-7
```
{{ form_errors(form.name) }}

{# render any "global" errors #}
{{ form_errors(form) }}
```

form_widget(view, variables)

Renders the HTML widget of a given field. If you apply this to an entire form or collection of fields, each underlying form row will be rendered.

Listing 93-8
```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, {'attr': {'class': 'foo'}}) }}
```

The second argument to `form_widget()` is an array of variables. The most common variable is `attr`, which is an array of HTML attributes to apply to the HTML widget. In some cases, certain types also have other template-related options that can be passed. These are discussed on a type-by-type basis. The `attributes` are not applied recursively to child fields if you're rendering many fields at once (e.g. `form_widget(form)`).

See "More about Form Variables" to learn more about the `variables` argument.

form_row(view, variables)

Renders the "row" of a given field, which is the combination of the field's label, errors and widget.

Listing 93-9
```
{# render a field row, but display a label with text "foo" #}
{{ form_row(form.name, {'label': 'foo'}) }}
```

The second argument to `form_row()` is an array of variables. The templates provided in Symfony only allow to override the label as shown in the example above.

See "More about Form Variables" to learn about the `variables` argument.

form_rest(view, variables)

This renders all fields that have not yet been rendered for the given form. It's a good idea to always have this somewhere inside your form as it'll render hidden fields for you and make any fields you forgot to render more obvious (since it'll render the field for you).

Listing 93-10
```
{{ form_rest(form) }}
```

Form Tests Reference

Tests can be executed by using the `is` operator in Twig to create a condition. Read *the Twig documentation*[1] for more information.

selectedchoice(selected_value)

This test will check if the current choice is equal to the `selected_value` or if the current choice is in the array (when `selected_value` is an array).

Listing 93-11
```
<option {% if choice is selectedchoice(value) %} selected="selected"{% endif %} ...>
```

1. http://twig.sensiolabs.org/doc/templates.html#test-operator

More about Form Variables

For a full list of variables, see: Form Variables Reference.

In almost every Twig function above, the final argument is an array of "variables" that are used when rendering that one part of the form. For example, the following would render the "widget" for a field and modify its attributes to include a special class:

Listing 93-12
```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

The purpose of these variables - what they do & where they come from - may not be immediately clear, but they're incredibly powerful. Whenever you render any part of a form, the block that renders it makes use of a number of variables. By default, these blocks live inside *form_div_layout.html.twig*[2].

Look at the `form_label` as an example:

Listing 93-13
```
{% block form_label %}
    {% if not compound %}
        {% set label_attr = label_attr|merge({'for': id}) %}
    {% endif %}

    {% if required %}
        {% set label_attr = label_attr|merge({
            'class': (label_attr.class|default('') ~ ' required')|trim
        }) %}
    {% endif %}

