Symfony

The Reference Book

for Symfony 2.1

generated on October 9, 2012

The Reference Book (2.1)

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
- ide
- test
- trust_proxy_headers
- **form**
    - enabled
- **csrf_protection**
    - enabled
    - field_name
- **session**
    - lifetime
- **templating**
    - assets_base_urls
    - assets_version
    - assets_version_format

secret

type: `string` **required**

This is a string that should be unique to your application. In practice, it's used for generating the CSRF tokens, but it could be used in any other context where having a unique string is useful. It becomes the service container parameter named `kernel.secret`.

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.

If you use TextMate or Mac Vim, you can simply use one of the following built-in values:

- `textmate`
- `macvim`

You can also specify a custom file link string. If you do this, all percentage signs (`%`) must be doubled to escape that character. For example, the full TextMate string would look like this:

Listing 1-1

```
framework:
    ide:  "txmt://open?url=file://%%f&line=%%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` PHP.ini value to the file link string. If this configuration value is set, then the `ide` option does not need to be specified.

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*.

trust_proxy_headers

type: `Boolean`

Configures if HTTP headers (like `HTTP_X_FORWARDED_FOR`, `X_FORWARDED_PROTO`, and `X_FORWARDED_HOST`) are trusted as indication for an SSL connection. By default, it is set to `false` and only SSL_HTTPS connections are indicated as secure.

You should enable this setting if your application is behind a reverse proxy.

form

csrf_protection

session

lifetime

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

This determines the lifetime of the session - in seconds. By default it will use `0`, which means the cookie is valid for the length of the browser session.

templating

assets_base_urls

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

This option allows you to define base URL's 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 URL's are provided, Symfony2 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 URL's for `http` and `https` requests. If a URL starts with `https://` or is *protocol-relative*[1] (i.e. starts with //) it will be added to both collections. URL's starting with `http://` will only be added to the `http` collection.

New in version 2.1: Unlike most configuration blocks, successive values for `assets_base_urls` will overwrite each other instead of being merged. This behavior was chosen because developers will typically define base URL's for each environment. Given that most projects tend to inherit configurations (e.g. `config_test.yml` imports `config_dev.yml`) and/or share a common base configuration (i.e. `config.yml`), merging could yield a set of base URL's for multiple environments.

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 1-2
```
1 <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:

Listing 1-3
```
1 # app/config/config.yml
2 framework:
3     # ...
4     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`

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

This specifies a *sprintf()*[2] 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 interpretted 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, we cannot modify it in-place (e.g. `/images/logo-v5.png`); however, we 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 forgo any URL rewriting. The latter option is useful if you would like older asset versions to remain accessible at their original URL.

Full Default Configuration

Listing 1-4

```
framework:

    # general configuration
    trust_proxy_headers:  false
    secret:               ~ # Required
    ide:                  ~
    test:                 ~
    default_locale:       en

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

    # esi configuration
    esi:
        enabled:              true

    # profiler configuration
    profiler:
        only_exceptions:      false
        only_master_requests:  false
        dsn:                  file:%kernel.cache_dir%/profiler
        username:
        password:
        lifetime:             86400
        matcher:
```

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

```
        ip:                   ~

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

# router configuration
router:
    resource:             ~ # Required
    type:                 ~
    http_port:            80
    https_port:           443
    # if false, an empty URL will be generated if a route is missing required parameters
    strict_requirements:  %kernel.debug%

# session configuration
session:
    auto_start:           false
    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

    # DEPRECATED! Please use: cookie_lifetime
    lifetime:             ~

    # DEPRECATED! Please use: cookie_path
    path:                 ~

    # DEPRECATED! Please use: cookie_domain
    domain:               ~

    # DEPRECATED! Please use: cookie_secure
    secure:               ~

    # DEPRECATED! Please use: cookie_httponly
    httponly:             ~

# 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:

        # A collection of named packages
        some_package_name:
            version:              ~
            version_format:       %%s?%%s
            base_urls:
                http:                 []
                ssl:                  []

# translator configuration
translator:
    enabled:              true
    fallback:             en

# validation configuration
validation:
    enabled:              true
    cache:                ~
    enable_annotations:   false

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

Chapter 2

AsseticBundle Configuration Reference

Full Default Configuration

Listing 2-1

```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:
```

```
32                  # A key-value array of options and values
33                  some_option_name: []
34      filters:
35
36          # An array of named filters (e.g. some_filter, some_other_filter)
37          some_filter:                []
38      twig:
39          functions:
40              # An array of named functions (e.g. some_function, some_other_function)
41              some_function:              []
```

Chapter 3
Configuration Reference

Listing 3-1
```
doctrine:
    dbal:
        default_connection:   default
        types:
            # A collection of custom types
            # Example
            some_custom_type:
                class:                Acme\HelloBundle\MyCustomType
                commented:            true

        connections:
            default:
                dbname:               database

            # 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 ommited)
                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:     ~
                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
ommited)
                        protocol:             ~

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

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

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

                        # Configuring MultipleActiveResultSets for the pdo_sqlsrv driver
                        MultipleActiveResultSets:  ~

    orm:
        default_entity_manager:  ~
        auto_generate_proxy_classes:  false
        proxy_dir:            %kernel.cache_dir%/doctrine/orm/Proxies
        proxy_namespace:      Proxies
        # search for the "ResolveTargetEntityListener" class for a cookbook about this
```

```
resolve_target_entities: []
entity_managers:
    # A collection of different named entity managers (e.g. some_em, another_em)
    some_em:
        query_cache_driver:
            type:                 array # Required
            host:                 ~
            port:                 ~
            instance_class:       ~
            class:                ~
        metadata_cache_driver:
            type:                 array # Required
            host:                 ~
            port:                 ~
            instance_class:       ~
            class:                ~
        result_cache_driver:
            type:                 array # Required
            host:                 ~
            port:                 ~
            instance_class:       ~
            class:                ~
        connection:           ~
        class_metadata_factory_name:  Doctrine\ORM\Mapping\ClassMetadataFactory
        default_repository_class:  Doctrine\ORM\EntityRepository
        auto_mapping:         false
        hydrators:

            # An array of hydrator names
            hydrator_name:                []
        mappings:
            # An array of mappings, which may be a bundle name or something else
            mapping_name:
                mapping:              true
                type:                 ~
                dir:                  ~
                alias:                ~
                prefix:               ~
                is_bundle:            ~
        dql:
            # a collection of string functions
            string_functions:
                # example
                # test_string: 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 3-2
```yaml
doctrine:
    orm:
        auto_mapping: true
        # the standard distribution overrides this to be true in debug, false otherwise
        auto_generate_proxy_classes: false
        proxy_namespace: Proxies
        proxy_dir: "%kernel.cache_dir%/doctrine/orm/Proxies"
        default_entity_manager: default
        metadata_cache_driver: array
        query_cache_driver: array
        result_cache_driver: array
```

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

Caching Drivers

For the caching drivers you can specify the values "array", "apc", "memcache", "memcached", "xcache" or "service".

The following example shows an overview of the caching configurations:

Listing 3-3
```yaml
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.

Besides default Doctrine options, there are some Symfony-related ones that you can configure. The following block shows all possible configuration keys:

Listing 3-4

```yaml
doctrine:
    dbal:
        dbname:               database
        host:                 localhost
        port:                 1234
        user:                 user
        password:             secret
        driver:               pdo_mysql
        driver_class:         MyNamespace\MyDriverImpl
        options:
            foo: bar
        path:                 "%kernel.data_dir%/data.sqlite"
        memory:               true
        unix_socket:          /tmp/mysql.sock
        wrapper_class:        MyDoctrineDbalConnectionWrapper
        charset:              UTF8
        logging:              "%kernel.debug%"
        platform_service:     MyOwnDatabasePlatformService
        mapping_types:
            enum: string
        types:
            custom: Acme\HelloBundle\MyCustomType
```

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

Listing 3-5

```yaml
doctrine:
    dbal:
        default_connection:       default
        connections:
            default:
                dbname:           Symfony2
                user:             root
                password:         null
                host:             localhost
```

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

```
10              customer:
11                  dbname:           customer
12                  user:             root
13                  password:         null
14                  host:             localhost
```

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]` if the name of the connection.

