Symfony

The Reference Book

Version: 2.3

generated on May 4, 2015

The Reference Book (2.3)

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

Contents at a Glance

Chapter 1
FrameworkBundle Configuration ("framework")

This reference document is a work in progress. It should be accurate, but all options are not yet fully covered.

The FrameworkBundle contains most of the "base" framework functionality and can be configured under the `framework` key in your application configuration. This includes settings related to sessions, translation, forms, validation, routing and more.

Configuration

- secret
- http_method_override
- ide
- test
- default_locale
- trusted_proxies
- **form**
    - *enabled*
- **csrf_protection**
    - *enabled*
    - field_name
- **session**
    - name
    - cookie_lifetime
    - cookie_path
    - cookie_domain

        - cookie_secure
        - cookie_httponly
        - gc_divisor
        - gc_probability
        - gc_maxlifetime
        - save_path

- **serializer**
        - *enabled*

- **templating**
        - assets_base_urls
        - assets_version
        - assets_version_format

- **profiler**
        - collect
        - *enabled*

- **translator**
        - *enabled*
        - fallbacks

- **validation**
        - *enabled*
        - cache
        - enable_annotations
        - translation_domain

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 generating the *CSRF tokens*, 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

New in version 2.3: The `http_method_override` option was introduced in Symfony 2.3.

type: `Boolean` **default**: `true`

This determines whether the `_method` request parameter is used as the intended HTTP method on POST requests. If enabled, the *`Request::enableHttpMethodParameterOverride`*[1] method gets called

automatically. It becomes the service container parameter named `kernel.http_method_override`. For more information, see *How to Use HTTP Methods beyond GET and POST in Routes*.

If you're using the *AppCache 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 1-1
```php
// web/app.php

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

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

ide

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

If you're using an IDE like TextMate or Mac Vim, then Symfony can turn all of the file paths in an exception message into a link, which will open that file in your IDE.

Symfony contains preconfigured urls for some popular IDEs, you can set them using the following keys:

- `textmate`
- `macvim`
- `emacs`
- `sublime`

New in version 2.3.14: The `emacs` and `sublime` editors were introduced in Symfony 2.3.14.

You can also specify a custom url string. If you do this, all percentage signs (`%`) must be doubled to escape that character. For example, if you have installed *PhpStormOpener*[2] and use PHPstorm, you will do something like:

Listing 1-2
```yaml
# app/config/config.yml
framework:
    ide: "pstorm://%%f:%%l"
```

Of course, since every developer uses a different IDE, it's better to set this on a system level. This can be done by setting the `xdebug.file_link_format` in the `php.ini` configuration to the url string. If this configuration value is set, then the `ide` option will be ignored.

test

type: `Boolean`

If this configuration parameter 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 `app/config/config_test.yml`). For more information, see *Testing*.

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#enableHttpMethodParameterOverride()
2. https://github.com/pinepain/PhpStormOpener

default_locale

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

The default locale is used if no `_locale` routing parameter has been set. It becomes the service container parameter named `kernel.default_locale` and it is also available with the *`Request::getDefaultLocale`*[3] method.

trusted_proxies

type: `array`

Configures the IP addresses that should be trusted as proxies. For more details, see *How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy*.

New in version 2.3: CIDR notation support was introduced in Symfony 2.3, so you can whitelist whole subnets (e.g. `10.0.0.0/8`, `fc00::/7`).

Listing 1-3
```yaml
# app/config/config.yml
framework:
    trusted_proxies:  [192.0.0.1, 10.0.0.0/8]
```

form

enabled

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

Whether or not to enable support for the Form component.

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.

If this is activated, the *validation system* is also enabled automatically.

csrf_protection

enabled

type: `boolean` **default**: `true` if form support is enabled, `false` otherwise

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`.

field_name

type: `string` **default**: `"_token"`

The name of the hidden field used to render the *CSRF token*.

3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getDefaultLocale()

session

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. It will use `null` by default, which means `session.cookie_lifetime` value from `php.ini` will be used. Setting this value to `0` means the cookie is valid for the length of the browser session.

cookie_path

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

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

cookie_domain

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

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

cookie_secure

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

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

cookie_httponly

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

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_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_divisor

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

See gc_probability.

gc_maxlifetime

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

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

save_path

type: `string` **default**: `%kernel.cache.dir%/sessions`

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

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

Listing 1-4
```yaml
# app/config/config.yml
framework:
    session:
        save_path: null
```

serializer

enabled

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

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

For more details, see *How to Use the Serializer*.

templating

assets_base_urls

default: `{ http: [], ssl: [] }`

This option allows you to define base URLs to be used for assets referenced from `http` and `ssl` (`https`) pages. A string value may be provided in lieu of a single-element array. If multiple base URLs are provided, Symfony will select one from the collection each time it generates an asset's path.

For your convenience, `assets_base_urls` can be set directly with a string or array of strings, which will be automatically organized into collections of base URLs for `http` and `https` requests. If a URL starts with `https://` or is *protocol-relative*[4] (i.e. starts with `//`) it will be added to both collections. URLs starting with `http://` will only be added to the `http` collection.

assets_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 1-5
```twig
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />
```

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

4. http://tools.ietf.org/html/rfc3986#section-4.2

Listing 1-6
```yaml
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'], assets_version: v2 }
```

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

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

assets_version_format

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

This specifies a *sprintf*[5] pattern that will be used with the assets_version option to construct an asset's path. By default, the pattern adds the asset's version as a query string. For example, if `assets_version_format` is set to `%%s?version=%%s` and `assets_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, `assets_version_format` is not limited to producing versioned query strings.

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

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

profiler

enabled

New in version 2.2: The `enabled` option was introduced in Symfony 2.2. Prior to Symfony 2.2, the profiler could only be disabled by omitting the `framework.profiler` configuration entirely.

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

The profiler can be enabled by setting this key to `true`. When you are using the Symfony Standard Edition, the profiler is enabled in the `dev` and `test` environments.

5. http://php.net/manual/en/function.sprintf.php

collect

New in version 2.3: The `collect` option was introduced in Symfony 2.3. Previously, when `profiler.enabled` was `false`, the profiler *was* actually enabled, but the collectors were disabled. Now, the profiler and the collectors can be controlled independently.

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 by hand:

Listing 1-7
```
1 $profiler->enable();
```

translator

enabled

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

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

fallbacks

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

New in version 2.3.25: The `fallbacks` option was introduced in Symfony 2.3.25. Prior to Symfony 2.3.25, it was called `fallback` and only allowed one fallback language defined as a string. Please note that you can still use the old `fallback` option if you want define only one fallback.

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

For more details, see *Translations*.

validation

enabled

type: `boolean` **default**: `true` if *form support is enabled*, `false` otherwise

Whether or not to enable validation support.

cache

type: `string`

This value is used to determine the service that is used to persist class metadata in a cache. The actual service name is built by prefixing the configured value with `validator.mapping.cache.` (e.g. if the value is `apc`, the `validator.mapping.cache.apc` service will be injected). The service has to implement the *`CacheInterface`*[6].

enable_annotations

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

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

6. http://api.symfony.com/2.3/Symfony/Component/Validator/Mapping/Cache/CacheInterface.html

translation_domain

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

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

Full default Configuration

Listing 1-8

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

    # form configuration
    form:
        enabled:              false
    csrf_protection:
        enabled:              false
        field_name:           _token

    # esi configuration
    esi:
        enabled:              false

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

    # profiler configuration
    profiler:
        enabled:              false
        collect:              true
        only_exceptions:      false
        only_master_requests: false
        dsn:                  file:%kernel.cache_dir%/profiler
        username:
        password:
        lifetime:             86400
        matcher:
            ip:                   ~

            # use the urldecoded format
            path:                 ~ # Example: ^/path to resource/
            service:              ~

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

        # set to true to throw an exception when a parameter does not match the
requirements
```

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

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

    # serializer configuration
    serializer:
       enabled: false

    # templating configuration
    templating:
        assets_version:       ~
        assets_version_format:  "%%s?%%s"
        hinclude_default_template:  ~
        form:
            resources:

                # Default:
                - FrameworkBundle:Form
        assets_base_urls:
            http:                 []
            ssl:                  []
        cache:                ~
        engines:              # Required

            # Example:
            - twig
        loaders:              []
        packages:

            # Prototype
            name:
                version:              ~
                version_format:       "%%s?%%s"
                base_urls:
                    http:                 []
                    ssl:                  []

    # translator configuration
    translator:
        enabled:              false
        fallbacks:            [en]
```

```yaml

    # validation configuration
    validation:
        enabled:              false
        cache:                ~
        enable_annotations:   false
        translation_domain:   validators

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

Chapter 2

DoctrineBundle Configuration ("doctrine")

Full Default Configuration

Listing 2-1

```yaml
doctrine:
    dbal:
        default_connection:   default
        types:
            # A collection of custom types
            # Example
            some_custom_type:
                class:               Acme\HelloBundle\MyCustomType
                commented:           true
        # If enabled all tables not prefixed with sf2_ will be ignored by the schema
        # tool. This is for custom tables which should not be altered automatically.
        #schema_filter:       ^sf2_

        connections:
            # A collection of different named connections (e.g. default, conn2, etc)
            default:
                dbname:              ~
                host:                localhost
                port:                ~
                user:                root
                password:            ~
                charset:             ~
                path:                ~
                memory:              ~

                # The unix socket to use for MySQL
                unix_socket:         ~

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

```

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

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

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

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

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

                # the version of your database engine
                server_version:       ~

                # when true, queries are logged to a "doctrine" monolog channel
                logging:              "%kernel.debug%"
                profiling:            "%kernel.debug%"
                driver_class:         ~
                wrapper_class:        ~
                options:
                    # an array of options
                    key:                  []
                mapping_types:
                    # an array of mapping types
                    name:                 []
                slaves:

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

                        # The unix socket to use for MySQL
                        unix_socket:          ~

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

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

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

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

```
91
92                      # True to use a pooled server with the oci8 driver
93                      pooled:               ~
94
95                      # the version of your database engine
96                      server_version:       ~
97
98                      # Configuring MultipleActiveResultSets for the pdo_sqlsrv driver
99                      MultipleActiveResultSets:  ~
100
101     orm:
102         default_entity_manager:  ~
103         auto_generate_proxy_classes:  false
104         proxy_dir:            "%kernel.cache_dir%/doctrine/orm/Proxies"
105         proxy_namespace:      Proxies
106         # search for the "ResolveTargetEntityListener" class for a cookbook about this
107         resolve_target_entities: []
108         entity_managers:
109             # A collection of different named entity managers (e.g. some_em, another_em)
110             some_em:
111                 query_cache_driver:
112                     type:                 array # Required
113                     host:                 ~
114                     port:                 ~
115                     instance_class:       ~
116                     class:                ~
117                 metadata_cache_driver:
118                     type:                 array # Required
119                     host:                 ~
120                     port:                 ~
121                     instance_class:       ~
122                     class:                ~
123                 result_cache_driver:
124                     type:                 array # Required
125                     host:                 ~
126                     port:                 ~
127                     instance_class:       ~
128                     class:                ~
129                 connection:           ~
130                 class_metadata_factory_name:  Doctrine\ORM\Mapping\ClassMetadataFactory
131                 default_repository_class:  Doctrine\ORM\EntityRepository
132                 auto_mapping:         false
133                 hydrators:
134
135                     # An array of hydrator names
136                     hydrator_name:                []
137                 mappings:
138                     # An array of mappings, which may be a bundle name or something else
139                     mapping_name:
140                         mapping:              true
141                         type:                 ~
142                         dir:                  ~
143                         alias:                ~
144                         prefix:               ~
145                         is_bundle:            ~
146                 dql:
147                     # a collection of string functions
148                     string_functions:
149                         # example
```

```
                    # test_string: Acme\HelloBundle\DQL\StringFunction

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

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

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

Configuration Overview

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

Listing 2-2

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

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

Caching Drivers

For the caching drivers you can specify the values `array`, `apc`, `memcache`, `memcached`, `redis`, `wincache`, `zenddata`, `xcache` or `service`.

The following example shows an overview of the caching configurations:

Listing 2-3

```
doctrine:
    orm:
        auto_mapping: true
        metadata_cache_driver: apc
        query_cache_driver:
            type: service
            id: my_doctrine_common_cache_service
        result_cache_driver:
            type: memcache
            host: localhost
```

```
            port: 11211
            instance_class: Memcache
```

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`, `xml`, `yml`, `php` or `staticphp`. This specifies which type of metadata type your mapping uses.

dir

Path to the mapping or entity files (depending on the driver). If this path is relative it is assumed to be relative to the bundle root. This only works if the name of your mapping is a bundle name. If you want to use this option to specify absolute paths you should prefix the path with the kernel parameters that exist in the DIC (for example `%kernel.root_dir%`).

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. This option defaults to the bundle namespace + `Entity`, for example for an application bundle called AcmeHelloBundle prefix would be `Acme\HelloBundle\Entity`.

alias

Doctrine offers a way to alias entity namespaces to simpler, shorter names to be used in DQL queries or for Repository access. When using a bundle the alias defaults to the bundle name.

is_bundle

This option is a derived value from `dir` and by default is set to `true` if dir is relative proved by a `file_exists()` check that returns `false`. It is `false` if the existence check returns `true`. In this case an absolute path was specified and the metadata files are most likely in a directory outside of a bundle.

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 2-4

```
doctrine:
    dbal:
        dbname:               database
        host:                 localhost
        port:                 1234
        user:                 user
```

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

```
        password:             secret
        driver:               pdo_mysql
        # the DBAL driverClass option
        driver_class:         MyNamespace\MyDriverImpl
        # the DBAL driverOptions option
        options:
            foo: bar
        path:                 "%kernel.data_dir%/data.sqlite"
        memory:               true
        unix_socket:          /tmp/mysql.sock
        # the DBAL wrapperClass option
        wrapper_class:        MyDoctrineDbalConnectionWrapper
        charset:              UTF8
        logging:              "%kernel.debug%"
        platform_service:     MyOwnDatabasePlatformService
        server_version:       5.6
        mapping_types:
            enum: string
        types:
            custom: Acme\HelloBundle\MyCustomType
        # the DBAL keepSlave option
        keep_slave:           false
```

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 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 2-5

```
doctrine:
    dbal:
        default_connection:       default
        connections:
            default:
                dbname:           Symfony
                user:             root
                password:         null
                host:             localhost
                server_version:   5.6
            customer:
                dbname:           customer
                user:             root
                password:         null
                host:             localhost
                server_version:   5.7
```

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

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

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 2-6

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

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

Chapter 3
SecurityBundle Configuration ("security")

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

Full default Configuration

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

Listing 3-1

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

    # strategy can be: none, migrate, invalidate
    session_fixation_strategy:  migrate
    hide_user_not_found:  true
    always_authenticate_before_granting:  false
    erase_credentials:    true
    access_decision_manager:
        strategy:             affirmative
        allow_if_all_abstain:  false
        allow_if_equal_granted_denied:  true
    acl:

        # any name configured in doctrine.dbal section
        connection:           ~
        cache:
            id:                   ~
            prefix:               sf2_acl_
        provider:             ~
        tables:
            class:                acl_classes
            entry:                acl_entries
            object_identity:      acl_object_identities
            object_identity_ancestors:  acl_object_identity_ancestors
```

```
            security_identity:    acl_security_identities
        voter:
            allow_if_object_identity_unavailable:  true

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

        # PBKDF2 encoder
        # see the note about PBKDF2 below for details on security and speed
        Acme\Your\Class\Name:
            algorithm:           pbkdf2
            hash_algorithm:      sha512
            encode_as_base64:    true
            iterations:          1000
            key_length:          40

        # Example options/values for what a custom encoder might look like
        Acme\DemoBundle\Entity\User3:
            id:                  my.encoder.id

        # BCrypt encoder
        # see the note about bcrypt below for details on specific dependencies
        Acme\DemoBundle\Entity\User4:
            algorithm:           bcrypt
            cost:                13

        # Plaintext encoder
        # it does not do any encoding
        Acme\DemoBundle\Entity\User5:
            algorithm:           plaintext
            ignore_case:         false

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

        my_entity_provider:
            entity:
                class:             SecurityBundle:User
                property:          username
                # name of a non-default entity manager
                manager_name:      ~

        # Example custom provider
        my_some_custom_provider:
            id:                  ~
```

```yaml

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

    firewalls:            # Required
        # Examples:
        somename:
            pattern: .*
            request_matcher: some.service.id
            access_denied_url: /foo/error403
            access_denied_handler: some.service.id
            entry_point: some.service.id
            provider: some_key_from_above
            # manages where each firewall stores session information
            # See "Firewall Context" below for more details
            context: context_key
            stateless: false
            x509:
                provider: some_key_from_above
            http_basic:
                provider: some_key_from_above
            http_digest:
                provider: some_key_from_above
            form_login:
                # submit the login form here
                check_path: /login_check

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

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

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

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

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

                # csrf token options
                csrf_parameter: _csrf_token
                intention:      authenticate
                csrf_provider:  my.csrf_provider.id

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

```yaml
                remember_me:    false

                # by default, a session must exist before submitting an authentication
request
                # if false, then Request::hasPreviousSession is not called during
authentication
                # new in Symfony 2.3
                require_previous_session: true

            remember_me:
                token_provider: name
                key: someS3cretKey
                name: NameOfTheCookie
                lifetime: 3600 # in seconds
                path: /foo
                domain: somedomain.foo
                secure: false
                httponly: true
                always_remember_me: false
                remember_me_parameter: _remember_me
            logout:
                path:   /logout
                target: /
                invalidate_session: false
                delete_cookies:
                    a: { path: null, domain: null }
                    b: { path: null, domain: null }
                handlers: [some.service.id, another.service.id]
                success_handler: some.service.id
            anonymous: ~

        # Default values and options for any firewall
        some_firewall_listener:
            pattern:              ~
            security:             true
            request_matcher:      ~
            access_denied_url:    ~
            access_denied_handler:  ~
            entry_point:          ~
            provider:             ~
            stateless:            false
            context:              ~
            logout:
                csrf_parameter:       _csrf_token
                csrf_provider:        ~
                intention:            logout
                path:                 /logout
                target:               /
                success_handler:      ~
                invalidate_session:   true
                delete_cookies:

                    # Prototype
                    name:
                        path:                 ~
                        domain:               ~
                handlers:             []
            anonymous:
                key:                  4f954a0667e01
```

```
            switch_user:
                provider:            ~
                parameter:           _switch_user
                role:                ROLE_ALLOWED_TO_SWITCH

    access_control:
        requires_channel:     ~

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

Form Login Configuration

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

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

The Login Form and Process

login_path

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

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

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

check_path

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

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

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

use_forward

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

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

username_parameter

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

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

password_parameter

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

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

post_only

type: `Boolean` **default**: `true`

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

Redirecting after Login

- `always_use_default_target_path` (type: `Boolean`, default: `false`)
- `default_target_path` (type: `string`, default: `/`)
- `target_path_parameter` (type: `string`, default: `_target_path`)
- `use_referer` (type: `Boolean`, default: `false`)

Using the PBKDF2 Encoder: Security and Speed

New in version 2.2: The PBKDF2 password encoder was introduced in Symfony 2.2.

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

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

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

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

Using the BCrypt Password Encoder

To use this encoder, you either need to use PHP Version 5.5 or install the *ircmaxell/password-compat*[2] library via Composer.

New in version 2.2: The BCrypt password encoder was introduced in Symfony 2.2.

Listing 3-2

```yaml
# app/config/security.yml
security:
    # ...

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

1. http://en.wikipedia.org/wiki/PBKDF2
2. https://packagist.org/packages/ircmaxell/password-compat

```
7             algorithm: bcrypt
8             cost:      15
```

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

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

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

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

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

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

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

HTTP-Digest Authentication

To use HTTP-Digest authentication you need to provide a realm and a key:

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

```
5               http_digest:
6                   key: "a_random_string"
7                   realm: "secure-api"
```

Chapter 4

AsseticBundle Configuration ("assetic")

Full Default Configuration

Listing 4-1

```yaml
assetic:
    debug:                "%kernel.debug%"
    use_controller:
        enabled:              "%kernel.debug%"
        profiler:             false
    read_from:            "%kernel.root_dir%/../web"
    write_to:             "%assetic.read_from%"
    java:                 /usr/bin/java
    node:                 /usr/bin/node
    ruby:                 /usr/bin/ruby
    sass:                 /usr/bin/sass
    # An key-value pair of any number of named elements
    variables:
        some_name:                []
    bundles:

        # Defaults (all currently registered bundles):
        - FrameworkBundle
        - SecurityBundle
        - TwigBundle
        - MonologBundle
        - SwiftmailerBundle
        - DoctrineBundle
        - AsseticBundle
        - ...
    assets:
        # An array of named assets (e.g. some_asset, some_other_asset)
        some_asset:
            inputs:               []
            filters:              []
            options:
```

```
                # A key-value array of options and values
                some_option_name: []
    filters:

        # An array of named filters (e.g. some_filter, some_other_filter)
        some_filter:                []
    twig:
        functions:
            # An array of named functions (e.g. some_function, some_other_function)
            some_function:                []
```

Chapter 5
SwiftmailerBundle Configuration ("swiftmailer")

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

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

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

Configuration

- transport
- username
- password
- host
- port
- encryption
- auth_mode
- **spool**
    - type
    - path
- sender_address
- **antiflood**
    - threshold
    - sleep
- delivery_address

- delivery_whitelist
- disable_delivery
- logging

transport

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

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

- smtp
- gmail (see *How to Use Gmail to Send Emails*)
- mail
- 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.

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.

encryption

type: `string`

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

auth_mode

type: `string`

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

spool

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

type

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

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

path

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

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

sender_address

type: `string`

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

antiflood

threshold

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

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

sleep

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

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

delivery_address

type: `string`

If set, all email messages will be sent to this address instead of being sent to their actual recipients. This is often useful when developing. For example, by setting this in the `config_dev.yml` file, you can guarantee that all emails sent during development go to a single account.

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

delivery_whitelist

type: `array`

Used in combination with `delivery_address`. If set, emails matching any of these patterns will be delivered like normal, instead of being sent to `delivery_address`. For details, see *the cookbook entry*.

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.

Full default Configuration

Listing 5-1
```yaml
swiftmailer:
    transport:            smtp
    username:             ~
    password:             ~
    host:                 localhost
    port:                 false
    encryption:           ~
    auth_mode:            ~
    spool:
        type:                 file
        path:                 "%kernel.cache_dir%/swiftmailer/spool"
    sender_address:       ~
    antiflood:
        threshold:            99
        sleep:                0
    delivery_address:     ~
    disable_delivery:     ~
    logging:              "%kernel.debug%"
```

Using multiple Mailers

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

Listing 5-2
```yaml
swiftmailer:
    default_mailer: second_mailer
    mailers:
        first_mailer:
            # ...
        second_mailer:
            # ...
```

Each mailer is registered as a service:

Listing 5-3
```php
// ...