    {% if label is empty %}
        {% set label = name|humanize %}
    {% endif %}

    <label
        {% for attrname, attrvalue in label_attr -%}
            {{ attrname }}="{{ attrvalue }}"
        {%- endfor %}
    >
        {{ label|trans({}, translation_domain) }}
    </label>
{% endblock form_label %}
```

This block makes use of several variables: `compound`, `label_attr`, `required`, `label`, `name` and `translation_domain`. These variables are made available by the form rendering system. But more importantly, these are the variables that you can override when calling `form_label()` (since in this example, you're rendering the label).

The exact variables available to override depends on which part of the form you're rendering (e.g. label versus widget) and which field you're rendering (e.g. a `choice` widget has an extra `expanded` option). If you get comfortable with looking through *form_div_layout.html.twig*[3], you'll always be able to see what options you have available.

Behind the scenes, these variables are made available to the `FormView` object of your form when the Form component calls `buildView()` and `finishView()` on each "node" of your form tree. To see what "view" variables a particular field has, find the source code for the form field (and its parent fields) and look at the above two functions.

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

If you're rendering an entire form at once (or an entire embedded form), the `variables` argument will only be applied to the form itself and not its children. In other words, the following will **not** pass a "foo" class attribute to all of the child fields in the form:

Listing 93-14
```
{# does **not** work - the variables are not recursive #}
{{ form_widget(form, { 'attr': {'class': 'foo'} }) }}
```

Form Variables Reference

The following variables are common to every field type. Certain field types may have even more variables and some variables here only really apply to certain types.

Assuming you have a `form` variable in your template and you want to reference the variables on the `name` field, accessing the variables is done by using a public `vars` property on the *FormView*[4] object:

Listing 93-15
```
<label for="{{ form.name.vars.id }}"
    class="{{ form.name.vars.required ? 'required' : '' }}">
    {{ form.name.vars.label }}
</label>
```

Variable	Usage
`form`	The current `FormView` instance.
`id`	The `id` HTML attribute to be rendered.
`name`	The name of the field (e.g. `title`) - but not the `name` HTML attribute, which is `full_name`.
`full_name`	The `name` HTML attribute to be rendered.
`errors`	An array of any errors attached to *this* specific field (e.g. `form.title.errors`). Note that you can't use `form.errors` to determine if a form is valid, since this only returns "global" errors: some individual fields may have errors. Instead, use the `valid` option.
`submitted`	Returns `true` or `false` depending on whether the whole form is submitted
`valid`	Returns `true` or `false` depending on whether the whole form is valid.
`value`	The value that will be used when rendering (commonly the `value` HTML attribute).
`disabled`	If `true`, `disabled="disabled"` is added to the field.
`required`	If `true`, a `required` attribute is added to the field to activate HTML5 validation. Additionally, a `required` class is added to the label.
`label`	The string label that will be rendered.
`multipart`	If `true`, `form_enctype` will render `enctype="multipart/form-data"`. This only applies to the root form element.
`attr`	A key-value array that will be rendered as HTML attributes on the field.
`label_attr`	A key-value array that will be rendered as HTML attributes on the label.
`compound`	Whether or not a field is actually a holder for a group of children fields (for example, a `choice` field, which is actually a group of checkboxes.

4. https://api.symfony.com/4.0/Symfony/Component/Form/FormView.html

Variable	Usage
`block_prefixes`	An array of all the names of the parent types.
`translation_domain`	The domain of the translations for this form.
`cache_key`	A unique key which is used for caching.
`data`	The normalized data of the type.
`method`	The method of the current form (POST, GET, etc.).
`action`	The action of the current form.

Chapter 94
Symfony Twig Extensions

Twig is the default template engine for Symfony. By itself, it already contains a lot of built-in functions, filters, tags and tests. You can learn more about them from the *Twig Reference*[1].

The Symfony framework adds quite a few extra functions, filters, tags and tests to seamlessly integrate the various Symfony components with Twig templates. The following sections describe these extra features.

Technically, most of the extensions live in the *Twig Bridge*[2]. That code might give you some ideas when you need to write your own Twig extension as described in *How to Write a custom Twig Extension*.

This reference only covers the Twig extensions provided by the Symfony framework. You are probably using some other bundles as well, and those might come with their own extensions not covered here.

The *Twig Extensions repository*[3] contains some additional Twig extensions that do not belong to the Twig core, so you might want to have a look at the *Twig Extensions documentation*[4].

Functions

render

Listing 94-1
```
1  {{ render(uri, options = []) }}
```

uri

type: `string` | `ControllerReference`

1. http://twig.sensiolabs.org/documentation#reference
2. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/Twig/Extension
3. https://github.com/twigphp/Twig-extensions
4. http://twig-extensions.readthedocs.io/en/latest/

`options` ***(optional)***
type: `array` **default**: `[]`

Renders the fragment for the given controller (using the controller function) or URI. For more information, see *How to Embed Controllers in a Template*.

The render strategy can be specified in the `strategy` key of the options.

The URI can be generated by other functions, like path and url.

render_esi

Listing 94-2
```
{{ render_esi(uri, options = []) }}
