



Symfony

The Reference Book

Version: 4.3

generated on October 21, 2019

The Reference Book (4.3)

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 (<https://github.com/symfony/symfony-docs/issues>). Based on tickets and users feedback, this book is continuously updated.

Contents at a Glance

Framework Configuration Reference (FrameworkBundle)	6
Doctrine Configuration Reference (DoctrineBundle)	39
Security Configuration Reference (SecurityBundle)	44
Mailer Configuration Reference (SwiftmailerBundle)	53
Twig Configuration Reference (TwigBundle)	58
Logging Configuration Reference (MonologBundle)	63
Profiler Configuration Reference (WebProfilerBundle)	64
Debug Configuration Reference (DebugBundle)	66
Configuring in the Kernel	68
Form Types Reference	71
TextType Field	73
TextareaType Field	79
EmailType Field	85
IntegerType Field	91
MoneyType Field	98
NumberType Field	106
PasswordType Field	114
PercentType Field	120
SearchType Field	127
UrlType Field	133
RangeType Field	139
TelType Field	144
ColorType Field	150
ChoiceType Field (select drop-downs, radio buttons & checkboxes)	156
EntityType Field	173
CountryType Field	187
LanguageType Field	196
LocaleType Field	205
TimezoneType Field	214
CurrencyType Field	224
DateType Field	232
DateIntervalType Field	242
DateTimeType Field	251
TimeType Field	260
BirthdayType Field	269
CheckboxType Field	276

FileType Field	283
RadioType Field.....	290
CollectionType Field	296
RepeatedType Field	307
HiddenType Field	313
ButtonType Field	317
ResetType Field	320
SubmitType Field.....	323
FormType Field	328
Validation Constraints Reference.....	341
NotBlank.....	344
Blank.....	346
NotNull.....	348
IsNull	350
IsTrue.....	352
IsFalse	354
Type.....	356
Email.....	359
Length	362
Url.....	366
Regex	370
Ip	374
Uuid.....	377
Json.....	380
EqualTo	382
NotEqualTo.....	385
IdenticalTo	388
NotIdenticalTo	391
LessThan	394
LessThanOrEqualTo	397
GreaterThan	400
GreaterThanOrEqual	403
Range	406
DivisibleBy	410
Unique	412
Positive.....	414
PositiveOrZero	416
Negative	418
NegativeOrZero	420
Date	422
DateTime	424
Time.....	426
Timezone.....	428
Choice.....	431
Collection.....	436
Count	440
UniqueEntity	443

Language	447
Locale.....	449
Country	451
File	453
Image	459
CardScheme	467
Currency	470
Luhn	472
Iban.....	474
Bic.....	476
Isbn	479
Issn.....	482
Callback	484
Expression	488
All	492
UserPassword	494
NotCompromisedPassword.....	496
Valid	499
Traverse.....	502
Twig Extensions Defined by Symfony	504
Built-in Symfony Service Tags.....	514
Built-in Symfony Events	527



Chapter 1

Framework Configuration Reference (FrameworkBundle)

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

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference framework
3
4 # displays the actual config values used by your application
5 $ 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: <https://symfony.com/schema/dic/symfony/symfony-1.0.xsd>

Configuration

- annotations
 - cache
 - debug
 - file_cache_dir
- assets
 - base_path
 - base_urls
 - json_manifest_path
 - packages
 - version_format
 - version_strategy

- version
- cache
 - app
 - default_doctrine_provider
 - default_memcached_provider
 - default_pdo_provider
 - default_psr6_provider
 - default_redis_provider
 - directory
 - pools
 - name
 - adapter
 - clearer
 - default_lifetime
 - provider
 - public
 - tags
 - prefix_seed
 - system
- csrf_protection
 - enabled
- default_locale
- disallow_search_engine_index
- esi
 - enabled
- form
 - enabled
- fragments
 - enabled
 - hinclude_default_template
 - path
- http_client
 - default_options
 - bindto
 - buffer
 - cafile
 - capath
 - ciphers
 - headers
 - http_version
 - local_cert
 - local_pk
 - max_redirects
 - no_proxy

- passphrase
 - peer_fingerprint
 - proxy
 - resolve
 - timeout
 - verify_host
 - verify_peer
- max_host_connections
- scoped_clients
 - scope
 - auth_basic
 - auth_bearer
 - base_uri
 - bindto
 - buffer
 - cafile
 - capath
 - ciphers
 - headers
 - http_version
 - local_cert
 - local_pk
 - max_redirects
 - no_proxy
 - passphrase
 - peer_fingerprint
 - proxy
 - query
 - resolve
 - timeout
 - verify_host
 - verify_peer
- http_method_override
- ide
- lock
 - enabled
 - resources
 - name
- php_errors
 - log
 - throw
- profiler
 - collect
 - dsn
 - enabled
 - only_exceptions
 - only_master_requests
- property_access

- magic_call
- throw_exception_on_invalid_index
- throw_exception_on_invalid_property_path
- property_info
 - enabled
- request:
 - formats
- router
 - http_port
 - https_port
 - resource
 - strict_requirements
 - type
 - utf8
- secret
- serializer
 - circular_reference_handler
 - enable_annotations
 - enabled
 - mapping
 - paths
 - name_converter
- session
 - cache_limiter
 - cookie_domain
 - cookie_httponly
 - cookie_lifetime
 - cookie_path
 - cookie_samesite
 - cookie_secure
 - enabled
 - gc_divisor
 - gc_maxlifetime
 - gc_probability
 - handler_id
 - metadata_update_threshold
 - name
 - save_path
 - sid_length
 - sid_bits_per_character
 - storage_id
 - use_cookies
- templating
 - cache
 - engines
 - form

- resources
- loaders
- test
- translator
 - default_path
 - enabled
 - fallbacks
 - formatter
 - logging
 - paths
- trusted_hosts
- trusted_proxies
- validation
 - cache
 - email_validation_mode
 - enable_annotations
 - enabled
 - mapping
 - paths
 - not_compromised_password
 - enabled
 - endpoint
 - static_method
 - strict_email
 - translation_domain
- workflows
 - enabled
 - name
 - audit_trail
 - initial_marking
 - marking_store
 - metadata
 - places
 - supports
 - support_strategy
 - transitions
 - type

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`.

Changing the Action and HTTP Method of Symfony forms.



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

```
1 // public/index.php
2
3 // ...
4 $kernel = new CacheKernel($kernel);
5
6 Request::enableHttpMethodParameterOverride(); // <-- add this line
7 $request = Request::createFromGlobals();
8 // ...
```

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`, `emacs`, `atom` and `vscode`.



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

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Request.php>

2. <https://github.com/aik099/PhpStormProtocol>

3. <https://github.com/sanduhrs/phpstorm-url-handler>

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

```
1 # config/packages/framework.yaml
2 framework:
3     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.



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

```
1 // /path/to/guest/.../file will be opened
2 // as /path/to/host/.../file on the host
3 // and /var/www/app/ as /projects/my_project/ also
4 'myide://%f:%l&/path/to/guest/>/path/to/host/&/var/www/app/>/projects/my_project/&...'
5
6 // example for PhpStorm
7 'phpstorm://open?file=%f&line=%l&/var/www/app/>/projects/my_project/'
```

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](#).

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Request.php>

disallow_search_engine_index

type: boolean **default:** true when the debug mode is enabled, false otherwise.

New in version 4.3: The `disallow_search_engine_index` option was introduced in Symfony 4.3.

If **true**, Symfony adds a **X-Robots-Tag: noindex** HTTP tag to all responses (unless your own app adds that header, in which case it's not modified). This *X-Robots-Tag HTTP header*⁵ tells search engines to not index your web site. This option is a protection measure in case you accidentally publish your site in debug mode.

trusted_hosts

type: array | string **default:** []

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"⁶ for more information about these kinds of attacks.

The Symfony `Request::getHost()` 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 of the regular expressions in this list, the application won't respond and the user will receive a 400 response.

Listing 1-5

```
1 # config/packages/framework.yaml
2 framework:
3     trusted_hosts: ['^example\.com$', '^example\.org$']
```

Hosts can also be configured to respond to any subdomain, via `^(.+\.)?example\.com$` for instance.

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(['^(.+\.)?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 post⁸.

form

enabled

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

5. https://developers.google.com/search/reference/robots_meta_tag

6. <http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html>

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Request.php>

8. <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>

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.

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

Listing 1-7

```
1 # config/packages/framework.yaml
2 framework:
3     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.

hinclude_default_template

type: string **default:** null

New in version 4.3: The `framework.fragments.hinclude_default_template` option was introduced in Symfony 4.3. In previous Symfony versions it was defined under `framework.templating.hinclude_default_template`.

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.

path

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

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

http_client

When the HttpClient component is installed, an HTTP client is available as a service named `http_client` or using the autowiring alias *HttpClientInterface*⁹.

This service can be configured using `framework.http_client.default_options`:

Listing 1-8

```
1 # config/packages/framework.yaml
2 framework:
3     # ...
4     http_client:
5         max_host_connections: 10
6         default_options:
7             headers: { 'X-Powered-By': 'ACME App' }
8             max_redirects: 7
```

Multiple pre-configured HTTP client services can be defined, each with its service name defined as a key under `scoped_clients`. Scoped clients inherit the default options defined for the `http_client` service. You can override these options and can define a few others:

Listing 1-9

```
1 # config/packages/framework.yaml
2 framework:
3     # ...
4     http_client:
5         scoped_clients:
6             my_api.client:
7                 auth_bearer: secret_bearer_token
8                 # ...
```

Options defined for scoped clients apply only to URLs that match either their `base_uri` or the `scope` option when it is defined. Non-matching URLs always use default options.

Each scoped client also defines a corresponding named autowiring alias. If you use for example `Symfony\Contracts\HttpClient\HttpClientInterface $myApiClient` as the type and name of an argument, autowiring will inject the `my_api.client` service into your autowired classes.

auth_basic

type: string

9. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Contracts/HttpClient/HttpClientInterface.php>

The username and password used to create the **Authorization** HTTP header used in HTTP Basic authentication. The value of this option must follow the format **username:password**.

auth_bearer

type: string

The token used to create the **Authorization** HTTP header used in HTTP Bearer authentication (also called token authentication).

base_uri

type: string

URI that is merged into relative URIs, following the rules explained in the *RFC 3986*¹⁰ standard. This is useful when all the requests you make share a common prefix (e.g. **https://api.github.com/**) so you can avoid adding it to every request.

Here are some common examples of how **base_uri** merging works in practice:

base_uri	Relative URI	Actual Requested URI
<i>http://foo.com</i>	<i>/bar</i>	<i>http://foo.com/bar</i>
<i>http://foo.com/foo</i>	<i>/bar</i>	<i>http://foo.com/bar</i>
<i>http://foo.com/foo</i>	<i>bar</i>	<i>http://foo.com/bar</i>
<i>http://foo.com/foo/</i>	<i>bar</i>	<i>http://foo.com/foo/bar</i>
<i>http://foo.com</i>	<i>http://baz.com</i>	<i>http://baz.com</i>
<i>http://foo.com/?bar</i>	<i>bar</i>	<i>http://foo.com/bar</i>

bindto

type: string

A network interface name, IP address, a host name or a UNIX socket to use as the outgoing network interface.

buffer

type: bool **default:** false

Buffering the response means that you can access its content multiple times without performing the request again. Pass **true** as the value of this option to enable buffering.

cafile

type: string

The path of the certificate authority file that contains one or more certificates used to verify the other servers' certificates.

capath

type: string

The path to a directory that contains one or more certificate authority files.

10. <https://www.ietf.org/rfc/rfc3986.txt>

ciphers

type: string

A list of the names of the ciphers allowed for the SSL/TLS connections. They can be separated by colons, commas or spaces (e.g. 'RC4-SHA:TLS13-AES-128-GCM-SHA256').

headers

type: array

An associative array of the HTTP headers added before making the request. This value must use the format ['header-name' => header-value, ...].

http_version

type: string | null **default:** null

The HTTP version to use, typically '1.1' or '2.0'. Leave it to **null** to let Symfony select the best version automatically.

local_cert

type: string

The path to a file that contains the *PEM formatted*¹¹ certificate used by the HTTP client. This is often combined with the **local_pk** and **passphrase** options.

local_pk

type: string

The path of a file that contains the *PEM formatted*¹² private key of the certificate defined in the **local_cert** option.

max_host_connections

type: integer **default:** 6

Defines the maximum amount of simultaneously open connections to a single host (considering a "host" the same as a "host name + port number" pair). This limit also applies for proxy connections, where the proxy is considered to be the host for which this limit is applied.

max_redirects

type: integer **default:** 20

The maximum number of redirects to follow. Use **0** to not follow any redirection.

no_proxy

type: string | null **default:** null

A comma separated list of hosts that do not require a proxy to be reached, even if one is configured. Use the '*' wildcard to match all hosts and an empty string to match none (disables the proxy).

11. https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail

12. https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail

passphrase

type: string

The passphrase used to encrypt the certificate stored in the file defined in the `local_cert` option.

peer_fingerprint

type: array

When negotiating a TLS or SSL connection, the server sends a certificate indicating its identity. A public key is extracted from this certificate and if it does not exactly match any of the public keys provided in this option, the connection is aborted before sending or receiving any data.

The value of this option is an associative array of `algorithm => hash` (e.g. `['pin-sha256' => '...']`).

proxy

type: string | null

The HTTP proxy to use to make the requests. Leave it to `null` to detect the proxy automatically based on your system configuration.

query

type: array

An associative array of the query string values added to the URL before making the request. This value must use the format `['parameter-name' => parameter-value, ...]`.

resolve

type: array

A list of hostnames and their IP addresses to pre-populate the DNS cache used by the HTTP client in order to avoid a DNS lookup for those hosts. This option is useful to improve security when IPs are checked before the URL is passed to the client and to make your tests easier.

The value of this option is an associative array of `domain => IP address` (e.g. `['symfony.com' => '46.137.106.254', ...]`).

scope

type: string

For scoped clients only: the regular expression that the URL must match before applying all other non-default options. By default, the scope is derived from `base_uri`.

timeout

type: float **default:** depends on your PHP config

Time, in seconds, to wait for a response. If the response stales for longer, a *TransportException*¹³ is thrown. Its default value is the same as the value of PHP's `default_socket_timeout`¹⁴ config option.

13. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpClient/Exception/TransportException.php>

14. <https://php.net/manual/en/filesystem.configuration.php#ini.default-socket-timeout>

verify_host

type: boolean

If **true**, the certificate sent by other servers is verified to ensure that their common name matches the host included in the URL. This is usually combined with **verify_peer** to also verify the certificate authenticity.

verify_peer

type: boolean

If **true**, the certificate sent by other servers when negotiating a TLS or SSL connection is verified for authenticity. Authenticating the certificate is not enough to be sure about the server, so you should combine this with the **verify_host** option.

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 1-10

```
$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.

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($mimeType)`¹⁵ or `Request::getMimeType($format)`¹⁶.

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 1-11

```
1 # config/packages/framework.yaml
2 framework:
3     request:
4         formats:
5             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`, `.yaml`, `.php`).

http_port

type: integer **default:** 80

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 behavior. When generating a route that has specific parameter requirements, the generator can behave differently in case the used parameters do not meet these requirements.

15. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Request.php>

16. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Request.php>

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.

utf8

type: boolean default: false

When this option is set to **true**, route patterns can include UTF-8 characters. If the charset of your application is UTF-8 (as defined in the `getCharset()` method of your kernel) it's recommended to set it to **true**. This will make non-UTF8 URLs to generate 404 errors.

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*¹⁷.

handler_id

type: string default: null

The service id used for session storage. The default **null** value means to use the native PHP session mechanism. Set it to **'session.handler.native_file'** to let Symfony manage the sessions itself using files to store the session metadata.

If you prefer to make Symfony store sessions in a database read *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.

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.

17. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.php>

`cookie_path`

type: string **default:** /

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

`cache_limiter`

type: string or int **default:** ''

If set to 0, Symfony won't set any particular header related to the cache and it will rely on the cache control method configured in the *session.cache-limiter*¹⁸ PHP.ini option.

Unlike the other session options, `cache_limiter` is set as a regular container parameter:

Listing 1-12

```
1 # config/services.yaml
2 parameters:
3     session.storage.options:
4         cache_limiter: 0
```

`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_samesite`

type: string or null **default:** null

It controls the way cookies are sent when the HTTP request was not originated from the same domain the cookies are associated to. Setting this option is recommended to mitigate *CSRF security attacks*¹⁹.

By default, browsers send all cookies related to the domain of the HTTP request. This may be a problem for example when you visit a forum and some malicious comment includes a link like https://some-bank.com/?send_money_to=attacker&amount=1000. If you were previously logged into your bank website, the browser will send all those cookies when making that HTTP request.

The possible values for this option are:

- `null`, use it to disable this protection. Same behavior as in older Symfony versions.
- `'strict'` (or the `Cookie::SAMESITE_STRICT` constant), use it to never send any cookie when the HTTP request is not originated from the same domain.
- `'lax'` (or the `Cookie::SAMESITE_LAX` constant), use it to allow sending cookies when the request originated from a different domain, but only when the user consciously made the request (by clicking a link or submitting a form with the GET method).



This option is available starting from PHP 7.3, but Symfony has a polyfill so you can use it with any older PHP version as well.

`cookie_secure`

type: boolean or string **default:** false

18. <https://www.php.net/manual/en/session.configuration.php#ini.session.cache-limiter>

19. https://en.wikipedia.org/wiki/Cross-site_request_forgery

This determines whether cookies should only be sent over secure connections. In addition to **true** and **false**, there's a special '**auto**' value that means **true** for HTTPS requests and **false** for HTTP requests.

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`.

sid_length

type: integer **default:** 32

This determines the length of session ID string, which can be an integer between **22** and **256** (both inclusive), being **32** the recommended value. Longer session IDs are harder to guess.

This option is related to the *session.sid_length PHP option*²⁰.

sid_bits_per_character

type: integer **default:** 4

This determines the number of bits in encoded session ID character. The possible values are **4** (0-9, a-f), **5** (0-9, a-v), and **6** (0-9, a-z, A-Z, "-", ", "). The more bits results in stronger session ID. **5** is recommended value for most environments.

This option is related to the *session.sid_bits_per_character PHP option*²¹.

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.

20. <https://php.net/manual/session.configuration.php#ini.session.sid-length>

21. <https://php.net/manual/session.configuration.php#ini.session.sid-bits-per-character>

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

Listing 1-13

```
1 # config/packages/framework.yaml
2 framework:
3     session:
4         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.

enabled

type: boolean **default:** true

Whether to enable the session support in the framework.

Listing 1-14

```
1 # config/packages/framework.yaml
2 framework:
3     session:
4         enabled: true
```

use_cookies

type: boolean **default:** null

This specifies if the session ID is stored on the client side using cookies or not. By default, it will use the value defined in the `php.ini` with the `session.use_cookies` directive.

assets

base_path

type: string

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

Listing 1-15

```
1 # config/packages/framework.yaml
2 framework:
3     # ...
4     assets:
5         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 1-16

```
1 # config/packages/framework.yaml
2 framework:
3     # ...
4     assets:
```



```

5     base_urls:
6         - 'http://cdn.example.com/'

```

packages

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

Listing 1-17

```

1  # config/packages/framework.yaml
2  framework:
3      # ...
4      assets:
5          packages:
6              avatars:
7                  base_urls: 'http://static_cdn.example.com/avatars'

```

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

Listing 1-18

```

1  

```

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 1-19

```

1  

```

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

Listing 1-20

```

1  # config/packages/framework.yaml
2  framework:
3      # ...
4      assets:
5          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*²² 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.



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 forget 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 1-21 1 # config/packages/framework.yaml
2 framework:
3     assets:
4         # this strategy is applied to every asset (including packages)
5         version_strategy: 'app.asset.my_versioning_strategy'
6     packages:
7         foo_package:
8             # this package removes any versioning (its assets won't be versioned)
9             version: ~
10        bar_package:
11            # this package uses its own strategy (the default strategy is ignored)
12            version_strategy: 'app.asset.another_version_strategy'
13        baz_package:
14            # this package inherits the default strategy
15            base_path: '/images'
```

22. <https://secure.php.net/manual/en/function.sprintf.php>



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*²³ and *gulp-rev*²⁴, respectively.

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

Listing 1-22

```
1  # config/packages/framework.yaml
2  framework:
3      assets:
4          # this manifest is applied to every asset (including packages)
5          json_manifest_path: "%kernel.project_dir%/public/build/manifest.json"
6          packages:
7              foo_package:
8                  # this package uses its own manifest (the default file is ignored)
9                  json_manifest_path: "%kernel.project_dir%/public/build/a_different_manifest.json"
10             bar_package:
11                 # this package uses the global manifest (the default file is used)
12                 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

Deprecated since version 4.3: The integration of the Templating component in FrameworkBundle has been deprecated since version 4.3 and will be removed in 5.0. That's why all the configuration options defined under **framework.templating** are deprecated too.

form

resources

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

23. <https://www.npmjs.com/package/webpack-manifest-plugin>

24. <https://www.npmjs.com/package/gulp-rev>

Deprecated since version 4.3: The integration of the Templating component in FrameworkBundle has been deprecated since version 4.3 and will be removed in 5.0. Form theming with PHP templates will no longer be supported and you'll need to use Twig instead.

A list of all resources for form theming in PHP. This setting is not required if you're using the Twig format for your themes.

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

Listing 1-23

```
1 # config/packages/framework.yaml
2 framework:
3     templating:
4         form:
5             resources:
6                 - 'form_themes'
```



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

See Applying Themes to all Forms 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*²⁵.

translator

enabled

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

25. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Templating/Loader/LoaderInterface.php>

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

fallbacks

type: string|array **default:** ['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.

formatter

type: string **default:** translator.formatter.default

The ID of the service used to format translation messages. The service class must implement the *MessageFormatterInterface*²⁶.

For more details, see Create a Custom Message Formatter.

paths

type: array **default:** []

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

default_path

type: string **default:** %kernel.project_dir%/translations

This option allows to define the path where the application translations files are stored.

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.

26. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Formatter/MessageFormatterInterface.php>

`throw_exception_on_invalid_property_path`

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

New in version 4.3: The `throw_exception_on_invalid_property_path` option was introduced in Symfony 4.3.

When enabled, the `property_accessor` service throws an exception when you try to access an invalid property path of an object.

`property_info`

`enabled`

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

`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*²⁷.

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.

`not_compromised_password`

The *NotCompromisedPassword* constraint makes HTTP requests to a public API to check if the given password has been compromised in a data breach.

`enabled`

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

New in version 4.3: The `enabled` option was introduced in Symfony 4.3.

27. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Mapping/Cache/CacheInterface.php>

If you set this option to **false**, no HTTP requests will be made and the given password will be considered valid. This is useful when you don't want or can't make HTTP requests, such as in **dev** and **test** environments or in continuous integration servers.

endpoint

type: string **default:** null

New in version 4.3: The **endpoint** option was introduced in Symfony 4.3.

By default, the *NotCompromisedPassword* constraint uses the public API provided by *haveibeenpwned.com*²⁸. This option allows to define a different, but compatible, API endpoint to make the password checks. It's useful for example when the Symfony application is run in an intranet without public access to Internet.

static_method

type: string | array **default:** ['loadValidatorMetadata']

Defines the name of the static method which is called to load the validation metadata of the class. You can define an array of strings with the names of several methods. In that case, all of them will be called in that order to load the metadata.

strict_email

type: Boolean **default:** false

Deprecated since version 4.1: The **strict_email** option was deprecated in Symfony 4.1. Use the new **email_validation_mode** option instead.

If this option is enabled, the *egulias/email-validator*²⁹ library will be used by the *Email* constraint validator. Otherwise, the validator uses a simple regular expression to validate email addresses.

email_validation_mode

type: string **default:** loose

It controls the way email addresses are validated by the *Email* validator. The possible values are:

- **loose**, it uses a simple regular expression to validate the address (it checks that at least one @ character is present, etc.). This validation is too simple and it's recommended to use the **html5** validation instead;
- **html5**, it validates email addresses using the same regular expression defined in the HTML5 standard, making the backend validation consistent with the one provided by browsers;
- **strict**, it uses the *egulias/email-validator*³⁰ library (which you must install separately) to validate the addresses according to the *RFC 5322*³¹.

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:

28. <https://haveibeenpwned.com/>

29. <https://github.com/egulias/EmailValidator>

30. <https://github.com/egulias/EmailValidator>

31. <https://tools.ietf.org/html/rfc5322>

Listing 1-24

```
1 # config/packages/framework.yaml
2 framework:
3     validation:
4         mapping:
5             paths:
6                 - "%kernel.project_dir%/validation/"
```

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*³² 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.

For more information, see Using Serialization Groups Annotations.

32. <http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/caching.html>

name_converter

type: string

The name converter to use. The *CamelCaseToSnakeCaseNameConverter*³³ name converter can be 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|int **default:** %kernel.debug%

Use the application logger instead of the PHP logger for logging PHP errors. When an integer value is used, it also sets the log level. Those integer values must be the same used in the *error_reporting PHP option*³⁴.

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`, `cache.adapter.memcached` and `cache.adapter.pdo`.

33. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Serializer/NameConverter/CamelCaseToSnakeCaseNameConverter.php>

34. <https://secure.php.net/manual/en/errorfunc.configuration.php#ini.error-reporting>

There's also a special adapter called `cache.adapter.array` which stores contents in memory using a PHP array and it's used to disable caching (mostly on the `dev` environment).



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.default_doctrine_provider` service.

`default_psr6_provider`

type: string

The service name to use as your default PSR-6 provider. It is available as the `cache.default_psr6_provider` service.

`default_redis_provider`

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

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

`default_memcached_provider`

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

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

`default_pdo_provider`

type: string **default:** `doctrine.dbal.default_connection`

The service id of the database connection, which should be either a PDO or a Doctrine DBAL instance. The provider is available as the `cache.default_pdo_provider` 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.

To configure a Redis cache pool with a default lifetime of 1 hour, do the following:

Listing 1-25

```
1 # config/packages/framework.yaml
2 framework:
3   cache:
4     pools:
5       cache.mycache:
6         adapter: cache.adapter.redis
7         default_lifetime: 3600
```

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 service name of the adapter to use. You can specify one of the default services that follow the pattern `cache.adapter.[type]`. Alternatively you can specify another cache pool as base, which will make this pool inherit the settings from the base pool as defaults.



Your service MUST implement the `Psr\Cache\CacheItemPoolInterface` interface.

public

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

Whether your service should be public or not.

tags

type: boolean | string **default:** `null`

Whether your service should be able to handle tags or not. Can also be the service id of another cache pool where tags will be stored.

default_lifetime

type: integer

Default lifetime of your cache items in seconds.

provider

type: string

Overwrite the default service name or DSN respectively, if you do not want to use what is configured as **default_X_provider** under **cache**. See the description of the default provider setting above for the type of adapter you use for information on how to specify the provider.

clearer

type: string

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

For more information, see `Psr6CacheClearer`³⁵.

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*³⁶ strategies and more generally, when you need to abstract out the actual deployment directory (for example, when warming caches offline).

lock

type: string | array

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.

enabled

type: boolean **default:** true

Whether to enable the support for lock or not. This setting is automatically set to **true** when one of the child settings is configured.

resources

type: array

A list of lock stores to be created by the framework extension.

Listing 1-26

```
1 # config/packages/lock.yaml
2 framework:
3     # these are all the supported lock stores
4     lock: ~
5     lock: 'flock'
6     lock: 'flock:///path/to/file'
7     lock: 'semaphore'
8     lock: 'memcached://m1.docker'
9     lock: ['memcached://m1.docker', 'memcached://m2.docker']
10    lock: 'redis://r1.docker'
11    lock: ['redis://r1.docker', 'redis://r2.docker']
12    lock: '%env(MEMCACHED_OR_REDIS_URL)%'
13
14 # named locks
15 lock:
```

35. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/CacheClearer/Psr6CacheClearer.php>

36. <http://martinfowler.com/bliki/BlueGreenDeployment.html>

```

16     invoice: ['redis://r1.docker', 'redis://r2.docker']
17     report: 'semaphore'

```

New in version 4.2: The `flock://` store was introduced in Symfony 4.2.

name

type: prototype

Name of the lock you want to create.



If you want to use the *RetryTillSaveStore* for non-blocking locks, you can do it by *decorating the store* service:

Listing 1-27

```

1 lock.invoice.retry_till_save.store:
2     class: Symfony\Component\Lock\Store\RetryTillSaveStore
3     decorates: lock.invoice.store
4     arguments: ['@lock.invoice.retry.till.save.store.inner', 100, 50]

```

workflows

type: array

A list of workflows to be created by the framework extension:

Listing 1-28

```

1 # config/packages/workflow.yaml
2 framework:
3     workflows:
4         my_workflow:
5             # ...

```

See also the article about using workflows in Symfony applications.

enabled

type: boolean **default:** false

Whether to enable the support for workflows or not. This setting is automatically set to **true** when one of the child settings is configured.

name

type: prototype

Name of the workflow you want to create.

audit_trail

type: bool

If set to **true**, the *AuditTrailListener*³⁷ will be enabled.

initial_marking

type: string | array

One of the **places** or **empty**. If not null and the supported object is not already initialized via the workflow, this place will be set.

37. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Workflow/EventListener/AuditTrailListener.php>

marking_store

type: array

Each marking store can define any of these options:

- arguments (**type:** array)
- service (**type:** string)
- type (**type:** string **allow value:** 'method')

metadata

type: array

Metadata available for the workflow configuration. Note that **places** and **transitions** can also have their own **metadata** entry.

places

type: array

All available places (**type:** string) for the workflow configuration.

supports

type: string | array

The FQCN (fully-qualified class name) of the object supported by the workflow configuration or an array of FQCN if multiple objects are supported.

support_strategy

type: string

transitions

type: array

Each marking store can define any of these options:

- from (**type:** string or array) value from the places, multiple values are allowed for both **workflow** and **state_machine**;
- guard (**type:** string) an *ExpressionLanguage* compatible expression to block the transition;
- name (**type:** string) the name of the transition;
- to (**type:** string or array) value from the places, multiple values are allowed only for **workflow**.

type

type: string **possible values:** 'workflow' or 'state_machine'

Defines the kind of workflow that is going to be created, which can be either a normal workflow or a state machine. Read *this article* to know their differences.



Chapter 2

Doctrine Configuration Reference (DoctrineBundle)

The DoctrineBundle integrates both the *DBAL* and *ORM* Doctrine projects in Symfony applications. All these options are configured under the **doctrine** key in your application configuration.

Listing 2-1

```
1  # displays the default config values defined by Symfony
2  $ php bin/console config:dump-reference doctrine
3
4  # displays the actual config values used by your application
5  $ php bin/console debug:config doctrine
```



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

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

```
1  doctrine:
2      dbal:
3          dbname:         database
4          host:           localhost
5          port:           1234
6          user:           user
7          password:       secret
8          driver:         pdo_mysql
9          # if the url option is specified, it will override the above config
10         url:            mysql://db_user:db_password@127.0.0.1:3306/db_name
```

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

```

11     # the DBAL driverClass option
12     driver_class: App\DBAL\MyDatabaseDriver
13     # the DBAL driverOptions option
14     options:
15         foo: bar
16     path: '%kernel.project_dir%/var/data/data.sqlite'
17     memory: true
18     unix_socket: /tmp/mysql.sock
19     # the DBAL wrapperClass option
20     wrapper_class: App\DBAL\MyConnectionWrapper
21     charset: UTF8
22     logging: '%kernel.debug%'
23     platform_service: App\DBAL\MyDatabasePlatformService
24     server_version: '5.6'
25     mapping_types:
26         enum: string
27     types:
28         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 **5.5999999999999996447286321199499070644378662109375**).

