# symfony-docs-it Documentation

*Release 2.7*

**Apr 10, 2017**

# Contents

CHAPTER 1

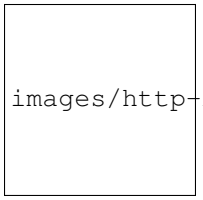
# Symfony e fondamenti di HTTP

Congratulazioni! Imparando Symfony, si tende a essere sviluppatori web più *produttivi*, *versatili* e *popolari* (in realtà, per quest'ultimo dovete sbrigarvela da soli). Symfony è costruito per tornare alle basi: per sviluppare strumenti che consentono di sviluppare più velocemente e costruire applicazioni più robuste, anche andando fuori strada. Symfony è costruito sulle migliori idee prese da diverse tecnologie: gli strumenti e i concetti che si stanno per apprendere rappresentano lo sforzo di centinaia di persone, in molti anni. In altre parole, non si sta semplicemente imparando "Symfony", si stanno imparando i fondamenti del web, le pratiche migliori per lo sviluppo e come usare tante incredibili librerie PHP, all'interno o dipendenti da Symfony. Tenetevi pronti.

Fedele alla filosofia di Symfony, questo capitolo inizia spiegando il concetto fondamentale comune allo sviluppo web: HTTP. Indipendentemente dalla propria storia o dal linguaggio di programmazione preferito, questo capitolo andrebbe letto da tutti.

## HTTP è semplice

HTTP (Hypertext Transfer Protocol per i geek) è un linguaggio testuale che consente a due macchine di comunicare tra loro. Tutto qui! Per esempio, se si controlla l'ultima vignetta di xkcd, ha luogo la seguente conversazione (approssimata):



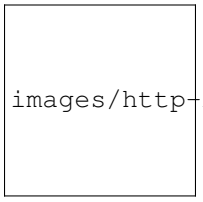

E sebbene il linguaggio usato in realtà sia un po' più formale, è ancora assolutamente semplice. HTTP è il termine usato per descrivere tale semplice linguaggio testuale. Non importa in quale linguaggio si sviluppi sul web, lo scopo di un server è *sempre* quello di interpretare semplici richieste testuali e restituire semplici risposte testuali.

Symfony è costruito fin dalle basi attorno a questa realtà. Che lo si comprenda o meno, HTTP è qualcosa che si usa ogni giorno. Con Symfony, si imparerà come padroneggiarlo.

## Passo 1: il client invia una richiesta

Ogni conversazione sul web inizia con una *richiesta*. La richiesta è un messaggio testuale creato da un client (per esempio un browser, un'applicazione mobile, ecc.) in uno speciale formato noto come HTTP. Il client invia la richiesta a un server e quindi attende una risposta.

Diamo uno sguardo alla prima parte dell'interazione (la richiesta) tra un browser e il server web di xkcd:




Nel gergo di HTTP, questa richiesta apparirebbe in realtà in questo modo:

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

Questo semplice messaggio comunica *ogni cosa* necessaria su quale risorsa esattamente il client sta richiedendo. La prima riga di ogni richiesta HTTP è la più importante e contiene due cose: l'URI e il metodo HTTP.

L'URI (p.e. `/`, `/contact`, ecc.) è l'indirizzo univoco o la locazione che identifica la risorsa che il client vuole. Il metodo HTTP (p.e. `GET`) definisce cosa si vuole *fare* con la risorsa. I metodi HTTP sono *verbi* della richiesta e definiscono i pochi modi comuni in cui si può agire sulla risorsa:

| | |
|---|---|
| *GET* | Recupera la risorsa dal server |
| *POST* | Crea una risorsa sul server |
| *PUT* | Aggiorna la risorsa sul server |
| *DELETE* | Elimina la risorsa dal server |

Tenendo questo a mente, si può immaginare come potrebbe apparire una richiesta HTTP per cancellare una specifica voce di un blog, per esempio:

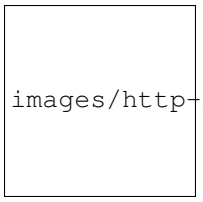
```
DELETE /blog/15 HTTP/1.1
```

**Note:** Ci sono in realtà nove metodi HTTP definiti dalla specifica HTTP, ma molti di essi non sono molto usati o supportati. In realtà, molti browser moderni non supportano nemmeno i metodi `PUT` e `DELETE`.

In aggiunta alla prima linea, una richiesta HTTP contiene sempre altre linee di informazioni, chiamate header. Gli header possono fornire un ampio raggio di informazioni, come l'`Host` richiesto, i formati di risposta accettati dal client (`Accept`) e l'applicazione usata dal client per eseguire la richiesta (`User-Agent`). Esistono molti altri header, che possono essere trovati nella pagina di Wikipedia Lista di header HTTP.

## Passo 2: Il server restituisce una risposta

Una volta che il server ha ricevuto la richiesta, sa esattamente la risorsa di cui il client ha bisogno (tramite l'URI) e cosa vuole fare il client con tale risorsa (tramite il metodo). Per esempio, nel caso di una richiesta GET, il server prepara la risorsa e la restituisce in una risposta HTTP. Consideriamo la risposta del server web di xkcd:




Tradotto in HTTP, la risposta rimandata al browser assomiglierà a questa:

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html

<html>
  <!-- ... HTML della vignetta di xkcd -->
</html>
```

La risposta HTTP contiene la risorsa richiesta (il contenuto HTML, in questo caso). oltre che altre informazioni sulla risposta. La prima riga è particolarmente importante e contiene il codice di stato della risposta HTTP (200, in questo caso). Il codice di stato comunica il risultato globale della richiesta al client. La richiesta è andata a buon fine? C'è stato un errore? Diversi codici di stato indicano successo, errore o che il client deve fare qualcosa (p.e. rimandare a un'altra pagina). Una lista completa può essere trovata nella pagina di Wikipedia Elenco dei codici di stato HTTP.

Come la richiesta, una risposta HTTP contiene parti aggiuntive di informazioni, note come header. Per esempio, un importante header di risposta HTTP è `Content-Type`. Il corpo della risorsa stessa potrebbe essere restituito in molti formati diversi, inclusi HTML, XML o JSON, mentre l'header `Content-Type` usa i tipi di media di Internet, come `text/html`, per dire al client quale formato è restituito. Una lista di tipi di media comuni si può trovare sulla voce di Wikipedia Lista di tipi di media comuni.

Esistono molti altri header, alcuni dei quali molto potenti. Per esempio, alcuni header possono essere usati per creare un potente sistema di cache.

### Richieste, risposte e sviluppo web

Questa conversazione richiesta-risposta è il processo fondamentale che guida tutta la comunicazione sul web. Questo processo è tanto importante e potente, quanto inevitabilmente semplice.

L'aspetto più importante è questo: indipendentemente dal linguaggio usato, il tipo di applicazione costruita (web, mobile, API JSON) o la filosofia di sviluppo seguita, lo scopo finale di un'applicazione è **sempre** quello di capire ogni richiesta e creare e restituire un'appropriata risposta.

L'architettura di Symfony è strutturata per corrispondere a questa realtà.

---

**Tip:** Per saperne di più sulla specifica HTTP, si può leggere la RFC HTTP 1.1 originale o la HTTP Bis, che è uno sforzo attivo di chiarire la specifica originale. Un importante strumento per verificare sia gli header di richiesta che quelli di risposta durante la navigazione è l'estensione Live HTTP Headers di Firefox.

---

## Richieste e risposte in PHP

Dunque, come interagire con la "richiesta" e creare una "risposta" quando si usa PHP? In realtà, PHP astrae un po' l'intero processo:

```
$uri = $_SERVER['REQUEST_URI'];
$pippo = $_GET['pippo'];

header('Content-type: text/html');
echo 'L\'URI richiesto è: '.$uri;
echo 'Il valore del parametro "pippo" è: '.$pippo;
```

Per quanto possa sembrare strano, questa piccola applicazione di fatto prende informazioni dalla richiesta HTTP e le usa per creare una risposta HTTP. Invece di analizzare il messaggio di richiesta HTTP grezzo, PHP prepara della variabili superglobali, come `$_SERVER` e `$_GET`, che contengono tutte le informazioni dalla richiesta. Similmente, invece di restituire un testo di risposta formattato come da HTTP, si può usare la funzione `header()` per creare header di risposta e stampare semplicemente il contenuto, che sarà la parte di contenuto del messaggio di risposta. PHP creerà una vera risposta HTTP e la restituirà al client:

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

L'URI richiesto è: /testing?pippo=symfony
Il valore del parametro "pippo" è: symfony
```

## Richieste e risposte in Symfony

Symfony fornisce un'alternativa all'approccio grezzo di PHP, tramite due classi che consentono di interagire con richiesta e risposta HTTP in modo più facile. La classe `Symfony\Component\HttpFoundation\Request` è una semplice rappresentazione orientata agli oggetti del messaggio di richiesta HTTP. Con essa, si hanno a portata di mano tutte le informazioni sulla richiesta:

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// l'URI richiesto (p.e. /about) tranne ogni parametro
$request->getPathInfo();

// recupera rispettivamente le variabili GET e POST
$request->query->get('pippo');
$request->request->get('pluto', 'valore predefinito se pluto non esiste');

// recupera le variabili SERVER
$request->server->get('HTTP_HOST');

// recupera un'istanza di UploadedFile identificata da pippo
$request->files->get('pippo');

// recupera il valore di un COOKIE
$request->cookies->get('PHPSESSID');

// recupera un header di risposta HTTP, con chiavi normalizzate e minuscole
$request->headers->get('host');
$request->headers->get('content_type');

$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // un array di lingue accettate dal client
```

Come bonus, la classe `Request` fa un sacco di lavoro in sottofondo, di cui non ci si dovrà mai preoccupare. Per esempio, il metodo `isSecure()` verifica **tre** diversi valori in PHP che possono indicare se l'utente si stia connettendo o meno tramite una connessione sicura (cioè HTTPS).

**ParameterBags e attributi di Request**

Come visto in precedenza, le variabili `$_GET` e `$_POST` sono accessibili rispettivamente tramite le proprietà pubbliche `query` e `request`. Entrambi questi oggetti sono oggetti della classe `Symfony\Component\HttpFoundation\ParameterBag`, che ha metodi come **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::get'**, **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::has'**, **:method:'Symfony\\Component\\HttpFoundation\\Par** e altri. In effetti, ogni proprietà pubblica usata nell'esempio precedente è un'istanza di ParameterBag. La classe Request ha anche una proprietà pubblica `attributes`, che contiene dati speciali relativi a come l'applicazione funziona internamente. Per il framework Symfony, `attributes` contiene valori restituiti dalla rotta corrispondente, come `_controller`, `id` (se si ha un parametro `{id}`), e anche il nome della rotta stessa (`_route`). La proprietà `attributes` è pensata apposta per essere un posto in cui preparare e memorizzare informazioni sulla richiesta relative al contesto.

Symfony fornisce anche una classe `Response`: una semplice rappresentazione PHP di un messaggio di risposta HTTP. Questo consente a un'applicazione di usare un'interfaccia orientata agli oggetti per costruire la risposta che occorre restituire al client:

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response();

$response->setContent('<html><body><h1>Ciao mondo!</h1></body></html>');
$response->setStatusCode(Response::HTTP_OK);
$response->headers->set('Content-Type', 'text/html');

// stampa gli header HTTP seguiti dal contenuto
$response->send();
```

Se Symfony offrisse solo questo, si avrebbe già a disposizione un kit di strumenti per accedere facilmente alle informazioni di richiesta e un'interfaccia orientata agli oggetti per creare la risposta. Anche imparando le molte potenti caratteristiche di Symfony, si tenga a mente che lo scopo di un'applicazione è sempre quello di *interpretare una richiesta e creare l'appropriata risposta, basata sulla logica dell'applicazione*.

**Tip:** Le classi `Request` e `Response` fanno parte di un componente a sé stante incluso con Symfony, chiamato `HttpFoundation`. Questo componente può essere usato in modo completamente indipendente da Symfony e fornisce anche classi per gestire sessioni e caricamenti di file.

## Il viaggio dalla richiesta alla risposta

Come lo stesso HTTP, gli oggetti `Request` e `Response` sono molto semplici. La parte difficile nella costruzione di un'applicazione è la scrittura di quello che sta in mezzo. In altre parole, il vero lavoro consiste nello scrivere il codice che interpreta l'informazione della richiesta e crea la risposta.

Un'applicazione probabilmente deve fare molte cose, come inviare email, gestire form, salvare dati in una base dati, rendere pagine HTML e proteggere contenuti. Come si può gestire tutto questo e mantenere al contempo il codice

organizzato e mantenibile?

Symfony è stato creato per risolvere questi problemi.

## Il front controller

Le applicazioni erano tradizionalmente costruite in modo che ogni "pagina" di un sito fosse un file fisico:

```
index.php
contact.php
blog.php
```

Ci sono molti problemi con questo approccio, inclusa la flessibilità degli URL (che succede se si vuole cambiare `blog.php` con `news.php` senza rompere tutti i collegamenti?) e il fatto che ogni file *deve* includere manualmente alcuni file necessari, in modo che la sicurezza, le connessioni alla base dati e l'aspetto del sito possano rimanere coerenti.

Una soluzione molto migliore è usare un front controller: un unico file PHP che gestisce ogni richiesta che arriva all'applicazione. Per esempio:

| | |
|---|---|
| `/index.php` | esegue `index.php` |
| `/index.php/contact` | esegue `index.php` |
| `/index.php/blog` | esegue `index.php` |

**Tip:** Usando il modulo `mod_rewrite` di Apache (o moduli equivalenti di altri server), gli URL possono essere facilmente puliti per essere semplicemente `/`, `/contact` e `/blog`.

Ora ogni richiesta è gestita esattamente nello stesso modo. Invece di singoli URL che eseguono diversi file PHP, è *sempre* eseguito il front controller, e il dirottamento di URL diversi sulle diverse parti dell'applicazione è gestito internamente. Questo risolve entrambi i problemi dell'approccio originario. Quasi tutte le applicazioni web moderne fanno in questo modo, incluse applicazioni come WordPress.

## Restare organizzati

Ma, all'interno del nostro front controller, come possiamo sapere quale pagina debba essere resa e come poterla rendere in modo facile? In un modo o nell'altro, occorre verificare l'URI in entrata ed eseguire parti diverse di codice, a seconda di tale valore. Le cose possono peggiorare rapidamente:

```
// index.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // il percorso dell'URI richiesto

if (in_array($path, array('', '/'))) {
    $response = new Response('Benvenuto nella homepage.');
} elseif ('/contact' === $path) {
    $response = new Response('Contattaci');
} else {
    $response = new Response('Pagina non trovata.', Response::HTTP_NOT_FOUND);
}
$response->send();
```

La soluzione a questo problema può essere difficile. Fortunatamente, è *esattamente* quello che Symfony è studiato per fare.

## Il flusso di un'applicazione Symfony

Quando si lascia a Symfony la gestione di ogni richiesta, la vita è molto più facile. Symfony segue lo stesso semplice schema per ogni richiesta:

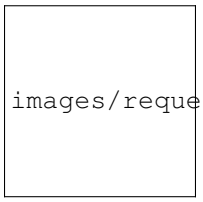



Fig. 1.1: Le richieste in entrata sono interpretate dal routing e passate alle funzioni del controllore, che restituisce oggetti `Response`.

Ogni "pagina" del proprio sito è definita in un file di configurazione delle rotte, che mappa diversi URL su diverse funzioni PHP. Il compito di ogni funzione PHP, chiamata controllore, è di usare l'informazione della richiesta, insieme a molti altri strumenti resi disponibili da Symfony, per creare e restituire un oggetto `Response`. In altre parole, il controllore è il posto in cui va il *proprio* codice: è dove si interpreta la richiesta e si crea la risposta.

È così facile! Rivediamolo:

- Ogni richiesta esegue un file front controller;
- Il sistema delle rotte determina quale funzione PHP deve essere eseguita, in base all'informazione proveniente dalla richiesta e alla configurazione delle rotte creata;
- La giusta funzione PHP è eseguita, con il proprio codice che crea e restituisce l'oggetto `Response` appropriato.

## Un richiesta Symfony in azione

Senza entrare troppo in dettaglio, vediamo questo processo in azione. Supponiamo di voler aggiungere una pagina `/contact` alla nostra applicazione Symfony. Primo, iniziamo aggiungendo una voce per `/contact` nel file di configurazione delle rotte:

Quando qualcuno vista la pagina `/contact`, questa rotta viene corrisposta e il controllore specificato è eseguito. Come si imparerà nel capitolo delle rotte, la stringa `AppBundle:Main:contact` è una sintassi breve che punta a uno specifico metodo PHP `contactAction` in una classe chiamata `MainController`:

```
// src/AppBundle/Controller/MainController.php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class MainController
{
    public function contactAction()
    {
        return new Response('<h1>Contattaci!</h1>');
    }
}
```

In questo semplice esempio, il controllore semplicemente crea un oggetto `Symfony\Component\HttpFoundation\Response` con il codice HTML `<h1>Contattaci!</h1>`. Nel capitolo sul controllore, si imparerà come un controllore possa rendere dei template, consentendo al codice di "presentazione" (cioè a qualsiasi cosa che scrive effettivamente HTML) di vivere in un file template separato. Questo consente al controllore di preoccuparsi solo delle cose difficili: interagire con la base dati, gestire l'invio di dati o l'invio di messaggi email.

## Symfony: costruire un'applicazione, non degli strumenti

Sappiamo dunque che lo scopo di un'applicazione è interpretare ogni richiesta in entrata e creare un'appropriata risposta. Al crescere di un'applicazione, diventa sempre più difficile mantenere il codice organizzato e mantenibile. Invariabilmente, gli stessi complessi compiti continuano a presentarsi: persistere nella base dati, rendere e riusare template, gestire form, inviare email, validare i dati degli utenti e gestire la sicurezza.

La buona notizia è che nessuno di questi problemi è unico. Symfony fornisce un framework pieno di strumenti che consentono di costruire un'applicazione, non di costruire degli strumenti. Con Symfony, nulla viene imposto: si è liberi di usare l'intero framework oppure un solo pezzo di Symfony.

### Strumenti isolati: i *componenti* di Symfony

Cos'*è* dunque Symfony? Primo, è un insieme di oltre venti librerie indipendenti, che possono essere usate in *qualsiasi* progetto PHP. Queste librerie, chiamate *componenti di Symfony*, contengono qualcosa di utile per quasi ogni situazione, comunque sia sviluppato il proprio progetto. Solo per nominarne alcuni:

**HttpFoundation** Contiene le classi `Request` e `Response`, insieme ad altre classi per gestire sessioni e caricamenti di file;

**Routing** Sistema di rotte potente e veloce, che consente di mappare uno specifico URI (p.e. `/contact`) ad alcune informazioni su come tale richiesta andrebbe gestita (p.e. eseguendo il metodo `contactAction()`);

**Form** Un framework completo e flessibile per creare form e gestire invii di dati;

**Validator** Un sistema per creare regole sui dati e quindi validarli, sia che i dati inviati dall'utente seguano o meno tali regole;

**Templating** Un insieme di strumenti per rendere template, gestire l'ereditarietà dei template (p.e. un template è decorato con un layout) ed eseguire altri compiti comuni sui template;

**Security** Una potente libreria per gestire tutti i tipi di sicurezza all'interno di un'applicazione;

**Translation** Un framework per tradurre stringhe nella propria applicazione.

Tutti questi componenti sono disaccoppiati e possono essere usati in *qualsiasi* progetto PHP, indipendentemente dall'uso del framework Symfony. Ogni parte di essi è stata realizzata per essere usata se necessario e sostituita in caso contrario.

### La soluzione completa il *framework* Symfony

Cos'*è* quindi il *framework* Symfony? Il *framework Symfony* è una libreria PHP che esegue due compiti distinti:

1. Fornisce una selezione di componenti (cioè i componenti di Symfony) e librerie di terze parti (p.e. Swiftmailer per l'invio di email);
2. Fornisce una pratica configurazione e una libreria "collante", che lega insieme tutti i pezzi.

Lo scopo del framework è integrare molti strumenti indipendenti, per fornire un'esperienza coerente allo sviluppatore. Anche il framework stesso è un bundle (cioè un plugin) che può essere configurato o sostituito interamente.

Symfony fornisce un potente insieme di strumenti per sviluppare rapidamente applicazioni web, senza imposizioni sulla propria applicazione. Gli utenti normali possono iniziare velocemente a sviluppare usando una distribuzione di Symfony, che fornisce uno scheletro di progetto con configurazioni predefinite ragionevoli. Gli utenti avanzati hanno il cielo come limite.

CHAPTER 2

# Symfony contro PHP puro

**Perché Symfony è meglio che aprire un file e scrivere PHP puro?**

Questo capitolo è per chi non ha mai usato un framework PHP, non ha familiarità con la filosofia MVC, oppure semplicemente si chiede il motivo di tutto il *clamore* su Symfony. Invece di *raccontare* che Symfony consente di sviluppare software più rapidamente e in modo migliore che con PHP puro, ve lo faremo vedere.

In questo capitolo, scriveremo una semplice applicazione in PHP puro e poi la rifattorizzeremo per essere più organizzata. Viaggeremo nel tempo, guardando le decisioni che stanno dietro ai motivi per cui lo sviluppo web si è evoluto durante gli ultimi anni per diventare quello che è ora.

Alla fine, vedremo come Symfony possa salvarci da compiti banali e consentirci di riprendere il controllo del nostro codice.

## Un semplice blog in PHP puro

In questo capitolo, costruiremo un'applicazione blog usando solo PHP puro. Per iniziare, creiamo una singola pagina che mostra le voci del blog, che sono state memorizzate nella base dati. La scrittura in puro PHP è sporca e veloce:

```
<?php
// index.php
$link = mysql_connect('localhost', 'mioutente', 'miapassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);
?>

<!DOCTYPE html>
<html>
    <head>
        <title>Lista dei post</title>
    </head>
    <body>
        <h1>Lista dei post</h1>
```

```
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>

<?php
mysql_close($link);
?>
```

Veloce da scrivere, rapido da eseguire e, al crescere dell'applicazione, impossibile da mantenere. Ci sono diversi problemi che occorre considerare:

- **Niente verifica degli errori**: Che succede se la connessione alla base dati fallisce?
- **Scarsa organizzazione**: Se l'applicazione cresce, questo singolo file diventerà sempre più immantenibile. Dove inserire il codice per gestire la compilazione di un form? Come validare i dati? Dove mettere il codice per inviare delle email?
- **Difficoltà nel riusare il codice**: Essendo tutto in un solo file, non c'è modo di riusare alcuna parte dell'applicazione per altre "pagine" del blog.

---

**Note:** Un altro problema non menzionato è il fatto che la base dati è legata a MySQL. Sebbene non affrontato qui, Symfony integra in pieno Doctrine, una libreria dedicata all'astrazione e alla mappatura della base dati.