```

`uri`
type: `string` | `ControllerReference`

`options` ***(optional)***
type: `array` **default**: `[]`

Generates an ESI tag when possible or falls back to the behavior of render function instead. For more information, see *How to Embed Controllers in a Template*.

The URI can be generated by other functions, like path and url.

The `render_esi()` function is an example of the shortcut functions of `render`. It automatically sets the strategy based on what's given in the function name, e.g. `render_hinclude()` will use the hinclude.js strategy. This works for all `render_*()` functions.

controller

Listing 94-3
```
{{ controller(controller, attributes = [], query = []) }}
```

`controller`
type: `string`

`attributes` ***(optional)***
type: `array` **default**: `[]`

`query` ***(optional)***
type: `array` **default**: `[]`

Returns an instance of `ControllerReference` to be used with functions like render() and render_esi().

asset

Listing 94-4
```
{{ asset(path, packageName = null) }}
```

`path`
: **type**: `string`

`packageName` ***(optional)***
: **type**: `string` | `null` **default**: `null`

Returns a public path to `path`, which takes into account the base path set for the package and the URL path. More information in Linking to Assets. Symfony provides various cache busting implementations via the version, version_strategy, and json_manifest_path configuration options.

asset_version

Listing 94-5
```
1  {{ asset_version(packageName = null) }}
```

`packageName` ***(optional)***
: **type**: `string` | `null` **default**: `null`

Returns the current version of the package, more information in Linking to Assets.

form

Listing 94-6
```
1  {{ form(view, variables = []) }}
```

`view`
: **type**: `FormView`

`variables` ***(optional)***
: **type**: `array` **default**: `[]`

Renders the HTML of a complete form, more information in the Twig Form reference.

form_start

Listing 94-7
```
1  {{ form_start(view, variables = []) }}
```

`view`
: **type**: `FormView`

`variables` ***(optional)***
: **type**: `array` **default**: `[]`

Renders the HTML start tag of a form, more information in the Twig Form reference.

form_end

Listing 94-8
```
1  {{ form_end(view, variables = []) }}
```

`view`
: **type**: `FormView`

`variables` ***(optional)***
: **type**: `array` **default**: `[]`

Renders the HTML end tag of a form together with all fields that have not been rendered yet, more information in the Twig Form reference.

form_widget

Listing 94-9
```
{{ form_widget(view, variables = []) }}
```

`view`
type: `FormView`

`variables` ***(optional)***
type: `array` **default**: `[]`

Renders a complete form or a specific HTML widget of a field, more information in the Twig Form reference.

form_errors

Listing 94-10
```
{{ form_errors(view) }}
```

`view`
type: `FormView`

Renders any errors for the given field or the global errors, more information in the Twig Form reference.

form_label

Listing 94-11
```
{{ form_label(view, label = null, variables = []) }}
```

`view`
type: `FormView`

`label` ***(optional)***
type: `string` **default**: `null`

`variables` ***(optional)***
type: `array` **default**: `[]`

Renders the label for the given field, more information in the Twig Form reference.

form_row

Listing 94-12
```
{{ form_row(view, variables = []) }}
```

`view`
type: `FormView`

`variables` ***(optional)***
type: `array` **default**: `[]`

Renders the row (the field's label, errors and widget) of the given field, more information in the Twig Form reference.

form_rest

Listing 94-13
```
{{ form_rest(view, variables = []) }}
```

`view`
type: `FormView`

`variables` ***(optional)***
type: `array` **default**: `[]`

Renders all fields that have not yet been rendered, more information in the Twig Form reference.

csrf_token

Listing 94-14
```
{{ csrf_token(intention) }}
```

`intention`
type: `string`

Renders a CSRF token. Use this function if you want CSRF protection without creating a form.

is_granted

Listing 94-15
```
{{ is_granted(role, object = null, field = null) }}
```

`role`
type: `string`

`object` ***(optional)***
type: `object`

`field` ***(optional)***
type: `string`

Returns `true` if the current user has the required role. Optionally, an object can be pasted to be used by the voter. More information can be found in Access Control in Templates.

logout_path

Listing 94-16
```
{{ logout_path(key = null) }}
```

`key` ***(optional)***
type: `string`

Generates a relative logout URL for the given firewall. If no key is provided, the URL is generated for the current firewall the user is logged into.

logout_url

Listing 94-17
```
{{ logout_url(key = null) }}
```

`key` ***(optional)***
type: `string`

Equal to the logout_path function, but it'll generate an absolute URL instead of a relative one.

path

Listing 94-18
```
{{ path(name, parameters = [], relative = false) }}
```

`name`
type: `string`

`parameters` ***(optional)***
type: `array` **default**: `[]`

`relative` ***(optional)***
type: `boolean` **default**: `false`

Returns the relative URL (without the scheme and host) for the given route. If `relative` is enabled, it'll create a path relative to the current path. More information in Linking to Pages.

Read Routing to learn more about the Routing component.

url

Listing 94-19
```
{{ url(name, parameters = [], schemeRelative = false) }}
```

`name`
type: `string`

`parameters` ***(optional)***
type: `array` **default**: `[]`

`schemeRelative` ***(optional)***
type: `boolean` **default**: `false`

Returns the absolute URL (with scheme and host) for the given route. If `schemeRelative` is enabled, it'll create a scheme-relative URL. More information in Linking to Pages.

Read Routing to learn more about the Routing component.

absolute_url

Listing 94-20
```
{{ absolute_url(path) }}
```

`path`
type: `string`

Returns the absolute URL from the passed relative path. For example, assume you're on the following page in your app: `http://example.com/products/hover-board`.

Listing 94-21
```
{{ absolute_url('/human.txt') }}
{# http://example.com/human.txt #}