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

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

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

Chapter 6
TwigBundle Configuration ("twig")

Listing 6-1

```yaml
twig:
    exception_controller:  twig.controller.exception:showAction
    form:
        resources:

            # Default:
            - form_div_layout.html.twig

            # Example:
            - MyBundle::form.html.twig
    globals:

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

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

    # The following were added in Symfony 2.3.
    # See http://twig.sensiolabs.org/doc/
recipes.html#using-the-template-name-to-set-the-default-escaping-strategy
    autoescape_service:       ~ # Example: @my_service
    autoescape_service_method: ~ # use in combination with autoescape_service option
    base_template_class:      ~ # Example: Twig_Template
    cache:                    "%kernel.cache_dir%/twig"
    charset:                  "%kernel.charset%"
    debug:                    "%kernel.debug%"
    strict_variables:         ~
    auto_reload:              ~
```

```
37      optimizations:             ~
38      paths:
            "%kernel.root_dir%/../vendor/acme/foo-bar/templates": foo_bar
```

Configuration

exception_controller

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

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

1. http://api.symfony.com/2.3/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html

Chapter 7
MonologBundle Configuration ("monolog")

Full Default Configuration

Listing 7-1

```yaml
monolog:
    handlers:

        # Examples:
        syslog:
            type:               stream
            path:               /var/log/symfony.log
            level:              ERROR
            bubble:             false
            formatter:          my_formatter
        main:
            type:               fingers_crossed
            action_level:       WARNING
            buffer_size:        30
            handler:            custom
        custom:
            type:               service
            id:                 my_handler

        # Default options and values for some "my_custom_handler"
        # Note: many of these options are specific to the "type".
        # For example, the "service" type doesn't use any options
        # except id and channels
        my_custom_handler:
            type:                ~ # Required
            id:                  ~
            priority:            0
            level:               DEBUG
            bubble:              true
            path:                "%kernel.logs_dir%/%kernel.environment%.log"
            ident:               false
```

```
            facility:             user
            max_files:            0
            action_level:         WARNING
            activation_strategy:  ~
            stop_buffering:       true
            buffer_size:          0
            handler:              ~
            members:              []
            channels:
                type:     ~
                elements: ~
            from_email:           ~
            to_email:             ~
            subject:              ~
            mailer:               ~
            email_prototype:
                id:                   ~ # Required (when the email_prototype is used)
                method:               ~
            formatter:            ~
```

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

Chapter 8
WebProfilerBundle Configuration ("web_profiler")

Full default Configuration

Listing 8-1

```yaml
web_profiler:

    # DEPRECATED, it is not useful anymore and can be removed safely from your
configuration
    verbose:              true

    # display the web debug toolbar at the bottom of pages with a summary of profiler info
    toolbar:              false
    position:             bottom

    # gives you the opportunity to look at the collected data before following the redirect
    intercept_redirects: false
```

Chapter 9

Configuring in the Kernel (e.g. AppKernel)

Some configuration can be done on the kernel class itself (usually called `app/AppKernel.php`). You can do this by overriding specific methods in the parent *`Kernel`*[1] class.

Configuration

- Charset
- Kernel Name
- Root Directory
- Cache Directory
- Log Directory

Charset

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

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

Listing 9-1

```php
// app/AppKernel.php

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

1. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html
2. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getCharset()

Kernel Name

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

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

The name of the kernel isn't usually directly important - it's used in the generation of cache files. If you have an application with multiple kernels, the easiest way to make each have a unique name is to duplicate the `app` directory and rename it to something else (e.g. `foo`).

Root Directory

type: `string` **default**: the directory of `AppKernel`

This returns the root directory of your kernel. If you use the Symfony Standard edition, the root directory refers to the `app` directory.

To change this setting, override the *`getRootDir()`*[4] method:

Listing 9-2
```php
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...

    public function getRootDir()
    {
        return realpath(parent::getRootDir().'/../');
    }
}
```

Cache Directory

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

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

Log Directory

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

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

3. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getName()
4. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getRootDir()
5. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getCacheDir()
6. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getLogDir()

Chapter 10

Form Types Reference

A form is composed of *fields*, each of which are built with the help of a field *type* (e.g. a `text` type, `choice` type, 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

- *text*
- *textarea*
- *email*
- *integer*
- *money*
- *number*
- *password*
- *percent*
- *search*
- *url*

Choice Fields

- *choice*
- *entity*
- *country*
- *language*
- *locale*
- *timezone*
- *currency*

Date and Time Fields

- *date*
- *datetime*
- *time*
- *birthday*

Other Fields

- *checkbox*
- *file*
- *radio*

Field Groups

- *collection*
- *repeated*

Hidden Fields

- *hidden*

Buttons

- *button*
- *reset*
- *submit*

Base Fields

- *form*

Chapter 11
text Field Type

The text field represents the most basic input text field.

Rendered as	`input text` field
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • max_length • read_only • required • trim
Parent type	*form*
Class	*TextType*[1]

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/TextType.html

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

Listing 11-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 11-2
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

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

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 11-4
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 11-5
```twig
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

type: `Boolean` **default**: `true`

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

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

Chapter 12
textarea Field Type

Renders a `textarea` HTML element.

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

Inherited Options

These options inherit from the *form* type:

attr

type: array **default**: Empty array

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/TextareaType.html

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

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

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 12-2
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 12-3
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

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

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 12-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

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

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

type: `Boolean` **default**: `true`

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

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

Chapter 13
email Field Type

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

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

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/EmailType.html

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

Listing 13-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 13-2
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

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

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 13-4
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 13-5
```twig
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

type: `Boolean` **default**: `true`

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

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

Chapter 14
integer Field Type

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 frontend functionality.

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

Rendered as	`input number` field
Options	• grouping • precision • rounding_mode
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*IntegerType*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/IntegerType.html

Field Options

grouping

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

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

precision

type: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `precision` 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**: `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` Rounding mode to round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Rounding mode to round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Rounding mode to round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Rounding mode to round towards positive infinity.

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 14-1
```
1 $builder->add('token', 'hidden', array(
2     'data' => 'abcdef',
3 ));
```

2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.html

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 14-2
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 14-3
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 14-4
```php
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

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

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

Listing 14-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 14-6
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 15
money Field Type

Renders an input text field and specializes in handling submitted "money" data.

This field type allows you to specify a currency, whose symbol is rendered next to the text field. There are also several other options for customizing how the input and output of the data is handled.

Rendered as	`input text` field
Options	• currency • divisor • grouping • precision
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*MoneyType*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/MoneyType.html

Field Options

currency

type: `string` **default**: `EUR`

Specifies the currency that the money is being specified in. This determines the currency symbol that should be shown by the text box. Depending on the currency - the currency symbol may be shown before or after the input text field.

This can be any *3 letter ISO 4217 code*[2]. You can also set this to false to hide the currency symbol.

divisor

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

If, for some reason, you need to divide your starting value by a number before rendering it to the user, you can use the `divisor` option. For example:

Listing 15-1
```
$builder->add('price', 'money', array(
    'divisor' => 100,
));
```

In this case, if the `price` field is set to `9900`, then the value `99` will actually be rendered to the user. When the user submits the value `99`, it will be multiplied by `100` and `9900` will ultimately be set back on your object.

grouping

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

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

precision

type: `integer` **default**: `2`

For some reason, if you need some precision 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 precision to `0`).

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

2. `http://en.wikipedia.org/wiki/ISO_4217`

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

Listing 15-2
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 15-3
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 15-4
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 15-5
```php
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
```

```
4  letters',
5      'invalid_message_parameters' => array('%num%' => 6),
   ));
```

label

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

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

Listing 15-6
```
1  {{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 15-7
```
1  {{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Form Variables

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

Chapter 16
number Field Type

Renders an input text field and specializes in handling number input. This type offers different options for the precision, rounding, and grouping that you want to use for your number.

Rendered as	`input text` field
Options	• grouping • precision • rounding_mode
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*NumberType*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/NumberType.html

Field Options

grouping

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

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

precision

type: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `precision` 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**: `IntegerToLocalizedStringTransformer::ROUND_HALFUP`

If a submitted number needs to be rounded (based on the `precision` option), you have several configurable options for that rounding. Each option is a constant on the *`IntegerToLocalizedStringTransformer`*[2]:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` Rounding mode to round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Rounding mode to round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Rounding mode to round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Rounding mode to round towards positive infinity.
- `IntegerToLocalizedStringTransformer::ROUND_HALFDOWN` Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
- `IntegerToLocalizedStringTransformer::ROUND_HALFEVEN` Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor.
- `IntegerToLocalizedStringTransformer::ROUND_HALFUP` Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.html

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

Listing 16-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 16-2
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 16-3
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 16-4
```php
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
```

```
4  letters',
5      'invalid_message_parameters' => array('%num%' => 6),
   ));
```

label

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

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

Listing 16-5
```
1  {{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 16-6
```
1  {{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 17
password Field Type

The `password` field renders an input password text box.

Rendered as	`input password` field
Options	• always_empty
Inherited options	• disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • max_length • read_only • required • trim
Parent type	*text*
Class	*PasswordType*[1]

Field Options

always_empty

type: `Boolean` **default**: `true`

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/PasswordType.html

If set to true, the field will *always* render blank, even if the corresponding field has a value. When set to false, the password field will be rendered with the `value` attribute set to its true value only upon submission.

Put simply, if for some reason you want to render your password field *with* the password value already entered into the box, set this to false and submit the form.

Inherited Options

These options inherit from the *form* type:

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 17-1
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 17-2
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 17-3
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 17-4
```twig
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

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

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. http://php.net/manual/en/function.trim.php

Chapter 18
percent Field Type

The `percent` type renders an input text field and specializes in handling percentage data. If your percentage data is stored as a decimal (e.g. `.95`), you can use this field out-of-the-box. If you store your data as a number (e.g. `95`), you should set the `type` option to `integer`.

This field adds a percentage sign "%" after the input box.

Rendered as	`input text` field
Options	• precision • type
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • invalid_message • invalid_message_parameters • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*PercentType*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/PercentType.html

Field Options

precision

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

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

type

type: `string` **default**: `fractional`

This controls how your data is stored on your object. For example, a percentage corresponding to "55%", might be stored as `.55` or `55` on your object. The two "types" handle these two cases:

- `fractional` If your data is stored as a decimal (e.g. `.55`), use this type. The data will be multiplied by `100` before being shown to the user (e.g. `55`). The submitted data will be divided by `100` on form submit so that the decimal value is stored (`.55`);
- `integer` If your data is stored as an integer (e.g. 55), then use this option. The raw value (`55`) is shown to the user and stored on your object. Note that this only works for integer values.

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 18-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 18-2
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 18-3
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;

- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 18-4
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label

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

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

Listing 18-5
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 18-6
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 19
search Field Type

This renders an `<input type="search" />` field, which is a text box with special functionality supported by some browsers.

Read about the input search field at *DiveIntoHTML5.info*[1]

Rendered as	`input search` field
Inherited options	• disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • max_length • read_only • required • trim
Parent type	*text*
Class	*SearchType*[2]

Inherited Options

These options inherit from the *form* type:

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

1. http://diveintohtml5.info/forms.html#type-search
2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/SearchType.html

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 19-1
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 19-2
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
```

```
7       ));
8   }
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 19-3
```
1  {{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 19-4
```
1  {{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

type: `Boolean` **default**: `true`

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

3. http://diveintohtml5.info/forms.html
4. http://php.net/manual/en/function.trim.php

Chapter 20
url Field Type

The `url` field is a text field that prepends the submitted value with a given protocol (e.g. `http://`) if the submitted value doesn't already have a protocol.

Rendered as	`input url` field
Options	• default_protocol
Inherited options	• data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • max_length • read_only • required • trim
Parent type	*text*
Class	*UrlType*[1]

Field Options

default_protocol

type: `string` **default**: `http`

1. `http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/UrlType.html`

If a value is submitted that doesn't begin with some protocol (e.g. `http://`, `ftp://`, etc), this protocol will be prepended to the string when the data is submitted to the form.

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 20-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 20-2
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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

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

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

Listing 20-3
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 20-4
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 20-5
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

trim

type: `Boolean` **default**: `true`

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

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

Chapter 21
choice Field Type

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* the `choice_list` or `choices` option.

Rendered as	can be various tags (see below)
Options	• choices • choice_list • empty_value • expanded • multiple • preferred_choices
Overridden options	• compound • empty_data • error_bubbling
Inherited options	• by_reference • data • disabled • error_mapping • inherit_data • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*`ChoiceType`*[1]

Example Usage

The easiest way to use this field is to specify the choices directly via the `choices` option. The key of the array becomes the value that's actually set on your underlying object (e.g. `m`), while the value is what the user sees on the form (e.g. `Male`).

Listing 21-1
```php
$builder->add('gender', 'choice', array(
    'choices'  => array('m' => 'Male', 'f' => 'Female'),
    'required' => false,
));
```

By setting `multiple` to true, you can allow the user to choose multiple values. The widget will be rendered as a multiple `select` tag or a series of checkboxes depending on the `expanded` option:

Listing 21-2
```php
$builder->add('availability', 'choice', array(
    'choices' => array(
        'morning'   => 'Morning',
        'afternoon' => 'Afternoon',
        'evening'   => 'Evening',
    ),
    'multiple' => true,
));
```

You can also use the `choice_list` option, which takes an object that can specify the choices for your widget.

Select Tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

Element Type	Expanded	Multiple
select tag	`false`	`false`
select tag (with `multiple` attribute)	`false`	`true`
radio buttons	`true`	`false`
checkboxes	`true`	`true`

Field Options

choices

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

This is the most basic way to specify the choices that should be used by this field. The `choices` option is an array, where the array key is the item value and the array value is the item's label:

Listing 21-3

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/ChoiceType.html

```php
$builder->add('gender', 'choice', array(
    'choices' => array('m' => 'Male', 'f' => 'Female'),
));
```

When the values to choose from are not integers or strings (but e.g. floats or booleans), you should use the choice_list option instead. With this option you are able to keep the original data format which is important to ensure that the user input is validated properly and useless database updates caused by a data type mismatch are avoided.

choice_list

type: *ChoiceListInterface*[2]

This is one way of specifying the options to be used for this field. The `choice_list` option must be an instance of the `ChoiceListInterface`. For more advanced cases, a custom class that implements the interface can be created to supply the choices.

With this option you can also allow float values to be selected as data.

Listing 21-4
```php
use Symfony\Component\Form\Extension\Core\ChoiceList\ChoiceList;

// ...
$builder->add('status', 'choice', array(
  'choice_list' => new ChoiceList(array(1, 0.5), array('Full', 'Half'))
));
```

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 21-5
```php
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 21-6
```php
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 21-7

2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/ChoiceList/ChoiceListInterface.html

```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 21-8
```php
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 21-9
```twig
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

Overridden Options

compound

type: `boolean` **default**: same value as `expanded` option

This option specifies if a form is compound. The value is by default overridden by the value of the `expanded` option.

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 21-10
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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).

Inherited Options

These options inherit from the *form* type:

by_reference

type: `Boolean` **default**: `true`

In most cases, if you have a `name` field, then you expect `setName()` to be called on the underlying object. In some cases, however, `setName()` may *not* be called. Setting `by_reference` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 21-11
```php
$builder = $this->createFormBuilder($article);
$builder
    ->add('title', 'text')
    ->add(
        $builder->create('author', 'form', array('by_reference' => ?))
            ->add('name', 'text')
            ->add('email', 'email')
    )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 21-12

```
$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 21-13
```
$article->setTitle('...');
$author = $article->getAuthor();
$author->setName('...');
$author->setEmail('...');
$article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *collection* form type 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 object (if there is one)

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

Listing 21-14
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 21-15
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 21-16
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 21-17
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

Field Variables

Variable	Type	Usage
multiple	`Boolean`	The value of the multiple option.
expanded	`Boolean`	The value of the expanded option.
preferred_choices	`array`	A nested array containing the `ChoiceView` objects of choices which should be presented to the user with priority.
choices	`array`	A nested array containing the `ChoiceView` objects of the remaining choices.
separator	`string`	The separator to use between choice groups.
empty_value	`mixed`	The empty value if not already in the list, otherwise `null`.
is_selected	`callable`	A callable which takes a `ChoiceView` and the selected value(s) and returns whether the choice is in the selected value(s).
empty_value_in_choices	`Boolean`	Whether the empty value is in the choice list.

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

It's significantly faster to use the *selectedchoice(selected_value)* test instead when using Twig.

Chapter 22
entity Field Type

A special `choice` 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	• class • data_class • em • group_by • property • query_builder
Overridden Options	• choices • choice_list
Inherited options	from the *choice* type: • empty_value • expanded • multiple • preferred_choices from the *form* type: • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr

	• mapped • read_only • required
Parent type	*choice*
Class	*EntityType*[1]

Basic Usage

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

Listing 22-1
```
$builder->add('users', 'entity', array(
    'class' => 'AcmeHelloBundle:User',
    'property' => 'username',
));
```

In this case, all `User` objects will be loaded from the database and rendered as either a `select` tag, a set or radio buttons or a series of checkboxes (this depends on the `multiple` and `expanded` values). If the entity object does not have a `__toString()` method the `property` option is needed.

Using a Custom Query for the Entities

If you need to specify 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. The easiest way to use the option is as follows:

Listing 22-2
```
use Doctrine\ORM\EntityRepository;
// ...

$builder->add('users', 'entity', array(
    'class' => 'AcmeHelloBundle:User',
    'query_builder' => function (EntityRepository $er) {
        return $er->createQueryBuilder('u')
            ->orderBy('u.username', 'ASC');
    },
));
```

Using Choices

If you already have the exact collection of entities that you want included in the choice element, you can simply 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 22-3
```
$builder->add('users', 'entity', array(
    'class' => 'AcmeHelloBundle:User',
    'choices' => $group->getUsers(),
));
```

1. http://api.symfony.com/2.3/Symfony/Bridge/Doctrine/Form/Type/EntityType.html

Select Tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

Element Type	Expanded	Multiple
select tag	`false`	`false`
select tag (with `multiple` attribute)	`false`	`true`
radio buttons	`true`	`false`
checkboxes	`true`	`true`

Field Options

class

type: `string` **required**

The class of your entity (e.g. `AcmeStoreBundle:Category`). This can be a fully-qualified class name (e.g. `Acme\StoreBundle\Entity\Category`) or the short alias name (as shown prior).

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 22-4

```
1 $builder->add('media', 'sonata_media_type', array(
2     'data_class' => 'Acme\DemoBundle\Entity\Media',
3 ));
```

em

type: `string` **default**: the default entity manager

If specified, the specified entity manager will be used to load the choices instead of the default entity manager.

group_by

type: `string`

This is a property path (e.g. `author.name`) used to organize the available choices in groups. It only works when rendered as a select tag and does so by adding `optgroup` elements around options. Choices that do not return a value for this property path are rendered directly under the select tag, without a surrounding optgroup.

property

type: `string`

This is the property that should be used for displaying the entities as text in the HTML element. If left blank, the entity object will be cast into a string and so must have a `__toString()` method.

The `property` option is the property path used to display the option. 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 22-5
```
$builder->add('gender', 'entity', array(
    'class' => 'MyBundle:Gender',
    'property' => 'translations[en].name',
));
```

query_builder

type: `Doctrine\ORM\QueryBuilder` or a Closure

If specified, this is used to query the subset of options (and their order) that should be used for the field. The value of this option can either be a `QueryBuilder` object or a Closure. If using a Closure, it should take a single argument, which is the `EntityRepository` of the entity.

Overridden Options

choice_list

default: *`EntityChoiceList`*[2]

The purpose of the `entity` type is to create and configure this `EntityChoiceList` for you, by using all of the above options. If you need to override this option, you may just consider using the *choice Field Type* directly.

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*.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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.

2. http://api.symfony.com/2.3/Symfony/Bridge/Doctrine/Form/ChoiceList/EntityChoiceList.html

- Add an empty value with "Choose an option" as the text:

Listing 22-6
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 22-7
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 22-8
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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.

If you are working with a collection of Doctrine entities, it will be helpful to read the documentation for the *collection Field Type* as well. In addition, there is a complete example in the cookbook article *How to Embed a Collection of Forms*.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 22-9
```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 22-10
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

This option expects an array of entity objects, unlike the `choice` field that requires an array of keys.

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 22-11
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 22-12
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
```

```
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 22-13
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 22-14
```
1 {{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 22-15
```
1 {{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 23
country Field Type

The `country` type 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 `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses all of the countries of the world. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select Tag, Checkboxes or Radio Buttons*)
Overridden Options	• choices
Inherited options	from the *choice* type • empty_value • error_bubbling • error_mapping • expanded • multiple • preferred_choices from the *form* type • data • disabled • empty_data

1. `http://php.net/manual/en/locale.getdefault.php`

	• label • label_attr • mapped • read_only • required
Parent type	*choice*
Class	*CountryType*[2]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getRegionBundle()->getCountryNames()`

The country type defaults the `choices` option to the whole list of countries. The locale is used to translate the countries names.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 23-1
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 23-2
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 23-3
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
```

2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/CountryType.html

```
    'required' => false,
));
```

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 23-4
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 23-5
```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 23-6
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 23-7
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 23-8
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

label_attr

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

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

Listing 23-10
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 24
language Field Type

The `language` type 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]

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of languages. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select Tag, Checkboxes or Radio Buttons*)
Overridden Options	• choices
Inherited options	from the *choice* type • empty_value • error_bubbling • error_mapping • expanded • multiple • preferred_choices from the *form* type • data

1. http://site.icu-project.org
2. http://php.net/manual/en/locale.getdefault.php

	• disabled • empty_data • label • label_attr • mapped • read_only • required
Parent type	*choice*
Class	*LanguageType*[3]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLanguageBundle()->getLanguageNames()`.