Chapter 4
Security Configuration Reference

The security system is one of the most powerful parts of Symfony2, 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 4-1

```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

        # Example options/values for what a custom encoder might look like
        Acme\Your\Class\Name:
            algorithm:            ~
            ignore_case:          false
            encode_as_base64:     true
            iterations:           5000
            id:                   ~

    providers:            # Required
        # Examples:
        memory:
            name:                memory
            users:
                foo:
                    password:           foo
                    roles:              ROLE_USER
                bar:
                    password:           bar
                    roles:              [ROLE_USER, ROLE_ADMIN]
        entity:
            entity:
                class:              SecurityBundle:User
                property:           username

        # Example custom provider
        some_custom_provider:
            id:                   ~
            chain:
                providers:           []

    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
            context: name
            stateless: false
            x509:
                provider: some_key_from_above
            http_basic:
                provider: some_key_from_above
            http_digest:
                provider: some_key_from_above
```

```yaml
        form_login:
            check_path: /login_check
            login_path: /login
            use_forward: false
            always_use_default_target_path: false
            default_target_path: /
            target_path_parameter: _target_path
            use_referer: false
            failure_path: /foo
            failure_forward: false
            failure_handler: some.service.id
            success_handler: some.service.id
            username_parameter: _username
            password_parameter: _password
            csrf_parameter: _csrf_token
            intention: authenticate
            csrf_provider: my.csrf_provider.id
            post_only: true
            remember_me: false
        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:      ~
```

```
145                 invalidate_session:   true
146                 delete_cookies:
147
148                     # Prototype
149                     name:
150                         path:                 ~
151                         domain:               ~
152                 handlers:             []
153             anonymous:
154                 key:                  4f954a0667e01
155             switch_user:
156                 provider:             ~
157                 parameter:            _switch_user
158                 role:                 ROLE_ALLOWED_TO_SWITCH
159
160     access_control:
161         requires_channel:     ~
162
163         # use the urldecoded format
164         path:                 ~ # Example: ^/path to resource/
165         host:                 ~
166         ip:                   ~
167         methods:              []
168         roles:                []
169     role_hierarchy:
170         ROLE_ADMIN:      [ROLE_ORGANIZER, ROLE_USER]
171         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:

The Login Form and Process

- `login_path` (type: `string`, default: `/login`) This is the URL that the user will be redirected to (unless `use_forward` is set to `true`) when he/she tries to access a protected resource but isn't fully authenticated.

  This URL **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 URL 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`)

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 Swiftmailer, which is responsible for creating and delivering email messages.

Configuration

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

type

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

The method used to store spooled messages. Currently only `file` is supported. However, 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 Swiftmailer's `Swift_Plugins_ImpersonatePlugin` class.

antiflood

threshold

type: `string` **default**: `99`

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

sleep

type: `string` **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.

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 Swiftmailer 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%"
```

Chapter 6
TwigBundle Configuration Reference

Listing 6-1

```yaml
twig:
    exception_controller:
Symfony\Bundle\TwigBundle\Controller\ExceptionController::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:           ~
    base_template_class:  ~ # Example: Twig_Template
    cache:                "%kernel.cache_dir%/twig"
    charset:              "%kernel.charset%"
    debug:                "%kernel.debug%"
    strict_variables:     ~
    auto_reload:          ~
    optimizations:        ~
```

Configuration

exception_controller

type: `string` **default**: `Symfony\\Bundle\\TwigBundle\\Controller\\ExceptionController::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/2.1/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html

Chapter 7
Configuration Reference

Listing 7-1

```yaml
monolog:
    handlers:

        # Examples:
        syslog:
            type:               stream
            path:               /var/log/symfony.log
            level:              ERROR
            bubble:             false
            formatter:          my_formatter
            processors:
                - some_callable
        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"
        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:              ~
            email_prototype:
                id:                   ~ # Required (when the email_prototype is used)
                factory-method:       ~
            channels:
                type:                 ~
                elements:             []
            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

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
    intercept_redirects:  false
```

Chapter 9

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). Symfony2 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 Symfony2:

Text Fields

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

Choice Fields

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

Date and Time Fields

- *date*

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

Other Fields

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

Field Groups

- *collection*
- *repeated*

Hidden Fields

- *hidden*
- *csrf*

Base Fields

- *field*
- *form*

Chapter 10
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
Options	• years
Inherited options	• widget • input • months • days • format • pattern • data_timezone • user_timezone • invalid_message • invalid_message_parameters
Parent type	*date*
Class	*`BirthdayType`*[1]

1. `http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/BirthdayType.html`

Field 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:

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 pattern option.
- `text`: renders a three field input of type text (month, day, year).
- `single_text`: renders a single input of type text. User's input is validated based on the format option.

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.

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`.

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 10-1
```
1  'days' => range(1,31)
```

format

type: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

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*[2]. For example, to render a single text box that expects the user to end `yyyy-MM-dd`, use the following options:

Listing 10-2
```
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    'format' => 'yyyy-MM-dd',
));
```

pattern

type: `string`

This option is only relevant when the widget is set to `choice`. The default pattern is based off the format option, and tries to match the characters `M`, `d`, and `y` in the format pattern. If no match is found, the default is the string `{{ year }}-{{ month }}-{{ day }}`. Tokens for this option include:

- `{{ year }}`: Replaced with the `year` widget
- `{{ month }}`: Replaced with the `month` widget
- `{{ day }}`: Replaced with the `day` widget

data_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[3]

user_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[4]

These options inherit from the *date* type:

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*).

---

2. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax
3. http://php.net/manual/en/timezones.php
4. http://php.net/manual/en/timezones.php

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 10-3
```
$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),
));
```

Chapter 11
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 text` field
Options	• value
Inherited options	• required • label • read_only • error_bubbling
Parent type	*field*
Class	*CheckboxType*[1]

Example Usage

Listing 11-1
```
1 $builder->add('public', 'checkbox', array(
2     'label'     => 'Show this entry publicly?',
3     'required'  => false,
4 ));
```

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/CheckboxType.html

Field Options

value

type: `mixed` **default**: `1`

The value that's actually used as the value for the checkbox. This does not affect the value that's set on your object.

Inherited options

These options inherit from the *field* type:

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.

label

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

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

Listing 11-2
```
1 {{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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

Chapter 12
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 • multiple • expanded • preferred_choices • empty_value • empty_data
Inherited options	• required • label • read_only • error_bubbling
Parent type	*form* (if expanded), `field` otherwise
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`).

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/ChoiceType.html

Listing 12-1
```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 12-2
```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 12-3
```php
$builder->add('gender', 'choice', array(
    'choices' => array('m' => 'Male', 'f' => 'Female')
));
```

choice_list

type: `Symfony\Component\Form\Extension\Core\ChoiceList\ChoiceListInterface`

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.

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.

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.

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 12-4
```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 12-5
```jinja
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 12-6
```php
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 12-7

```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 12-8
```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

empty_data

type: `mixed` **default**: `array()` if `multiple` or `expanded`, `''` otherwise

This option determines what value the field will return when the `empty_value` choice is selected.