{{ absolute_url('products_icon.png') }}
{# http://example.com/products/products_icon.png #}
```

relative_path

Listing 94-22
```
{{ relative_path(path) }}
```

`path`
type: `string`

Returns the relative path from the passed absolute URL. For example, assume you're on the following page in your app: `http://example.com/products/hover-board`.

Listing 94-23
```
{{ relative_path('http://example.com/human.txt') }}
{# ../human.txt #}

{{ relative_path('http://example.com/products/products_icon.png') }}
{# products_icon.png #}
```

expression

Creates an *`Expression`*[5] in Twig. See "Template Expressions".

Filters

humanize

Listing 94-24
```
{{ text|humanize }}
```

`text`
type: `string`

Makes a technical name human readable (i.e. replaces underscores by spaces or transforms camelCase text like `helloWorld` to `hello world` and then capitalizes the string).

trans

Listing 94-25
```
{{ message|trans(arguments = [], domain = null, locale = null) }}
```

`message`
type: `string`

`arguments` ***(optional)***
type: `array` **default**: `[]`

`domain` ***(optional)***
type: `string` **default**: `null`

`locale` ***(optional)***
type: `string` **default**: `null`

Translates the text into the current language. More information in Translation Filters.

5. https://api.symfony.com/4.0/Symfony/Component/ExpressionLanguage/Expression.html

transchoice

Listing 94-26
```
{{ message|transchoice(count, arguments = [], domain = null, locale = null) }}
```

`message`
type: `string`

`count`
type: `integer`

`arguments` ***(optional)***
type: `array` **default**: `[]`

`domain` ***(optional)***
type: `string` **default**: `null`

`locale` ***(optional)***
type: `string` **default**: `null`

Translates the text with pluralization support. More information in Translation Filters.

yaml_encode

Listing 94-27
```
{{ input|yaml_encode(inline = 0, dumpObjects = false) }}
```

`input`
type: `mixed`

`inline` ***(optional)***
type: `integer` **default**: `0`

`dumpObjects` ***(optional)***
type: `boolean` **default**: `false`

Transforms the input into YAML syntax. See Writing YAML Files for more information.

yaml_dump

Listing 94-28
```
{{ value|yaml_dump(inline = 0, dumpObjects = false) }}
```

`value`
type: `mixed`

`inline` ***(optional)***
type: `integer` **default**: `0`

`dumpObjects` ***(optional)***
type: `boolean` **default**: `false`

Does the same as *yaml_encode()*[6], but includes the type in the output.

6. `#reference-yaml_encode`

abbr_class

Listing 94-29
```
{{ class|abbr_class }}
```

`class`
type: `string`

Generates an `<abbr>` element with the short name of a PHP class (the FQCN will be shown in a tooltip when a user hovers over the element).

abbr_method

Listing 94-30
```
{{ method|abbr_method }}
```

`method`
type: `string`

Generates an `<abbr>` element using the `FQCN::method()` syntax. If `method` is `Closure`, `Closure` will be used instead and if `method` doesn't have a class name, it's shown as a function (`method()`).

format_args

Listing 94-31
```
{{ args|format_args }}
```

`args`
type: `array`

Generates a string with the arguments and their types (within `<em>` elements).

format_args_as_text

Listing 94-32
```
{{ args|format_args_as_text }}
```

`args`
type: `array`

Equal to the format_args filter, but without using HTML tags.

file_excerpt

Listing 94-33
```
{{ file|file_excerpt(line, srcContext = 3) }}
```

`file`
type: `string`

`line`
type: `integer`

`srcContext` ***(optional)***
type: `integer`

Generates an excerpt of a code file around the given `line` number. The `srcContext` argument defines the total number of lines to display around the given line number (use `-1` to display the whole file).

format_file

Listing 94-34
```
{{ file|format_file(line, text = null) }}
```

`file`
: **type**: `string`

`line`
: **type**: `integer`

`text` ***(optional)***
: **type**: `string` **default**: `null`

Generates the file path inside an `<a>` element. If the path is inside the kernel root directory, the kernel root directory path is replaced by `kernel.root_dir` (showing the full path in a tooltip on hover).

format_file_from_text

Listing 94-35
```
{{ text|format_file_from_text }}
```

`text`
: **type**: `string`

Uses format_file to improve the output of default PHP errors.

file_link

Listing 94-36
```
{{ file|file_link(line) }}
```

`file`
: **type**: `string`

`line`
: **type**: `integer`

Generates a link to the provided file and line number using a preconfigured scheme.

Tags

form_theme

Listing 94-37
```
{% form_theme form resources %}
```

`form`
: **type**: `FormView`

`resources`
: **type**: `array | string`

Sets the resources to override the form theme for the given form view instance. You can use `_self` as resources to set it to the current resource. More information in *How to Customize Form Rendering*.

trans

Listing 94-38
```
{% trans with vars from domain into locale %}{% endtrans %}
```

`vars` ***(optional)***
type: `array` **default**: `[]`

`domain` ***(optional)***
type: `string` **default**: `string`

`locale` ***(optional)***
type: `string` **default**: `string`

Renders the translation of the content. More information in Twig Templates.

transchoice

Listing 94-39
```
{% transchoice count with vars from domain into locale %}{% endtranschoice %}
```

`count`
type: `integer`

`vars` ***(optional)***
type: `array` **default**: `[]`

`domain` ***(optional)***
type: `string` **default**: `null`

`locale` ***(optional)***
type: `string` **default**: `null`

Renders the translation of the content with pluralization support, more information in Twig Templates.

trans_default_domain

Listing 94-40
```
{% trans_default_domain domain %}
```

`domain`
type: `string`

This will set the default domain in the current template.

stopwatch

Listing 94-41
```
{% stopwatch 'name' %}...{% endstopwatch %}
```

This will time the run time of the code inside it and put that on the timeline of the WebProfilerBundle.

Tests

selectedchoice

Listing 94-42
```
{% if choice is selectedchoice(selectedValue) %}
```

`choice`
: **type**: `ChoiceView`

`selectedValue`
: **type**: `string`

Checks if `selectedValue` was checked for the provided choice field. Using this test is the most effective way.

Global Variables

app

The `app` variable is available everywhere and gives access to many commonly needed objects and values. It is an instance of *`GlobalVariables`*[7].

The available attributes are:

- `app.user`, a PHP object representing the current user;
- `app.request`, a *`Request`*[8] object;
- `app.session`, a *`Session`*[9] object;
- `app.environment`, a string with the name of the execution environment;
- `app.debug`, a boolean telling whether the debug mode is enabled in the app;
- `app.token`, a *`TokenInterface`*[10] object representing the security token
- `app.flashes`, returns flash messages from the session

7. https://api.symfony.com/4.0/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html
8. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Request.html
9. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Session/Session.html
10. https://api.symfony.com/4.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html

Chapter 95
Built-in Symfony Service Tags

Service tags are the mechanism used by the *DependencyInjection component* to flag services that require special processing, like console commands or Twig extensions.

These are the most common tags provided by Symfony components, but in your application there could be more tags available provided by third-party bundles:

Tag Name	Usage
auto_alias	Define aliases based on the value of container parameters
console.command	Add a command
controller.argument_value_resolver	Register a value resolver for controller arguments such as `Request`
data_collector	Create a class that collects custom data for the profiler
doctrine.event_listener	Add a Doctrine event listener
doctrine.event_subscriber	Add a Doctrine event subscriber
form.type	Create a custom form field type
form.type_extension	Create a custom "form extension"
form.type_guesser	Add your own logic for "form type guessing"
kernel.cache_clearer	Register your service to be called during the cache clearing process
kernel.cache_warmer	Register your service to be called during the cache warming process
kernel.event_listener	Listen to different events/hooks in Symfony
kernel.event_subscriber	To subscribe to a set of different events/hooks in Symfony
kernel.fragment_renderer	Add new HTTP content rendering strategies
monolog.logger	Logging with a custom logging channel
monolog.processor	Add a custom processor for logging

Tag Name	Usage
routing.loader	Register a custom service that loads routes
routing.expression_language_provider	Register a provider for expression language functions in routing
security.expression_language_provider	Register a provider for expression language functions in security
security.voter	Add a custom voter to Symfony's authorization logic
security.remember_me_aware	To allow remember me authentication
serializer.encoder	Register a new encoder in the `serializer` service
serializer.normalizer	Register a new normalizer in the `serializer` service
swiftmailer.default.plugin	Register a custom SwiftMailer Plugin
templating.helper	Make your service available in PHP templates
translation.loader	Register a custom service that loads translations
translation.extractor	Register a custom service that extracts translation messages from a file
translation.dumper	Register a custom service that dumps translation messages
twig.extension	Register a custom Twig Extension
twig.loader	Register a custom service that loads Twig templates
validator.constraint_validator	Create your own custom validation constraint
validator.initializer	Register a service that initializes objects before validation

auto_alias

Purpose: Define aliases based on the value of container parameters

Consider the following configuration that defines three different but related services:

Listing 95-1
```yaml
services:
    app.mysql_lock:
        class: App\Lock\MysqlLock
        public: false
    app.postgresql_lock:
        class: App\Lock\PostgresqlLock
        public: false
    app.sqlite_lock:
        class: App\Lock\SqliteLock
        public: false
```

Instead of dealing with these three services, your application needs a generic `app.lock` service that will be an alias to one of these services, depending on some configuration. Thanks to the `auto_alias` option, you can automatically create that alias based on the value of a configuration parameter.

Considering that a configuration parameter called `database_type` exists. Then, the generic `app.lock` service can be defined as follows:

Listing 95-2
```yaml
services:
    app.mysql_lock:
        # ...
    app.postgresql_lock:
```