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

```

1 doctrine:
2     dbal:
3         default_connection: default
4         connections:
5             default:
6                 dbname: Symfony
7                 user: root
8                 password: null
9                 host: localhost
10                server_version: '5.6'
11            customer:
12                dbname: customer
13                user: root
14                password: null
15                host: localhost
16                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

```
1 doctrine:
2   orm:
3     auto_mapping: true
4     # the standard distribution overrides this to be true in debug, false otherwise
5     auto_generate_proxy_classes: false
6     proxy_namespace: Proxies
7     proxy_dir: '%kernel.cache_dir%/doctrine/orm/Proxies'
8     default_entity_manager: default
9     metadata_cache_driver: array
10    query_cache_driver: array
11    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

```
1 doctrine:
2   orm:
3     # ...
4     query_cache_driver:
5       # ...
6     metadata_cache_driver:
7       # ...
8     result_cache_driver:
9       # ...
10    connection: ~
11    class_metadata_factory_name: Doctrine\ORM\Mapping\ClassMetadataFactory
12    default_repository_class: Doctrine\ORM\EntityRepository
13    auto_mapping: false
14    hydrators:
15      # ...
16    mappings:
17      # ...
18    dql:
19      # ...
20    filters:
21      # ...
```

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

```
1 doctrine:
2   orm:
3     auto_mapping: true
4     # each caching driver type defines its own config options
5     metadata_cache_driver: apc
6     result_cache_driver:
```

```

7         type: memcache
8         host: localhost
9         port: 11211
10        instance_class: Memcache
11        # the 'service' type requires to define the 'id' option too
12        query_cache_driver:
13            type: service
14            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

```

1 doctrine:
2     # ...

```

```

3     orm:
4         # ...
5         auto_mapping: true
6         mappings:
7             # ...
8             AppBundle:
9                 type: xml
10                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

```

1 doctrine:
2     # ...
3     orm:
4         # ...
5         mappings:
6             # ...
7             SomeEntityNamespace:
8                 type: annotation
9                 dir: '%kernel.project_dir%/src/Entity'
10                is_bundle: false
11                prefix: App\Entity
12                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

Security Configuration Reference (SecurityBundle)

The SecurityBundle integrates the *Security component* in Symfony applications. All these options are configured under the **security** key in your application configuration.

Listing 3-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference security
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config security
```



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

Configuration

Basic Options:

- `access_denied_url`
- `always_authenticate_before_granting`
- `erase_credentials`
- `hide_user_not_found`
- `session_fixation_strategy`

Advanced Options:

Some of these options define tens of sub-options and they are explained in separate articles:

- `access_control`
- `encoders`
- `firewalls`

- providers
- role_hierarchy

access_denied_url

type: string **default:** null

Defines the URL where the user is redirected after a **403** HTTP error (unless you define a custom access deny handler). Example: `/no-permission`

always_authenticate_before_granting

type: boolean **default:** false

If **true**, the user is asked to authenticate before each call to the `isGranted()` method in services and controllers or `is_granted()` from templates.

erase_credentials

type: boolean **default:** true

If **true**, the `eraseCredentials()` method of the user object is called after authentication.

hide_user_not_found

type: boolean **default:** true

If **true**, when a user is not found a generic exception of type *BadCredentialsException*¹ is thrown with the message "Bad credentials".

If **false**, the exception thrown is of type *UsernameNotFoundException*² and it includes the given not found username.

session_fixation_strategy

type: string **default:** `SessionAuthenticationStrategy::MIGRATE`

*Session Fixation*³ is a security attack that permits an attacker to hijack a valid user session. Applications that don't assign new session IDs when authenticating users are vulnerable to this attack.

The possible values of this option are:

- **NONE** constant from *SessionAuthenticationStrategy*⁴ Don't change the session after authentication. This is **not recommended**.
- **MIGRATE** constant from *SessionAuthenticationStrategy*⁵ The session ID is updated, but the rest of session attributes are kept.
- **INVALIDATE** constant from *SessionAuthenticationStrategy*⁶ The entire session is regenerated, so the session ID is updated but all the other session attributes are lost.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/Exception/BadCredentialsException.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/Exception/UsernameNotFoundException.php>

3. https://www.owasp.org/index.php/Session_fixation

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Http/Session/SessionAuthenticationStrategy.php>

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Http/Session/SessionAuthenticationStrategy.php>

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Http/Session/SessionAuthenticationStrategy.php>

access_control

Defines the security protection of the URLs of your application. It's used for example to trigger the user authentication when trying to access to the backend and to allow anonymous users to the login form page.

This option is explained in detail in *How Does the Security access_control Work?*.

encoders

This option defines the algorithm used to *encode* the password of the users. Although Symfony calls it "*password encoding*" for historical reasons, this is in fact, "*password hashing*".

If your app defines more than one user class, each of them can define its own encoding algorithm. Also, each algorithm defines different config options:

Listing 3-2

```
1  # config/packages/security.yaml
2  security:
3      # ...
4
5      encoders:
6          # auto encoder with default options
7          App\Entity\User: 'auto'
8
9          # auto encoder with custom options
10         App\Entity\User:
11             algorithm: 'auto'
12             cost:      15
13
14         # Sodium encoder with default options
15         App\Entity\User: 'sodium'
16
17         # Sodium encoder with custom options
18         App\Entity\User:
19             algorithm: 'sodium'
20             memory_cost: 16384 # Amount in KiB. (16384 = 16 MiB)
21             time_cost: 2      # Number of iterations
22             threads: 4        # Number of parallel threads
23
24         # PBKDF2 encoder using SHA512 hashing with default options
25         App\Entity\User: 'sha512'
```

Deprecated since version 4.3: The **threads** configuration option was deprecated in Symfony 4.3. No alternative is provided because starting from Symfony 5.0 this value will be hardcoded to **1** (one thread).

New in version 4.3: The **sodium** algorithm was introduced in Symfony 4.3. In previous Symfony versions it was called **argon2i**.



You can also create your own password encoders as services and you can even select a different password encoder for each user instance. Read *this article* for more details.

Using the Sodium Password Encoder

New in version 4.3: The **SodiumPasswordEncoder** was introduced in Symfony 4.3. In previous Symfony versions it was called **Argon2iPasswordEncoder**.

It uses the *Argon2 key derivation function*⁷ and it's the encoder recommended by Symfony. Argon2 support was introduced in PHP 7.2, but if you use an earlier PHP version, you can install the *libsodium*⁸ PHP extension.

The encoded passwords are **96** characters long, but due to the hashing requirements saved in the resulting hash this may change in the future, so make sure to allocate enough space for them to be persisted. Also, passwords include the *cryptographic salt*⁹ inside them (it's generated automatically for each new password) so you don't have to deal with it.

Using the "auto" Password Encoder

It selects automatically the best possible encoder. Currently, it tries to use Sodium by default and falls back to the *bcrypt password hashing function*¹⁰ if not possible. In the future, when PHP adds new hashing techniques, it may use different password hashers.

It produces encoded passwords with **60** characters long, so make sure to allocate enough space for them to be persisted. Also, passwords include the *cryptographic salt*¹¹ inside them (it's generated automatically for each new password) so you don't have to deal with it.

Its only configuration option is **cost**, which is an integer in the range of **4-31** (by default, **13**). Each single increment of the cost **doubles the time** it takes to encode a password. It's designed this way so the password strength can be adapted to the future improvements in computation power.

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 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 PBKDF2 Encoder

Using the *PBKDF2*¹² encoder is no longer recommended since PHP added support for Sodium and BCrypt. Legacy application still using it are encouraged to upgrade to those newer encoding algorithms.

firewalls

This is arguably the most important option of the security config file. It defines the authentication mechanism used for each URL (or URL pattern) of your application:

Listing 3-3

```
1  # config/packages/security.yaml
2  security:
3      # ...
4      firewalls:
5          # 'main' is the name of the firewall (can be chosen freely)
6          main:
7              # 'pattern' is a regular expression matched against the incoming
8              # request URL. If there's a match, authentication is triggered
9              pattern: ^/admin
```

7. <https://en.wikipedia.org/wiki/Argon2>

8. <https://pecl.php.net/package/libsodium>

9. [https://en.wikipedia.org/wiki/Salt_\(cryptography\)](https://en.wikipedia.org/wiki/Salt_(cryptography))

10. <https://en.wikipedia.org/wiki/Bcrypt>

11. [https://en.wikipedia.org/wiki/Salt_\(cryptography\)](https://en.wikipedia.org/wiki/Salt_(cryptography))

12. <https://en.wikipedia.org/wiki/PBKDF2>

```
10         # the rest of options depend on the authentication mechanism
11         # ...
```

Read this article to learn about how to restrict firewalls by host and HTTP methods.

In addition to some common config options, the most important firewall options depend on the authentication mechanism, which can be any of these:

Listing 3-4

```
1  # config/packages/security.yaml
2  security:
3      # ...
4      firewalls:
5          main:
6              # ...
7              x509:
8                  # ...
9              remote_user:
10                 # ...
11             simple_preauth:
12                 # ...
13             guard:
14                 # ...
15             form_login:
16                 # ...
17             form_login_ldap:
18                 # ...
19             json_login:
20                 # ...
21             simple_form:
22                 # ...
23             http_basic:
24                 # ...
25             http_basic_ldap:
26                 # ...
27             http_digest:
28                 # ...
```

form_login Authentication

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 *Using the form_login Authentication Provider*.

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.

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.

Options Related to 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`.

Options Related to 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.

`logout_on_user_change`

type: boolean **default:** true

Deprecated since version 4.1: The `logout_on_user_change` option was deprecated in Symfony 4.1.

If **false** this option makes Symfony to not trigger a logout when the user has changed. Doing that is deprecated, so this option should set to **true** or unset to avoid getting deprecation messages.

The user is considered to have changed when the user class implements *EquatableInterface*¹³ and the `isEqualTo()` method returns **false**. Also, when any of the properties required by the *UserInterface*¹⁴ (like the username, password or salt) changes.

`success_handler`

type: string **default:** 'security.logout.success_handler'

The service ID used for handling a successful logout. The service must implement *LogoutSuccessHandlerInterface*¹⁵.

`csrf_parameter`

type: string **default:** '_csrf_token'

The name of the parameter that stores the CSRF token value.

`csrf_token_generator`

type: string **default:** null

The `id` of the service used to generate the CSRF tokens. Symfony provides a default service whose ID is `security.csrf.token_manager`.

`csrf_token_id`

type: string **default:** 'logout'

An arbitrary string used to generate the token value (and check its validity afterwards).

13. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/User/EquatableInterface.php>

14. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/User/UserInterface.php>

15. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Http/Logout/LogoutSuccessHandlerInterface.php>

LDAP Authentication

There are several options for connecting against an LDAP server, using the `form_login_ldap`, `http_basic_ldap` and `json_login_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`, `http_basic` and `json_login` authentication providers. Use `form_login_ldap`, `http_basic_ldap` and `json_login_ldap`, which will attempt to `bind` against an 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 an LDAP server, you will need to use the *LDAP User Provider* and any of these authentication providers: `form_login_ldap` or `http_basic_ldap` or `json_login_ldap`.

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

```
1 # config/packages/security.yaml
2 security:
3   # ...
4
```

```

5     firewalls:
6         somename:
7             # ...
8             context: my_context
9         othername:
10            # ...
11            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.

User Checkers

During the authentication of a user, additional checks might be required to verify if the identified user is allowed to log in. Each firewall can include a **user_checker** option to define the service used to perform those checks.

Learn more about user checkers in *How to Create and Enable Custom User Checkers*.

providers

This options defines how the application users are loaded (from a database, an LDAP server, a configuration file, etc.) Read the following articles to learn more about each of those providers:

- Load users from a database
- Load users from an LDAP server
- Load users from a configuration file
- Create your own user provider

role_hierarchy

Instead of associating many roles to users, this option allows you to define role inheritance rules by creating a role hierarchy, as explained in *Hierarchical Roles*.



Chapter 4

Mailer Configuration Reference (SwiftmailerBundle)

The SwiftmailerBundle integrates the Swiftmailer library in Symfony applications to *send emails*. All these options are configured under the **swiftmailer** key in your application configuration.

Listing 4-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference swiftmailer
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config swiftmailer
```



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

Configuration

- antiflood
 - sleep
 - threshold
- auth_mode
- command
- delivery_addresses
- delivery_whitelist
- disable_delivery
- encryption
- host
- local_domain
- logging

- password
- port
- sender_address
- source_ip
- spool
 - path
 - type
- timeout
- transport
- url
- username

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 *Sending Emails with Mailer*.

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_ReducingPlugin`. 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: `url`, `transport`, `username`, `password`, `host`, `port`, `timeout`, `source_ip`, `local_domain`, `encryption`, `auth_mode`. For details, see: Configuration Based on Environment Variables.

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

```
1 swiftmailer:
2     default_mailer: second_mailer
3     mailers:
4         first_mailer:
5             # ...
6         second_mailer:
7             # ...
```

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

Listing 4-3

```
1 // ...
2
3 // returns the first mailer
4 $container->get('swiftmailer.mailer.first_mailer');
5
6 // also returns the second mailer since it is the default mailer
7 $container->get('swiftmailer.mailer');
8
9 // returns the second mailer
10 $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

```
1 # config/services.yaml
2 services:
3     _defaults:
4         bind:
5             # this injects the second mailer when type-hinting constructor arguments with \Swift_Mailer
6             \Swift_Mailer: '@swiftmailer.mailer.second_mailer'
7             # this injects the second mailer when a service constructor argument is called $specialMailer
8             $specialMailer: '@swiftmailer.mailer.second_mailer'
9
10    App\Some\Service:
11        # this injects the second mailer only for this argument of this service
12        $differentMailer: '@swiftmailer.mailer.second_mailer'
13
14    # ...
```



Chapter 5

Twig Configuration Reference (TwigBundle)

The TwigBundle integrates the Twig library in Symfony applications to render templates. All these options are configured under the **twig** key in your application configuration.

Listing 5-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference twig
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config twig
```



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

Configuration

- auto_reload
- autoescape
- autoescape_service
- autoescape_service_method
- base_template_class
- cache
- charset
- date
 - format
 - interval_format
 - timezone
- debug
- default_path
- exception_controller
- form_themes

- `globals`
- `number_format`
 - `decimals`
 - `decimal_point`
 - `thousands_separator`
- `optimizations`
- `paths`
- `strict_variables`

`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 attacks¹ 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`

1. https://en.wikipedia.org/wiki/Cross-site_scripting

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 | false **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 **false** 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.

`interval_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.

default_path

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

The path to the directory where Symfony will look for the application Twig templates by default. If you store the templates in more than one directory, use the `paths` option too.

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*²) 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 an *event listener* to the `kernel.exception` event.

form_themes

type: array of string **default:** `['form_div_layout.html.twig']`

Defines one or more *form themes* which are applied to all the forms of the application:

Listing 5-2

```
1 # config/packages/twig.yaml
2 twig:
3     form_themes: ['bootstrap_4_layout.html.twig', 'form/my_theme.html.twig']
4     # ...
```

The order in which themes are defined is important because each theme overrides all the previous one. When rendering a form field whose block is not defined in the form theme, Symfony falls back to the previous themes until the first one.

These global themes are applied to all forms, even those which use the `form_theme` Twig tag, but you can disable global themes for specific forms.

globals

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

It defines the global variables injected automatically into all Twig templates. Learn more about *Twig global variables*.

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.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/TwigBundle/Controller/ExceptionController.php>

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

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*³.

paths

type: array **default:** null

Defines the directories where application templates are stored in addition to the directory defined in the `default_path` option:

Listing 5-3

```
1 # config/packages/twig.yaml
2 twig:
3   # ...
4   paths:
5     'email/default/templates': ~
6     'backend/templates': 'admin'
```

Read more about template directories and namespaces.

strict_variables

type: boolean **default:** false

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`.

3. <https://twig.symfony.com/doc/2.x/api.html#optimizer-extension>



Chapter 6

Logging Configuration Reference (MonologBundle)

The MonologBundle integrates the Monolog *logging* library in Symfony applications. All these options are configured under the **monolog** key in your application configuration.

Listing 6-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference monolog
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config monolog
```



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



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



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.

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



Chapter 7

Profiler Configuration Reference (WebProfilerBundle)

The WebProfilerBundle is a **development tool** that provides detailed technical information about each request execution and displays it in both the web debug toolbar and the *profiler*. All these options are configured under the **web_profiler** key in your application configuration.

Listing 7-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference web_profiler
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config web_profiler
```



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



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

Configuration

- `excluded_ajax_paths`
- `intercept_redirects`
- `toolbar`

`excluded_ajax_paths`

type: string **default:** `'^/((index|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 if they are heavy and you want to exclude some of them.

`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.

`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.



Chapter 8

Debug Configuration Reference (DebugBundle)

The DebugBundle integrates the *VarDumper component* in Symfony applications. All these options are configured under the **debug** key in your application configuration.

Listing 8-1

```
1 # displays the default config values defined by Symfony
2 $ php bin/console config:dump-reference debug
3
4 # displays the actual config values used by your application
5 $ php bin/console debug:config debug
```



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

Configuration

- `dump_destination`
- `max_items`
- `min_depth`
- `max_string_length`

`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 8-2

```
1 # config/packages/debug.yaml
2 debug:
3     dump_destination: php://stderr
```

Configure it to **"tcp://%env(VAR_DUMPER_SERVER)%"** in order to use the ServerDumper feature.



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
- Container Build Time

Charset

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

This option defines the charset that is used in the application. This value is exposed via the `kernel.charset` configuration parameter and the `getCharset()`² method.

To change this value, override the `getCharset()` method and return another charset:

Listing 9-1

```
1 // src/Kernel.php
2 use Symfony\Component\HttpKernel\Kernel as BaseKernel;
3 // ...
4
5 class Kernel extends BaseKernel
6 {
7     public function getCharset()
8     {
9         return 'ISO-8859-1';
10    }
11 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

Kernel Name

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

Deprecated since version 4.2: The `kernel.name` parameter and the `Kernel::getName()` method were deprecated in Symfony 4.2. If you need a unique ID for your kernels use the `kernel.container_class` parameter or the `Kernel::getContainerClass()` method.

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*.

This value is exposed via the `kernel.name` configuration parameter and the `getName()`³ method.

To change this setting, override the `getName()` 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`.

Project Directory

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

This returns the absolute path of the root directory of your Symfony project, which is used by applications to perform operations with file paths relative to the project's root directory.

By default, its value is calculated automatically as the directory where the main `composer.json` file is stored. This value is exposed via the `kernel.project_dir` configuration parameter and the `getProjectDir()`⁴ method.

If you don't use Composer, or have moved the `composer.json` file location or have deleted it entirely (for example in the production servers), you can override the `getProjectDir()`⁵ method to return the right project directory:

Listing 9-2

```
1 // src/Kernel.php
2 use Symfony\Component\HttpKernel\Kernel as BaseKernel;
3 // ...
4
5 class Kernel extends BaseKernel
6 {
7     // ...
8
9     public function getProjectDir(): string
10     {
11         return \dirname(__DIR__);
12     }
13 }
```

Cache Directory

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

This returns the absolute path of the cache directory of your Symfony project. It's calculated automatically based on the current environment.

This value is exposed via the `kernel.cache_dir` configuration parameter and the `getCacheDir()`⁶ method. To change this setting, override the `getCacheDir()` method to return the right cache directory.

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

Log Directory

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

This returns the absolute path of the log directory of your Symfony project. It's calculated automatically based on the current environment.

This value is exposed via the `kernel.log_dir` configuration parameter and the `getLogDir()` method. To change this setting, override the `getLogDir()` method to return the right log directory.

Container Build Time

type: string **default:** the result of executing `time()`

Symfony follows the *reproducible builds*⁸ philosophy, which ensures that the result of compiling the exact same source code doesn't produce different results. This helps checking that a given binary or executable code was compiled from some trusted source code.

In practice, the compiled *service container* of your application will always be the same if you don't change its source code. This is exposed via these configuration parameters:

- `container.build_hash`, a hash of the contents of all your source files;
- `container.build_time`, a timestamp of the moment when the container was built (the result of executing PHP's `time`⁹ function);
- `container.build_id`, the result of merging the two previous parameters and encoding the result using CRC32.

Since the `container.build_time` value will change every time you compile the application, the build will not be strictly reproducible. If you care about this, the solution is to use another configuration parameter called `kernel.container_build_time` and set it to a non-changing build time to achieve a strict reproducible build:

Listing 9-3

```
1 # config/services.yaml
2 parameters:
3     # ...
4     kernel.container_build_time: '1234567890'
```

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Kernel.php>

8. https://en.wikipedia.org/wiki/Reproducible_builds

9. <https://secure.php.net/manual/en/function.time.php>



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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Overridden options	<ul style="list-style-type: none">• compound
Parent type	<i>FormType</i>
Class	<i>TextType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/TextType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 11-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 11-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 11-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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

From an HTTP perspective, submitted data is always a string or an array of strings. So by default, the form will treat any empty string as null. If you prefer to get an empty string, explicitly set the **empty_data** option to an 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 11-4

```
$builder->add('name', null, [
    '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:** []

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 11-5

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 11-6

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 11-7

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 11-8

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 11-9

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 11-10

```
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 11-11

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 11-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>TextareaType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 12-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/TextareaType.php>



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

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 12-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 12-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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

2. <https://symfony.com/doc/current/bundles/FOSCKEditorBundle/index.html>

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 12-4

```
$builder->add('name', null, [
    '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:** []

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 12-5

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 12-6

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 12-7

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 12-8

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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 12-9

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 12-10

```
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 12-11

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 12-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>EmailType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 13-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/EmailType.php>

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 13-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 13-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 13-4

```
$builder->add('name', null, [
    '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:** []

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 13-5

```
1 public function configureOptions(OptionsResolver $resolver)  
2 {  
3     $resolver->setDefaults([  
4         'error_mapping' => [  
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 the **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 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 13-6

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.?' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 13-7

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 13-8

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 13-9

```

1 {{ form_label(form.name, 'Your name') }}

```

label_attr

type: array **default:** []

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 13-10

```

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 13-11

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

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

2. <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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 13-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

`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.

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	<ul style="list-style-type: none">• grouping• rounding_mode
Overridden options	<ul style="list-style-type: none">• compound• scale
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• invalid_message• invalid_message_parameters• label• label_attr• label_format• mapped• required

	<ul style="list-style-type: none"> • row_attr
Parent type	<i>FormType</i>
Class	<i>IntegerType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 14-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

grouping

type: boolean **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

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/IntegerType.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.php>

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

Deprecated since version 4.2: The **scale** option is deprecated since Symfony 4.2 and will be removed in 5.0.

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 overridden to **0** for **IntegerType**.

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 14-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 14-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 14-4

```
$builder->add('name', null, [
    '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:** []

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 14-5

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 14-6

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 14-7

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 14-8

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 14-9 1 $builder->add('someField', SomeFormType::class, [  
2           // ...  
3           'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4           'invalid_message_parameters' => ['%num%' => 6],  
5       ]);
```

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 14-10 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 14-11 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 14-12 1 // ...  
2 $profileFormBuilder->add('address', AddressType::class, [  
3     'label_format' => 'form.address.%name%',  
4 ]);  
5  
6 $invoiceFormBuilder->add('invoice', AddressType::class, [  
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 14-13

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• currency• divisor• grouping• rounding_mode• scale
Overridden options	<ul style="list-style-type: none">• compound
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• invalid_message• invalid_message_parameters• label• label_attr• label_format• mapped• required

	<ul style="list-style-type: none"> • row_attr
Parent type	<i>FormType</i>
Class	<i>MoneyType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 15-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

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 15-2

```

1 use Symfony\Component\Form\Extension\Core\Type\MoneyType;
2 // ...
3
4 $builder->add('price', MoneyType::class, [
5     'divisor' => 100,
6 ]);

```

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: boolean **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**.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/MoneyType.php>
2. https://en.wikipedia.org/wiki/ISO_4217

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*³:

- `NumberToLocalizedStringTransformer::ROUND_DOWN` Round towards zero. It rounds 1.4 to 1 and -1.4 to -1.
- `NumberToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity. It rounds 1.4 to 1 and -1.4 to -2.
- `NumberToLocalizedStringTransformer::ROUND_UP` Round away from zero. It rounds 1.4 to 2 and -1.4 to -2.
- `NumberToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity. It rounds 1.4 to 2 and -1.4 to -1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down. It rounds 2.5 and 1.6 to 2, 1.5 and 1.4 to 1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor. It rounds 2.5, 1.6 and 1.5 to 2 and 1.4 to 1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up. It rounds 2.5 to 3, 1.6 and 1.5 to 2 and 1.4 to 1.

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

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*:

attr

type: array **default:** []

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:

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/DataTransformer/NumberToLocalizedStringTransformer.php>

Listing 15-3

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the form type row element.

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 15-4

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 15-5

```
$builder->add('name', null, [
    '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:** []

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 15-6

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 15-7

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 15-8 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

```
Listing 15-9 1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 15-10 1 $builder->add('someField', SomeFormType::class, [
2     // ...
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',
4     'invalid_message_parameters' => ['%num%' => 6],
5 ]);
```

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 15-11 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 15-12

```
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 15-13

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 15-14

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

Form Variables

Variable	Type	Usage
money_pattern	string	The format to use to display the money, including the currency.

4. <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	<ul style="list-style-type: none">• grouping• html5• input• scale• rounding_mode
Overridden options	<ul style="list-style-type: none">• compound
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• invalid_message• invalid_message_parameters• label• label_attr• label_format• mapped• required• row_attr

Parent type	<i>FormType</i>
Class	<i>NumberType¹</i>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 16-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

grouping

type: boolean **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**.

html5

type: boolean **default:** false

New in version 4.3: The **html5** option was introduced in Symfony 4.3.

If set to **true**, the HTML input will be rendered as a native HTML5 **type="number"** form.

input

type: string **default:** number

New in version 4.3: The **input** option was introduced in Symfony 4.3.