---

## Isolare la presentazione

Il codice può beneficiare immediatamente dalla separazione della "logica" dell'applicazione dal codice che prepara la "presentazione" in HTML:

```
<?php
// index.php
$link = mysql_connect('localhost', 'mioutente', 'miapassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);

$posts = array();
while ($row = mysql_fetch_assoc($result)) {
    $posts[] = $row;
}

mysql_close($link);

// include il codice HTML di presentazione
require 'templates/list.php';
```

Il codice HTML ora è in un file separato (`templates/list.php`), che è essenzialmente un file HTML che usa una sintassi PHP per template:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Lista dei post</title>
    </head>
    <body>
        <h1>Lista dei post</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/read?id=<?php echo $post['id'] ?>">
                    <?php echo $post['title'] ?>
                </a>
            </li>
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

Per convenzione, il file che contiene tutta la logica dell'applicazione, cioè `index.php`, è noto come "controllore". Il termine controllore è una parola che ricorrerà spesso, quale che sia il linguaggio o il framework scelto. Si riferisce semplicemente alla parte del *proprio* codice che processa l'input proveniente dall'utente e prepara la risposta.

In questo caso, il nostro controllore prepara i dati estratti dalla base dati e quindi include un template, per presentare tali dati. Con il controllore isolato, è possibile cambiare facilmente *solo* il file template necessario per rendere le voci del blog in un qualche altro formato (p.e. `list.json.php` per il formato JSON).

## Isolare la logica dell'applicazione (il dominio)

Finora l'applicazione contiene una singola pagina. Ma se una seconda pagina avesse bisogno di usare la stessa connessione alla base dati, o anche lo stesso array di post del blog? Rifattorizziamo il codice in modo che il comportamento centrale e le funzioni di accesso ai dati dell'applicazioni siano isolati in un nuovo file, chiamato `model.php`:

```
<?php
// model.php
function open_database_connection()
{
    $link = mysql_connect('localhost', 'mioutente', 'miapassword');
    mysql_select_db('blog_db', $link);

    return $link;
}

function close_database_connection($link)
{
    mysql_close($link);
}

function get_all_posts()
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
```

```
    }
    close_database_connection($link);

    return $posts;
}
```

**Tip:** Il nome `model.php` è usato perché la logica e l'accesso ai dati di un'applicazione sono tradizionalmente noti come il livello del "modello". In un'applicazione ben organizzata la maggior parte del codice che rappresenta la "logica di business" dovrebbe stare nel modello (invece che stare in un controllore). Diversamente da questo esempio, solo una parte (o niente) del modello riguarda effettivamente l'accesso a una base dati.

Il controllore (`index.php`) è ora molto semplice:

```
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Ora, l'unico compito del controllore è prendere i dati dal livello del modello dell'applicazione (il modello) e richiamare un template per rendere tali dati. Questo è un esempio molto semplice del pattern model-view-controller.

## Isolare il layout

A questo punto, l'applicazione è stata rifattorizzata in tre parti distinte, offrendo diversi vantaggi e l'opportunità di riusare quasi tutto su pagine diverse.

L'unica parte del codice che *non può* essere riusata è il layout. Sistemiamo questo aspetto, creando un nuovo file `layout.php`:

```
<!-- templates/layout.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $title ?></title>
    </head>
    <body>
        <?php echo $content ?>
    </body>
</html>
```

Il template (`templates/list.php`) ora può essere semplificato, per "estendere" il layout:

```
<?php $title = 'Lista dei post' ?>

<?php ob_start() ?>
    <h1>Lista dei post</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/read?id=<?php echo $post['id'] ?>">
                <?php echo $post['title'] ?>
            </a>
        </li>
```

```
        <?php endforeach ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Qui abbiamo introdotto una metodologia che consente il riuso del layout. Sfortunatamente, per poterlo fare, si è costretti a usare alcune brutte funzioni PHP (`ob_start()`, `ob_get_clean()`) nel template. Symfony usa un componente Templating, che consente di poter fare ciò in modo pulito e facile. Lo vedremo in azione tra poco.

## Aggiungere al blog una pagina "show"

La pagina "elenco" del blog è stata ora rifattorizzata in modo che il codice sia meglio organizzato e riusabile. Per provarlo, aggiungiamo al blog una pagina "mostra", che mostra un singolo post del blog identificato dal parametro `id`.

Per iniziare, creiamo nel file `model.php` una nuova funzione, che recupera un singolo risultato del blog a partire da un id dato:

```
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $id = intval($id);
    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;
    $result = mysql_query($query);
    $row = mysql_fetch_assoc($result);

    close_database_connection($link);

    return $row;
}
```

Quindi, creiamo un file chiamato `show.php`, il controllore per questa nuova pagina:

```
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Infine, creiamo un nuovo file template, `templates/show.php`, per rendere il singolo post del blog:

```
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?php echo $post['title'] ?></h1>

    <div class="date"><?php echo $post['date'] ?></div>
    <div class="body">
        <?php echo $post['body'] ?>
    </div>
<?php $content = ob_get_clean() ?>
```

```
<?php include 'layout.php' ?>
```

La creazione della seconda pagina è stata molto facile e non ha implicato alcuna duplicazione di codice. Tuttavia, questa pagina introduce alcuni altri problemi, che un framework può risolvere. Per esempio, un parametro `id` mancante o non valido causerà un errore nella pagina. Sarebbe meglio se facesse rendere una pagina 404, ma non possiamo ancora farlo in modo facile. Inoltre, avendo dimenticato di pulire il parametro `id` con la funzione `mysql_real_escape_string()`, la base dati è a rischio di attacchi di tipo SQL injection.

Un altro grosso problema è che ogni singolo controllore deve includere il file `model.php`. Che fare se poi occorresse includere un secondo file o eseguire un altro compito globale (p.e. garantire la sicurezza)? Nella situazione attuale, tale codice dovrebbe essere aggiunto a ogni singolo file. Se lo si dimentica in un file, speriamo che non sia qualcosa legato alla sicurezza.

## Un "front controller" alla riscossa

La soluzione è usare un front controller: un singolo file PHP attraverso il quale *tutte* le richieste sono processate. Con un front controller, gli URI dell'applicazione cambiano un poco, ma iniziano a diventare più flessibili:

```
Senza un front controller
/index.php          => Pagina della lista dei post (index.php eseguito)
/show.php           => Pagina che mostra il singolo post (show.php eseguito)

Con index.php come front controller
/index.php          => Pagina della lista dei post (index.php eseguito)
/index.php/show     => Pagina che mostra il singolo post (index.php eseguito)
```

**Tip:** La parte dell'URI `index.php` può essere rimossa se si usano le regole di riscrittura di Apache (o equivalente). In questo caso, l'URI risultante della pagina che mostra il post sarebbe semplicemente `/show`.

Usando un front controller, un singolo file PHP (`index.php` in questo caso) rende *ogni* richiesta. Per la pagina che mostra il post, `/index.php/show` eseguirà in effetti il file `index.php`, che ora è responsabile per gestire internamente le richieste, in base all'URI. Come vedremo, un front controller è uno strumento molto potente.

### Creazione del front controller

Stiamo per fare un **grosso** passo avanti con l'applicazione. Con un solo file a gestire tutte le richieste, possiamo centralizzare cose come gestione della sicurezza, caricamento della configurazione, rotte. In questa applicazione, `index.php` deve essere abbastanza intelligente da rendere la lista dei post *oppure* il singolo post, in base all'URI richiesto:

```
<?php
// index.php

// carica e inizializza le librerie globali
require_once 'model.php';
require_once 'controllers.php';

// dirotta internamente la richiesta
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' == $uri) {
    list_action();
```

```
} elseif ('/index.php/show' == $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Pagina non trovata</h1></body></html>';
}
```

Per una migliore organizzazione, entrambi i controllori (precedentemente `index.php` e `show.php`) sono ora funzioni PHP, entrambe spostate in un file separato, `controllers.php`:

```
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

Come front controller, `index.php` ha assunto un nuovo ruolo, che include il caricamento delle librerie principali e la gestione delle rotte dell'applicazione, in modo che sia richiamato uno dei due controllori (le funzioni `list_action()` e `show_action()`). In realtà. il front controller inizia ad assomigliare molto al meccanismo con cui Symfony gestisce le richieste.

---

**Tip:** Un altro vantaggio di un front controller sono gli URL flessibili. Si noti che l'URL della pagina del singolo post può essere cambiato da `/show` a `/read` solo cambiando un unico punto del codice. Prima, occorreva rinominare un file. In Symfony, gli URL sono ancora più flessibili.

---

Finora, l'applicazione si è evoluta da un singolo file PHP a una struttura organizzata e che consente il riuso del codice. Dovremmo essere contenti, ma non ancora soddisfatti. Per esempio, il sistema delle rotte è instabile e non riconosce che la pagina della lista (`/index.php`) dovrebbe essere accessibile anche tramite `/` (con le regole di riscrittura di Apache). Inoltre, invece di sviluppare il blog, abbiamo speso diverso tempo sull'"architettura" del codice (p.e. rotte, richiamo dei controllori, template, ecc.). Ulteriore tempo sarebbe necessario per gestire l'invio di form, la validazione dell'input, i log e la sicurezza. Perché dovremmo reinventare soluzioni a tutti questi problemi comuni?

## Aggiungere un tocco di Symfony

Symfony alla riscossa! Prima di usare effettivamente Symfony, occorre accertarsi che PHP sappia come trovare le classi di Symfony. Possiamo farlo grazie all'autoloader fornito da Symfony. Un autoloader è uno strumento che rende possibile l'utilizzo di classi PHP senza includere esplicitamente il file che contiene la classe.

Nella cartella radice, creare un file `composer.json` con il seguente contenuto:

```
{
    "require": {
        "symfony/symfony": "2.6.*"
    },
    "autoload": {
        "files": ["model.php","controllers.php"]
    }
}
```

Quindi, scaricare Composer ed eseguire il seguente comando, che scaricherà Symfony in una cartella `vendor/`:

```
$ composer install
```

Oltre a scaricare le dipendenza, Composer genera un file `vendor/autoload.php`, che si occupa di auto-caricare tutti i file del framework Symfony, nonché dei file menzionati nella sezione autoload di `composer.json`.

Una delle idee principali della filosofia di Symfony è che il compito principale di un'applicazione sia quello di interpretare ogni richiesta e restituire una risposta. A tal fine, Symfony fornice sia una classe `Symfony\Component\HttpFoundation\Request` che una classe `Symfony\Component\HttpFoundation\Response`. Queste classi sono rappresentazioni orientate agli oggetti delle richieste grezze HTTP processate e delle risposte HTTP restituite. Usiamole per migliorare il nostro blog:

```
<?php
// index.php
require_once 'vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ('/' == $uri) {
    $response = list_action();
} elseif ('/show' == $uri && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Pagina non trovata</h1></body></html>';
    $response = new Response($html, Response::HTTP_NOT_FOUND);
}

// mostra gli header e invia la risposta
$response->send();
```

I controllori sono ora responsabili di restituire un oggetto `Response`. Per rendere le cose più facili, si può aggiungere una nuova funzione `render_template()`, che si comporta un po' come il sistema di template di Symfony:

```
// controllers.php
use Symfony\Component\HttpFoundation\Response;

function list_action()
{
    $posts = get_all_posts();
    $html = render_template('templates/list.php', array('posts' => $posts));

    return new Response($html);
}

function show_action($id)
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', array('post' => $post));

    return new Response($html);
}
```

```
// funzione aiutante per rendere i template
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

Prendendo una piccola parte di Symfony, l'applicazione è diventata più flessibile e più affidabile. La classe `Request` fornisce un modo di accedere alle informazioni sulla richiesta HTTP. Nello specifico, il metodo `getPathInfo()` restituisce un URI più pulito (restituisce sempre `/show` e mai `/index.php/show`). In questo modo, anche se l'utente va su `/index.php/show`, l'applicazione è abbastanza intelligente per dirottare la richiesta a `show_action()`.

L'oggetto `Response` dà flessibilità durante la costruzione della risposta HTTP, consentendo di aggiungere header e contenuti HTTP tramite un'interfaccia orientata agli oggetti. Mentre in questa applicazione le risposte molto semplici, tale flessibilità ripagherà quando l'applicazione cresce.

## L'applicazione di esempio in Symfony

Il blog ha fatto *molta* strada, ma contiene ancora troppo codice per un'applicazione così semplice. Durante il cammino, abbiamo anche inventato un semplice sistema di rotte e un metodo che usa `ob_start()` e `ob_get_clean()` per rendere i template. Se, per qualche ragione, si avesse bisogno di continuare a costruire questo "framework" da zero, si potrebbero almeno utilizzare i componenti Routing e Templating, che già risolvono questi problemi.

Invece di risolvere nuovamente problemi comuni, si può lasciare a Symfony il compito di occuparsene. Ecco la stessa applicazione di esempio, ora costruita in Symfony:

```
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')
            ->getManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
            ->execute();

        return $this->render('Blog/list.html.php', array('posts' => $posts));
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getManager()
            ->getRepository('AppBundle:Post')
            ->find($id);

        if (!$post) {
```

```
            // mostra la pagina 404 page not found
            throw $this->createNotFoundException();
        }

        return $this->render('Blog/show.html.php', array('post' => $post));
    }
}
```

I due controllori sono ancora leggeri. Ognuno usa la libreria ORM Doctrine per recuperare oggetti dalla base dati e il componente `Templating` per rendere un template e restituire un oggetto `Response`. Il template della lista è ora un po' più semplice:

```
<!-- app/Resources/views/Blog/list.html.php -->
<?php $view->extend('layout.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>Lista dei post</h1>
<ul>
    <?php foreach ($posts as $post): ?>
    <li>
        <a href="<?php echo $view['router']->generate(
            'blog_show',
            array('id' => $post->getId())
        ) ?>">
            <?php echo $post->getTitle() ?>
        </a>
    </li>
    <?php endforeach ?>
</ul>
```

Il layout è quasi identico:

```
<!-- app/Resources/views/layout.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $view['slots']->output(
            'title',
            'Titolo predefinito'
        ) ?></title>
    </head>
    <body>
        <?php echo $view['slots']->output('_content') ?>
    </body>
</html>
```

**Note:** Lasciamo il template di show come esercizio, visto che dovrebbe essere banale crearlo basandosi sul template della lista.

Quando il motore di Symfony (chiamato `Kernel`) parte, ha bisogno di una mappa che gli consenta di sapere quali controllori eseguire, in base alle informazioni della richiesta. Una configurazione delle rotte fornisce tali informazioni in un formato leggibile:

```
# app/config/routing.yml
blog_list:
```

```
    path:     /blog
    defaults: { _controller: AppBundle:Blog:list }

blog_show:
    path:     /blog/show/{id}
    defaults: { _controller: AppBundle:Blog:show }
```

Ora che Symfony gestisce tutti i compiti più comuni, il front controller è semplicissimo. E siccome fa così poco, non si avrà mai bisogno di modificarlo una volta creato (e se si usa una distribuzione di Symfony, non servirà nemmeno crearlo!):

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->handle(Request::createFromGlobals())->send();
```

L'unico compito del front controller è inizializzare il motore di Symfony (il `Kernel`) e passargli un oggetto `Request` da gestire. Il nucleo di Symfony quindi usa la mappa delle rotte per determinare quale controllore richiamare. Proprio come prima, il metodo controllore è responsabile di restituire l'oggetto `Response` finale. Non resta molto altro da fare.

Per una rappresentazione visuale di come Symfony gestisca ogni richiesta, si veda il *diagramma di flusso della richiesta*.

### Dove consegna Symfony

Nei capitoli successivi, impareremo di più su come funziona ogni pezzo di Symfony e sull'organizzazione raccomandata di un progetto. Per ora, vediamo come migrare il blog da PHP puro a Symfony ci abbia migliorato la vita:

- L'applicazione ora ha un **codice organizzato chiaramente e coerentemente** (sebbene Symfony non obblighi a farlo). Questo promuove la **riusabilità** e consente a nuovi sviluppatori di essere produttivi nel progetto in modo più rapido.
- Il 100% del codice che si scrive è per la *propria* applicazione. **Non occorre sviluppare o mantenere utilità a basso livello**, come *autoload*, rotte o rendere i controllori.
- Symfony dà **accesso a strumenti open source**, come Doctrine e i componenti Templating, Security, Form, Validation e Translation (solo per nominarne alcuni).
- L'applicazione ora gode di **URL pienamente flessibili**, grazie al componente Routing.
- L'architettura HTTP-centrica di Symfony dà accesso a strumenti potenti, come la **cache HTTP** fornita dalla **cache HTTP interna di Symfony** o a strumenti ancora più potenti, come Varnish. Questi aspetti sono coperti in un capitolo successivo, tutto dedicato alla cache.

Ma forse la parte migliore nell'usare Symfony è l'accesso all'intero insieme di **strumenti open source di alta qualità sviluppati dalla comunità di Symfony**! Si possono trovare dei buoni bundle su KnpBundles.com.

## Template migliori

Se lo si vuole usare, Symfony ha un motore di template predefinito, chiamato Twig, che rende i template più veloci da scrivere e più facili da leggere. Questo vuol dire che l'applicazione di esempio può contenere ancora meno codice!

Prendiamo per esempio il template della lista, scritto in Twig:

```
{# app/Resources/views/blog/list.html.twig #}
{% extends "layout.html.twig" %}

{% block title %}Lista dei post{% endblock %}

{% block body %}
    <h1>Lista dei post</h1>
    <ul>
        {% for post in posts %}
        <li>
            <a href="{{ path('blog_show', {'id': post.id}) }}">
                {{ post.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

Il template corrispondente `layout.html.twig` è anche più facile da scrivere:

```
{# app/Resources/views/layout.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}Titolo predefinito{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Twig è ben supportato in Symfony. Pur essendo sempre supportati i template PHP, continueremo a discutere dei molti vantaggi offerti da Twig. Per ulteriori informazioni, vedere il capitolo dei template.

## Imparare di più con le ricette

- /cookbook/templating/PHP
- /cookbook/controller/service

CHAPTER 3

# Installare e configurare Symfony

Lo scopo di questo capitolo è mettere in grado di avere un'applicazione funzionante basata su Symfony. Per semplificare il processo di creazione di nuove applicazioni, Symfony fornisce un installatore.

## Installare l'installatore di Symfony

L'utilizzo dell'**installatore di Symfony** è l'unico modo raccomandato di creare nuove applicazioni Symfony. Questo installatore è un'applicazione PHP, che va installata solo una volta e che può quindi creare tutte le applicazioni Symfony.

**Note:** L'installatore richiede PHP 5.4 o successivi. Se si usa ancora la vecchia versione PHP 5.3, non si può usare l'installatore di Symfony. Leggere *Creare applicazioni Symfony senza l'installatore* per sapere come procedere.

A seconda del sistema operativo, l'installatore va installato in modi diversi.

### Sistemi Linux e Mac OS X

Aprire un terminale ed eseguire i seguenti tre comandi:

```
$ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
$ sudo chmod a+x /usr/local/bin/symfony
```

Questo creerà nel sistema un comando globale `symfony`.

### Sistemi Windows

Aprire la console dei comandi ed eseguire il seguente comando:

```
c:\> php -r "readfile('https://symfony.com/installer');" > symfony
```

Quindi, spostare il file `symfony.phar` nella cartella dei progetti ed eseguirlo, come segue:

```
c:\> move symfony c:\progetti
c:\progetti\> php symfony
```

## Creare l'applicazione Symfony

Una volta che l'installatore Symfony è pronto, creare la prima applicazione Symfony con il comando `new`:

```
# Linux, Mac OS X
$ symfony new progetto

# Windows
c:\> cd projects/
c:\projects\> php symfony.phar new progetto
```

Questo comando crea una nuova cartella, chiamata `progetto`, che contiene un nuovo progetto, basato sulla versione di Symfony più recente. Inoltre, l'installatore verifica se il sistema soddisfa i requisiti tecnici per eseguire applicazioni Symfony. In caso negativo, si vedrà una lista di modifiche necessarie a soddisfare tali requisiti.

**Tip:** Per ragioni di sicurezza, tutte le versioni di Symfony sono firmate digitalmente prima di essere distribuite. Se si vuole verificare l'integrità di una versione di Symfony, seguire i passi spiegati in questo post.

**Note:** Se l'installatore non funziona o non mostra nulla, assicurarsi che l'*estensione Phar* sia installata e abilitata.

### Basare un progetto su una specifica versione di Symfony

Se un progetto deve essere basato su una specifica versione di Symfony, passare il numero di versione come secondo parametro del comando `new`:

```
# usa la versione più recente di un ramo di Symfony
$ symfony new progetto 2.3
$ symfony new progetto 2.5
$ symfony new progetto 2.6

# usa una specifica versione di Symfony
$ symfony new progetto 2.3.26
$ symfony new progetto 2.6.5

# usa la versione LTS (Long Term Support) più recente
$ symfony new progetto lts
```

Se si vuole basare un progetto sull'ultima versione LTS di Symfony, passare `lts` come secondo parametro del comando `new`:

```
$ symfony new progetto lts
```

Leggere il processo di rilascio di Symfony per comprendere meglio il motivo per cui esistono varie versioni di Symfony e quale usare per i propri progetti.

## Creare applicazioni Symfony senza l'installatore

Se si usa ancora PHP 5.3 o se non si può eseguire l'installatore per altre ragioni, si possono creare applicazioni Symfony usando un metodo alternativo di installazione, basato su Composer.

Composer è un gestore di dipendenze, usato da applicazioni PHP moderne, e può essere usato per creare nuove applicazioni basate sul framework Symfony. Se non lo si ha già installato globalmente, seguire la prossima sezione.

### Installare Composer globalmente

Iniziare con installare Composer globalmente.

### Creare un'applicazione Symfony con Composer

Una volta installato Composer, eseguire il comando `create-project` per creare una nuova applicazione Symfony, basata sull'ultima versione stabile:

```
$ composer create-project symfony/framework-standard-edition progetto
```

Se si deve basare l'applicazione su una specifica versione di Symfony, fornire la versione come secondo parametro del comando `create-project`:

```
$ composer create-project symfony/framework-standard-edition progetto '2.3.*'
```

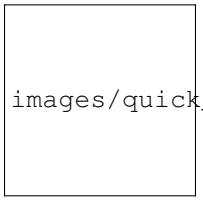
**Tip:** Con una connessione Internet lenta, si potrebbe pensare come Composer non stia facendo nulla. Nel caso, aggiungere l'opzione `-vvv` al comando precedente per mostrare un output dettagliato di tutto ciò che Composer sta facendo.

## Eseguire l'applicazione Symfony

Symfony sfrutta il server web interno fornito da PHP per eseguire applicazioni mentre le si sviluppa. Quindi, per eseguire un'applicazione Symfony basta andare nella cartella del progetto ed eseguire il seguente comando:

```
$ cd progetto/
$ php app/console server:run
```

Quindi, aprire un browser ed accedere all'URL `http://localhost:8000` per vedere la pagina di benvenuto di Symfony:



Al posto di questa pagina di benvenuto, si potrebbe vedere una pagina bianca o di errore. Questo dipende da un problema di configurazione dei permessi delle cartelle. Ci sono varie possibili soluzioni, a seconda del sistema operativo. Sono tutte spiegate nella sezione *Impostazione dei permessi*.