```
        # ...
    app.sqlite_lock:
        # ...
    app.lock:
        tags:
            - { name: auto_alias, format: "app.%database_type%_lock" }
```

The `format` option defines the expression used to construct the name of the service to alias. This expression can use any container parameter (as usual, wrapping their names with `%` characters).

When using the `auto_alias` tag, it's not mandatory to define the aliased services as private. However, doing that (like in the above example) makes sense most of the times to prevent accessing those services directly instead of using the generic service alias.

You need to manually add the `Symfony\Component\DependencyInjection\Compiler\AutoAliasServicePass` compiler pass to the container for this feature to work.

console.command

Purpose: Add a command to the application

For details on registering your own commands in the service container, read *How to Define Commands as Services*.

controller.argument_value_resolver

Purpose: Register a value resolver for controller arguments such as `Request`

Value resolvers implement the *`ArgumentValueResolverInterface`*[1] and are used to resolve argument values for controllers as described here: *Extending Action Argument Resolving*.

data_collector

Purpose: Create a class that collects custom data for the profiler

For details on creating your own custom data collection, read the *How to Create a custom Data Collector* article.

doctrine.event_listener

Purpose: Add a Doctrine event listener

For details on creating Doctrine event listeners, read the *Doctrine Event Listeners and Subscribers* article.

1. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Controller/ArgumentValueResolverInterface.html

doctrine.event_subscriber

Purpose: Add a Doctrine event subscriber

For details on creating Doctrine event subscribers, read the *Doctrine Event Listeners and Subscribers* article.

form.type

Purpose: Create a custom form field type

For details on creating your own custom form type, read the *How to Create a Custom Form Field Type* article.

form.type_extension

Purpose: Create a custom "form extension"

For details on creating Form type extensions, read the *How to Create a Form Type Extension* article.

form.type_guesser

Purpose: Add your own logic for "form type guessing"

This tag allows you to add your own logic to the form guessing process. By default, form guessing is done by "guessers" based on the validation metadata and Doctrine metadata (if you're using Doctrine) or Propel metadata (if you're using Propel).

> *For information on how to create your own type guesser, see Creating a custom Type Guesser.*

kernel.cache_clearer

Purpose: Register your service to be called during the cache clearing process

Cache clearing occurs whenever you call `cache:clear` command. If your bundle caches files, you should add custom cache clearer for clearing those files during the cache clearing process.

In order to register your custom cache clearer, first you must create a service class:

Listing 95-3

```
// src/Cache/MyClearer.php
namespace App\Cache;

use Symfony\Component\HttpKernel\CacheClearer\CacheClearerInterface;

class MyClearer implements CacheClearerInterface
{
    public function clear($cacheDirectory)
    {
        // clear your cache
    }
}
```

If you're using the default services.yaml configuration, your service will be automatically tagged with `kernel.cache_clearer`. But, you can also register it manually:

Listing 95-4

```yaml
services:
    App\Cache\MyClearer:
        tags: [kernel.cache_clearer]
```

kernel.cache_warmer

Purpose: Register your service to be called during the cache warming process

Cache warming occurs whenever you run the `cache:warmup` or `cache:clear` command (unless you pass `--no-warmup` to `cache:clear`). It is also run when handling the request, if it wasn't done by one of the commands yet.

The purpose is to initialize any cache that will be needed by the application and prevent the first user from any significant "cache hit" where the cache is generated dynamically.

To register your own cache warmer, first create a service that implements the *CacheWarmerInterface*[2] interface:

Listing 95-5
```php
// src/Cache/MyCustomWarmer.php
namespace App\Cache;

use Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerInterface;

class MyCustomWarmer implements CacheWarmerInterface
{
    public function warmUp($cacheDirectory)
    {
        // ... do some sort of operations to "warm" your cache
    }

    public function isOptional()
    {
        return true;
    }
}
```

The `isOptional()` method should return true if it's possible to use the application without calling this cache warmer. In Symfony, optional warmers are always executed by default (you can change this by using the `--no-optional-warmers` option when executing the command).

If you're using the default services.yaml configuration, your service will be automatically tagged with `kernel.cache_warmer`. But, you can also register it manually:

Listing 95-6
```yaml
services:
    App\Cache\MyCustomWarmer:
        tags:
            - { name: kernel.cache_warmer, priority: 0 }
```

The `priority` value is optional and defaults to 0. The higher the priority, the sooner it gets executed.

If your cache warmer fails its execution because of any exception, Symfony won't try to execute it again for the next requests. Therefore, your application and/or bundles should be prepared for when the contents generated by the cache warmer are not available.

2. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/CacheWarmer/CacheWarmerInterface.html

Core Cache Warmers

Cache Warmer Class Name	Priority
TemplatePathsCacheWarmer[3]	20
RouterCacheWarmer[4]	0
TemplateCacheCacheWarmer[5]	0

kernel.event_listener

Purpose: To listen to different events/hooks in Symfony

During the execution of a Symfony application, different events are triggered and you can also dispatch custom events. This tag allows you to *hook* your own classes into any of those events.

For a full example of this listener, read the *Events and Event Listeners* article.

Core Event Listener Reference

For the reference of Event Listeners associated with each kernel event, see the *Symfony Events Reference*.

kernel.event_subscriber

Purpose: To subscribe to a set of different events/hooks in Symfony

This is an alternative way to create an event listener, and is the recommended way (instead of using `kernel.event_listener`). See Creating an Event Subscriber.

kernel.fragment_renderer

Purpose: Add a new HTTP content rendering strategy

To add a new rendering strategy - in addition to the core strategies like `EsiFragmentRenderer` - create a class that implements *`FragmentRendererInterface`*[6], register it as a service, then tag it with `kernel.fragment_renderer`.

monolog.logger

Purpose: To use a custom logging channel with Monolog

Monolog allows you to share its handlers between several logging channels. The logger service uses the channel `app` but you can change the channel when injecting the logger in a service.

Listing 95-7