The format of the input data - i.e. the format that the number is stored on your underlying object. Valid values are **number** and **string**. Setting this option to **string** can be useful if the underlying data is a string for precision reasons (for example, Doctrine uses strings for the **decimal** type).

scale

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`

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/NumberType.php>

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. It rounds 1.4 to 1 and -1.4 to -1.
- `NumberToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity. It rounds 1.4 to 1 and -1.4 to -2.
- `NumberToLocalizedStringTransformer::ROUND_UP` Round away from zero. It rounds 1.4 to 2 and -1.4 to -2.
- `NumberToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity. It rounds 1.4 to 2 and -1.4 to -1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down. It rounds 2.5 and 1.6 to 2, 1.5 and 1.4 to 1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor. It rounds 2.5, 1.6 and 1.5 to 2 and 1.4 to 1.
- `NumberToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up. It rounds 2.5 to 3, 1.6 and 1.5 to 2 and 1.4 to 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.

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 16-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

data

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

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/DataTransformer/NumberToLocalizedStringTransformer.php>

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 16-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 16-4

```
$builder->add('name', null, [
    '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:** []

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 16-5 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 16-6 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 16-7 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 16-8

```

1 {{ form_help(form.name, 'Your name', {
2   'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 16-9

```

1 $builder->add('someField', SomeFormType::class, [
2   // ...
3   'invalid_message' => 'You entered an invalid value, it should include %num% letters',
4   'invalid_message_parameters' => ['%num%' => 6],
5 ]);
```

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 16-10

```

1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 16-11

```

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 16-12

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type `row`:

Listing 16-13

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



Chapter 17

PasswordType Field

The **PasswordType** field renders an input password text box.

Rendered as	input password field
Options	<ul style="list-style-type: none">• <code>always_empty</code>
Overridden options	<ul style="list-style-type: none">• <code>trim</code>
Inherited options	<ul style="list-style-type: none">• <code>attr</code>• <code>disabled</code>• <code>empty_data</code>• <code>error_bubbling</code>• <code>error_mapping</code>• <code>help</code>• <code>help_attr</code>• <code>help_html</code>• <code>label</code>• <code>label_attr</code>• <code>label_format</code>• <code>mapped</code>• <code>required</code>• <code>row_attr</code>
Parent type	<i>TextType</i>
Class	<i>PasswordType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/PasswordType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 17-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Field Options

always_empty

type: boolean **default:** true

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.

If 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* 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*:

attr

type: array **default:** []

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 17-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 17-3

```
$builder->add('name', null, [
    '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:** []

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 17-4

```

1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 17-5

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.*' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 17-6

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 17-7

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 17-8

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array default: []

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 17-9

```
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 17-10

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 17-11

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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. **0.95**), 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**.

When **symbol** is not **false**, the field will render the given string after the input.

Rendered as	input text field
Options	<ul style="list-style-type: none">• scale• symbol• type
Overridden options	<ul style="list-style-type: none">• compound
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• invalid_message• invalid_message_parameters• label• label_attr• label_format• mapped• required• row_attr

Parent type	<i>FormType</i>
Class	<i>PercentType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 18-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

scale

type: integer **default:** 0

By default, the input numbers are rounded. To allow for more decimal places, use this option.

symbol

type: boolean or string **default:** %

New in version 4.3: The **symbol** option was introduced in Symfony 4.3.

By default, fields are rendered with a percentage sign % after the input. Setting the value to **false** will not display the percentage sign. Setting the value to a **string** (e.g. ?), will show that string instead of the default % sign.

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 **0.55** or **55** on your object. The two "types" handle these two cases:

- **fractional** If your data is stored as a decimal (e.g. 0.55), 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 (0.55);
- **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.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/PercentType.php>

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 18-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 18-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 18-4

```
$builder->add('name', null, [
    '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:** []

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 18-5

```
1 public function configureOptions(OptionsResolver $resolver)  
2 {  
3     $resolver->setDefaults([  
4         'error_mapping' => [  
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 the **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 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 18-6

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.t' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 18-7

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 18-8

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 18-9

```

1 $builder->add('someField', SomeFormType::class, [
2     // ...
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',
4     'invalid_message_parameters' => ['%num%' => 6],
5 ]);

```

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 18-10 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array default: []

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 18-11 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 18-12 1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 18-13

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• attr• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>SearchType</i> ²

1. <http://diveintohtml5.info/forms.html#type-search>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/SearchType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 19-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 19-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 19-3

```
$builder->add('name', null, [
    '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:** []

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 19-4

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 19-5

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 19-6

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 19-7

```
1  {{ form_help(form.name, 'Your name', {  
2      'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3  }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 19-8

```
1  {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 19-9

```
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 19-10

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

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

Listing 19-11

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the *attr* option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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.

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	<ul style="list-style-type: none">• default_protocol
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>UrlType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/UrlType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 20-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Field Options

default_protocol

type: string **default:** http

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*:

attr

type: array **default:** []

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 20-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 20-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 20-4

```
$builder->add('name', null, [
    '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:** []

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 20-5

```

1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 20-6

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.*' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 20-7

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 20-8

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 20-9 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array default: []

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 20-10 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 20-11 1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 20-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>RangeType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 21-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/RangeType.php>

Basic Usage

Listing 21-2

```
1 use Symfony\Component\Form\Extension\Core\Type\RangeType;
2 // ...
3
4 $builder->add('name', RangeType::class, [
5     'attr' => [
6         'min' => 5,
7         'max' => 50
8     ]
9 ]);
```

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 21-3

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 21-4

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 21-5

```
$builder->add('name', null, [
    '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:** []

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 21-6

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 21-7

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 21-8

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 21-9

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 21-10

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 21-11

```
1 {{ form_label(form.name, 'Your name', {
2     'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}
3 }}
```

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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 21-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

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 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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>TelType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/TelType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 22-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 22-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 22-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 22-4

```
$builder->add('name', null, [
    '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:** `[]`

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 22-5

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 22-6

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 22-7

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 22-8

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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 22-9

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 22-10 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 22-11 1 // ...  
2 $profileFormBuilder->add('address', AddressType::class, [  
3     'label_format' => 'form.address.%name%',  
4 ]);  
5  
6 $invoiceFormBuilder->add('invoice', AddressType::class, [  
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 22-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr• trim
Parent type	<i>TextType</i>
Class	<i>ColorType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/ColorType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 23-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 23-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 23-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 23-4

```
$builder->add('name', null, [
    '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:** `[]`

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 23-5

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 23-6

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 23-7

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 23-8

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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 23-9

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 23-10 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 23-11 1 // ...  
2 $profileFormBuilder->add('address', AddressType::class, [  
3     'label_format' => 'form.address.%name%',  
4 ]);  
5  
6 $invoiceFormBuilder->add('invoice', AddressType::class, [  
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 23-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• choices• choice_attr• choice_label• choice_loader• choice_name• choice_translation_domain• choice_value• expanded• group_by• multiple• placeholder• preferred_choices
Overridden options	<ul style="list-style-type: none">• compound• empty_data• error_bubbling• trim
Inherited options	<ul style="list-style-type: none">• attr• by_reference• data

	<ul style="list-style-type: none"> • disabled • error_mapping • help • help_attr • help_html • inherit_data • label • label_attr • label_format • mapped • required • row_attr • translation_domain • label_translation_parameters • attr_translation_parameters • help_translation_parameters
Parent type	<i>FormType</i>
Class	<i>ChoiceType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 24-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Example Usage

The easiest way to use this field is to define the **choices** option to specify the choices as an associative array where the keys are the labels displayed to end users and the array values are the internal values used in the form field:

Listing 24-2

```

1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('isAttending', ChoiceType::class, [
5     'choices' => [
6         'Maybe' => null,
7         'Yes' => true,
8         'No' => false,
9     ],
10 ]);

```

This will create a **select** drop-down like this:

Attending?

Maybe
 Yes
 ✓ No

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/ChoiceType.php>

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 24-3 1 use App\Entity\Category;
2 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
3 // ...
4
5 $builder->add('category', ChoiceType::class, [
6     'choices' => [
7         new Category('Cat1'),
8         new Category('Cat2'),
9         new Category('Cat3'),
10        new Category('Cat4'),
11    ],
12    'choice_label' => function(Category $category, $key, $value) {
13        return strtoupper($category->getName());
14    },
15    'choice_attr' => function(Category $category, $key, $value) {
16        return ['class' => 'category_'.strtolower($category->getName())];
17    },
18    'group_by' => function(Category $category, $key, $value) {
19        // randomly assign things into 2 groups
20        return rand(0, 1) == 1 ? 'Group A' : 'Group B';
21    },
22    'preferred_choices' => function(Category $category, $key, $value) {
23        return $category->getName() == 'Cat2' || $category->getName() == 'Cat3';
24    },
25 ]];
```

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 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)

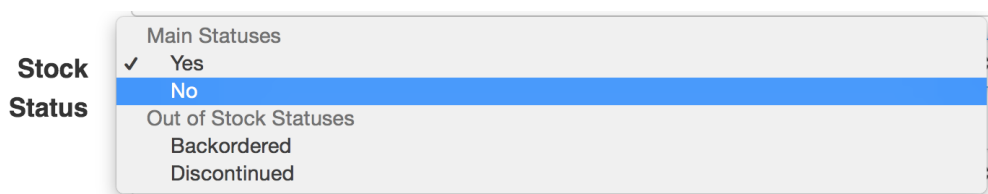
Normally, the array key of each item in the **choices** option is used as the text that's shown to the user. But that can be completely customized via the `choice_label` option. Check it out for more details.

Grouping Options

You can group the `<option>` elements of a `<select>` into `<optgroup>` by passing a multi-dimensional `choices` array:

Listing 24-4

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('stockStatus', ChoiceType::class, [
5     'choices' => [
6         'Main Statuses' => [
7             'Yes' => 'stock_yes',
8             'No' => 'stock_no',
9         ],
10        'Out of Stock Statuses' => [
11            'Backordered' => 'stock_backordered',
12            'Discontinued' => 'stock_discontinued',
13        ],
14    ],
15 ]);
```



To get fancier, use the `group_by` option.

Field Options

choices

type: array **default:** []

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 24-5

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('inStock', ChoiceType::class, [
5     'choices' => ['In Stock' => true, 'Out of Stock' => false],
6 ]);
```

If there are choice values that are not scalar or the stringified representation is not unique Symfony will use incrementing integers as values. When the form gets submitted the correct values with the correct types will be assigned to the model.

choice_attr

type: array, callable or string **default:** []

Use this to add additional HTML attributes to each choice. This can be an associative array where the keys match the choice keys and the values are the attributes 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 24-6 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('attending', ChoiceType::class, [
5     'choices' => [
6         'Yes' => true,
7         'No' => false,
8         'Maybe' => null,
9     ],
10    'choice_attr' => function($choice, $key, $value) {
11        // adds a class like attending_yes, attending_no, etc
12        return ['class' => 'attending_'.strtolower($key)];
13    },
14 ]);
```

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 24-7 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('attending', ChoiceType::class, [
5     'choices' => [
6         'yes' => true,
7         'no' => false,
8         'maybe' => null,
9     ],
10    'choice_label' => function ($choice, $key, $value) {
11        if (true === $choice) {
12            return 'Definitely!';
13        }
14
15        return strtoupper($key);
16
17        // or if you want to translate some key
18        //return 'form.choice.'.$key;
19    },
20 ]);
```

This method is called for *each* choice, passing you the **\$choice** and **\$key** from the choices array (additional **\$value** is related to choice_value). This will give you:

Attending

Definitely!

NO

☒ MAYBE

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 24-8 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('attending', ChoiceType::class, [
5     'choices' => [
6         new Status(Status::YES),
```



```

7         new Status(Status::NO),
8         new Status(Status::MAYBE),
9     ],
10    'choice_label' => 'displayName',
11 ];

```

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.

You can use an instance of *CallbackChoiceLoader*³ if you want to take advantage of lazy loading:

Listing 24-9

```

1 use Symfony\Component\Form\ChoiceList\Loader\CallbackChoiceLoader;
2 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
3 // ...
4
5 $builder->add('constants', ChoiceType::class, [
6     'choice_loader' => new CallbackChoiceLoader(function() {
7         return StaticClass::getConstants();
8     }),
9 ]);

```

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.



The configured value must be a valid form name. Make sure to only return valid names when using a callable. Valid form names must be composed of letters, digits, underscores, dashes and colons and must not start with a dash or a colon.

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.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/ChoiceList/Loader/ChoiceLoaderInterface.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/ChoiceList/Loader/CallbackChoiceLoader.php>

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 24-10

```
'choice_value' => function (MyOptionEntity $entity = null) {  
    return $entity ? $entity->getId() : '';  
},
```

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: string or callable **default:** null

You can group the `<option>` elements of a `<select>` into `<optgroup>` 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 24-11

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;  
2 // ...  
3  
4 $builder->add('publishAt', ChoiceType::class, [  
5     'choices' => [  
6         'now' => new \DateTime('now'),  
7         'tomorrow' => new \DateTime('+1 day'),  
8         '1 week' => new \DateTime('+1 week'),  
9         '1 month' => new \DateTime('+1 month'),  
10    ],  
11    'group_by' => function($choice, $key, $value) {  
12        if ($choice <= new \DateTime('+3 days')) {  
13            return 'Soon';  
14        } else {  
15            return 'Later';  
16        }  
17    },  
18    ]);
```

This groups the dates that are within 3 days into "Soon" and everything else into a "Later" `<optgroup>`:

Publish at

- Soon
 - ✓ now
 - tomorrow
 - 1 week
- Later
 - 1 month

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 24-12 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
5     'placeholder' => 'Choose an option',
6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 24-13 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
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 24-14 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, [
6     'required' => false,
7 ]);
```

preferred_choices

type: array, callable or string **default:** []

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 24-15 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
```

```

6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork' => 'muppets',
9         'Pirate' => 'arr',
10    ],
11    'preferred_choices' => ['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 24-16

```

1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  $builder->add('publishAt', ChoiceType::class, [
5      'choices' => [
6          'now' => new \DateTime('now'),
7          'tomorrow' => new \DateTime('+1 day'),
8          '1 week' => new \DateTime('+1 week'),
9          '1 month' => new \DateTime('+1 month'),
10     ],
11     'preferred_choices' => function ($choice, $key, $value) {
12         // prefer options within 3 days
13         return $choice <= 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 24-17

```

1  {{ 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 `[]` (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 24-18

```
$builder->add('name', null, [  
    '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:** []

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 24-19

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 24-20

```
1 use Symfony\Component\Form\Extension\Core\Type\EmailType;
2 use Symfony\Component\Form\Extension\Core\Type\FormType;
3 use Symfony\Component\Form\Extension\Core\Type\TextType;
4 // ...
5
6 $builder = $this->createFormBuilder($article);
7 $builder
8     ->add('title', TextType::class)
9     ->add(
10         $builder->create('author', FormType::class, ['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 24-21

```
$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 24-22

```
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 24-23

```

1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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:** []

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 24-24

```

1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 24-25

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 24-26

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 24-27

```
1 {{ form_help(form.name, 'Your name', {  
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 24-28

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 24-29


```

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 24-30

```

1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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.

4. <http://diveintohtml5.info/forms.html>

This is superficial and independent of 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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

```
Listing 24-31 $builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

`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.

`label_translation_parameters`

type: array **default:** []

New in version 4.3: The `label_translation_parameters` option was introduced in Symfony 4.3.

The content of the `label` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 24-32 1 # translations/messages.en.yaml
               2 form.order.id: 'Identifier of the order to %company%'
```

You can specify the placeholder values as follows:

```
Listing 24-33 1 $builder->add('id', null, [
               2     'label' => 'form.order.id',
               3     'label_translation_parameters' => [
               4         '%company%' => 'ACME Inc.',
               5     ],
               6 ]);
```

The `label_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

`attr_translation_parameters`

type: array **default:** []

New in version 4.3: The **attr_translation_parameters** option was introduced in Symfony 4.3.

The content of the **title** and **placeholder** values defined in the **attr** option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 24-34 1 # translations/messages.en.yaml
2 form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3 form.order.id.title: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

```
Listing 24-35 1 $builder->add('id', null, [
2     'attr' => [
3         'placeholder' => 'form.order.id.placeholder',
4         'title' => 'form.order.id.title',
5     ],
6     'attr_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The **attr_translation_parameters** option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

help_translation_parameters

type: array **default:** []

New in version 4.3: The **help_translation_parameters** option was introduced in Symfony 4.3.

The content of the **help** option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 24-36 1 # translations/messages.en.yaml
2 form.order.id.help: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

```
Listing 24-37 1 $builder->add('id', null, [
2     'help' => 'form.order.id.help',
3     'help_translation_parameters' => [
4         '%company%' => 'ACME Inc.',
5     ],
6 ]);
```

The **help_translation_parameters** option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

Field Variables

Variable	Type	Usage
multiple	boolean	The value of the multiple option.
expanded	boolean	The value of the expanded option.

Variable	Type	Usage
preferred_choices	array	A nested array containing the <code>ChoiceView</code> objects of choices which should be presented to the user with priority.
choices	array	A nested array containing the <code>ChoiceView</code> 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 <code>null</code> .
choice_translation_domain	mixed	<code>boolean</code> , <code>null</code> or <code>string</code> to determine if the value should be translated.
is_selected	callable	A callable which takes a <code>ChoiceView</code> 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.



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	<ul style="list-style-type: none">• choice_label• class• em• query_builder
Overridden options	<ul style="list-style-type: none">• choice_name• choice_value• choices• data_class
Inherited options	<p>from the <i>ChoiceType</i>:</p> <ul style="list-style-type: none">• choice_attr• choice_translation_domain• expanded• group_by• multiple• placeholder• preferred_choices• translation_domain• trim <p>from the <i>FormType</i>:</p> <ul style="list-style-type: none">• attr• data

	<ul style="list-style-type: none"> • disabled • empty_data • error_bubbling • error_mapping • help • help_attr • help_html • label • label_attr • label_format • mapped • required • row_attr • label_translation_parameters • attr_translation_parameters • help_translation_parameters
Parent type	<i>ChoiceType</i>
Class	<i>EntityType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 25-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Basic Usage

The **entity** type has just one required option: the entity which should be listed inside the choice field:

Listing 25-2

```
1 use App\Entity\User;
2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
3 // ...
4
5 $builder->add('users', EntityType::class, [
6     // looks for choices from this entity
7     'class' => User::class,
8
9     // uses the User.username property as the visible option string
10    'choice_label' => 'username',
11
12    // used to render a select box, check boxes or radios
13    // 'multiple' => true,
14    // 'expanded' => true,
15 ]);
```

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.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bridge/Doctrine/Form/Type/EntityType.php>

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 25-3

```
1 use App\Entity\User;
2 use Doctrine\ORM\EntityRepository;
3 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
4 // ...
5
6 $builder->add('users', EntityType::class, [
7     'class' => User::class,
8     'query_builder' => function (EntityRepository $er) {
9         return $er->createQueryBuilder('u')
10             ->orderBy('u.username', 'ASC');
11     },
12     'choice_label' => 'username',
13 ]);
```



Using form collections may result in making too many database requests to fetch related entities. This is known as the "*N + 1 query problem*" and it can be solved by joining related records when querying for Doctrine associations.

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 25-4

```
1 use App\Entity\User;
2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
3 // ...
4
5 $builder->add('users', EntityType::class, [
6     'class' => User::class,
7     'choices' => $group->getUsers(),
8 ]);
```

Select Tag, Checkboxes or Radio Buttons

This field may be rendered as one of several 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 25-5

```
1 use App\Entity\Category;
2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
3 // ...
4
5 $builder->add('category', EntityType::class, [
6     'class' => Category::class,
7     'choice_label' => 'displayName',
8 ]);
```

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 25-6

```
1 use App\Entity\Category;
2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
3 // ...
4
5 $builder->add('category', EntityType::class, [
6     'class' => Category::class,
7     'choice_label' => function ($category) {
8         return $category->getDisplayName();
9     }
10 ]);
```

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 25-7

```
1 use App\Entity\Genre;
2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
3 // ...
4
5 $builder->add('genre', EntityType::class, [
6     'class' => Genre::class,
7     'choice_label' => 'translations[en].name',
8 ]);
```

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).

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/PropertyAccess/PropertyPath.php>

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 at the 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.



The configured value must be a valid form name. Make sure to only return valid names when using a callable. Valid form names must be composed of letters, digits, underscores, dashes and colons and must not start with a dash or a colon.

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 25-8

```
'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:** []

Use this to add additional HTML attributes to each choice. This can be an associative array where the keys match the choice keys and the values are the attributes 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 25-9

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;  
2 // ...  
3  
4 $builder->add('attending', ChoiceType::class, [  
5     'choices' => [  
6         'Yes' => true,  
7         'No' => false,  
8         'Maybe' => null,  
9     ],  
10    'choice_attr' => function($choice, $key, $value) {  
11        // adds a class like attending_yes, attending_no, etc  
12        return ['class' => 'attending_'.$key];  
13    },  
14 ]);
```

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: string or callable **default:** null

You can group the `<option>` elements of a `<select>` into `<optgroup>` by passing a multi-dimensional array to **choices**. See the Grouping Options section about that.

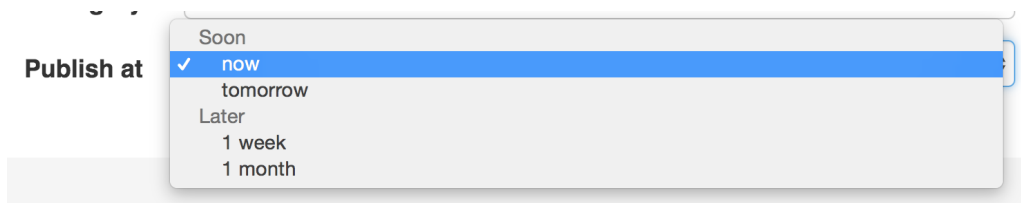
The **group_by** option is an alternative way to group choices, which gives you a bit more flexibility.

Take the following example:

Listing 25-10

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('publishAt', ChoiceType::class, [
5     'choices' => [
6         'now' => new \DateTime('now'),
7         'tomorrow' => new \DateTime('+1 day'),
8         '1 week' => new \DateTime('+1 week'),
9         '1 month' => new \DateTime('+1 month'),
10    ],
11    'group_by' => function($choice, $key, $value) {
12        if ($choice <= new \DateTime('+3 days')) {
13            return 'Soon';
14        } else {
15            return 'Later';
16        }
17    },
18 ]);
```

This groups the dates that are within 3 days into "Soon" and everything else into a "Later" `<optgroup>`:



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 25-11 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
                2 // ...
                3
                4 $builder->add('states', ChoiceType::class, [
                5     'placeholder' => 'Choose an option',
                6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 25-12 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
                2 // ...
                3
                4 $builder->add('states', ChoiceType::class, [
                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 25-13 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, [
                6     'required' => false,
                7 ]);
```

preferred_choices

type: array or callable **default:** []

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. This option expects an array of entity objects:

```
Listing 25-14 1 use App\Entity\User;
                2 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
                3 // ...
                4
                5 $builder->add('users', EntityType::class, [
                6     'class' => User::class,
                7     // this method must return an array of User entities
                8     'preferred_choices' => $group->getPreferredUsers(),
                9 ]);
```

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 1 `{{ form_widget(form.publishAt, { 'separator': '====' }) }}`

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:

attr

type: array **default:** []

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 25-16 `$builder->add('body', TextareaType::class, [
 'attr' => ['class' => 'tinymce'],
]);`

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 25-17 1 `use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 `[]` (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 25-18

```
$builder->add('name', null, [
    '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:** `[]`

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 25-19 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 25-20 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 25-21 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

```
Listing 25-22 1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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 25-23 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 25-24 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 25-25 1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 25-26

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

label_translation_parameters

type: array **default:** []

New in version 4.3: The **label_translation_parameters** option was introduced in Symfony 4.3.

The content of the label option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 25-27

```
1 # translations/messages.en.yaml
2 form.order.id: 'Identifiant de la commande de %company%'
```

You can specify the placeholder values as follows:

Listing 25-28

```
1 $builder->add('id', null, [
2     'label' => 'form.order.id',
3     'label_translation_parameters' => [
4         '%company%' => 'ACME Inc.',
5     ],
6 ]);
```

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

The `label_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

`attr_translation_parameters`

type: array **default:** []

New in version 4.3: The `attr_translation_parameters` option was introduced in Symfony 4.3.

The content of the `title` and `placeholder` values defined in the `attr` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 25-29 1 # translations/messages.en.yaml
2 form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3 form.order.id.title: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

```
Listing 25-30 1 $builder->add('id', null, [
2     'attr' => [
3         'placeholder' => 'form.order.id.placeholder',
4         'title' => 'form.order.id.title',
5     ],
6     'attr_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The `attr_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

`help_translation_parameters`

type: array **default:** []

New in version 4.3: The `help_translation_parameters` option was introduced in Symfony 4.3.

The content of the `help` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 25-31 1 # translations/messages.en.yaml
2 form.order.id.help: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

```
Listing 25-32 1 $builder->add('id', null, [
2     'help' => 'form.order.id.help',
3     'help_translation_parameters' => [
4         '%company%' => 'ACME Inc.',
5     ],
6 ]);
```

The `help_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.



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)
Options	<ul style="list-style-type: none">• <code>choice_translation_locale</code>
Overridden options	<ul style="list-style-type: none">• <code>choices</code>
Inherited options	<p>from the <i>ChoiceType</i></p> <ul style="list-style-type: none">• <code>error_bubbling</code>• <code>error_mapping</code>• <code>expanded</code>• <code>multiple</code>• <code>placeholder</code>• <code>preferred_choices</code>• <code>trim</code> <p>from the <i>FormType</i></p>

1. <https://secure.php.net/manual/en/locale.getdefault.php>

	<ul style="list-style-type: none"> • attr • data • disabled • empty_data • help • help_attr • help_html • label • label_attr • label_format • mapped • required • row_attr
Parent type	<i>ChoiceType</i>
Class	<i>CountryType</i> ²



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 26-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

choice_translation_locale

type: string or null **default:** null

This option determines if the choice values should be translated into a different locale than the current one.

The values of the **choice_translation_locale** option can be **null** (reuse the current translation locale) or a string which represents the exact translation locale to use.

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**.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/CountryType.php>

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:** []

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 26-2 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 26-3 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

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 26-4 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
5     'placeholder' => 'Choose an option',
6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 26-5 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
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 26-6 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, [
6     'required' => false,
7 ]);
```

preferred_choices

type: array, callable or string **default:** []

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 26-7 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork' => 'muppets',
9         'Pirate' => 'arr',
10    ],
```

```

11     'preferred_choices' => ['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 26-8

```

1  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2  // ...
3
4  $builder->add('publishAt', ChoiceType::class, [
5      'choices' => [
6          'now' => new \DateTime('now'),
7          'tomorrow' => new \DateTime('+1 day'),
8          '1 week' => new \DateTime('+1 week'),
9          '1 month' => new \DateTime('+1 month'),
10     ],
11     'preferred_choices' => function ($choice, $key, $value) {
12         // prefer options within 3 days
13         return $choice <= 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 26-9

```

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*:

attr

type: array **default:** []

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 26-10

```

$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);

```

Use the `row_attr` option if you want to add these attributes to the form type row element.

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 26-11

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 [] (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 26-12

```
$builder->add('name', null, [
    '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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 26-13

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 26-14

```
1 {{ form_help(form.name, 'Your name', {  
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 26-15

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 26-16

```
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 26-17

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type `row`:

Listing 26-18

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



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)
Options	<ul style="list-style-type: none">• <code>choice_translation_locale</code>
Overridden options	<ul style="list-style-type: none">• <code>choices</code>
Inherited options	<p>from the <i>ChoiceType</i></p> <ul style="list-style-type: none">• <code>error_bubbling</code>• <code>error_mapping</code>• <code>expanded</code>• <code>multiple</code>• <code>placeholder</code>• <code>preferred_choices</code>• <code>trim</code>

1. <http://site.icu-project.org>

2. <https://secure.php.net/manual/en/locale.getdefault.php>

	from the <i>FormType</i> <ul style="list-style-type: none"> • attr • data • disabled • empty_data • help • help_attr • help_html • label • label_attr • label_format • mapped • required • row_attr
Parent type	<i>ChoiceType</i>
Class	<i>LanguageType</i> ³



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 27-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

choice_translation_locale

type: string or null **default:** null

This option determines if the choice values should be translated into a different locale than the current one.

The values of the **choice_translation_locale** option can be **null** (reuse the current translation locale) or a string which represents the exact translation locale to use.

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**.

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/LanguageType.php>

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:** []

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 27-2 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 27-3 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

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 27-4 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
5     'placeholder' => 'Choose an option',
6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 27-5 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
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 27-6 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, [
6     'required' => false,
7 ]);
```

preferred_choices

type: array, callable or string **default:** []

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 27-7 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork' => 'muppets',
9         'Pirate' => 'arr',
10    ],
```

```

11     'preferred_choices' => ['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 27-8

```

1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('publishAt', ChoiceType::class, [
5     'choices' => [
6         'now' => new \DateTime('now'),
7         'tomorrow' => new \DateTime('+1 day'),
8         '1 week' => new \DateTime('+1 week'),
9         '1 month' => new \DateTime('+1 month'),
10    ],
11    'preferred_choices' => function ($choice, $key, $value) {
12        // prefer options within 3 days
13        return $choice <= 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 27-9

```

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*:

attr

type: array **default:** []

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 27-10

```

$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);

```


Use the `row_attr` option if you want to add these attributes to the form type row element.