For example, if you want the `gender` field to be set to `null` when no value is selected, you can do it like this:

Listing 12-9
```php
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

Inherited options

These options inherit from the *field* type:

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.

label

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

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

Listing 12-10

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

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

read_only

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

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

error_bubbling

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

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.

Chapter 13
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	• type • options • allow_add • allow_delete • prototype
Inherited options	• label • error_bubbling • by_reference
Parent type	*form*
Class	*CollectionType*[1]

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 13-1
```
1 $builder->add('emails', 'collection', array(
2     // each item in the array will be an "email" field
```

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/CollectionType.html

```
    '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 13-2
```
{{ form_row(form.emails) }}
```

A much more flexible method would look like this:

Listing 13-3
```
{{ 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 13-4
```
<input type="email" id="form_emails_1" 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 13-5
```
<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 13-6

```
<form action="..." method="POST" {{ form_enctype(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>

<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() {
            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 our 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 our list
            var newLi = jQuery('<li></li>').html(newWidget);
            newLi.appendTo(jQuery('#email-fields-list'));

            return false;
        });
    })
</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 like we've done above.

Field Options

type

type: `string` or *FormTypeInterface*[2] **required**

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.

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 13-7

```
$builder->add('favorite_cities', 'collection', array(
    'type'   => 'choice',
    'options'  => array(
        'choices'  => array(
            'nashville' => 'Nashville',
            'paris'     => 'Paris',
            'berlin'    => 'Berlin',
            'london'    => 'London',
        ),
    ),
));
```

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.

2. http://api.symfony.com/2.1/Symfony/Component/Form/FormTypeInterface.html

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*.

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 13-8
```
1 {{ form_row(form.emails.vars.prototype) }}
```

Note that all you really need is the "widget", but depending on how you're rendering your form, having the entire "form row" may be easier for you.

If you're rendering the entire collection field at once, then the prototype form row is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection.

For details on how to actually use this option, see the above example as well as *Allowing "new" tags with the "prototype"*.

Inherited options

These options inherit from the *field* type. Not all options are listed here - only the most applicable to this type:

label

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

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

Listing 13-9
```
1 {{ form_label(form.name, 'Your name') }}
```

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.

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 understand this further, let's look at a simple example:

Listing 13-10
```
$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 `bind` on the form:

Listing 13-11
```
$article->setTitle('...');
$article->getAuthor()->setName('...');
$article->getAuthor()->setEmail('...');
```

Notice that `setAuthor` is not called. The author is modified by reference.

If we set `by_reference` to false, binding looks like this:

Listing 13-12
```
$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 setter (e.g. `setAuthors`) to be called.

Chapter 14
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*)
Inherited options	• multiple • expanded • preferred_choices • empty_value • error_bubbling • required • label • read_only
Parent type	*choice*
Class	*CountryType*[2]

1. http://php.net/manual/en/locale.getdefault.php
2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/CountryType.html

Inherited options

These options inherit from the *choice* type:

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.

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.

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 14-1
```
$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 14-2
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 14-3
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 14-4

```
$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:

Listing 14-5
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling

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

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.

These options inherit from the *field* type:

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.

label

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

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

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

read_only

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

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

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

Chapter 15
csrf Field Type

The `csrf` type is a hidden input field containing a CSRF token.

Rendered as	`input hidden` field
Options	• `csrf_provider` • `intention` • `property_path`
Parent type	`hidden`
Class	*`CsrfType`*[1]

Field Options

csrf_provider

type: `Symfony\Component\Form\CsrfProvider\CsrfProviderInterface`

The `CsrfProviderInterface` object that should generate the CSRF token. If not set, this defaults to the default provider.

intention

type: `string`

An optional unique identifier used to generate the CSRF token.

property_path

type: any **default**: `the field's value`

1. `http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Csrf/Type/CsrfType.html`

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`

Chapter 16
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	• widget • input • empty_value • years • months • days • format • pattern • data_timezone • user_timezone
Inherited options	• invalid_message • invalid_message_parameters
Parent type	`field` (if text), `form` otherwise
Class	*`DateType`*[1]

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/DateType.html

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 16-1
```
$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 16-2
```
$builder->add('publishedAt', 'date', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

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 pattern option.
- `text`: renders a three field input of type text (month, day, year).
- `single_text`: renders a single input of type text. User's input is validated based on the format option.

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.

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 16-3
```
$builder->add('dueDate', 'date', array(
    'empty_value' => '',
));
```

Alternatively, you can specify a string to be displayed for the "blank" value:

Listing 16-4
```
$builder->add('dueDate', 'date', array(
    'empty_value' => array('year' => 'Year', 'month' => 'Month', 'day' => 'Day')
));
```

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`.

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`.

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 16-5
```
'days' => range(1,31)
```

format

type: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

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*[2]. For example, to render a single text box that expects the user to end `yyyy-MM-dd`, use the following options:

Listing 16-6
```
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    'format' => 'yyyy-MM-dd',
));
```

2. http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax

pattern

type: `string`

This option is only relevant when the widget is set to `choice`. The default pattern is based off the format option, and tries to match the characters `M`, `d`, and `y` in the format pattern. If no match is found, the default is the string `{{ year }}-{{ month }}-{{ day }}`. Tokens for this option include:

- `{{ year }}`: Replaced with the `year` widget
- `{{ month }}`: Replaced with the `month` widget
- `{{ day }}`: Replaced with the `day` widget

data_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the *PHP supported timezones*[3]

user_timezone

type: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the *PHP supported timezones*[4]

Inherited options

These options inherit from the *field* type:

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 16-7
```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),
));
```

3. http://php.net/manual/en/timezones.php
4. http://php.net/manual/en/timezones.php

Chapter 17
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_widget • time_widget • input • date_format • hours • minutes • seconds • years • months • days • with_seconds • data_timezone • user_timezone
Inherited options	• invalid_message • invalid_message_parameters
Parent type	*form*
Class	*`DateTimeType`*[1]

1. `http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/DateTimeType.html`

Field Options

date_widget

type: `string` **default**: `choice`

Defines the `widget` option for the *date* type

time_widget

type: `string` **default**: `choice`

Defines the `widget` option for the *time* type

input

type: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05 12:15:00`)
- `datetime` (a `DateTime` object)
- `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.

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.

hours

type: `integer` **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`.

minutes

type: `integer` **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`.

seconds

type: `integer` **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`.

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`.

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`.

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 17-1
```
1  'days' => range(1,31)
```

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.

data_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]

user_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]

Inherited options

These options inherit from the *field* type:

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).

2. `http://php.net/manual/en/timezones.php`
3. `http://php.net/manual/en/timezones.php`

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 17-2
```
$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),
));
```

Chapter 18

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	• max_length • required • label • trim • read_only • error_bubbling
Parent type	*field*
Class	*EmailType*[1]

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/EmailType.html

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.

label

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

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

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

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 19
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 • property • group_by • query_builder • em
Inherited options	• required • label • multiple • expanded • preferred_choices • empty_value • read_only • error_bubbling
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 19-1

1. `http://api.symfony.com/2.1/Symfony/Bridge/Doctrine/Form/Type/EntityType.html`

```php
$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 19-2

```php
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');
    },
));
```

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).

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.

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 tags around options. Choices that do not return a value for this property path are rendered directly under the select tag, without a surrounding optgroup.

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.

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.

Inherited options

These options inherit from the *choice* type:

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.

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.

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 19-3
```
1 $builder->add('foo_choices', 'choice', array(
2     '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 19-4
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 19-5
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 19-6
```
$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 19-7
```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

These options inherit from the *field* type:

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.

label

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

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

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

Listing 19-8
```
{{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

Chapter 20
file Field Type

The `file` type represents a file input in your form.

Rendered as	`input file` field
Inherited options	• required • label • read_only • error_bubbling
Parent type	*form*
Class	*`FileType`*[1]

Basic Usage

Let's say you have this form definition:

Listing 20-1
```
1  $builder->add('attachment', 'file');
```

Don't forget to add the `enctype` attribute in the form tag: `<form action="#" method="post" {{ form_enctype(form) }}>`.

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 20-2

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/FileType.html
2. http://api.symfony.com/2.1/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 20-3

```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 *field* type:

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.

label

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

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

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

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

read_only

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

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

error_bubbling

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

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.