---

**Note:** Il server interno di PHP è disponibile in PHP 5.4 o successivi. Se si usa ancora la vecchia versione 5.3, occorrerà configurare un *host virtuale* nel proprio server web.

---

Il comando `server:run` è disponibile solo durante lo sviluppo di un'applicazione. Per eseguire applicazioni Symfony su server di produzione, si dovrà configurare un server web Apache o Nginx, come spiegato in /cookbook/configuration/web_server_configuration.

Dopo aver finito di lavorare su un'applicazione Symfony, si può fermare il server premendo *Ctrl+C* nel terminale:

## Verifica della configurazione di un'applicazione Symfony

Le applicazioni Symfony dispongono di un test per la configurazione del server, che mostra se l'ambiente è pronto per usare Symfony. Accedere al seguente URL per verificare la propria configurazione:

```
http://localhost:8000/config.php
```

Se ci sono problemi, correggerli prima di procedere.

**Impostare i permessi**

Un problema comune durante l'installazione è che le cartelle `app/cache` e `app/logs` devono essere scrivibili sia dal server web che dall'utente della linea di comando. Su sistemi UNIX, se l'utente del server web è diverso da quello della linea di comando, si possono provare le soluzioni seguenti.

**1. Usare lo stesso utente per CLI e server web**

In ambienti di sviluppo, è pratica comune usare lo stesso utente per CLI e server web, evitando così problemi di permessi per nuovi progetti. Lo si può fare modificando la configurazione del server web (cioè solitamente httpd.conf o apache2.conf per Apache) e impostandone l'utente in modo che sia lo stesso di CLI (p.e. per Apache, aggiornare i valori User e Group).

**2. Usare ACL su un sistema che supporta chmod +a**

Molti sistemi consento di usare il comando `chmod +a`. Provare prima questo e, in caso di errore, provare il metodo successivo. Viene usato un comando per cercare di determinare l'utente con cui gira il server web e impostarlo come `HTTPDUSER`:

```
$ rm -rf app/cache/*
$ rm -rf app/logs/*

$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx
↪' | grep -v root | head -1 | cut -d\  -f1`
$ sudo chmod +a "$HTTPDUSER allow delete,write,append,file_inherit,directory_inherit
↪" app/cache app/logs
$ sudo chmod +a "`whoami` allow delete,write,append,file_inherit,directory_inherit"␣
↪app/cache app/logs
```

**3. Usare ACL su un sistema che non supporta chmod +a**

Alcuni sistemi non supportano `chmod +a`, ma supportano un altro programma chiamato `setfacl`. Si potrebbe aver bisogno di abilitare il supporto ACL sulla propria partizione e installare setfacl prima di usarlo (come nel caso

di Ubuntu). Viene usato un comando per cercare di determinare l'utente con cui gira il server web e impostarlo come `HTTPDUSER`:

```
$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx
↪' | grep -v root | head -1 | cut -d\  -f1`
$ sudo setfacl -R -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX app/cache app/logs
$ sudo setfacl -dR -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX app/cache app/logs
```

Se non funziona, provare aggiungendo l'opzione `-n`.

**4. Senza usare ACL**

Se non è possibile modificare l'ACL delle cartelle, occorrerà modificare l'umask in modo che le cartelle cache e log siano scrivibili dal gruppo o da tutti (a seconda che gli utenti di server web e linea di comando siano o meno nello stesso gruppo). Per poterlo fare, inserire la riga seguente all'inizio dei file `app/console`, `web/app.php` e `web/app_dev.php`:

```
umask(0002); // Imposta i permessi a 0775

// oppure

umask(0000); // Imposta i permessi a 0777
```

Si noti che l'uso di ACL è raccomandato quando si ha accesso al server, perché la modifica di umask non è thread-safe.

## Aggiornare applicazioni Symfony

A questo punti, si dispone di un'applicazione Symfony pienamente funzionale, in cui si può sviluppare il proprio progetto. Un'applicazione Symfony dipende da varie librerie esterne. Queste sono scaricate nella cartella `vendor/` e sono gestite esclusivamente da Composer.

L'aggiornamento frequente di queste librerie di terze parti è una buona pratica, per prevenire bug e vulnerabilità di sicurezza. Eseguire il comando `update` di Composer per aggiornarle tutte insieme:

```
$ cd progetto/
$ composer update
```

A seconda della complessità del progetto, questo processo di aggiornamento può impiegare anche vari minuti per essere completato.

---

**Tip:** Symfony fornisce un comando per verificare se le dipendenze di un progetto contengano vulnerabilità note:

```
$ php app/console security:check
```

Una buona pratica di sicurezza consiste nell'eseguire regolarmente questo comando, per poter aggiornare o sostituire delle dipendenze compromesse, il prima possibile.

---

## Installare l'applicazione demo di Symfony

L'applicazione demo di Symfony è un'applicazione funzionate, che mostra il modo raccomandato di sviluppare applicazioni Symfony. L'applicazione è stata concepita come strumento di apprendimento per novizi di Symfony e il suo codice sorgente contiene vai commenti e note utili.

Per scaricare l'applicazione demo di Symfony, eseguire il comando `demo` dell'installatore di Symfony:

```
# Linux, Mac OS X
$ symfony demo

# Windows
c:\progetti\> php symfony demo
```

Una volta scaricata, entrare nella cartella `symfony_demo/` ed eseguire il server web interno di PHP, tramite il comando `php app/console server:run`. Accedere all'URL `http://localhost:8000` con un browser per iniziare a usare l'applicazione demo di Symfony.

## Installare una distribuzione di Symfony

Il progetto Symfony impacchetta "distribuzioni", che sono applicazioni pienamente funzionali, che includono le librerie del nucleo di Symfony, una selezione di bundle utili, una struttura di cartelle appropriata e alcune configurazioni predefinite. In effetti, quando è stata creata un'applicazione Symfony, nelle sezioni precedenti, in realtà è stata scaricata la distribuzione predefinita fornita da Symfony, chiamata *Symfony Standard Edition*.

*Symfony Standard Edition* è la distribuzione più popolare ed è anche la scelta migliore per sviluppatore che iniziano con Symfony. Tuttavia, la comunità di Symfony ha pubblicato altre distribuzioni, che si potrebbe voler usare in un'applicazione:

- Symfony CMF Standard Edition è una distribuzione pensata per partire con il progetto Symfony CMF, che rende più facile per gli sviluppatori l'aggiunta di funzionalità CMS ad applicazioni basate sul framework Symfony.
- Symfony REST Edition mostra come costruire un'applicazione che fornisca un'API REST, usando FOSRestBundle e vari altri bundle correlati.

## Uso di un controllo dei sorgenti

Se si usa un sistema di controllo di versione, come Git, si può tranquillamente eseguire il commit do tutto il codice del progetto. Questo perché le applicazioni Symfony contengono già un file `.gitignore`, preparato appositamente per Symfony.

Per istruzioni specifiche su come impostare al meglio un progetto per essere memorizzato in Git, vedere /cookbook/workflow/new_project_git.

### Usare un'applicazione Symfony versionata

Quando si usa Composer èer gestire le dipendenze di un'applicazione, si raccomanda di ignorare l'intera cartella `vendor/`, prima di eseguire commit di codice nel repository. Questo vuole dire che, quando si esegue il checkout di un'applicazione Symfony da un repository Git, non ci sarà la cartella `vendor/` e l'applicazione non funzionerà immediatamente.

Per farlo funzionare, eseguire il checkout dell'applicazione Symfony ed eseguire il comando `install` di Composer, per scaricare e installare tutte le dipendenze richieste dall'applicazione:

```
$ cd progetto/
$ composer install
```

Come fa Composer a sapere quali dipendenze installare? Perché quando si esegue il commit di un'applicazione Symfony su un repository, si includono i file `composer.json` e `composer.lock` nel commit. Questi file dicono a Composer quali dipendenze (e in quali specifiche versioni) installare nell'applicazione.

## Iniziare lo sviluppo

Ora che si dispone di un'applicazione Symfony pienamente funzionale, si può iniziare lo sviluppo! La distribuzione potrebbe contenere del codice di esempio, verificare sul file `README.md` incluso (aprirlo come file di testo) per conoscere l'eventuale codice di esempio incluso nella distribuzione.

Chi è nuovo su Symfony può fare riferimento a "*Creare pagine in Symfony*", dove si imparerà come creare pagine, cambiare configurazione e ogni altra cosa necessaria per la nuova applicazione.

Assicurarsi di dare un'occhiata anche al ricettario, che contiene una grande varietà di ricette, pensate per risolvere problemi specifici con Symfony.

CHAPTER 4

# Creare pagine in Symfony

La creazione di una nuova pagina in Symfony, che si tratti di una pagina HTML o di un JSON, è un semplice processo in due passi:

1. *Creare una rotta*: Una rotta definisce l'URL (p.e. `/about`) verso la pagina e specifica un controllore;
2. *Creare un controllore*: Un controllore è una funzione PHP, che costruisce la pagina. Si prende la richiesta in entrata e la si trasforma in un oggetto `Response` di Symfony, che contiene HTML, una stringa JSON o altro.

Proprio come sul web, ogni interazione sul web inizia con una richiesta HTTP. Il lavoro di un'applicazione è semplicemente quello di interpretare la richiesta e restituire l'appropriata risposta.

## Ambienti e front controller

Ogni applicazione Symfony gira in un ambiente. Un ambiente è un insieme specifico di configurazioni e bundle caricati, rappresentato da una stringa. La stessa applicazione può girare con diverse configurazioni, se eseguita in diversi ambienti. Symfony dispone di tre ambienti predefiniti: `dev`, `test` e `prod`. È comunque possibile crearne di altri.

Gli ambienti sono utili, perché consentono a una singola applicazione di avere un ambiente (dev) pensato per il debug e un altro (prod) ottimizziato per la velocità. Si possono anche caricare bundle specifici, in base all'ambiente. Per esempio, Symfony dispone di un WebProfilerBundle (descritto più avanti), abilitato solamente in `dev` e in `test`.

Symfony dispone di due front controller pubblici: `app_dev.php` fornisce l'ambiente `dev`, mentre `app.php` fornisce l'ambiente `prod`. Ogni accesso via web a Symfony normalmente passa per uno di questi due front controller. (L'ambiente `test` normalmente si usa solo quando si eseguono i test e quindi non dispone di un front controller dedicato. La linea di comando fornisce ugualmente un front controller utilizzabile con qualsiasi ambiente.)

Quando il front controller inizializza il kernel, fornisce due parametri: l'ambiente e la modalità di debug con cui il kernel deve girare. Per far rispondere velocemente l'applicazione, Symfony mantiene una cache sotto la cartella `app/cache/`. Quando il debug è abilitato (come in `app_dev.php`), la cache viene rinfrescata automaticamente a ogni modifica del codice o della configurazione. In debug, Symfony va più lentamente, ma le modifiche sono rispettate senza dover pulire a mano la cache.

### La pagina "numero casuale"

In questo capitolo, svilupperemo un'applicazione per generare numeri casuali. Quando avremo finito, l'utente sarà in grado di ottenere un numero casuale tra `1` e il limite superiore, impostato da URL:

```
http://localhost/app_dev.php/random/100
```

In realtà, si potrà sostituire `100` con qualsiasi altro numero, che funga da limite superiore per il numero da generare. Per creare la pagina, seguiamo il semplice processo in due passi.

---

**Note:** La guida presume che Symfony sia stato già scaricato e il server web configurato. L'URL precedente presume che `localhost` punti alla cartella `web` del nuovo progetto Symfony. Per informazioni dettagliate su questo processo, vedere la documentazione del server web usato. Ecco le pagine di documentazione per alcuni server web:

- Per il server Apache, fare riferimento alla documentazione su DirectoryIndex di Apache.
- Per Nginx, fare riferimento alla documentazione su HttpCoreModule di Nginx.

---

## Prima di iniziare: creare il bundle

Prima di iniziare, occorrerà creare un *bundle*. In Symfony, un bundle è come un plugin, tranne per il fatto che tutto il codice nella propria applicazione starà dentro a un bundle.

Un bundle non è nulla di più di una cartella che ospita ogni cosa correlata a una specifica caratteristica, incluse classi PHP, configurazioni e anche fogli di stile e file JavaScript (si veda *Il sistema dei bundle*).

A seconda della modalità di installazione di Symfony, si potrebbe già a vere un bundle, chiamato AcmeDemoBundle. Controllare nella cartella `src/` del progetto se c'è una cartella `DemoBundle/` sotto la cartella `Acme/`. Se tali cartelle esistono, saltare il resto di questa sezione e andare direttamente alla creazione della rotta.

Per creare un bundle chiamato `AcmeHelloBundle` (un bundle creato appositamente in questo capitolo), eseguire il seguente comando e seguire le istruzioni su schermo (usando tutte le opzioni predefinite):

```
$ php app/console generate:bundle --namespace=Acme/DemoBundle --format=yml
```

Dietro le quinte, viene creata una cartella per il bundle in `src/Acme/DemoBundle`. Inoltre viene aggiunta automaticamente una riga al file `app/AppKernel.php`, in modo che il bundle sia registrato nel kernel:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Acme\DemoBundle\AcmeDemoBundle(),
    );
    // ...

    return $bundles;
}
```

Ora che si è impostato il bundle, si può iniziare a costruire la propria applicazione, dentro il bundle stesso.

## Passo 1: creare la rotta

Per impostazione predefinita, il file di configurazione delle rotte in un'applicazione Symfony si trova in `app/config/routing.yml`. Come ogni configurazione in Symfony, si può anche scegliere di usare XML o PHP per configurare le rotte.

Se si guarda il file principale delle rotte, si vedrà che Symfony ha già aggiunto una voce, quando è stato generato AcmeDemoBundle:

Questa voce è molto basica: dice a Symfony di caricare la configurazione delle rotte dal file `Resources/config/routing.yml` (`routing.xml` o `routing.php` rispettivamente negli esempi di codice XML e PHP), che si trova dentro AcmeDemoBundle. Questo vuol dire che si mette la configurazione delle rotte direttamente in `app/config/routing.yml` o si organizzano le proprie rotte attraverso la propria applicazione e le si importano da qui.

---

**Note:** Non si è limitati a caricare configurazioni di rotte che condividono lo stesso formato. Per esempio, si potrebbe anche caricare un file YAML in una configurazione XML e viceversa.

---

Ora che il file `routing.yml` del bundle è stato importato, aggiungere la nuova rotta, che definisce l'URL della pagina che stiamo per creare:

Il routing consiste di due pezzi di base: il percorso (`path`), che è l'URL a cui la rotta corrisponderà, e un array `defaults`, che specifica il controllore che sarà eseguito. La sintassi dei segnaposto nello schema (`{limit}`) è un jolly. Vuol dire che `/random/10`, `/random/327` o ogni altro URL simile corrisponderanno a questa rotta. Il parametro del segnaposto `{limit}` sarà anche passato al controllore, in modo da poter usare il suo valore per salutare personalmente l'utente.

---

**Note:** Il sistema delle rotte ha molte altre importanti caratteristiche per creare strutture di URL flessibili e potenti nella propria applicazioni. Per maggiori dettagli, si veda il capitolo dedicato alle Rotte.

---

## Passo 2: creare il controllore

Quando un URL come `/hello/Ryan` viene gestita dall'applicazione, la rotta `hello` viene corrisposta e il controllore `AcmeDemoBundle:Hello:index` eseguito dal framework. Il secondo passo del processo di creazione della pagina è quello di creare tale controllore.

Il controllore ha il nome *logico* `AcmeDemoBundle:Random:index` ed è mappato sul metodo `indexAction` di una classe PHP chiamata `Acme\DemoBundle\Controller\RandomController`. Iniziamo creando questo file dentro il nostro AcmeDemoBundle:

```
// src/Acme/DemoBundle/Controller/RandomController.php
namespace Acme\DemoBundle\Controller;

class RandomController
{
}
```

In realtà, il controllore non è nulla di più di un metodo PHP, che va creato e che Symfony eseguirà. È qui che il codice usa l'informazione dalla richiesta per costruire e preparare la risorsa che è stata richiesta. Tranne per alcuni casi avanzati, il prodotto finale di un controllore è sempre lo stesso: un oggetto `Response` di Symfony.

Creare il metodo `indexAction`, che Symfony eseguirà quando la rotta `hello` sarà corrisposta:

```
// src/Acme/DemoBundle/Controller/RandomController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class RandomController
{
    public function indexAction($limit)
    {
        return new Response(
            '<html><body>Numero: '.rand(1, $limit).'</body></html>'
        );
    }
}
```

Il controllore è semplice: esso crea un nuovo oggetto `Response`, il cui primo parametro è il contenuto che sarà usato dalla risposta (in questo esempio, una piccola pagina HTML).

Congratulazioni! Dopo aver creato solo una rotta e un controllore, abbiamo già una pagina pienamente funzionante! Se si è impostato tutto correttamente, la propria applicazione dovrebbe salutare:

```
http://localhost/app_dev.php/random/10
```

**Tip:** Si può anche vedere l'applicazione nell'*ambiente* "prod", visitando:

```
http://localhost/app.php/random/10
```

Se si ottiene un errore, è probabilmente perché occorre pulire la cache, eseguendo:

```
$ php app/console cache:clear --env=prod --no-debug
```

Un terzo passo, facoltativo ma comune, del processo è quello di creare un template.

**Note:** I controllori sono il punto principale di ingresso del codice e un ingrediente chiave della creazione di pagine. Si possono trovare molte più informazioni nel capitolo sul controllore.

## Passo 3 (facoltativo): creare il template

I template consentono di spostare tutta la presentazione (p.e. il codice HTML) in un file separato e riusare diverse porzioni del layout della pagina. Invece di scrivere il codice HTML dentro al controllore, meglio rendere un template:

```
1  // src/Acme/DemoBundle/Controller/RandomController.php
2  namespace Acme\DemoBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6  class RandomController extends Controller
7  {
8      public function indexAction($limit)
9      {
10         $number = rand(1, $limit);
11
```

```
12          return $this->render(
13              'AcmeDemoBundle:Random:index.html.twig',
14              array('number' => $number)
15          );
16
17          // rende invece un template PHP
18          // return $this->render(
19          //     'AcmeDemoBundle:Random:index.html.php',
20          //     array('number' => $number)
21          // );
22      }
23  }
```

**Note:** Per poter usare il metodo **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::render'**, il controllore deve estendere la classe `Symfony\Bundle\FrameworkBundle\Controller\Controller`, che aggiunge scorciatoie per compiti comuni nei controllori. Ciò viene fatto nell'esempio precedente aggiungendo l'istruzione `use` alla riga 4 ed estendendo `Controller` alla riga 6.

Il metodo `render()` crea un oggetto `Response` riempito con il contenuto del template dato. Come ogni altro controllore, alla fine l'oggetto `Response` viene restituito.

Si noti che ci sono due diversi esempi su come rendere il template. Per impostazione predefinita, Symfony supporta due diversi linguaggi di template: i classici template PHP e i template, concisi ma potenti, Twig. Non ci si allarmi, si è liberi di scegliere tra i due, o anche tutti e due nello stesso progetto.

Il controllore rende il template `AcmeDemoBundle:Hello:index.html.twig`, che usa la seguente convenzioni dei nomi:

> **NomeBundle**:**NomeControllore**:**NomeTemplate**

Questo è il nome *logico* del template, che è mappato su una locazione fisica, usando la seguente convenzione:

> **/percorso/di/NomeBundle**/Resources/views/**NomeControllore/NomeTemplate**

In questo caso, `AcmeHelloBundle` è il nome del bundle, `Hello` è il controllore e `index.html.twig` il template:

Analizziamo il template Twig riga per riga:

- *riga 2*: Il token `extends` definisce un template padre. Il template definisce esplicitamente un file di layout, dentro il quale sarà inserito.
- *riga 4*: Il token `block` dice che ogni cosa al suo interno va posta dentro un blocco chiamato `body`. Come vedremo, è responsabilità del template padre (`base.html.twig`) rendere alla fine il blocco chiamato `body`.

Il template padre, `::base.html.twig`, manca delle porzioni **NomeBundle** e **NomeControllore** del suo nome (per questo ha il doppio duepunti (`::`) all'inizio). Questo vuol dire che il template risiede fuori dai bundle, nella cartella `app`:

Il template di base definisce il layout HTML e rende il blocco `body`, che era stato definito nel template `index.html.twig`. Rende anche un blocco `title`, che si può scegliere di definire nel template nel template `index.html.twig`. Poiché non è stato definito il blocco `title` nel template figlio, il suo valore predefinito è "Benvenuto!".

I template sono un modo potente per rendere e organizzare il contenuto della propria pagina. Un template può rendere qualsiasi cosa, dal codice HTML al CSS, o ogni altra cosa che il controllore abbia bisogno di restituire.

Nel ciclo di vita della gestione di una richiesta, il motore dei template è solo uno strumento opzionale. Si ricordi che lo scopo di ogni controllore è quello di restituire un oggetto `Response`. I template sono uno strumento potente, ma facoltativo, per creare il contenuto per un oggetto `Response`.

### Struttura delle cartelle

Dopo solo poche sezioni, si inizia già a capire la filosofia che sta dietro alla creazione e alla resa delle pagine in Symfony. Abbiamo anche già iniziato a vedere come i progetti Symfony siano strutturati e organizzati. Alla fine di questa sezione, sapremo dove cercare e inserire i vari tipi di file, e perché.

Sebbene interamente flessibili, per impostazione predefinita, ogni applicazione Symfony ha la stessa struttura di cartelle raccomandata:

**app/** Questa cartella contiene la configurazione dell'applicazione;

**src/** Tutto il codice PHP del progetto sta all'interno di questa cartella;

**vendor/** Ogni libreria dei venditori è inserita qui, per convenzione;

**web/** Questa è la cartella radice del web e contiene ogni file accessibile pubblicamente;

**See also:**

Si può facilmente ridefinire la struttura predefinita delle cartelle. Vedere /cookbook/configuration/override_dir_structure per maggiori informazioni.

## La cartella web

La cartella radice del web è la casa di tutti i file pubblici e statici, inclusi immagini, fogli di stile, file JavaScript. È anche il posto in cui stanno tutti i front controller:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Il file del front controller (`app.php` in questo esempio) è il file PHP che viene eseguito quando si usa un'applicazione Symfony e il suo compito è quello di usare una classe kernel, `AppKernel`, per inizializzare l'applicazione.

---

**Tip:** Aver un front controller vuol dire avere URL diverse e più flessibili rispetto a una tipica applicazione in puro PHP. Quando si usa un front controller, gli URL sono formattati nel modo seguente:

```
http://localhost/app.php/random/10
```

Il front controller, `app.php`, viene eseguito e l'URL "interno" `/random/10` è dirottato internamente, usando la configurazione delle rotte. Usando `mod_rewrite` di Apache, si può forzare l'esecuzione del file `app.php` senza bisogno di specificarlo nell'URL:

```
http://localhost/random/10
```

---

Sebbene i front controller siano essenziali nella gestione di ogni richiesta, raramente si avrà bisogno di modificarli o anche di pensarci. Saranno brevemente menzionati ancora nella sezione *Ambienti*.

## La cartella dell'applicazione (app)

Come visto nel front controller, la classe `AppKernel` è il punto di ingresso principale dell'applicazione ed è responsabile di tutta la configurazione. Per questo è memorizzata nella cartella `app/`.

Questa classe deve implementare due metodi, che definiscono tutto ciò di cui Symfony ha bisogno di sapere sulla propria applicazione. Non ci si deve preoccupare di questi metodi all'inizio, Symfony li riempie al posto nostro con delle impostazioni predefinite.

**`registerBundles()`** Restituisce un array di tutti bundle necessari per eseguire l'applicazione (vedere *Il sistema dei bundle*);

**`registerContainerConfiguration()`** Carica il file della configurazione principale dell'applicazione (vedere la sezione *Configurazione dell'applicazione*).

Nello sviluppo quotidiano, per lo più si userà la cartella `app/` per modificare i file di configurazione e delle rotte nella cartella `app/config/` (vedere *Configurazione dell'applicazione*). Essa contiene anche la cartella della cache dell'applicazione (`app/cache`), la cartella dei log (`app/logs`) e la cartella dei file risorsa a livello di applicazione, come i template (`app/Resources`). Ognuna di queste cartella sarà approfondita nei capitoli successivi.

**Autoload**

Quando Symfony si carica, un file speciale chiamato `app/autoload.php` viene incluso. Questo file è responsabile di configurare l'autoloader, che auto-caricherà i file dell'applicazione dalla cartella `src/` e le librerie di terze parti dalla cartella `vendor/` menzionate nel file `composer.json`.

Grazie all'autoloader, non si avrà mai bisogno di usare le istruzioni `include` o `require`. Al posto loro, Composer usa lo spazio dei nomi di una classe per determinare la sua posizione e includere automaticamente il file al posto nostro, nel momento in cui la classe è necessaria.

L'autoloader è già configurato per cercare nella cartella `src/` tutte le proprie classi PHP. Per poterlo far funzionare, il nome della classe e quello del file devono seguire lo stesso schema:

```
Nome della classe:
    Acme\DemoBundle\Controller\RandomController
Percorso:
    src/Acme/DemoBundle/Controller/RandomController.php
```

## La cartella dei sorgenti (src)

Detto semplicemente, la cartella `src/` contiene tutto il codice (codice PHP, template, file di configurazione, fogli di stile, ecc.) che guida la *propria* applicazione. Quando si sviluppa, gran parte del lavoro sarà svolto dentro uno o più bundle creati in questa cartella.

Ma cos'è esattamente un bundle?

### Configurazione dell'applicazione

Un'applicazione è composta da un insieme di bundle, che rappresentano tutte le caratteristiche e le capacità dell'applicazione stessa. Ogni bundle può essere personalizzato tramite file di configurazione, scritti in YAML, XML o PHP. Per impostazione predefinita, il file di configurazione principale risiede nella cartella `app/config/` è si chiama `config.yml`, `config.xml` o `config.php`, a seconda del formato scelto:

**Note:** Vedremo esattamente come caricare ogni formato di file nella prossima sezione, *Ambienti*.

Ogni voce di primo livello, come `framework` o `twig`, definisce la configurazione per un particolare bundle. Per esempio, la voce `framework` definisce la configurazione per il bundle del nucleo di Symfony FrameworkBundle e include configurazioni per rotte, template e altri sistemi fondamentali.

Per ora, non ci preoccupiamo delle opzioni di configurazione specifiche di ogni sezione. Il file di configurazione ha delle opzioni predefinite impostate. Leggendo ed esplorando ogni parte di Symfony, le opzioni di configurazione specifiche saranno man mano approfondite.

**Formati di configurazione**

Nei vari capitoli, tutti gli esempi di configurazione saranno mostrati in tutti e tre i formati (YAML, XML e PHP). Ciascuno ha i suoi vantaggi e svantaggi. La scelta è lasciata allo sviluppatore:

- *YAML*: Semplice, pulito e leggibile (se ne può sapere di più in "/components/yaml/yaml_format");
- *XML*: Più potente di YAML e supportato nell'autocompletamento dagli IDE;
- *PHP*: Molto potente, ma meno leggibile dei formati di configurazione standard.

## Esportazione della configurazione predefinita

Si può esportare la configurazione predefinita per un bundle in yaml sulla console, usando il comando `config:dump-reference`. Ecco un esempio di esportazione della configurazione predefinita di FrameworkBundle:

```
$ app/console config:dump-reference FrameworkBundle
```

Si può anche usare l'alias dell'estensione (voce di configurazione):

```
$ app/console config:dump-reference framework
```

**Note:** Vedere la ricetta /cookbook/bundles/extension per informazioni sull'aggiunta di configurazioni per un bundle.

### Ambienti

Un'applicazione può girare in vari ambienti. I diversi ambienti condividono lo stesso codice PHP (tranne per il front controller), ma usano differenti configurazioni. Per esempio, un ambiente `dev` salverà nei log gli avvertimenti e gli errori, mentre un ambiente `prod` solamente gli errori. Alcuni file sono ricostruiti a ogni richiesta nell'ambiente `dev` (per facilitare gli sviluppatori), ma salvati in cache nell'ambiente `prod`. Tutti gli ambienti stanno insieme nella stessa macchina e sono eseguiti nella stessa applicazione.

Un progetto Symfony generalmente inizia con tre ambienti (`dev`, `test` e `prod`), ma creare nuovi ambienti è facile. Si può vedere la propria applicazione in ambienti diversi, semplicemente cambiando il front controller nel browser. Per vedere l'applicazione in ambiente `dev`, accedere all'applicazione tramite il front controller di sviluppo:

```
http://localhost/app_dev.php/random/10
```

Se si preferisce vedere come l'applicazione si comporta in ambiente di produzione, richiamare invece il front controller `prod`:

```
http://localhost/app.php/random/10
```

Essendo l'ambiente `prod` ottimizzato per la velocità, la configurazione, le rotte e i template Twig sono compilato in classi in puro PHP e messi in cache. Per vedere delle modifiche in ambiente `prod`, occorrerà pulire tali file in cache e consentire che siano ricostruiti:

```
$ php app/console cache:clear --env=prod --no-debug
```

**Note:** Se si apre il file `web/app.php`, si troverà che è configurato esplicitamente per usare l'ambiente `prod`:

```
$kernel = new AppKernel('prod', false);
```

Si può creare un nuovo front controller per un nuovo ambiente, copiando questo file e cambiando `prod` con un altro valore.

**Note:** L'ambiente `test` è usato quando si eseguono i test automatici e non può essere acceduto direttamente tramite il browser. Vedere il capitolo sui test per maggiori dettagli.

## Configurazione degli ambienti

La classe `AppKernel` è responsabile del caricare effettivamente i file di configurazione scelti:

```
// app/AppKernel.php
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader->load(
        __DIR__.'/config/config_'.$this->getEnvironment().'.yml'
    );
}
```

Sappiamo già che l'estensione `.yml` può essere cambiata in `.xml` o `.php`, se si preferisce usare XML o PHP per scrivere la propria configurazione. Si noti anche che ogni ambiente carica i propri file di configurazione. Consideriamo il file di configurazione per l'ambiente `dev`.

La voce `imports` è simile all'istruzione `include` di PHP e garantisce che il file di configurazione principale (`config.yml`) sia caricato per primo. Il resto del file gestisce la configurazione per aumentare il livello di log, oltre ad altre impostazioni utili all'ambiente di sviluppo.

Sia l'ambiente `prod` che quello `test` seguono lo stesso modello: ogni ambiente importa il file di configurazione di base e quindi modifica i suoi file di configurazione per soddisfare le esigenze dello specifico ambiente. Questa è solo una convenzione, ma consente di riusare la maggior parte della propria configurazione e personalizzare solo le parti diverse tra gli ambienti.

### Riepilogo

Congratulazioni! Ora abbiamo visto ogni aspetto fondamentale di Symfony e scoperto quanto possa essere facile e flessibile. Pur essendoci ancora *moltissime* caratteristiche da scoprire, assicuriamoci di tenere a mente alcuni aspetti fondamentali:

- creare una pagine è un processo in tre passi, che coinvolge una **rotta**, un **controllore** e (opzionalmente) un **template**.
- ogni progetto contiene solo alcune cartelle principali: `web/` (risorse web e front controller), `app/` (configurazione), `src/` (i propri bundle) e `vendor/` (codice di terze parti) (c'è anche la cartella `bin/`, usata per aiutare nell'aggiornamento delle librerie dei venditori);
- ogni caratteristica in Symfony (incluso in nucleo del framework stesso) è organizzata in *bundle*, insiemi strutturati di file relativi a tale caratteristica;
- la **configurazione** per ciascun bundle risiede nella cartella `app/config` e può essere specificata in YAML, XML o PHP;
- la **configurazione dell'applicazione** globale si trova nella cartella `app/config`;
- ogni **ambiente** è accessibile tramite un diverso front controller (p.e. `app.php` e `app_dev.php`) e carica un diverso file di configurazione.

Da qui in poi, ogni capitolo introdurrà strumenti sempre più potenti e concetti sempre più avanzati. Più si imparerà su Symfony, più si apprezzerà la flessibilità della sua architettura e la potenza che dà nello sviluppo rapido di applicazioni.

CHAPTER 5

# Il controllore

Un controllore è una funzione PHP da creare, che prende le informazioni dalla richiesta HTTP e crea e restituisce una risposta HTTP (come oggetto `Response` di Symfony). La risposta potrebbe essere una pagina HTML, un documento XML, un array serializzato JSON, una immagine, un rinvio, un errore 404 o qualsiasi altra cosa possa venire in mente. Il controllore contiene una qualunque logica arbitraria di cui la *propria applicazione* necessita per rendere il contenuto di una pagina.

Per vedere quanto questo è semplice, diamo un'occhiata a un controllore di Symfony in azione. Il seguente controllore renderebbe una pagina che stampa semplicemente `Ciao mondo!`:

```
use Symfony\Component\HttpFoundation\Response;

public function helloAction()
{
    return new Response('Ciao mondo!');
}
```

L'obiettivo di un controllore è sempre lo stesso: creare e restituire un oggetto `Response`. Lungo il percorso, potrebbe leggere le informazioni dalla richiesta, caricare una risorsa da una base dati, inviare un'email, o impostare informazioni sulla sessione dell'utente. Ma in ogni caso, il controllore alla fine restituirà un oggetto `Response` che verrà restituito al client.

Non c'è nessuna magia e nessun altro requisito di cui preoccuparsi! Di seguito alcuni esempi comuni:

- Il *controllore A* prepara un oggetto `Response` che rappresenta il contenuto della homepage di un sito.
- Il *controllore B* legge il parametro `slug` da una richiesta per caricare un blog da una base dati e creare un oggetto `Response` che visualizza quel blog. Se lo `slug` non viene trovato nella base dati, crea e restituisce un oggetto `Response` con codice di stato 404.
- Il *controllore C* gestisce l'invio di un form contatti. Legge le informazioni del form dalla richiesta, salva le informazioni del contatto nella base dati e invia una email con le informazioni del contatto al webmaster. Infine, crea un oggetto `Response`, che rinvia il browser del client alla pagina di ringraziamento del form contatti.

## Richieste, controllori, ciclo di vita della risposta

Ogni richiesta gestita da un progetto Symfony passa attraverso lo stesso semplice ciclo di vita. Il framework si occupa dei compiti ripetitivi e infine esegue un controllore, che ospita il codice personalizzato dell'applicazione:

1. Ogni richiesta è gestita da un singolo file con il controllore principale (ad esempio `app.php` o `app_dev.php`) che inizializza l'applicazione;
2. Il `Router` legge le informazioni dalla richiesta (ad esempio l'URI), trova una rotta che corrisponde a tali informazioni e legge il parametro `_controller` dalla rotta;
3. Viene eseguito il controllore della rotta corrispondente e il codice all'interno del controllore crea e restituisce un oggetto `Response`;
4. Le intestazioni HTTP e il contenuto dell'oggetto `Response` vengono rispedite al client.

Creare una pagina è facile, basta creare un controllore (#3) e fare una rotta che mappa un URL su un controllore (#2).

---

**Note:** Anche se ha un nome simile, il "controllore principale" (front controller) è diverso dagli altri "controllori" di cui si parla in questo capitolo. Un controllore principale è un breve file PHP che è presente nella propria cartella web e sul quale sono dirette tutte le richieste. Una tipica applicazione avrà un front controller produzione (ad esempio `app.php`) e un frot controller per lo sviluppo (ad esempio `app_dev.php`). Probabilmente non si avrà mai bisogno di modificare, visualizzare o preoccuparsi dei front controller dell'applicazione.

---

## Un semplice controllore

Mentre un controllore può essere un qualsiasi callable PHP (una funzione, un metodo di un oggetto, o una `Closure`), in Symfony, un controllore di solito è un unico metodo all'interno di un oggetto controllore. I controllori sono anche chiamati *azioni*.

```
// src/AppBundle/Controller/HelloController.php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class HelloController
{
    public function indexAction($name)
    {
        return new Response('<html><body>Ciao '.$name.'!</body></html>');
    }
}
```

---

**Tip:** Si noti che il *controllore* è il metodo `indexAction`, che si trova all'interno di una *classe controllore* (`HelloController`). Non bisogna confondersi con i nomi: una *classe controllore* è semplicemente un modo comodo per raggruppare insieme vari controllori/azioni. Tipicamente, la classe controllore ospiterà diversi controllori/azioni (ad esempio `updateAction`, `deleteAction`, ecc).

---

Questo controllore è piuttosto semplice, ma vediamo di analizzarlo:

- *linea 3*: Symfony sfrutta la funzionalità degli spazi dei nomi di PHP 5.3 per utilizzarla nell'intera classe dei controllori. La parola chiave `use` importa la classe `Response`, che il controllore deve restituire.

- *linea 6*: Il nome della classe è la concatenazione di un nome per la classe controllore (ad esempio `Hello`) e la parola `Controller`. Questa è una convenzione che fornisce coerenza ai controllori e permette loro di essere referenziati solo dalla prima parte del nome (ad esempio `Hello`) nella configurazione delle rotte.
- *linea 8*: A ogni azione in una classe controllore viene aggiunto il suffisso `Action` mentre nella configurazione delle rotte viene utilizzato come riferimento il solo nome dell'azione (`index`). Nella sezione successiva, verrà creata una rotta che mappa un URI in questa azione. Si imparerà come i segnaposto delle rotte (`{name}`) diventano parametri del metodo dell'azione (`$name`).
- *linea 10*: Il controllore crea e restituisce un oggetto `Response`.

## Mappare un URL in un controllore

Il nuovo controllore restituisce una semplice pagina HTML. Per visualizzare questa pagina nel browser, è necessario creare una rotta che mappa uno specifico schema URL nel controllore:

Andando in `/hello/ryan` (p.e. `http://localhost:8000/app_dev.php/hello/ryan` se si usa il server web interno) Symfony esegue il controllore `HelloController::indexAction()` e passa `ryan` nella variabile `$name`. Creare una "pagina" significa semplicemente creare un metodo controllore e associargli una rotta.

Simple, right?

**La sintassi AppBundle:Hello:index del controllore**

Se si usano i formati YML o XML, si farà riferimento al controllore usando una speciale sintassi abbrevviata: `AppBundle:Hello:index`. Per maggiori dettagli sul formato del controllore, vedere *Schema per il nome dei controllori*.

**See also:**

Si può imparare molto di più sul sistema delle rotte leggendo il capitolo sulle rotte.

### I parametri delle rotte come parametri del controllore

Si è già appreso che la rotta punta a un metodo `HelloController::indexAction()`, che si trova all'interno di un bundle `AppBundle`. La cosa più interessante è il parametro passato a tale metodo:

```
// src/AppBundle/Controller/HelloController.php
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

/**
 * @Route("/hello/{name}", name="hello")
 */
public function indexAction($name)
{
    // ...
}
```

Il controllore ha un solo parametro, `$name`, che corrisponde al parametro `{name}` della rotta corrispondente (`ryan` se si va su `/hell/ryan`). Infatti, quando viene eseguito il controllore, Symfony abbina ogni parametro del controllore a un parametro della rotta. Quindi il valore di `{name}` viene passato a `$name`.

Vedere il seguente esempio:

Per questo il controllore può richiedere diversi parametri:

```
public function indexAction($firstName, $lastName)
{
    // ...
}
```

La mappatura dei parametri delle rotte nei parametri del controllore è semplice e flessibile. Tenere in mente le seguenti linee guida mentre si sviluppa.

- **L'ordine dei parametri del controllore non ha importanza**

  Symfony abbina i **nomi** dei parametri delle rotte e i **nomi** delle variabili dei metodi dei controllori. I parametri del controllore possono essere totalmente riordinati e continuare a funzionare perfettamente:

  ```
  public function indexAction($lastName, $firstName)
  {
      // ...
  }
  ```

- **Ogni parametro richiesto del controllore, deve corrispondere a uno dei parametri della rotta**

  Il codice seguente genererebbe un `RuntimeException`, perché non c'è nessun parametro `foo` definito nella rotta:

  ```
  public function indexAction($firstName, $lastName, $foo)
  {
      // ...
  }
  ```

  Rendere il parametro facoltativo metterebbe le cose a posto. Il seguente esempio non lancerebbe un'eccezione:

  ```
  public function indexAction($firstName, $lastName, $foo = 'bar')
  {
      // ...
  }
  ```

- **Non tutti i parametri delle rotte devono essere parametri del controllore**

  Se, per esempio, `last_name` non è importante per il controllore, si può ometterlo del tutto:

  ```
  public function indexAction($firstName)
  {
      // ...
  }
  ```

**Tip:** Ogni rotta ha anche un parametro speciale `_route`, che è equivalente al nome della rotta che è stata abbinata (ad esempio `hello`). Anche se di solito non è utile, questa è ugualmente disponibile come parametro del controllore. Si possono anche passare altre variabili alla rotta, dai parametri del controllore. Vedere /cookbook/routing/extra_information.

## La `Request` come parametro del controllore

Che fare se si ha bisogno di leggere i parametri della query string o un header o accedere a un file caricato? Tutte queste informazioni sono memorizzate nell'oggetto `Request` di Symfony. Per ottenerlo in un controllore, basta aggiungerlo come parametro e **forzare il tipo a Request**:

```
use Symfony\Component\HttpFoundation\Request;

public function indexAction($firstName, $lastName, Request $request)
{
    $page = $request->query->get('page', 1);

    // ...
}
```

**See also:**

Per saperne di più su come ottenere informazioni dalla richiesta, si veda accedere alla informazioni sulla richiesta.

## La classe base del controllore

Per comodità, Symfony ha una classe base `Controller`, che aiuta nelle attività più comuni del controllore e dà alla classe controllore l'accesso ai servizi, tramite il contenitore (vedere *Accesso ad altri servizi*).

Aggiungere la dichiarazione `use` sopra alla classe `Controller` e modificare `HelloController` per estenderla:

```
// src/AppBundle/Controller/HelloController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HelloController extends Controller
{
    // ...
}
```

Questo in realtà non cambia nulla su come lavora il controllore: dà solo accesso a dei metodi aiutanti, resi disponibili dalla classe base del controllore. Questi metodi sono solo scorciatoie per usare funzionalità del nucleo di Symfony, che sono a disposizione con o senza la classe base di `Controller`. Un ottimo modo per vedere le funzionalità del nucleo in azione è quello di guardare nella classe Controller.

**See also:**

È inoltre possibile definire i controllori come servizi. È opzionale, ma può dare maggiore controllo sulle esatte dipendenze e sugli oggetti iniettati dentro al controllore.

### Rinvio

Se si vuole rinviare l'utente a un'altra pagina, usare il metodo `redirectToRoute`:

```
public function indexAction()
{
    return $this->redirectToRoute('homepage');

    // redirectToRoute è equivalente all'uso combinato di redirect() E generateUrl():
    // return $this->redirect($this->generateUrl('homepage'), 301);
}
```

New in version 2.6: Il metodo `redirectToRoute()` è stato aggiunto in Symfony 2.6. In precedenza (e anche ora), si potevano usare `redirect()` e `generateUrl()` insieme (vedere esempio precedente).

Oppure, se si vuole rinviare all'esterno, basta usare `redirect()` e passare l'URL:

```
public function indexAction()
{
    return $this->redirect('http://symfony.com/doc');
}
```

Per impostazione predefinita, il metodo `redirect()` esegue un rinvio 302 (temporaneo). Per eseguire un rinvio 301 (permanente), modificare il secondo parametro:

```
public function indexAction()
{
    return $this->redirectToRoute('homepage', array(), 301);
}
```

**Tip:** Il metodo `redirect()` è semplicemente una scorciatoia che crea un oggetto `Response` specializzato nel rinviare l'utente. È equivalente a:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

public function indexAction()
{
    return new RedirectResponse($this->generateUrl('homepage'));
}
```

## Rendere i template

Se si serve dell'HTML, si vorrà rendere un template. Il metodo `render()` rende un template **e** ne inserisce il contenuto in un oggetto `Response`:

```
// rende app/Resources/views/hello/index.html.twig
return $this->render('hello/index.html.twig', array('name' => $name));
```

Si possono anche mettere template in sottocartelle. Meglio però evitare di creare strutture inutilmente profonde:

```
// rende app/Resources/views/hello/greetings/index.html.twig
return $this->render('hello/greetings/index.html.twig', array(
    'name' => $name
));
```

Il motore di template di Symfony è spiegato in gran dettaglio nel capitolo Template.

**Riferimenti a template che si trovano in un bundle**

Si possono anche mettere template nella cartella `Resources/views` di un bundle e farvi riferimento con la sintassi `NomeBundle:NomeCartella:NomeFile`. Per esempio, `AppBundle:Hello:index.html.twig` si riferisce a un template collocato in `src/AppBundle/Resources/views/Hello/index.html.twig`. Vedere *Riferimenti ai template in un bundle*.

### Accesso ad altri servizi

Symfony dispone di vari oggetti utili, chiamati servizi. Si possono usare per rendere template, inviare email, interrogare la base dati e per ogni altro "lavoro" immaginabile. Quando si installa un nuovo bundle, probabilmente si avranno a disposizione *ulteriori* servizi.

Quando si estende la classe base del controllore, è possibile accedere a qualsiasi servizio di Symfony attraverso il metodo `get()`. Di seguito si elencano alcuni servizi comuni che potrebbero essere utili:

```
$templating = $this->get('templating');

$router = $this->get('router');

$mailer = $this->get('mailer');
```

Ci sono innumerevoli altri servizi disponibili. Per elencarli tutti, utilizzare il comando di console `container:debug`:

```
$ php app/console debug:container
```

New in version 2.6: Prima di Symfony 2.6, questo comando si chiamava `container:debug`.