The choices option defaults to all languages. The default locale is used to translate the languages names.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 24-1
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 24-2
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 24-3

3. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/LanguageType.html

```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 24-4
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 24-5
```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 24-6
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 24-7
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 24-8
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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 24-9
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 24-10
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

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

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

4. `http://diveintohtml5.info/forms.html`

Chapter 25
locale Field Type

The `locale` type 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 `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of locales. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select Tag, Checkboxes or Radio Buttons*)
Overridden Options	• choices
Inherited options	from the *choice* type • empty_value • error_bubbling • error_mapping • expanded • multiple • preferred_choices from the *form* type

1. http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
2. http://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
3. http://php.net/manual/en/locale.getdefault.php

	• data • disabled • empty_data • label • label_attr • mapped • read_only • required
Parent type	*choice*
Class	*LocaleType*[4]

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLocaleBundle()->getLocaleNames()`

The choices option defaults to all locales. It uses the default locale to specify the language.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 25-1
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 25-2
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 25-3

4. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/LocaleType.html

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 25-4
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 25-5
```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 25-6
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 25-7
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 25-8
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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 25-9
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 25-10
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

If true, an *HTML5 required attribute*[5] will be rendered. The corresponding `label` will also render with a `required` class.

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

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

5. `http://diveintohtml5.info/forms.html`

Chapter 26
timezone Field Type

The `timezone` type 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 `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of timezones. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select Tag, Checkboxes or Radio Buttons*)
Overridden Options	• choices
Inherited options	from the *choice* type • empty_value • expanded • multiple • preferred_choices from the *form* type • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • read_only • required
Parent type	*choice*

Class	*TimezoneType*[1]

Overridden Options

choices

default: *TimezoneChoiceList*[2]

The Timezone type defaults the choices to all timezones returned by *DateTimeZone::listIdentifiers()*[3], broken down by continent.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 26-1
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 26-2
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 26-3
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

expanded

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

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/TimezoneType.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/ChoiceList/TimezoneChoiceList.html
3. http://php.net/manual/en/datetimezone.listidentifiers.php

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 26-4
```php
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 26-5
```twig
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 26-6
```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 26-7
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 26-8
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 26-9
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 26-10
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

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

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.

Chapter 27
currency Field Type

The `currency` type is a subset of the *choice type* that allows the user to select from a large list of *3-letter ISO 4217*[1] currencies.

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of currencies. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select Tag, Checkboxes or Radio Buttons*)
Overridden Options	• choices
Inherited options	from the *choice* type • empty_value • error_bubbling • expanded • multiple • preferred_choices from the *form* type • data • disabled • empty_data • label • label_attr • mapped • read_only • required
Parent type	*choice*
Class	*CurrencyType*[2]

1. `http://en.wikipedia.org/wiki/ISO_4217`

Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getCurrencyBundle()->getCurrencyNames()`

The choices option defaults to all currencies.

Inherited Options

These options inherit from the *choice* type:

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 27-1
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 27-2
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

Listing 27-3
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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`

2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/CurrencyType.html

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

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.

preferred_choices

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

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

Listing 27-4
```php
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is 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 27-5
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 27-6
```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `multiple` is `false` and `expanded` is `false`, then `''` (empty string);
- Otherwise `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 27-7
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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 27-8
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 27-9
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Chapter 28
date Field Type

A field that allows the user to modify date information via a variety of different HTML elements.

The underlying data used for this field type can be a `DateTime` object, a string, a timestamp or an array. As long as the input option is set correctly, the field will take care of all of the details.

The field can be rendered as a single text box, three text boxes (month, day, and year) or three select boxes (see the widget option).

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• days • empty_value • format • input • model_timezone • months • view_timezone • widget • years
Overridden Options	• by_reference • error_bubbling
Inherited options	• data • disabled • error_mapping • inherit_data • invalid_message • invalid_message_parameters • mapped

	• read_only
Parent type	*form*
Class	*DateType*[1]

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `publishedAt` field whose underlying date is a `DateTime` object. The following configures the `date` type for that field as three different choice fields:

Listing 28-1
```
$builder->add('publishedAt', 'date', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

The `input` option *must* be changed to match the type of the underlying date data. For example, if the `publishedAt` field's data were a unix timestamp, you'd need to set `input` to `timestamp`:

Listing 28-2
```
$builder->add('publishedAt', 'date', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

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 28-3
```
'days' => range(1,31)
```

empty_value

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 `empty_value` option can be used to add a "blank" entry to the top of each select box:

Listing 28-4
```
$builder->add('dueDate', 'date', array(
    'empty_value' => '',
));
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/DateType.html

Alternatively, you can specify a string to be displayed for the "blank" value:

Listing 28-5
```
$builder->add('dueDate', 'date', array(
    'empty_value' => array('year' => 'Year', 'month' => 'Month', 'day' => 'Day')
));
```

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 28-6
```
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    // this is actually the default format for single_text
    'format' => 'yyyy-MM-dd',
));
```

If you want your field to be rendered as an HTML5 "date" field, you have to use a `single_text` widget with the `yyyy-MM-dd` format (the *RFC 3339*[4] format) which is the default value if you use the `single_text` widget.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[5].

model_timezone

type: `string` **default**: system default timezone

2. http://www.php.net/manual/en/class.intldateformatter.php#intl.intldateformatter-constants
3. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax
4. http://tools.ietf.org/html/rfc3339
5. http://php.net/manual/en/function.date.php#refsect1-function.date-changelog

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`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[7].

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

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.

error_bubbling

default: `false`

Inherited Options

These options inherit from the *form* type:

6. `http://php.net/manual/en/timezones.php`
7. `http://php.net/manual/en/timezones.php`

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 28-7
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 28-8
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

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

- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 28-9
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

Field Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).
date_pattern	`string`	A string with the date format to use.

Chapter 29
datetime Field Type

This field type allows the user to modify data that represents a specific date and time (e.g. `1984-06-05 12:15:30`).

Can be rendered as a text input or select tags. The underlying format of the data can be a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• date_format • date_widget • days • empty_value • format • hours • input • minutes • model_timezone • months • seconds • time_widget • view_timezone • widget • with_minutes • with_seconds • years
Inherited options	• data • disabled • inherit_data • invalid_message

	• invalid_message_parameters • mapped • read_only
Parent type	*form*
Class	*DateTimeType*[1]

Field Options

date_format

type: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

Defines the `format` option that will be passed down to the date field. See the *date type's format option* for more details.

date_widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

days

type: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 29-1
```
'days' => range(1,31)
```

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/DateTimeType.html

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

Listing 29-3
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` 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
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

format

type: `string` **default**: `Symfony\Component\Form\Extension\Core\Type\DateTimeType::HTML5_FORMAT`

If the `widget` option is set to `single_text`, this option specifies the format of the input, i.e. how Symfony will interpret the given input as a datetime string. It defaults to the *RFC 3339*[2] format which is used by the HTML5 `datetime` field. Keeping the default value will cause the field to be rendered as an `input` field with `type="datetime"`.

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`.

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)
- `array` (e.g. `array(2011, 06, 05, 12, 15, 0)`)
- `timestamp` (e.g. `1307276100`)

The value that comes back from the form will also be normalized back into this format.

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[3].

minutes

type: `array` **default**: 0 to 59

2. http://tools.ietf.org/html/rfc3339
3. http://php.net/manual/en/function.date.php#refsect1-function.date-changelog

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*[4].

months

type: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

seconds

type: `array` **default**: 0 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

time_widget

type: `string` **default**: `choice`

Defines the `widget` option for the *time* type

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*[5].

widget

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

Defines the `widget` option for both the *date* type and *time* type. This can be overridden with the date_widget and time_widget options.

with_minutes

New in version 2.2: The `with_minutes` option was introduced in Symfony 2.2.

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.

---

4. `http://php.net/manual/en/timezones.php`
5. `http://php.net/manual/en/timezones.php`

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`.

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 29-5
```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

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

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 29-6
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

Field Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).

Chapter 30
time Field Type

A field to capture time input.

This can be rendered as a text field, a series of text fields (e.g. hour, minute, second) or a series of select fields. The underlying data can be stored as a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	can be various tags (see below)
Options	• empty_value • hours • input • minutes • model_timezone • seconds • view_timezone • widget • with_minutes • with_seconds
Overridden Options	• by_reference • error_bubbling
Inherited Options	• data • disabled • error_mapping • inherit_data • invalid_message • invalid_message_parameters • mapped • read_only

Parent type	form
Class	*TimeType*[1]

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `startTime` field whose underlying time data is a `DateTime` object. The following configures the `time` type for that field as two different choice fields:

Listing 30-1
```
$builder->add('startTime', 'time', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

The `input` option *must* be changed to match the type of the underlying date data. For example, if the `startTime` field's data were a unix timestamp, you'd need to set `input` to `timestamp`:

Listing 30-2
```
$builder->add('startTime', 'time', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 30-3
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 30-4
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/TimeType.html

Listing 30-5
```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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`.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `12:17:26`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('hour' => 12, 'minute' => 17, 'second' => 26)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

minutes

type: `array` **default**: 0 to 59

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`.

model_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[2].

seconds

type: `array` **default**: 0 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[3].

2. http://php.net/manual/en/timezones.php
3. http://php.net/manual/en/timezones.php

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders one, two (default) or three select inputs (hour, minute, second), depending on the with_minutes and with_seconds options.
- `text`: renders one, two (default) or three text inputs (hour, minute, second), depending on the with_minutes and with_seconds options.
- `single_text`: renders a single input of type `time`. User's input will be validated against the form `hh:mm` (or `hh:mm:ss` if using seconds).

Combining the widget type `single_text` and the with_minutes option set to `false` can cause unexpected behavior in the client as the input type `time` might not support selecting an hour only.

with_minutes

New in version 2.2: The `with_minutes` option was introduced in Symfony 2.2.

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.

error_bubbling

default: `false`

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 30-6
```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 30-7
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 30-8
```php
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

Form Variables

Variable	Type	Usage
widget	`mixed`	The value of the widget option.
with_minutes	`Boolean`	The value of the with_minutes option.
with_seconds	`Boolean`	The value of the with_seconds option.
type	`string`	Only present when widget is `single_text` and HTML5 is activated, contains the input type to use (`datetime`, `date` or `time`).

Chapter 31
birthday Field Type

A *date* 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 *date* type, but with a more appropriate default for the years option. The years option defaults to 120 years ago to the current year.

Underlying Data Type	can be `DateTime`, `string`, `timestamp`, or `array` (see the *input option*)
Rendered as	can be three select boxes or 1 or 3 text boxes, based on the widget option
Overridden options	• years
Inherited options	from the *date* type: • days • empty_value • format • input • model_timezone • months • view_timezone • widget from the *form* type: • data • disabled • inherit_data • invalid_message • invalid_message_parameters • mapped • read_only

Parent type	*date*
Class	*BirthdayType*[1]

Overridden Options

years

type: `array` **default**: 120 years ago to the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

Inherited Options

These options inherit from the *date* type:

days

type: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

Listing 31-1
```
'days' => range(1,31)
```

empty_value

New in version 2.3: Since Symfony 2.3, empty values are also supported if the `expanded` option is set to true.

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 31-2
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 31-3
```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/BirthdayType.html

Listing 31-4
```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

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 31-5
```php
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    // this is actually the default format for single_text
    'format' => 'yyyy-MM-dd',
));
```

If you want your field to be rendered as an HTML5 "date" field, you have to use a `single_text` widget with the `yyyy-MM-dd` format (the *RFC 3339*[4] format) which is the default value if you use the `single_text` widget.

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

If `timestamp` is used, `DateType` is limited to dates between Fri, 13 Dec 1901 20:45:54 GMT and Tue, 19 Jan 2038 03:14:07 GMT on 32bit systems. This is due to a *limitation in PHP itself*[5].

model_timezone

type: `string` **default**: system default timezone

2. http://www.php.net/manual/en/class.intldateformatter.php#intl.intldateformatter-constants
3. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax
4. http://tools.ietf.org/html/rfc3339
5. http://php.net/manual/en/function.date.php#refsect1-function.date-changelog

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`.

view_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[7].

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the format option.
- `text`: renders a three field input of type `text` (month, day, year).
- `single_text`: renders a single input of type `date`. User's input is validated based on the format option.

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 31-6
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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.

6. http://php.net/manual/en/timezones.php
7. http://php.net/manual/en/timezones.php

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 31-7
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

Chapter 32
checkbox Field Type

Creates a single input checkbox. This should always be used for a field that has a Boolean value: if the box is checked, the field will be set to true, if the box is unchecked, the value will be set to false.

Rendered as	`input checkbox` field
Options	• value
Overridden options	• compound • empty_data
Inherited options	• data • disabled • error_bubbling • error_mapping • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*CheckboxType*[1]

Example Usage

Listing 32-1

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/CheckboxType.html

```
1 $builder->add('public', 'checkbox', array(
2     'label'    => 'Show this entry publicly?',
3     'required' => false,
4 ));
```

Field Options

value

type: `mixed` **default**: `1`

The value that's actually used as the value for the checkbox or radio button. This does not affect the value that's set on your object.

To make a checkbox or radio button checked by default, use the data option.

Overridden Options

compound

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

This option specifies if a form is compound. As it's not the case for checkbox, by default the value is overridden with the `false` value.

empty_data

type: `string` **default**: `mixed`

This option determines what value the field will return when the `empty_value` choice is selected. In the checkbox and the radio type, the value of `empty_data` is overriden by the value returned by the data transformer (see *How to Use Data Transformers*).

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 32-2

```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 32-3
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

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

- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 32-4
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 32-5
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Form Variables

Variable	Type	Usage
checked	`Boolean`	Whether or not the current input is checked.

Chapter 33
file Field Type

The `file` type represents a file input in your form.

Rendered as	`input file` field
Inherited options	• disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • read_only • required
Parent type	*form*
Class	*FileType*[1]

Basic Usage

Say you have this form definition:

Listing 33-1
```
1  $builder->add('attachment', 'file');
```

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 33-2

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/FileType.html
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/File/UploadedFile.html

```php
use Symfony\Component\HttpFoundation\File\UploadedFile;

public function uploadAction()
{
    // ...

    if ($form->isValid()) {
        $someNewFilename = ...

        $form['attachment']->getData()->move($dir, $someNewFilename);

        // ...
    }

    // ...
}
```

The `move()` method takes a directory and a file name as its arguments. You might calculate the filename in one of the following ways:

Listing 33-3

```php
// use the original file name
$file->move($dir, $file->getClientOriginalName());

// compute a random name and try to guess the extension (more secure)
$extension = $file->guessExtension();
if (!$extension) {
    // extension cannot be guessed
    $extension = 'bin';
}
$file->move($dir, 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 the *cookbook* for an example of how to manage a file upload associated with a Doctrine entity.

Inherited Options

These options inherit from the *form* type:

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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 `null`.

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 33-4
```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 33-5
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

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

- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 33-6
```
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 33-7
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

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

Form Variables

Variable	Type	Usage
type	`string`	The type variable is set to `file`, in order to render as a file input field.

Chapter 34
radio Field Type

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 `radio` type isn't usually used directly. More commonly it's used internally by other types such as *choice*. If you want to have a Boolean field, use *checkbox*.

Rendered as	`input radio` field
Inherited options	from the *checkbox* type: • value from the *form* type: • data • disabled • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • read_only • required
Parent type	*checkbox*
Class	*`RadioType`*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/RadioType.html

Inherited Options

These options inherit from the *checkbox* type:

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 *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 34-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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 `empty_value` choice is selected. In the checkbox and the radio type, the value of `empty_data` is overriden by the value returned by the data transformer (see *How to Use Data Transformers*).

error_bubbling

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

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

error_mapping

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 34-2
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 34-3
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 34-4

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

mapped

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

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

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

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

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

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

Form Variables

Variable	Type	Usage
checked	`Boolean`	Whether or not the current input is checked.

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

Chapter 35
collection Field Type

This field type is used to render a "collection" of some field or form. In the easiest sense, it could be an array of `text` fields that populate an array `emails` field. 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 type option
Options	• allow_add • allow_delete • options • prototype • prototype_name • type
Inherited options	• by_reference • cascade_validation • empty_data • error_bubbling • error_mapping • label • label_attr • mapped • required
Parent type	*form*
Class	*CollectionType*[1]

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/CollectionType.html

If you are working with a collection of Doctrine entities, pay special attention to the allow_add, allow_delete and by_reference options. You can also see a complete example in the cookbook article *How to Embed a Collection of Forms*.

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 35-1
```
$builder->add('emails', 'collection', array(
    // each item in the array will be an "email" field
    'type'   => 'email',
    // these options are passed to each "email" type
    'options'  => array(
        'required'  => false,
        'attr'      => array('class' => 'email-box')
    ),
));
```

The simplest way to render this is all at once:

Listing 35-2
```
{{ form_row(form.emails) }}
```

A much more flexible method would look like this:

Listing 35-3
```
{{ form_label(form.emails) }}
{{ form_errors(form.emails) }}

<ul>
{% for emailField in form.emails %}
    <li>
        {{ form_errors(emailField) }}
        {{ form_widget(emailField) }}
    </li>
{% endfor %}
</ul>
```

In both cases, no input fields would render unless your `emails` data array already contained some emails.

In this simple example, it's still impossible to add new addresses or remove existing addresses. Adding new addresses is possible by using the allow_add option (and optionally the prototype option) (see example below). Removing emails from the `emails` array is possible with the allow_delete option.

Adding and Removing Items

If allow_add is set to `true`, then if any unrecognized items are submitted, they'll be added seamlessly to the array of items. This is great in theory, but takes a little bit more effort in practice to get the client-side JavaScript correct.

Following along with the previous example, suppose you start with two emails in the `emails` data array. In that case, two input fields will be rendered that will look something like this (depending on the name of your form):

Listing 35-4
```
<input type="email" id="form_emails_0" name="form[emails][0]" value="foo@foo.com" />
<input type="email" id="form_emails_1" name="form[emails][1]" value="bar@bar.com" />
```

To allow your user to add another email, just set allow_add to `true` and - via JavaScript - render another field with the name `form[emails][2]` (and so on for more and more fields).

To help make this easier, setting the prototype option to `true` allows you to render a "template" field, which you can then use in your JavaScript to help you dynamically create these new fields. A rendered prototype field will look like this:

Listing 35-5
```
<input type="email" id="form_emails___name__" name="form[emails][__name__]" value="" />
```

By replacing `__name__` with some unique value (e.g. `2`), you can build and insert new HTML fields into your form.

Using jQuery, a simple example might look like this. If you're rendering your collection fields all at once (e.g. `form_row(form.emails)`), then things are even easier because the `data-prototype` attribute is rendered automatically for you (with a slight difference - see note below) and all you need is the JavaScript:

Listing 35-6
```
{{ form_start(form) }}
    {# ... #}

    {# store the prototype on the data-prototype attribute #}
    <ul id="email-fields-list" data-prototype="{{
form_widget(form.emails.vars.prototype)|e }}">
    {% for emailField in form.emails %}
        <li>
            {{ form_errors(emailField) }}
            {{ form_widget(emailField) }}
        </li>
    {% endfor %}
    </ul>

    <a href="#" id="add-another-email">Add another email</a>

    {# ... #}
{{ form_end(form) }}

<script type="text/javascript">
    // keep track of how many email fields have been rendered
    var emailCount = '{{ form.emails|length }}';

    jQuery(document).ready(function() {
        jQuery('#add-another-email').click(function(e) {
            e.preventDefault();

            var emailList = jQuery('#email-fields-list');

            // grab the prototype template
            var newWidget = emailList.attr('data-prototype');
            // replace the "__name__" used in the id and name of the prototype
            // with a number that's unique to your emails
            // end name attribute looks like name="contact[emails][2]"
            newWidget = newWidget.replace(/__name__/g, emailCount);
            emailCount++;

            // create a new list element and add it to the list
```

```
            var newLi = jQuery('<li></li>').html(newWidget);
            newLi.appendTo(emailList);
        });
    })
</script>
```

If you're rendering the entire collection at once, then the prototype is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection. The only difference is that the entire "form row" is rendered for you, meaning you wouldn't have to wrap it in any container element as it was done above.

Field Options

allow_add

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

If set to `true`, then if unrecognized items are submitted to the collection, they will be added as new items. The ending array will contain the existing items as well as the new item that was in the submitted data. See the above example for more details.

The prototype option can be used to help render a prototype item that can be used - with JavaScript - to create new form items dynamically on the client side. For more information, see the above example and *Allowing "new" Tags with the "Prototype"*.

If you're embedding entire other forms to reflect a one-to-many database relationship, you may need to manually ensure that the foreign key of these new objects is set correctly. If you're using Doctrine, this won't happen automatically. See the above link for more details.

allow_delete

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

If set to `true`, then if an existing item is not contained in the submitted data, it will be correctly absent from the final array of items. This means that you can implement a "delete" button via JavaScript which simply removes a form element from the DOM. When the user submits the form, its absence from the submitted data will mean that it's removed from the final array.

For more information, see *Allowing Tags to be Removed*.

Be careful when using this option when you're embedding a collection of objects. In this case, if any embedded forms are removed, they *will* correctly be missing from the final array of objects. However, depending on your application logic, when one of those objects is removed, you may want to delete it or at least remove its foreign key reference to the main object. None of this is handled automatically. For more information, see *Allowing Tags to be Removed*.

options

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

This is the array that's passed to the form type specified in the type option. For example, if you used the *choice* type as your 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 35-7
```
$builder->add('favorite_cities', 'collection', array(
    'type'   => 'choice',
    'options'  => array(
        'choices'  => array(
            'nashville' => 'Nashville',
            'paris'     => 'Paris',
            'berlin'    => 'Berlin',
            'london'    => 'London',
        ),
    ),
));
```

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 35-8
```
{{ 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_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.

type

type: `string` or *`FormTypeInterface`*[2] **required**

2. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeInterface.html

This is the field type for each item in this collection (e.g. `text`, `choice`, etc). For example, if you have an array of email addresses, you'd use the *email* type. If you want to embed a collection of some other form, create a new instance of your form type and pass it as this option.