Chapter 21

The Abstract "field" Type

The `field` form type is deprecated as of Symfony 2.1. Please use the *Form field type* instead.

Chapter 22
form Field Type

See *FormType*[1].

The `form` type predefines a couple of options that are then available on all fields.

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 22-1
```php
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

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.

cascade_validation

type: Boolean **default**: false

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/FormType.html
2. http://diveintohtml5.info/forms.html

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, you can also use the `Valid` constraint in your model to force validation on a child object stored on a property.

disabled

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

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

Listing 22-2

```
use Symfony\Component\Form\TextField

$field = new TextField('status', array(
    'data' => 'Old data',
    'disabled' => true,
));
$field->submit('New data');

// prints "Old data"
echo $field->getData();
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

property_path

type: `any` **default**: `the field's value`

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`

attr

type: array **default**: Empty array

If you want to add extra attributes to HTML field representation 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 some widget:

Listing 22-3
```php
$builder->add('body', 'textarea', array(
    'attr' => array('class' => 'tinymce'),
));
```

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 23
hidden Field Type

The hidden type represents a hidden input field.

Rendered as	`input hidden` field
Inherited options	• `data` • `property_path`
Parent type	*field*
Class	*`HiddenType`*[1]

Inherited Options

These options inherit from the *field* 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 23-1

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

property_path

type: any **default**: `the field's value`

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/HiddenType.html

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`

Chapter 24
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 text` field
Options	• rounding_mode • grouping
Inherited options	• required • label • read_only • error_bubbling • invalid_message • invalid_message_parameters
Parent type	*field*
Class	*IntegerType*[1]

Field Options

rounding_mode

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/IntegerType.html

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.

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`.

Inherited options

These options inherit from the *field* type:

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.

label

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

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

Listing 24-1
```
1 {{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.html
3. http://diveintohtml5.info/forms.html

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.

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 24-2
```
$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),
));
```

Chapter 25
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* (e.g. `fr` or `zh-Hant`).

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 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*)
Inherited options	• multiple • expanded • preferred_choices • empty_value • error_bubbling • required • label • read_only
Parent type	*choice*
Class	*LanguageType*[2]

1. http://php.net/manual/en/locale.getdefault.php
2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/LanguageType.html

Inherited Options

These options inherit from the *choice* type:

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.

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.

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 25-1
```
$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 25-2
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 25-3
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 25-4

```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 25-5
```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling

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

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.

These options inherit from the *field* type:

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.

label

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

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

Listing 25-6
```
{{ form_label(form.name, 'Your name') }}
```

read_only

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

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

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

Chapter 26
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 ISO639-1 *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

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 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*)
Inherited options	• multiple • expanded • preferred_choices • empty_value • error_bubbling • required • label • read_only
Parent type	*choice*
Class	*LanguageType*[2]

1. http://php.net/manual/en/locale.getdefault.php
2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/LanguageType.html

Inherited options

These options inherit from the *choice* type:

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.

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.

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 26-1
```
$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 26-2
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 26-3
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 26-4

```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 26-5
```php
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling

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

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.

These options inherit from the *field* type:

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.

label

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

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

Listing 26-6
```twig
{{ form_label(form.name, 'Your name') }}
```

read_only

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

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

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

Chapter 27
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 • precision • grouping
Inherited options	• required • label • read_only • error_bubbling • invalid_message • invalid_message_parameters
Parent type	*field*
Class	*MoneyType*[1]

Field Options

currency

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/MoneyType.html

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 also be set 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 27-1
```
$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.

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`).

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`.

Inherited Options

These options inherit from the *field* type:

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.

label

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

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

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

Listing 27-2
```
{{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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 27-3
```
$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),
));
```

Chapter 28
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	• rounding_mode • precision • grouping
Inherited options	• required • label • read_only • error_bubbling • invalid_message • invalid_message_parameters
Parent type	*field*
Class	*NumberType*[1]

Field Options

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`).

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/NumberType.html

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.

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`.

Inherited Options

These options inherit from the *field* type:

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.

label

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

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

2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/DataTransformer/IntegerToLocalizedStringTransformer.html
3. http://diveintohtml5.info/forms.html

Listing 28-1
```
1 {{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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 28-2
```
1 $builder->add('some_field', 'some_type', array(
2     // ...
3     'invalid_message'            => 'You entered an invalid value - it should include %num%
4 letters',
5     'invalid_message_parameters' => array('%num%' => 6),
  ));
```

Chapter 29
password Field Type

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

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

Field Options

always_empty

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

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

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.

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/PasswordType.html

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

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.

label

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

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

Listing 29-1
```
1 {{ form_label(form.name, 'Your name') }}
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 30
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	• type • precision
Inherited options	• required • label • read_only • error_bubbling • invalid_message • invalid_message_parameters
Parent type	*field*
Class	*PercentType*[1]

Options

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:

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/PercentType.html

- `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.

precision

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

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

Inherited Options

These options inherit from the *field* type:

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.

label

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

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

Listing 30-1
```
1 {{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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).

---

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

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 30-2
```
$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),
));
```

Chapter 31

radio Field Type

Creates a single radio button. This should always be used for a field that has a Boolean value: if the radio button is selected, the field will be set to true, if the button is not selected, the value will be set to false.

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
Options	• value
Inherited options	• required • label • read_only • error_bubbling
Parent type	*field*
Class	*RadioType*[1]

Field Options

value

type: `mixed` **default**: `1`

The value that's actually used as the value for the radio button. This does not affect the value that's set on your object.

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/RadioType.html

Inherited Options

These options inherit from the *field* type:

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.

label

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

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

Listing 31-1
```
1 {{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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

Chapter 32

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 his or her password or email to verify accuracy.

Rendered as	input `text` field by default, but see type option
Options	• type • options • first_options • second_options • first_name • second_name
Inherited options	• invalid_message • invalid_message_parameters • error_bubbling
Parent type	*field*
Class	*RepeatedType*[1]

Example Usage

Listing 32-1

```
1  $builder->add('password', 'repeated', array(
2      'type' => 'password',
3      'invalid_message' => 'The password fields must match.',
4      'options' => array('attr' => array('class' => 'password-field')),
5      'required' => true,
```

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/RepeatedType.html

```
6      'first_options'  => array('label' => 'Password'),
7      'second_options' => array('label' => 'Repeat Password'),
8  ));
```

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.

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

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.

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.

first_options

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

New in version 2.1: The `first_options` option is new in Symfony 2.1.

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 32-2
```
1  $builder->add('password', 'repeated', array(
2      'first_options'  => array('label' => 'Password'),
```

```
3     'second_options' => array('label' => 'Repeat Password'),
4 ));
```

second_options

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

New in version 2.1: The `second_options` option is new in Symfony 2.1.

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).

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.

second_name

type: `string` **default**: `second`

The same as `first_name`, but for the second field.

Inherited options

These options inherit from the *field* type:

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 32-3

```
$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),
));
```

error_bubbling

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

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.

Chapter 33
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	• max_length • required • label • trim • read_only • error_bubbling
Parent type	*text*
Class	*SearchType*[2]

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

1. http://diveintohtml5.info/forms.html#type-search
2. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/SearchType.html

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.

label

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

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

Listing 33-1
```
1 {{ form_label(form.name, 'Your name') }}
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 34

text Field Type

The text field represents the most basic input text field.