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 27-11

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 [] (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 27-12

```
$builder->add('name', null, [
    '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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 27-13

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 27-14

```
1 {{ form_help(form.name, 'Your name', {  
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 27-15

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 27-16

```
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 27-17

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

4. <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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type `row`:

Listing 27-18

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



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)
Options	<ul style="list-style-type: none">• <code>choice_translation_locale</code>
Overridden options	<ul style="list-style-type: none">• <code>choices</code>
Inherited options	<p>from the <i>ChoiceType</i></p> <ul style="list-style-type: none">• <code>error_bubbling</code>• <code>error_mapping</code>• <code>expanded</code>• <code>multiple</code>• <code>placeholder</code>

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>

	<ul style="list-style-type: none"> • preferred_choices • trim <p>from the <i>FormType</i></p> <ul style="list-style-type: none"> • attr • data • disabled • empty_data • help • help_attr • help_html • label • label_attr • label_format • mapped • required • row_attr
Parent type	<i>ChoiceType</i>
Class	<i>LocaleType</i> ⁴



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 28-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

choice_translation_locale

type: string or null **default:** null

This option determines if the choice values should be translated into a different locale than the current one.

The values of the **choice_translation_locale** option can be **null** (reuse the current translation locale) or a string which represents the exact translation locale to use.

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.

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/LocaleType.php>



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:** `[]`

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 28-2 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 28-3 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

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 28-4 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
5     'placeholder' => 'Choose an option',
6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 28-5 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
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 28-6 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, [
6     'required' => false,
7 ]);
```

preferred_choices

type: array, callable or string **default:** []

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 28-7


```

1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork' => 'muppets',
9         'Pirate' => 'arr',
10    ],
11    'preferred_choices' => ['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:

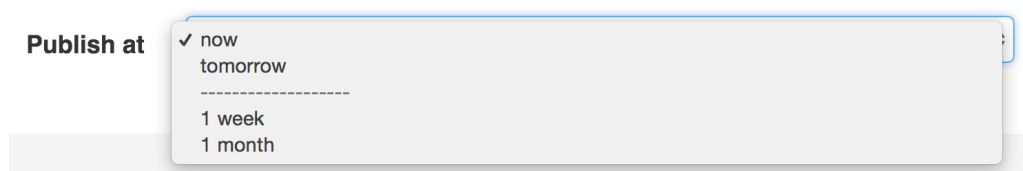
Listing 28-8

```

1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('publishAt', ChoiceType::class, [
5     'choices' => [
6         'now' => new \DateTime('now'),
7         'tomorrow' => new \DateTime('+1 day'),
8         '1 week' => new \DateTime('+1 week'),
9         '1 month' => new \DateTime('+1 month'),
10    ],
11    'preferred_choices' => function ($choice, $key, $value) {
12        // prefer options within 3 days
13        return $choice <= 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 28-9

```

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*:

attr

type: array **default:** []

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 28-10 $builder->add('body', TextareaType::class, [  
    'attr' => ['class' => 'tinymce'],  
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 28-11 1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;  
2 // ...  
3  
4 $builder->add('token', HiddenType::class, [  
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 overridden 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 [] (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 28-12 $builder->add('name', null, [  
    '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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 28-13

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 28-14

```
1 {{ form_help(form.name, 'Your name', {  
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 28-15

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 28-16

```
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 28-17

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*⁵ will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 28-18

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• input• intl• regions
Overridden options	<ul style="list-style-type: none">• choices
Inherited options	<p>from the <i>ChoiceType</i></p> <ul style="list-style-type: none">• choice_translation_domain• expanded• multiple• placeholder• preferred_choices• trim <p>from the <i>FormType</i></p> <ul style="list-style-type: none">• attr• data• disabled• empty_data

	<ul style="list-style-type: none"> • error_bubbling • error_mapping • help • help_attr • help_html • label • label_attr • label_format • mapped • required • row_attr
Parent type	<i>ChoiceType</i>
Class	<i>TimezoneType¹</i>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 29-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

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:

- `datetimezone` (a `\DateTimeZone` object)
- `intltimezone` (an `\IntlTimeZone` object)
- `string` (e.g. `America/New_York`)

New in version 4.3: The `intltimezone` input type was introduced in Symfony 4.3.

intl

type: boolean **default:** false

New in version 4.3: This option was introduced in Symfony 4.3.

If this option is set to **true**, the timezone selector will display the timezones from the *ICU Project*² via the *Intl component* instead of the regular PHP timezones.

Although both sets of timezones are pretty similar, only the ones from the Intl component can be translated to any language. To do so, set the desired locale with the `choice_translation_locale` option.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/TimezoneType.php>

2. <http://site.icu-project.org/>



The *Timezone constraint* can validate both timezone sets and adapts to the selected set automatically.

regions

type: `int` **default:** `\DateTimeZone::ALL`

Deprecated since version 4.2: This option was deprecated in Symfony 4.2.

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()`³, 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*:

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.

multiple

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

3. <https://secure.php.net/manual/en/datetimezone.listidentifiers.php>

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 29-2 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
5     'placeholder' => 'Choose an option',
6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 29-3 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('states', ChoiceType::class, [
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 29-4 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, [
6     'required' => false,
7 ]);
```

preferred_choices

type: array, callable or string **default:** []

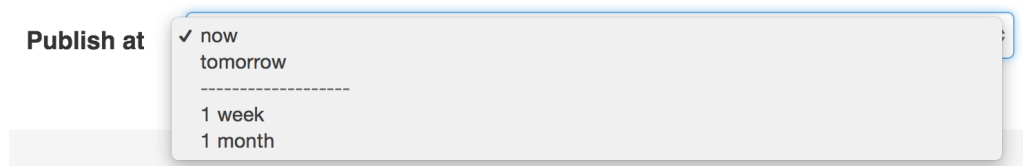
This option allows you to move certain choices to the top of your list with a visual separator between them and the rest of the options. If you have a form of languages, you can list the most popular on top, like Bork Bork and Pirate:

```
Listing 29-5 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork' => 'muppets',
9         'Pirate' => 'arr',
10    ],
11     'preferred_choices' => ['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 29-6 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('publishAt', ChoiceType::class, [
5     'choices' => [
6         'now' => new \DateTime('now'),
7         'tomorrow' => new \DateTime('+1 day'),
8         '1 week' => new \DateTime('+1 week'),
9         '1 month' => new \DateTime('+1 month'),
10    ],
11    'preferred_choices' => function ($choice, $key, $value) {
12        // prefer options within 3 days
13        return $choice <= 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 29-7 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*:

attr

type: array **default:** []

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 29-8 $builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 29-9

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 [] (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 29-10

```
$builder->add('name', null, [
    '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:** []

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 29-11 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 29-12 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```

Listing 29-13 $builder->add('zipCode', null, [
                'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
            ]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

```

Listing 29-14 1 {{ form_help(form.name, 'Your name', {
                2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
                3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 29-15 1 {{ form_label(form.name, 'Your name') }}

```

label_attr

type: array **default:** []

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 29-16 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 29-17 1 // ...
                2 $profileFormBuilder->add('address', AddressType::class, [

```

```

3     'label_format' => 'form.address.%name%',
4   });
5
6   $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 29-18

```

$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);

```

4. <http://diveintohtml5.info/forms.html>

Use the `attr` option if you want to add these attributes to the form type widget element.

New in version 4.3: The **`row_attr`** option was introduced in Symfony 4.3.



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 4217¹ 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)
Options	<ul style="list-style-type: none">• <code>choice_translation_locale</code>
Overridden options	<ul style="list-style-type: none">• <code>choices</code>
Inherited options	<p>from the <i>ChoiceType</i></p> <ul style="list-style-type: none">• <code>error_bubbling</code>• <code>expanded</code>• <code>multiple</code>• <code>placeholder</code>• <code>preferred_choices</code>• <code>trim</code> <p>from the <i>FormType</i> type</p> <ul style="list-style-type: none">• <code>attr</code>• <code>data</code>• <code>disabled</code>• <code>empty_data</code>• <code>help</code>• <code>help_attr</code>

1. https://en.wikipedia.org/wiki/ISO_4217

	<ul style="list-style-type: none"> • help_html • label • label_attr • label_format • mapped • required • row_attr
Parent type	<i>ChoiceType</i>
Class	<i>CurrencyType</i> ²



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 30-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

Field Options

choice_translation_locale

type: string or null **default:** null

This option determines if the choice values should be translated into a different locale than the current one.

The values of the **choice_translation_locale** option can be **null** (reuse the current translation locale) or a string which represents the exact translation locale to use.

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*:

² <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/CurrencyType.php>

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 30-2 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
              2 // ...
              3
              4 $builder->add('states', ChoiceType::class, [
              5     'placeholder' => 'Choose an option',
              6 ]);
```

- Guarantee that no "empty" value option is displayed:

```
Listing 30-3 1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
              2 // ...
              3
              4 $builder->add('states', ChoiceType::class, [
              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 30-4 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, [
              6     'required' => false,
              7 ]);
```

preferred_choices

type: array, callable or string **default:** []

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 30-5

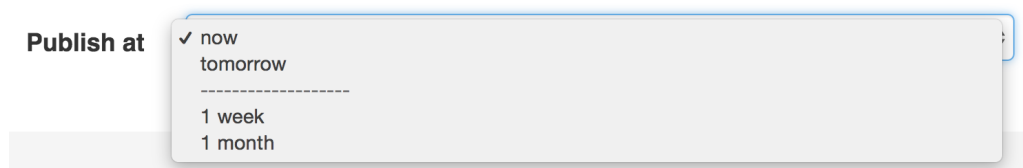
```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('language', ChoiceType::class, [
5     'choices' => [
6         'English' => 'en',
7         'Spanish' => 'es',
8         'Bork'    => 'muppets',
9         'Pirate' => 'arr',
10    ],
11    'preferred_choices' => ['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 30-6

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 // ...
3
4 $builder->add('publishAt', ChoiceType::class, [
5     'choices' => [
6         'now' => new \DateTime('now'),
7         'tomorrow' => new \DateTime('+1 day'),
8         '1 week' => new \DateTime('+1 week'),
9         '1 month' => new \DateTime('+1 month'),
10    ],
11    'preferred_choices' => function ($choice, $key, $value) {
12        // prefer options within 3 days
13        return $choice <= 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 30-7

```
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*:

attr

type: array **default:** []

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 30-8

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 30-9

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 [] (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 30-10

```
$builder->add('name', null, [
    '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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 30-11

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 30-12

```
1 {{ form_help(form.name, 'Your name', {
2   'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 30-13

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 30-14

```
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 30-15

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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 of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 30-16

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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



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 <code>DateTime</code> , string, timestamp, or array (see the <code>input</code> option)
Rendered as	single text box or three select fields
Options	<ul style="list-style-type: none">• <code>choice_translation_domain</code>• <code>days</code>• <code>placeholder</code>• <code>format</code>• <code>html5</code>• <code>input</code>• <code>input_format</code>• <code>model_timezone</code>• <code>months</code>• <code>view_timezone</code>• <code>widget</code>• <code>years</code>
Overridden options	<ul style="list-style-type: none">• <code>by_reference</code>• <code>compound</code>• <code>data_class</code>• <code>error_bubbling</code>
Inherited options	<ul style="list-style-type: none">• <code>attr</code>• <code>data</code>• <code>disabled</code>• <code>error_mapping</code>

	<ul style="list-style-type: none"> • help • help_attr • help_html • inherit_data • invalid_message • invalid_message_parameters • mapped • row_attr
Parent type	<i>FormType</i>
Class	<i>DateTime</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 31-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Basic Usage

This field type is highly configurable. 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 31-2

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2 // ...
3
4 $builder->add('publishedAt', DateTimeType::class, [
5     'widget' => 'choice',
6 ]);
```

If your underlying date is *not* a **DateTime** object (e.g. it's a unix timestamp or a **DateTimeImmutable** object), configure the input option:

Listing 31-3

```
$builder->add('publishedAt', DateTimeType::class, [
    'widget' => 'choice',
    'input'  => 'datetime_immutable'
]);
```

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 31-4

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2 // ...
3
4 $builder->add('publishedAt', DateTimeType::class, [
5     // renders it as a single text box
6     'widget' => 'single_text',
7 ]);
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/DateTimeType.php>

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 31-5

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2 // ...
3
4 $builder->add('publishedAt', DateTimeType::class, [
5     'widget' => 'single_text',
6
7     // prevents rendering it as type="date", to avoid HTML5 date pickers
8     'html5' => false,
9
10    // adds a class that can be selected in JavaScript
11    'attr' => ['class' => 'js-datepicker'],
12 ]);
```

Then, add the following JavaScript code in your template to initialize the date picker:

Listing 31-6

```
1 <script>
2     $(document).ready(function() {
3         // you may need to change this code if you are not using Bootstrap Datepicker
4         $('js-datepicker').datepicker({
5             format: 'yyyy-mm-dd'
6         });
7     });
8 </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

2. <https://github.com/eternicode/bootstrap-datepicker>

List of days available to the day field type. This option is only relevant when the **widget** option is set to **choice**:

Listing 31-7 `'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 31-8

```
$builder->add('dueDate', DateType::class, [
    'placeholder' => 'Select a value',
]);
```

Alternatively, you can use an array that configures different placeholder values for the year, month and day fields:

Listing 31-9

```
1 $builder->add('dueDate', DateType::class, [
2     'placeholder' => [
3         'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
4     ]
5 ]);
```

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 31-10

```
1 use Symfony\Component\Form\Extension\Core\Type\DateType;
2 // ...
3
4 $builder->add('dateCreated', DateType::class, [
5     'widget' => 'single_text',
6     // this is actually the default format for single_text
7     'format' => 'yyyy-MM-dd',
8 ]);
```



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 3339⁵ format) which is the default value if you use the `single_text` widget.

Deprecated since version 4.3: Using the **format** option when the **html5** option is enabled is deprecated since Symfony 4.3.

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>

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 datepicker, 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)
- datetime_immutable (a DateTimeImmutable object)
- array (e.g. ['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*⁶.

input_format

type: string **default:** Y-m-d

New in version 4.3: The **input_format** option was introduced in Symfony 4.3.

If the **input** option is set to **string**, this option specifies the format of the date. This must be a valid *PHP date format*⁷.

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*⁸.

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

6. <https://php.net/manual/en/function.date.php#refsect1-function.date-changelog>

7. <https://secure.php.net/manual/en/function.date.php>

8. <https://php.net/manual/en/timezones.php>

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*⁹.

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.

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*:

9. <https://php.net/manual/en/timezones.php>

attr

type: array **default:** []

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 31-11 $builder->add('body', TextareaType::class, [  
    'attr' => ['class' => 'tinymce'],  
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 31-12 1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;  
2 // ...  
3  
4 $builder->add('token', HiddenType::class, [  
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 overridden 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:** []

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 31-13 1 public function configureOptions(OptionsResolver $resolver)  
2 {  
3     $resolver->setDefaults([  
4         'error_mapping' => [  
5             'matchingCityAndZipCode' => 'city',  
6         ],  
7     ];
```

```

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 the `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 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 31-14

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.*' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 31-15

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 31-16

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 31-17 1 $builder->add('someField', SomeFormType::class, [  
2             // ...  
3             'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4             'invalid_message_parameters' => ['%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`.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

```
Listing 31-18 $builder->add('body', TextareaType::class, [  
             'row_attr' => ['class' => 'text-editor', 'id' => '...'],  
         ]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

Field Variables

Variable	Type	Usage
widget	mixed	The value of the widget option.
type	string	Only present when widget is <code>single_text</code> and HTML5 is activated, contains the input type to use (<code>datetime</code> , <code>date</code> or <code>time</code>).
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 8601¹ 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none"> • attr • data • disabled • help • help_attr • help_html • inherit_data • invalid_message • invalid_message_parameters • mapped • row_attr
Parent type	<i>FormType</i>
Class	<i>DateIntervalType</i> ²



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 32-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Basic Usage

This field type is highly configurable. 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 32-2

```
1 $builder->add('remindEvery', DateIntervalType::class, [
2     'widget'      => 'integer', // render a text field for each part
3     // 'input'    => 'string', // if you want the field to return a ISO 8601 string back to you
4
5     // customize which text boxes are shown
6     'with_years'  => false,
7     'with_months' => false,
8     'with_days'   => true,
9     'with_hours'  => true,
10 ]);
```

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 32-3

```
1 // values displayed to users range from 0 to 30 (both inclusive)
2 'days' => range(1, 31),
```

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/DateIntervalType.php>

```

3
4 // values displayed to users range from 1 to 31 (both inclusive)
5 'days' => array_combine(range(1, 31), 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 32-4

```

$builder->add('remindEvery', DateIntervalType::class, [
    'placeholder' => '',
]);

```

Alternatively, you can specify a string to be displayed for the "blank" value:

Listing 32-5

```

$builder->add('remindEvery', DateIntervalType::class, [
    'placeholder' => ['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 32-6

```

1 // values displayed to users range from 0 to 23 (both inclusive)
2 'hours' => range(1, 24),
3
4 // values displayed to users range from 1 to 24 (both inclusive)
5 'hours' => array_combine(range(1, 24), 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 8601³ standard, e.g. P7Y6M5DT12H15M30S)
- dateinterval (a DateInterval object)
- array (e.g. ['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 32-7

```

1 'labels' => [
2     'invert' => null,

```

3. https://en.wikipedia.org/wiki/ISO_8601

```

3     'years' => null,
4     'months' => null,
5     'weeks' => null,
6     'days' => null,
7     'hours' => null,
8     'minutes' => null,
9     'seconds' => null,
10 ]

```

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 32-8

```

1 // values displayed to users range from 0 to 59 (both inclusive)
2 'minutes' => range(1, 60),
3
4 // values displayed to users range from 1 to 60 (both inclusive)
5 'minutes' => array_combine(range(1, 60), 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**:

Listing 32-9

```

1 // values displayed to users range from 0 to 11 (both inclusive)
2 'months' => range(1, 12),
3
4 // values displayed to users range from 1 to 12 (both inclusive)
5 'months' => array_combine(range(1, 12), range(1, 12)),

```

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**:

Listing 32-10

```

1 // values displayed to users range from 0 to 59 (both inclusive)
2 'seconds' => range(1, 60),
3
4 // values displayed to users range from 1 to 60 (both inclusive)
5 'seconds' => array_combine(range(1, 60), range(1, 60)),

```

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**:

Listing 32-11

```

1 // values displayed to users range from 0 to 51 (both inclusive)
2 'weeks' => range(1, 52),

```

```

3
4 // values displayed to users range from 1 to 52 (both inclusive)
5 'weeks' => array_combine(range(1, 52), range(1, 52)),

```

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 32-12

```
1 // values displayed to users range from 0 to 99 (both inclusive)
2 'years' => range(1, 100),
3
4 // values displayed to users range from 1 to 100 (both inclusive)
5 'years' => array_combine(range(1, 100), range(1, 100)),
```

Inherited Options

These options inherit from the *form* type:

attr

type: array **default:** []

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 32-13

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 32-14

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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.

help

type: string default: null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 32-15

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array default: []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 32-16

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool default: false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 32-17

```
1 $builder->add('someField', SomeFormType::class, [  
2     // ...  
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4     'invalid_message_parameters' => ['%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`.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 32-18

```
$builder->add('body', TextareaType::class, [  
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],  
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

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 <code>DateTime</code> , string, timestamp, or array (see the <code>input</code> option)
Rendered as	single text box or three select fields
Options	<ul style="list-style-type: none">• <code>choice_translation_domain</code>• <code>date_format</code>• <code>date_label</code>• <code>date_widget</code>• <code>days</code>• <code>placeholder</code>• <code>format</code>• <code>hours</code>• <code>html5</code>• <code>input</code>• <code>input_format</code>• <code>minutes</code>• <code>model_timezone</code>• <code>months</code>• <code>seconds</code>• <code>time_label</code>• <code>time_widget</code>• <code>view_timezone</code>• <code>widget</code>• <code>with_minutes</code>• <code>with_seconds</code>• <code>years</code>

Overridden options	<ul style="list-style-type: none"> • <code>by_reference</code> • <code>compound</code> • <code>data_class</code> • <code>error_bubbling</code>
Inherited options	<ul style="list-style-type: none"> • <code>attr</code> • <code>data</code> • <code>disabled</code> • <code>help</code> • <code>help_attr</code> • <code>help_html</code> • <code>inherit_data</code> • <code>invalid_message</code> • <code>invalid_message_parameters</code> • <code>mapped</code> • <code>row_attr</code>
Parent type	<code>FormType</code>
Class	<code>DateTimeType</code> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 33-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

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.

Deprecated since version 4.3: Using the `date_format` option when the form is rendered as an HTML 5 datetime input is deprecated since Symfony 4.3.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/DateTimeType.php>

date_label

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

Sets the label that will be used when rendering the date widget. Setting it to **false** will suppress the label:

Listing 33-2

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2
3 $builder->add('startDateTime', DateTimeType::class, [
4     'date_label' => 'Starts On',
5 ]);
```

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.

Deprecated since version 4.3: Using the **date_widget** option when the **widget** option is set to **single_text** is deprecated since Symfony 4.3.

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 33-3

```
'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 33-4

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2
3 $builder->add('startDateTime', DateTimeType::class, [
4     'placeholder' => 'Select a value',
5 ]);
```

Alternatively, you can use an array that configures different placeholder values for the year, month, day, hour, minute and second fields:

Listing 33-5

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2
3 $builder->add('startDateTime', DateTimeType::class, [
4     'placeholder' => [
5         'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
6         'hour' => 'Hour', 'minute' => 'Minute', 'second' => 'Second',
7     ]
8 ]);
```

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 *datetime local*² format which is used by the HTML5 **datetime-local** field. Keeping the default value will cause the field to be rendered as an **input** field with **type="datetime-local"**. For more information on valid formats, see *Date/Time Format Syntax*³.

Deprecated since version 4.3: Using the **format** option when the **html5** option is enabled is deprecated since Symfony 4.3.

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 datepicker, 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 12:15:00)
- datetime (a DateTime object)
- datetime_immutable (a DateTimeImmutable object)
- array (e.g. [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, **DateTime** 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.

input_format

type: string **default:** Y-m-d H:i:s

2. <http://w3c.github.io/html-reference/datatypes.html#form.data.datetime-local>

3. <http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax>

4. <https://php.net/manual/en/function.date.php#refsect1-function.date-changelog>

New in version 4.3: The **input_format** option was introduced in Symfony 4.3.

If the **input** option is set to **string**, this option specifies the format of the date. This must be a valid *PHP date format*⁵.

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*⁶.

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_label

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

Sets the label that will be used when rendering the time widget. Setting it to **false** will suppress the label:

Listing 33-6

```
1 use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
2
3 $builder->add('startDateTime', DateTimeType::class, [
4     'time_label' => 'Starts On',
5 ]);
```

time_widget

type: string **default:** choice

Defines the **widget** option for the *TimeType*.

Deprecated since version 4.3: Using the **time_widget** option when the **widget** option is set to **single_text** is deprecated since Symfony 4.3.

5. <https://secure.php.net/manual/en/function.date.php>

6. <https://php.net/manual/en/timezones.php>

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:** 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

7. <https://php.net/manual/en/timezones.php>

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*:

attr

type: array **default:** []

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 33-7

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 33-8

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 33-9

```
$builder->add('zipCode', null, [  
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',  
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 33-10

```
1 {{ form_help(form.name, 'Your name', {  
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}  
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 33-11

```

1 $builder->add('someField', SomeFormType::class, [
2     // ...
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',
4     'invalid_message_parameters' => ['%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**.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 33-12

```

$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);

```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

Field Variables

Variable	Type	Usage
widget	mixed	The value of the widget option.
type	string	Only present when widget is <code>single_text</code> 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 <code>DateTime</code> , string, timestamp, or array (see the <code>input</code> option)
Rendered as	can be various tags (see below)
Options	<ul style="list-style-type: none">• <code>choice_translation_domain</code>• <code>placeholder</code>• <code>hours</code>• <code>html5</code>• <code>input</code>• <code>input_format</code>• <code>minutes</code>• <code>model_timezone</code>• <code>seconds</code>• <code>view_timezone</code>• <code>widget</code>• <code>with_minutes</code>• <code>with_seconds</code>
Overridden options	<ul style="list-style-type: none">• <code>by_reference</code>• <code>compound</code>• <code>data_class</code>• <code>error_bubbling</code>
Inherited options	<ul style="list-style-type: none">• <code>attr</code>• <code>data</code>• <code>disabled</code>

	<ul style="list-style-type: none"> • error_mapping • help • help_attr • help_html • inherit_data • invalid_message • invalid_message_parameters • mapped • row_attr
Parent type	FormType
Class	TimeType ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 34-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Basic Usage

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 34-2

```
1 use Symfony\Component\Form\Extension\Core\Type\TimeType;
2 // ...
3
4 $builder->add('startTime', TimeType::class, [
5     'input' => 'datetime',
6     'widget' => 'choice',
7 ]);
```

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 34-3

```
1 use Symfony\Component\Form\Extension\Core\Type\TimeType;
2 // ...
3
4 $builder->add('startTime', TimeType::class, [
5     'input' => 'timestamp',
6     'widget' => 'choice',
7 ]);
```

The field also supports an **array** and **string** as valid **input** option values.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/TimeType.php>

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

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 34-4

```
$builder->add('startTime', 'time', [  
    'placeholder' => 'Select a value',  
]);
```

Alternatively, you can use an array that configures different placeholder values for the hour, minute and second fields:

Listing 34-5

```
1 $builder->add('startTime', 'time', [  
2     'placeholder' => [  
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 datepicker, 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)
- datetime_immutable (a DateTimeImmutable object)

- array (e.g. ['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.

input_format

type: string **default:** H:i:s

New in version 4.3: The **input_format** option was introduced in Symfony 4.3.

If the **input** option is set to **string**, this option specifies the format of the time. This must be a valid *PHP time format*².

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*³.

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*⁴.

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).

2. <https://secure.php.net/manual/en/function.date.php>

3. <https://php.net/manual/en/timezones.php>

4. <https://php.net/manual/en/timezones.php>



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.

`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*:

`attr`

type: array **default:** []

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 34-6

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 34-7

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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: []

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 34-8

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 34-9

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 34-10

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 34-11

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 34-12 1 $builder->add('someField', SomeFormType::class, [  
2             // ...  
3             'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4             'invalid_message_parameters' => ['%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**.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

```
Listing 34-13 $builder->add('body', TextareaType::class, [  
1             'row_attr' => ['class' => 'text-editor', 'id' => '...'],  
2         ]);
```

*Use the **attr** option if you want to add these attributes to the the form type widget element.*

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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.

Variable	Type	Usage
type	string	Only present when widget is <code>single_text</code> and HTML5 is activated, contains the input type to use (<code>datetime</code> , <code>date</code> or <code>time</code>).



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 <i>DateTime</i> , <i>string</i> , <i>timestamp</i> , or <i>array</i> (see the input option)
Rendered as	can be three select boxes or 1 or 3 text boxes, based on the widget option
Overridden options	<ul style="list-style-type: none">• years
Inherited options	<p>from the <i>DateType</i>:</p> <ul style="list-style-type: none">• choice_translation_domain• days• placeholder• format• input• input_format• model_timezone• months• view_timezone• widget <p>from the <i>FormType</i>:</p> <ul style="list-style-type: none">• attr• data• disabled• help• help_attr• help_html

	<ul style="list-style-type: none"> • <code>inherit_data</code> • <code>invalid_message</code> • <code>invalid_message_parameters</code> • <code>mapped</code> • <code>row_attr</code>
Parent type	<code>DateType</code>
Class	<code>BirthdayType</code> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 35-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

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 35-2

```

'days' => range(1,31)

```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/BirthdayType.php>

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 35-3

```
$builder->add('birthdate', BirthdayType::class, [
    'placeholder' => 'Select a value',
]);
```

Alternatively, you can use an array that configures different placeholder values for the year, month and day fields:

Listing 35-4

```
1 $builder->add('birthdate', BirthdayType::class, [
2     'placeholder' => [
3         'year' => 'Year', 'month' => 'Month', 'day' => 'Day',
4     ]
5 ]);
```

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 35-5

```
1 use Symfony\Component\Form\Extension\Core\Type\DateType;
2 // ...
3
4 $builder->add('dateCreated', DateType::class, [
5     'widget' => 'single_text',
6     // this is actually the default format for single_text
7     'format' => 'yyyy-MM-dd',
8 ]);
```



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 3339⁴ 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)
- `datetime_immutable` (a `DateTimeImmutable` object)

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>

- array (e.g. ['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.

input_format

type: string **default:** Y-m-d

New in version 4.3: The **input_format** option was introduced in Symfony 4.3.

If the **input** option is set to **string**, this option specifies the format of the date. This must be a valid *PHP date format*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.