```
services:
    App\Log\CustomLogger:
        arguments: ['@logger']
        tags:
            - { name: monolog.logger, channel: app }
```

3. https://api.symfony.com/4.0/Symfony/Bundle/FrameworkBundle/CacheWarmer/TemplatePathsCacheWarmer.html
4. https://api.symfony.com/4.0/Symfony/Bundle/FrameworkBundle/CacheWarmer/RouterCacheWarmer.html
5. https://api.symfony.com/4.0/Symfony/Bundle/TwigBundle/CacheWarmer/TemplateCacheCacheWarmer.html
6. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Fragment/FragmentRendererInterface.html

You can also configure custom channels in the configuration and retrieve the corresponding logger service from the service container directly (see Configure Additional Channels without Tagged Services).

monolog.processor

Purpose: Add a custom processor for logging

Monolog allows you to add processors in the logger or in the handlers to add extra data in the records. A processor receives the record as an argument and must return it after adding some extra data in the `extra` attribute of the record.

The built-in `IntrospectionProcessor` can be used to add the file, the line, the class and the method where the logger was triggered.

You can add a processor globally:

Listing 95-8
```
services:
    Monolog\Processor\IntrospectionProcessor:
        tags: [monolog.processor]
```

If your service is not a callable (using `__invoke()`) you can add the `method` attribute in the tag to use a specific method.

You can add also a processor for a specific handler by using the `handler` attribute:

Listing 95-9
```
services:
    Monolog\Processor\IntrospectionProcessor:
        tags:
            - { name: monolog.processor, handler: firephp }
```

You can also add a processor for a specific logging channel by using the `channel` attribute. This will register the processor only for the `security` logging channel used in the Security component:

Listing 95-10
```
services:
    Monolog\Processor\IntrospectionProcessor:
        tags:
            - { name: monolog.processor, channel: security }
```

You cannot use both the `handler` and `channel` attributes for the same tag as handlers are shared between all channels.

routing.loader

Purpose: Register a custom service that loads routes

To enable a custom routing loader, add it as a regular service in one of your configuration and tag it with `routing.loader`:

Listing 95-11
```
services:
    App\Routing\CustomLoader:
        tags: [routing.loader]
```

For more information, see *How to Create a custom Route Loader*.

routing.expression_language_provider

Purpose: Register a provider for expression language functions in routing

This tag is used to automatically register expression function providers for the routing expression component. Using these providers, you can add custom functions to the routing expression language.

security.expression_language_provider

Purpose: Register a provider for expression language functions in security

This tag is used to automatically register expression function providers for the security expression component. Using these providers, you can add custom functions to the security expression language.

security.remember_me_aware

Purpose: To allow remember me authentication

This tag is used internally to allow remember-me authentication to work. If you have a custom authentication method where a user can be remember-me authenticated, then you may need to use this tag.

If your custom authentication factory extends *`AbstractFactory`*[7] and your custom authentication listener extends *`AbstractAuthenticationListener`*[8], then your custom authentication listener will automatically have this tagged applied and it will function automatically.

security.voter

Purpose: To add a custom voter to Symfony's authorization logic

When you call `isGranted()` on Symfony's authorization checker, a system of "voters" is used behind the scenes to determine if the user should have access. The `security.voter` tag allows you to add your own custom voter to that system.

For more information, read the *How to Use Voters to Check User Permissions* article.

serializer.encoder

Purpose: Register a new encoder in the `serializer` service

The class that's tagged should implement the *`EncoderInterface`*[9] and *`DecoderInterface`*[10].

For more details, see *How to Use the Serializer*.

7. https://api.symfony.com/4.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.html
8. https://api.symfony.com/4.0/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.html
9. https://api.symfony.com/4.0/Symfony/Component/Serializer/Encoder/EncoderInterface.html
10. https://api.symfony.com/4.0/Symfony/Component/Serializer/Encoder/DecoderInterface.html

serializer.normalizer

Purpose: Register a new normalizer in the Serializer service

The class that's tagged should implement the *NormalizerInterface*[11] and *DenormalizerInterface*[12].

For more details, see *How to Use the Serializer*.

The priorities of the default normalizers can be found in the *registerSerializerConfiguration()*[13] method.

swiftmailer.default.plugin

Purpose: Register a custom SwiftMailer Plugin

If you're using a custom SwiftMailer plugin (or want to create one), you can register it with SwiftMailer by creating a service for your plugin and tagging it with `swiftmailer.default.plugin` (it has no options).

`default` in this tag is the name of the mailer. If you have multiple mailers configured or have changed the default mailer name for some reason, you should change it to the name of your mailer in order to use this tag.

A SwiftMailer plugin must implement the `Swift_Events_EventListener` interface. For more information on plugins, see *SwiftMailer's Plugin Documentation*[14].

Several SwiftMailer plugins are core to Symfony and can be activated via different configuration. For details, see *SwiftmailerBundle Configuration ("swiftmailer")*.

templating.helper

Purpose: Make your service available in PHP templates

To enable a custom template helper, add it as a regular service in one of your configuration, tag it with `templating.helper` and define an `alias` attribute (the helper will be accessible via this alias in the templates):

Listing 95-12
```
services:
    App\Templating\AppHelper:
        tags:
            - { name: templating.helper, alias: alias_name }
```

translation.loader

Purpose: To register a custom service that loads translations

11. https://api.symfony.com/4.0/Symfony/Component/Serializer/Normalizer/NormalizerInterface.html
12. https://api.symfony.com/4.0/Symfony/Component/Serializer/Normalizer/DenormalizerInterface.html
13. https://api.symfony.com/4.0/Symfony/Bundle/FrameworkBundle/DependencyInjection/FrameworkExtension.html#method_registerSerializerConfiguration
14. http://swiftmailer.org/docs/plugins.html

By default, translations are loaded from the filesystem in a variety of different formats (YAML, XLIFF, PHP, etc).

Learn how to load custom formats in the components section.

Now, register your loader as a service and tag it with `translation.loader`:

Listing 95-13
```yaml
services:
    App\Translation\MyCustomLoader:
        tags:
            - { name: translation.loader, alias: bin }
```

The `alias` option is required and very important: it defines the file "suffix" that will be used for the resource files that use this loader. For example, suppose you have some custom `bin` format that you need to load. If you have a `bin` file that contains French translations for the `messages` domain, then you might have a file `translations/messages.fr.bin`.

When Symfony tries to load the `bin` file, it passes the path to your custom loader as the `$resource` argument. You can then perform any logic you need on that file in order to load your translations.

If you're loading translations from a database, you'll still need a resource file, but it might either be blank or contain a little bit of information about loading those resources from the database. The file is key to trigger the `load()` method on your custom loader.

translation.extractor

Purpose: To register a custom service that extracts messages from a file

When executing the `translation:update` command, it uses extractors to extract translation messages from a file. By default, the Symfony Framework has a *`TwigExtractor`*[15] and a *`PhpExtractor`*[16], which help to find and extract translation keys from Twig templates and PHP files.

You can create your own extractor by creating a class that implements *`ExtractorInterface`*[17] and tagging the service with `translation.extractor`. The tag has one required option: `alias`, which defines the name of the extractor:

Listing 95-14
```php
// src/Acme/DemoBundle/Translation/FooExtractor.php
namespace Acme\DemoBundle\Translation;

use Symfony\Component\Translation\Extractor\ExtractorInterface;
use Symfony\Component\Translation\MessageCatalogue;

class FooExtractor implements ExtractorInterface
{
    protected $prefix;