Rendered as	`input text` field
Inherited options	• max_length • required • label • trim • read_only • error_bubbling
Parent type	*field*
Class	*TextType*[1]

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/TextType.html

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.

label

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

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

Listing 34-1
```
{{ form_label(form.name, 'Your name') }}
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 35

textarea Field Type

Renders a `textarea` HTML element.

Rendered as	`textarea` tag
Inherited options	• max_length • required • label • trim • read_only • error_bubbling
Parent type	*field*
Class	*TextareaType*[1]

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/TextareaType.html

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.

label

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

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

Listing 35-1
```
{{ form_label(form.name, 'Your name') }}
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 36
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	• widget • input • with_seconds • hours • minutes • seconds • data_timezone • user_timezone
Inherited options	• invalid_message • invalid_message_parameters
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`.

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/TimeType.html

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 three different choice fields:

Listing 36-1
```
$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 36-2
```
$builder->add('startTime', 'time', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

widget

type: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders two (or three if with_seconds is true) select inputs.
- `text`: renders a two or three text inputs (hour, minute, second).
- `single_text`: renders a single input of type text. User's input will be validated against the form `hh:mm` (or `hh:mm:ss` if using seconds).

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.

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.

hours

type: `integer` **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`.

minutes

type: `integer` **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`.

seconds

type: `integer` **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`.

data_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]

user_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]

Inherited options

These options inherit from the *field* type:

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 36-3

2. `http://php.net/manual/en/timezones.php`
3. `http://php.net/manual/en/timezones.php`

```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),
));
```

Chapter 37

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 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*)
Inherited options	• multiple • expanded • preferred_choices • empty_value • error_bubbling • required • label • read_only
Parent type	*choice*
Class	*TimezoneType*[1]

Inherited options

These options inherit from the *choice* type:

multiple

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

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/TimezoneType.html

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.

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.

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 37-1
```
$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 37-2
```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

empty_value

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 both the `expanded` and `multiple` options are set to false.

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

Listing 37-3
```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

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

Listing 37-4
```
$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:

Listing 37-5

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

These options inherit from the *field* type:

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.

label

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

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

Listing 37-6
```twig
{{ form_label(form.name, 'Your name') }}
```

read_only

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

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

error_bubbling

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

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.

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

Chapter 38
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	• max_length • required • label • trim • read_only • error_bubbling
Parent type	*text*
Class	*UrlType*[1]

Field Options

default_protocol

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

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

1. `http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Core/Type/UrlType.html`

Inherited Options

These options inherit from the *field* type:

max_length

type: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

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.

label

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

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

Listing 38-1
```
1 {{ form_label(form.name, 'Your name') }}
```

trim

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

If true, the whitespace of the submitted string value will be stripped via the `trim()` 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.

read_only

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

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

error_bubbling

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

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.

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

Chapter 39

Twig Template Form Function Reference

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_label(form.name, 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 39-1
```
{{ form_label(form.name) }}

{# The two following syntaxes are equivalent #}
{{ form_label(form.name, 'Your Name', {'label_attr': {'class': 'foo'}}) }}
{{ form_label(form.name, null, {'label': 'Your name', 'label_attr': {'class': 'foo'}}) }}
```

form_errors(form.name)

Renders any errors for the given field.

Listing 39-2
```
{{ form_errors(form.name) }}

{# render any "global" errors #}
{{ form_errors(form) }}
```

form_widget(form.name, 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 39-3
```twig
{# 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.

form_row(form.name, variables)

Renders the "row" of a given field, which is the combination of the field's label, errors and widget.

Listing 39-4
```twig
{# 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.

form_rest(form, 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 39-5
```twig
{{ form_rest(form) }}
```

form_enctype(form)

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 39-6
```twig
<form action="{{ path('form_submit') }}" method="post" {{ form_enctype(form) }}>
```

Chapter 40

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 Symfony2, constraints are similar: They are assertions that a condition is true.

Supported Constraints

The following constraints are natively available in Symfony2:

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*
- *MinLength*
- *MaxLength*
- *Length*
- *Url*
- *Regex*
- *Ip*

Number Constraints

- *Max*
- *Min*
- *Range*

Date Constraints

- *Date*
- *DateTime*
- *Time*

Collection Constraints

- *Choice*
- *Collection*
- *Count*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Other Constraints

- *Callback*
- *All*
- *UserPassword*
- *Valid*

Chapter 41
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 41-1
```
properties:
    firstName:
        - NotBlank: ~
```

Options

message

type: `string` **default**: `This value should not be blank`

This is the message that will be shown if the value is blank.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/NotBlank.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/NotBlankValidator.html

Chapter 42
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 42-1
```
properties:
    firstName:
        - Blank: ~
```

Options

message

type: `string` **default**: `This value should be blank`

This is the message that will be shown if the value is not blank.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Blank.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/BlankValidator.html

Chapter 43
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 43-1

```
properties:
    firstName:
        - NotNull: ~
```

Options

message

type: `string` **default**: `This value should not be null`

This is the message that will be shown if the value is `null`.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/NotNull.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/NotNullValidator.html

Chapter 44
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 44-1
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - Null: ~
```

Options

message

type: `string` **default**: `This value should be null`

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Null.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/NullValidator.html

This is the message that will be shown if the value is not `null`.

Chapter 45
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 45-1
```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/2.1/Symfony/Component/Validator/Constraints/True.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/TrueValidator.html

```
11      }
12  }
```

Then you can constrain this method with `True`.

Listing 45-2
```
1  # src/Acme/BlogBundle/Resources/config/validation.yml
2  Acme\BlogBundle\Entity\Author:
3      getters:
4          tokenValid:
5              - "True": { message: "The token is invalid" }
```

If the `isTokenValid()` returns false, the validation will fail.

Options

message

type: `string` **default**: `This value should be true`

This message is shown if the underlying data is not true.

Chapter 46
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 46-1

```
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 46-2

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/False.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/FalseValidator.html

```yaml
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author
    getters:
        stateInvalid:
            - "False":
                message: You've entered an invalid state.
```

When using YAML, be sure to surround `False` with quotes (`"False"`) or else YAML will convert this into a Boolean value.

Options

message

type: `string` **default**: `This value should be false`

This message is shown if the underlying data is not false.

Chapter 47
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 47-1
```yaml
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        age:
            - Type:
                type: integer
                message: The value {{ value }} is not a valid {{ type }}.
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Type.html
2. http://api.symfony.com/2.1/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]

message

type: `string` **default**: `This value should be of type {{ type }}`

The message if the underlying data is not of the given type.

3. http://php.net/is_array
4. http://php.net/is_bool
5. http://php.net/is_callable
6. http://php.net/is_float
7. http://php.net/is_double
8. http://php.net/is_int
9. http://php.net/is_integer
10. http://php.net/is_long
11. http://php.net/is_null
12. http://php.net/is_numeric
13. http://php.net/is_object
14. http://php.net/is_real
15. http://php.net/is_resource
16. http://php.net/is_scalar
17. http://php.net/is_string

Chapter 48
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 48-1

```yaml
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        email:
            - Email:
                message: The email "{{ value }}" is not a valid email.
                checkMX: true
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Email.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/EmailValidator.html

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

New in version 2.1: The `checkHost` option was added in Symfony 2.1

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 49
MinLength

Validates that the length of a string is at least as long as the given limit.

Applies to	*property or method*
Options	• limit • message • charset
Class	*MinLength*[1]
Validator	*MinLengthValidator*[2]

Basic Usage

Listing 49-1
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Blog:
    properties:
        firstName:
            - MinLength: { limit: 3, message: "Your name must have at least {{ limit }} characters." }
```

Options

limit

type: `integer` [*default option*]

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MinLength.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MinLengthValidator.html

This required option is the "min" value. Validation will fail if the length of the give string is **less** than this number.

message

type: `string` **default**: `This value is too short. It should have {{ limit }} characters or more`

The message that will be shown if the underlying string has a length that is shorter than the limit option.

charset

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

If the PHP extension "mbstring" is installed, then the PHP function *mb_strlen*[3] will be used to calculate the length of the string. The value of the `charset` option is passed as the second argument to that function.