These options inherit from the *FormType*:

attr

type: array **default:** []

5. <https://php.net/manual/en/function.date.php#refsect1-function.date-changelog>

6. <https://secure.php.net/manual/en/function.date.php>

7. <https://php.net/manual/en/timezones.php>

8. <https://php.net/manual/en/timezones.php>

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 35-6

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 35-7

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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.

help

type: string default: null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 35-8

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array default: []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 35-9

```
1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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:** []

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 35-10

```
1 $builder->add('someField', SomeFormType::class, [  
2     // ...  
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4     'invalid_message_parameters' => ['%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**.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Listing 35-11

Use the `attr` option if you want to add these attributes to the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



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. Optionally you can specify an array of values that, if submitted, will be evaluated to "false" as well (this differs from what HTTP defines, but can be handy if you want to handle submitted values like "0" or "false").

Rendered as	input checkbox field
Options	<ul style="list-style-type: none">• false_values• value
Overridden options	<ul style="list-style-type: none">• compound• empty_data
Inherited options	<ul style="list-style-type: none">• attr• data• disabled• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr
Parent type	<i>FormType</i>
Class	<i>CheckboxType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 36-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Example Usage

Listing 36-2

```
1 use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
2 // ...
3
4 $builder->add('public', CheckboxType::class, [
5     'label' => 'Show this entry publicly?',
6     'required' => false,
7 ]);
```

Field Options

false_values

type: array **default:** [null]

An array of values to be interpreted as **false**.

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

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/CheckboxType.php>

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 overridden by the value returned by the data transformer (see *How to Use Data Transformers*).

Inherited Options

These options inherit from the *FormType*:

attr

type: array **default:** []

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 36-3

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 36-4

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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:** []

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 36-5 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 36-6 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.*' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 36-7 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 36-8

```

1 {{ form_help(form.name, 'Your name', {
2   'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 36-9

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 36-10

```

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 36-11

```

1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 36-12

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

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

Form Variables

Variable	Type	Usage
checked	boolean	Whether or not the current input is checked.



Chapter 37

FileType Field

The **FileType** represents a file input in your form.

Rendered as	input file field
Options	<ul style="list-style-type: none">• multiple
Overridden options	<ul style="list-style-type: none">• compound• data_class• empty_data
Inherited options	<ul style="list-style-type: none">• attr• disabled• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr
Parent type	<i>FormType</i>
Class	<i>FileType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/FileType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 37-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Basic Usage

Say you have this form definition:

Listing 37-2

```
use Symfony\Component\Form\Extension\Core\Type\FileType;
// ...

$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 37-3

```
1 use Symfony\Component\HttpFoundation\File\UploadedFile;
2
3 public function upload()
4 {
5     // ...
6
7     if ($form->isSubmitted() && $form->isValid()) {
8         $someNewFilename = ...
9
10        $file = $form['attachment']->getData();
11        $file->move($directory, $someNewFilename);
12
13        // ...
14    }
15
16    // ...
17 }
```

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 37-4

```
1 // use the original file name
2 $file->move($directory, $file->getClientOriginalName());
3
4 // compute a random name and try to guess the extension (more secure)
5 $extension = $file->guessExtension();
6 if (!$extension) {
7     // extension cannot be guessed
8     $extension = 'bin';
9 }
10 $file->move($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.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/File/UploadedFile.php>

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

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*:

attr

type: array **default:** []

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 37-5

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

disabled

type: boolean **default:** false

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/File/File.php>

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:** `[]`

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 37-6 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 37-7 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

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

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 37-8 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 37-9

```
1 {{ form_help(form.name, 'Your name', {
2   'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 37-10

```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 37-11

```
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 37-12

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
```

```

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 of 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.

row_attr

type: array default: []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 37-13

```

$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);

```

Use the `attr` option if you want to add these attributes to the the form type widget element.

4. <http://diveintohtml5.info/forms.html>

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

Form Variables

Variable	Type	Usage
type	string	The type variable is set to <code>file</code> , in order to render as a file input field.



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	<p>from the <i>CheckboxType</i>:</p> <ul style="list-style-type: none">• value <p>from the <i>FormType</i>:</p> <ul style="list-style-type: none">• attr• data• disabled• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr
Parent type	<i>CheckboxType</i>
Class	<i>RadioType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/RadioType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 38-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

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*:

attr

type: array **default:** []

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 38-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 38-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 overridden 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:** []

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 38-4 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 38-5 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ]);
```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

```
Listing 38-6 $builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

```
Listing 38-7 1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}
```

help_html

type: bool **default:** false

By default, the contents of the **help** option are escaped before rendering them in the template. Set this option to **true** to not escape them, which is useful when the help contains HTML elements.

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 38-8 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 38-9

```

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 38-10

```

1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3   'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

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

This is superficial and independent of 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.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 38-11

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

Form Variables

Variable	Type	Usage
checked	boolean	Whether or not the current input is checked.



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	<ul style="list-style-type: none">• allow_add• allow_delete• delete_empty• entry_options• entry_type• prototype• prototype_data• prototype_name
Inherited options	<ul style="list-style-type: none">• attr• by_reference• empty_data• error_bubbling• error_mapping• help• help_attr• help_html• label• label_attr• label_format• mapped• required• row_attr
Parent type	<i>FormType</i>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 39-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```



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 39-2

```
1 use Symfony\Component\Form\Extension\Core\Type\CollectionType;
2 use Symfony\Component\Form\Extension\Core\Type\EmailType;
3 // ...
4
5 $builder->add('emails', CollectionType::class, [
6     // each entry in the array will be an "email" field
7     'entry_type' => EmailType::class,
8     // these options are passed to each "email" type
9     'entry_options' => [
10         'attr' => ['class' => 'email-box'],
11     ],
12 ]);
```

The simplest way to render this is all at once:

Listing 39-3

```
1 {{ form_row(form.emails) }}
```

A much more flexible method would look like this:

Listing 39-4

```
1 {{ form_label(form.emails) }}
2 {{ form_errors(form.emails) }}
3
4 <ul>
5 {% for emailField in form.emails %}
6     <li>
7         {{ form_errors(emailField) }}
8         {{ form_widget(emailField) }}
9     </li>
10 {% endfor %}
11 </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.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/CollectionType.php>

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 39-5

```
1 <input type="email" id="form_emails_0" name="form[emails][0]" value="foo@foo.com"/>
2 <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 39-6

```
1 <input type="email"
2     id="form_emails__name__"
3     name="form[emails][__name__]"
4     value=""
5 />
```

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 39-7

```
1 // add-collection-widget.js
2 jQuery(document).ready(function () {
3     jQuery('.add-another-collection-widget').click(function (e) {
4         var list = jQuery(jQuery(this).attr('data-list-selector'));
5         // Try to find the counter of the list or use the length of the list
6         var counter = list.data('widget-counter') || list.children().length;
7
8         // grab the prototype template
9         var newWidget = list.attr('data-prototype');
10        // replace the "__name__" used in the id and name of the prototype
11        // with a number that's unique to your emails
12        // end name attribute looks like name="contact[emails][2]"
13        newWidget = newWidget.replace(/__name__/g, counter);
14        // Increase the counter
15        counter++;
16        // And store it, the length cannot be used if deleting widgets is allowed
17        list.data('widget-counter', counter);
18
19        // create a new list element and add it to the list
20        var newElem = jQuery(list.attr('data-widget-tags')).html(newWidget);
21        newElem.appendTo(list);
22    });
23 });
```

And update the template as follows:

Listing 39-8

```
1 {{ form_start(form) }}
2     {# ... #}
3
4     {# store the prototype on the data-prototype attribute #}
```

```

5     <ul id="email-fields-list"
6         data-prototype="{{ form_widget(form.emails.vars.prototype)|e }}"
7         data-widget-tags="{{ ' <li></li>'|e }}"
8         data-widget-counter="{{ form.emails|length }}"
9     {% for emailField in form.emails %}
10         <li>
11             {{ form_errors(emailField) }}
12             {{ form_widget(emailField) }}
13         </li>
14     {% endfor %}
15 </ul>
16
17 <button type="button"
18     class="add-another-collection-widget"
19     data-list-selector="#email-fields-list">Add another email</button>
20
21 {# ... #}
22 {{ form_end(form) }}
23
24 <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 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 39-9

```
1 use Symfony\Component\Form\Extension\Core\Type\CollectionType;
2 // ...
3
4 $builder->add('users', CollectionType::class, [
5     // ...
6     'delete_empty' => function (User $user = null) {
7         return null === $user || empty($user->getFirstName());
8     },
9 ]);
```

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:** []

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 39-10

```
1 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
2 use Symfony\Component\Form\Extension\Core\Type\CollectionType;
3 // ...
4
5 $builder->add('favoriteCities', CollectionType::class, [
6     'entry_type' => ChoiceType::class,
7     'entry_options' => [
8         'choices' => [
9             'Nashville' => 'nashville',
10            'Paris' => 'paris',
11            'Berlin' => 'berlin',
12            'London' => 'london',
13        ],
14    ],
15 ]);
```

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 39-11 1 `{{ 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 39-12 1 `use Symfony\Component\Form\Extension\Core\Type\CollectionType;`
2 `use Symfony\Component\Form\Extension\Core\Type\TextType;`
3 `// ...`
4
5 `$builder->add('tags', CollectionType::class, [`
6 `'entry_type' => TextType::class,`
7 `'allow_add' => true,`
8 `'prototype' => true,`
9 `'prototype_data' => 'New Tag Placeholder',`
10 `]);`

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:

attr

type: array **default:** []

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 39-13

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 39-14

```
1 use Symfony\Component\Form\Extension\Core\Type\EmailType;
2 use Symfony\Component\Form\Extension\Core\Type\FormType;
3 use Symfony\Component\Form\Extension\Core\Type\TextType;
4 // ...
5
6 $builder = $this->createFormBuilder($article);
7 $builder
8     ->add('title', TextType::class)
9     ->add(
10         $builder->create('author', FormType::class, ['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 39-15

```
$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 39-16

```
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.

empty_data

type: mixed

The default value is `[]` (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 39-17

```
$builder->add('name', null, [
    '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:** `[]`

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 39-18

```

1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 39-19

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '. => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 39-20

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 39-21

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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 39-22 1 {{ form_label(form.name, 'Your name') }}
```

label_attr

type: array **default:** []

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 39-23 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 39-24 1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*² will be rendered. The corresponding **label** will also render with a **required** class.

This is superficial and independent of 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.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 39-25

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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	<ul style="list-style-type: none">• first_name• first_options• options• second_name• second_options• type
Overridden options	<ul style="list-style-type: none">• error_bubbling
Inherited options	<ul style="list-style-type: none">• attr• data• error_mapping• help• help_attr• help_html• invalid_message• invalid_message_parameters• mapped• row_attr
Parent type	<i>FormType</i>
Class	<i>RepeatedType</i> ¹

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/RepeatedType.php>



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 40-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Example Usage

Listing 40-2

```
1 use Symfony\Component\Form\Extension\Core\Type\PasswordType;
2 use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
3 // ...
4
5 $builder->add('password', RepeatedType::class, [
6     'type' => PasswordType::class,
7     'invalid_message' => 'The password fields must match.',
8     'options' => ['attr' => ['class' => 'password-field']],
9     'required' => true,
10    'first_options' => ['label' => 'Password'],
11    'second_options' => ['label' => 'Repeat Password'],
12 ]);
```

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 40-3

```
1 {{ form_row(form.password) }}
```

To render each field individually, use something like this:

Listing 40-4

```
1 {# .first and .second may vary in your use - see the note below #}
2 {{ form_row(form.password.first) }}
3 {{ 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:** `[]`

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 40-5 1 use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
2 // ...
3
4 $builder->add('password', RepeatedType::class, [
5     'first_options' => ['label' => 'Password'],
6     'second_options' => ['label' => 'Repeat Password'],
7 ]);
```

`options`

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

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:** `[]`

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*:

`attr`

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

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 40-6

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

`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 40-7

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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:** `[]`

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 40-8

```

1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 40-9

```

1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.*' => 'city',
4     ],
5 ]);

```

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 40-10

```

$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);

```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 40-11

```

1 {{ form_help(form.name, 'Your name', {
2     'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3 }) }}

```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

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:** []

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 40-12 1 $builder->add('someField', SomeFormType::class, [  
2             // ...  
3             'invalid_message' => 'You entered an invalid value, it should include %num% letters',  
4             'invalid_message_parameters' => ['%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**.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

```
Listing 40-13 $builder->add('body', TextareaType::class, [  
             'row_attr' => ['class' => 'text-editor', 'id' => '...'],  
         ]);
```

*Use the **attr** option if you want to add these attributes to the the form type widget element.*

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.



Chapter 41

HiddenType Field

The hidden type represents a hidden input field.

Rendered as	input hidden field
Overridden options	<ul style="list-style-type: none">• compound• error_bubbling• required
Inherited options	<ul style="list-style-type: none">• attr• data• error_mapping• mapped• property_path• row_attr
Parent type	<i>FormType</i>
Class	<i>HiddenType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 41-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/HiddenType.php>

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

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*:

attr

type: array **default:** []

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 41-2

```
$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);
```

Use the `row_attr` option if you want to add these attributes to the the form type row element.

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 41-3

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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:** []

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 41-4 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the `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 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 41-5 1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
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**.

property_path

type: `PropertyPathInterface|string|null` **default:** null

By default (when the value of this option is `null`) form fields read from and write to the properties with the same names in the form's domain object. The `property_path` option lets you define which property a field reads from and writes to. The value of this option can be any *valid PropertyAccess syntax*.

`row_attr`

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 41-6

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



Chapter 42

ButtonType Field

A simple, non-responsive button.

Rendered as	button tag
Inherited options	<ul style="list-style-type: none">• attr• attr_translation_parameters• disabled• label• label_translation_parameters• row_attr• translation_domain
Parent type	none
Class	<i>ButtonType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 42-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

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).

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/ButtonType.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php>

attr

type: array **default:** []

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 42-2 1 use Symfony\Component\Form\Extension\Core\Type\ButtonType;
2 // ...
3
4 $builder->add('save', ButtonType::class, [
5     'attr' => ['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 42-3 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.

label_translation_parameters

type: array **default:** []

New in version 4.3: The **label_translation_parameters** option was introduced in Symfony 4.3.

The content of the label option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 42-4 1 # translations/messages.en.yaml
2 form.order.submit_to_company: 'Send an order to %company%'
```

You can specify the placeholder values as follows:

```
Listing 42-5 1 use Symfony\Component\Form\Extension\Core\Type\ButtonType;
2 // ...
3
4 $builder->add('send', ButtonType::class, [
5     'label' => 'form.order.submit_to_company',
6     'label_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
```

```

8     ],
9  });

```

The `label_translation_parameters` option of buttons is merged with the same option of its parents, so buttons can reuse and/or override any of the parent placeholders.

attr_translation_parameters

type: array **default:** []

New in version 4.3: The `attr_translation_parameters` option was introduced in Symfony 4.3.

The content of the `title` and `placeholder` values defined in the `attr` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 42-6

```

1  # translations/messages.en.yaml
2  form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3  form.order.id.title: 'This will be the reference in communications with %company%'

```

You can specify the placeholder values as follows:

Listing 42-7

```

1  $builder->add('id', null, [
2      'attr' => [
3          'placeholder' => 'form.order.id.placeholder',
4          'title' => 'form.order.id.title',
5      ],
6      'attr_translation_parameters' => [
7          '%company%' => 'ACME Inc.',
8      ],
9  ]);

```

The `attr_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 42-8

```

$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);

```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



Chapter 43

ResetType Field

A button that resets all fields to their original values.

Rendered as	input reset tag
Inherited options	<ul style="list-style-type: none">• attr• attr_translation_parameters• disabled• label• label_translation_parameters• row_attr• translation_domain
Parent type	<i>ButtonType</i>
Class	<i>ResetType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 43-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

Inherited Options

attr

type: array **default:** []

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/ResetType.php>

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 43-2 1 use Symfony\Component\Form\Extension\Core\Type\ResetType;
2 // ...
3
4 $builder->add('save', ResetType::class, [
5     'attr' => ['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 43-3 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.

label_translation_parameters

type: array **default:** []

The content of the label option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 43-4 1 # translations/messages.en.yaml
2 form.order.reset: 'Reset an order to %company%'
```

You can specify the placeholder values as follows:

```
Listing 43-5 1 use Symfony\Component\Form\Extension\Core\Type\ResetType;
2 // ...
3
4 $builder->add('send', ResetType::class, [
5     'label' => 'form.order.reset',
6     'label_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The **label_translation_parameters** option of buttons is merged with the same option of its parents, so buttons can reuse and/or override any of the parent placeholders.

attr_translation_parameters

type: array **default:** []

New in version 4.3: The `attr_translation_parameters` option was introduced in Symfony 4.3.

The content of the `title` and `placeholder` values defined in the `attr` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 43-6

```
1 # translations/messages.en.yaml
2 form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3 form.order.id.title: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

Listing 43-7

```
1 $builder->add('id', null, [
2     'attr' => [
3         'placeholder' => 'form.order.id.placeholder',
4         'title' => 'form.order.id.title',
5     ],
6     'attr_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The `attr_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 43-8

```
$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);
```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.



Chapter 44

SubmitType Field

A submit button.

Rendered as	button submit tag
Inherited options	<ul style="list-style-type: none">• attr• attr_translation_parameters• disabled• label• label_format• label_translation_parameters• row_attr• translation_domain• validation_groups
Parent type	<i>ButtonType</i>
Class	<i>SubmitType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 44-1

```
1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType
```

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 44-2

```
if ($form->get('save')->isClicked()) {
    // ...
}
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/SubmitType.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/ClickableInterface.php>

Inherited Options

attr

type: array **default:** []

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 44-3 1 use Symfony\Component\Form\Extension\Core\Type\SubmitType;
2 // ...
3
4 $builder->add('save', SubmitType::class, [
5     'attr' => ['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 44-4 1 {{ form_widget(form.save, { 'label': 'Click me' }) }}
```

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 1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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*.

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.

label_translation_parameters

type: array **default:** []

The content of the label option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 44-6

```
1 # translations/messages.en.yaml
2 form.order.submit_to_company: 'Send an order to %company%'
```

You can specify the placeholder values as follows:

Listing 44-7

```
1 use Symfony\Component\Form\Extension\Core\Type\SubmitType;
2 // ...
3
4 $builder->add('send', SubmitType::class, [
5     'label' => 'form.order.submit_to_company',
6     'label_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The `label_translation_parameters` option of buttons is merged with the same option of its parents, so buttons can reuse and/or override any of the parent placeholders.

attr_translation_parameters

type: array **default:** []

New in version 4.3: The `attr_translation_parameters` option was introduced in Symfony 4.3.

The content of the `title` and `placeholder` values defined in the `attr` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 44-8

```

1 # translations/messages.en.yaml
2 form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3 form.order.id.title: 'This will be the reference in communications with %company%'

```

You can specify the placeholder values as follows:

Listing 44-9

```

1 $builder->add('id', null, [
2     'attr' => [
3         'placeholder' => 'form.order.id.placeholder',
4         'title' => 'form.order.id.title',
5     ],
6     'attr_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);

```

The `attr_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 44-10

```

$builder->add('body', TextareaType::class, [
    'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);

```

Use the `attr` option if you want to add these attributes to the the form type widget element.

New in version 4.3: The `row_attr` option was introduced in Symfony 4.3.

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 44-11

```

1 use Symfony\Component\Form\Extension\Core\Type\SubmitType;
2 // ...
3
4 $form = $this->createFormBuilder($user)
5     ->add('previousStep', SubmitType::class, [
6         'validation_groups' => false,
7     ])
8     ->add('nextStep', SubmitType::class, [
9         'validation_groups' => ['Registration'],
10    ])
11    ->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	<ul style="list-style-type: none">• action• allow_extra_fields• by_reference• compound• constraints• data• data_class• empty_data• error_bubbling• error_mapping• extra_fields_message• help• help_attr• help_html• help_translation_parameters• inherit_data• invalid_message• invalid_message_parameters• label_attr• label_format• mapped• method• post_max_size_message• property_path• required• trim• validation_groups
---------	---

Inherited options	<ul style="list-style-type: none"> • attr • auto_initialize • block_name • block_prefix • disabled • label • row_attr • translation_domain • label_translation_parameters • attr_translation_parameters
Parent	none
Class	<i>FormType</i> ¹



The full list of options defined and inherited by this form type is available running this command in your app:

Listing 45-1

```

1 # replace 'FooType' by the class name of your form type
2 $ php bin/console debug:form FooType

```

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 45-2

```

1 use Symfony\Component\Form\Extension\Core\Type\EmailType;
2 use Symfony\Component\Form\Extension\Core\Type\FormType;

```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/FormType.php>

```

3 use Symfony\Component\Form\Extension\Core\Type\TextType;
4 // ...
5
6 $builder = $this->createFormBuilder($article);
7 $builder
8     ->add('title', TextType::class)
9     ->add(
10         $builder->create('author', FormType::class, ['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 45-3

```

$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 45-4

```

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.

compound

type: boolean **default:** true

A compound form can be either an entire `<form>` element or a group of form fields (rendered for example inside a `<div>` or `<tr>` container elements). Compound forms use the *DataMapperInterface* to initialize their children or to write back their submitted data.

A simple (non-compound) form is rendered as any of these HTML elements: `<input>` (*TextType*, *FileType*, *HiddenType*), `<textarea>` (*TextareaType*) or `<select>` (*ChoiceType*).

Some core types like date related types or the *ChoiceType* are simple or compound depending on other options (such as `expanded` or `widget`). They will either behave as a simple text field or as a group of text or choice fields.

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.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraint.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Validator/Type/FormTypeValidatorExtension.php>

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 45-5

```
1 use Symfony\Component\Form\Extension\Core\Type\HiddenType;
2 // ...
3
4 $builder->add('token', HiddenType::class, [
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 overridden 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 45-6

```
1 use App\Entity\Media;
2 use App\Form\MediaType;
3 // ...
4
5 $builder->add('media', MediaType::class, [
6     'data_class' => Media::class,
7 ]);
```

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 `[]` (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 45-7

```
$builder->add('name', null, [
    '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:** []

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 45-8

```
1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'error_mapping' => [
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 the **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 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 45-9

```
1 $resolver->setDefaults([
2     'error_mapping' => [
3         '.' => 'city',
4     ],
5 ];
```

```
4     ],
5 });
```

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.

help

type: string **default:** null

Allows you to define a help message for the form field, which by default is rendered below the field:

Listing 45-10

```
$builder->add('zipCode', null, [
    'help' => 'The ZIP/Postal code for your credit card\'s billing address.',
]);
```

help_attr

type: array **default:** []

Sets the HTML attributes for the element used to display the help message of the form field. Its value is an associative array with HTML attribute names as keys. These attributes can also be set in the template:

Listing 45-11

```
1  {{ form_help(form.name, 'Your name', {
2    'help_attr': {'class': 'CUSTOM_LABEL_CLASS'}
3  }) }}
```

help_html

type: bool **default:** false

By default, the contents of the `help` option are escaped before rendering them in the template. Set this option to `true` to not escape them, which is useful when the help contains HTML elements.

help_translation_parameters

type: array **default:** []

New in version 4.3: The `help_translation_parameters` option was introduced in Symfony 4.3.

The content of the `help` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

Listing 45-12

```
1  # translations/messages.en.yaml
2  form.order.id.help: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

Listing 45-13

```
1  $builder->add('id', null, [
2    'help' => 'form.order.id.help',
3    'help_translation_parameters' => [
```

```

4         '%company%' => 'ACME Inc.',
5     ],
6 );

```

The `help_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

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:** []

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 45-14

```

1 $builder->add('someField', SomeFormType::class, [
2     // ...
3     'invalid_message' => 'You entered an invalid value, it should include %num% letters',
4     'invalid_message_parameters' => ['%num%' => 6],
5 ]);

```

label_attr

type: array **default:** []

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 45-15

```

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 45-16

```
1 // ...
2 $profileFormBuilder->add('address', AddressType::class, [
3     'label_format' => 'form.address.%name%',
4 ]);
5
6 $invoiceFormBuilder->add('invoice', AddressType::class, [
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**.

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 matching routes by HTTP method.



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

By default (when the value of this option is `null`) form fields read from and write to the properties with the same names in the form's domain object. The `property_path` option lets you define which property a field reads from and writes to. The value of this option can be any *valid PropertyAccess syntax*.

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 of 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.

4. <http://diveintohtml5.info/forms.html>

5. <https://secure.php.net/manual/en/function.trim.php>

validation_groups

type: array, string, callable, *GroupSequence*⁶ or null **default:** null

This option is only valid on the root form and is used to specify which groups will be used by the validator.

For **null** the validator will just use the **Default** group.

If you specify the groups as an array or string they will be used by the validator as they are:

```
Listing 45-17 1 public function configureOptions(OptionsResolver $resolver)
2 {
3     $resolver->setDefaults([
4         'validation_groups' => 'Registration',
5     ]);
6 }
```

This is equivalent to passing the group as array:

```
Listing 45-18 'validation_groups' => ['Registration'],
```

The form's data will be *validated against all given groups*.

If the validation groups depend on the form's data a callable may be passed to the option. Symfony will then pass the form when calling it:

```
Listing 45-19 1 use Symfony\Component\Form\FormInterface;
2 use Symfony\Component\OptionsResolver\OptionsResolver;
3
4 // ...
5 public function configureOptions(OptionsResolver $resolver)
6 {
7     $resolver->setDefaults([
8         'validation_groups' => function (FormInterface $form) {
9             $entity = $form->getData();
10
11             return $entity->isUser() ? ['User'] : ['Company'];
12         },
13     ]);
14 }
```

You can read more about this in [How to Choose Validation Groups Based on the Submitted Data](#).



When your form contains multiple submit buttons, you can change the validation group depending on *which button is used* to submit the form.

If you need advanced logic to determine the validation groups have a look at *[How to Dynamically Configure Form Validation Groups](#)*.

In some cases, you want to validate your groups step by step. To do this, you can pass a *GroupSequence*⁷ to this option. This enables you to validate against multiple groups, like when you pass multiple groups in an array, but with the difference that a group is only validated if the previous groups pass without errors. Here's an example:

```
Listing 45-20 1 use Symfony\Component\Form\AbstractType;
2 use Symfony\Component\Validator\Constraints\GroupSequence;
3 // ...
4
5 class MyType extends AbstractType
6 {
```

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GroupSequence.php>

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GroupSequence.php>

```

7      // ...
8      public function configureOptions(OptionsResolver $resolver)
9      {
10         $resolver->setDefaults([
11             'validation_groups' => new GroupSequence(['First', 'Second']),
12         ]);
13     }
14 }

```

Read the article *How to Sequentially Apply Validation Groups* to find out more about this.

Inherited Options

The following options are defined in the *BaseType*⁸ 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).

attr

type: array **default:** []

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 45-21

```

$builder->add('body', TextareaType::class, [
    'attr' => ['class' => 'tinymce'],
]);

```

Use the *row_attr* option if you want to add these attributes to the the form type row element.

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 add a custom block name to the ones used by default 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.

If you set for example this option to **my_custom_name** and the field is of type **text**, Symfony will use the following names (and in this order) to find the block used to render the widget of the field: **_my_custom_name_widget**, **text_widget** and **form_widget**.

8. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php>

block_prefix

type: string or null **default:** null (see Knowing which block to customize)

New in version 4.3: The **block_prefix** option was introduced in Symfony 4.3.

Allows you to add a custom block prefix and 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 all of them without the need to create a new form type.

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 45-22 1 `{{ form_label(form.name, 'Your name') }}`

row_attr

type: array **default:** []

An associative array of the HTML attributes added to the element which is used to render the form type row:

Listing 45-23 `$builder->add('body', TextareaType::class, [
 'row_attr' => ['class' => 'text-editor', 'id' => '...'],
]);`

Use the attr option if you want to add these attributes to the the form type widget element.

New in version 4.3: The **row_attr** option was introduced in Symfony 4.3.

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.

label_translation_parameters

type: array **default:** []

New in version 4.3: The **label_translation_parameters** option was introduced in Symfony 4.3.