Per maggiori informazioni, vedere il capitolo /book/service_container.

## Gestire gli errori e le pagine 404

Quando qualcosa non si trova, si dovrebbe utilizzare bene il protocollo HTTP e restituire una risposta 404. Per fare questo, si lancia uno speciale tipo di eccezione. Se si sta estendendo la classe base del controllore, procedere come segue:

```
public function indexAction()
{
    // recuperare l'oggetto dalla base dati
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('Il prodotto non esiste');
    }

    return $this->render(...);
}
```

Il metodo `createNotFoundException()` crea uno speciale oggetto `Symfony\Component\HttpKernel\Exception\Not` che infine innesca una risposta HTTP 404 all'interno di Symfony.

Naturalmente si è liberi di lanciare qualunque classe `Exception` nel controllore: Symfony restituirà automaticamente un codice di risposta HTTP 500.

```
throw new \Exception('Qualcosa è andato storto!');
```

In ogni caso, all'utente finale viene mostrata una pagina di errore predefinita e allo sviluppatore viene mostrata una pagina di errore completa di debug (cioè usando `app_dev.php`, vedere *Ambienti e front controller*).

Entrambe le pagine di errore possono essere personalizzate. Per ulteriori informazioni, leggere nel ricettario "/cookbook/controller/error_pages".

# Gestione della sessione

Symfony fornisce un oggetto sessione che si può utilizzare per memorizzare le informazioni sull'utente (che sia una persona reale che utilizza un browser, un bot, o un servizio web) attraverso le richieste. Per impostazione predefinita, Symfony memorizza gli attributi in un cookie utilizzando le sessioni PHP native.

Memorizzare e recuperare informazioni dalla sessione può essere fatto da qualsiasi controllore:

```
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $session = $request->getSession();

    // memorizza un attributo per riutilizzarlo durante una successiva richiesta dell
↪'utente
    $session->set('pippo', 'pluto');

    // in un altro controllore per un'altra richiesta
    $pippo = $session->get('pippo');

    // usa un valore predefinito, se la chiave non esiste
    $filters = $session->get('filters', array());
}
```

Questi attributi rimarranno sull'utente per il resto della sessione utente.

## Messaggi flash

È anche possibile memorizzare messaggi di piccole dimensioni, all'interno della sessione dell'utente e solo per la richiesta successiva. Ciò è utile quando si elabora un form: si desidera rinviare e avere un messaggio speciale mostrato sulla richiesta *successiva*. I messaggi di questo tipo sono chiamati messaggi "flash".

Per esempio, immaginiamo che si stia elaborando un form inviato:

```
use Symfony\Component\HttpFoundation\Request;

public function updateAction(Request $request)
{
    $form = $this->createForm(...);

    $form->handleRequest($request);

    if ($form->isValid()) {
        // fare una qualche elaborazione

        $this->get('session')->getFlashBag()->add(
            'notice',
            'Le modifiche sono state salvate!'
        );

        // $this->addFlash è equivalente a $this->get('session')->getFlashBag()->add

        return $this->redirectToRoute(...);
    }
```

```
    return $this->render(...);
}
```

Dopo l'elaborazione della richiesta, il controllore imposta un messaggio flash `notice` e poi rinvia. Il nome (`notice`) non è significativo, è solo quello che si utilizza per identificare il tipo del messaggio.

Nel template dell'azione successiva, il seguente codice può essere utilizzato per rendere il messaggio `notice`:

Per come sono stati progettati, i messaggi flash sono destinati a vivere esattamente per una richiesta (hanno la "durata di un flash"). Sono progettati per essere utilizzati con un rinvio, esattamente come è stato fatto in questo esempio.

## L'oggetto Response

L'unico requisito per un controllore è restituire un oggetto `Response`. La classe `Symfony\Component\HttpFoundation\Response` è una astrazione PHP sulla risposta HTTP, il messaggio testuale che contiene gli header HTTP e il contenuto che viene inviato al client:

```
use Symfony\Component\HttpFoundation\Response;

// crea una semplice risposta JSON con un codice di stato 200 (predefinito)
$response = new Response('Ciao '.$name, 200);

// crea una risposta JSON con un codice di stato 200
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

La proprietà `headers` è un oggetto `Symfony\Component\HttpFoundation\HeaderBag` con alcuni utili metodi per leggere e modificare gli header `Response`. I nomi degli header sono normalizzati in modo che l'utilizzo di `Content-Type` sia equivalente a `content-type` o anche a `content_type`.

Ci sono anche alcune classi speciali, che facilitano alcuni tipi di risposta:

- Per JSON, `Symfony\Component\HttpFoundation\JsonResponse`. Vedere component-http-foundation-json-response.
- Per i file, `Symfony\Component\HttpFoundation\BinaryFileResponse`. Vedere component-http-foundation-serving-files.
- Per le risposte in flussi, `Symfony\Component\HttpFoundation\StreamedResponse`. Per streaming-response.

**See also:**

Niente paura! Ci sono molte altre informazioni nell'oggetto `Response` nella documentazione sui componenti. Vedere component-http-foundation-response.

## L'oggetto Request

Oltre ai valori dei segnaposto delle rotte, il controllore ha anche accesso all'oggetto `Request`. Il framework inietta l'oggetto `Request` nel controllore, se una variabile è forzata a *SymfonyComponentHttpFoundationRequest*:

```
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
```

```
    $request->isXmlHttpRequest(); // è una richiesta Ajax?

    $request->getPreferredLanguage(array('en', 'fr'));

    $request->query->get('page'); // recupera un parametro $_GET

    $request->request->get('page'); // recupera un parametro $_POST
}
```

Come l'oggetto `Response`, le intestazioni della richiesta sono memorizzate in un oggetto `HeaderBag` e sono facilmente accessibili.

**See also:**

Niente paura! Ci sono molte altre informazioni nell'oggetto `Request` nella documentazione sui componenti. Vedere component-http-foundation-response.

## Creare pagine statiche

Si può creare una pagina statica, senza nemmeno creare un controllore (basta una rotta e un template).

Vedere /cookbook/templating/render_without_controller.

### Inoltro a un altro controllore

Si può anche facilmente inoltrare internamente a un altro controllore con il metodo **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::forward'**. Invece di redirigere il browser dell'utente, fa una sotto richiesta interna e chiama il controllore specificato. Il metodo `forward()` restituisce l'oggetto `Response` che è tornato da quel controllore:

```
public function indexAction($name)
{
    $response = $this->forward('AppBundle:Something:fancy', array(
        'name'  => $name,
        'color' => 'green',
    ));

    // ... modificare ulteriormente la risposta o restituirla direttamente

    return $response;
}
```

Si noti che il metodo `forward()` utilizza la stessa rappresentazione stringa del controllore (vedere *Schema per il nome dei controllori*). In questo caso, l'obiettivo della classe del controllore sarà `SomethingController::fancyAction()` in `AppBundle`. L'array passato al metodo diventa un insieme di parametri sul controllore risultante. La stessa interfaccia viene utilizzata quando si incorporano controllori nei template (vedere *Inserire controllori*). L'obiettivo del metodo controllore dovrebbe essere simile al seguente:

```
public function fancyAction($name, $color)
{
    // ... creare e restituire un oggetto Response
}
```

E proprio come quando si crea un controllore per una rotta, l'ordine dei parametri di `fancyAction` non è importante. Symfony controlla i nomi degli indici chiave (ad esempio `name`) con i nomi dei parametri del metodo (ad esempio `$name`). Se si modifica l'ordine dei parametri, Symfony continuerà a passare il corretto valore di ogni variabile.

## Considerazioni finali

Ogni volta che si crea una pagina, è necessario scrivere del codice che contiene la logica per quella pagina. In Symfony, questo codice si chiama controllore, ed è una funzione PHP che può fare qualsiasi cosa occorra per restituire l'oggetto finale `Response`, che verrà restituito all'utente.

Per rendere la vita più facile, si può scegliere di estendere una classe base `Controller`, che contiene metodi scorciatoia per molti compiti comuni del controllore. Per esempio, dal momento che non si vuole mettere il codice HTML nel controllore, è possibile utilizzare il metodo `render()` per rendere e restituire il contenuto da un template.

In altri capitoli, si vedrà come il controllore può essere usato per persistere e recuperare oggetti da una base dati, processare i form inviati, gestire la cache e altro ancora.

## Imparare di più dal ricettario

- /cookbook/controller/error_pages
- /cookbook/controller/service

CHAPTER 6

# Le rotte

URL ben realizzati sono una cosa assolutamente da avere per qualsiasi applicazione web seria. Questo significa lasciarsi alle spalle URL del tipo `index.php?article_id=57` in favore di qualcosa come `/read/intro-to-symfony`.

Avere flessibilità è ancora più importante. Che cosa succede se è necessario modificare l'URL di una pagina da `/blog` a `/news`? Quanti collegamenti bisogna cercare e aggiornare per realizzare la modifica? Se si stanno utilizzando le rotte di Symfony la modifica è semplice.

Le rotte di Symfony consentono di definire URL creativi che possono essere mappati in differenti aree dell'applicazione. Entro la fine del capitolo, si sarà in grado di:

- Creare rotte complesse che mappano i controllori
- Generare URL all'interno di template e controllori
- Caricare le risorse delle rotte dai bundle (o da altre parti)
- Eseguire il debug delle rotte

## Le rotte in azione

Una *rotta* è una mappatura tra uno schema di URL e un controllore. Per esempio, supponiamo che si voglia gestire un qualsiasi URL tipo `/blog/my-post` o `/blog/all-about-symfony` e inviarlo a un controllore che cerchi e visualizzi quel post del blog. La rotta è semplice:

Lo schema definito dalla rotta `blog_show` si comporta come `/blog/*`, dove al carattere jolly viene dato il nome `slug`. Per l'URL `/blog/my-blog-post`, la variabile `slug` ottiene il valore `my-blog-post`, che è disponibile per l'utilizzo nel controllore (proseguire nella lettura). `blog_show` è il nome interno della rotta, che non ha ancora senso e che necessita solamente di essere unico. Sarà usato più avanti per generare URL.

Se non si vogliono usare le annotazioni, per questioni di preferenza o per non dipendere da SensioFrameworkExtraBundle, si possono anche usare Yaml, XML o PHP. In questi formati, il parametro `_controller` è una chiave speciale, che dice a Symfony quale controllore eseguire quando un URL corrisponde alla rotta. La stringa `_controller`

è chiamata *nome logico*. Segue uno schema che punta a specifici classe e metodo PHP, in questo caso al metodo `AppBundle\Controller\BlogController::showAction`.

Congratulazioni! Si è appena creata la prima rotta, collegandola ad un controllore. Ora, quando si visita `/blog/my-post`, verrà eseguito il controllore `showAction` e la variabile `$slug` avrà valore `my-post`.

Questo è l'obiettivo delle rotte di Symfony: mappare l'URL di una richiesta in un controllore. Lungo la strada, si impareranno tutti i trucchi per mappare facilmente anche gli URL più complessi.

## Le rotte: funzionamento interno

Quando all'applicazione viene fatta una richiesta, questa contiene un indirizzo alla esatta "risorsa" che il client sta richiedendo. Questo indirizzo è chiamato URL, (o URI) e potrebbe essere `/contact`, `/blog/read-me`, o qualunque altra cosa. Prendere ad esempio la seguente richiesta HTTP:

```
GET /blog/my-blog-post
```

L'obiettivo del sistema delle rotte di Symfony è quello di analizzare questo URL e determinare quale controller dovrebbe essere eseguito. L'intero processo è il seguente:

1. La richiesta è gestita dal front controller di Symfony (ad esempio `app.php`);
2. Il nucleo di Symfony (ad es. il kernel) chiede al router di ispezionare la richiesta;
3. Il router verifica la corrispondenza dell'URL in arrivo con una specifica rotta e restituisce informazioni sulla rotta, tra le quali il controllore che deve essere eseguito;
4. Il kernel di Symfony esegue il controllore, che alla fine restituisce un oggetto `Response`.

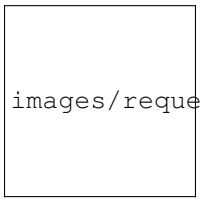



Fig. 6.1: Lo strato delle rotte è uno strumento che traduce l'URL in ingresso in uno specifico controllore da eseguire.

## Creazione delle rotte

Symfony carica tutte le rotte per l'applicazione da un singolo file con la configurazione delle rotte. Il file generalmente è `app/config/routing.yml`, ma può essere configurato per essere qualunque cosa (compreso un file XML o PHP) tramite il file di configurazione dell'applicazione:

**Tip:** Anche se tutte le rotte sono caricate da un singolo file, è una pratica comune includere ulteriori risorse di rotte all'interno del file. Per farlo, basta indicare nel file di routing principale quale file esterni debbano essere inclusi. Vedere la sezione *Includere risorse esterne per le rotte* per maggiori informazioni.

### Configurazione di base delle rotte

Definire una rotta è semplice e una tipica applicazione avrà molte rotte. Una rotta di base è costituita da due parti: il `pattern` da confrontare e un array `defaults`:

Questa rotta corrisponde alla homepage (`/`) e la mappa nel controllore `AppBundle:Main:homepage`. La stringa `_controller` è tradotta da Symfony in una funzione PHP effettiva, ed eseguita. Questo processo verrà spiegato a breve nella sezione *Schema per il nome dei controllori*.

## Rotte con segnaposti

Naturalmente il sistema delle rotte supporta rotte molto più interessanti. Molte rotte conterranno uno o più segnaposto "jolly":

Lo schema verrà soddisfatto da qualsiasi cosa del tipo `/blog/*`. Meglio ancora, il valore corrispondente il segnaposto `{slug}` sarà disponibile all'interno del controllore. In altre parole, se l'URL è `/blog/hello-world`, una variabile `$slug`, con un valore `hello-world`, sarà disponibile nel controllore. Questo può essere usato, ad esempio, per caricare il post sul blog che verifica questa stringa.

Tuttavia lo schema *non* deve corrispondere semplicemente a `/blog`. Questo perché, per impostazione predefinita, tutti i segnaposto sono obbligatori. Questo comportamento può essere cambiato aggiungendo un valore segnaposto all'array `defaults`.

## Segnaposto obbligatori e opzionali

Per rendere le cose più eccitanti, aggiungere una nuova rotta che visualizza un elenco di tutti i post disponibili del blog per questa applicazione immaginaria di blog:

Finora, questa rotta è la più semplice possibile: non contiene segnaposto e corrisponde solo all'esatto URL `/blog`. Ma cosa succede se si ha bisogno di questa rotta per supportare l'impaginazione, dove `/blog/2` visualizza la seconda pagina dell'elenco post del blog? Bisogna aggiornare la rotta per avere un nuovo segnaposto `{page}`:

Come il precedente segnaposto `{slug}`, il valore che verifica `{page}` sarà disponibile all'interno del controllore. Il suo valore può essere usato per determinare quale insieme di post del blog devono essere visualizzati per una data pagina.

Un attimo però! Dal momento che i segnaposto per impostazione predefinita sono obbligatori, questa rotta non avrà più corrispondenza con il semplice `/blog`. Invece, per vedere la pagina 1 del blog, si avrà bisogno di utilizzare l'URL `/blog/1`! Dal momento che non c'è soluzione per una complessa applicazione web, modificare la rotta per rendere il parametro `{page}` opzionale. Questo si fa includendolo nella collezione `defaults`:

Aggiungendo `page` alla chiave `defaults`, il segnaposto `{page}` non è più obbligatorio. L'URL `/blog` corrisponderà a questa rotta e il valore del parametro `page` verrà impostato a `1`. Anche l'URL `/blog/2` avrà corrispondenza, dando al parametro `page` il valore `2`. Perfetto.

| URL | Rotta | Parametri |
|---|---|---|
| `/blog` | `blog` | `{page}` = `1` |
| `/blog/1` | `blog` | `{page}` = `1` |
| `/blog/2` | `blog` | `{page}` = `2` |

**Caution:** Si possono ovviamente avere più segnaposto opzionali (p.e. `/blog/{slug}/{page}`), ma ogni cosa dopo un segnaposto opzionale deve essere opzionale a sua volta. Per esempio, `/{page}/blog` è un percorso valido, ma `page` sarà sempre obbligatorio (cioè richiamando solo `/blog` la rotta non corrisponderà).

**Tip:** Le rotte con parametri facoltativi alla fine non avranno corrispondenza da richieste con barra finale (p.e. `/blog/` non corrisponderà, `/blog` invece sì).

## Aggiungere requisiti

Si dia uno sguardo veloce alle rotte che sono state create finora:

Si riesce a individuare il problema? Notare che entrambe le rotte hanno schemi che verificano URL del tipo `/blog/*`. Il router di Symfony sceglie sempre la **prima** rotta corrispondente che trova. In altre parole, la rotta `blog_show` non sarà *mai* trovata. Invece, un URL del tipo `/blog/my-blog-post` verrà abbinato alla prima rotta (`blog`) restituendo il valore senza senso `my-blog-post` per il parametro `{page}`.

| URL | Rotta | Parametri |
|---|---|---|
| `/blog/2` | `blog` | `{page}` = `2` |
| `/blog/my-blog-post` | `blog` | `{page}` = `"my-blog-post"` |

La risposta al problema è aggiungere *requisiti* o *condizioni (see :ref:'book-routing-conditions'). alle rotte. Le rotte in questo esempio potrebbero funzionare perfettamente se lo schema ''/blog/{page}'' fosse verificato *solo* per gli URL dove `{page}` fosse un numero intero. Fortunatamente, i requisiti possono essere scritti tramite espressioni regolari e aggiunti per ogni parametro. Per esempio:

Il requisito `\d+` è una espressione regolare che dice che il valore del parametro `{page}` deve essere una cifra (cioè un numero). La rotta `blog` sarà comunque abbinata a un URL del tipo `/blog/2` (perché 2 è un numero), ma non sarà più abbinata a un URL tipo `/blog/my-blog-post` (perché `my-blog-post` *non* è un numero).

Come risultato, un URL tipo `/blog/my-blog-post` ora verrà correttamente abbinato alla rotta `blog_show`.

| URL | Rotta | Parametri |
|---|---|---|
| `/blog/2` | `blog` | `{page}` = `2` |
| `/blog/my-blog-post` | `blog_show` | `{slug}` = `my-blog-post` |
| `/blog/2-my-blog-post` | `blog_show` | `{slug}` = `2-my-blog-post` |

**Vincono sempre le rotte che compaiono prima**

Il significato di tutto questo è che l'ordine delle rotte è molto importante. Se la rotta `blog_show` fosse stata collocata sopra la rotta `blog`, l'URL `/blog/2` sarebbe stato abbinato a `blog_show` invece di `blog` perché il parametro `{slug}` di `blog_show` non ha requisiti. Utilizzando l'ordinamento appropriato e dei requisiti intelligenti, si può realizzare qualsiasi cosa.

Poiché i requisiti dei parametri sono espressioni regolari, la complessità e la flessibilità di ogni requisito dipende da come li si scrive. Si supponga che la pagina iniziale dell'applicazione sia disponibile in due diverse lingue, in base all'URL:

Per le richieste in entrata, la porzione `{locale}` dell'URL viene controllata tramite l'espressione regolare `(en|fr)`.

| Percorso | Parametri |
|---|---|
| `/` | `{_locale}` = `"en"` |
| `/en` | `{_locale}` = `"en"` |
| `/fr` | `{_locale}` = `"fr"` |
| `/es` | *non corrisponde alla rotta* |

## Aggiungere requisiti al metodo HTTP

In aggiunta agli URL, si può anche verificare il *metodo* della richiesta entrante (ad esempio GET, HEAD, POST, PUT, DELETE). Si supponga di avere un form contatti con due controllori: uno per visualizzare il form (su una richiesta GET) e uno per l'elaborazione del form dopo che è stato inviato (su una richiesta POST). Questo può essere realizzato con la seguente configurazione per le rotte:

Nonostante il fatto che queste due rotte abbiano schemi identici (`/contact`), la prima rotta corrisponderà solo a richieste GET e la seconda rotta corrisponderà solo a richieste POST. Questo significa che è possibile visualizzare il form e inviarlo utilizzando lo stesso URL ma controllori distinti per le due azioni.

**Note:** Se non viene specificato alcune metodo, la rotta verrà abbinata a *tutti* i metodi.

## Aggiungere un host

Si può anche far corrispondere un *host* HTTP della richiesta in arrivo. Per maggiori informazioni, vedere /components/routing/hostname_pattern nella documentazione del componente Routing.

## Corrispondenza di rotte tramite condizioni

New in version 2.4: Le condizioni sulle rotte sono state aggiunte in Symfony 2.4.

Come visto, una rotta può essere fatta per corrispondere solo ad alcuni caratteri jolly (tramite espressioni regolari), metodi HTTP o nomi di host. Ma il sistema delle rotte può essere esteso per una flessibilità pressoché infiinta, usando le condizioni:

La voce `condition` è un'espressione, la cui sintassi si può approfondire in /components/expression_language/syntax. Grazie a essa, la rotta non corrisponderà a meno che il metodo HTTP non sia GET o HEAD *e* se l'header `User-Agent` sarà `firefox`.

Si può usare qualsiasi logica complessa necessaria nell'espressione, sfruttando due variabili passate all'espressione stessa:

**`context`** Un'istanza di `Symfony\Component\Routing\RequestContext`, che contiene informazioni essenziali sulla rotta corrisposta;

**`request`** L'oggetto `Symfony\Component\HttpFoundation\Request` di Symfony (vedere component-http-foundation-request).

**Caution:** Le condizioni *non* sono considerate durante la generazione di un URL.

**Le espressioni sono compilate in PHP**

Dietro le quinte, le espressioni sono compilate in PHP puro. L'esempio precedente genererà il seguente codice PHP nella cartella della cache:

```
if (rtrim($pathinfo, '/contact') === '' && (
    in_array($context->getMethod(), array(0 => "GET", 1 => "HEAD"))
    && preg_match("/firefox/i", $request->headers->get("User-Agent"))
)) {
    // ...
}
```

Per questo motivo, l'uso di `condition` non causerà un sovraccarico, a parte il tempo necessario all'esecuzione del codice PHP.

### Esempio di rotte avanzate

A questo punto, si ha tutto il necessario per creare una complessa struttura di rotte in Symfony. Quello che segue è un esempio di quanto flessibile può essere il sistema delle rotte:

Come si sarà visto, questa rotta verrà soddisfatta solo quando la porzione `{culture}` dell'URL è `en` o `fr` e se `{year}` è un numero. Questa rotta mostra anche come sia possibile utilizzare un punto tra i segnaposto al posto di una barra. Gli URL corrispondenti a questa rotta potrebbero essere del tipo:

- `/articles/en/2010/my-post`
- `/articles/fr/2010/my-post.rss`
- `/articles/en/2013/my-latest-post.html`

**Il parametro speciale `_format` per le rotte**

Questo esempio mette in evidenza lo speciale parametro per le rotte `_format`. Quando si utilizza questo parametro, il valore cercato diventa il "formato della richiesta" dell'oggetto `Request`. In definitiva, il formato della richiesta è usato per cose tipo impostare il `Content-Type` della risposta (per esempio una richiesta di formato `json` si traduce in un `Content-Type` con valore `application/json`). Può essere utilizzato anche nel controllore per rendere un template diverso per ciascun valore di `_format`. Il parametro `_format` è un modo molto potente per rendere lo stesso contenuto in formati diversi.

**Note:** A volte si desidera che alcune parti delle rotte siano configurabili in modo globale. Symfony fornisce un modo per poterlo fare, sfruttando i parametri del contenitore di servizi. Si può approfondire in "/cookbook/routing/service_container_parameters".

### Parametri speciali per le rotte

Come si è visto, ogni parametro della rotta o valore predefinito è disponibile come parametro nel metodo del controllore. Inoltre, ci sono tre parametri speciali: ciascuno aggiunge una funzionalità all'interno dell'applicazione:

**`_controller`** Come si è visto, questo parametro viene utilizzato per determinare quale controllore viene eseguito quando viene trovata la rotta;

**`_format`** Utilizzato per impostare il formato della richiesta (*per saperne di più*);

**`_locale`** Utilizzato per impostare il locale sulla richiesta (*per saperne di più*).

## Schema per il nome dei controllori

Ogni rotta deve avere un parametro `_controller`, che determina quale controllore dovrebbe essere eseguito quando si accoppia la rotta. Questo parametro utilizza un semplice schema stringa, chiamato *nome logico del controllore*, che Symfony mappa in uno specifico metodo PHP di una certa classe. Lo schema ha tre parti, ciascuna separata da due punti:

**bundle**:**controllore**:**azione**

Per esempio, se `_controller` ha valore `AcmeBlogBundle:Blog:show` significa:

| Bundle | Classe controllore | Nome metodo |
|---|---|---|
| AppBundle | `BlogController` | `showAction` |

Il controllore potrebbe essere simile a questo:

```
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        // ...
    }
}
```

Si noti che Symfony aggiunge la stringa `Controller` al nome della classe (`Blog` => `BlogController`) e `Action` al nome del metodo (`show` => `showAction`).

Si potrebbe anche fare riferimento a questo controllore con il nome completo di classe e metodo: `Acme\BlogBundle\Controller\BlogController::showAction`. Ma seguendo alcune semplici convenzioni, il nome logico è più conciso e permette una maggiore flessibilità.

---

**Note:** Oltre all'utilizzo del nome logico o il nome completo della classe, Symfony supporta un terzo modo per fare riferimento a un controllore. Questo metodo utilizza solo un separatore due punti (ad esempio `nome_servizio:indexAction`) e fa riferimento al controllore come un servizio (vedere /cookbook/controller/service).

---

## Parametri delle rotte e parametri del controllore

I parametri delle rotte (ad esempio `{slug}`) sono particolarmente importanti perché ciascuno è reso disponibile come parametro al metodo del controllore:

```
public function showAction($slug)
{
  // ...
}
```

In realtà, l'intera collezione `defaults` viene unita con i valori del parametro per formare un singolo array. Ogni chiave di questo array è disponibile come parametro sul controllore.

In altre parole, per ogni parametro del metodo del controllore, Symfony cerca per un parametro della rotta con quel nome e assegna il suo valore a tale parametro. Nell'esempio avanzato di cui sopra, qualsiasi combinazioni (in qualsiasi ordine) delle seguenti variabili potrebbe essere usati come parametri per il metodo `showAction()`:

- `$_locale`
- `$year`
- `$title`
- `$_format`
- `$_controller`
- `$_route`

Dal momento che il segnaposto e la collezione `defaults` vengono uniti insieme, è disponibile anche la variabile `$_controller`. Per una trattazione più dettagliata, vedere *I parametri delle rotte come parametri del controllore*.

---

**Tip:** È inoltre possibile utilizzare una variabile speciale `$_route`, che è impostata sul nome della rotta che è stata abbinata.

---

Si possono anche aggiungere ulteriori informazioni alla definizione di una rotta e accedervi da un controllore. Per maggiori informazioni su questo argomento, vedere /cookbook/routing/extra_information.

## Includere risorse esterne per le rotte

Tutte le rotte vengono caricate attraverso un singolo file di configurazione, generalmente `app/config/routing.yml` (vedere *Creazione delle rotte* sopra). In genere, però, si desidera caricare le rotte da altri posti, come un file di rotte presente all'interno di un bundle. Questo può essere fatto "importando" il file:

---

**Note:** Quando si importano le risorse in formato YAML, la chiave (ad esempio `acme_hello`) non ha un significato particolare. Basta essere sicuri che sia unica, in modo che nessun'altra linea la sovrascriva.

---

La chiave `resource` carica la data risorsa di rotte. In questo esempio la risorsa è il percorso completo di un file, dove la sintassi scorciatoia `@AcmeHelloBundle` viene risolta con il percorso del bundle. Il file importato potrebbe essere come questo:

---

**Note:** Si possono anche includere altri file di configurazione, opzione spesso usata per importare rotte da bundle di terze parti:

---

### Prefissare le rotte importate

Si può anche scegliere di fornire un "prefisso" per le rotte importate. Per esempio, si supponga di volere che la rotta `acme_hello` abbia uno schema finale con `/admin/hello/{name}` invece di `/hello/{name}`:

La stringa `/site` ora verrà preposta allo schema di ogni rotta caricata dalla nuova risorsa delle rotte.

### Espressioni regolari per gli host nelle rotte importate

Si può impostare un'espressione regolare sull'host nelle rotte importate. Per maggiori informazioni, vedere component-routing-host-imported.

## Visualizzare e fare il debug delle rotte

L'aggiunta e la personalizzazione di rotte è utile, ma lo è anche essere in grado di visualizzare e recuperare informazioni dettagliate sulle rotte. Il modo migliore per vedere tutte le rotte dell'applicazione è tramite il comando di console `router:debug`. Eseguire il comando scrivendo il codice seguente dalla cartella radice del progetto

```
$ php app/console router:debug
```

Il comando visualizzerà un utile elenco di *tutte* le rotte configurate nell'applicazione:

```
homepage                ANY       /
contact                 GET       /contact
contact_process         POST      /contact
article_show            ANY       /articles/{_locale}/{year}/{title}.{_format}
blog                    ANY       /blog/{page}
blog_show               ANY       /blog/{slug}
```

Inoltre è possibile ottenere informazioni molto specifiche su una singola rotta mettendo il nome della rotta dopo il comando:

```
$ php app/console router:debug article_show
```

Si può verificare quale rotta, se esiste, corrisponda a un percorso, usando il comando `router:match`:

```
$ php app/console router:match /blog/my-latest-post
```

Questo comando mostrerà quale rotta corrisponde all'URL.

```
Route "blog_show" matches
```

## Generazione degli URL

Il sistema delle rotte dovrebbe anche essere usato per generare gli URL. In realtà, il routing è un sistema bidirezionale: mappa l'URL in un controllore + parametri e una rotta + parametri di nuovo in un URL. I metodi **:method:'Symfony\\Component\\Routing\\Router::match'** e **:method:'Symfony\\Component\\Routing\\Router::generate'** formano questo sistema bidirezionale. Si prenda la rotta dell'esempio precedente `blog_show`:

```
$params = $this->get('router')->match('/blog/my-blog-post');
// array(
//     'slug'        => 'my-blog-post',
//     '_controller' => 'AppBundle:Blog:show',
// )

$uri = $this->get('router')->generate('blog_show', array(
    'slug' => 'my-blog-post'
));
// /blog/my-blog-post
```

Per generare un URL, è necessario specificare il nome della rotta (ad esempio `blog_show`) ed eventuali caratteri jolly (ad esempio `slug = my-blog-post`) usati nello schema per questa rotta. Con queste informazioni, qualsiasi URL può essere generata facilmente:

```
class MainController extends Controller
{
    public function showAction($slug)
    {
        // ...

        $url = $this->generateUrl(
            'blog_show',
            array('slug' => 'my-blog-post')
        );
    }
}
```

**Note:** In controllori che estendono la classe base di Symfony `Symfony\Bundle\FrameworkBundle\Controller\Controll` si può usare il metodo **:method:'Symfony\\Component\\Routing\\Router::generate'** del servizio `router`:

```
use Symfony\Component\DependencyInjection\ContainerAware;

class MainController extends ContainerAware
{
    public function showAction($slug)
    {
        // ...

        $url = $this->container->get('router')->generate(
            'blog_show',
            array('slug' => 'my-blog-post')
        );
    }
}
```

In una delle prossime sezioni, si imparerà a generare URL dall'interno di un template.

**Tip:** Se la propria applicazione usa richieste AJAX, si potrebbe voler generare URL in JavaScript, che siano basate sulla propria configurazione delle rotte. Usando FOSJsRoutingBundle, lo si può fare:

```
var url = Routing.generate(
    'blog_show',
    {"slug": 'my-blog-post'}
);
```

Per ultetiori informazioni, vedere la documentazione del bundle.

## Generare URL con query string

Il metodo `generate` accetta un array di valori jolly per generare l'URI. Ma se si passano quelli extra, saranno aggiunti all'URI come query string:

```
$this->get('router')->generate('blog', array(
    'page' => 2,
    'category' => 'Symfony'
));
// /blog/2?category=Symfony
```

## Generare URL da un template

Il luogo più comune per generare un URL è all'interno di un template quando si creano i collegamenti tra le varie pagine dell'applicazione. Questo viene fatto esattamente come prima, ma utilizzando una funzione aiutante per i template:

### Generare URL assoluti

Per impostazione predefinita, il router genera URL relativi (ad esempio `/blog`). Per generare un URL assoluto, è sufficiente passare `true` come terzo parametro del metodo `generate()`:

```
$this->generateUrl('blog_show', array('slug' => 'my-blog-post'), true);
// http://www.example.com/blog/my-blog-post
```

In un template Twig, basta usare la funzione `url()` (che genera un URL assoluto) al posto della funzione `path()` (che genera un URL relativo). In PHP, passare `true` a `generateUrl()`:

---

**Note:** L'host che viene usato quando si genera un URL assoluto è rilevato automaticamente in base all'oggetto `Request` corrente. Quando si generano URL assolute fuori dal contesto web (per esempio da riga di comando), non funzionerà. Vedere /cookbook/console/sending_emails per una possibile soluzione.

---

## Riassunto

Il routing è un sistema per mappare l'URL delle richieste in arrivo in una funzione controllore che dovrebbe essere chiamata a processare la richiesta. Il tutto permette sia di creare URL "belle" che di mantenere la funzionalità dell'applicazione disaccoppiata da questi URL. Il routing è un meccanismo bidirezionale, nel senso che dovrebbe anche essere utilizzato per generare gli URL.

## Imparare di più dal ricettario

- /cookbook/routing/scheme

CHAPTER 7

# Creare e usare i template

Come noto, il controllore è responsabile della gestione di ogni richiesta che arriva a un'applicazione Symfony. In realtà, il controllore delega la maggior parte del lavoro pesante ad altri punti, in modo che il codice possa essere testato e riusato. Quando un controllore ha bisogno di generare HTML, CSS o ogni altro contenuto, passa il lavoro al motore dei template. In questo capitolo si imparerà come scrivere potenti template, che possano essere riusati per restituire del contenuto all'utente, popolare corpi di email e altro. Si impareranno scorciatoie, modi intelligenti di estendere template e come riusare il codice di un template

**Note:** La resa dei template è spiegata nel capitolo relativo al *controllore* del libro.

## Template

Un template è un semplice file testuale che può generare qualsiasi formato basato sul testo (HTML, XML, CSV, LaTeX ...). Il tipo più familiare di template è un template *PHP*, un file testuale analizzato da PHP che contiene un misto di testo e codice PHP:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Benvenuto in Symfony!</title>
    </head>
    <body>
        <h1><?php echo $page_title ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item): ?>
                <li>
                    <a href="<?php echo $item->getHref() ?>">
                        <?php echo $item->getCaption() ?>
                    </a>
                </li>
```

```
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

Ma Symfony possiede un linguaggio di template ancora più potente, chiamato Twig. Twig consente di scrivere template concisi e leggibili, più amichevoli per i grafici e, in molti modi, più potenti dei template PHP:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Benvenuto in Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Twig definisce due tipi di sintassi speciali:

**{{ ... }}** "Dice qualcosa": stampa una variabile o il risultato di un'espressione nel template;

**{% ... %}** "Fa qualcosa": un **tag** che controlla la logica del template; è usato per eseguire istruzioni, come il ciclo `for` dell'esempio.

**{# ... #}** "Commenta qualcosa": è l'equivalente della sintassi `/* commento */` di PHP. È usato per aggiungere commenti su riga singola o su righe multiple. Il contenuto dei commenti non viene incluso nella resa delle pagine.

Twig contiene anche dei **filtri**, che modificano il contenuto prima che sia reso. L'esempio seguente rende la variabile `title` tutta maiuscola, prima di renderla:

```
{{ title|upper }}
```

Twig ha una lunga lista di tag e filtri, disponibili in maniera predefinita. Si possono anche aggiungere le proprie estensioni a Twig, se necessario.

**Tip:** È facile registrare un'estensione di Twig: basta creare un nuovo servizio e assegnarli il *tag* `twig.extension`.

Come vedremo nella documentazione, Twig supporta anche le funzioni e si possono aggiungere facilmente nuove funzioni. Per esempio, di seguito viene usato un tag standard `for` e la funzione `cycle` per stampare dieci tag div, con classi alternate `odd` e `even`:

```
{% for i in 0..10 %}
    <div class="{{ cycle(['odd', 'even'], i) }}">
      <!-- un po' di codice HTML -->
    </div>
{% endfor %}
```

In questo capitolo, gli esempi dei template saranno mostrati sia in Twig che in PHP.

**Tip:** Se si sceglie di non usare Twig e lo si disabilita, si dovrà implementare un proprio gestore di eccezioni, tramite l'evento `kernel.exception`.

**Perché Twig?**

I template di Twig sono pensati per essere semplici e non considerano i tag PHP. Questo è intenzionale: il sistema di template di Twig è fatto per esprimere una presentazione, non logica di programmazione. Più si usa Twig, più se ne può apprezzare benefici e distinzione. E, ovviamente, essere amati da tutti i grafici del mondo.

Twig può anche far cose che PHP non può fare, come il controllo degli spazi vuoti, sandbox, escape automatico o contestualizzato e inclusione di funzioni e filtri personalizzati, che hanno effetti solo sui template. Twig possiede poche caratteristiche, che rendono la scrittura di template più facile e concisa. Si prenda il seguente esempio, che combina un ciclo con un'istruzione logica `if`:

```
<ul>
    {% for user in users if user.active %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>Nessun utente trovato</li>
    {% endfor %}
</ul>
```

### Cache di template Twig

Twig è veloce. Ogni template Twig è compilato in una classe nativa PHP, che viene resa a runtime. Le classi compilate sono situate nella cartella `app/cache/{environment}/twig` (dove `{environment}` è l'ambiente, come `dev` o `prod`) e in alcuni casi possono essere utili durante il debug. Vedere *Ambienti* per maggiori informazioni sugli ambienti.

Quando si abilita la modalità di `debug` (tipicamente in ambiente `dev`), un template Twig viene automaticamente ricompilato a ogni modifica subita. Questo vuol dire che durante lo sviluppo si possono tranquillamente effettuare cambiamenti a un template Twig e vedere immediatamente le modifiche, senza doversi preoccupare di pulire la cache.

Quando la modalità di `debug` è disabilitata (tipicamente in ambiente `prod`), tuttavia, occorre pulire la cache di Twig, in modo che i template Twig siano rigenerati. Si ricordi di farlo al deploy della propria applicazione.

## Ereditarietà dei template e layout

Molto spesso, i template di un progetto condividono elementi comuni, come la testata, il piè di pagina, una barra laterale e altro. In Symfony, ci piace pensare a questo problema in modo differente: un template può essere decorato da un altro template. Funziona esattamente come per le classi PHP: l'ereditarietà dei template consente di costruire un template "layout" di base, che contiene tutti gli elementi comuni del proprio sito, definiti come **blocchi** (li si pensi come "classi PHP con metodi base"). Un template figlio può estendere un layout di base e sovrascrivere uno qualsiasi dei suoi blocchi (li si pensi come "sottoclassi PHP che sovrascrivono alcuni metodi della classe genitrice").

Primo, costruire un file per il layout di base:

**Note:** Sebbene la discussione sull'ereditarietà dei template sia relativa a Twig, la filosofia è condivisa tra template Twig e template PHP.

Questo template definisce lo scheletro del documento HTML di base di una semplice pagina a due colonne. In questo esempio, tre aree `{% block %}` sono definite (`title`, `sidebar` e `body`). Ciascun blocco può essere sovrascritto da un template figlio o lasciato alla sua implementazione predefinita. Questo template potrebbe anche essere reso direttamente. In questo caso, i blocchi `title`, `sidebar` e `body` manterrebbero semplicemente i valori predefiniti usati in questo template.

Un template figlio potrebbe assomigliare a questo:

---

**Note:** Il template padre è identificato da una speciale sintassi di stringa (`base.html.twig`) che indica che il template si trova nella cartella `app/Resources/views` del progetto. Si può acnhe usare il nome logico equivalente, `::base.html.twig`. Questa convenzione di nomi è spiegata nel dettaglio in *Nomi e posizioni dei template*.

---

La chiave dell'ereditarietà dei template è il tag `{% extends %}`. Questo dice al motore dei template di valutare prima il template base, che imposta il layout e definisce i vari blocchi. Quindi viene reso il template figlio e i blocchi `title` e `body` del padre vengono rimpiazzati da quelli del figlio. A seconda del valore di `blog_entries`, l'output potrebbe assomigliare a questo:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>I post fighi del mio blog</title>
    </head>
    <body>
        <div id="sidebar">
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
        </div>

        <div id="content">
            <h2>Il mio primo post</h2>
            <p>Il testo del primo post.</p>

            <h2>Un altro post</h2>
            <p>Il testo del secondo post.</p>
        </div>
    </body>
</html>
```

Si noti che, siccome il template figlio non definisce un blocco `sidebar`, viene usato al suo posto il valore del template padre. Il contenuto di un tag `{% block %}` in un template padre è sempre usato come valore predefinito.

Si possono usare tanti livelli di ereditarietà quanti se ne desiderano. Nella prossima sezione, sarà spiegato un modello comune a tre livelli di ereditarietà, insieme al modo in cui i template sono organizzati in un progetto Symfony.

Quando si lavora con l'ereditarietà dei template, ci sono alcuni concetti da tenere a mente:

- se si usa `{% extends %}` in un template, deve essere il primo tag di quel template.
- Più tag `{% block %}` si hanno in un template, meglio è. Si ricordi che i template figli non devono definire tutti i blocchi del padre, quindi si possono creare molti blocchi nei template base e dar loro dei valori predefiniti adeguati. Più blocchi si hanno in un template base, più sarà flessibile il layout.
- Se ci si trova ad aver duplicato del contenuto in un certo numero di template, vuol dire che probabilmente si dovrebbe spostare tale contenuto in un `{% block %}` di un template padre. In alcuni casi, una soluzione

migliore potrebbe essere spostare il contenuto in un nuovo template e usare `include` (vedere *Includere altri template*).

- Se occorre prendere il contenuto di un blocco da un template padre, si può usare la funzione `{{ parent() }}`. È utile quando si vuole aggiungere il contenuto di un template padre, invece di sovrascriverlo completamente:

```
{% block sidebar %}
    <h3>Sommario</h3>

    {# ... #}

    {{ parent() }}
{% endblock %}
```

## Nomi e posizioni dei template

Per impostazione predefinita, i template possono stare in una di queste posizioni:

**app/Resources/views/** La cartella `views` di un'applicazione può contenere template di base a livello di applicazione (p.e. i layout dell'applicazione), ma anche template che sovrascrivono template di bundle (vedere *Sovrascrivere template dei bundle*);

**percorso/bundle/Resources/views/** Ogni bundle ha i suoi template, nella sua cartella `Resources/views` (e nelle sottocartelle). Se si pensa di voler condividere un bundle, si dovrebbero mettere i template nel bundle invece che nella cartella `app/`.

La maggior parte dei template usati si trovano nella cartella `app/Resources/views/`. Il percorso che si userà sarà relativo a tale cartella. Per esempio, per rendere o estendere `app/Resources/views/base.html.twig`, si userà il percorso `base.html.twig` e per rendere o estendere `app/Resources/views/blog/index.html.twig`, si userà il percorso `blog/index.html.twig` path.

### Riferimenti ai template in un bundle

Symfony usa una sintassi stringa **bundle**:**controllore**:**template** per i template. Questo consente diversi tipi di template, ciascuno in un posto specifico:

- `AcmeBlogBundle:Blog:index.html.twig`: Questa sintassi è usata per specificare un template per una determinata pagina. Le tre parti della stringa, ognuna separata da due-punti (`:`), hanno il seguente significato:
  - `AcmeBlogBundle`: (*bundle*) il template è dentro AcmeBlogBundle (p.e. `src/Acme/BlogBundle`);
  - `Blog`: (*cartella*) indica che il template è nella sottocartella `Blog` di `Resources/views`;
  - `index.html.twig`: (*nome di file*) il nome del file è `index.html.twig`.

  Ipotizzando che AcmeBlogBundle sia dentro `src/Acme/BlogBundle`, il percorso finale del layout sarebbe `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.
- `AcmeBlogBundle::layout.html.twig`: Questa sintassi si riferisce a un template di base specifico di AcmeBlogBundle. Poiché la parte centrale, "cartella", manca, (p.e. `Blog`), il template si trova in `Resources/views/layout.html.twig` dentro AcmeBlogBundle. Ci sono due simboli di "due punti" al centro della stringa, quando manca la parte della sottocartella del controllore.

Nella sezione *Sovrascrivere template dei bundle* si potrà trovare come ogni template dentro AcmeBlogBundle, per esempio, possa essere sovrascritto mettendo un template con lo stesso nome nella cartella `app/Resources/AcmeBlogBundle/views/`. Questo dà la possibilità di sovrascrivere template di qualsiasi bundle.

**Tip:** Si spera che la sintassi dei nomi risulti familiare: è la stessa convenzione di nomi usata per lo *Schema per il nome dei controllori*.

### Suffissi dei template

Ogni nome di template ha due estensioni, che specificano *formato* e *motore* del template stesso.

| Nome del file | Formato | Motore |
|---|---|---|
| `blog/index.html.twig` | HTML | Twig |
| `blog/index.html.php` | HTML | PHP |
| `blog/index.css.twig` | CSS | Twig |

Per impostazione predefinita, ogni template Symfony può essere scritto in Twig o in PHP, e l'ultima parte dell'estensione (p.e. `.twig` o `.php`) specifica quale di questi due *motori* va usata. La prima parte dell'estensione, (p.e. `.html`, `.css`, ecc.) è il formato finale che il template genererà. Diversamente dal motore, che determina il modo in cui Symfony analizza il template, si tratta di una tattica organizzativa usata nel caso in cui alcune risorse debbano essere rese come HTML (`index.html.twig`), XML (`index.xml.twig`) o in altri formati. Per maggiori informazioni, leggere la sezione *Formati di template*.