Inherited Options

These options inherit from the *form* type. Not all options are listed here - only the most applicable to this type:

by_reference

type: `Boolean` **default**: `true`

In most cases, if you have a `name` field, then you expect `setName()` to be called on the underlying object. In some cases, however, `setName()` may *not* be called. Setting `by_reference` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 35-9
```
$builder = $this->createFormBuilder($article);
$builder
    ->add('title', 'text')
    ->add(
        $builder->create('author', 'form', array('by_reference' => ?))
            ->add('name', 'text')
            ->add('email', 'email')
    )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 35-10
```
$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 35-11
```
$article->setTitle('...');
$author = $article->getAuthor();
$author->setName('...');
$author->setEmail('...');
$article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *collection* form type 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.

cascade_validation

type: Boolean **default**: false

Set this option to `true` to force validation on embedded form types. For example, if you have a `ProductType` with an embedded `CategoryType`, setting `cascade_validation` to `true` on `ProductType` will cause the data from `CategoryType` to also be validated.

Instead of using this option, it is recommended that you use the `Valid` constraint in your model to force validation on a child object stored on a property. This cascades only the validation but not the use of the `validation_group` option on child forms. You can read more about this in the section about *Embedding a Single Object*.

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`.

empty_data

type: `mixed`

The default value is `array()` (empty array).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 35-12
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 35-13
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 35-14
```twig
{{ form_label(form.name, 'Your name') }}
```

label_attr

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

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

Listing 35-15
```twig
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

required

type: `Boolean` **default**: `true`

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

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

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

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.

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

Chapter 36
repeated Field Type

This is a special field "group", that creates two identical fields whose values must match (or a validation error is thrown). The most common use is when you need the user to repeat their password or email to verify accuracy.

Rendered as	input `text` field by default, but see type option
Options	• first_name • first_options • options • second_name • second_options • type
Overridden Options	• error_bubbling
Inherited options	• data • error_mapping • invalid_message • invalid_message_parameters • mapped
Parent type	*form*
Class	*RepeatedType*[1]

Example Usage

Listing 36-1

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/RepeatedType.html

```php
$builder->add('password', 'repeated', array(
    'type' => 'password',
    'invalid_message' => 'The password fields must match.',
    'options' => array('attr' => array('class' => 'password-field')),
    'required' => true,
    'first_options'  => array('label' => 'Password'),
    'second_options' => array('label' => 'Repeat Password'),
));
```

Upon a successful form submit, the value entered into both of the "password" fields becomes the data of the `password` key. In other words, even though two fields are actually rendered, the end data from the form is just the single value (usually a string) that you need.

The most important option is `type`, which can be any field type and determines the actual type of the two underlying fields. The `options` option is passed to each of those individual fields, meaning - in this example - any option supported by the `password` type 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 36-2
```twig
{{ form_row(form.password) }}
```

To render each field individually, use something like this:

Listing 36-3
```twig
{# .first and .second may vary in your use - see the note below #}
{{ form_row(form.password.first) }}
{{ form_row(form.password.second) }}
```

The names `first` and `second` are the default names for the two sub-fields. However, these names can be controlled via the first_name and second_name options. If you've set these options, then use those values instead of `first` and `second` when rendering.

Validation

One of the key features of the `repeated` field is internal validation (you don't need to do anything to set this up) that forces the two fields to have a matching value. If the two fields don't match, an error will be shown to the user.

The `invalid_message` is used to customize the error that will be displayed when the two fields do not match each other.

Field Options

first_name

type: `string` **default**: `first`

This is the actual field name to be used for the first field. This is mostly meaningless, however, as the actual data entered into both of the fields will be available under the key assigned to the `repeated` field

itself (e.g. `password`). However, if you don't specify a label, this field name is used to "guess" the label for you.

first_options

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

Additional options (will be merged into *options* above) that should be passed *only* to the first field. This is especially useful for customizing the label:

Listing 36-4
```
$builder->add('password', 'repeated', array(
    'first_options'  => array('label' => 'Password'),
    'second_options' => array('label' => 'Repeat Password'),
));
```

options

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

This options array will be passed to each of the two underlying fields. In other words, these are the options that customize the individual field types. For example, if the `type` option is set to `password`, this array might contain the options `always_empty` or `required` - both options that are supported by the `password` field type.

second_name

type: `string` **default**: `second`

The same as `first_name`, but for the second field.

second_options

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

Additional options (will be merged into *options* above) that should be passed *only* to the second field. This is especially useful for customizing the label (see first_options).

type

type: `string` **default**: `text`

The two underlying fields will be of this field type. For example, passing a type of `password` will render two password fields.

Overridden Options

error_bubbling

default: `false`

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 36-5
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

error_mapping

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 36-6
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 36-7
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

mapped

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

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

Chapter 37

hidden Field Type

The hidden type represents a hidden input field.

Rendered as	`input hidden` field
Overriden options	• error_bubbling • required
Inherited options	• data • error_mapping • mapped • property_path
Parent type	*form*
Class	*HiddenType*[1]

Overridden Options

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.

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/HiddenType.html

Inherited Options

These options inherit from the *form* type:

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 37-1
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

error_mapping

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 37-2
```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

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: `any` **default**: `the field's name`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value is the field's name.

If you wish the field to be ignored when reading or writing to the object you can set the `property_path` option to `false`, but using `property_path` for this purpose is deprecated, you should use the `mapped` option.

New in version 2.1: The `mapped` option was introduced in Symfony 2.1 for this use-case.

Chapter 38
button Field Type

New in version 2.3: The `button` type was introduced in Symfony 2.3

A simple, non-responsive button.

Rendered as	`button` tag
Inherited options	• attr • disabled • label • translation_domain
Parent type	none
Class	*`ButtonType`*[1]

Inherited Options

The following options are defined in the *`BaseType`*[2] class. The `BaseType` class is the parent class for both the `button` type and the *form type*, but it is not part of the form type tree (i.e. it can not be used as a form type on its own).

attr

type: array **default**: Empty array

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 38-1

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/ButtonType.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/BaseType.html

```
1 $builder->add('save', 'button', array(
2     'attr' => array('class' => 'save'),
3 ));
```

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 38-2
```
1 {{ form_widget(form.save, { 'label': 'Click me' }) }}
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

Chapter 39

reset Field Type

New in version 2.3: The `reset` type was introduced in Symfony 2.3

A button that resets all fields to their original values.

Rendered as	`input reset` tag
Inherited options	• attr • disabled • label • label_attr • translation_domain
Parent type	*button*
Class	*ResetType*[1]

Inherited Options

attr

type: array **default**: Empty array

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 39-1

```
$builder->add('save', 'button', array(
    'attr' => array('class' => 'save'),
));
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/ResetType.html

disabled

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

If you don't want a user to be able to click a button, you can set the disabled option to true. It will not be possible to submit the form with this button, not even when bypassing the browser and sending a request manually, for example with cURL.

label

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

Sets the label that will be displayed on the button. The label can also be directly set inside the template:

Listing 39-2
```
{{ form_widget(form.save, { 'label': 'Click me' }) }}
```

label_attr

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

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

Listing 39-3
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

Chapter 40
submit Field Type

New in version 2.3: The `submit` type was introduced in Symfony 2.3

A submit button.

Rendered as	`button submit` tag
Inherited options	• attr • disabled • label • label_attr • translation_domain • validation_groups
Parent type	*button*
Class	*SubmitType*[1]

The Submit button has an additional method *isClicked()*[2] that lets you check whether this button was used to submit the form. This is especially useful when *a form has multiple submit buttons*:

Listing 40-1
```
1 if ($form->get('save')->isClicked()) {
2     // ...
3 }
```

Inherited Options

attr

type: array **default**: Empty array

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/SubmitType.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/ClickableInterface.html#isClicked()

If you want to add extra attributes to the HTML representation of the button, you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for the button:

Listing 40-2
```
$builder->add('save', 'button', array(
    'attr' => array('class' => 'save'),
));
```

disabled

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

If you don't want a user to be able to click a button, you can set the disabled option to true. It will not be possible to submit the form with this button, not even when bypassing the browser and sending a request manually, for example with cURL.

label

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

Sets the label that will be displayed on the button. The label can also be directly set inside the template:

Listing 40-3
```
{{ form_widget(form.save, { 'label': 'Click me' }) }}
```

label_attr

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

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

Listing 40-4
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this button.

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 40-5
```
$form = $this->createFormBuilder($user)
    ->add('previousStep', 'submit', array(
        'validation_groups' => false,
    ))
    ->add('nextStep', 'submit', array(
        'validation_groups' => array('Registration'),
```

```
7       ))
8       ->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 the Form chapter of the book.

Form Variables

Variable	Type	Usage
clicked	`Boolean`	Whether the button is clicked or not.

Chapter 41
form Field Type

The `form` type predefines a couple of options that are then available on all types for which `form` is the parent type.

Options	• action • by_reference • cascade_validation • compound • constraints • data • data_class • empty_data • error_bubbling • error_mapping • extra_fields_message • inherit_data • invalid_message • invalid_message_parameters • label_attr • mapped • max_length • method • pattern • post_max_size_message • property_path • read_only • required • trim
Inherited options	• attr • auto_initialize • block_name

	• disabled • label • translation_domain
Parent	none
Class	*FormType*[1]

Field Options

action

New in version 2.3: The `action` option was introduced in Symfony 2.3.

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.

by_reference

type: `Boolean` **default**: `true`

In most cases, if you have a `name` field, then you expect `setName()` to be called on the underlying object. In some cases, however, `setName()` may *not* be called. Setting `by_reference` ensures that the setter is called in all cases.

To explain this further, here's a simple example:

Listing 41-1
```
$builder = $this->createFormBuilder($article);
$builder
    ->add('title', 'text')
    ->add(
        $builder->create('author', 'form', array('by_reference' => ?))
            ->add('name', 'text')
            ->add('email', 'email')
    )
```

If `by_reference` is true, the following takes place behind the scenes when you call `submit()` (or `handleRequest()`) on the form:

Listing 41-2
```
$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 41-3
```
$article->setTitle('...');
$author = $article->getAuthor();
$author->setName('...');
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/FormType.html

```
$author->setEmail('...');
$article->setAuthor($author);
```

So, all that `by_reference=false` really does is force the framework to call the setter on the parent object.

Similarly, if you're using the *collection* form type 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.

cascade_validation

type: Boolean **default**: false

Set this option to `true` to force validation on embedded form types. For example, if you have a `ProductType` with an embedded `CategoryType`, setting `cascade_validation` to `true` on `ProductType` will cause the data from `CategoryType` to also be validated.

Instead of using this option, it is recommended that you use the `Valid` constraint in your model to force validation on a child object stored on a property. This cascades only the validation but not the use of the `validation_group` option on child forms. You can read more about this in the section about *Embedding a Single Object*.

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`.

compound

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

This option specifies if a form is compound. This is independent of whether the form actually has children. A form can be compound but not have any children at all (e.g. an empty collection form).

constraints

type: array or *`Constraint`*[2] **default**: `null`

Allows you to attach one or more validation constraints to a specific field. For more information, see *Adding Validation*. This option is added in the *`FormTypeValidatorExtension`*[3] form extension.

data

type: `mixed` **default**: Defaults to field of the underlying object (if there is one)

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

Listing 41-4
```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraint.html
3. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Validator/Type/FormTypeValidatorExtension.html

The default values for form fields are taken directly from the underlying data structure (e.g. an entity or an array). The `data` option overrides this default value.

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 41-5
```
1 $builder->add('media', 'sonata_media_type', array(
2     'data_class' => 'Acme\DemoBundle\Entity\Media',
3 ));
```

empty_data

type: `mixed`

The actual default value of this option depends on other field options:

- If `data_class` is set and `required` is `true`, then `new $data_class()`;
- If `data_class` is set and `required` is `false`, then `null`;
- If `data_class` is not set and `compound` is `true`, then `array()` (empty array);
- If `data_class` is not set and `compound` is `false`, then `''` (empty string).

This option determines what value the field will return when the submitted value is empty.

But you can customize this to your needs. For example, if you want the `gender` choice field to be explicitly set to `null` when no value is selected, you can do it like this:

Listing 41-6
```
1 $builder->add('gender', 'choice', array(
2     'choices' => array(
3         'm' => 'Male',
4         'f' => 'Female'
5     ),
6     'required'    => false,
7     'empty_value' => 'Choose your gender',
8     'empty_data'  => null
9 ));
```

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

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

New in version 2.1: The `error_mapping` option was introduced in Symfony 2.1.

type: `array` **default**: `empty`

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 41-7
```php
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'error_mapping' => array(
            'matchingCityAndZipCode' => 'city',
        ),
    ));
}
```

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

- The left side contains property paths;
- If the violation is generated on a property or method of a class, its path is simply `propertyName`;
- If the violation is generated on an entry of an `array` or `ArrayAccess` object, the property path is `[indexName]`;
- You can construct nested property paths by concatenating them, separating properties by dots. For example: `addresses[work].matchingCityAndZipCode`;
- The left side of the error mapping also accepts a dot `.`, which refers to the field itself. That means that any error added to the field is added to the given nested field instead;
- The right side contains simply the names of fields in the form.

extra_fields_message

type: `string` **default**: `This form should not contain extra fields.`

This is the validation error message that's used if the submitted form data contains one or more fields that are not part of the form definition. The placeholder `{{ extra_fields }}` can be used to display a comma separated list of the submitted extra field names.

inherit_data

New in version 2.3: The `inherit_data` option was introduced in Symfony 2.3. Before, it was known as `virtual`.

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 *time* field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules (*reference*).

invalid_message_parameters

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

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

Listing 41-8
```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num%
letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

label_attr

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

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

Listing 41-9
```
{{ form_label(form.name, 'Your name', {'label_attr': {'class': 'CUSTOM_LABEL_CLASS'}}) }}
```

mapped

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

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

max_length

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

If this option is not null, an attribute `maxlength` is added, which is used by some browsers to limit the amount of text in a field.

This is just a browser validation, so data must still be validated server-side.

method

New in version 2.3: The `method` option was introduced in Symfony 2.3.

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. For more information, see *How to Use HTTP Methods beyond GET and POST in Routes*.

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`.

pattern

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

This adds an HTML5 `pattern` attribute to restrict the field input by a given regular expression.

The `pattern` attribute provides client-side validation for convenience purposes only and must not be used as a replacement for reliable server-side validation.

When using validation constraints, this option is set automatically for some constraints to match the server-side validation.

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: `any` **default**: `the field's name`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value is the field's name.

If you wish the field to be ignored when reading or writing to the object you can set the `property_path` option to `false`, but using `property_path` for this purpose is deprecated, you should use the `mapped` option.

New in version 2.1: The `mapped` option was introduced in Symfony 2.1 for this use-case.

read_only

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

If this option is true, the field will be rendered with the `readonly` attribute so that the field is not editable.

required

type: `Boolean` **default**: `true`

If true, an *HTML5 required attribute*[4] will be rendered. The corresponding `label` will also render with a `required` class.

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

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

trim

type: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the *`trim`*[5] function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

Inherited Options

The following options are defined in the *`BaseType`*[6] class. The `BaseType` class is the parent class for both the `form` type and the *button type*, but it is not part of the form type tree (i.e. it can not be used as a form type on its own).

attr

type: array **default**: Empty array

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

Listing 41-10

4. http://diveintohtml5.info/forms.html
5. http://php.net/manual/en/function.trim.php
6. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Core/Type/BaseType.html

```php
$builder->add('body', 'textarea', array(
    'attr' => array('class' => 'tinymce'),
));
```

auto_initialize

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

An internal option: sets whether the form should be initialized automatically. For all fields, this option should only be `true` for root forms. You won't need to change this option and probably won't need to worry about it.

block_name

type: `string` **default**: the form's name (see *Knowing which block to customize*)

Allows you to override the block name used to render the form type. Useful for example if you have multiple instances of the same form and you need to personalize the rendering of the forms individually.

disabled

New in version 2.1: The `disabled` option was introduced in Symfony 2.1.

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 41-11
```twig
{{ form_label(form.name, 'Your name') }}
```

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 field.

Chapter 42

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*
- *Null*
- *True*
- *False*
- *Type*

String Constraints

- *Email*
- *Length*
- *Url*
- *Regex*
- *Ip*

Number Constraints

- *Range*

Comparison Constraints

- *EqualTo*
- *NotEqualTo*
- *IdenticalTo*
- *NotIdenticalTo*
- *LessThan*
- *LessThanOrEqual*
- *GreaterThan*
- *GreaterThanOrEqual*

Date Constraints

- *Date*
- *DateTime*
- *Time*

Collection Constraints

- *Choice*
- *Collection*
- *Count*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Financial and other Number Constraints

- *CardScheme*
- *Currency*
- *Luhn*
- *Iban*
- *Isbn*
- *Issn*

Other Constraints

- *Callback*
- *All*
- *UserPassword*
- *Valid*

Chapter 43
NotBlank

Validates that a value is not blank, defined as not equal to a blank string and also not equal to `null`. To force that a value is simply not equal to `null`, see the *NotNull* constraint.

Applies to	*property or method*
Options	• message
Class	*NotBlank*[1]
Validator	*NotBlankValidator*[2]

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not blank, you could do the following:

Listing 43-1

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    protected $firstName;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotBlank.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotBlankValidator.html

Options

message

type: `string` **default**: `This value should not be blank.`

This is the message that will be shown if the value is blank.

Chapter 44
Blank

Validates that a value is blank, defined as equal to a blank string or equal to `null`. To force that a value strictly be equal to `null`, see the *Null* constraint. To force that a value is *not* blank, see *NotBlank*.

Applies to	*property or method*
Options	• message
Class	*Blank*[1]
Validator	*BlankValidator*[2]

Basic Usage

If, for some reason, you wanted to ensure that the `firstName` property of an `Author` class were blank, you could do the following:

Listing 44-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Blank()
     */
    protected $firstName;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Blank.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/BlankValidator.html

Options

message

type: `string` **default**: `This value should be blank.`

This is the message that will be shown if the value is not blank.

Chapter 45
NotNull

Validates that a value is not strictly equal to `null`. To ensure that a value is simply not blank (not a blank string), see the *NotBlank* constraint.

Applies to	*property or method*
Options	• message
Class	*NotNull*[1]
Validator	*NotNullValidator*[2]

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not strictly equal to `null`, you would:

Listing 45-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotNull()
     */
    protected $firstName;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotNull.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotNullValidator.html

Options

message

type: `string` **default**: `This value should not be null.`

This is the message that will be shown if the value is `null`.

Chapter 46
Null

Validates that a value is exactly equal to `null`. To force that a property is simply blank (blank string or `null`), see the *Blank* constraint. To ensure that a property is not null, see *NotNull*.

Applies to	*property or method*
Options	• message
Class	*Null*[1]
Validator	*NullValidator*[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 46-1

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Null()
     */
    protected $firstName;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Null.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NullValidator.html

When using YAML, be sure to surround `Null` with quotes (`'Null'`) or else YAML will convert this into a `null` value.

Options

message

type: `string` **default**: `This value should be null.`

This is the message that will be shown if the value is not `null`.

Chapter 47
True

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 *False*.

Applies to	*property or method*
Options	• message
Class	*True*[1]
Validator	*TrueValidator*[2]

Basic Usage

This constraint can be applied to properties (e.g. a `termsAccepted` property on a registration model) or to a "getter" method. It's most powerful in the latter case, where you can assert that a method returns a true value. For example, suppose you have the following method:

Listing 47-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    protected $token;

    public function isTokenValid()
    {
        return $this->token == $this->generateToken();
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/True.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/TrueValidator.html

```
    }
}
```

Then you can constrain this method with `True`.

Listing 47-2
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    protected $token;

    /**
     * @Assert\True(message = "The token is invalid")
     */
    public function isTokenValid()
    {
        return $this->token == $this->generateToken();
    }
}
```

If the `isTokenValid()` returns false, the validation will fail.

When using YAML, be sure to surround `True` with quotes (`'True'`) or else YAML will convert this into a `true` Boolean value.

Options

message

type: `string` **default**: `This value should be true.`

This message is shown if the underlying data is not true.

Chapter 48
False

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 *True*.

Applies to	*property or method*
Options	• message
Class	*`False`*[1]
Validator	*`FalseValidator`*[2]

Basic Usage

The `False` 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 48-1

```
protected $state;

protected $invalidStates = array();

public function isStateInvalid()
{
    return in_array($this->state, $this->invalidStates);
}
```

In this case, the underlying object is only valid if the `isStateInvalid` method returns **false**:

Listing 48-2

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/False.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/FalseValidator.html

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\False(
     *     message = "You've entered an invalid state."
     * )
     */
     public function isStateInvalid()
     {
        // ...
     }
}
```

When using YAML, be sure to surround `False` with quotes (`'False'`) or else YAML will convert this into a `false` Boolean value.

Options

message

type: `string` **default**: `This value should be false.`

This message is shown if the underlying data is not false.

Chapter 49
Type

Validates that a value is of a specific data type. For example, if a variable should be an array, you can use this constraint with the `array` type option to validate this.

Applies to	*property or method*
Options	• *type* • message
Class	*Type*[1]
Validator	*TypeValidator*[2]

Basic Usage

Listing 49-1

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Type(type="integer", message="The value {{ value }} is not a valid {{ type }}.")
     */
    protected $age;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Type.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/TypeValidator.html

Options

type

type: `string` [*default option*]

This required option is the fully qualified class name or one of the PHP datatypes as determined by PHP's `is_` functions.