3. `http://php.net/manual/en/function.mb-strlen.php`

Chapter 50
MaxLength

Validates that the length of a string is not larger than the given limit.

Applies to	*property or method*
Options	• limit • message • charset
Class	*MaxLength*[1]
Validator	*MaxLengthValidator*[2]

Basic Usage

Listing 50-1

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Blog:
    properties:
        summary:
            - MaxLength: 100
```

Options

limit

type: `integer` [*default option*]

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MaxLength.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MaxLengthValidator.html

This required option is the "max" value. Validation will fail if the length of the give string is **greater** than this number.

message

type: `string` **default**: `This value is too long. It should have {{ limit }} characters or less`

The message that will be shown if the underlying string has a length that is longer than the limit option.

charset

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

If the PHP extension "mbstring" is installed, then the PHP function *mb_strlen*[3] will be used to calculate the length of the string. The value of the `charset` option is passed as the second argument to that function.

3. `http://php.net/manual/en/function.mb-strlen.php`

Chapter 51
Length

Validates that a given string length is *between* some minimum and maximum value.

New in version 2.1: The Length constraint was added in Symfony 2.1.

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
```yaml
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        firstName:
            - Length:
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Length.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/LengthValidator.html

```
6                 min: 2
7                 max: 50
8                 minMessage: Your first name must be at least 2 characters length
9                 maxMessage: Your first name cannot be longer than than 50 characters length
```

Options

min

type: `integer` [*default option*]

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` [*default option*]

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 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
```yaml
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        bioUrl:
            - Url:
```

Options

message

type: `string` **default**: `This value is not a valid URL`

This message is shown if the URL is invalid.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Url.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/UrlValidator.html

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 • 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/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        description:
            - Regex: "/^\w+/"
```

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
```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Regex.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/RegexValidator.html

```
properties:
    firstName:
        - Regex:
            pattern: "/\d/"
            match:   false
            message: Your name cannot contain a number
```

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.

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.

3. http://php.net/manual/en/function.preg-match.php

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

```yaml
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        ipAddress:
            - Ip:
```

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:

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Ip.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/IpValidator.html

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
Max

Validates that a given number is *less* than some maximum number.

Applies to	*property or method*
Options	• limit • message • invalidMessage
Class	*Max*[1]
Validator	*MaxValidator*[2]

Basic Usage

To verify that the "age" field of a class is not greater than "50", you might add the following:

Listing 55-1
```yaml
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        age:
            - Max: { limit: 50, message: You must be 50 or under to enter. }
```

Options

limit

type: `integer` [*default option*]

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Max.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MaxValidator.html

This required option is the "max" value. Validation will fail if the given value is **greater** than this max value.

message

type: `string` **default**: `This value should be {{ limit }} or less`

The message that will be shown if the underlying value is greater than the limit 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
Min

Validates that a given number is *greater* than some minimum number.

Applies to	*property or method*
Options	• limit • message • invalidMessage
Class	*Min*[1]
Validator	*MinValidator*[2]

Basic Usage

To verify that the "age" field of a class is "18" or greater, you might add the following:

Listing 56-1
```yaml
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        age:
            - Min: { limit: 18, message: You must be 18 or older to enter. }
```

Options

limit

type: `integer` [*default option*]

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Min.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/MinValidator.html

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

message

type: `string` **default**: `This value should be {{ limit }} or more`

The message that will be shown if the underlying value is less than the limit 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 57
Range

Validates that a given number is *between* some minimum and maximum number.

New in version 2.1: The Range constraint was added in Symfony 2.1.

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 57-1
```yaml
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        height:
            - Range:
                min: 120
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Range.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/RangeValidator.html

```
7                 max: 180
8                 minMessage: You must be at least 120cm tall to enter
9                 maxMessage: You cannot be taller than 180cm to enter
```

Options

min

type: `integer` [*default option*]

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

max

type: `integer` [*default option*]

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 58
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 58-1
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        birthday:
            - Date: ~
```

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.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Date.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/DateValidator.html

Chapter 59
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 59-1
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        createdAt:
            - DateTime: ~
```

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.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/DateTime.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/DateTimeValidator.html

Chapter 60
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 60-1
```
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Event:
    properties:
        startsAt:
            - Time: ~
```

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.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Time.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/TimeValidator.html

Chapter 61
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 61-1
```
1  # src/Acme/BlogBundle/Resources/config/validation.yml
2  Acme\BlogBundle\Entity\Author:
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Choice.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/ChoiceValidator.html

```
    properties:
        gender:
            - Choice:
                choices:  [male, female]
                message:  Choose a valid gender.
```

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 61-2
```php
// src/Acme/BlogBundle/Entity/Author.php
class Author
{
    public static function getGenders()
    {
        return array('male', 'female');
    }
}
```

You can pass the name of this method to the *callback_* option of the `Choice` constraint.

Listing 61-3
```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { callback: getGenders }
```

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 61-4
```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { callback: [Util, getGenders] }
```

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 62
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 `Min` 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 62-1
```
1  namespace Acme\BlogBundle\Entity;
2
3  class Author
4  {
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Collection.html
2. http://api.symfony.com/2.1/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 62-2

```
properties:
    profileData:
        - Collection:
            fields:
                personal_email: Email
                short_bio:
                    - NotBlank
                    - MaxLength:
                        limit:   100
                        message: Your short bio is too long!
            allowMissingfields: true
```

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.

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 63
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.

New in version 2.1: The Count constraint was added in Symfony 2.1.

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 63-1
```yaml
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        emails:
            - Count:
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Count.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/CountValidator.html

```
6                 min: 1
7                 max: 5
8                 minMessage: You must specify at least one email
9                 maxMessage: You cannot specify more than 5 emails
```

Options

min

type: `integer` [*default option*]

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` [*default option*]

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 64
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
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 64-1

```
// Acme/UserBundle/Entity/User.php
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\ORM\Mapping as ORM;

// DON'T forget this use statement!!!
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity("email")
```