    /**
     * Extracts translation messages from a template directory to the catalogue.
     */
    public function extract($directory, MessageCatalogue $catalogue)
    {
        // ...
    }

    /**
     * Sets the prefix that should be used for new found messages.
     */
```

15. https://api.symfony.com/4.0/Symfony/Bridge/Twig/Translation/TwigExtractor.html
16. https://api.symfony.com/4.0/Symfony/Bundle/FrameworkBundle/Translation/PhpExtractor.html
17. https://api.symfony.com/4.0/Symfony/Component/Translation/Extractor/ExtractorInterface.html

```
22      public function setPrefix($prefix)
23      {
24          $this->prefix = $prefix;
25      }
26  }
```

Listing 95-15
```
1  services:
2      App\Translation\CustomExtractor:
3          tags:
4              - { name: translation.extractor, alias: foo }
```

translation.dumper

Purpose: To register a custom service that dumps messages to a file

After a translation extractor has extracted all messages from the templates, the dumpers are executed to dump the messages to a translation file in a specific format.

Symfony already comes with many dumpers:

- *CsvFileDumper*[18]
- *IcuResFileDumper*[19]
- *IniFileDumper*[20]
- *MoFileDumper*[21]
- *PoFileDumper*[22]
- *QtFileDumper*[23]
- *XliffFileDumper*[24]
- *YamlFileDumper*[25]

You can create your own dumper by extending *FileDumper*[26] or implementing *DumperInterface*[27] and tagging the service with `translation.dumper`. The tag has one option: `alias` This is the name that's used to determine which dumper should be used.

Listing 95-16
```
1  services:
2      App\Translation\JsonFileDumper:
3          tags:
4              - { name: translation.dumper, alias: json }
```

Learn how to dump to custom formats in the components section.

twig.extension

Purpose: To register a custom Twig Extension

To enable a Twig extension, add it as a regular service in one of your configuration and tag it with `twig.extension`. If you're using the default services.yaml configuration, the service is auto-registered and auto-tagged. But, you can also register it manually:

18. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/CsvFileDumper.html
19. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/IcuResFileDumper.html
20. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/IniFileDumper.html
21. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/MoFileDumper.html
22. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/PoFileDumper.html
23. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/QtFileDumper.html
24. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/XliffFileDumper.html
25. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/YamlFileDumper.html
26. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/FileDumper.html
27. https://api.symfony.com/4.0/Symfony/Component/Translation/Dumper/DumperInterface.html

Listing 95-17
```yaml
services:
    App\Twig\AppExtension:
        tags: [twig.extension]
```

For information on how to create the actual Twig Extension class, see *Twig's documentation*[28] on the topic or read the *How to Write a custom Twig Extension* article.

Before writing your own extensions, have a look at the *Twig official extension repository*[29] which already includes several useful extensions. For example `Intl` and its `localizeddate` filter that formats a date according to user's locale. These official Twig extensions also have to be added as regular services:

Listing 95-18
```yaml
services:
    Twig\Extensions\IntlExtension:
        tags: [twig.extension]
```

twig.loader

Purpose: Register a custom service that loads Twig templates

By default, Symfony uses only one *Twig Loader*[30] - *`FilesystemLoader`*[31]. If you need to load Twig templates from another resource, you can create a service for the new loader and tag it with `twig.loader`.

If you use the default services.yaml configuration, the service will be automatically tagged thanks to autoconfiguration. But, you can also register it manually:

Listing 95-19
```yaml
services:
    App\Twig\CustomLoader:
        tags:
            - { name: twig.loader, priority: 0 }
```

The `priority` value is optional and defaults to `0`. The higher priority loaders are tried first.

validator.constraint_validator

Purpose: Create your own custom validation constraint

This tag allows you to create and register your own custom validation constraint. For more information, read the *How to Create a custom Validation Constraint* article.

validator.initializer

Purpose: Register a service that initializes objects before validation

This tag provides a very uncommon piece of functionality that allows you to perform some sort of action on an object right before it's validated. For example, it's used by Doctrine to query for all of the lazily-

28. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension
29. https://github.com/fabpot/Twig-extensions
30. http://twig.sensiolabs.org/doc/api.html#loaders
31. https://api.symfony.com/4.0/Symfony/Bundle/TwigBundle/Loader/FilesystemLoader.html

loaded data on an object before it's validated. Without this, some data on a Doctrine entity would appear to be "missing" when validated, even though this is not really the case.

If you do need to use this tag, just make a new class that implements the *`ObjectInitializerInterface`*[32] interface. Then, tag it with the `validator.initializer` tag (it has no options).

For an example, see the `DoctrineInitializer` class inside the Doctrine Bridge.

32. https://api.symfony.com/4.0/Symfony/Component/Validator/ObjectInitializerInterface.html

Chapter 96

Built-in Symfony Events

During the handling of an HTTP request, the Symfony framework (or any application using the *HttpKernel component*) dispatches some *events* which you can use to modify how the request is handled.

Kernel Events

Each event dispatched by the HttpKernel component is a subclass of *`KernelEvent`*[1], which provides the following information:

`getRequestType()`[2]
: Returns the *type* of the request (`HttpKernelInterface::MASTER_REQUEST` or `HttpKernelInterface::SUB_REQUEST`).

`getKernel()`[3]
: Returns the Kernel handling the request.

`getRequest()`[4]
: Returns the current `Request` being handled.

`kernel.request`

Event Class: *`GetResponseEvent`*[5]

This event is dispatched very early in Symfony, before the controller is determined. It's useful to add information to the Request or return a Response early to stop the handling of the request.

Read more on the kernel.request event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-1
```
$ php bin/console debug:event-dispatcher kernel.request
```

1. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
2. **https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/KernelEvent.html#method_getRequestType**
3. **https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/KernelEvent.html#method_getKernel**
4. **https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/KernelEvent.html#method_getRequest**
5. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html

kernel.controller

Event Class: *FilterControllerEvent*[6]

This event is dispatched after the controller to be executed has been resolved but before executing it. It's useful to initialize things later needed by the controller, such as *param converters*[7], and even to change the controller entirely:

Listing 96-2
```
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $event)
{
    // ...