- *array*[3]
- *bool*[4]
- *callable*[5]
- *float*[6]
- *double*[7]
- *int*[8]
- *integer*[9]
- *long*[10]
- *null*[11]
- *numeric*[12]
- *object*[13]
- *real*[14]
- *resource*[15]
- *scalar*[16]
- *string*[17]

Also, you can use `ctype_` functions from corresponding *built-in PHP extension*[18]. Consider *a list of ctype functions*[19]:

- *alnum*[20]
- *alpha*[21]
- *cntrl*[22]
- *digit*[23]
- *graph*[24]
- *lower*[25]
- *print*[26]

---

3. http://php.net/manual/en/function.is-array.php
4. http://php.net/manual/en/function.is-bool.php
5. http://php.net/manual/en/function.is-callable.php
6. http://php.net/manual/en/function.is-float.php
7. http://php.net/manual/en/function.is-double.php
8. http://php.net/manual/en/function.is-int.php
9. http://php.net/manual/en/function.is-integer.php
10. http://php.net/manual/en/function.is-long.php
11. http://php.net/manual/en/function.is-null.php
12. http://php.net/manual/en/function.is-numeric.php
13. http://php.net/manual/en/function.is-object.php
14. http://php.net/manual/en/function.is-real.php
15. http://php.net/manual/en/function.is-resource.php
16. http://php.net/manual/en/function.is-scalar.php
17. http://php.net/manual/en/function.is-string.php
18. http://php.net/book.ctype.php
19. http://php.net/ref.ctype.php
20. http://php.net/manual/en/function.ctype-alnum.php
21. http://php.net/manual/en/function.ctype-alpha.php
22. http://php.net/manual/en/function.ctype-cntrl.php
23. http://php.net/manual/en/function.ctype-digit.php
24. http://php.net/manual/en/function.ctype-graph.php
25. http://php.net/manual/en/function.ctype-lower.php

- *punct*[27]
- *space*[28]
- *upper*[29]
- *xdigit*[30]

Make sure that the proper *locale*[31] is set before using one of these.

message

type: `string` **default**: `This value should be of type {{ type }}.`

The message if the underlying data is not of the given type.

26. http://php.net/manual/en/function.ctype-print.php
27. http://php.net/manual/en/function.ctype-punct.php
28. http://php.net/manual/en/function.ctype-space.php
29. http://php.net/manual/en/function.ctype-upper.php
30. http://php.net/manual/en/function.ctype-xdigit.php
31. http://php.net/manual/en/function.setlocale.php

Chapter 50
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	• message • checkMX • checkHost
Class	*Email*[1]
Validator	*EmailValidator*[2]

Basic Usage

Listing 50-1

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Email(
     *     message = "The email '{{ value }}' is not a valid email.",
     *     checkMX = true
     * )
     */
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Email.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/EmailValidator.html

```
14         protected $email;
15     }
```

Options

message

type: `string` **default**: `This value is not a valid email address.`

This message is shown if the underlying data is not a valid email address.

checkMX

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

If true, then the *`checkdnsrr`*[3] PHP function will be used to check the validity of the MX record of the host of the given email.

checkHost

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

If true, then the *`checkdnsrr`*[4] 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.

3. http://php.net/manual/en/function.checkdnsrr.php
4. http://php.net/manual/en/function.checkdnsrr.php

Chapter 51
Length

Validates that a given string length is *between* some minimum and maximum value.

Applies to	*property or method*
Options	• min • max • charset • minMessage • maxMessage • exactMessage
Class	*Length*[1]
Validator	*LengthValidator*[2]

Basic Usage

To verify that the `firstName` field length of a class is between "2" and "50", you might add the following:

Listing 51-1

```
// src/Acme/EventBundle/Entity/Participant.php
namespace Acme\EventBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Length(
     *      min = 2,
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Length.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LengthValidator.html

```
11          *      max = 50,
12          *      minMessage = "Your first name must be at least {{ limit }} characters long",
13          *      maxMessage = "Your first name cannot be longer than {{ limit }} characters"
14          * )
15          */
16         protected $firstName;
17     }
```

Options

min

type: `integer`

This required option is the "min" length value. Validation will fail if the given value's length is **less** than this min value.

max

type: `integer`

This required option is the "max" length value. Validation will fail if the given value's length is **greater** than this max value.

charset

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

The charset to be used when computing value's length. The *grapheme_strlen*[3] PHP function is used if available. If not, the *mb_strlen*[4] PHP function is used if available. If neither are available, the *strlen*[5] PHP function is used.

minMessage

type: `string` **default**: `This value is too short. It should have {{ limit }} characters or more.`

The message that will be shown if the underlying value's length is less than the min option.

maxMessage

type: `string` **default**: `This value is too long. It should have {{ limit }} characters or less.`

The message that will be shown if the underlying value's length is more than the max option.

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.

---

3. http://php.net/manual/en/function.grapheme-strlen.php
4. http://php.net/manual/en/function.mb-strlen.php
5. http://php.net/manual/en/function.strlen.php

Chapter 52
Url

Validates that a value is a valid URL string.

Applies to	*property or method*
Options	• message • protocols
Class	*Url*[1]
Validator	*UrlValidator*[2]

Basic Usage

Listing 52-1

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url()
     */
     protected $bioUrl;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Url.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/UrlValidator.html

Options

message

type: `string` **default**: `This value is not a valid URL.`

This message is shown if the URL is invalid.

protocols

type: `array` **default**: `array('http', 'https')`

The protocols that will be considered to be valid. For example, if you also needed `ftp://` type URLs to be valid, you'd redefine the `protocols` array, listing `http`, `https`, and also `ftp`.

Chapter 53
Regex

Validates that a value matches a regular expression.

Applies to	*property or method*
Options	• pattern • htmlPattern • match • message
Class	*Regex*[1]
Validator	*RegexValidator*[2]

Basic Usage

Suppose you have a `description` field and you want to verify that it begins with a valid word character. The regular expression to test for this would be `/^\w+/`, indicating that you're looking for at least one or more word characters at the beginning of your string:

Listing 53-1

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex("/^\w+/")
     */
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Regex.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/RegexValidator.html

```
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 53-2

```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\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 }
```

Options

pattern

type: `string` [*default option*]

This required option is the regular expression pattern that the input will be matched against. By default, this validator will fail if the input string does *not* match this regular expression (via the *preg_match*[3] PHP function). However, if match is set to false, then validation will fail if the input string *does* match this pattern.

htmlPattern

New in version 2.1: The `htmlPattern` option was introduced in Symfony 2.1

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 53-3

```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
```

3. http://php.net/manual/en/function.preg-match.php

```
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.

Chapter 54

Ip

Validates that a value is a valid IP address. By default, this will validate the value as IPv4, but a number of different options exist to validate as IPv6 and many other combinations.

Applies to	*property or method*
Options	• version • message
Class	*Ip*[1]
Validator	*IpValidator*[2]

Basic Usage

Listing 54-1

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Ip
     */
     protected $ipAddress;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Ip.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IpValidator.html

Options

version

type: `string` **default**: `4`

This determines exactly *how* the IP address is validated and can take one of a variety of different values:

All ranges

- `4` - Validates for IPv4 addresses
- `6` - Validates for IPv6 addresses
- `all` - Validates all IP formats

No private ranges

- `4_no_priv` - Validates for IPv4 but without private IP ranges
- `6_no_priv` - Validates for IPv6 but without private IP ranges
- `all_no_priv` - Validates for all IP formats but without private IP ranges

No reserved ranges

- `4_no_res` - Validates for IPv4 but without reserved IP ranges
- `6_no_res` - Validates for IPv6 but without reserved IP ranges
- `all_no_res` - Validates for all IP formats but without reserved IP ranges

Only public ranges

- `4_public` - Validates for IPv4 but without private and reserved ranges
- `6_public` - Validates for IPv6 but without private and reserved ranges
- `all_public` - Validates for all IP formats but without private and reserved ranges

message

type: `string` **default**: `This is not a valid IP address.`

This message is shown if the string is not a valid IP address.

Chapter 55
Range

Validates that a given number is *between* some minimum and maximum number.

Applies to	*property or method*
Options	• min • max • minMessage • maxMessage • invalidMessage
Class	*Range*[1]
Validator	*RangeValidator*[2]

Basic Usage

To verify that the "height" field of a class is between "120" and "180", you might add the following:

Listing 55-1

```php
// src/Acme/EventBundle/Entity/Participant.php
namespace Acme\EventBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Range(
     *      min = 120,
     *      max = 180,
     *      minMessage = "You must be at least {{ limit }}cm tall to enter",
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Range.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/RangeValidator.html

```
13          *      maxMessage = "You cannot be taller than {{ limit }}cm to enter"
14          * )
15          */
16         protected $height;
17     }
```

Options

min

type: `integer`

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

max

type: `integer`

This required option is the "max" value. Validation will fail if the given value is **greater** than this max value.

minMessage

type: `string` **default**: `This value should be {{ limit }} or more.`

The message that will be shown if the underlying value is less than the min option.

maxMessage

type: `string` **default**: `This value should be {{ limit }} or less.`

The message that will be shown if the underlying value is more than the max option.

invalidMessage

type: `string` **default**: `This value should be a valid number.`

The message that will be shown if the underlying value is not a number (per the *is_numeric*[3] PHP function).

3. http://www.php.net/manual/en/function.is-numeric.php

Chapter 56
EqualTo

New in version 2.3: The `EqualTo` constraint was introduced in Symfony 2.3.

Validates that a value is equal to another value, defined in the options. To force that a value is *not* equal, see *NotEqualTo*.

This constraint compares using `==`, so `3` and `"3"` are considered equal. Use *IdenticalTo* to compare with `===`.

Applies to	*property or method*
Options	• value • message
Class	*EqualTo*[1]
Validator	*EqualToValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is equal to `20`, you could do the following:

Listing 56-1
```
1  // src/Acme/SocialBundle/Entity/Person.php
2  namespace Acme\SocialBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/EqualTo.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/EqualToValidator.html

```
8       /**
9        * @Assert\EqualTo(
10       *     value = 20
11       * )
12       */
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be equal to {{ compared_value }}.`

This is the message that will be shown if the value is not equal.

Chapter 57
NotEqualTo

New in version 2.3: The `NotEqualTo` constraint was introduced in Symfony 2.3.

Validates that a value is **not** equal to another value, defined in the options. To force that a value is equal, see *EqualTo*.

This constraint compares using `!=`, so `3` and `"3"` are considered equal. Use *NotIdenticalTo* to compare with `!==`.

Applies to	*property or method*
Options	• value • message
Class	*NotEqualTo*[1]
Validator	*NotEqualToValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is not equal to `15`, you could do the following:

Listing 57-1
```
1  // src/Acme/SocialBundle/Entity/Person.php
2  namespace Acme\SocialBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
7  {
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotEqualTo.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotEqualToValidator.html

```
8       /**
9        * @Assert\NotEqualTo(
10       *     value = 15
11       * )
12       */
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should not be equal to {{ compared_value }}.`

This is the message that will be shown if the value is equal.

Chapter 58
IdenticalTo

New in version 2.3: The `IdenticalTo` constraint was introduced in Symfony 2.3.

Validates that a value is identical to another value, defined in the options. To force that a value is *not* identical, see *NotIdenticalTo*.

This constraint compares using `===`, so `3` and `"3"` are *not* considered equal. Use *EqualTo* to compare with `==`.

Applies to	*property or method*
Options	• value • message
Class	*IdenticalTo*[1]
Validator	*IdenticalToValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is equal to `20` and an integer, you could do the following:

Listing 58-1
```
1  // src/Acme/SocialBundle/Entity/Person.php
2  namespace Acme\SocialBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IdenticalTo.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IdenticalToValidator.html

```
7  {
8      /**
9       * @Assert\IdenticalTo(
10      *     value = 20
11      * )
12      */
13     protected $age;
14 }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be identical to {{ compared_value_type }} {{ compared_value }}.`

This is the message that will be shown if the value is not identical.

Chapter 59
NotIdenticalTo

New in version 2.3: The `NotIdenticalTo` constraint was introduced in Symfony 2.3.

Validates that a value is **not** identical to another value, defined in the options. To force that a value is identical, see *IdenticalTo*.

This constraint compares using `!==`, so `3` and `"3"` are considered not equal. Use *NotEqualTo* to compare with `!=`.

Applies to	*property or method*
Options	• value • message
Class	*NotIdenticalTo*[1]
Validator	*NotIdenticalToValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is *not* equal to `15` and *not* an integer, you could do the following:

Listing 59-1
```
1  // src/Acme/SocialBundle/Entity/Person.php
2  namespace Acme\SocialBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Person
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotIdenticalTo.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/NotIdenticalToValidator.html

```
7  {
8      /**
9       * @Assert\NotIdenticalTo(
10      *     value = 15
11      * )
12      */
13     protected $age;
14 }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should not be identical to {{ compared_value_type }} {{ compared_value }}.`

This is the message that will be shown if the value is not equal.

Chapter 60
LessThan

New in version 2.3: The `LessThan` constraint was introduced in Symfony 2.3.

Validates that a value is less than another value, defined in the options. To force that a value is less than or equal to another value, see *LessThanOrEqual*. To force a value is greater than another value, see *GreaterThan*.

Applies to	*property or method*
Options	• value • message
Class	*LessThan*[1]
Validator	*LessThanValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is less than `80`, you could do the following:

Listing 60-1

```
// src/Acme/SocialBundle/Entity/Person.php
namespace Acme\SocialBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\LessThan(
     *     value = 80
     * )
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LessThan.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LessThanValidator.html

```
12          */
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be less than {{ compared_value }}.`

This is the message that will be shown if the value is not less than the comparison value.

Chapter 61
LessThanOrEqual

New in version 2.3: The `LessThanOrEqual` constraint was introduced in Symfony 2.3.

Validates that a value is less than or equal to another value, defined in the options. To force that a value is less than another value, see *LessThan*.

Applies to	*property or method*
Options	• value • message
Class	*LessThanOrEqual*[1]
Validator	*LessThanOrEqualValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is less than or equal to `80`, you could do the following:

Listing 61-1

```
// src/Acme/SocialBundle/Entity/Person.php
namespace Acme\SocialBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\LessThanOrEqual(
     *     value = 80
     * )
     */
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LessThanOrEqual.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LessThanOrEqualValidator.html

```
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be less than or equal to {{ compared_value }}.`

This is the message that will be shown if the value is not less than or equal to the comparison value.

Chapter 62
GreaterThan

New in version 2.3: The `GreaterThan` constraint was introduced in Symfony 2.3.

Validates that a value is greater than another value, defined in the options. To force that a value is greater than or equal to another value, see *GreaterThanOrEqual*. To force a value is less than another value, see *LessThan*.

Applies to	*property or method*
Options	• value • message
Class	*GreaterThan*[1]
Validator	*GreaterThanValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is greater than `18`, you could do the following:

Listing 62-1

```
// src/Acme/SocialBundle/Entity/Person.php
namespace Acme\SocialBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\GreaterThan(
     *     value = 18
     * )
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/GreaterThan.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/GreaterThanValidator.html

```
12          */
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be greater than {{ compared_value }}.`

This is the message that will be shown if the value is not greater than the comparison value.

Chapter 63
GreaterThanOrEqual

New in version 2.3: The `GreaterThanOrEqual` constraint was introduced in Symfony 2.3.

Validates that a value is greater than or equal to another value, defined in the options. To force that a value is greater than another value, see *GreaterThan*.

Applies to	*property or method*
Options	• value • message
Class	*GreaterThanOrEqual*[1]
Validator	*GreaterThanOrEqualValidator*[2]

Basic Usage

If you want to ensure that the `age` of a `Person` class is greater than or equal to `18`, you could do the following:

Listing 63-1

```
// src/Acme/SocialBundle/Entity/Person.php
namespace Acme\SocialBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Person
{
    /**
     * @Assert\GreaterThanOrEqual(
     *     value = 18
     * )
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/GreaterThanOrEqual.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/GreaterThanOrEqualValidator.html

```
12          */
13      protected $age;
14  }
```

Options

value

type: `mixed` [*default option*]

This option is required. It defines the value to compare to. It can be a string, number or object.

message

type: `string` **default**: `This value should be greater than or equal to {{ compared_value }}.`

This is the message that will be shown if the value is not greater than or equal to the comparison value.

Chapter 64
Date

Validates that a value is a valid date, meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid YYYY-MM-DD format.

Applies to	*property or method*
Options	• message
Class	*Date*[1]
Validator	*DateValidator*[2]

Basic Usage

Listing 64-1

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Date()
     */
     protected $birthday;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Date.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/DateValidator.html

Options

message

type: `string` **default**: `This value is not a valid date.`

This message is shown if the underlying data is not a valid date.

Chapter 65
DateTime

Validates that a value is a valid "datetime", meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid YYYY-MM-DD HH:MM:SS format.

Applies to	*property or method*
Options	• message
Class	*DateTime*[1]
Validator	*DateTimeValidator*[2]

Basic Usage

Listing 65-1

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\DateTime()
     */
     protected $createdAt;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/DateTime.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/DateTimeValidator.html

Options

message

type: `string` **default**: `This value is not a valid datetime.`

This message is shown if the underlying data is not a valid datetime.

Chapter 66
Time

Validates that a value is a valid time, meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid "HH:MM:SS" format.

Applies to	*property or method*
Options	• message
Class	*Time*[1]
Validator	*TimeValidator*[2]

Basic Usage

Suppose you have an Event class, with a `startAt` field that is the time of the day when the event starts:

Listing 66-1
```php
// src/Acme/EventBundle/Entity/Event.php
namespace Acme\EventBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Event
{
    /**
     * @Assert\Time()
     */
     protected $startsAt;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Time.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/TimeValidator.html

Options

message

type: `string` **default**: `This value is not a valid time.`

This message is shown if the underlying data is not a valid time.

Chapter 67
Choice

This constraint is used to ensure that the given value is one of a given set of *valid* choices. It can also be used to validate that each item in an array of items is one of those valid choices.

Applies to	*property or method*
Options	• choices • callback • multiple • min • max • message • multipleMessage • minMessage • maxMessage • strict
Class	*Choice*[1]
Validator	*ChoiceValidator*[2]

Basic Usage

The basic idea of this constraint is that you supply it with an array of valid values (this can be done in several ways) and it validates that the value of the given property exists in that array.

If your valid choice list is simple, you can pass them in directly via the choices option:

Listing 67-1
```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Choice.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/ChoiceValidator.html

```
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      /**
9       * @Assert\Choice(choices = {"male", "female"}, message = "Choose a valid gender.")
10      */
11     protected $gender;
12 }
```

Supplying the Choices with a Callback Function

You can also use a callback function to specify your options. This is useful if you want to keep your choices in some central location so that, for example, you can easily access those choices for validation or for building a select form element.

Listing 67-2
```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
3
4  class Author
5  {
6      public static function getGenders()
7      {
8          return array('male', 'female');
9      }
10 }
```

You can pass the name of this method to the callback option of the `Choice` constraint.

Listing 67-3
```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      /**
9       * @Assert\Choice(callback = "getGenders")
10      */
11     protected $gender;
12 }
```

If the static callback is stored in a different class, for example `Util`, you can pass the class name and the method as an array.

Listing 67-4
```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class Author
7  {
8      /**
```

```
 9       * @Assert\Choice(callback = {"Util", "getGenders"})
10       */
11      protected $gender;
12  }
```

Available Options

choices

type: `array` [*default option*]

A required option (unless callback is specified) - this is the array of options that should be considered in the valid set. The input value will be matched against this array.

callback

type: `string|array|Closure`

This is a callback method that can be used instead of the choices option to return the choices array. See Supplying the Choices with a Callback Function for details on its usage.

multiple

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

If this option is true, the input value is expected to be an array instead of a single, scalar value. The constraint will check that each value of the input array can be found in the array of valid choices. If even one of the input values cannot be found, the validation will fail.

min

type: `integer`

If the `multiple` option is true, then you can use the `min` option to force at least XX number of values to be selected. For example, if `min` is 3, but the input array only contains 2 valid items, the validation will fail.

max

type: `integer`

If the `multiple` option is true, then you can use the `max` option to force no more than XX number of values to be selected. For example, if `max` is 3, but the input array contains 4 valid items, the validation will fail.

message

type: `string` **default**: `The value you selected is not a valid choice.`

This is the message that you will receive if the `multiple` option is set to `false`, and the underlying value is not in the valid array of choices.

multipleMessage

type: `string` **default**: `One or more of the given values is invalid.`

This is the message that you will receive if the `multiple` option is set to `true`, and one of the values on the underlying array being checked is not in the array of valid choices.

minMessage

type: `string` **default**: `You must select at least {{ limit }} choices.`

This is the validation error message that's displayed when the user chooses too few choices per the min option.

maxMessage

type: `string` **default**: `You must select at most {{ limit }} choices.`

This is the validation error message that's displayed when the user chooses too many options per the max option.

strict

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

If true, the validator will also check the type of the input value. Specifically, this value is passed to as the third argument to the PHP *`in_array`*[3] method when checking to see if a value is in the valid choices array.

3. http://php.net/manual/en/function.in-array.php

Chapter 68
Collection

This constraint is used when the underlying data is a collection (i.e. an array or an object that implements `Traversable` and `ArrayAccess`), but you'd like to validate different keys of that collection in different ways. For example, you might validate the `email` key using the `Email` constraint and the `inventory` key of the collection with the `Range` constraint.

This constraint can also make sure that certain collection keys are present and that extra keys are not present.

Applies to	*property or method*
Options	• fields • allowExtraFields • extraFieldsMessage • allowMissingFields • missingFieldsMessage
Class	*Collection*[1]
Validator	*CollectionValidator*[2]

Basic Usage

The `Collection` constraint allows you to validate the different keys of a collection individually. Take the following example:

Listing 68-1
```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
3
4  class Author
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Collection.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CollectionValidator.html

```
{
    protected $profileData = array(
        'personal_email',
        'short_bio',
    );

    public function setProfileData($key, $value)
    {
        $this->profileData[$key] = $value;
    }
}
```

To validate that the `personal_email` element of the `profileData` array property is a valid email address and that the `short_bio` element is not blank but is no longer than 100 characters in length, you would do the following:

Listing 68-2

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Collection(
     *     fields = {
     *         "personal_email" = @Assert\Email,
     *         "short_bio" = {
     *             @Assert\NotBlank(),
     *             @Assert\Length(
     *                 max = 100,
     *                 maxMessage = "Your short bio is too long!"
     *             )
     *         }
     *     },
     *     allowMissingFields = true
     * )
     */
     protected $profileData = array(
         'personal_email',
         'short_bio',
     );
}
```

Presence and Absence of Fields

By default, this constraint validates more than simply whether or not the individual fields in the collection pass their assigned constraints. In fact, if any keys of a collection are missing or if there are any unrecognized keys in the collection, validation errors will be thrown.

If you would like to allow for keys to be absent from the collection or if you would like "extra" keys to be allowed in the collection, you can modify the allowMissingFields and allowExtraFields options respectively. In the above example, the `allowMissingFields` option was set to true, meaning that if either of the `personal_email` or `short_bio` elements were missing from the `$personalData` property, no validation error would occur.

Required and optional Field Constraints

New in version 2.3: The `Required` and `Optional` constraints were moved to the namespace `Symfony\Component\Validator\Constraints\` in Symfony 2.3.

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 68-3

```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Collection(
     *     fields={
     *         "personal_email"  = @Assert\Required({@Assert\NotBlank, @Assert\Email}),
     *         "alternate_email" = @Assert\Optional(@Assert\Email)
     *     }
     * )
     */
     protected $profileData = array('personal_email');
}
```

Even without `allowMissingFields` set to true, you can now omit the `alternate_email` property completely from the `profileData` array, since it is `Optional`. However, if the `personal_email` field does not exist in the array, the `NotBlank` constraint will still be applied (since it is wrapped in `Required`) and you will receive a constraint violation.

Options

fields

type: `array` [*default option*]

This option is required, and is an associative array defining all of the keys in the collection and, for each key, exactly which validator(s) should be executed against that element of the collection.

allowExtraFields

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

If this option is set to `false` and the underlying collection contains one or more elements that are not included in the fields option, a validation error will be returned. If set to `true`, extra fields are ok.

extraFieldsMessage

type: `Boolean` **default**: `The fields {{ fields }} were not expected.`

The message shown if allowExtraFields is false and an extra field is detected.

allowMissingFields

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

If this option is set to `false` and one or more fields from the fields option are not present in the underlying collection, a validation error will be returned. If set to `true`, it's ok if some fields in the fields option are not present in the underlying collection.

missingFieldsMessage

type: `Boolean` **default**: `The fields {{ fields }} are missing.`

The message shown if allowMissingFields is false and one or more fields are missing from the underlying collection.

Chapter 69
Count

Validates that a given collection's (i.e. an array or an object that implements Countable) element count is *between* some minimum and maximum value.

Applies to	*property or method*
Options	• min • max • minMessage • maxMessage • exactMessage
Class	*Count*[1]
Validator	*CountValidator*[2]

Basic Usage

To verify that the `emails` array field contains between 1 and 5 elements you might add the following:

Listing 69-1
```php
// src/Acme/EventBundle/Entity/Participant.php
namespace Acme\EventBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Count(
     *      min = "1",
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Count.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CountValidator.html

```
11          *      max = "5",
12          *      minMessage = "You must specify at least one email",
13          *      maxMessage = "You cannot specify more than {{ limit }} emails"
14          * )
15          */
16         protected $emails = array();
17     }
```

Options

min

type: `integer`

This required option is the "min" count value. Validation will fail if the given collection elements count is **less** than this min value.

max

type: `integer`

This required option is the "max" count value. Validation will fail if the given collection elements count is **greater** than this max value.

minMessage

type: `string` **default**: `This collection should contain {{ limit }} elements or more.`

The message that will be shown if the underlying collection elements count is less than the min option.

maxMessage

type: `string` **default**: `This collection should contain {{ limit }} elements or less.`

The message that will be shown if the underlying collection elements count is more than the max option.

exactMessage

type: `string` **default**: `This collection should contain exactly {{ limit }} elements.`

The message that will be shown if min and max values are equal and the underlying collection elements count is not exactly this value.