The content of the label option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 45-24 1 # translations/messages.en.yaml
2 form.order.id: 'Identifier of the order to %company%'
```

You can specify the placeholder values as follows:

```
Listing 45-25 1 $builder->add('id', null, [
2     'label' => 'form.order.id',
3     'label_translation_parameters' => [
4         '%company%' => 'ACME Inc.',
5     ],
6 ]);
```

The `label_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.

`attr_translation_parameters`

type: array **default:** []

New in version 4.3: The `attr_translation_parameters` option was introduced in Symfony 4.3.

The content of the `title` and `placeholder` values defined in the `attr` option is translated before displaying it, so it can contain translation placeholders. This option defines the values used to replace those placeholders.

Given this translation message:

```
Listing 45-26 1 # translations/messages.en.yaml
2 form.order.id.placeholder: 'Enter unique identifier of the order to %company%'
3 form.order.id.title: 'This will be the reference in communications with %company%'
```

You can specify the placeholder values as follows:

```
Listing 45-27 1 $builder->add('id', null, [
2     'attr' => [
3         'placeholder' => 'form.order.id.placeholder',
4         'title' => 'form.order.id.title',
5     ],
6     'attr_translation_parameters' => [
7         '%company%' => 'ACME Inc.',
8     ],
9 ]);
```

The `attr_translation_parameters` option of children fields is merged with the same option of their parents, so children can reuse and/or override any of the parent placeholders.



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*
- *Json*
- *Uuid*
- *UserPassword*
- *NotCompromisedPassword*

Comparison Constraints

- *EqualTo*
- *NotEqualTo*
- *IdenticalTo*
- *NotIdenticalTo*
- *LessThan*
- *LessThanOrEqual*
- *GreaterThan*
- *GreaterThanOrEqual*
- *Range*
- *DivisibleBy*
- *Unique*

Number Constraints

- *Positive*
- *PositiveOrZero*
- *Negative*
- *NegativeOrZero*

Date Constraints

- *Date*
- *DateTime*
- *Time*
- *Timezone*

Choice Constraints

- *Choice*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Financial and other Number Constraints

- *Bic*
- *CardScheme*
- *Currency*
- *Luhn*
- *Iban*
- *Isbn*
- *Issn*

Other Constraints

- *Callback*

- *Expression*
- *All*
- *Valid*
- *Traverse*
- *Collection*
- *Count*
- *UniqueEntity*



Chapter 47

NotBlank

Validates that a value is not blank - meaning not equal to a blank string, a blank array, **false** or **null** (null behavior is configurable). To check that a value is not equal to **null**, see the *NotNull* constraint.

Applies to	property or method
Options	<ul style="list-style-type: none">• allowNull• groups• message• normalizer• payload
Class	<i>NotBlank</i> ¹
Validator	<i>NotBlankValidator</i> 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 47-1

```
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\NotBlank
10      */
11     protected $firstName;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotBlank.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotBlankValidator.php>

Options

allowNull

type: bool **default:** false

If set to **true**, **null** values are considered valid and won't trigger a constraint violation.

New in version 4.3: The **allowNull** option was introduced in Symfony 4.3.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should not be blank.

This is the message that will be shown if the value is blank.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

normalizer

type: a PHP callable³ **default:** null

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the **'trim'** string to apply the *trim*⁴ PHP function in order to ignore leading and trailing whitespace during validation.

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://www.php.net/callable>

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



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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Blank</i> ¹
Validator	<i>BlankValidator</i> 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

```
1 // src/Entity/Author.php  
2 namespace App\Entity;  
3  
4 use Symfony\Component\Validator\Constraints as Assert;  
5  
6 class Author
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Blank.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/BlankValidator.php>

```

7 {
8     /**
9     * @Assert\Blank
10    */
11    protected $firstName;
12 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be blank.

This is the message that will be shown if the value is not blank.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 49

NotNull

Validates that a value is not strictly equal to `null`. To ensure that a value is not blank (not a blank string), see the *NotBlank* constraint.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>NotNull</i> ¹
Validator	<i>NotNullValidator</i> 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

```
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\NotNull
10     */
11     protected $firstName;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotNull.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotNullValidator.php>

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should not be null.

This is the message that will be shown if the value is **null**.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 50

IsNull

Validates that a value is exactly equal to **null**. To force that a property is 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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>IsNull</i> ¹
Validator	<i>IsNullValidator</i> 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

```
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\IsNull
10      */
11     protected $firstName;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsNull.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsNullValidator.php>

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be null.

This is the message that will be shown if the value is not `null`.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>IsTrue</i> ¹
Validator	<i>IsTrueValidator</i> 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

```
1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  class Author
5  {
6      protected $token;
7
8      public function isTokenValid()
9      {
10         return $this->token == $this->generateToken();
11     }
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsTrue.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsTrueValidator.php>


```

11     }
12 }

```

Then you can constrain this method with **IsTrue**.

Listing 51-2

```

1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      protected $token;
9
10     /**
11      * @Assert\IsTrue(message="The token is invalid.")
12      */
13     public function isTokenValid()
14     {
15         return $this->token == $this->generateToken();
16     }
17 }

```

If the `isTokenValid()` returns false, the validation will fail.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be true.

This message is shown if the underlying data is not true.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>IsFalse</i> ¹
Validator	<i>IsFalseValidator</i> 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

```
1 protected $state;
2
3 protected $invalidStates = [];
4
5 public function isStateInvalid()
6 {
7     return in_array($this->state, $this->invalidStates);
8 }
```

In this case, the underlying object is only valid if the **isStateInvalid()** method returns **false**:

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsFalse.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsFalseValidator.php>

Listing 52-2

```
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\IsFalse(
10        *     message = "You've entered an invalid state."
11        * )
12        */
13        public function isStateInvalid()
14        {
15            // ...
16        }
17    }
```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be false.

This message is shown if the underlying data is not false.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 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	<ul style="list-style-type: none">• groups• message• payload• type
Class	<i>Type</i> ¹
Validator	<i>TypeValidator</i> ²

Basic Usage

This will check that **id** is an instance of `Ramsey\Uuid\UuidInterface`, **firstName** is of type **string** and **age** is an **integer**.

Listing 53-1

```
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\Type("Ramsey\Uuid\UuidInterface")
10     */
11     protected $id;
12
13     /**
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Type.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/TypeValidator.php>

```

14     * @Assert\Type("string")
15     */
16     protected $firstName;
17
18     /**
19     * @Assert\Type(
20     *     type="integer",
21     *     message="The value {{ value }} is not a valid {{ type }}."
22     * )
23     */
24     protected $age;
25 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be of type {{ type }}.

The message if the underlying data is not of the given type.

You can use the following parameters in this message:

Parameter	Description
{{ type }}	The expected type
{{ value }}	The current (invalid) 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.

type

type: string [default option]

This required option is either the FQCN (fully qualified class name) of some PHP class/interface or a valid PHP datatype (checked by PHP's `is_()` functions):

- `array`³
- `bool`⁴
- `callable`⁵

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>

- *float*⁶
- *double*⁷
- *int*⁸
- *integer*⁹
- *iterable*¹⁰
- *long*¹¹
- *null*¹²
- *numeric*¹³
- *object*¹⁴
- *real*¹⁵
- *resource*¹⁶
- *scalar*¹⁷
- *string*¹⁸

Also, you can use `ctype_()` functions from corresponding *built-in PHP extension*¹⁹. Consider a list of *ctype functions*²⁰:

- *alnum*²¹
- *alpha*²²
- *cntrl*²³
- *digit*²⁴
- *graph*²⁵
- *lower*²⁶
- *print*²⁷
- *punct*²⁸
- *space*²⁹
- *upper*³⁰
- *xdigit*³¹

Make sure that the proper *locale*³² is set before using one of these.

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-iterable.php>
11. <https://secure.php.net/manual/en/function.is-long.php>
12. <https://secure.php.net/manual/en/function.is-null.php>
13. <https://secure.php.net/manual/en/function.is-numeric.php>
14. <https://secure.php.net/manual/en/function.is-object.php>
15. <https://secure.php.net/manual/en/function.is-real.php>
16. <https://secure.php.net/manual/en/function.is-resource.php>
17. <https://secure.php.net/manual/en/function.is-scalar.php>
18. <https://secure.php.net/manual/en/function.is-string.php>
19. <https://php.net/book.ctype>
20. <https://php.net/ref.ctype>
21. <https://secure.php.net/manual/en/function.ctype-alnum.php>
22. <https://secure.php.net/manual/en/function.ctype-alpha.php>
23. <https://secure.php.net/manual/en/function.ctype-cntrl.php>
24. <https://secure.php.net/manual/en/function.ctype-digit.php>
25. <https://secure.php.net/manual/en/function.ctype-graph.php>
26. <https://secure.php.net/manual/en/function.ctype-lower.php>
27. <https://secure.php.net/manual/en/function.ctype-print.php>
28. <https://secure.php.net/manual/en/function.ctype-punct.php>
29. <https://secure.php.net/manual/en/function.ctype-space.php>
30. <https://secure.php.net/manual/en/function.ctype-upper.php>
31. <https://secure.php.net/manual/en/function.ctype-xdigit.php>
32. <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	<ul style="list-style-type: none">• checkHost• checkMX• groups• message• mode• normalizer• payload
Class	<i>Email</i> ¹
Validator	<i>EmailValidator</i> ²

Basic Usage

Listing 54-1

```
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\Email(
10      *     message = "The email '{{ value }}' is not a valid email.",
11      *     checkMX = true
12      * )
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Email.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/EmailValidator.php>

```

13     */
14     protected $email;
15 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

checkHost

type: boolean **default:** false

Deprecated since version 4.2: This option was deprecated in Symfony 4.2.

If true, then the *checkdnsrr*³ 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.

checkMX

type: boolean **default:** false

Deprecated since version 4.2: This option was deprecated in Symfony 4.2.

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.



This option is not reliable because it depends on the network conditions and some valid servers refuse to respond to those requests.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

3. <https://secure.php.net/manual/en/function.checkdnsrr.php>

4. <https://secure.php.net/manual/en/function.checkdnsrr.php>

mode

type: string **default:** loose

This option is optional and defines the pattern the email address is validated against. Valid values are:

- loose
- strict
- html5

loose

A simple regular expression. Allows all values with an "@" symbol in, and a "." in the second host part of the email address.

strict

Uses the *egulias/email-validator*⁵ library to perform an RFC compliant validation. You will need to install that library to use this mode.

html5

This matches the pattern used for the *HTML5 email input element*⁶.

normalizer

type: a PHP callable⁷ **default:** null

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the 'trim' string to apply the *trim*⁸ PHP function in order to ignore leading and trailing whitespace during validation.

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. <https://packagist.org/packages/egulias/email-validator>

6. <https://www.w3.org/TR/html5/sec-forms.html#email-state-typeemail>

7. <https://www.php.net/callable>

8. <https://secure.php.net/manual/en/function.trim.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	<ul style="list-style-type: none">• charset• charsetMessage• exactMessage• groups• max• maxMessage• min• minMessage• normalizer• payload
Class	<i>Length</i> ¹
Validator	<i>LengthValidator</i> 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

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Length.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LengthValidator.php>

```

1  // src/Entity/Participant.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
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    }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

charset

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

The charset to be used when computing value's length with the *mb_check_encoding*³ and *mb_strlen*⁴ PHP functions.

charsetMessage

type: string **default:** This value does not match the expected {{ charset }} charset.

The message that will be shown if the value is not using the given charset.

You can use the following parameters in this message:

Parameter	Description
{{ charset }}	The expected charset
{{ value }}	The current (invalid) value

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.

You can use the following parameters in this message:

3. <https://secure.php.net/manual/en/function.mb-check-encoding.php>

4. <https://secure.php.net/manual/en/function.mb-strlen.php>

Parameter	Description
<code>{{ limit }}</code>	The exact expected length
<code>{{ value }}</code>	The current (invalid) value

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

max

type: integer

This option is the "max" length value. Validation will fail if the given value's length is **greater** than this max value.

This option is required when the **min** option is not defined.

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.

You can use the following parameters in this message:

Parameter	Description
<code>{{ limit }}</code>	The expected maximum length
<code>{{ value }}</code>	The current (invalid) value

min

type: integer

This option is the "min" length value. Validation will fail if the given value's length is **less** than this min value.

This option is required when the **max** option is not defined.

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.

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.

You can use the following parameters in this message:

Parameter	Description
<code>{{ limit }}</code>	The expected minimum length

Parameter	Description
{{ value }}	The current (invalid) value

normalizer

type: a *PHP callable*⁵ **default:** `null`

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the `'trim'` string to apply the *trim*⁶ PHP function in order to ignore leading and trailing whitespace during validation.

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. <https://www.php.net/callable>

6. <https://secure.php.net/manual/en/function.trim.php>



Chapter 56

Url

Validates that a value is a valid URL string.

Applies to	property or method
Options	<ul style="list-style-type: none">• checkDNS• dnsMessage• groups• message• normalizer• payload• protocols• relativeProtocol
Class	<code>Url</code> ¹
Validator	<code>UrlValidator</code> ²

Basic Usage

Listing 56-1

```
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\Url
10      */
11     protected $bioUrl;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Url.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/UrlValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

checkDNS

type: boolean **default:** false

Deprecated since version 4.1: This option was deprecated in Symfony 4.1 and will be removed in Symfony 5.0, because checking the DNS records is not reliable enough to validate the existence of the host. Use the *checkdnsrr*³ PHP function if you still want to use this kind of validation.

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*⁴ class:

Listing 56-2

```

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\Url(
10        *     checkDNS = "ANY"
11        * )
12        */
13     protected $bioUrl;
14 }
```

This option uses the *checkdnsrr*⁵ 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.

Deprecated since version 4.1: This option was deprecated in Symfony 4.1 and will be removed in Symfony 5.0, because checking the DNS records is not reliable enough to validate the existence of the host. Use the *checkdnsrr*⁶ PHP function if you still want to use this kind of validation.

This message is shown when the **checkDNS** option is set to **true** and the DNS check failed.

Listing 56-3

```

1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
```

3. <https://secure.php.net/manual/en/function.checkdnsrr.php>

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Url.php>

5. <https://secure.php.net/manual/en/function.checkdnsrr.php>

6. <https://secure.php.net/manual/en/function.checkdnsrr.php>

```

8      /**
9       * @Assert\Url(
10        *     dnsMessage = "The host '{{ value }}' could not be resolved."
11        * )
12        */
13        protected $bioUrl;
14    }

```

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid URL.

This message is shown if the URL is invalid.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

Listing 56-4

```

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\Url(
10        *     message = "The url '{{ value }}' is not a valid url",
11        * )
12        */
13        protected $bioUrl;
14    }

```

normalizer

type: a PHP callable⁷ **default:** null

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the 'trim' string to apply the *trim*⁸ PHP function in order to ignore leading and trailing whitespace during validation.

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.

7. <https://www.php.net/callable>

8. <https://secure.php.net/manual/en/function.trim.php>

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.

protocols

type: array **default:** ['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 56-5

```
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\Url(
10      *     protocols = {"http", "https", "ftp"}
11      * )
12      */
13     protected $bioUrl;
14 }
```

relativeProtocol

type: boolean **default:** false

If `true`, the protocol is considered optional when validating the syntax of the given URL. This means that both `http://` and `https://` are valid but also relative URLs that contain no protocol (e.g. `//example.com`).

Listing 56-6

```
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\Url(
10      *     relativeProtocol = true
11      * )
12      */
13     protected $bioUrl;
14 }
```



Chapter 57

Regex

Validates that a value matches a regular expression.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• htmlPattern• match• message• pattern• normalizer• payload
Class	<i>Regex</i> ¹
Validator	<i>RegexValidator</i> 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

```
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\Regex("/^\w+/")
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Regex.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/RegexValidator.php>

```

10     */
11     protected $description;
12 }

```

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

```

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\Regex(
10        *     pattern="/\d/",
11        *     match=false,
12        *     message="Your name cannot contain a number"
13        * )
14        */
15     protected $firstName;
16 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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

```

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\Regex(
10     *     pattern      = "/^[a-z]+$/i",
11     *     htmlPattern = "[a-zA-Z]+$"
12     * )
13     */
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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

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](https://secure.php.net/manual/en/function.preg-match.php)*³ PHP function). However, if match is set to false, then validation will fail if the input string *does* match this pattern.

normalizer

type: a PHP callable⁴ default: **null**

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the **'trim'** string to apply the *[trim](https://secure.php.net/manual/en/function.trim.php)*⁵ PHP function in order to ignore leading and trailing whitespace during validation.

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.

3. <https://secure.php.net/manual/en/function.preg-match.php>

4. <https://www.php.net/callable>

5. <https://secure.php.net/manual/en/function.trim.php>

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	<ul style="list-style-type: none">• groups• message• normalizer• payload• version
Class	<i>Ip</i> ¹
Validator	<i>IpValidator</i> ²

Basic Usage

Listing 58-1

```
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\Ip
10      */
11     protected $ipAddress;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Ip.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IpValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This is not a valid IP address.

This message is shown if the string is not a valid IP address.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

normalizer

type: a PHP callable³ **default:** null

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the **'trim'** string to apply the *trim*⁴ PHP function in order to ignore leading and trailing whitespace during validation.

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.

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

3. <https://www.php.net/callable>

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

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



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	<ul style="list-style-type: none">• groups• message• normalizer• payload• strict• versions
Class	<i>Uuid</i> ³
Validator	<i>UuidValidator</i> ⁴

Basic Usage

Listing 59-1

```
1 // src/Entity/File.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class File
7 {
8     /**
9      * @Assert\Uuid
```

-
1. http://en.wikipedia.org/wiki/Universally_unique_identifier
 2. <http://tools.ietf.org/html/rfc4122>
 3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Uuid.php>
 4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/UuidValidator.php>

```

10      */
11      protected $identifier;
12  }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This is not a valid UUID.

This message is shown if the string is not a valid UUID.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

normalizer

type: a PHP callable⁵ **default:** null

This option allows to define the PHP callable applied to the given value before checking if it is valid.

For example, you may want to pass the 'trim' string to apply the *trim*⁶ PHP function in order to ignore leading and trailing whitespace during validation.

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.

strict

type: boolean **default:** true

5. <https://www.php.net/callable>

6. <https://secure.php.net/manual/en/function.trim.php>

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*⁷. 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.

7. http://en.wikipedia.org/wiki/Universally_unique_identifier#Variants_and_versions



Chapter 60

Json

Validates that a value has valid *JSON*¹ syntax.

Applies to	property or method
Options	<ul style="list-style-type: none">• message• payload
Class	<i>Json</i> ²
Validator	<i>JsonValidator</i> ³

Basic Usage

The **Json** constraint can be applied to a property or a "getter" method:

Listing 60-1

```
1 // src/Entity/Book.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Book
7 {
8     /**
9      * @Assert\Json(
10      *     message = "You've entered an invalid Json."
11      * )
12      */
13     private $chapters;
14 }
```

1. <https://en.wikipedia.org/wiki/JSON>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Json.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/JsonValidator.php>

Options

message

type: string **default:** This value should be valid JSON.

This message is shown if the underlying data is not a valid JSON 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 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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>EqualTo</i> ¹
Validator	<i>EqualToValidator</i> 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

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Person
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/EqualTo.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/EqualToValidator.php>

```

7 {
8     /**
9      * @Assert\EqualTo("Mary")
10     */
11     protected $firstName;
12
13     /**
14      * @Assert\EqualTo(
15      *     value = 20
16      * )
17     */
18     protected $age;
19 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The expected value
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>NotEqualTo</i> ¹
Validator	<i>NotEqualToValidator</i> 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

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Person
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotEqualTo.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotEqualToValidator.php>

```

7 {
8     /**
9      * @Assert\NotEqualTo("Mary")
10     */
11     protected $firstName;
12
13     /**
14      * @Assert\NotEqualTo(
15      *     value = 15
16      * )
17     */
18     protected $age;
19 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The expected value
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>IdenticalTo</i> ¹
Validator	<i>IdenticalToValidator</i> 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
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IdenticalTo.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IdenticalToValidator.php>

```

4 use Symfony\Component\Validator\Constraints as Assert;
5
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

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The expected value
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>NotIdenticalTo</i> ¹
Validator	<i>NotIdenticalToValidator</i> 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
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotIdenticalTo.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotIdenticalToValidator.php>

```

4 use Symfony\Component\Validator\Constraints as Assert;
5
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

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The expected value
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: `mixed` [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>LessThan</i> ¹
Validator	<i>LessThanValidator</i> 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     /**
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LessThan.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LessThanValidator.php>

```

9      * @Assert\LessThan(5)
10     */
11     protected $siblings;
12
13     /**
14      * @Assert\LessThan(
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 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

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The upper limit
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: mixed [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>LessThanOrEqual</i> ¹
Validator	<i>LessThanOrEqualValidator</i> 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

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Person
7 {
8     /**
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LessThanOrEqual.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LessThanOrEqualValidator.php>

```

9      * @Assert\LessThanOrEqual(5)
10     */
11     protected $siblings;
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 $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 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 $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 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 $dateOfBirth;
12 }

```

3. <https://php.net/manual/en/datetime.formats.php>

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The upper limit
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: mixed [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>GreaterThan</i> ¹
Validator	<i>GreaterThanValidator</i> 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

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Person
7 {
8     /**
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThan.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThanValidator.php>


```

9      * @Assert\GreaterThan(5)
10     */
11     protected $siblings;
12
13     /**
14      * @Assert\GreaterThan(
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 next day:

Listing 67-2

```

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\GreaterThan("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 67-3

```

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\GreaterThan("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 67-4

```

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\GreaterThan("+5 hours")
10     */
11     protected $deliveryDate;
12 }

```

3. <https://php.net/manual/en/datetime.formats.php>

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The lower limit
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: mixed [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



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	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>GreaterThanOrEqual</i> ¹
Validator	<i>GreaterThanOrEqualValidator</i> 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

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Person
7 {
8     /**
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThanOrEqual.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThanOrEqualValidator.php>

```

9      * @Assert\GreaterThanOrEqual(5)
10     */
11     protected $siblings;
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

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	The lower limit
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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`.

value

type: mixed [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



Chapter 69

Range

Validates that a given number or **DateTime** object is *between* some minimum and maximum.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• invalidMessage• max• maxMessage• min• minMessage• payload
Class	<i>Range</i> ¹
Validator	<i>RangeValidator</i> ²

Basic Usage

To verify that the "height" field of a class is between "120" and "180", you might add the following:

Listing 69-1

```
1  // src/Entity/Participant.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Participant
7  {
8      /**
9       * @Assert\Range(
10        *     min = 120,
11        *     max = 180,
12        *     minMessage = "You must be at least {{ limit }}cm tall to enter",
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Range.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/RangeValidator.php>

```

13     *      maxMessage = "You cannot be taller than {{ limit }}cm to enter"
14     * )
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 69-2

```

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 69-3

```

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 69-4

```

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      */
14     protected $deliveryDate;
15 }

```

3. <https://php.net/manual/en/datetime.formats.php>

```

12     * )
13     */
14     protected $deliveryDate;
15 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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).

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

max

type: number or string (date format)

This required option is the "max" value. Validation will fail if the given value is **greater** than this max value.

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.

You can use the following parameters in this message:

Parameter	Description
{{ limit }}	The upper limit
{{ value }}	The current (invalid) value

min

type: number or string (date format)

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

4. <https://php.net/manual/en/function.is-numeric.php>

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.

You can use the following parameters in this message:

Parameter	Description
{{ limit }}	The lower limit
{{ value }}	The current (invalid) 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 70

DivisibleBy

Validates that a value is divisible by another value, defined in the options.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload• propertyPath• value
Class	<i>DivisibleBy</i> ¹
Validator	<i>DivisibleByValidator</i> ²

Basic Usage

The following constraints ensure that:

- the weight of the Item is provided in increments of 0.25
- the quantity of the Item must be divisible by 5

Listing 70-1

```
1  // src/Entity/Item.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Item
7  {
8
9      /**
10       * @Assert\DivisibleBy(0.25)
11       */
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/DivisibleBy.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/DivisibleByValidator.php>

```

12     protected $weight;
13
14     /**
15      * @Assert\DivisibleBy(
16      *     value = 5
17      * )
18      */
19     protected $quantity;
20 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be a multiple of {{ compared_value }}.

This is the message that will be shown if the value is not divisible by 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`.

value

type: mixed [default option]

This option is required. It defines the value to compare to. It can be a string, number or object.



Chapter 71

Unique

Validates that all the elements of the given collection are unique (none of them is present more than once). Elements are compared strictly, so '7' and 7 are considered different elements (a string and an integer, respectively).

If you want to apply different validation constraints to the elements of a collection or want to make sure that certain collection keys are present, use the Collection constraint.

If you want to validate that the value of an entity property is unique among all entities of the same type (e.g. the registration email of all users) use the UniqueEntity constraint.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<code>Unique</code> ¹
Validator	<code>UniqueValidator</code> ²

Basic Usage

This constraint can be applied to any property of type `array` or `\Traversable`. In the following example, `$contactEmails` is an array of strings:

Listing 71-1

```
1 // src/Entity/Person.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Unique.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/UniqueValidator.php>

```

6 class Person
7 {
8     /**
9      * @Assert\Unique
10     */
11     protected $contactEmails;
12 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This collection should contain only unique elements.

This is the message that will be shown if at least one element is repeated in the collection.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The repeated 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 72

Positive

New in version 4.3: The **Positive** constraint was introduced in Symfony 4.3.

Validates that a value is a positive number. Zero is neither positive nor negative, so you must use *PositiveOrZero* if you want to allow zero as value.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Positive</i> ¹
Validator	<i>GreaterThanValidator</i> ²

Basic Usage

The following constraint ensures that the **income** of an **Employee** is a positive number (greater than zero):

Listing 72-1

```
1  // src/Entity/Employee.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Employee
7  {
8      /**
9       * @Assert\Positive
10      */
11     protected $income;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Positive.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThanValidator.php>

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be positive.

The default message supplied when the value is not greater than zero.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	Always zero
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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 73

PositiveOrZero

New in version 4.3: The **PositiveOrZero** constraint was introduced in Symfony 4.3.

Validates that a value is a positive number or equal to zero. If you don't want to allow zero as value, use *Positive* instead.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>PositiveOrZero</i> ¹
Validator	<i>GreaterThanOrEqualValidator</i> ²

Basic Usage

The following constraint ensures that the number of **siblings** of a **Person** is positive or zero:

Listing 73-1

```
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       * @Assert\PositiveOrZero
10      */
11     protected $siblings;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/PositiveOrZero.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/GreaterThanOrEqualValidator.php>

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be either positive or zero.

The default message supplied when the value is not greater than or equal to zero.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	Always zero
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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 74

Negative

New in version 4.3: The **Negative** constraint was introduced in Symfony 4.3.

Validates that a value is a negative number. Zero is neither positive nor negative, so you must use *NegativeOrZero* if you want to allow zero as value.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Negative</i> ¹
Validator	<i>LesserThanValidator</i> ²

Basic Usage

The following constraint ensures that the **withdraw** of a bank account **TransferItem** is a negative number (lesser than zero):

Listing 74-1

```
1  // src/Entity/TransferItem.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class TransferItem
7  {
8      /**
9       * @Assert\Negative
10      */
11     protected $withdraw;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Negative.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LesserThanValidator.php>

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be negative.

The default message supplied when the value is not less than zero.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	Always zero
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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

NegativeOrZero

New in version 4.3: The **NegativeOrZero** constraint was introduced in Symfony 4.3.

Validates that a value is a negative number or equal to zero. If you don't want to allow zero as value, use *Negative* instead.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>NegativeOrZero</i> ¹
Validator	<i>LesserThanOrEqualValidator</i> ²

Basic Usage

The following constraint ensures that the `level` of a `UnderGroundGarage` is a negative number or equal to zero:

Listing 75-1

```
1  // src/Entity/TransferItem.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class UnderGroundGarage
7  {
8      /**
9       * @Assert\NegativeOrZero
10      */
11     protected $level;
12 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NegativeOrZero.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LesserThanOrEqualValidator.php>

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value should be either negative or zero.

The default message supplied when the value is not less than or equal to zero.

You can use the following parameters in this message:

Parameter	Description
{{ compared_value }}	Always zero
{{ compared_value_type }}	The expected value type
{{ value }}	The current (invalid) 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 76

Date

Validates that a value is a valid date, meaning 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	<ul style="list-style-type: none">• groups• message• payload
Class	<code>Date</code> ¹
Validator	<code>DateValidator</code> ²

Basic Usage

Listing 76-1

```
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\Date
10      * @var string A "Y-m-d" formatted value
11      */
12      protected $birthday;
13  }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Date.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/DateValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid date.

This message is shown if the underlying data is not a valid date.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 77

DateTime

Validates that a value is a valid "datetime", meaning a string (or an object that can be cast into a string) that follows a specific format.

Applies to	property or method
Options	<ul style="list-style-type: none">• format• groups• message• payload
Class	<i>DateTime</i> ¹
Validator	<i>DateTimeValidator</i> ²

Basic Usage

Listing 77-1

```
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\DateTime
10      * @var string A "Y-m-d H:i:s" formatted value
11      */
12      protected $createdAt;
13  }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/DateTime.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/DateTimeValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

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.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid datetime.

This message is shown if the underlying data is not a valid datetime.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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.