    // the controller can be changed to any PHP callable
    $event->setController($myCustomController);
}
```

Read more on the kernel.controller event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-3
```
$ php bin/console debug:event-dispatcher kernel.controller
```

kernel.view

Event Class: *GetResponseForControllerResultEvent*[8]

This event is dispatched after the controller has been executed but *only* if the controller does *not* return a *Response*[9] object. It's useful to transform the returned value (e.g. a string with some HTML contents) into the `Response` object needed by Symfony:

Listing 96-4
```
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $value = $event->getControllerResult();
    $response = new Response();

    // ... somehow customize the Response from the return value

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

Read more on the kernel.view event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-5
```
$ php bin/console debug:event-dispatcher kernel.view
```

kernel.response

Event Class: *FilterResponseEvent*[10]

6. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html
7. https://symfony.com/doc/master/bundles/SensioFrameworkExtraBundle/annotations/converters.html
8. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
9. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Response.html

This event is dispatched after the controller or any `kernel.view` listener returns a `Response` object. It's useful to modify or replace the response before sending it back (e.g. add/modify HTTP headers, add cookies, etc.):

Listing 96-6
```
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();

    // ... modify the response object
}
```

Read more on the kernel.response event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-7
```
$ php bin/console debug:event-dispatcher kernel.response
```

kernel.finish_request

Event Class: *FinishRequestEvent*[11]

This event is dispatched after a sub request has finished. It's useful to reset the global state of the application (for example, the translator listener resets the translator's locale to the one of the parent request):

Listing 96-8
```
public function onKernelFinishRequest(FinishRequestEvent $event)
{
    if (null === $parentRequest = $this->requestStack->getParentRequest()) {
        return;
    }

    // reset the locale of the subrequest to the locale of the parent request
    $this->setLocale($parentRequest);
}
```

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-9
```
$ php bin/console debug:event-dispatcher kernel.finish_request
```

kernel.terminate

Event Class: *PostResponseEvent*[12]

This event is dispatched after the response has been sent (after the execution of the *handle()*[13] method). It's useful to perform slow or complex tasks that don't need to be completed to send the response (e.g. sending emails).

Read more on the kernel.terminate event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-10
```
$ php bin/console debug:event-dispatcher kernel.terminate
```

10. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html
11. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/FinishRequestEvent.html
12. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/PostResponseEvent.html
13. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/HttpKernel.html#method_handle

kernel.exception

Event Class: *GetResponseForExceptionEvent*[14]

This event is dispatched as soon as an error occurs during the handling of the HTTP request. It's useful to recover from errors or modify the exception details sent as response:

Listing 96-11
```
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $event)
{
    $exception = $event->getException();
    $response = new Response();
    // setup the Response object based on the caught exception
    $event->setResponse($response);

    // you can alternatively set a new Exception
    // $exception = new \Exception('Some special exception');
    // $event->setException($exception);
}
```

The TwigBundle registers an *ExceptionListener*[15] that forwards the `Request` to a given controller defined by the `exception_listener.controller` parameter.

Symfony uses the following logic to determine the HTTP status code of the response:

- If *isClientError()*[16], *isServerError()*[17] or *isRedirect()*[18] is true, then the status code on your `Response` object is used;
- If the original exception implements *HttpExceptionInterface*[19], then `getStatusCode()` is called on the exception and used (the headers from `getHeaders()` are also added);
- If both of the above aren't true, then a 500 status code is used.

If you want to overwrite the status code of the exception response, which you should not without a good reason, call `GetResponseForExceptionEvent::allowCustomResponseCode()` first and then set the status code on the response:

Listing 96-12
```
$event->allowCustomResponseCode();
$response = new Response('No Content', 204);
$event->setResponse($response);
```

The status code sent to the client in the above example will be `204`. If `$event->allowCustomResponseCode()` is omitted, then the kernel will set an appropriate status code based on the type of exception thrown.

Read more on the kernel.exception event.

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 96-13
```
$ php bin/console debug:event-dispatcher kernel.exception
```

14. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
15. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
16. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Response.html#method_isClientError
17. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Response.html#method_isServerError
18. https://api.symfony.com/4.0/Symfony/Component/HttpFoundation/Response.html#method_isRedirect
19. https://api.symfony.com/4.0/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.html

Chapter 97
Requirements for Running Symfony

Symfony 4.0 requires **PHP 7.1.3** or higher to run, in addition to other minor requirements. To make things simple, Symfony provides a tool to quickly check if your system meets all those requirements. Run this command to install the tool:

Listing 97-1
```
$ cd your-project/
$ composer require symfony/requirements-checker
```

Beware that PHP can define a different configuration for the command console and the web server, so you need to check requirements in both environments.

Checking Requirements for the Web Server

The requirements checker tool creates a file called `check.php` in the `public/` directory of your project. Open that file with your browser to check the requirements.

Once you've fixed all the reported issues, uninstall the requirements checker to avoid leaking internal information about your application to visitors:

Listing 97-2
```
$ cd your-project/
$ composer remove symfony/requirements-checker
```

Checking Requirements for the Command Console

The requirements checker tool adds a script to your Composer configuration to check the requirements automatically. There's no need to execute any command; if there is any issue, you'll see them in the console output.