Chapter 70
UniqueEntity

Validates that a particular field (or fields) in a Doctrine entity is (are) unique. This is commonly used, for example, to prevent a new user to register using an email address that already exists in the system.

Applies to	*class*
Options	• fields • message • em • repositoryMethod • errorPath • ignoreNull
Class	*UniqueEntity*[1]
Validator	*UniqueEntityValidator*[2]

Basic Usage

Suppose you have an AcmeUserBundle bundle with a `User` entity that has an `email` field. You can use the `UniqueEntity` constraint to guarantee that the `email` field remains unique between all of the constraints in your user table:

Listing 70-1
```php
// Acme/UserBundle/Entity/Author.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\ORM\Mapping as ORM;

// DON'T forget this use statement!!!
```

1. http://api.symfony.com/2.3/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntity.html
2. http://api.symfony.com/2.3/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntityValidator.html

```php
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class Author
{
    /**
     * @var string $email
     *
     * @ORM\Column(name="email", type="string", length=255, unique=true)
     * @Assert\Email()
     */
    protected $email;

    // ...
}
```

Options

fields

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

This required option is the field (or list of fields) on which this entity should be unique. For example, if you specified both the `email` and `name` field in a single `UniqueEntity` constraint, then it would enforce that the combination value where unique (e.g. two users could have the same email, as long as they don't have the same name also).

If you need to require two fields to be individually unique (e.g. a unique `email` *and* a unique `username`), you use two `UniqueEntity` entries, each with a single field.

message

type: `string` **default**: `This value is already used.`

The message that's displayed when this constraint fails.

em

type: `string`

The name of the entity manager to use for making the query to determine the uniqueness. If it's left blank, the correct entity manager will be determined for this class. For that reason, this option should probably not need to be used.

repositoryMethod

type: `string` **default**: `findBy`

The name of the repository method to use for making the query to determine the uniqueness. If it's left blank, the `findBy` method will be used. This method should return a countable result.

errorPath

type: `string` **default**: The name of the first field in fields

New in version 2.1: The `errorPath` option was introduced in Symfony 2.1.

If the entity violates the constraint the error message is bound to the first field in fields. If there is more than one field, you may want to map the error message to another field.

Consider this example:

Listing 70-2

```php
// src/Acme/AdministrationBundle/Entity/Service.php
namespace Acme\AdministrationBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity(
 *     fields={"host", "port"},
 *     errorPath="port",
 *     message="This port is already in use on that host."
 * )
 */
class Service
{
    /**
     * @ORM\ManyToOne(targetEntity="Host")
     */
    public $host;

    /**
     * @ORM\Column(type="integer")
     */
    public $port;
}
```

Now, the message would be bound to the `port` field with this configuration.

ignoreNull

type: `Boolean` **default**: `true`

New in version 2.1: The `ignoreNull` option was introduced in Symfony 2.1.

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.

Chapter 71
Language

Validates that a value is a valid language *Unicode language identifier* (e.g. `fr` or `zh-Hant`).

Applies to	*property or method*
Options	• message
Class	*Language*[1]
Validator	*LanguageValidator*[2]

Basic Usage

Listing 71-1

```php
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Language()
     */
     protected $preferredLanguage;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Language.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LanguageValidator.html

Options

message

type: `string` **default**: `This value is not a valid language.`

This message is shown if the string is not a valid language code.

Chapter 72
Locale

Validates that a value is a valid locale.

The "value" for each locale is either the two letter *ISO 639-1*[1] *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the *ISO 3166-1 alpha-2*[2] *country* code (e.g. `fr_FR` for French/France).

Applies to	*property or method*
Options	• message
Class	*Locale*[3]
Validator	*LocaleValidator*[4]

Basic Usage

Listing 72-1

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Locale()
     */
```

1. http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
2. http://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Locale.html
4. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LocaleValidator.html

```
11         protected $locale;
12     }
```

Options

message

type: `string` **default**: `This value is not a valid locale.`

This message is shown if the string is not a valid locale.

Chapter 73
Country

Validates that a value is a valid *ISO 3166-1 alpha-2*[1] country code.

Applies to	*property or method*
Options	• message
Class	*Country*[2]
Validator	*CountryValidator*[3]

Basic Usage

Listing 73-1

```php
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Country()
     */
     protected $country;
}
```

1. http://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Country.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CountryValidator.html

Options

message

type: `string` **default**: `This value is not a valid country.`

This message is shown if the string is not a valid country code.

Chapter 74
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 *file* form type.

If the file you're validating is an image, try the *Image* constraint.

Applies to	*property or method*
Options	• maxSize • mimeTypes • maxSizeMessage • mimeTypesMessage • notFoundMessage • notReadableMessage • uploadIniSizeErrorMessage • uploadFormSizeErrorMessage • uploadErrorMessage
Class	*`File`*[3]
Validator	*`FileValidator`*[4]

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/File/File.html
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/File/UploadedFile.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/File.html
4. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/FileValidator.html

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a *file* form type. 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 74-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $bioFile;

    public function setBioFile(File $file = null)
    {
        $this->bioFile = $file;
    }

    public function getBioFile()
    {
        return $this->bioFile;
    }
}
```

To guarantee that the `bioFile` `File` object is valid, and that it is below a certain file size and a valid PDF, add the following:

Listing 74-2
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\File(
     *     maxSize = "1024k",
     *     mimeTypes = {"application/pdf", "application/x-pdf"},
     *     mimeTypesMessage = "Please upload a valid PDF"
     * )
     */
    protected $bioFile;
}
```

The `bioFile` property is validated to guarantee that it is a real file. Its size and mime type are also validated because the appropriate options have been specified.

Options

maxSize

type: `mixed`

If set, the size of the underlying file must be below this file size in order to be valid. The size of the file can be given in one of the following formats:

- **bytes**: To specify the `maxSize` in bytes, pass a value that is entirely numeric (e.g. `4096`);
- **kilobytes**: To specify the `maxSize` in kilobytes, pass a number and suffix it with a lowercase "k" (e.g. `200k`);
- **megabytes**: To specify the `maxSize` in megabytes, pass a number and suffix it with a capital "M" (e.g. `4M`).

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*[5].

maxSizeMessage

type: `string` **default**: `The file is too large ({{ size }} {{ suffix }}). Allowed maximum size is {{ limit }} {{ suffix }}.`

The message displayed if the file is larger than the maxSize option.

mimeTypesMessage

type: `string` **default**: `The mime type of the file is invalid ({{ type }}). Allowed mime types are {{ types }}.`

The message displayed if the mime type of the file is not a valid mime type per the mimeTypes option.

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.

notReadableMessage

type: `string` **default**: `The file is not readable.`

The message displayed if the file exists, but the PHP `is_readable` function fails when passed the path to the file.

uploadIniSizeErrorMessage

type: `string` **default**: `The file is too large. Allowed maximum size is {{ limit }} {{ suffix }}.`

The message that is displayed if the uploaded file is larger than the `upload_max_filesize` `php.ini` setting.

uploadFormSizeErrorMessage

type: `string` **default**: `The file is too large.`

5. `http://www.iana.org/assignments/media-types/index.html`

The message that is displayed if the uploaded file is larger than allowed by the HTML file input field.

uploadErrorMessage

type: `string` **default**: `The file could not be uploaded.`

The message that is displayed if the uploaded file could not be uploaded for some unknown reason, such as the file upload failed or it couldn't be written to disk.

Chapter 75
Image

The Image constraint works exactly like the *File* constraint, except that its mimeTypes and *mimeTypesMessage* options are automatically setup to work for image files specifically.

Additionally, as of Symfony 2.1, it has options so you can validate against the width and height of the image.

See the *File* constraint for the bulk of the documentation on this constraint.

Applies to	*property or method*
Options	• mimeTypes • minWidth • maxWidth • maxHeight • minHeight • mimeTypesMessage • sizeNotDetectedMessage • maxWidthMessage • minWidthMessage • maxHeightMessage • minHeightMessage • See *File* for inherited options
Class	*Image*[1]
Validator	*ImageValidator*[2]

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a *file* form type. For example, suppose you're creating an author form where you can upload a "headshot" image for the

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Image.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/ImageValidator.html

author. In your form, the `headshot` property would be a `file` type. The `Author` class might look as follows:

Listing 75-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $headshot;

    public function setHeadshot(File $file = null)
    {
        $this->headshot = $file;
    }

    public function getHeadshot()
    {
        return $this->headshot;
    }
}
```

To guarantee that the `headshot` `File` object is a valid image and that it is between a certain size, add the following:

Listing 75-2
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Image(
     *     minWidth = 200,
     *     maxWidth = 400,
     *     minHeight = 200,
     *     maxHeight = 400
     * )
     */
    protected $headshot;
}
```

The `headshot` property is validated to guarantee that it is a real image and that it is between a certain width and height.

Options

This constraint shares all of its options with the *File* constraint. It does, however, modify two of the default option values and add several other options.

mimeTypes

type: `array` or `string` **default**: `image/*`

You can find a list of existing image mime types on the *IANA website*[3].

mimeTypesMessage

type: `string` **default**: `This file is not a valid image.`

minWidth

type: `integer`

If set, the width of the image file must be greater than or equal to this value in pixels.

maxWidth

type: `integer`

If set, the width of the image file must be less than or equal to this value in pixels.

minHeight

type: `integer`

If set, the height of the image file must be greater than or equal to this value in pixels.

maxHeight

type: `integer`

If set, the height of the image file must be less than or equal to this value in pixels.

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 four size constraint options has been set.

maxWidthMessage

type: `string` **default**: `The image width is too big ({{ width }}px). Allowed maximum width is {{ max_width }}px.`

The error message if the width of the image exceeds maxWidth.

minWidthMessage

type: `string` **default**: `The image width is too small ({{ width }}px). Minimum width expected is {{ min_width }}px.`

The error message if the width of the image is less than minWidth.

maxHeightMessage

type: `string` **default**: `The image height is too big ({{ height }}px). Allowed maximum height is {{ max_height }}px.`

The error message if the height of the image exceeds maxHeight.

3. http://www.iana.org/assignments/media-types/image/index.html

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.

Chapter 76
CardScheme

New in version 2.2: The `CardScheme` constraint was introduced in Symfony 2.2.

This constraint ensures that a credit card number is valid for a given credit card company. It can be used to validate the number before trying to initiate a payment through a payment gateway.

Applies to	*property or method*
Options	• schemes • message
Class	*CardScheme*[1]
Validator	*CardSchemeValidator*[2]

Basic Usage

To use the `CardScheme` validator, simply apply it to a property or method on an object that will contain a credit card number.

Listing 76-1

```
// src/Acme/SubscriptionBundle/Entity/Transaction.php
namespace Acme\SubscriptionBundle\Entity\Transaction;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\CardScheme(schemes = {"VISA"}, message = "Your credit card number is
invalid.")
     */
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CardScheme.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CardSchemeValidator.html

```
11      protected $cardNumber;
12  }
```

Available Options

schemes

type: `mixed` [*default option*]

This option is required and represents the name of the number scheme used to validate the credit card number, it can either be a string or an array. Valid values are:

- `AMEX`
- `CHINA_UNIONPAY`
- `DINERS`
- `DISCOVER`
- `INSTAPAYMENT`
- `JCB`
- `LASER`
- `MAESTRO`
- `MASTERCARD`
- `VISA`

For more information about the used schemes, see *Wikipedia: Issuer identification number (IIN)*[3].

message

type: `string` **default**: `Unsupported card type or invalid card number.`

The message shown when the value does not pass the `CardScheme` check.

3. http://en.wikipedia.org/wiki/Bank_card_number#Issuer_identification_number_.28IIN.29

Chapter 77
Currency

New in version 2.3: The `Currency` constraint was introduced in Symfony 2.3.

Validates that a value is a valid *3-letter ISO 4217*[1] currency name.

Applies to	*property or method*
Options	• message
Class	*Currency*[2]
Validator	*CurrencyValidator*[3]

Basic Usage

If you want to ensure that the `currency` property of an `Order` is a valid currency, you could do the following:

Listing 77-1
```
// src/Acme/EcommerceBundle/Entity/Order.php
namespace Acme\EcommerceBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Order
{
    /**
     * @Assert\Currency
     */
    protected $currency;
}
```

1. http://en.wikipedia.org/wiki/ISO_4217
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Currency.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CurrencyValidator.html

Options

message

type: `string` **default**: `This value is not a valid currency.`

This is the message that will be shown if the value is not a valid currency.

Chapter 78
Luhn

New in version 2.2: The `Luhn` constraint was introduced in Symfony 2.2.

This constraint is used to ensure that a credit card number passes the *Luhn algorithm*[1]. It is useful as a first step to validating a credit card: before communicating with a payment gateway.

Applies to	*property or method*
Options	• message
Class	*Luhn*[2]
Validator	*LuhnValidator*[3]

Basic Usage

To use the Luhn validator, simply apply it to a property on an object that will contain a credit card number.

Listing 78-1
```php
// src/Acme/SubscriptionBundle/Entity/Transaction.php
namespace Acme\SubscriptionBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\Luhn(message = "Please check your credit card number.")
     */
```

1. http://en.wikipedia.org/wiki/Luhn_algorithm
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Luhn.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/LuhnValidator.html

```
11      protected $cardNumber;
12  }
```

Available Options

message

type: `string` **default**: `Invalid card number.`

The default message supplied when the value does not pass the Luhn check.

Chapter 79
Iban

New in version 2.3: The Iban constraint was introduced in Symfony 2.3.

This constraint is used to ensure that a bank account number has the proper format of an *International Bank Account Number (IBAN)*[1]. IBAN is an internationally agreed means of identifying bank accounts across national borders with a reduced risk of propagating transcription errors.

Applies to	*property or method*
Options	• message
Class	*Iban*[2]
Validator	*IbanValidator*[3]

Basic Usage

To use the Iban validator, simply apply it to a property on an object that will contain an International Bank Account Number.

Listing 79-1

```
// src/Acme/SubscriptionBundle/Entity/Transaction.php
namespace Acme\SubscriptionBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Transaction
{
    /**
     * @Assert\Iban(message = "This is not a valid International Bank Account Number
(IBAN).")
```

1. http://en.wikipedia.org/wiki/International_Bank_Account_Number
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Iban.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IbanValidator.html

```
11          */
12      protected $bankAccountNumber;
    }
```

Available Options

message

type: `string` **default**: `This is not a valid International Bank Account Number (IBAN).`

The default message supplied when the value does not pass the Iban check.

Chapter 80
Isbn

New in version 2.3: The Isbn constraint was introduced in Symfony 2.3.

This constraint validates that an *International Standard Book Number (ISBN)*[1] is either a valid ISBN-10, a valid ISBN-13 or both.

Applies to	*property or method*
Options	• isbn10 • isbn13 • isbn10Message • isbn13Message • bothIsbnMessage
Class	*Isbn*[2]
Validator	*IsbnValidator*[3]

Basic Usage

To use the `Isbn` validator, simply apply it to a property or method on an object that will contain a ISBN number.

Listing 80-1
```php
// src/Acme/BookcaseBundle/Entity/Book.php
namespace Acme\BookcaseBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Book
{
```

1. http://en.wikipedia.org/wiki/Isbn
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Isbn.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IsbnValidator.html

```
8       /**
9        * @Assert\Isbn(
10       *     isbn10 = true,
11       *     isbn13 = true,
12       *     bothIsbnMessage = "This value is neither a valid ISBN-10 nor a valid ISBN-13."
13       * )
14       */
15      protected $isbn;
16  }
```

Available Options

isbn10

type: `boolean`

If this required option is set to `true` the constraint will check if the code is a valid ISBN-10 code.

isbn13

type: `boolean`

If this required option is set to `true` the constraint will check if the code is a valid ISBN-13 code.

isbn10Message

type: `string` **default**: `This value is not a valid ISBN-10.`

The message that will be shown if the isbn10 option is true and the given value does not pass the ISBN-10 check.

isbn13Message

type: `string` **default**: `This value is not a valid ISBN-13.`

The message that will be shown if the isbn13 option is true and the given value does not pass the ISBN-13 check.

bothIsbnMessage

type: `string` **default**: `This value is neither a valid ISBN-10 nor a valid ISBN-13.`

The message that will be shown if both the isbn10 and isbn13 options are true and the given value does not pass the ISBN-13 nor the ISBN-13 check.

Chapter 81

Issn

New in version 2.3: The Issn constraint was introduced in Symfony 2.3.

Validates that a value is a valid *International Standard Serial Number (ISSN)*[1].

Applies to	*property or method*
Options	• message • caseSensitive • requireHyphen
Class	*Issn*[2]
Validator	*IssnValidator*[3]

Basic Usage

Listing 81-1

```php
// src/Acme/JournalBundle/Entity/Journal.php
namespace Acme\JournalBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Journal
{
    /**
     * @Assert\Issn
     */
     protected $issn;
}
```

1. http://en.wikipedia.org/wiki/Issn
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Issn.html
3. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/IssnValidator.html

Options

message

type: `String` default: `This value is not a valid ISSN.`

The message shown if the given value is not a valid ISSN.

caseSensitive

type: `Boolean` default: `false`

The validator will allow ISSN values to end with a lower case 'x' by default. When switching this to `true`, the validator requires an upper case 'X'.

requireHyphen

type: `Boolean` default: `false`

The validator will allow non hyphenated ISSN values by default. When switching this to `true`, the validator requires a hyphenated ISSN value.

Chapter 82
Callback

The purpose of the Callback assertion is to let you create completely custom validation rules and to assign any validation errors to specific fields on your object. If you're using validation with forms, this means that you can make these custom errors display next to a specific field, instead of simply at the top of your form.

This process works by specifying one or more *callback* methods, each of which will be called during the validation process. Each of those methods can do anything, including creating and assigning validation errors.

A callback method itself doesn't *fail* or return any value. Instead, as you'll see in the example, a callback method has the ability to directly add validator "violations".

Applies to	*class*
Options	• methods
Class	*Callback*[1]
Validator	*CallbackValidator*[2]

Setup

Listing 82-1

```
1  // src/Acme/BlogBundle/Entity/Author.php
2  namespace Acme\BlogBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Callback.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/CallbackValidator.html

```
5
6  /**
7   * @Assert\Callback(methods={"isAuthorValid"})
8   */
9  class Author
10 {
11 }
```

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 82-2

```
1  // ...
2  use Symfony\Component\Validator\ExecutionContextInterface;
3
4  class Author
5  {
6      // ...
7      private $firstName;
8
9      public function isAuthorValid(ExecutionContextInterface $context)
10     {
11         // somehow you have an array of "fake names"
12         $fakeNames = array();
13
14         // check if the name is actually a fake name
15         if (in_array($this->getFirstName(), $fakeNames)) {
16             $context->addViolationAt('firstname', 'This name sounds totally fake!',
17 array(), null);
18         }
19     }
   }
```

Options

methods

type: `array` **default**: `array()` [*default option*]

This is an array of the methods that should be executed during the validation process. Each method can be one of the following formats:

1. **String method name**

   > If the name of a method is a simple string (e.g. `isAuthorValid`), that method will be called on the same object that's being validated and the `ExecutionContextInterface` will be the only argument (see the above example).