1. http://api.symfony.com/2.1/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntity.html
2. http://api.symfony.com/2.1/Symfony/Bridge/Doctrine/Validator/Constraints/UniqueEntityValidator.html

```
 */
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 determined for this class. For that reason, this option should probably not need to be used.

Chapter 65
Language

Validates that a value is a valid language code.

Applies to	*property or method*
Options	• message
Class	*Language*[1]
Validator	*LanguageValidator*[2]

Basic Usage

Listing 65-1
```yaml
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        preferredLanguage:
            - Language:
```

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.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Language.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/LanguageValidator.html

Chapter 66
Locale

Validates that a value is a valid locale.

The "value" for each locale is either the two letter ISO639-1 *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

Applies to	*property or method*
Options	• message
Class	*Locale*[1]
Validator	*LocaleValidator*[2]

Basic Usage

Listing 66-1
```yaml
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        locale:
            - Locale:
```

Options

message

type: `string` **default**: `This value is not a valid locale`

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Locale.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/LocaleValidator.html

This message is shown if the string is not a valid locale.

Chapter 67
Country

Validates that a value is a valid two-letter country code.

Applies to	*property or method*
Options	• message
Class	*Country*[1]
Validator	*CountryValidator*[2]

Basic Usage

Listing 67-1
```yaml
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        country:
            - Country:
```

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.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Country.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/CountryValidator.html

Chapter 68
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/2.1/Symfony/Component/HttpFoundation/File/File.html
2. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/File/UploadedFile.html
3. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/File.html
4. http://api.symfony.com/2.1/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 68-1
```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 68-2
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author
    properties:
        bioFile:
            - File:
                maxSize: 1024k
                mimeTypes: [application/pdf, application/x-pdf]
                mimeTypesMessage: Please upload a valid PDF
```

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 }}). Allowed maximum size is {{ limit }}`

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 }}`

The message that is displayed if the uploaded file is larger than the `upload_max_filesize` PHP.ini setting.

uploadFormSizeErrorMessage

type: `string` **default**: `The file is too large`

The message that is displayed if the uploaded file is larger than allowed by the HTML file input field.

uploadErrorMessage

type: `string` **default**: `The file could not be uploaded`

The message that is displayed if the uploaded file could not be uploaded for some unknown reason, such as the file upload failed or it couldn't be written to disk.

5. `http://www.iana.org/assignments/media-types/index.html`

Chapter 69
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	*File*[1]
Validator	*FileValidator*[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/2.1/Symfony/Component/Validator/Constraints/File.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/FileValidator.html

author. In your form, the `headshot` property would be a `file` type. The `Author` class might look as follows:

Listing 69-1
```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 69-2
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author
    properties:
        headshot:
            - Image:
                minWidth: 200
                maxWidth: 400
                minHeight: 200
                maxHeight: 400
```

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`

3. http://www.iana.org/assignments/media-types/image/index.html

New in version 2.1: All of the min/max width/height options are new to Symfony 2.1.

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.

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 70
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 70-1
```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    constraints:
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Callback.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/CallbackValidator.html

```
4          - Callback:
5              methods:   [isAuthorValid]
```

The Callback Method

The callback method is passed a special `ExecutionContext` object. You can set "violations" directly on this object and determine to which field those errors should be attributed:

Listing 70-2

```
1  // ...
2  use Symfony\Component\Validator\ExecutionContext;
3
4  class Author
5  {
6      // ...
7      private $firstName;
8
9      public function isAuthorValid(ExecutionContext $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->addViolationAtSubPath('firstname', 'This name sounds totally fake!',
17 array(), null);
18         }
       }
```

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 `ExecutionContext` 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 70-3

```
1  # src/Acme/BlogBundle/Resources/config/validation.yml
2  Acme\BlogBundle\Entity\Author:
3      constraints:
4          - Callback:
               methods:
```

```
5                 -    [Acme\BlogBundle\MyStaticValidatorClass,
6 isAuthorValid]
```

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 `ExecutionContext`:

Listing 70-4

```
1  namespace Acme\BlogBundle;
2
3  use Symfony\Component\Validator\ExecutionContext;
4  use Acme\BlogBundle\Entity\Author;
5
6  class MyStaticValidatorClass
7  {
8      static public function isAuthorValid(Author $author,
9  ExecutionContext $context)
10     {
11         // ...
12     }
   }
```

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 71
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 71-1
```yaml
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        favoriteColors:
            - All:
                - NotBlank:  ~
                - MinLength: 5
```

Now, each entry in the `favoriteColors` array will be validated to not be blank and to be at least 5 characters long.

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/All.html
2. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/AllValidator.html

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 72
UserPassword

New in version 2.1: This constraint is new in version 2.1.

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 his password, but needs to enter his old password for security.

This should **not** be used to validate a login form, since this is done automatically by the security system.

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	• message
Class	*UserPassword*[1]
Validator	*UserPasswordValidator*[2]

Basic Usage

Suppose you have a *PasswordChange* class, that's used in a form where the user can change his password by entering his 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/2.1/Symfony/Component/Security/Core/Validator/Constraint/UserPassword.html
2. http://api.symfony.com/2.1/Symfony/Component/Security/Core/Validator/Constraint/UserPasswordValidator.html

Listing 72-1
```yaml
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Form\Model\ChangePassword:
    properties:
        oldPassword:
            - Symfony\Component\Security\Core\Validator\Constraint\UserPassword:
                message: "Wrong value for your current password"
```

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 73
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
Class	*Type*[1]

Basic Usage

In the following example, we create two classes `Author` and `Address` that both have constraints on their properties. Furthermore, `Author` stores an `Address` instance in the `$address` property.

Listing 73-1
```php
// src/Acme/HelloBundle/Address.php
class Address
{
    protected $street;
    protected $zipCode;
}
```

Listing 73-2
```php
// src/Acme/HelloBundle/Author.php
class Author
{
    protected $firstName;
    protected $lastName;
    protected $address;
}
```

1. http://api.symfony.com/2.1/Symfony/Component/Validator/Constraints/Type.html

Listing 73-3
```yaml
# src/Acme/HelloBundle/Resources/config/validation.yml
Acme\HelloBundle\Address:
    properties:
        street:
            - NotBlank: ~
        zipCode:
            - NotBlank: ~
            - MaxLength: 5

Acme\HelloBundle\Author:
    properties:
        firstName:
            - NotBlank: ~
            - MinLength: 4
        lastName:
            - NotBlank: ~
```

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 73-4
```yaml
# src/Acme/HelloBundle/Resources/config/validation.yml
Acme\HelloBundle\Author:
    properties:
        address:
            - Valid: ~
```

If you validate an author with an invalid address now, you can see that the validation of the `Address` fields failed.

> AcmeHelloBundleAuthor.address.zipCode: This value is too long. It should have 5 characters or less

Options

traverse

type: `string` **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`.

Chapter 74

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 Symfony2. There may also be tags in other bundles you use that aren't listed here. For example, the AsseticBundle has several tags that aren't listed here.

Tag Name	Usage
data_collector	Create a class that collects custom data for the profiler
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_warmer	Register your service to be called during the cache warming process
kernel.event_listener	Listen to different events/hooks in Symfony
kernel.event_subscriber	To subscribe to a set of different events/hooks in Symfony
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
security.listener.factory	Necessary when creating a custom authentication system
swiftmailer.plugin	Register a custom SwiftMailer Plugin
templating.helper	Make your service available in PHP templates
translation.loader	Register a custom service that loads translations

twig.extension	Register a custom Twig Extension
validator.constraint_validator	Create your own custom validation constraint
validator.initializer	Register a service that initializes objects before validation

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*.

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 74-1
```php
// src/Acme/MainBundle/Form/Type/MyFormTypeExtension.php
namespace Acme\MainBundle\Form\Type\MyFormTypeExtension;

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 74-2
```yaml
services:
    main.form.type.my_form_type_extension:
        class: Acme\MainBundle\Form\Type\MyFormTypeExtension
        tags:
            - { name: form.type_extension, alias: field }
```

1. http://api.symfony.com/2.1/Symfony/Component/Form/Extension/Csrf/Type/FormTypeCsrfExtension.html
2. http://api.symfony.com/2.1/Symfony/Component/Form/FormTypeExtensionInterface.html
3. http://api.symfony.com/2.1/Symfony/Component/Form/AbstractTypeExtension.html

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 "field", use the "field" 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).

To add your own form type guesser, create a class that implements the *FormTypeGuesserInterface*[4] interface. Next, tag its service definition with `form.type_guesser` (it has no options).

To see an example of how this class might look, see the `ValidatorTypeGuesser` class in the `Form` component.

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`). 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*[5] interface:

Listing 74-3
```php
// src/Acme/MainBundle/Cache/MyCustomWarmer.php
namespace Acme\MainBundle\Cache;

use Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerInterface;

class MyCustomWarmer implements CacheWarmerInterface
{
    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 2.0, optional warmers are always executed anyways, so this function has no real effect.