**Note:** I "motori" disponibili possono essere configurati e se ne possono aggiungere di nuovi. Vedere *Configurazione dei template* per maggiori dettagli.

## Tag e aiutanti

Dopo aver parlato delle basi dei template, di che nomi abbiano e di come si possa usare l'ereditarietà, la parte più difficile è passata. In questa sezione, si potranno conoscere un gran numero di strumenti disponibili per aiutare a compiere i compiti più comuni sui template, come includere altri template, collegare pagine e inserire immagini.

Symfony dispone di molti tag di Twig specializzati e di molte funzioni, che facilitano il lavoro del progettista di template. In PHP, il sistema di template fornisce un sistema estensibile di *aiutanti*, che fornisce utili caratteristiche nel contesto dei template.

Abbiamo già visto i tag predefiniti (`{% block %}` e `{% extends %}`), così come un esempio di aiutante PHP (`$view['slots']`). Vediamone alcuni altri.

### Includere altri template

Spesso si vorranno includere lo stesso template o lo stesso pezzo di codice in pagine diverse. Per esempio, in un'applicazione con "nuovi articoli", il codice del template che mostra un articolo potrebbe essere usato sulla pagina dei dettagli dell'articolo, un una pagina che mostra gli articoli più popolari o in una lista degli articoli più recenti.

Quando occorre riusare un pezzo di codice PHP, tipicamente si posta il codice in una nuova classe o funzione PHP. Lo stesso vale per i template. Spostando il codice del template da riusare in un template a parte, può essere incluso in qualsiasi altro template. Primo, creare il template che occorrerà riusare.

Includere questo template da un altro template è semplice:

Il template è incluso usando il tag `{{ include }}`. Si noti che il nome del template segue le stesse tipiche convenzioni. Il template `articleDetails.html.twig` usa una variabile `article`, che viene passata. In questo caso,

lo si può evitare, perché tutte le variabili disponibili in `list.html.twig` lo sono anche in `articleDetails.html.twig` (a meno che non si imposti with_context a `false`.

**Tip:** La sintassi `{'article': article}` è la sintassi standard di Twig per gli array associativi (cioè con chiavi non numeriche). Se si avesse bisogno di passare più elementi, si può fare in questo modo: `{'pippo': pippo, 'pluto': pluto}`.

## Inserire controllori

A volte occorre fare di più che includere semplici template. Si supponga di avere nel proprio layout una barra laterale, che contiene i tre articoli più recenti. Recuperare i tre articoli potrebbe implicare una query alla base dati o l'esecuzione di altra logica, che non si può fare dentro a un template.

La soluzione è semplicemente l'inserimento del risultato di un intero controllore dal proprio template. Primo, creare un controllore che rende un certo numero di articoli recenti:

```
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

// ...

class ArticleController extends Controller
{
    public function recentArticlesAction($max = 3)
    {
        // chiamare la base dati o altra logica
        // per ottenere "$max" articoli recenti
        $articles = ...;

        return $this->render(
            'article/recent_list.html.twig',
            array('articles' => $articles)
        );
    }
}
```

Il template `recentList` è molto semplice:

**Note:** Si noti che l'URL dell'articolo è stato inserito a mano in questo esempio (p.e. `/article/*slug*`). Questa non è una buona pratica. Nella prossima sezione, vedremo come farlo correttamente.

Per includere il controllore, occorrerà farvi riferimento con la sintassi standard per i controllori (cioè **bundle**:**controllore**:**azione**):

Ogni volta che ci si trova ad aver bisogno di una variabile o di un pezzo di informazione a cui non si ha accesso in un template, considerare di rendere un controllore. I controllori sono veloci da eseguire e promuovono buona organizzazione e riuso del codice. Ovviamente, come tutti i controllori, dovrebbero idealmente essere snelli, perché la maggior parte del codice dovrebbe trovarsi nei servizi, che sono riusabili.

## Contenuto asincrono con hinclude.js

Si possono inserire controllori in modo asincrono, con la libreria hinclude.js. Poiché il contenuto incluso proviene da un'altra pagina (o da un altro controllore), Symfony usa l'aiutante standard `render` per configurare i tag `hinclude`:

**Note:** hinclude.js deve essere incluso nella pagina.

**Note:** Quando si usa un controllore invece di un URL, occorre abilitare la configurazione `fragments`:

Il contenuto predefinito (visibile durante il caricamento o senza JavaScript) può essere impostato in modo globale nella configurazione dell'applicazione:

Si possono definire template predefiniti per funzione `render` (che sovrascriveranno qualsiasi template predefinito globale):

Oppure si può specificare una stringa da mostrare come contenuto predefinito:

## Collegare le pagine

Creare collegamenti alle altre pagine della propria applicazione è uno dei lavori più comuni per un template. Invece di inserire a mano URL nei template, usare la funzione `path` di Twig (o l'helper `router` in PHP) per generare URL basati sulla configurazione delle rotte. Più avanti, se si vuole modificare l'URL di una particolare pagina, tutto ciò di cui si avrà bisogno è cambiare la configurazione delle rotte: i template genereranno automaticamente il nuovo URL.

Primo, collegare la pagina "_welcome", accessibile tramite la seguente configurazione delle rotte:

Per collegare la pagina, usare la funzione `path` di Twig e riferirsi alla rotta:

Come ci si aspettava, questo genererà l'URL `/`. Vediamo come funziona con una rotta più complessa:

In questo caso, occorre specificare sia il nome della rotta (`article_show`) che il valore del parametro `{slug}`. Usando questa rotta, rivisitiamo il template `recentList` della sezione precedente e colleghiamo correttamente gli articoli:

**Tip:** Si può anche generare un URL assoluto, usando la funzione `url` di Twig:

```
<a href="{{ url('_welcome') }}">Home</a>
```

Lo stesso si può fare nei template PHP, passando un terzo parametro al metodo `generate()`:

```
<a href="<?php echo $view['router']->generate(
    '_welcome',
    array(),
    true
) ?>">Home</a>
```

## Collegare le risorse

I template solitamente hanno anche riferimenti a immagini, JavaScript, fogli di stile e altre risorse. Certamente, si potrebbe inserire manualmente il percorso a tali risorse (p.e. `/images/logo.png`), ma Symfony fornisce un'opzione più dinamica, tramite la funzione `asset` di Twig:

Lo scopo principale della funzione `asset` è rendere più portabile la propria applicazione. Se l'applicazione si trova nella radice dell'host (p.e. http://example.com), i percorsi resi dovrebbero essere del tipo `/images/logo.png`. Se invece l'applicazione si trova in una sotto-cartella (p.e. http://example.com/my_app), ogni percorso dovrebbe includere la sotto-cartella (p.e. `/my_app/images/logo.png`). La funzione `asset` si prende cura di questi aspetti, determinando in che modo è usata l'applicazione e generando i percorsi adeguati.

Inoltre, se si usa la funzione `asset`, Symfony può aggiungere automaticamente un parametro all'URL della risorsa, per garantire che le risorse statiche aggiornate non siano messe in cache. Per esempio, `/images/logo.png` potrebbe comparire come `/images/logo.png?v2`. Per ulteriori informazioni, vedere l'opzione di configurazione ref-framework-assets-version. New in version 2.5: L'impostazione di URL versionati per singola risorsa è stato introdotto in Symfony 2.5.

Se occorre specificare una versione per una risorsa specifica, si può impostare il quarto parametro (o il parametro `version`) alla versione desiderata:

Se non si fornisce una versione o si passa `null`, sarà usata la versione predefinita (da ref-framework-assets-version). Se si passa `false`, l'URL versionato sarà disattivato per questa risorsa.

New in version 2.5: Gli URL assoluti per le risorse sono stati introdotti in Symfony 2.5.

Se occorrono URL assoluti per gli asset, si può impostare il terzo parametro (o il parametro `absolute`) a `true`:

## Includere fogli di stile e Javascript in Twig

Nessun sito sarebbe completo senza l'inclusione di file Javascript e fogli di stile. In Symfony, l'inclusione di tali risorse è gestita elegantemente sfruttando l'ereditarietà dei template.

---

**Tip:** Questa sezione insegnerà la filosofia che sta dietro l'inclusione di fogli di stile e Javascript in Symfony. Symfony dispone di un'altra libreria, chiamata Assetic, che segue la stessa filosofia, ma consente di fare cose molto più interessanti con queste risorse. Per maggiori informazioni sull'uso di Assetic, vedere /cookbook/assetic/asset_management.

---

Iniziamo aggiungendo due blocchi al template di base, che conterranno le risorse: uno chiamato `stylesheets`, dentro al tag `head`, e l'altro chiamato `javascripts`, appena prima della chiusura del tag `body`. Questi blocchi conterranno tutti i fogli di stile e i Javascript che occorrerano al sito:

```
{# app/Resources/views/base.html.twig #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('css/main.css') }}" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('js/main.js') }}"></script>
        {% endblock %}
    </body>
</html>
```

È così facile! Ma che succede quando si ha bisogno di includere un foglio di stile o un Javascript aggiuntivo in un template figlio? Per esempio, supponiamo di avere una pagina di contatti e che occorra includere un foglio di stile `contact.css` *solo* su tale pagina. Da dentro il template della pagina di contatti, fare come segue:

```
{# app/Resources/views/Contact/contact.html.twig #}
{% extends 'base.html.twig' %}

{% block stylesheets %}
```

```
    {{ parent() }}

    <link href="{{ asset('css/contact.css') }}" rel="stylesheet" />
{% endblock %}

{# ... #}
```

Nel template figlio, basta sovrascrivere il blocco `stylesheets` e inserire il nuovo tag del foglio di stile nel blocco stesso. Ovviamente, poiché vogliamo aggiungere contenuto al blocco padre (e non *sostituirlo*), occorre usare la funzione `parent()` di Twig, per includere tutto ciò che sta nel blocco `stylesheets` del template di base.

Si possono anche includere risorse dalla cartella `Resources/public` del proprio bundle. Occorre poi eseguire il comando `php app/console assets:install target [--symlink]`, che copia (o collega) i file nella posizione corretta (la posizione predefinita è sotto la cartella "web").

```
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" rel="stylesheet" />
```

Il risultato finale è una pagina che include i fogli di stile `main.css` e `contact.css`.

## Variabili globali nei template

Durante ogni richiesta, Symfony imposta una variabile globale `app`, sia nei template Twig che in quelli PHP. La variabile `app` è un'istanza di `Symfony\Bundle\FrameworkBundle\Templating\GlobalVariables`, che dà accesso automaticamente ad alcune variabili specifiche dell'applicazione:

**`app.security`** Il contesto della sicurezza.

**`app.user`** L'oggetto dell'utente attuale.

**`app.request`** L'oggetto richiesta.

**`app.session`** L'oggetto sessione.

**`app.environment`** L'ambiente attuale (dev, prod, ecc).

**`app.debug`** True se in debug. False altrimenti.

**Tip:** Si possono aggiungere le proprie variabili globali ai template. Si veda la ricetta Variabili globali.

## Configurare e usare il servizio `templating`

Il cuore del sistema dei template di Symfony è il motore dei template. L'oggetto speciale `Engine` è responsabile della resa dei template e della restituzione del loro contenuto. Quando si rende un template in un controllore, per esempio, si sta in realtà usando il servizio del motore dei template. Per esempio:

```
return $this->render('article/index.html.twig');
```

equivale a:

```
use Symfony\Component\HttpFoundation\Response;

$engine = $this->container->get('templating');
$content = $engine->render('article/index.html.twig');
```

```
return $response = new Response($content);
```

Il motore (o "servizio") dei template è pre-configurato per funzionare automaticamente dentro a Symfony. Può anche essere ulteriormente configurato nel file di configurazione dell'applicazione:

Sono disponibili diverse opzioni di configurazione, coperte nell'Appendice: configurazione.

**Note:** Il motore `twig` è obbligatorio per poter usare il profilatore web (così come molti altri bundle di terze parti).

## Sovrascrivere template dei bundle

La comunità di Symfony si vanta di creare e mantenere bundle di alta qualità (vedere KnpBundles.com) per un gran numero di diverse caratteristiche. Quando si usa un bundle di terze parti, probabilmente occorrerà sovrascrivere e personalizzare uno o più dei suoi template.

Si supponga di aver incluso l'immaginario bundle AcmeBlogBundle in un progetto. Pur essendo soddisfatti, si vuole sovrascrivere la pagina "list" del blog, per personalizzare il codice e renderlo specifico per l'applicazione. Analizzando il controllore `Blog` di AcmeBlogBundle, si trova:

```
public function indexAction()
{
    // logica per recuperare i blog
    $blogs = ...;

    $this->render(
        'AcmeBlogBundle:Blog:index.html.twig',
        array('blogs' => $blogs)
    );
}
```

Quando viene reso `AcmeBlogBundle:Blog:index.html.twig`, Symfony cerca il template in due diversi posti:

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

Per sovrascrivere il template del bundle, basta copiare il file `index.html.twig` dal bundle a `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (la cartella `app/Resources/AcmeBlogBundle` non esiste ancora, quindi occorre crearla). Ora si può personalizzare il template.

**Caution:** Se si aggiunge un template in una nuova posizione, *potrebbe* essere necessario pulire la cache (`php app/console cache:clear`), anche in modalità debug.

Questa logica si applica anche ai template base dei bundle. Si supponga che ogni template in AcmeBlogBundle erediti da un template base chiamato `AcmeBlogBundle::layout.html.twig`. Esattamente come prima, Symfony cercherà il template i questi due posti:

1. `app/Resources/AcmeBlogBundle/views/layout.html.twig`
2. `src/Acme/BlogBundle/Resources/views/layout.html.twig`

Anche qui, per sovrascrivere il template, basta copiarlo dal bundle a `app/Resources/AcmeBlogBundle/views/layout.html.twig`. Ora lo si può personalizzare.

Facendo un passo indietro, si vedrà che Symfony inizia sempre a cercare un template nella cartella `app/Resources/{NOME_BUNDLE}/views/`. Se il template non c'è, continua verificando nella cartella `Resources/views` del bundle stesso. Questo vuol dire che ogni template di bundle può essere sovrascritto, inserendolo nella sotto-cartella `app/Resources` appropriata.

---

**Note:** Si possono anche sovrascrivere template da dentro un bundle, usando l'ereditarietà dei bundle. Per maggiori informazioni, vedere /cookbook/bundles/inheritance.

---

### Sovrascrivere template del nucleo

Essendo il framework Symfony esso stesso un bundle, i template del nucleo possono essere sovrascritti allo stesso modo. Per esempio, TwigBundle contiene diversi template "exception" ed "error", che possono essere sovrascritti, copiandoli dalla cartella `Resources/views/Exception` di TwigBundle a, come si può immaginare, la cartella `app/Resources/TwigBundle/views/Exception`.

## Ereditarietà a tre livelli

Un modo comune per usare l'ereditarietà è l'approccio a tre livelli. Questo metodo funziona perfettamente con i tre diversi tipi di template di cui abbiamo appena parlato:

- Creare un file `app/Resources/views/base.html.twig` che contenga il layout principale per la propria applicazione (come nell'esempio precedente). Internamente, questo template si chiama `base.html.twig`;
- Creare un template per ogni "sezione" del proprio sito. Per esempio, il blog avrebbe un template di nome `Blog/layout.html.twig`, che contiene solo elementi specifici alla sezione blog;

```
{# app/Resources/views/blog/layout.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Applicazione blog</h1>

    {% block content %}{% endblock %}
{% endblock %}
```

- Creare i singoli template per ogni pagina, facendo estendere il template della sezione appropriata. Per esempio, la pagina "index" avrebbe un nome come `Blog/index.html.twig` e mostrerebbe la lista dei post del blog.

```
{# app/Resources/views/blog/index.html.twig #}
{% extends 'blog/layout.html.twig' %}

{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

Si noti che questo template estende il template di sezione (`Blog/layout.html.twig`), che a sua volte estende il layout base dell'applicazione (`::base.html.twig`). Questo è il modello di ereditarietà a tre livelli.

Durante la costruzione della propria applicazione, si può scegliere di seguire questo metodo oppure semplicemente far estendere direttamente a ogni template di pagina il template base dell'applicazione (p.e. `{% extends 'base.html.twig' %}`). Il modello a tre template è una best practice usata dai bundle dei venditori, in modo che il template base di un bundle possa essere facilmente sovrascritto per estendere correttamente il layout base della propria applicazione.

## Escape dell'output

Quando si genera HTML da un template, c'è sempre il rischio che una variabile possa mostrare HTML indesiderato o codice pericoloso lato client. Il risultato è che il contenuto dinamico potrebbe rompere il codice HTML della pagina risultante o consentire a un utente malintenzionato di eseguire un attacco Cross Site Scripting (XSS). Consideriamo questo classico esempio:

Si immagini che l'utente inserisca nel suo nome il seguente codice:

```
<script>alert('ciao!')</script>
```

Senza alcun escape dell'output, il template risultante causerebbe la comparsa di una finestra di alert JavaScript:

```
Ciao <script>alert('ciao!')</script>
```

Sebbene possa sembrare innocuo, se un utente arriva a tal punto, lo stesso utente sarebbe in grado di scrivere Javascript che esegua azioni dannose all'interno dell'area di un utente legittimo e ignaro.

La risposta a questo problema è l'escape dell'output. Con l'escape attivo, lo stesso template verrebbe reso in modo innocuo e scriverebbe alla lettera il tag `script` su schermo:

```
Ciao &lt;script&gt;alert(&#39;ciao!&#39;)&lt;/script&gt;
```

L'approccio dei sistemi di template Twig e PHP a questo problema sono diversi. Se si usa Twig, l'escape è attivo in modo predefinito e si è al sicuro. In PHP, l'escape dell'output non è automatico, il che vuol dire che occorre applicarlo a mano, dove necessario.

### Escape dell'output in Twig

Se si usano i template Twig, l'escape dell'output è attivo in modo predefinito. Questo vuol dire che si è protetti dalle conseguenze non intenzionali del codice inviato dall'utente. Per impostazione predefinita, l'escape dell'output assume che il contenuto sia sotto escape per l'output HTML.

In alcuni casi, si avrà bisogno di disabilitare l'escape dell'output, quando si avrà bisogno di rendere una variabile affidabile che contiene markup. Supponiamo che gli utenti amministratori siano abilitati a scrivere articoli che contengano codice HTML. Per impostazione predefinita, Twig mostrerà l'articolo con escape.

Per renderlo normalmente, aggiungere il filtro `raw`:

```
{{ article.body|raw }}
```

Si può anche disabilitare l'escape dell'output dentro a un `{% block %}` o per un intero template. Per maggiori informazioni, vedere Escape dell'output nella documentazione di Twig.

### Escape dell'output in PHP

L'escape dell'output non è automatico, se si usano i template PHP. Questo vuol dire che, a meno che non scelga esplicitamente di passare una variabile sotto escape, non si è protetti. Per usare l'escape, usare il metodo speciale

`escape()`:

```
Ciao <?php echo $view->escape($name) ?>
```

Per impostazione predefinita, il metodo `escape()` assume che la variabile sia resa in un contesto HTML (quindi l'escape renderà la variabile sicura per HTML). Il secondo parametro consente di cambiare contesto. Per esempio per mostrare qualcosa in una stringa Javascript, usare il contesto `js`:

```
var myMsg = 'Ciao <?php echo $view->escape($name, 'js') ?>';
```

## Debug

Quando si usa PHP, si può ricorrere a **:phpfunction:'var_dump'**, se occorre trovare rapidamente il valore di una variabile passata. Può essere utile, per esempio, nel proprio controllore. Si può ottenere lo stesso risultato con Twig, usando l'estensione debug.

Si può fare un dump dei parametri nei template, usando la funzione `dump`:

```
{# app/Resources/views/article/recent_list.html.twig #}
{{ dump(articles) }}

{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Il dump delle variabili avverrà solo se l'impostazione `debug` (in `config.yml`) è `true`. Questo vuol dire che, per impostazione predefinita, il dump avverrà in ambiente `dev`, ma non in `prod`.

## Verifica sintattica

Si possono cercare eventuali errori di sintassi nei template Twig, usando il comando `twig:lint`:

```
# Verifica per nome del file:
$ php app/console twig:lint app/Resources/views/article/recent_list.html.twig

# oppure per cartella:
$ php app/console twig:lint app/Resources/views
```

## Formati di template

I template sono un modo generico per rendere contenuti in *qualsiasi* formato. Pur usando nella maggior parte dei casi i template per rendere contenuti HTML, un template può generare altrettanto facilmente Javascript, CSS, XML o qualsiasi altro formato desiderato.

Per esempio, la stessa "risorsa" spesso è resa in molti formati diversi. Per rendere una pagina in XML, basta includere il formato nel nome del template:

- *nome del template XML*: `article/index.xml.twig`
- *nome del file del template XML*: `index.xml.twig`

In realtà, questo non è niente più che una convenzione sui nomi e il template non è effettivamente resto in modo diverso in base al suo formato.

In molti casi, si potrebbe voler consentire a un singolo controllore di rendere formati diversi, in base al "formato di richiesta". Per questa ragione, una soluzione comune è fare come segue:

```
public function indexAction(Request $request)
{
    $format = $request->getRequestFormat();

    return $this->render('article/index.'.$format.'.twig');
}
```

Il metodo `getRequestFormat` dell'oggetto `Request` ha come valore predefinito `html`, ma può restituire qualsiasi altro formato, in base al formato richiesto dall'utente. Il formato di richiesta è spesso gestito dalle rotte, quando una rotta è configurata in modo che `/contact` imposti il formato di richiesta a `html`, mentre `/contact.xml` lo imposti a `xml`. Per maggiori informazioni, vedere *Esempi avanzati nel capitolo delle rotte*.

Per creare collegamenti che includano il formato, usare la chiave `_format` come parametro:

## Considerazioni finali

Il motore dei template in Symfony è un potente strumento, che può essere usato ogni volta che occorre generare contenuto relativo alla presentazione in HTML, XML o altri formati. Sebbene i template siano un modo comune per generare contenuti in un controllore, i loro utilizzo non è obbligatorio. L'oggetto `Response` restituito da un controllore può essere creato con o senza l'uso di un template:

```
// crea un oggetto Response il cui contenuto è il template reso
$response = $this->render('article/index.html.twig');

// crea un oggetto Response il cui contenuto è semplice testo
$response = new Response('contenuto della risposta');
```

Il motore dei template di Symfony è molto flessibile e mette a disposizione due sistemi di template: i tradizionali template *PHP* e i potenti e raffinati template *Twig*. Entrambi supportano una gerarchia di template e sono distribuiti con un ricco insieme di funzioni aiutanti, capaci di eseguire i compiti più comuni.