2. **Static array callback**

Each method can also be specified as a standard array callback:

Listing 82-3
```php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Callback(methods={
 *     { "Acme\BlogBundle\MyStaticValidatorClass", "isAuthorValid" }
 * })
 */
class Author
{
}
```

In this case, the static method `isAuthorValid` will be called on the `Acme\BlogBundle\MyStaticValidatorClass` class. It's passed both the original object being validated (e.g. `Author`) as well as the `ExecutionContextInterface`:

Listing 82-4
```php
namespace Acme\BlogBundle;

use Symfony\Component\Validator\ExecutionContextInterface;
use Acme\BlogBundle\Entity\Author;

class MyStaticValidatorClass
{
    public static function isAuthorValid(Author $author,
ExecutionContextInterface $context)
    {
        // ...
    }
}
```

If you specify your `Callback` constraint via PHP, then you also have the option to make your callback either a PHP closure or a non-static callback. It is *not* currently possible, however, to specify a *service* as a constraint. To validate using a service, you should *create a custom validation constraint* and add that new constraint to your class.

Chapter 83

All

When applied to an array (or Traversable object), this constraint allows you to apply a collection of constraints to each element of the array.

Applies to	*property or method*
Options	• constraints
Class	*All*[1]
Validator	*AllValidator*[2]

Basic Usage

Suppose that you have an array of strings, and you want to validate each entry in that array:

Listing 83-1

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\All({
     *     @Assert\NotBlank,
     *     @Assert\Length(min = 5)
     * })
     */
     protected $favoriteColors = array();
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/All.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/AllValidator.html

Now, each entry in the `favoriteColors` array will be validated to not be blank and to be at least 5 characters long.

Options

constraints

type: `array` [*default option*]

This required option is the array of validation constraints that you want to apply to each element of the underlying array.

Chapter 84
UserPassword

Since Symfony 2.2, the `UserPassword*` classes in the *`Symfony\Component\Security\Core\Validator\Constraint`*[1] namespace are deprecated and will be removed in Symfony 2.3. Please use the `UserPassword*` classes in the *`Symfony\Component\Security\Core\Validator\Constraints`*[2] namespace instead.

This validates that an input value is equal to the current authenticated user's password. This is useful in a form where a user can change their password, but needs to enter their old password for security.

This should **not** be used to validate a login form, since this is done automatically by the security system.

Applies to	*property or method*
Options	• message
Class	*`UserPassword`*[3]
Validator	*`UserPasswordValidator`*[4]

Basic Usage

Suppose you have a *PasswordChange* 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:

1. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Validator/Constraint.html
2. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Validator/Constraints.html
3. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Validator/Constraints/UserPassword.html
4. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Validator/Constraints/UserPasswordValidator.html

Listing 84-1

```php
// src/Acme/UserBundle/Form/Model/ChangePassword.php
namespace Acme\UserBundle\Form\Model;

use Symfony\Component\Security\Core\Validator\Constraints as SecurityAssert;

class ChangePassword
{
    /**
     * @SecurityAssert\UserPassword(
     *     message = "Wrong value for your current password"
     * )
     */
     protected $oldPassword;
}
```

Options

message

type: `message` **default**: `This value should be the user current password.`

This is the message that's displayed when the underlying string does *not* match the current user's password.

Chapter 85
Valid

This constraint is used to enable validation on objects that are embedded as properties on an object being validated. This allows you to validate an object and all sub-objects associated with it.

Applies to	*property or method*
Options	• traverse • deep
Class	*Valid*[1]

By default the `error_bubbling` option is enabled for the *collection Field Type*, which passes the errors to the parent form. If you want to attach the errors to the locations where they actually occur you have to set `error_bubbling` to `false`.

Basic Usage

In the following example, create two classes `Author` and `Address` that both have constraints on their properties. Furthermore, `Author` stores an `Address` instance in the `$address` property.

Listing 85-1
```php
// src/Acme/HelloBundle/Entity/Address.php
namespace Acme\HelloBundle\Entity;

class Address
{
    protected $street;
    protected $zipCode;
}
```

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraints/Valid.html

Listing 85-2
```php
// src/Acme/HelloBundle/Entity/Author.php
namespace Acme\HelloBundle\Entity;

class Author
{
    protected $firstName;
    protected $lastName;
    protected $address;
}
```

Listing 85-3
```php
// src/Acme/HelloBundle/Entity/Address.php
namespace Acme\HelloBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Address
{
    /**
     * @Assert\NotBlank()
     */
    protected $street;

    /**
     * @Assert\NotBlank
     * @Assert\Length(max = 5)
     */
    protected $zipCode;
}

// src/Acme/HelloBundle/Entity/Author.php
namespace Acme\HelloBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank
     * @Assert\Length(min = 4)
     */
    protected $firstName;

    /**
     * @Assert\NotBlank
     */
    protected $lastName;

    protected $address;
}
```

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 85-4
```php
// src/Acme/HelloBundle/Entity/Author.php
namespace Acme\HelloBundle\Entity;

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 85-5
```
1 Acme\\HelloBundle\\Author.address.zipCode:
2     This value is too long. It should have 5 characters or less.
```

Options

traverse

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

If this constraint is applied to a property that holds an array of objects, then each object in that array will be validated only if this option is set to `true`.

deep

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

If this constraint is applied to a property that holds an array of objects, then each object in that array will be validated recursively if this option is set to `true`.

Chapter 86

Twig Template Form Function and Variable Reference

When working with forms in a template, there are two powerful things at your disposal:

- *Functions* for rendering each part of a form
- *Variables* for getting *any* information about any field

You'll use functions often to render your fields. Variables, on the other hand, are less commonly-used, but infinitely powerful since you can access a fields label, id attribute, errors, and anything else about the field.

Form Rendering Functions

This reference manual covers all the possible Twig functions available for rendering forms. There are several different functions available, and each is responsible for rendering a different part of a form (e.g. labels, errors, widgets, etc).

form(view, variables)

Renders the HTML of a complete form.

Listing 86-1
```
{# render the form and change the submission method #}
{{ form(form, {'method': 'GET'}) }}
```

You will mostly use this helper for prototyping or if you use custom form themes. If you need more flexibility in rendering the form, you should use the other helpers to render individual parts of the form instead:

Listing 86-2
```
{{ form_start(form) }}
    {{ form_errors(form) }}
```

```
3
4     {{ form_row(form.name) }}
5     {{ form_row(form.dueDate) }}
6
7     {{ form_row(form.submit, { 'label': 'Submit me' }) }}
8 {{ form_end(form) }}
```

form_start(view, variables)

Renders the start tag of a form. This helper takes care of printing the configured method and target action of the form. It will also include the correct `enctype` property if the form contains upload fields.

Listing 86-3
```
1 {# render the start tag and change the submission method #}
2 {{ form_start(form, {'method': 'GET'}) }}
```

form_end(view, variables)

Renders the end tag of a form.

Listing 86-4
```
1 {{ form_end(form) }}
```

This helper also outputs `form_rest()` unless you set `render_rest` to false:

Listing 86-5
```
1 {# don't render unrendered fields #}
2 {{ form_end(form, {'render_rest': false}) }}
```

form_label(view, label, variables)

Renders the label for the given field. You can optionally pass the specific label you want to display as the second argument.

Listing 86-6
```
1 {{ form_label(form.name) }}
2
3 {# The two following syntaxes are equivalent #}
4 {{ form_label(form.name, 'Your Name', {'label_attr': {'class': 'foo'}}) }}
5 {{ form_label(form.name, null, {'label': 'Your name', 'label_attr': {'class': 'foo'}}) }}
```

See "*More about Form Variables*" to learn about the `variables` argument.

form_errors(view)

Renders any errors for the given field.

Listing 86-7
```
1 {{ form_errors(form.name) }}
2
```

```
{# render any "global" errors #}
{{ form_errors(form) }}
```

form_widget(view, variables)

Renders the HTML widget of a given field. If you apply this to an entire form or collection of fields, each underlying form row will be rendered.

Listing 86-8
```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, {'attr': {'class': 'foo'}}) }}
```

The second argument to `form_widget` is an array of variables. The most common variable is `attr`, which is an array of HTML attributes to apply to the HTML widget. In some cases, certain types also have other template-related options that can be passed. These are discussed on a type-by-type basis. The `attributes` are not applied recursively to child fields if you're rendering many fields at once (e.g. `form_widget(form)`).

See "*More about Form Variables*" to learn more about the `variables` argument.

form_row(view, variables)

Renders the "row" of a given field, which is the combination of the field's label, errors and widget.

Listing 86-9
```
{# render a field row, but display a label with text "foo" #}
{{ form_row(form.name, {'label': 'foo'}) }}
```

The second argument to `form_row` is an array of variables. The templates provided in Symfony only allow to override the label as shown in the example above.

See "*More about Form Variables*" to learn about the `variables` argument.

form_rest(view, variables)

This renders all fields that have not yet been rendered for the given form. It's a good idea to always have this somewhere inside your form as it'll render hidden fields for you and make any fields you forgot to render more obvious (since it'll render the field for you).

Listing 86-10
```
{{ form_rest(form) }}
```

form_enctype(view)

This helper was deprecated in Symfony 2.3 and will be removed in Symfony 3.0. You should use `form_start()` instead.

If the form contains at least one file upload field, this will render the required `enctype="multipart/form-data"` form attribute. It's always a good idea to include this in your form tag:

Listing 86-11
```twig
<form action="{{ path('form_submit') }}" method="post" {{ form_enctype(form) }}>
```

Form Tests Reference

Tests can be executed by using the `is` operator in Twig to create a condition. Read *the Twig documentation*[1] for more information.

selectedchoice(selected_value)

This test will check if the current choice is equal to the `selected_value` or if the current choice is in the array (when `selected_value` is an array).

Listing 86-12
```twig
<option {% if choice is selectedchoice(value) %} selected="selected"{% endif %} ...>
```

More about Form Variables

For a full list of variables, see: *Form Variables Reference*.

In almost every Twig function above, the final argument is an array of "variables" that are used when rendering that one part of the form. For example, the following would render the "widget" for a field, and modify its attributes to include a special class:

Listing 86-13
```twig
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

The purpose of these variables - what they do & where they come from - may not be immediately clear, but they're incredibly powerful. Whenever you render any part of a form, the block that renders it makes use of a number of variables. By default, these blocks live inside *form_div_layout.html.twig*[2].

Look at the `form_label` as an example:

Listing 86-14
```twig
{% block form_label %}
    {% if not compound %}
        {% set label_attr = label_attr|merge({'for': id}) %}
    {% endif %}
    {% if required %}
        {% set label_attr = label_attr|merge({'class': (label_attr.class|default('') ~ '
required')|trim}) %}
    {% endif %}
    {% if label is empty %}
        {% set label = name|humanize %}
    {% endif %}
    <label{% for attrname, attrvalue in label_attr %} {{ attrname }}="{{ attrvalue }}"{%
```

1. http://twig.sensiolabs.org/doc/templates.html#test-operator
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
endfor %}>{{ label|trans({}, translation_domain) }}</label>
{% endblock form_label %}
```

This block makes use of several variables: `compound`, `label_attr`, `required`, `label`, `name` and `translation_domain`. These variables are made available by the form rendering system. But more importantly, these are the variables that you can override when calling `form_label` (since in this example, you're rendering the label).

The exact variables available to override depends on which part of the form you're rendering (e.g. label versus widget) and which field you're rendering (e.g. a `choice` widget has an extra `expanded` option). If you get comfortable with looking through *form_div_layout.html.twig*[3], you'll always be able to see what options you have available.

Behind the scenes, these variables are made available to the `FormView` object of your form when the Form component calls `buildView` and `finishView` on each "node" of your form tree. To see what "view" variables a particular field has, find the source code for the form field (and its parent fields) and look at the above two functions.

If you're rendering an entire form at once (or an entire embedded form), the `variables` argument will only be applied to the form itself and not its children. In other words, the following will **not** pass a "foo" class attribute to all of the child fields in the form:

Listing 86-15
```
{# does **not** work - the variables are not recursive #}
{{ form_widget(form, { 'attr': {'class': 'foo'} }) }}
```

Form Variables Reference

The following variables are common to every field type. Certain field types may have even more variables and some variables here only really apply to certain types.

Assuming you have a `form` variable in your template, and you want to reference the variables on the `name` field, accessing the variables is done by using a public `vars` property on the *`FormView`*[4] object:

Listing 86-16
```
<label for="{{ form.name.vars.id }}"
    class="{{ form.name.vars.required ? 'required' : '' }}">
    {{ form.name.vars.label }}
</label>
```

New in version 2.3: The `method` and `action` variables were introduced in Symfony 2.3.

Variable	Usage
`form`	The current `FormView` instance.
`id`	The `id` HTML attribute to be rendered.
`name`	The name of the field (e.g. `title`) - but not the `name` HTML attribute, which is `full_name`.
`full_name`	The `name` HTML attribute to be rendered.

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
4. http://api.symfony.com/2.3/Symfony/Component/Form/FormView.html

Variable	Usage
`errors`	An array of any errors attached to *this* specific field (e.g. `form.title.errors`). Note that you can't use `form.errors` to determine if a form is valid, since this only returns "global" errors: some individual fields may have errors. Instead, use the `valid` option.
`valid`	Returns `true` or `false` depending on whether the whole form is valid.
`value`	The value that will be used when rendering (commonly the `value` HTML attribute).
`read_only`	If `true`, `readonly="readonly"` is added to the field.
`disabled`	If `true`, `disabled="disabled"` is added to the field.
`required`	If `true`, a `required` attribute is added to the field to activate HTML5 validation. Additionally, a `required` class is added to the label.
`max_length`	Adds a `maxlength` HTML attribute to the element.
`pattern`	Adds a `pattern` HTML attribute to the element.
`label`	The string label that will be rendered.
`multipart`	If `true`, `form_enctype` will render `enctype="multipart/form-data"`. This only applies to the root form element.
`attr`	A key-value array that will be rendered as HTML attributes on the field.
`label_attr`	A key-value array that will be rendered as HTML attributes on the label.
`compound`	Whether or not a field is actually a holder for a group of children fields (for example, a `choice` field, which is actually a group of checkboxes.
`block_prefixes`	An array of all the names of the parent types.
`translation_domain`	The domain of the translations for this form.
`cache_key`	A unique key which is used for caching.
`data`	The normalized data of the type.
`method`	The method of the current form (POST, GET, etc.).
`action`	The action of the current form.

Chapter 87
Symfony Twig Extensions

Twig is the default template engine for Symfony. By itself, it already contains a lot of built-in functions, filters, tags and tests (learn more about them from the *Twig Reference*[1]).

Symfony adds custom extensions on top of Twig to integrate some components into the Twig templates. The following sections describe the custom *functions*, *filters*, *tags* and *tests* that are available when using the Symfony Core Framework.

There may also be tags in bundles you use that aren't listed here.

Functions

render

New in version 2.2: The `render()` function was introduced in Symfony 2.2. Prior, the `{% render %}` tag was used and had a different signature.

Listing 87-1
```
1 {{ render(uri, options) }}
```

`uri`
type: `string` | `ControllerReference`

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

Renders the fragment for the given controller (using the controller function) or URI. For more information, see *Embedding Controllers*.

The render strategy can be specified in the `strategy` key of the options.

1. `http://twig.sensiolabs.org/documentation#reference`

The URI can be generated by other functions, like path and url.

render_esi

Listing 87-2
```
{{ render_esi(uri, options) }}
```

uri
type: `string` | `ControllerReference`

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

Generates an ESI tag when possible or falls back to the behavior of render function instead. For more information, see *Embedding Controllers*.

The URI can be generated by other functions, like path and url.

The `render_esi()` function is an example of the shortcut functions of `render`. It automatically sets the strategy based on what's given in the function name, e.g. `render_hinclude()` will use the hinclude.js strategy. This works for all `render_*()` functions.

controller

New in version 2.2: The `controller()` function was introduced in Symfony 2.2.

Listing 87-3
```
{{ controller(controller, attributes, query) }}
```

controller
type: `string`

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

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

Returns an instance of `ControllerReference` to be used with functions like *render()* and *render_esi()*.

asset

Listing 87-4
```
{{ asset(path, packageName) }}
```

path
type: `string`

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

Returns a public path to `path`, which takes into account the base path set for the package and the URL path. More information in *Linking to Assets*.

assets_version

Listing 87-5
```
{{ assets_version(packageName) }}
```

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

Returns the current version of the package, more information in *Linking to Assets*.

form

Listing 87-6
```
{{ form(view, variables) }}
```

view
type: `FormView`

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

Renders the HTML of a complete form, more information in *the Twig Form reference*.

form_start

Listing 87-7
```
{{ form_start(view, variables) }}
```

view
type: `FormView`

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

Renders the HTML start tag of a form, more information in *the Twig Form reference*.

form_end

Listing 87-8
```
{{ form_end(view, variables) }}
```

view
type: `FormView`

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

Renders the HTML end tag of a form together with all fields that have not been rendered yet, more information in *the Twig Form reference*.

form_enctype

Listing 87-9
```
{{ form_enctype(view) }}
```

view
type: FormView

Renders the required `enctype="multipart/form-data"` attribute if the form contains at least one file upload field, more information in *the Twig Form reference*.

form_widget

Listing 87-10
```
{{ form_widget(view, variables) }}
```

view
type: FormView

variables
type: array **default**: []

Renders a complete form or a specific HTML widget of a field, more information in *the Twig Form reference*.

form_errors

Listing 87-11
```
{{ form_errors(view) }}
```

view
type: FormView

Renders any errors for the given field or the global errors, more information in *the Twig Form reference*.

form_label

Listing 87-12
```
{{ form_label(view, label, variables) }}
```

view
type: FormView

label
type: string **default**: null

variables
type: array **default**: []

Renders the label for the given field, more information in *the Twig Form reference*.

form_row

Listing 87-13
```
{{ form_row(view, variables) }}
```

view
type: `FormView`

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

Renders the row (the field's label, errors and widget) of the given field, more information in *the Twig Form reference*.

form_rest

Listing 87-14
```
{{ form_rest(view, variables) }}
```

view
type: `FormView`

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

Renders all fields that have not yet been rendered, more information in *the Twig Form reference*.

csrf_token

Listing 87-15
```
{{ csrf_token(intention) }}
```

intention
type: `string`

Renders a CSRF token. Use this function if you want CSRF protection without creating a form.

is_granted

Listing 87-16
```
{{ is_granted(role, object, field) }}
```

role
type: `string`

object
type: `object`

field
type: `string`

Returns `true` if the current user has the required role. Optionally, an object can be pasted to be used by the voter. More information can be found in *Access Control in Templates*.

You can also pass in the field to use ACE for a specific field. Read more about this in *Scope of Access Control Entries*.

logout_path

Listing 87-17
```
{{ logout_path(key) }}
```

key
type: `string`

Generates a relative logout URL for the given firewall.

logout_url

Listing 87-18
```
{{ logout_url(key) }}
```

key
type: `string`

Equal to the logout_path function, but it'll generate an absolute URL instead of a relative one.

path

Listing 87-19
```
{{ path(name, parameters, relative) }}
```

name
type: `string`

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

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

Returns the relative URL (without the scheme and host) for the given route. If `relative` is enabled, it'll create a path relative to the current path. More information in *Linking to Pages*.

url

Listing 87-20
```
{{ url(name, parameters, schemeRelative) }}
```

name
type: `string`

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

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

Returns the absolute URL (with scheme and host) for the given route. If `schemeRelative` is enabled, it'll create a scheme-relative URL. More information in *Linking to Pages*.

Filters

humanize

New in version 2.1: The `humanize` filter was introduced in Symfony 2.1

Listing 87-21
```
1  {{ text|humanize }}
```

`text`
type: `string`

Makes a technical name human readable (i.e. replaces underscores by spaces and capitalizes the string).

trans

Listing 87-22
```
1  {{ message|trans(arguments, domain, locale) }}
```

`message`
type: `string`

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

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

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

Translates the text into the current language. More information in *Translation Filters*.

transchoice

Listing 87-23
```
1  {{ message|transchoice(count, arguments, domain, locale) }}
```

`message`
type: `string`

`count`
type: `integer`

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

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

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

Translates the text with pluralization support. More information in *Translation Filters*.

yaml_encode

Listing 87-24
```
{{ input|yaml_encode(inline, dumpObjects) }}
```

input
    type: mixed

inline
    type: integer **default**: 0

dumpObjects
    type: boolean **default**: false

Transforms the input into YAML syntax. See *Writing YAML Files* for more information.

yaml_dump

Listing 87-25
```
{{ value|yaml_dump(inline, dumpObjects) }}
```

value
    type: mixed

inline
    type: integer **default**: 0

dumpObjects
    type: boolean **default**: false

Does the same as *yaml_encode()*[2], but includes the type in the output.

abbr_class

Listing 87-26
```
{{ 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 87-27
```
{{ 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()`).

2. #reference-yaml_encode

format_args

Listing 87-28
```
{{ args|format_args }}
```

args
type: `array`

Generates a string with the arguments and their types (within `<em>` elements).

format_args_as_text

Listing 87-29
```
{{ args|format_args_as_text }}
```

args
type: `array`

Equal to the format_args filter, but without using HTML tags.

file_excerpt

Listing 87-30
```
{{ file|file_excerpt(line) }}
```

file
type: `string`

line
type: `integer`

Generates an excerpt of seven lines around the given `line`.

format_file

Listing 87-31
```
{{ file|format_file(line, text) }}
```

file
type: `string`

line
type: `integer`

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

Generates the file path inside an `<a>` element. If the path is inside the kernel root directory, the kernel root directory path is replaced by `kernel.root_dir` (showing the full path in a tooltip on hover).

format_file_from_text

Listing 87-32
```
{{ text|format_file_from_text }}
```

`text`
: **type**: `string`

Uses format_file to improve the output of default PHP errors.

file_link

Listing 87-33
```
{{ file|file_link(line) }}
```

`line`
: **type**: `integer`

Generates a link to the provided file (and optionally line number) using a preconfigured scheme.

Tags

form_theme

Listing 87-34
```
{% 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 87-35
```
{% trans with vars from domain into locale %}{% endtrans %}
```

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

`domain`
: **type**: `string` **default**: `string`

`locale`
: **type**: `string` **default**: `string`

Renders the translation of the content. More information in *Twig Templates*.

transchoice

Listing 87-36
```
{% transchoice count with vars from domain into locale %}{% endtranschoice %}
```

`count`
: **type**: `integer`

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

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

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

Renders the translation of the content with pluralization support, more information in *Twig Templates*.

trans_default_domain

Listing 87-37
```
{% trans_default_domain domain %}
```

`domain`
: **type**: `string`

This will set the default domain in the current template.