To register your warmer with Symfony, give it the kernel.cache_warmer tag:

Listing 74-4
```yaml
services:
    main.warmer.my_custom_warmer:
        class: Acme\MainBundle\Cache\MyCustomWarmer
```

4. http://api.symfony.com/2.1/Symfony/Component/Form/FormTypeGuesserInterface.html
5. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/CacheWarmer/CacheWarmerInterface.html

```
4           tags:
5               - { name: kernel.cache_warmer, priority: 0 }
```

The `priority` value is optional, and defaults to 0. This value can be from -255 to 255, and the warmers will be executed in the order of their priority.

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*.

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 listener not listed here.

kernel.request

Listener Class Name	Priority
ProfilerListener[6]	1024
TestSessionListener[7]	192
SessionListener[8]	128
RouterListener[9]	32
LocaleListener[10]	16
Firewall[11]	8

kernel.controller

Listener Class Name	Priority
RequestDataCollector[12]	0

6. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
7. http://api.symfony.com/2.1/Symfony/Bundle/FrameworkBundle/EventListener/TestSessionListener.html
8. http://api.symfony.com/2.1/Symfony/Bundle/FrameworkBundle/EventListener/SessionListener.html
9. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/RouterListener.html
10. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/LocaleListener.html
11. http://api.symfony.com/2.1/Symfony/Component/Security/Http/Firewall.html
12. http://api.symfony.com/2.1/Symfony/Bundle/FrameworkBundle/DataCollector/RequestDataCollector.html

kernel.response

Listener Class Name	Priority
EsiListener[13]	0
ResponseListener[14]	0
ResponseListener[15]	0
ProfilerListener[16]	-100
TestSessionListener[17]	-128
WebDebugToolbarListener[18]	-128
StreamedResponseListener[19]	-1024

kernel.exception

Listener Class Name	Priority
ProfilerListener[20]	0
ExceptionListener[21]	-128

kernel.terminate

Listener Class Name	Priority
EmailSenderListener[22]	0

kernel.event_subscriber

Purpose: To subscribe to a set of different events/hooks in Symfony

New in version 2.1: The ability to add kernel event subscribers is new to 2.1.

To enable a custom subscriber, add it as a regular service in one of your configuration, and tag it with `kernel.event_subscriber`:

Listing 74-5
```
services:
    kernel.subscriber.your_subscriber_name:
        class: Fully\Qualified\Subscriber\Class\Name
```

13. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/EsiListener.html
14. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/ResponseListener.html
15. http://api.symfony.com/2.1/Symfony/Bundle/SecurityBundle/EventListener/ResponseListener.html
16. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
17. http://api.symfony.com/2.1/Symfony/Bundle/FrameworkBundle/EventListener/TestSessionListener.html
18. http://api.symfony.com/2.1/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
19. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/StreamedResponseListener.html
20. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
21. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
22. http://api.symfony.com/2.1/Symfony/Bundle/SwiftmailerBundle/EventListener/EmailSenderListener.html

```
4           tags:
5               - { name: kernel.event_subscriber }
```

Your service must implement the *SymfonyComponentEventDispatcherEventSubscriberInterface*[23] interface.

If your service is created by a factory, you **MUST** correctly set the `class` parameter for this tag to work correctly.

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 74-6
```
services:
    my_service:
        class: Fully\Qualified\Loader\Class\Name
        arguments: [@logger]
        tags:
            - { name: monolog.logger, channel: acme }
```

This works only when the logger service is a constructor argument, not when it is injected through a setter.

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.

Let's see how you can use the built-in `IntrospectionProcessor` to add the file, the line, the class and the method where the logger was triggered.

You can add a processor globally:

Listing 74-7
```
1  services:
2      my_service:
3          class: Monolog\Processor\IntrospectionProcessor
4          tags:
5              - { name: monolog.processor }
```

23. http://api.symfony.com/2.1/SymfonyComponentEventDispatcherEventSubscriberInterface.html

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 74-8
```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, handler: firephp }
```

You can also add a processor for a specific logging channel by using the `channel` attribute. This will register the processor only for the `security` logging channel used in the Security component:

Listing 74-9
```
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 74-10
```
services:
    routing.loader.your_loader_name:
        class: Fully\Qualified\Loader\Class\Name
        tags:
            - { name: routing.loader }
```

security.listener.factory

Purpose: Necessary when creating a custom authentication system

This tag is used when creating your own custom authentication system. For details, see *How to create a custom Authentication Provider*.

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`*[24] and your custom authentication listener extends *`AbstractAuthenticationListener`*[25], then your custom authentication listener will automatically have this tagged applied and it will function automatically.

security.voter

Purpose: To add a custom voter to Symfony's authorization logic

When you call `isGranted` on Symfony's 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*.

swiftmailer.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.plugin` (it has no options).

A SwiftMailer plugin must implement the `Swift_Events_EventListener` interface. For more information on plugins, see *SwiftMailer's Plugin Documentation*[26].

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 74-11
```
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

24. http://api.symfony.com/2.1/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.html
25. http://api.symfony.com/2.1/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.html
26. http://swiftmailer.org/docs/plugins.html

By default, translations are loaded form the filesystem in a variety of different formats (YAML, XLIFF, PHP, etc). If you need to load translations from some other source, first create a class that implements the *`LoaderInterface`*[27] interface:

Listing 74-12
```php
// src/Acme/MainBundle/Translation/MyCustomLoader.php
namespace Acme\MainBundle\Translation;

use Symfony\Component\Translation\Loader\LoaderInterface
use Symfony\Component\Translation\MessageCatalogue;

class MyCustomLoader implements LoaderInterface
{
    public function load($resource, $locale, $domain = 'messages')
    {
        $catalogue = new MessageCatalogue($locale);

        // some how load up some translations from the "resource"
        // then set them into the catalogue
        $catalogue->set('hello.world', 'Hello World!', $domain);

        return $catalogue;
    }
}
```

Your custom loader's `load` method is responsible for returning a *`MessageCatalogue`*[28].

Now, register your loader as a service and tag it with `translation.loader`:

Listing 74-13
```yaml
services:
    main.translation.my_custom_loader:
        class: Acme\MainBundle\Translation\MyCustomLoader
        tags:
            - { name: translation.loader, alias: bin }
```

Listing 74-14
```xml
<service id="main.translation.my_custom_loader"
class="Acme\MainBundle\Translation\MyCustomLoader">
    <tag name="translation.loader" alias="bin" />
</service>
```

Listing 74-15
```php
$container
    ->register('main.translation.my_custom_loader',
'Acme\MainBundle\Translation\MyCustomLoader')
    ->addTag('translation.loader', array('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.

27. http://api.symfony.com/2.1/Symfony/Component/Translation/Loader/LoaderInterface.html
28. http://api.symfony.com/2.1/Symfony/Component/Translation/MessageCatalogue.html

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.

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 74-16
```
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*[29] 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*[30] 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 74-17
```
services:
    twig.extension.intl:
        class: Twig_Extensions_Extension_Intl
        tags:
            - { name: twig.extension }
```

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`*[31] interface. Then, tag it with the `validator.initializer` tag (it has no options).

For an example, see the `EntityInitializer` class inside the Doctrine Bridge.

29. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension
30. http://github.com/fabpot/Twig-extensions
31. http://api.symfony.com/2.1/Symfony/Component/Validator/ObjectInitializerInterface.html

Chapter 75

Requirements for running Symfony2

To run Symfony2, 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 75-1
```
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

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)
- 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.autostart = 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.