3. <https://secure.php.net/manual/en/datetime.createfromformat.php>



Chapter 78

Time

Validates that a value is a valid time, meaning 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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Time</i> ¹
Validator	<i>TimeValidator</i> ²

Basic Usage

Suppose you have an Event class, with a **startsAt** field that is the time of the day when the event starts:

Listing 78-1

```
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\Time
10      * @var string A "H:i:s" formatted value
11      */
12      protected $startsAt;
13  }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Time.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/TimeValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid time.

This message is shown if the underlying data is not a valid time.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 79

Timezone

New in version 4.3: The **Timezone** constraint was introduced in Symfony 4.3.

Validates that a value is a valid timezone identifier (e.g. **Europe/Paris**).

Applies to	property or method
Options	<ul style="list-style-type: none">• countryCode• groups• intlCompatible• message• payload• zone
Class	<i>Timezone</i> ¹
Validator	<i>TimezoneValidator</i> ²

Basic Usage

Suppose you have a **UserSettings** class, with a **timezone** field that is a string which contains any of the *PHP timezone identifiers*³ (e.g. **America/New_York**):

Listing 79-1

```
1 // src/Entity/UserSettings.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class UserSettings
7 {
8     /**
9      * @Assert\Timezone
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Timezone.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/TimezoneValidator.php>

3. <https://www.php.net/manual/en/timezones.php>

```

10     */
11     protected $timezone;
12 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

countryCode

type: string **default:** null

If the **zone** option is set to `\DateTimeZone::PER_COUNTRY`, this option restricts the valid timezone identifiers to the ones that belong to the given country.

The value of this option must be a valid *ISO 3166-1 alpha-2*⁴ country code (e.g. **CN** for China).

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

intlCompatible

type: boolean **default:** false

This constraint considers valid both the *PHP timezone identifiers*⁵ and the ICU timezones provided by Symfony's *Intl component*

However, the timezones provided by the Intl component can be different from the timezones provided by PHP's Intl extension (because they use different ICU versions). If this option is set to **true**, this constraint only considers valid the values compatible with the PHP `\IntlTimeZone::createTimeZone()` method.

message

type: string **default:** This value is not a valid timezone.

This message is shown if the underlying data is not a valid timezone identifier.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

payload

type: mixed **default:** null

4. https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

5. <https://www.php.net/manual/en/timezones.php>

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.

zone

type: string **default:** \DateTimeZone::ALL

Set this option to any of the following constants to restrict the valid timezone identifiers to the ones that belong to that geographical zone:

- \DateTimeZone::AFRICA
- \DateTimeZone::AMERICA
- \DateTimeZone::ANTARCTICA
- \DateTimeZone::ARCTIC
- \DateTimeZone::ASIA
- \DateTimeZone::ATLANTIC
- \DateTimeZone::AUSTRALIA
- \DateTimeZone::EUROPE
- \DateTimeZone::INDIAN
- \DateTimeZone::PACIFIC

In addition, there are some special zone values:

- \DateTimeZone::ALL accepts any timezone excluding deprecated timezones;
- \DateTimeZone::ALL_WITH_BC accepts any timezone including deprecated timezones;
- \DateTimeZone::PER_COUNTRY restricts the valid timezones to a certain country (which is defined using the `countryCode` option).



Chapter 80

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	<ul style="list-style-type: none">• callback• choices• groups• max• maxMessage• message• min• minMessage• multiple• multipleMessage• payload
Class	<i>Choice</i> ¹
Validator	<i>ChoiceValidator</i> 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 80-1

```
1 // src/Entity/Author.php
2 namespace App\Entity;
3
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Choice.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/ChoiceValidator.php>

```

4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Author
7 {
8     const GENRES = ['fiction', 'non-fiction'];
9
10    /**
11     * @Assert\Choice({"New York", "Berlin", "Tokyo"})
12     */
13    protected $city;
14
15    /**
16     * You can also directly provide an array constant to the "choices" option in the annotation
17     *
18     * @Assert\Choice(choices=Author::GENRES, message="Choose a valid genre.")
19     */
20    protected $genre;
21 }

```

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 access those choices for validation or for building a select form element:

Listing 80-2

```

1 // src/Entity/Author.php
2 namespace App\Entity;
3
4 class Author
5 {
6     public static function getGenres()
7     {
8         return ['fiction', 'non-fiction'];
9     }
10 }

```

You can pass the name of this method to the callback option of the **Choice** constraint.

Listing 80-3

```

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\Choice(callback="getGenres")
10     */
11    protected $genre;
12 }

```

If the callback is defined in a different class and is static, for example **App\Entity\Genre**, you can pass the class name and the method as an array.

Listing 80-4

```

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\Choice(callback={"App\Entity\Genre", "getGenres"})

```



```

10      */
11      protected $genre;
12  }

```

Available Options

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.

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.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

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.

You can use the following parameters in this message:

Parameter	Description
{{ choices }}	A comma-separated list of available choices
{{ value }}	The current (invalid) value

New in version 4.3: The `{{ choices }}` parameter was introduced in Symfony 4.3.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

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.

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.

You can use the following parameters in this message:

Parameter	Description
{{ choices }}	A comma-separated list of available choices
{{ value }}	The current (invalid) value

New in version 4.3: The **{{ choices }}** parameter was introduced in Symfony 4.3.

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.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 81

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.

*If you want to validate that all the elements of the collection are unique use the **Unique** constraint.*

Applies to	property or method
Options	<ul style="list-style-type: none">• allowExtraFields• allowMissingFields• extraFieldsMessage• fields• groups• missingFieldsMessage• payload
Class	<i>Collection</i> ¹
Validator	<i>CollectionValidator</i> ²

Basic Usage

The **Collection** constraint allows you to validate the different keys of a collection individually. Take the following example:

Listing 81-1

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Collection.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CollectionValidator.php>

```

1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  class Author
5  {
6      protected $profileData = [
7          'personal_email' => '...',
8          'short_bio' => '...',
9      ];
10
11     public function setProfileData($key, $value)
12     {
13         $this->profileData[$key] = $value;
14     }
15 }

```

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 81-2

```

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\Collection(
10        *     fields = {
11        *         "personal_email" = @Assert\Email,
12        *         "short_bio" = {
13        *             @Assert\NotBlank,
14        *             @Assert\Length(
15        *                 max = 100,
16        *                 maxMessage = "Your short bio is too long!"
17        *             )
18        *         }
19        *     },
20        *     allowMissingFields = true
21        * )
22        */
23     protected $profileData = [
24         'personal_email' => '...',
25         'short_bio' => '...',
26     ];
27 }

```

Presence and Absence of Fields

By default, this constraint validates more than 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 81-3

```
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\Collection(
10        *     fields={
11        *         "personal_email" = @Assert\Required({@Assert\NotBlank, @Assert\Email}),
12        *         "alternate_email" = @Assert\Optional(@Assert\Email)
13        *     }
14        * )
15        */
16     protected $profileData = ['personal_email'];
17 }
```

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

`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.

`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.

`extraFieldsMessage`

type: string **default:** This field was not expected.

The message shown if `allowExtraFields` is false and an extra field is detected.

You can use the following parameters in this message:

Parameter	Description
{{ field }}	The key of the extra field detected

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.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ field }}	The key of the missing field defined in fields

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 82

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	<ul style="list-style-type: none">• exactMessage• groups• max• maxMessage• min• minMessage• payload
Class	<i>Count</i> ¹
Validator	<i>CountValidator</i> ²

Basic Usage

To verify that the **emails** array field contains between 1 and 5 elements you might add the following:

Listing 82-1

```
1 // src/Entity/Participant.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Participant
7 {
8     /**
9      * @Assert\Count(
10      *     min = 1,
11      *     max = 5,
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Count.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CountValidator.php>


```

12     *      minMessage = "You must specify at least one email",
13     *      maxMessage = "You cannot specify more than {{ limit }} emails"
14     * )
15     */
16     protected $emails = [];
17 }

```

Options

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.

You can use the following parameters in this message:

Parameter	Description
{{ count }}	The current collection size
{{ limit }}	The exact expected collection size

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

max

type: integer

This option is the "max" count value. Validation will fail if the given collection elements count is **greater** than this max value.

This option is required when the **min** option is not defined.

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.

You can use the following parameters in this message:

Parameter	Description
{{ count }}	The current collection size
{{ limit }}	The upper limit

min

type: integer

This option is the "min" count value. Validation will fail if the given collection elements count is **less** than this min value.

This option is required when the **max** option is not defined.

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.

You can use the following parameters in this message:

Parameter	Description
<code>{{ count }}</code>	The current collection size
<code>{{ limit }}</code>	The lower limit

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

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.

If you want to validate that all the elements of the collection are unique use the Unique constraint.

Applies to	class
Options	<ul style="list-style-type: none">• em• entityClass• errorPath• fields• groups• ignoreNull• message• payload• repositoryMethod
Class	<code>UniqueEntity</code> ¹
Validator	<code>UniqueEntityValidator</code> 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 rows in your user table:

Listing 83-1

```
1 // src/Entity/User.php
2 namespace App\Entity;
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntity.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntityValidator.php>

```

3
4 use Doctrine\ORM\Mapping as ORM;
5
6 // DON'T forget the following use statement!!!
7 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
8
9 use Symfony\Component\Validator\Constraints as Assert;
10
11 /**
12  * @ORM\Entity
13  * @UniqueEntity("email")
14  */
15 class User
16 {
17     /**
18      * @ORM\Column(name="email", type="string", length=255, unique=true)
19      * @Assert>Email
20      */
21     protected $email;
22 }

```



This constraint doesn't provide any protection against *race conditions*³. They may occur when another entity is persisted by an external process after this validation has passed and before this entity is actually persisted in the database.



This constraint cannot deal with duplicates found in a collection of items that haven't been persisted as entities yet. You'll need to create your own validator to handle that case.

Options

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.

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:

3. https://en.wikipedia.org/wiki/Race_condition

Listing 83-2

```
1  // src/Entity/Service.php
2  namespace App\Entity;
3
4  use Doctrine\ORM\Mapping as ORM;
5  use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
6
7  /**
8   * @ORM\Entity
9   * @UniqueEntity(
10    *     fields={"host", "port"},
11    *     errorPath="port",
12    *     message="This port is already in use on that host."
13    * )
14   */
15  class Service
16  {
17      /**
18       * @ORM\ManyToOne(targetEntity="App\Entity\Host")
19       */
20      public $host;
21
22      /**
23       * @ORM\Column(type="integer")
24       */
25      public $port;
26  }
```

Now, the message would be bound to the **port** field with this configuration.

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.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

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>"*

You can use the following parameters in this message:

Parameter	Description
<code>{{ value }}</code>	The current (invalid) 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.

repositoryMethod

type: string **default:** findBy

The name of the repository method used to determine the uniqueness. If it's left blank, `findBy()` will be used. The method receives as its argument a `fieldName => value` associative array (where `fieldName` is each of the fields configured in the `fields` option). The method should return a *countable PHP variable*⁴.

4. <https://php.net/manual/function.is-countable.php>



Chapter 84

Language

Validates that a value is a valid language *Unicode language identifier* (e.g. **fr** or **zh-Hant**).

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Language</i> ¹
Validator	<i>LanguageValidator</i> ²

Basic Usage

Listing 84-1

```
1  // src/Entity/User.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class User
7  {
8      /**
9       * @Assert\Language
10      */
11     protected $preferredLanguage;
12 }
```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Language.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LanguageValidator.php>

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid language.

This message is shown if the string is not a valid language code.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 85

Locale

Validates that a value is a valid locale.

The "value" for each locale is any of the *ICU format locale IDs*¹. For example, the two letter *ISO 639-1*² language code (e.g. `fr`), or the language code followed by an underscore (`_`) and the *ISO 3166-1 alpha-2*³ country code (e.g. `fr_FR` for French/France).

Applies to	property or method
Options	<ul style="list-style-type: none">• canonicalize• groups• message• payload
Class	<code>Locale</code> ⁴
Validator	<code>LocaleValidator</code> ⁵

Basic Usage

Listing 85-1

```
1 // src/Entity/User.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class User
7 {
8     /**
9      * @Assert\Locale(
```

-
1. <http://userguide.icu-project.org/locale>
 2. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
 3. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
 4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Locale.php>
 5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LocaleValidator.php>

```

10     *      canonicalize = true
11     * )
12     */
13     protected $locale;
14 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

canonicalize

type: boolean **default:** false

Deprecated since version 4.1: Using this option with value **false** was deprecated in Symfony 4.1 and it will throw an exception in Symfony 5.0. Use **true** instead.

If **true**, the `Locale::canonicalize()`⁶ method will be applied before checking the validity of the given locale (e.g. `FR-fr.utf8` is transformed into `fr_FR`).

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid locale.

This message is shown if the string is not a valid locale.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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.

6. <https://secure.php.net/manual/en/locale.canonicalize.php>



Chapter 86

Country

Validates that a value is a valid *ISO 3166-1 alpha-2*¹ country code.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Country</i> ²
Validator	<i>CountryValidator</i> ³

Basic Usage

Listing 86-1

```
1  // src/Entity/User.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class User
7  {
8      /**
9       * @Assert\Country
10      */
11     protected $country;
12 }
```

1. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Country.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CountryValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This value is not a valid country.

This message is shown if the string is not a valid country code.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 87

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	<ul style="list-style-type: none">• <code>binaryFormat</code>• <code>disallowEmptyMessage</code>• <code>groups</code>• <code>maxSize</code>• <code>maxSizeMessage</code>• <code>mimeTypes</code>• <code>mimeTypesMessage</code>• <code>notFoundMessage</code>• <code>notReadableMessage</code>• <code>payload</code>• <code>uploadCantWriteErrorMessage</code>• <code>uploadErrorMessage</code>• <code>uploadExtensionErrorMessage</code>• <code>uploadFormSizeErrorMessage</code>• <code>uploadIniSizeErrorMessage</code>• <code>uploadNoFileErrorMessage</code>• <code>uploadNoTmpDirErrorMessage</code>

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/File/File.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/File/UploadedFile.php>

	<ul style="list-style-type: none"> • <code>uploadPartialErrorMessage</code>
Class	<i>File</i> ³
Validator	<i>FileValidator</i> ⁴

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 87-1

```

1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\HttpFoundation\File\File;
5
6  class Author
7  {
8      protected $bioFile;
9
10     public function setBioFile(File $file = null)
11     {
12         $this->bioFile = $file;
13     }
14
15     public function getBioFile()
16     {
17         return $this->bioFile;
18     }
19 }
```

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 87-2

```

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\File(
10        *     maxSize = "1024k",
11        *     mimeTypes = {"application/pdf", "application/x-pdf"},
12        *     mimeTypesMessage = "Please upload a valid PDF"
13        *)
14      */
15     protected $bioFile;
16 }
```

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.



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/File.php>

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/FileValidator.php>

Options

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*⁵.

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.

You can use the following parameters in this message:

Parameter	Description
{{ file }}	Absolute file path
{{ name }}	Base file name

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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	Example
(none)	byte	1 byte	4096
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*⁶.

5. http://en.wikipedia.org/wiki/Binary_prefix

6. http://en.wikipedia.org/wiki/Binary_prefix

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.

You can use the following parameters in this message:

Parameter	Description
{{ file }}	Absolute file path
{{ limit }}	Maximum file size allowed
{{ name }}	Base file name
{{ size }}	File size of the given file
{{ suffix }}	Suffix for the used file size unit (see above)

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.

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.

You can use the following parameters in this message:

Parameter	Description
{{ file }}	Absolute file path
{{ name }}	Base file name
{{ type }}	The MIME type of the given file
{{ types }}	The list of allowed MIME types

notFoundMessage

type: string **default:** The file could not be found.

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.

You can use the following parameters in this message:

Parameter	Description
{{ file }}	Absolute file path

7. <http://www.iana.org/assignments/media-types/index.html>

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.

You can use the following parameters in this message:

Parameter	Description
{{ file }}	Absolute file path

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.

uploadCantWriteErrorMessage

type: string **default:** Cannot write temporary file to disk.

The message that is displayed if the uploaded file can't be stored in the temporary folder.

This message has no parameters.

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.

This message has no parameters.

uploadExtensionErrorMessage

type: string **default:** A PHP extension caused the upload to fail.

The message that is displayed if a PHP extension caused the file upload to fail.

This message has no parameters.

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.

This message has no parameters.

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.

You can use the following parameters in this message:

Parameter	Description
{{ limit }}	Maximum file size allowed
{{ suffix }}	Suffix for the used file size unit (see above)

`uploadNoFileErrorMessage`

type: string **default:** No file was uploaded.

The message that is displayed if no file was uploaded.

This message has no parameters.

`uploadNoTmpDirErrorMessage`

type: string **default:** No temporary folder was configured in php.ini.

The message that is displayed if the php.ini setting `upload_tmp_dir` is missing.

This message has no parameters.

`uploadPartialErrorMessage`

type: string **default:** The file was only partially uploaded.

The message that is displayed if the uploaded file is only partially uploaded.

This message has no parameters.



Chapter 88

Image

The `Image` constraint works exactly like the `File` constraint, except that its `mimeType`s and `mimeType`sMessage 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	<ul style="list-style-type: none">• <code>allowLandscape</code>• <code>allowLandscapeMessage</code>• <code>allowPortrait</code>• <code>allowPortraitMessage</code>• <code>allowSquare</code>• <code>allowSquareMessage</code>• <code>corruptedMessage</code>• <code>detectCorrupted</code>• <code>groups</code>• <code>maxHeight</code>• <code>maxHeightMessage</code>• <code>maxPixels</code>• <code>maxPixelsMessage</code>• <code>maxRatio</code>• <code>maxRatioMessage</code>• <code>maxWidth</code>• <code>maxWidthMessage</code>• <code>mimeType</code>s• <code>mimeType</code>sMessage• <code>minHeight</code>• <code>minHeightMessage</code>• <code>minPixels</code>• <code>minPixelsMessage</code>• <code>minRatio</code>• <code>minRatioMessage</code>

	<ul style="list-style-type: none"> • minWidth • minWidthMessage • sizeNotDetectedMessage • See <i>File</i> for inherited options
Class	<i>Image</i> ¹
Validator	<i>ImageValidator</i> 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 88-1

```

1  // src/Entity/Author.php
2  namespace App\Entity;
3
4  use Symfony\Component\HttpFoundation\File\File;
5
6  class Author
7  {
8      protected $headshot;
9
10     public function setHeadshot(File $file = null)
11     {
12         $this->headshot = $file;
13     }
14
15     public function getHeadshot()
16     {
17         return $this->headshot;
18     }
19 }
```

To guarantee that the **headshot** **File** object is a valid image and that it is between a certain size, add the following:

Listing 88-2

```

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\Image(
10        *     minWidth = 200,
11        *     maxWidth = 400,
12        *     minHeight = 200,
13        *     maxHeight = 400
14        * )
15       */
16     protected $headshot;
17 }
```

The **headshot** property is validated to guarantee that it is a real image and that it is between a certain width and height.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Image.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/ImageValidator.php>

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 88-3

```
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\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.

allowLandscape

type: Boolean **default:** true

If this option is false, the image cannot be landscape oriented.

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**.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current height
{{ width }}	The current width

allowPortrait

type: Boolean **default:** true

If this option is false, the image cannot be portrait oriented.

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**.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current height
{{ width }}	The current width

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 leave this option as its default **true** value and set allowLandscape and allowPortrait both to **false**.

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**.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current height
{{ width }}	The current width

corruptedMessage

type: string **default:** The image file is corrupted.

The error message when the detectCorrupted option is enabled and the image is corrupted.

This message has no parameters.

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*³ function, which requires the *PHP GD extension*⁴ to be enabled.

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

maxHeight

type: integer

3. <https://secure.php.net/manual/en/function.imagecreatefromstring.php>

4. <http://php.net/manual/en/book.image.php>

If set, the height of the image file must be less than or equal to this value in pixels.

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.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current (invalid) height
{{ max_height }}	The maximum allowed height

maxPixels

type: integer

If set, the amount of pixels of the image file must be less than or equal to this value.

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.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current image height
{{ max_pixels }}	The maximum allowed amount of pixels
{{ pixels }}	The current amount of pixels
{{ width }}	The current image width

maxRatio

type: float

If set, the aspect ratio (**width / height**) of the image file must be less than or equal to this value.

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.

You can use the following parameters in this message:

Parameter	Description
{{ max_ratio }}	The maximum required ratio

Parameter	Description
{{ ratio }}	The current (invalid) ratio

maxWidth

type: integer

If set, the width of the image file must be less than or equal to this value in pixels.

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.

You can use the following parameters in this message:

Parameter	Description
{{ max_width }}	The maximum allowed width
{{ width }}	The current (invalid) width

mimeTypes

type: array or string **default:** image/*

You can find a list of existing image mime types on the *IANA website*⁵.

mimeTypesMessage

type: string **default:** This file is not a valid image.

minHeight

type: integer

If set, the height of the image file must be greater than or equal to this value in pixels.

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.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current (invalid) height
{{ min_height }}	The minimum required height

5. <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.

minPixelsMessage

type: string **default:** The image has too 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.

You can use the following parameters in this message:

Parameter	Description
{{ height }}	The current image height
{{ min_pixels }}	The minimum required amount of pixels
{{ pixels }}	The current amount of pixels
{{ width }}	The current image width

minRatio

type: float

If set, the aspect ratio (**width / height**) of the image file must be greater than or equal to this value.

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.

You can use the following parameters in this message:

Parameter	Description
{{ min_ratio }}	The minimum required ratio
{{ ratio }}	The current (invalid) ratio

minWidth

type: integer

If set, the width of the image file must be greater than or equal to this value in pixels.

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.

You can use the following parameters in this message:

Parameter	Description
{{ min_width }}	The minimum required width
{{ width }}	The current (invalid) width

sizeNotDetectedMessage

type: string **default:** The size of the image could not be detected.

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.

This message has no parameters.



Chapter 89

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	<ul style="list-style-type: none">• groups• message• payload• schemes
Class	<i>CardScheme</i> ¹
Validator	<i>CardSchemeValidator</i> ²

Basic Usage

To use the **CardScheme** validator, apply it to a property or method on an object that will contain a credit card number.

Listing 89-1

```
1  // src/Entity/Transaction.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Transaction
7  {
8      /**
9       * @Assert\CardScheme(
10        *     schemes={"VISA"},
11        *     message="Your credit card number is invalid."
12        * )
13        */
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CardScheme.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CardSchemeValidator.php>

```
14     protected $cardNumber;  
15 }
```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** Unsupported card type or invalid card number.

The message shown when the value does not pass the **CardScheme** check.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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.

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
- MIR

- UATP
- VISA

New in version 4.3: The **UATP** and **MIR** number schemes were introduced in Symfony 4.3.

For more information about the used schemes, see *Wikipedia: Issuer identification number (IIN)*³.

3. https://en.wikipedia.org/wiki/Bank_card_number#Issuer_identification_number_.28IIN.29



Chapter 90

Currency

Validates that a value is a valid 3-letter ISO 4217¹ currency name.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Currency</i> ²
Validator	<i>CurrencyValidator</i> 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 90-1

```
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\Currency
10     */
11     protected $currency;
12 }
```

1. https://en.wikipedia.org/wiki/ISO_4217

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Currency.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CurrencyValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 91

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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Luhn</i> ²
Validator	<i>LuhnValidator</i> ³

Basic Usage

To use the Luhn validator, apply it to a property on an object that will contain a credit card number.

Listing 91-1

```
1  // src/Entity/Transaction.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Transaction
7  {
8      /**
9       * @Assert\Luhn(message="Please check your credit card number.")
10      */
11     protected $cardNumber;
12 }
```

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

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Luhn.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/LuhnValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** Invalid card number.

The default message supplied when the value does not pass the Luhn check.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 92

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	<ul style="list-style-type: none">• groups• message• payload
Class	<i>Iban</i> ²
Validator	<i>IbanValidator</i> ³

Basic Usage

To use the Iban validator, apply it to a property on an object that will contain an International Bank Account Number.

Listing 92-1

```
1  // src/Entity/Transaction.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Transaction
7  {
8      /**
9       * @Assert\Iban(
10        *     message="This is not a valid International Bank Account Number (IBAN)."
11        * )
12        */
```

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

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Iban.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IbanValidator.php>

```
13     protected $bankAccountNumber;  
14 }
```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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 93

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. You may also check that the BIC is associated with a given IBAN.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• iban• ibanMessage• ibanPropertyPath• message• payload
Class	<i>Bic</i> ²
Validator	<i>BicValidator</i> ³

Basic Usage

To use the Bic validator, apply it to a property on an object that will contain a Business Identifier Code (BIC).

Listing 93-1

```
1  // src/Entity/Transaction.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Transaction
7  {
8      /**
9       * @Assert\Bic
```

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

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Bic.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/BicValidator.php>

```

10     */
11     protected $businessIdentifierCode;
12 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

iban

type: string **default:** null

New in version 4.3: The **iban** option was introduced in Symfony 4.3.

An IBAN value to validate that the BIC is associated with it.

ibanMessage

type: string **default:** This Business Identifier Code (BIC) is not associated with IBAN {{ iban }}.

New in version 4.3: The **ibanMessage** option was introduced in Symfony 4.3.

The default message supplied when the value does not pass the combined BIC/IBAN check.

ibanPropertyPath

type: string **default:** null

New in version 4.3: The **ibanPropertyPath** option was introduced in Symfony 4.3.

It defines the object property whose value stores the IBAN used to check the BIC with.

For example, if you want to compare the **\$bic** property of some object with regard to the **\$iban** property of the same object, use **propertyPath="iban"** in the comparison constraint of **\$bic**.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) BIC 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 94

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	<ul style="list-style-type: none">• bothIsbnMessage• groups• isbn10Message• isbn13Message• message• payload• type
Class	<i>Isbn</i> ²
Validator	<i>IsbnValidator</i> ³

Basic Usage

To use the **Isbn** validator, apply it to a property or method on an object that will contain an ISBN.

Listing 94-1

```
1 // src/Entity/Book.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Book
7 {
8     /**
9      * @Assert\Isbn(
10      *     type = "isbn10",
```

1. <https://en.wikipedia.org/wiki/Isbn>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Isbn.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IsbnValidator.php>

```

11     *    message = "This value is not valid."
12     * )
13     */
14     protected $isbn;
15 }

```



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Available Options

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) value

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.

You can use the following parameters in this message:

Parameter	Description
{{ value }}	The current (invalid) 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.

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.



Chapter 95

Issn

Validates that a value is a valid *International Standard Serial Number (ISSN)*¹.

Applies to	property or method
Options	<ul style="list-style-type: none">• caseSensitive• groups• message• payload• requireHyphen
Class	<i>Issn</i> ²
Validator	<i>IssnValidator</i> ³

Basic Usage

Listing 95-1

```
1  // src/Entity/Journal.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Journal
7  {
8      /**
9       * @Assert\Issn
10      */
11     protected $issn;
12 }
```

1. <https://en.wikipedia.org/wiki/Issn>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Issn.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/IssnValidator.php>



As with most of the other constraints, **null** and empty strings are considered valid values. This is to allow them to be optional values. If the value is mandatory, a common solution is to combine this constraint with *NotBlank*.

Options

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'.

groups

type: `array` | `string`

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: `string` default: `This value is not a valid ISSN.`

The message shown if the given value is not a valid ISSN.

You can use the following parameters in this message:

Parameter	Description
<code>{{ value }}</code>	The current (invalid) 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.

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.



Chapter 96

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 instead of displaying custom errors at the top of the form, you can display them next to the field they apply to.

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 or property/method
Options	<ul style="list-style-type: none">• callback• groups• payload
Class	<code>Callback</code> ¹
Validator	<code>CallbackValidator</code> ²

Configuration

Listing 96-1

```
1 // src/Entity/Author.php
2 namespace App\Entity;
3
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Callback.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/CallbackValidator.php>

```

4 use Symfony\Component\Validator\Constraints as Assert;
5 use Symfony\Component\Validator\Context\ExecutionContextInterface;
6
7 class Author
8 {
9     /**
10      * @Assert\Callback
11      */
12     public function validate(ExecutionContextInterface $context, $payload)
13     {
14         // ...
15     }
16 }

```

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 96-2

```

1 // ...
2 use Symfony\Component\Validator\Context\ExecutionContextInterface;
3
4 class Author
5 {
6     // ...
7     private $firstName;
8
9     public function validate(ExecutionContextInterface $context, $payload)
10    {
11        // somehow you have an array of "fake names"
12        $fakeNames = [/* ... */];
13
14        // check if the name is actually a fake name
15        if (in_array($this->getFirstName(), $fakeNames)) {
16            $context->buildViolation('This name sounds totally fake!')
17                ->atPath('firstName')
18                ->addViolation();
19        }
20    }
21 }

```

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 96-3

```

1 public static function validate($object, ExecutionContextInterface $context, $payload)
2 {
3     // somehow you have an array of "fake names"
4     $fakeNames = [/* ... */];
5
6     // check if the name is actually a fake name
7     if (in_array($object->getFirstName(), $fakeNames)) {
8         $context->buildViolation('This name sounds totally fake!')
9             ->atPath('firstName')
10             ->addViolation();
11     }
12 }
13 }

```

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 96-4

```
1 namespace Acme;
2
3 use Symfony\Component\Validator\Context\ExecutionContextInterface;
4
5 class Validator
6 {
7     public static function validate($object, ExecutionContextInterface $context, $payload)
8     {
9         // ...
10    }
11 }
```

You can then use the following configuration to invoke this validator:

Listing 96-5

```
1 // src/Entity/Author.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 /**
7  * @Assert\Callback({"Acme\Validator", "validate"})
8  */
9 class Author
10 {
11 }
```



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 96-6

```
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 use Symfony\Component\Validator\Mapping\ClassMetadata;
7
8 class Author
9 {
10     public static function loadValidatorMetadata(ClassMetadata $metadata)
11     {
12         $callback = function ($object, ExecutionContextInterface $context, $payload) {
13             // ...
14         };
15
16         $metadata->addConstraint(new Assert\Callback($callback));
17     }
18 }
```

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 [`'<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.

groups

type: `array` | `string`

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Context/ExecutionContextInterface.php>

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Context/ExecutionContextInterface.php>



Chapter 97

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	<ul style="list-style-type: none">• expression• groups• message• payload• values
Class	<i>Expression</i> ¹
Validator	<i>ExpressionValidator</i> ²

Basic Usage

Imagine you have a class **BlogPost** with **category** and **isTechnicalPost** properties:

Listing 97-1

```
1  // src/Model/BlogPost.php
2  namespace App\Model;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class BlogPost
7  {
8      private $category;
9
10     private $isTechnicalPost;
11
12     // ...
13
14     public function getCategory()
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Expression.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/ExpressionValidator.php>


```

15     {
16         return $this->category;
17     }
18
19     public function setIsTechnicalPost($isTechnicalPost)
20     {
21         $this->isTechnicalPost = $isTechnicalPost;
22     }
23
24     // ...
25 }

```

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 97-2

```

1  // src/Model/BlogPost.php
2  namespace App\Model;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  /**
7   * @Assert\Expression(
8   *     "this.getCategory() in ['php', 'symfony'] or !this.isTechnicalPost()",
9   *     message="If this is a tech post, the category should be either php or symfony!"
10  * )
11  */
12  class BlogPost
13  {
14      // ...
15  }

```

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 97-3

```

1  // src/Model/BlogPost.php
2  namespace App\Model;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class BlogPost
7  {
8      // ...
9
10     /**
11      * @Assert\Expression(
12      *     "this.getCategory() in ['php', 'symfony'] or value == false",
13      *     message="If this is a tech post, the category should be either php or symfony!"
14      * )
15      */
16     private $isTechnicalPost;
17
18     // ...
19 }

```

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);

groups

type: `array` | `string`

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: `string` **default:** `This value is not valid.`

The default message supplied when the expression evaluates to false.