Tests

selectedchoice

Listing 87-38
```
{% if choice is selectedchoice(selectedValue) %}
```

`choice`
: **type**: `ChoiceView`

`selectedValue`
: **type**: `string`

Checks if `selectedValue` was checked for the provided choice field. Using this test is the most effective way.

Global Variables

app

The `app` variable is available everywhere and gives access to many commonly needed objects and values. It is an instance of *`GlobalVariables`*[3].

The available attributes are:

- `app.user`
- `app.request`
- `app.session`
- `app.environment`
- `app.debug`

3. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html

- `app.security`

Symfony Standard Edition Extensions

The Symfony Standard Edition adds some bundles to the Symfony Core Framework. Those bundles can have other Twig extensions:

- **Twig Extensions** includes some interesting extensions that do not belong to the Twig core. You can read more in *the official Twig Extensions documentation*[4];
- **Assetic** adds the `{% stylesheets %}`, `{% javascripts %}` and `{% image %}` tags. You can read more about them in *the Assetic Documentation*.

4. `http://twig.sensiolabs.org/doc/extensions/index.html`

Chapter 88
The Dependency Injection Tags

Dependency Injection Tags are little strings that can be applied to a service to "flag" it to be used in some special way. For example, if you have a service that you would like to register as a listener to one of Symfony's core events, you can flag it with the `kernel.event_listener` tag.

You can learn a little bit more about "tags" by reading the "*Tags*" section of the Service Container chapter.

Below is information about all of the tags available inside Symfony. There may also be tags in other bundles you use that aren't listed here.

Tag Name	Usage
assetic.asset	Register an asset to the current asset manager
assetic.factory_worker	Add a factory worker
assetic.filter	Register a filter
assetic.formula_loader	Add a formula loader to the current asset manager
assetic.formula_resource	Adds a resource to the current asset manager
assetic.templating.php	Remove this service if PHP templating is disabled
assetic.templating.twig	Remove this service if Twig templating is disabled
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

Tag Name	Usage
kernel.event_subscriber	To subscribe to a set of different events/hooks in Symfony
kernel.fragment_renderer	Add new HTTP content rendering strategies
monolog.logger	Logging with a custom logging channel
monolog.processor	Add a custom processor for logging
routing.loader	Register a custom service that loads routes
security.voter	Add a custom voter to Symfony's authorization logic
security.remember_me_aware	To allow remember me authentication
serializer.encoder	Register a new encoder in the `serializer` service
serializer.normalizer	Register a new normalizer in the `serializer` service
swiftmailer.default.plugin	Register a custom SwiftMailer Plugin
templating.helper	Make your service available in PHP templates
translation.loader	Register a custom service that loads translations
translation.extractor	Register a custom service that extracts translation messages from a file
translation.dumper	Register a custom service that dumps translation messages
twig.extension	Register a custom Twig Extension
twig.loader	Register a custom service that loads Twig templates
validator.constraint_validator	Create your own custom validation constraint
validator.initializer	Register a service that initializes objects before validation

assetic.asset

Purpose: Register an asset with the current asset manager

assetic.factory_worker

Purpose: Add a factory worker

A Factory worker is a class implementing `Assetic\Factory\Worker\WorkerInterface`. Its `process($asset)` method is called for each asset after asset creation. You can modify an asset or even return a new one.

In order to add a new worker, first create a class:

Listing 88-1
```php
use Assetic\Asset\AssetInterface;
use Assetic\Factory\Worker\WorkerInterface;

class MyWorker implements WorkerInterface
{
    public function process(AssetInterface $asset)
    {
        // ... change $asset or return a new one
    }
```

```
10
11 }
```

And then register it as a tagged service:

Listing 88-2
```
1 services:
2     acme.my_worker:
3         class: MyWorker
4         tags:
5             - { name: assetic.factory_worker }
```

assetic.filter

Purpose: Register a filter

AsseticBundle uses this tag to register common filters. You can also use this tag to register your own filters.

First, you need to create a filter:

Listing 88-3
```
1  use Assetic\Asset\AssetInterface;
2  use Assetic\Filter\FilterInterface;
3
4  class MyFilter implements FilterInterface
5  {
6      public function filterLoad(AssetInterface $asset)
7      {
8          $asset->setContent('alert("yo");' . $asset->getContent());
9      }
10
11     public function filterDump(AssetInterface $asset)
12     {
13         // ...
14     }
15 }
```

Second, define a service:

Listing 88-4
```
1 services:
2     acme.my_filter:
3         class: MyFilter
4         tags:
5             - { name: assetic.filter, alias: my_filter }
```

Finally, apply the filter:

Listing 88-5
```
1 {% javascripts
2     '@AcmeBaseBundle/Resources/public/js/global.js'
3     filter='my_filter'
4 %}
5     <script src="{{ asset_url }}"></script>
6 {% endjavascripts %}
```

You can also apply your filter via the `assetic.filters.my_filter.apply_to` config option as it's described here: *How to Apply an Assetic Filter to a specific File Extension*. In order to do that, you

must define your filter service in a separate xml config file and point to this file's path via the `assetic.filters.my_filter.resource` configuration key.

assetic.formula_loader

Purpose: Add a formula loader to the current asset manager

A Formula loader is a class implementing `Assetic\\Factory\Loader\\FormulaLoaderInterface` interface. This class is responsible for loading assets from a particular kind of resources (for instance, twig template). Assetic ships loaders for PHP and Twig templates.

An `alias` attribute defines the name of the loader.

assetic.formula_resource

Purpose: Adds a resource to the current asset manager

A resource is something formulae can be loaded from. For instance, Twig templates are resources.

assetic.templating.php

Purpose: Remove this service if PHP templating is disabled

The tagged service will be removed from the container if the `framework.templating.engines` config section does not contain php.

assetic.templating.twig

Purpose: Remove this service if Twig templating is disabled

The tagged service will be removed from the container if `framework.templating.engines` config section does not contain `twig`.

data_collector

Purpose: Create a class that collects custom data for the profiler

For details on creating your own custom data collection, read the cookbook article: *How to Create a custom Data Collector*.

doctrine.event_listener

Purpose: Add a Doctrine event listener

For details on creating Doctrine event listeners, read the cookbook article: *How to Register Event Listeners and Subscribers*.

doctrine.event_subscriber

Purpose: Add a Doctrine event subscriber

For details on creating Doctrine event subscribers, read the cookbook article: *How to Register Event Listeners and Subscribers*.

form.type

Purpose: Create a custom form field type

For details on creating your own custom form type, read the cookbook article: *How to Create a Custom Form Field Type*.

form.type_extension

Purpose: Create a custom "form extension"

Form type extensions are a way for you took "hook into" the creation of any field in your form. For example, the addition of the CSRF token is done via a form type extension (*FormTypeCsrfExtension*[1]).

A form type extension can modify any part of any field in your form. To create a form type extension, first create a class that implements the *FormTypeExtensionInterface*[2] interface. For simplicity, you'll often extend an *AbstractTypeExtension*[3] class instead of the interface directly:

Listing 88-6
```php
// src/Acme/MainBundle/Form/Type/MyFormTypeExtension.php
namespace Acme\MainBundle\Form\Type;

use Symfony\Component\Form\AbstractTypeExtension;

class MyFormTypeExtension extends AbstractTypeExtension
{
    // ... fill in whatever methods you want to override
    // like buildForm(), buildView(), finishView(), setDefaultOptions()
}
```

In order for Symfony to know about your form extension and use it, give it the `form.type_extension` tag:

Listing 88-7
```yaml
services:
    main.form.type.my_form_type_extension:
        class: Acme\MainBundle\Form\Type\MyFormTypeExtension
        tags:
            - { name: form.type_extension, alias: field }
```

The `alias` key of the tag is the type of field that this extension should be applied to. For example, to apply the extension to any form/field, use the "form" value.

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).

---

1. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Csrf/Type/FormTypeCsrfExtension.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeExtensionInterface.html
3. http://api.symfony.com/2.3/Symfony/Component/Form/AbstractTypeExtension.html

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 88-8
```php
// src/Acme/MainBundle/Cache/MyClearer.php
namespace Acme\MainBundle\Cache;

use Symfony\Component\HttpKernel\CacheClearer\CacheClearerInterface;

class MyClearer implements CacheClearerInterface
{
    public function clear($cacheDir)
    {
        // clear your cache
    }

}
```

Then register this class and tag it with `kernel.cache_clearer`:

Listing 88-9
```yaml
services:
    my_cache_clearer:
        class: Acme\MainBundle\Cache\MyClearer
        tags:
            - { name: 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` task (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`*[4] interface:

Listing 88-10
```php
// src/Acme/MainBundle/Cache/MyCustomWarmer.php
namespace Acme\MainBundle\Cache;

use Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerInterface;

class MyCustomWarmer implements CacheWarmerInterface
{
```

4. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/CacheWarmer/CacheWarmerInterface.html

```
    public function warmUp($cacheDir)
    {
        // ... do some sort of operations to "warm" your cache
    }

    public function isOptional()
    {
        return true;
    }
}
```

The `isOptional` method should return true if it's possible to use the application without calling this cache warmer. In Symfony, optional warmers are always executed by default (you can change this by using the `--no-optional-warmers` option when executing the command).

To register your warmer with Symfony, give it the `kernel.cache_warmer` tag:

Listing 88-11
```
services:
    main.warmer.my_custom_warmer:
        class: Acme\MainBundle\Cache\MyCustomWarmer
        tags:
            - { name: kernel.cache_warmer, priority: 0 }
```

The `priority` value is optional, and defaults to 0. The higher the priority, the sooner it gets executed.

Core Cache Warmers

Cache Warmer Class Name	Priority
TemplatePathsCacheWarmer[5]	20
RouterCacheWarmer[6]	0
TemplateCacheCacheWarmer[7]	0

kernel.event_listener

Purpose: To listen to different events/hooks in Symfony

This tag allows you to hook your own classes into Symfony's process at different points.

For a full example of this listener, read the *How to Create an Event Listener* cookbook entry.

For another practical example of a kernel listener, see the cookbook article: *How to Register a new Request Format and Mime Type*.

5. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/CacheWarmer/TemplatePathsCacheWarmer.html
6. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/CacheWarmer/RouterCacheWarmer.html
7. http://api.symfony.com/2.3/Symfony/Bundle/TwigBundle/CacheWarmer/TemplateCacheCacheWarmer.html

Core Event Listener Reference

When adding your own listeners, it might be useful to know about the other core Symfony listeners and their priorities.

All listeners listed here may not be listening depending on your environment, settings and bundles. Additionally, third-party bundles will bring in additional listeners not listed here.

kernel.request

Listener Class Name	Priority
ProfilerListener[8]	1024
TestSessionListener[9]	192
SessionListener[10]	128
RouterListener[11]	32
LocaleListener[12]	16
Firewall[13]	8

kernel.controller

Listener Class Name	Priority
RequestDataCollector[14]	0

kernel.response

Listener Class Name	Priority
EsiListener[15]	0
ResponseListener[16]	0
ResponseListener[17]	0
ProfilerListener[18]	-100
TestSessionListener[19]	-128
WebDebugToolbarListener[20]	-128

8. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
9. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/EventListener/TestSessionListener.html
10. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/EventListener/SessionListener.html
11. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/RouterListener.html
12. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/LocaleListener.html
13. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall.html
14. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/DataCollector/RequestDataCollector.html
15. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/EsiListener.html
16. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ResponseListener.html
17. http://api.symfony.com/2.3/Symfony/Bundle/SecurityBundle/EventListener/ResponseListener.html
18. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
19. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/EventListener/TestSessionListener.html
20. http://api.symfony.com/2.3/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html

Listener Class Name	Priority
StreamedResponseListener[21]	-1024

kernel.exception

Listener Class Name	Priority
ProfilerListener[22]	0
ExceptionListener[23]	-128

kernel.terminate

Listener Class Name	Priority
EmailSenderListener[24]	0

kernel.event_subscriber

Purpose: To subscribe to a set of different events/hooks in Symfony

To enable a custom subscriber, add it as a regular service in one of your configuration, and tag it with `kernel.event_subscriber`:

Listing 88-12
```
services:
    kernel.subscriber.your_subscriber_name:
        class: Fully\Qualified\Subscriber\Class\Name
        tags:
            - { name: kernel.event_subscriber }
```

Your service must implement the *EventSubscriberInterface*[25] interface.

If your service is created by a factory, you **MUST** correctly set the `class` parameter for this tag to work correctly.

kernel.fragment_renderer

Purpose: Add a new HTTP content rendering strategy

21. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/StreamedResponseListener.html
22. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
23. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
24. https://github.com/symfony/SwiftmailerBundle/blob/master/EventListener/EmailSenderListener.php
25. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

To add a new rendering strategy - in addition to the core strategies like `EsiFragmentRenderer` - create a class that implements *`FragmentRendererInterface`*[26], register it as a service, then tag it with `kernel.fragment_renderer`.

monolog.logger

Purpose: To use a custom logging channel with Monolog

Monolog allows you to share its handlers between several logging channels. The logger service uses the channel `app` but you can change the channel when injecting the logger in a service.

Listing 88-13
```yaml
services:
    my_service:
        class: Fully\Qualified\Loader\Class\Name
        arguments: ["@logger"]
        tags:
            - { name: monolog.logger, channel: acme }
```

If you use MonologBundle 2.4 or higher, you can configure custom channels in the configuration and retrieve the corresponding logger service from the service container directly (see *Configure Additional Channels without Tagged Services*).

monolog.processor

Purpose: Add a custom processor for logging

Monolog allows you to add processors in the logger or in the handlers to add extra data in the records. A processor receives the record as an argument and must return it after adding some extra data in the `extra` attribute of the record.

The built-in `IntrospectionProcessor` can be used to add the file, the line, the class and the method where the logger was triggered.

You can add a processor globally:

Listing 88-14
```yaml
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: 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 88-15
```yaml
services:
    my_service:
```

26. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Fragment/FragmentRendererInterface.html

```
3           class: Monolog\Processor\IntrospectionProcessor
4           tags:
5               - { 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 88-16
```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, channel: security }
```

You cannot use both the `handler` and `channel` attributes for the same tag as handlers are shared between all channels.

routing.loader

Purpose: Register a custom service that loads routes

To enable a custom routing loader, add it as a regular service in one of your configuration, and tag it with `routing.loader`:

Listing 88-17
```
services:
    routing.loader.your_loader_name:
        class: Fully\Qualified\Loader\Class\Name
        tags:
            - { name: routing.loader }
```

For more information, see *How to Create a custom Route Loader*.

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`*[27] and your custom authentication listener extends *`AbstractAuthenticationListener`*[28], then your custom authentication listener will automatically have this tagged applied and it will function automatically.

security.voter

Purpose: To add a custom voter to Symfony's authorization logic

27. http://api.symfony.com/2.3/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.html
28. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.html

When you call `isGranted` on Symfony's security context, 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 cookbook article: *How to Implement your own Voter to Blacklist IP Addresses*.

serializer.encoder

Purpose: Register a new encoder in the `serializer` service

The class that's tagged should implement the *`EncoderInterface`*[29] and *`DecoderInterface`*[30].

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`*[31] and *`DenormalizerInterface`*[32].

For more details, see *How to Use the Serializer*.

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*[33].

Several SwiftMailer plugins are core to Symfony and can be activated via different configuration. For details, see *SwiftmailerBundle Configuration ("swiftmailer")*.

templating.helper

Purpose: Make your service available in PHP templates

To enable a custom template helper, add it as a regular service in one of your configuration, tag it with `templating.helper` and define an `alias` attribute (the helper will be accessible via this alias in the templates):

Listing 88-18

29. http://api.symfony.com/2.3/Symfony/Component/Serializer/Encoder/EncoderInterface.html
30. http://api.symfony.com/2.3/Symfony/Component/Serializer/Encoder/DecoderInterface.html
31. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/NormalizerInterface.html
32. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/DenormalizerInterface.html
33. http://swiftmailer.org/docs/plugins.html

```yaml
services:
    templating.helper.your_helper_name:
        class: Fully\Qualified\Helper\Class\Name
        tags:
            - { 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 88-19
```yaml
services:
    main.translation.my_custom_loader:
        class: Acme\MainBundle\Translation\MyCustomLoader
        tags:
            - { name: translation.loader, alias: bin }
```

The `alias` option is required and very important: it defines the file "suffix" that will be used for the resource files that use this loader. For example, suppose you have some custom `bin` format that you need to load. If you have a `bin` file that contains French translations for the `messages` domain, then you might have a file `app/Resources/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

New in version 2.1: The ability to add message extractors was introduced in Symfony 2.1.

When executing the `translation:update` command, it uses extractors to extract translation messages from a file. By default, the Symfony framework has a *`TwigExtractor`*[34] and a *`PhpExtractor`*[35], which help to find and extract translation keys from Twig templates and PHP files.

You can create your own extractor by creating a class that implements *`ExtractorInterface`*[36] and tagging the service with `translation.extractor`. The tag has one required option: `alias`, which defines the name of the extractor:

Listing 88-20
```php
// src/Acme/DemoBundle/Translation/FooExtractor.php
namespace Acme\DemoBundle\Translation;
```

34. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Translation/TwigExtractor.html
35. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Translation/PhpExtractor.html
36. http://api.symfony.com/2.3/Symfony/Component/Translation/Extractor/ExtractorInterface.html

```php

use Symfony\Component\Translation\Extractor\ExtractorInterface;
use Symfony\Component\Translation\MessageCatalogue;

class FooExtractor implements ExtractorInterface
{
    protected $prefix;

    /**
     * Extracts translation messages from a template directory to the catalogue.
     */
    public function extract($directory, MessageCatalogue $catalogue)
    {
        // ...
    }

    /**
     * Sets the prefix that should be used for new found messages.
     */
    public function setPrefix($prefix)
    {
        $this->prefix = $prefix;
    }
}
```

Listing 88-21

```yaml
services:
    acme_demo.translation.extractor.foo:
        class: Acme\DemoBundle\Translation\FooExtractor
        tags:
            - { name: translation.extractor, alias: foo }
```

translation.dumper

Purpose: To register a custom service that dumps messages to a file

New in version 2.1: The ability to add message dumpers was introduced in Symfony 2.1.

After an *Extractor*[37] 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*[38]
- *IcuResFileDumper*[39]
- *IniFileDumper*[40]
- *MoFileDumper*[41]
- *PoFileDumper*[42]
- *QtFileDumper*[43]
- *XliffFileDumper*[44]

37. #reference-translation.extractor
38. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/CsvFileDumper.html
39. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/IcuResFileDumper.html
40. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/IniFileDumper.html
41. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/MoFileDumper.html
42. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/PoFileDumper.html
43. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/QtFileDumper.html

- *YamlFileDumper*[45]

You can create your own dumper by extending *FileDumper*[46] or implementing *DumperInterface*[47] 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 88-22
```
services:
    acme_demo.translation.dumper.json:
        class: Acme\DemoBundle\Translation\JsonFileDumper
        tags:
            - { 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`:

Listing 88-23
```
services:
    twig.extension.your_extension_name:
        class: Fully\Qualified\Extension\Class\Name
        tags:
            - { name: twig.extension }
```

For information on how to create the actual Twig Extension class, see *Twig's documentation*[48] on the topic or read the cookbook article: *How to Write a custom Twig Extension*.

Before writing your own extensions, have a look at the *Twig official extension repository*[49] 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 88-24
```
services:
    twig.extension.intl:
        class: Twig_Extensions_Extension_Intl
        tags:
            - { name: twig.extension }
```

twig.loader

Purpose: Register a custom service that loads Twig templates

By default, Symfony uses only one *Twig Loader*[50] - *FilesystemLoader*[51]. 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`:

44. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/XliffFileDumper.html
45. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/YamlFileDumper.html
46. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/FileDumper.html
47. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/DumperInterface.html
48. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension
49. https://github.com/twigphp/Twig-extensions
50. http://twig.sensiolabs.org/doc/api.html#loaders
51. http://api.symfony.com/2.3/Symfony/Bundle/TwigBundle/Loader/FilesystemLoader.html

Listing 88-25
```yaml
services:
    acme.demo_bundle.loader.some_twig_loader:
        class: Acme\DemoBundle\Loader\SomeTwigLoader
        tags:
            - { name: twig.loader }
```

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 cookbook article: *How to Create a custom Validation Constraint*.

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`*[52] interface. Then, tag it with the `validator.initializer` tag (it has no options).

For an example, see the `EntityInitializer` class inside the Doctrine Bridge.

52. http://api.symfony.com/2.3/Symfony/Component/Validator/ObjectInitializerInterface.html

Chapter 89
Requirements for Running Symfony

To run Symfony, your system needs to adhere to a list of requirements. You can easily see if your system passes all requirements by running the `web/config.php` in your Symfony distribution. Since the CLI often uses a different `php.ini` configuration file, it's also a good idea to check your requirements from the command line via:

Listing 89-1
```
$ php app/check.php
```

Below is the list of required and optional requirements.

Required

- PHP needs to be a minimum version of PHP 5.3.3
- JSON needs to be enabled
- ctype needs to be enabled
- Your `php.ini` needs to have the `date.timezone` setting

Be aware that Symfony has some known limitations when using a PHP version less than 5.3.8 or equal to 5.3.16. For more information see the *Requirements section of the README*[1].

Optional

- You need to have the PHP-XML module installed
- You need to have at least version 2.6.21 of libxml
- PHP tokenizer needs to be enabled
- mbstring functions need to be enabled
- iconv needs to be enabled
- POSIX needs to be enabled (only on *nix)

1. `https://github.com/symfony/symfony#requirements`

- Intl needs to be installed with ICU 4+
- APC 3.0.17+ (or another opcode cache needs to be installed)
- `php.ini` recommended settings
    - `short_open_tag = Off`
    - `magic_quotes_gpc = Off`
    - `register_globals = Off`
    - `session.auto_start = Off`

Doctrine

If you want to use Doctrine, you will need to have PDO installed. Additionally, you need to have the PDO driver installed for the database server you want to use.