You can use the following parameters in this message:

Parameter	Description
<code>{{ value }}</code>	The current (invalid) 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.

values

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

The values of the custom variables used in the expression. Values can be of any type (numeric, boolean, strings, null, etc.)

Listing 97-4

```
1 // src/Model/Analysis.php
2 namespace App\Model;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Analysis
7 {
```

```
8      /**
9      * @Assert\Expression(
10     *     "value + error_margin < threshold",
11     *     values = { "error_margin": 0.25, "threshold": 1.5 }
12     * )
13     */
14     private $metric;
15
16     // ...
17 }
```



Chapter 98

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	<ul style="list-style-type: none">• constraints• groups• payload
Class	<i>All</i> ¹
Validator	<i>AllValidator</i> ²

Basic Usage

Suppose that you have an array of strings and you want to validate each entry in that array:

Listing 98-1

```
1  // src/Entity/User.php
2  namespace App\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class User
7  {
8      /**
9       * @Assert\All({
10        *     @Assert\NotBlank,
11        *     @Assert\Length(min=5)
12        * })
13       */
14     protected $favoriteColors = [];
15 }
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/All.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/AllValidator.php>

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.

groups

type: **array** | **string**

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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 99

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	<ul style="list-style-type: none">• groups• message• payload
Class	<code>UserPassword</code> ¹
Validator	<code>UserPasswordValidator</code> 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 99-1

```
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
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/Validator/Constraints/UserPassword.php>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Core/Validator/Constraints/UserPasswordValidator.php>

```

7 {
8     /**
9     * @SecurityAssert\UserPassword(
10     *     message = "Wrong value for your current password"
11     * )
12     */
13     protected $oldPassword;
14 }

```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

This message has no parameters.

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 100

NotCompromisedPassword

New in version 4.3: The **NotCompromisedPassword** constraint was introduced in Symfony 4.3.

Validates that the given password has not been compromised by checking that it is not included in any of the public data breaches tracked by *haveibeenpwned.com*¹.

Applies to	property or method
Options	<ul style="list-style-type: none">• groups• message• payload• skipOnError• threshold
Class	<i>NotCompromisedPassword</i> ²
Validator	<i>NotCompromisedPasswordValidator</i> ³

Basic Usage

The following constraint ensures that the **rawPassword** property of the **User** class doesn't store a compromised password:

Listing 100-1

```
1 // src/Entity/User.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class User
7 {
8     /**
```

1. <https://haveibeenpwned.com/>

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotCompromisedPassword.php>

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/NotCompromisedPasswordValidator.php>


```

9      * @Assert\NotCompromisedPassword
10     */
11     protected $rawPassword;
12 }

```

In order to make the password validation, this constraint doesn't send the raw password value to the **haveibeenpwned.com** API. Instead, it follows a secure process known as *k-anonymity password validation*⁴.

In practice, the raw password is hashed using SHA-1 and only the first bytes of the hash are sent. Then, the **haveibeenpwned.com** API compares those bytes with the SHA-1 hashes of all leaked passwords and returns the list of hashes that start with those same bytes. That's how the constraint can check if the password has been compromised without fully disclosing it.

For example, if the password is **test**, the entire SHA-1 hash is **a94a8fe5ccb19ba61c4c0873d391e987982fbbd3** but the validator only sends **a94a8** to the **haveibeenpwned.com** API.

When using this constraint inside a Symfony application, define the `not_compromised_password` option to avoid making HTTP requests in the `dev` and `test` environments.

Available Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

message

type: string **default:** This password has been leaked in a data breach, it must not be used. Please use another password.

The default message supplied when the password has been compromised.

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.

skipOnError

type: boolean **default:** false

When the HTTP request made to the **haveibeenpwned.com** API fails for any reason, an exception is thrown (no validation error is displayed). Set this option to **true** to not throw the exception and consider the password valid.

4. <https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/>

threshold

type: integer **default:** 1

This value defines the number of times a password should have been leaked publicly to consider it compromised. Think carefully before setting this option to a higher value because it could decrease the security of your application.



Chapter 101

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	<ul style="list-style-type: none">• groups• payload• traverse
Class	<i>Valid</i> 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 101-1

```
1 // src/Entity/Address.php
2 namespace App\Entity;
3
4 class Address
5 {
6     protected $street;
7     protected $zipCode;
8 }
```

Listing 101-2

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Valid.php>

```

1 // src/Entity/Author.php
2 namespace App\Entity;
3
4 class Author
5 {
6     protected $firstName;
7     protected $lastName;
8     protected $address;
9 }

```

Listing 101-3

```

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 101-4

```

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 101-5

```
1 App\Entity\Author.address.zipCode:
2     This value is too long. It should have 5 characters or less.
```

Options

groups

type: array | string

It defines the validation group or groups this constraint belongs to. Read more about *validation groups*.

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.

traverse

type: boolean **default:** true

If this constraint is applied to a **Traversable**, then all containing values will be validated if this option is set to **true**. This option is ignored on arrays: Arrays are traversed in either case. Keys are not validated.



Chapter 102

Traverse

Objects do not validate nested objects by default unless explicitly using this constraint. If only specific nested objects should be validated by cascade, consider using the *Valid* instead.

Applies to	class
Options	<ul style="list-style-type: none">• payload
Class	<i>Traverse</i> ¹

Basic Usage

In the following example, create three classes **Book**, **Author** and **Editor** that all have constraints on their properties. Furthermore, **Book** stores an **Author** and an **Editor** instance that must be valid too. Instead of adding the **Valid** constraint to both fields, configure the **Traverse** constraint on the **Book** class.

Listing 102-1

```
1 // src/Entity/Book.php
2 namespace App\Entity;
3
4 use Doctrine\ORM\Mapping as ORM;
5 use Symfony\Component\Validator\Constraints as Assert;
6
7 /**
8  * @ORM\Entity
9  * @Assert\Traverse
10 */
11 class Book
12 {
13     /**
14      * @var Author
15      *
16      * @ORM\ManyToOne(targetEntity="App\Entity\Author")
17      */
```

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/Constraints/Traverse.php>

```

18     protected $author;
19
20     /**
21      * @var Editor
22      *
23      * @ORM\ManyToOne(targetEntity="App\Entity\Editor")
24      */
25     protected $editor;
26
27     // ...
28 }

```

Options

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 103

Twig Extensions Defined by Symfony

Twig is the template engine used in Symfony applications. There are tens of *default filters and functions defined by Twig*¹, but Symfony also defines some filters, functions and tags to integrate the various Symfony components with Twig templates. This article explains them all.



If these extensions provided by Symfony are not enough, you can *create a custom Twig extension* to define even more filters and functions.

Functions

render

Listing 103-1 1 `{{ render(uri, options = []) }}`

uri

type: string | ControllerReference

options (optional)

type: array **default:** []

Makes a request to the given internal URI or controller and returns the result. The render strategy can be specified in the **strategy** key of the options. It's commonly used to embed controllers in templates.

render_esi

Listing 103-2 1 `{{ render_esi(uri, options = []) }}`

uri

type: string | ControllerReference

1. <https://twig.symfony.com/doc/2.x/#reference>

options (*optional*)

type: array **default:** []

It's similar to the `render` function and defines the same arguments. However, it generates an ESI tag when *ESI support* is enabled or falls back to the behavior of `render` otherwise.



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 103-3 1 {{ 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 103-4 1 {{ asset(path, packageName = null) }}

path

type: string

packageName (*optional*)

type: string | null **default:** null

Returns the public path of the given asset path (which can be a CSS file, a JavaScript file, an image path, etc.). This function takes into account where the application is installed (e.g. in case the project is accessed in a host subdirectory) and the optional asset package base path.

Symfony provides various cache busting implementations via the `version`, `version_strategy`, and `json_manifest_path` configuration options.

Read more about linking to web assets from templates.

asset_version

Listing 103-5 1 {{ asset_version(packageName = null) }}

packageName (*optional*)

type: string | null **default:** null

Returns the current version of the package, more information in [Linking to CSS, JavaScript and Image Assets](#).

csrf_token

Listing 103-6 1 {{ csrf_token(intention) }}

intention

type: string - an arbitrary string used to generate the token value.

Renders a CSRF token. Use this function if you want *CSRF protection* in a regular HTML form not managed by the Symfony Form component.

is_granted

Listing 103-7 1 {{ is_granted(role, object = null, field = null) }}

role

type: string, string[]

object (optional)

type: object

field (optional)

type: string

Returns **true** if the current user has the given role. If several roles are passed in an array, **true** is returned if the user has at least one of them.

Optionally, an object can be passed to be used by the voter. More information can be found in Access Control in Templates.

logout_path

Listing 103-8 1 {{ 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 103-9 1 {{ 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 103-10 1 {{ path(route_name, route_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.

Read more about Symfony routing and about creating links in Twig templates.

url

Listing 103-11 1 {{ url(route_name, route_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.

Read more about Symfony routing and about creating links in Twig templates.

absolute_url

Listing 103-12 1 {{ absolute_url(path) }}

path

type: string

Returns the absolute URL from the passed relative path. Combine it with the `asset()` function to generate absolute URLs for web assets. Read more about Linking to CSS, JavaScript and Image Assets.

relative_path

Listing 103-13 1 {{ 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 103-14 1 {{ relative_path('http://example.com/human.txt') }}
                2 {# ../human.txt #}
                3
                4 {{ relative_path('http://example.com/products/products_icon.png') }}
                5 {# products_icon.png #}
```

expression

Creates an *Expression*² related to the *ExpressionLanguage* component.

Form Related Functions

The following functions related to Symfony Forms are also available. They are explained in the article about *customizing form rendering*:

- `form()`
- `form_start()`
- `form_end()`
- `form_widget()`
- `form_errors()`
- `form_label()`
- `form_help()`
- `form_row()`
- `form_rest()`

Filters

humanize

Listing 103-15 1 `{{ 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 103-16 1 `{{ 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.

2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/ExpressionLanguage/Expression.php>

transchoice

Deprecated since version 4.2: The **transchoice** filter is deprecated since Symfony 4.2 and will be removed in 5.0. Use the *ICU MessageFormat* with the **trans** filter instead.

Listing 103-17 1 `{{ 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 103-18 1 `{{ 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 103-19 1 `{{ 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()*³, but includes the type in the output.

3. `#reference-yaml_encode`

abbr_class

Listing 103-20 1 {{ 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 103-21 1 {{ 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 103-22 1 {{ args|format_args }}

args
type: array

Generates a string with the arguments and their types (within `` elements).

format_args_as_text

Listing 103-23 1 {{ args|format_args_as_text }}

args
type: array

Equal to the `format_args` filter, but without using HTML tags.

file_excerpt

Listing 103-24 1 {{ 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 103-25 1 `{{ 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.project_dir** (showing the full path in a tooltip on hover).

format_file_from_text

Listing 103-26 1 `{{ text|format_file_from_text }}`

text

type: string

Uses `format_file` to improve the output of default PHP errors.

file_link

Listing 103-27 1 `{{ file|file_link(line) }}`

file

type: string

line

type: integer

Generates a link to the provided file and line number using a preconfigured scheme.

file_relative

Listing 103-28 1 `{{ file|file_relative }}`

file

type: string

It transforms the given absolute file path into a new file path relative to project's root directory:

Listing 103-29 1 `{{ ' /var/www/blog/templates/admin/index.html.twig'|file_relative }}`
2 `{# if project root dir is '/var/www/blog/', it returns 'templates/admin/index.html.twig' #}`

If the given file path is out of the project directory, a **null** value will be returned.

Tags

form_theme

Listing 103-30 1 {% 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 103-31 1 {% 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 Using Twig Tags.

transchoice

Deprecated since version 4.2: The **transchoice** tag is deprecated since Symfony 4.2 and will be removed in 5.0. Use the *ICU MessageFormat* with the **trans** tag instead.

Listing 103-32 1 {% 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 Using Twig Tags.

trans_default_domain

Listing 103-33


```
1 {% trans_default_domain domain %}
```

domain

type: string

This will set the default domain in the current template.

stopwatch

Listing 103-34 1 {% stopwatch 'name' %}...{% endstopwatch %}

This will time the run time of the code inside it and put that on the timeline of the WebProfilerBundle.

Tests

The following tests related to Symfony Forms are available. They are explained in the article about *customizing form rendering*:

- selectedchoice()
- rootform()

Global Variables

app

The **app** variable is injected automatically by Symfony in all templates and provides access to lots of useful application information. Read more about the Twig global app variable.



Chapter 104

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
kernel.reset	Allows to clean up services between requests
mime.mime_type_guesser	Add your own logic for guessing MIME types

Tag Name	Usage
monolog.logger	Logging with a custom logging channel
monolog.processor	Add a custom processor for logging
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 <code>serializer</code> service
serializer.normalizer	Register a new normalizer in the <code>serializer</code> 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
twig.runtime	Register a lazy-loaded Twig Extension
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 104-1

```

1  services:
2      app.mysql_lock:
3          class: App\Lock\MysqlLock
4          public: false
5      app.postgresql_lock:
6          class: App\Lock\PostgresqlLock
7          public: false
8      app.sqlite_lock:
9          class: App\Lock\SqliteLock
10         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 104-2

```

1  services:
2      app.mysql_lock:
3          # ...
4      app.postgresql_lock:
5          # ...
6      app.sqlite_lock:
7          # ...
8      app.lock:
9          tags:
10             - { 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 events* article.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Controller/ArgumentValueResolverInterface.php>

doctrine.event_subscriber

Purpose: Add a Doctrine event subscriber

For details on creating Doctrine event subscribers, read the *Doctrine events* 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 104-3 1 // src/Cache/MyClearer.php
2 namespace App\Cache;
3
4 use Symfony\Component\HttpKernel\CacheClearer\CacheClearerInterface;
5
6 class MyClearer implements CacheClearerInterface
7 {
8     public function clear($cacheDirectory)
9     {
10         // clear your cache
11     }
12 }
```

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 104-4

```

1 services:
2     App\Cache\MyClearer:
3         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 104-5

```

1 // src/Cache/MyCustomWarmer.php
2 namespace App\Cache;
3
4 use Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerInterface;
5
6 class MyCustomWarmer implements CacheWarmerInterface
7 {
8     public function warmUp($cacheDirectory)
9     {
10         // ... do some sort of operations to "warm" your cache
11     }
12
13     public function isOptional()
14     {
15         return true;
16     }
17 }

```

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 104-6

```

1 services:
2     App\Cache\MyCustomWarmer:
3         tags:
4             - { name: kernel.cache_warmer, priority: 0 }

```



The **priority** is optional and its value is a positive or negative integer that defaults to **0**. The higher the number, the earlier that warmers are 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://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/CacheWarmer/CacheWarmerInterface.php>

In addition to your own cache warmers, Symfony components and third-party bundles define cache warmers too for their own purposes. You can list them all with the following command:

```
Listing 104-7 1 $ php bin/console debug:container --tag=kernel.cache_warmer
```

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*³, register it as a service, then tag it with `kernel.fragment_renderer`.

kernel.reset

Purpose: Clean up services between requests

During the `kernel.terminate` event, Symfony looks for any service tagged with the `kernel.reset` tag to reinitialize their state. This is done by calling to the method whose name is configured in the `method` argument of the tag.

This is mostly useful when running your projects in application servers that reuse the Symfony application between requests to improve performance. This tag is applied for example to the built-in *data collectors* of the profiler to delete all their information.

mime.mime_type_guesser

Purpose: Add your own logic for guessing MIME types

3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Fragment/FragmentRendererInterface.php>

This tag is used to register your own MIME type guessers in case the guessers provided by the *Mime component* don't fit your needs.

New in version 4.3: The `mime.mime_type_guesser` tag was introduced in Symfony 4.3.

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 104-8

```
1 services:
2     App\Log\CustomLogger:
3         arguments: ['@logger']
4         tags:
5             - { name: monolog.logger, channel: app }
```



You can create *custom channels* and even autowire logging channels.

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 104-9

```
1 services:
2     Monolog\Processor\IntrospectionProcessor:
3         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 104-10

```
1 services:
2     Monolog\Processor\IntrospectionProcessor:
3         tags:
4             - { 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 104-11

```
1 services:
2     Monolog\Processor\IntrospectionProcessor:
```



```

3     tags:
4         - { 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 104-12

```

1 services:
2     App\Routing\CustomLoader:
3         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*⁴ and your custom authentication listener extends *AbstractAuthenticationListener*⁵, then your custom authentication listener will automatically have this tag applied and it will function automatically.

4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.php>

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.php>

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*⁶ and *DecoderInterface*⁷.

For more details, see *How to Use the Serializer*.

serializer.normalizer

Purpose: Register a new normalizer in the Serializer service

The class that's tagged should implement the *NormalizerInterface*⁸ and *DenormalizerInterface*⁹.

For more details, see *How to Use the Serializer*.

The priorities of the default normalizers can be found in the *registerSerializerConfiguration()*¹⁰ 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*¹¹.

Several SwiftMailer plugins are core to Symfony and can be activated via different configuration. For details, see *Mailer Configuration Reference (SwiftmailerBundle)*.

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Serializer/Encoder/EncoderInterface.php>

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Serializer/Encoder/DecoderInterface.php>

8. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Serializer/Normalizer/NormalizerInterface.php>

9. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Serializer/Normalizer/DenormalizerInterface.php>

10. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/FrameworkBundle/DependencyInjection/FrameworkExtension.php>

11. <http://swiftmailer.org/docs/plugins.html>

templating.helper

Purpose: Make your service available in PHP templates

Deprecated since version 4.3: The **templating.helper** tag is deprecated since version 4.3 and will be removed in 5.0; use Twig instead.

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 104-13 1 services:
                2     App\Templating\AppHelper:
                3         tags:
                4             - { name: templating.helper, alias: alias_name }
```

translation.loader

Purpose: To register a custom service that loads translations

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 104-14 1 services:
                2     App\Translation\MyCustomLoader:
                3         tags:
                4             - { 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*¹² and a *PhpExtractor*¹³, which help to find and extract translation keys from Twig templates and PHP files.

12. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bridge/Twig/Translation/TwigExtractor.php>

13. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Extractor/PhpExtractor.php>

You can create your own extractor by creating a class that implements *ExtractorInterface*¹⁴ and tagging the service with `translation.extractor`. The tag has one required option: `alias`, which defines the name of the extractor:

```
Listing 104-15 1 // src/Acme/DemoBundle/Translation/FooExtractor.php
2 namespace Acme\DemoBundle\Translation;
3
4 use Symfony\Component\Translation\Extractor\ExtractorInterface;
5 use Symfony\Component\Translation\MessageCatalogue;
6
7 class FooExtractor implements ExtractorInterface
8 {
9     protected $prefix;
10
11     /**
12      * Extracts translation messages from a template directory to the catalogue.
13      */
14     public function extract($directory, MessageCatalogue $catalogue)
15     {
16         // ...
17     }
18
19     /**
20      * Sets the prefix that should be used for new found messages.
21      */
22     public function setPrefix($prefix)
23     {
24         $this->prefix = $prefix;
25     }
26 }
```

```
Listing 104-16 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*¹⁵
- *IcuResFileDumper*¹⁶
- *IniFileDumper*¹⁷
- *MoFileDumper*¹⁸
- *PoFileDumper*¹⁹
- *QtFileDumper*²⁰
- *XliffFileDumper*²¹
- *YamlFileDumper*²²

14. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Extractor/ExtractorInterface.php>
15. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/CsvFileDumper.php>
16. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/IcuResFileDumper.php>
17. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/IniFileDumper.php>
18. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/MoFileDumper.php>
19. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/PoFileDumper.php>
20. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/QtFileDumper.php>
21. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/XliffFileDumper.php>
22. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/YamlFileDumper.php>

You can create your own dumper by extending *FileDumper*²³ or implementing *DumperInterface*²⁴ 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 104-17 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:

```
Listing 104-18 1 services:
                2     App\Twig\AppExtension:
                3         tags: [twig.extension]
                4
                5     # optionally you can define the priority of the extension (default = 0).
                6     # Extensions with higher priorities are registered earlier. This is mostly
                7     # useful to register late extensions that override other extensions.
                8     App\Twig\AnotherExtension:
                9         tags: [{ name: twig.extension, priority: -100 }]
```

For information on how to create the actual Twig Extension class, see *Twig's documentation*²⁵ 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*²⁶ 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 104-19 1 services:
                2     Twig\Extensions\IntlExtension:
                3         tags: [twig.extension]
```

twig.loader

Purpose: Register a custom service that loads Twig templates

By default, Symfony uses only one *Twig Loader*²⁷ - *FilesystemLoader*²⁸. 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:

23. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/FileDumper.php>
24. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Translation/Dumper/DumperInterface.php>
25. <https://twig.symfony.com/doc/2.x/advanced.html#creating-an-extension>
26. <https://github.com/fabpot/Twig-extensions>
27. <https://twig.symfony.com/doc/2.x/api.html#loaders>
28. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/TwigBundle/Loader/FilesystemLoader.php>

Listing 104-20

```
1 services:
2     App\Twig\CustomLoader:
3         tags:
4             - { name: twig.loader, priority: 0 }
```



The **priority** is optional and its value is a positive or negative integer that defaults to **0**. Loaders with higher numbers are tried first.

twig.runtime

Purpose: To register a custom Lazy-Loaded Twig Extension

Lazy-Loaded Twig Extensions are defined as regular services but the need to be tagged with **twig.runtime**. If you're using the default services.yaml configuration, the service is auto-registered and auto-tagged. But, you can also register it manually:

Listing 104-21

```
1 services:
2     App\Twig\AppExtension:
3         tags: [twig.runtime]
```

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-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*²⁹ interface. Then, tag it with the **validator.initializer** tag (it has no options).

For an example, see the **DoctrineInitializer** class inside the Doctrine Bridge.

29. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Validator/ObjectInitializerInterface.php>



Chapter 105

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: *RequestEvent*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 105-1 1 \$ php bin/console debug:event-dispatcher kernel.request

-
1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/KernelEvent.php>
 2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/KernelEvent.php>
 3. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/KernelEvent.php>
 4. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/KernelEvent.php>
 5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/RequestEvent.php>

kernel.controller

Event Class: *ControllerEvent*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 105-2 1 use Symfony\Component\HttpKernel\Event\ControllerEvent;
2
3 public function onKernelController(ControllerEvent $event)
4 {
5     // ...
6
7     // the controller can be changed to any PHP callable
8     $event->setController($myCustomController);
9 }
```

Read more on the kernel.controller event.

Execute this command to find out which listeners are registered for this event and their priorities:

```
Listing 105-3 1 $ php bin/console debug:event-dispatcher kernel.controller
```

kernel.controller_arguments

Event Class: *ControllerArgumentsEvent*⁸

This event is dispatched just before a controller is called. It's useful to configure the arguments that are going to be passed to the controller. Typically, this is used to map URL routing parameters to their corresponding named arguments; or pass the current request when the **Request** type-hint is found:

```
Listing 105-4 1 use Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent;
2
3 public function onKernelControllerArguments(ControllerArgumentsEvent $event)
4 {
5     // ...
6
7     // get controller and request arguments
8     $namedArguments = $event->getRequest()->attributes->all();
9     $controllerArguments = $event->getArguments();
10
11     // set the controller arguments to modify the original arguments or add new ones
12     $event->setArguments($newArguments);
13 }
```

Execute this command to find out which listeners are registered for this event and their priorities:

```
Listing 105-5 1 $ php bin/console debug:event-dispatcher kernel.controller_arguments
```

kernel.view

Event Class: *ViewEvent*⁹

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/ControllerEvent.php>

7. <https://symfony.com/doc/master/bundles/SensioFrameworkExtraBundle/annotations/converters.html>

8. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/ControllerArgumentsEvent.php>

9. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/ViewEvent.php>

This event is dispatched after the controller has been executed but *only* if the controller does *not* return a **Response**¹⁰ 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 105-6 1 use Symfony\Component\HttpFoundation\Response;
2 use Symfony\Component\HttpKernel\Event\ViewEvent;
3
4 public function onKernelView(ViewEvent $event)
5 {
6     $value = $event->getControllerResult();
7     $response = new Response();
8
9     // ... somehow customize the Response from the return value
10
11     $event->setResponse($response);
12 }
```

Read more on the `kernel.view` event.

Execute this command to find out which listeners are registered for this event and their priorities:

```
Listing 105-7 1 $ php bin/console debug:event-dispatcher kernel.view
```

kernel.response

Event Class: *ResponseEvent*¹¹

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 105-8 1 use Symfony\Component\HttpKernel\Event\ResponseEvent;
2
3 public function onKernelResponse(ResponseEvent $event)
4 {
5     $response = $event->getResponse();
6
7     // ... modify the response object
8 }
```

Read more on the `kernel.response` event.

Execute this command to find out which listeners are registered for this event and their priorities:

```
Listing 105-9 1 $ php bin/console debug:event-dispatcher kernel.response
```

kernel.finish_request

Event Class: *FinishRequestEvent*¹²

This event is dispatched after the **kernel.response** event. 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 105-10

-
- 10. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Response.php>
 - 11. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/ResponseEvent.php>
 - 12. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/FinishRequestEvent.php>

```

1 use Symfony\Component\HttpKernel\Event\FinishRequestEvent;
2
3 public function onKernelFinishRequest(FinishRequestEvent $event)
4 {
5     if (null === $parentRequest = $this->requestStack->getParentRequest()) {
6         return;
7     }
8
9     // reset the locale of the subrequest to the locale of the parent request
10    $this->setLocale($parentRequest);
11 }

```

Execute this command to find out which listeners are registered for this event and their priorities:

Listing 105-11 `$ php bin/console debug:event-dispatcher kernel.finish_request`

kernel.terminate

Event Class: *TerminateEvent*¹³

This event is dispatched after the response has been sent (after the execution of the *handle()*¹⁴ 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 105-12 `$ php bin/console debug:event-dispatcher kernel.terminate`

kernel.exception

Event Class: *ExceptionEvent*¹⁵

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 105-13

```

1 use Symfony\Component\HttpFoundation\Response;
2 use Symfony\Component\HttpKernel\Event\ExceptionEvent;
3
4 public function onKernelException(ExceptionEvent $event)
5 {
6     $exception = $event->getException();
7     $response = new Response();
8     // setup the Response object based on the caught exception
9     $event->setResponse($response);
10
11    // you can alternatively set a new Exception
12    // $exception = new \Exception('Some special exception');
13    // $event->setException($exception);
14 }

```



The TwigBundle registers an *ExceptionListener*¹⁶ that forwards the Request to a given controller defined by the `exception_listener.controller` parameter.

13. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/TerminateEvent.php>

14. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/HttpKernel.php>

15. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Event/ExceptionEvent.php>

Symfony uses the following logic to determine the HTTP status code of the response:

- If `isClientError()`¹⁷, `isServerError()`¹⁸ or `isRedirect()`¹⁹ is true, then the status code on your `Response` object is used;
- If the original exception implements `HttpExceptionInterface`²⁰, 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 `ExceptionEvent::allowCustomResponseCode()` first and then set the status code on the response:

Listing 105-14

```
$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 105-15 1 $ php bin/console debug:event-dispatcher kernel.exception
```

16. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/EventListener/ExceptionListener.php>

17. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Response.php>

18. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Response.php>

19. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Response.php>

20. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.php>