Complessivamente, l'argomento template dovrebbe essere considerato come un potente strumento a disposizione. In alcuni casi, si potrebbe non aver bisogno di rendere un template, in Symfony, questo non è assolutamente un problema.

## Imparare di più con il ricettario

- /cookbook/templating/PHP
- /cookbook/controller/error_pages
- /cookbook/templating/twig_extension

CHAPTER 8

# Configurare Symfony (e gli ambienti)

(TODO da tradure...)

# CHAPTER 9

## Il sistema dei bundle

Un bundle è simile a un plugin in altri software, ma anche meglio. La differenza fondamentale è che *tutto* è un bundle in Symfony, incluse le funzionalità fondamentali del framework o il codice scritto per la propria applicazione. I bundle sono cittadini di prima classe in Symfony. Questo fornisce la flessibilità di usare caratteristiche già pronte impacchettate in *bundle di terze parti* o di distribuire i propri bundle. Rende facile scegliere quali caratteristiche abilitare nella propria applicazione per ottimizzarla nel modo preferito.

---

**Note:** Pur trovando qui i fondamentali, un'intera ricetta è dedicata all'organizzazione e alle pratiche migliori in bundle.

---

Un bundle è semplicemente un insieme strutturato di file dentro una cartella, che implementa una singola caratteristica. Si potrebbe creare un `BlogBundle`, un `ForumBundle` o un bundle per la gestione degli utenti (molti di questi già esistono come bundle open source). Ogni cartella contiene tutto ciò che è relativo a quella caratteristica, inclusi file PHP, template, fogli di stile, JavaScript, test e tutto il resto. Ogni aspetto di una caratteristica esiste in un bundle e ogni caratteristica risiede in un bundle.

Un'applicazione è composta di bundle, come definito nel metodo `registerBundles()` della classe `AppKernel`:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new AppBundle\AppBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
```

```
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

Col metodo `registerBundles()`, si ha il controllo totale su quali bundle siano usati dalla propria applicazione (inclusi i bundle del nucleo di Symfony).

**Tip:** Un bundle può stare *ovunque*, purché possa essere auto-caricato (tramite l'autoloader configurato in `app/autoload.php`).

## Creare un bundle

Symfony Standard Edition contiene un task utile per creare un bundle pienamente funzionante. Ma anche creare un bundle a mano è molto facile.

Per dimostrare quanto è semplice il sistema dei bundle, creiamo un nuovo bundle, chiamato `AcmeTestBundle`, e abilitiamolo.

**Tip:** La parte `Acme` è solo un nome fittizio, che andrebbe sostituito da un nome di "venditore" che rappresenti la propria organizzazione (p.e. `ABCTestBundle` per un'azienda chiamata `ABC`).

Iniziamo creando una cartella `src/Acme/TestBundle/` e aggiungendo un nuovo file chiamato `AcmeTestBundle.php`:

```
// src/Acme/TestBundle/AcmeTestBundle.php
namespace Acme\TestBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeTestBundle extends Bundle
{
}
```

**Tip:** Il nome AcmeTestBundle segue le convenzioni sui nomi dei bundle. Si potrebbe anche scegliere di accorciare il nome del bundle semplicemente a TestBundle, chiamando la classe `TestBundle` (e chiamando il file `TestBundle.php`).

Questa classe vuota è l'unico pezzo necessario a creare un nuovo bundle. Sebbene solitamente vuota, questa classe è potente e può essere usata per personalizzare il comportamento del bundle.

Ora che il bundle è stato creato, va abilitato tramite la classe `AppKernel`:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
```

```
        // registra il bundle
        new Acme\TestBundle\AcmeTestBundle(),
    );
    // ...

    return $bundles;
}
```

Sebbene non faccia ancora nulla, AcmeTestBundle è ora pronto per essere usato.

Symfony fornisce anche un'interfaccia a linea di comando per generare uno scheletro di base per un bundle:

```
$ php app/console generate:bundle --namespace=Acme/TestBundle
```

Lo scheletro del bundle è generato con controllore, template e rotte, tutti personalizzabili. Approfondiremo più avanti la linea di comando di Symfony.

---

**Tip:** Ogni volta che si crea un nuovo bundle o che si usa un bundle di terze parti, assicurarsi sempre che il bundle sia abilitato in `registerBundles()`. Se si usa il comando `generate:bundle`, l'abilitazione è automatica.

---

## Struttura delle cartelle dei bundle

La struttura delle cartelle di un bundle è semplice e flessibile. Per impostazione predefinita, il sistema dei bundle segue un insieme di convenzioni, che aiutano a mantenere il codice coerente tra tutti i bundle di Symfony. Si dia un'occhiata a `AcmeHelloBundle`, perché contiene alcuni degli elementi più comuni di un bundle:

**`Controller/`** contiene i controllori del bundle (p.e. `HelloController.php`);

**`DependencyInjection/`** contiene alcune estensioni di classi, che possono importare configurazioni di servizi, registrare passi di compilatore o altro (tale cartella non è indispensabile);

**`Resources/config/`** contiene la configurazione, compresa la configurazione delle rotte (p.e. `routing.yml`);

**`Resources/views/`** contiene i template, organizzati per nome di controllore (p.e. `Hello/index.html.twig`);

**`Resources/public/`** contiene le risorse per il web (immagini, fogli di stile, ecc.) ed è copiata o collegata simbolicamente alla cartella `web/` del progetto, tramite il comando `assets:install`;

**`Tests/`** contiene tutti i test del bundle.

Un bundle può essere grande o piccolo, come la caratteristica che implementa. Contiene solo i file che occorrono e niente altro.

Andando avanti nel libro, si imparerà come persistere gli oggetti in una base dati, creare e validare form, creare traduzioni per la propria applicazione, scrivere test e molto altro. Ognuno di questi ha il suo posto e il suo ruolo dentro il bundle.

CHAPTER 10

# Basi di dati e Doctrine

Uno dei compiti più comuni e impegnativi per qualsiasi applicazione implica la persistenza e la lettura di informazioni da una base dati. Sebbene il framework Symfony non si integri con un ORM in modo predefinito, Symfony Standard Edition, la distribuzione più usata, dispone di un'integrazione con Doctrine, una libreria il cui unico scopo è quello di fornire potenti strumenti per facilitare tali compiti. In questo capitolo, si imparerà la filosofia alla base di Doctrine e si vedrà quanto possa essere facile lavorare con una base dati.

**Note:** Doctrine è totalmente disaccoppiato da Symfony e il suo utilizzo è facoltativo. Questo capitolo è tutto su Doctrine ORM, che si prefigge lo scopo di consentire una mappatura tra oggetti una base dati relazionale (come *MySQL*, *PostgreSQL* o *Microsoft SQL*). Se si preferisce l'uso di query grezze, lo si può fare facilmente, come spiegato nella ricetta "/cookbook/doctrine/dbal".

Si possono anche persistere dati su MongoDB usando la libreria ODM Doctrine. Per ulteriori informazioni, leggere la documentazione di "DoctrineMongoDBBundle'_".

## Un semplice esempio: un prodotto

Il modo più facile per capire come funziona Doctrine è quello di vederlo in azione. In questa sezione, configureremo una base dati, creeremo un oggetto `Product`, lo persisteremo nella base dati e lo recupereremo da esso.

### Configurazione della base dati

Prima di iniziare, occorre configurare le informazioni sulla connessione alla base dati. Per convenzione, questa informazione solitamente è configurata in un file `app/config/parameters.yml`:

```
# app/config/parameters.yml
parameters:
    database_driver:    pdo_mysql
    database_host:      localhost
    database_name:      progetto_test
```

```
    database_user:      root
    database_password:  password

# ...
```

**Note:** La definizione della configurazione tramite `parameters.yml` è solo una convenzione. I parametri definiti in tale file sono riferiti dal file di configurazione principale durante le impostazioni iniziali di Doctrine:

Separando le informazioni sula base dati in un file a parte, si possono mantenere facilmente diverse versioni del file su ogni server. Si possono anche facilmente memorizzare configurazioni di basi dati (o altre informazioni sensibili) fuori dal progetto, come per esempio dentro la configurazione di Apache. Per ulteriori informazioni, vedere /cookbook/configuration/external_parameters.

Ora che Doctrine ha informazioni sulla base dati, si può fare in modo che crei la base dati al posto nostro:

```
$ php app/console doctrine:database:create
```

**Impostazioni dei caratteri della base dati**

Uno sbaglio che anche programmatori esperti commettono all'inizio di un progetto Symfony è dimenticare di impostare charset e collation nella base dati, finendo con collation di tipo latin, che sono predefinite la maggior parte delle volte. Lo si potrebbe fare anche solo all'inizio, ma spesso si dimentica che lo si può fare anche durante lo sviluppo, in modo abbastanza semplice:

```
$ php app/console doctrine:database:drop --force
$ php app/console doctrine:database:create
```

Non c'è modo di configurare tali valori predefiniti in Doctrine, che prova a essere il più agnostico possibile in termini di configurazione di ambienti. Un modo per risolvere la questione è usare dei valori definiti a livello di server.

Impostare UTF8 come predefinito in MySQL è semplice, basta aggiungere poche righe al file di configurazione (solitamente `my.cnf`):

```
[mysqld]
# La versione 5.5.3 ha introdotto "utf8mb4", che è raccomandato
collation-server     = utf8mb4_general_ci # Sostituisce utf8_general_ci
character-set-server = utf8mb4            # Sostituisce utf8
```

Si raccomanda di non usare il set di caratteri `utf8` di MySQL, poiché non supporta caratteri unicode a 4 byte, quindi le stringhe che li contenessero sarebbero troncate. Il problema è stato risolta nel nuovo set di caratteri utf8mb4.

**Note:** Se si vuole usare SQLite come base dati, occorre impostare il percorso in cui si trova il relativo file:

## Creare una classe entità

Supponiamo di star costruendo un'applicazione in cui si devono elencare dei prodotti. Senza nemmeno pensare a Doctrine o alle basi dati, già sappiamo di aver bisogno di un oggetto `Product` che rappresenti questi prodotti. Creare questa classe dentro la cartella `Entity` di AppBundle:

```
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

class Product
{
    protected $name;
    protected $price;
    protected $description;
}
```

La classe, spesso chiamata "entità" (che vuol dire *una classe di base che contiene dati*), è semplice e aiuta a soddisfare i requisiti di business di necessità di prodotti dell'applicazione. Questa classe non può ancora essere persistita in una base dati, è solo una semplice classe PHP.

**Tip:** Una volta imparati i concetti dietro a Doctrine, si può fare in modo che Doctrine crei questa classe entità al posto nostro. Questo comando porrà delle domande, per aiutare nella costruzione dell'entità:

```
$ php app/console doctrine:generate:entity
```

## Aggiungere informazioni di mappatura

Doctrine consente di lavorare con le basi dati in un modo molto più interessante rispetto al semplice recupero di righe da tabelle basate su colonne in un array. Invece, Doctrine consente di persistere interi *oggetti* sula base dati e di recuperare interi oggetti dalla base dati. Funziona mappando una classe PHP su una tabella di base dati e le proprietà della classe PHP sulle colonne della tabella:


Per fare in modo che Doctrine possa fare ciò, occorre solo creare dei "metadati", ovvero la configurazione che dice esattamente a Doctrine come la classe `Product` e le sue proprietà debbano essere *mappate* sula base dati. Questi metadati possono essere specificati in diversi formati, inclusi YAML, XML o direttamente dentro la classe `Product`, tramite annotazioni:

**Note:** Un bundle può accettare un solo formato di definizione dei metadati. Per esempio, non è possibile mischiare definizioni di metadati in YAML con definizioni tramite annotazioni.

**Tip:** Il nome della tabella è facoltativo e, se omesso, sarà determinato automaticamente in base al nome della classe entità.

Doctrine consente di scegliere tra una grande varietà di tipi di campo, ognuno con le sue opzioni Per informazioni sui tipi disponibili, vedere la sezione *Riferimento sui tipi di campo di Doctrine*.

**See also:**

Si può anche consultare Basic Mapping Documentation di Doctrine per tutti i dettagli sulla mappatura. Se si usano le annotazioni, occorrerà aggiungere a ogni annotazione il prefisso `ORM\` (p.e. `ORM\Column(..)`), che non è mostrato

nella documentazione di Doctrine. Occorrerà anche includere l'istruzione `use Doctrine\ORM\Mapping as ORM;`, che *importa* il prefisso `ORM` delle annotazioni.

**Caution:** Si faccia attenzione che il nome della classe e delle proprietà scelti non siano mappati a delle parole riservate di SQL (come `group` o `user`). Per esempio, se il nome di una classe entità è `Group`, allora il nome predefinito della tabella sarà `group`, che causerà un errore SQL in alcuni sistemi di basi dati. Vedere Reserved SQL keywords documentation di Doctrine per sapere come fare correttamente escape di tali nomi. In alternativa, se si può scegliere liberamente lo schema della base dati, usare semplicemente un nome diverso di tabella o di colonna. Vedere Persistent classes e Property Mapping nella documentazione di Doctrine.

**Note:** Se si usa un'altra libreria o programma che utilizza le annotazioni (come Doxygen), si dovrebbe inserire l'annotazione `@IgnoreAnnotation` nella classe, per indicare a Symfony quali annotazioni ignorare.

Per esempio, per evitare che l'annotazione `@fn` sollevi un'eccezione, aggiungere il seguente:

```
/**
 * @IgnoreAnnotation("fn")
 */
class Product
// ...
```

## Generare getter e setter

Sebbene ora Doctrine sappia come persistere un oggetto `Product` nella base dati, la classe stessa non è molto utile. Poiché `Product` è solo una normale classe PHP, occorre creare dei metodi getter e setter (p.e. `getName()`, `setName()`) per poter accedere alle sue proprietà (essendo le proprietà protette). Fortunatamente, Doctrine può farlo al posto nostro, basta eseguire:

```
$ php app/console doctrine:generate:entities AppBundle/Entity/Product
```

Il comando si assicura che i getter e i setter siano generati per la classe `Product`. È un comando sicuro, lo si può eseguire diverse volte: genererà i getter e i setter solamente se non esistono (ovvero non sostituirà eventuali metodi già presenti).

**Caution:** Si tenga a mente che il generatore di entità di Doctrine produce semplici getter e setter. Si dovrebbero controllare le entità generate e sistemare getter e setter per adattarli alle proprie necessità.

**Di più su `doctrine:generate:entities`**

Con il comando `doctrine:generate:entities` si può:

- generare getter e setter;
- generare classi repository configurate con l'annotazione `@ORM\Entity(repositoryClass="...")`;
- generare il costruttore appropriato per relazioni 1:n e n:m.

Il comando `doctrine:generate:entities` salva una copia di backup del file originale `Product.php`, chiamata `Product.php~`. In alcuni casi, la presenza di questo file può causare un errore "Cannot redeclare

class". Il file può essere rimosso senza problemi. Si può anche usare l'opzione `--no-backup`, per prevenire la generazione di questi file di backup.

Si noti che non è *necessario* usare questo comando. Doctrine non si appoggia alla generazione di codice. Come con le normali classi PHP, occorre solo assicurarsi che le proprietà protected/private abbiano metodi getter e setter. Questo comando è stato creato perché è una cosa comune da fare quando si usa Doctrine.

Si possono anche generare tutte le entità note (cioè ogni classe PHP con informazioni di mappatura di Doctrine) di un bundle o di un intero spazio dei nomi:

```
# genera tutte le entità in AppBundle
$ php app/console doctrine:generate:entities AppBundle

# genera tutte le entità dei bundle nello spazio dei nomi Acme
$ php app/console doctrine:generate:entities Acme
```

**Note:** Doctrine non si cura se le proprietà siano protette o private, o se siano o meno presenti getter o setter per una proprietà. I getter e i setter sono generati qui solo perché necessari per interagire col l'oggetto PHP.

## Creare tabelle e schema della base dati

Ora si ha una classe `Product` usabile, con informazioni di mappatura che consentono a Doctrine di sapere esattamente come persisterla. Ovviamente, non si ha ancora la corrispondente tabella `product` nella propria base dati. Fortunatamente, Doctrine può creare automaticamente tutte le tabelle della base dati necessarie a ogni entità nota nella propria applicazione. Per farlo, eseguire:

```
$ php app/console doctrine:schema:update --force
```

**Tip:** Questo comando è incredibilmente potente. Confronta ciò che la propria base dati *dovrebbe* essere (basandosi sulle informazioni di mappatura delle entità) con ciò che *effettivamente* è, quindi genera le istruzioni SQL necessarie per *aggiornare* la base dati e portarlo a ciò che dovrebbe essere. In altre parole, se si aggiunge una nuova proprietà con metadati di mappatura a `Product` e si esegue nuovamente il task, esso genererà l'istruzione "alter table" necessaria per aggiungere questa nuova colonna alla tabella `product` esistente.

Un modo ancora migliore per trarre vantaggio da questa funzionalità è tramite le migrazioni, che consentono di generare queste istruzioni SQL e di memorizzarle in classi di migrazione, che possono essere eseguite sistematicamente sul server di produzione, per poter tracciare e migrare lo schema della base dati in modo sicuro e affidabile.

La propria base dati ora ha una tabella `product` pienamente funzionante, con le colonne corrispondenti ai metadati specificati.

## Persistere gli oggetti nella base dati

Ora che l'entità `Product` è stata mappata alla corrispondente tabella `product`, si è pronti per persistere i dati nella base dati. Da dentro un controllore, è molto facile. Aggiungere il seguente metodo a `DefaultController` del bundle:

```
// src/AppBundle/Controller/DefaultController.php

// ...
```

```
use AppBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

// ...
public function createAction()
{
    $product = new Product();
    $product->setName('Pippo Pluto');
    $product->setPrice('19.99');
    $product->setDescription('Lorem ipsum dolor');

    $em = $this->getDoctrine()->getManager();

    $em->persist($product);
    $em->flush();

    return new Response('Creato prodotto con id '.$product->getId());
}
```

**Note:** Se si sta seguendo questo esempio, occorrerà creare una rotta che punti a questa azione, per poterla vedere in azione.

**Tip:** Questo articolo mostra come si interagisce con Doctrine dall'interno di un controllore, usando il metodo **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::getDoctrine'** del controllore. Tale metodo è una scorciatoia per ottenere il servizio `doctrine`. Si può interagire con Doctrine in altri contesti, iniettandolo come servizio. Vedere /book/service_container per maggiori informazioni sulla creazione di servizi.

Analizziamo questo esempio:

- **righe 10-13** In questa sezione, si istanzia e si lavora con l'oggetto `$product`, come qualsiasi altro normale oggetto PHP;
- **riga 14** Questa riga recupera l'oggetto *gestore di entità* di Doctrine, responsabile della gestione del processo di persistenza e del recupero di oggetti dalla base dati;
- **riga 16** Il metodo `persist()` dice a Doctrine di "gestire" l'oggetto `$product`. Questo non fa (ancora) eseguire una query sula base dati.
- **riga 17** Quando il metodo `flush()` è richiamato, Doctrine cerca tutti gli oggetti che sta gestendo, per vedere se hanno bisogno di essere persistiti sulla base dati. In questo esempio, l'oggetto `$product` non è stato ancora persistito, quindi il gestore di entità esegue una query `INSERT` e crea una riga nella tabella `product`.

**Note:** Di fatto, essendo Doctrine consapevole di tutte le proprie entità gestite, quando si chiama il metodo `flush()`, esso calcola un insieme globale di modifiche ed esegue le query nell'ordine corretto, usando dei prepared statement per migliorare le prestazioni. Per esempio, se si persiste un totale di 100 oggetti `Product` e quindi si richiama `flush()`, Doctrine eseguirà 100 query `INSERT` in un singolo oggetto prepared statement.

Quando si creano o aggiornano oggetti, il flusso è sempre lo stesso. Nella prossima sezione, si vedrà come Doctrine sia abbastanza intelligente da usare una query `UPDATE` se il record è già esistente nella base dati.

**Tip:** Doctrine fornisce una libreria che consente di caricare dati di test in un progetto (le cosiddette "fixture"). Per

informazioni, vedere la documentazione di "DoctrineFixturesBundle".

## Recuperare oggetti dalla base dati

Recuperare un oggetto dalla base dati è ancora più facile. Per esempio, supponiamo di aver configurato una rotta per mostrare uno specifico `Product`, in base al valore del suo `id`:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException(
            'Nessun prodotto trovato per l\'id '.$id
        );
    }

    // ... fare qualcosa, come passare l'oggetto $product a un template
}
```

**Tip:** Si può ottenere lo stesso risultato senza scrivere codice usando la scorciatoia `@ParamConverter`. Vedere la documentazione di FrameworkExtraBundle per maggiori dettagli.

Quando si cerca un particolare tipo di oggetto, si usa sempre quello che è noto come il suo "repository". Si può pensare a un repository come a una classe PHP il cui unico compito è quello di aiutare nel recuperare entità di una certa classe. Si può accedere all'oggetto repository per una classe entità tramite:

```
$repository = $this->getDoctrine()
    ->getRepository('AppBundle:Product');
```

**Note:** La stringa `AppBundle:Product` è una scorciatoia utilizzabile ovunque in Doctrine al posto del nome intero della classe dell'entità (cioè `AppBundle\Entity\Product`). Questo funzionerà finché le entità rimarranno sotto lo spazio dei nomi `Entity` del bundle.

Una volta ottenuto il repository, si avrà accesso a tanti metodi utili:

```
// cerca per chiave primaria (di solito "id")
$product = $repository->find($id);

// nomi di metodi dinamici per cercare in base al valore di una colonna
$product = $repository->findOneById($id);
$product = $repository->findOneByName('pippo');

// trova *tutti* i prodotti
$products = $repository->findAll();

// trova un gruppo di prodotti in base a un valore arbitrario di una colonna
$products = $repository->findByPrice(19.99);
```

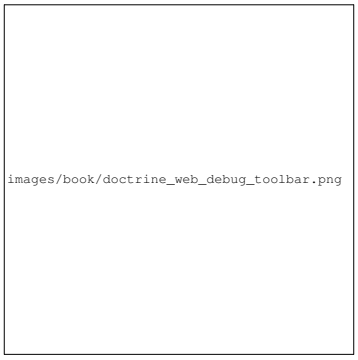
**Note:** Si possono ovviamente fare anche query complesse, su cui si può avere maggiori informazioni nella sezione *Cercare gli oggetti*.

Si possono anche usare gli utili metodi `findBy` e `findOneBy` per recuperare facilmente oggetti in base a condizioni multiple:

```
// cerca un prodotto in base a nome e prezzo
$product = $repository->findOneBy(
    array('name' => 'pippo', 'price' => 19.99)
);

// cerca tutti i prodotti in base al nome, ordinati per prezzo
$product = $repository->findBy(
    array('name' => 'pippo'),
    array('price' => 'ASC')
);
```

**Tip:** Quando si rende una pagina, si può vedere il numero di query eseguite nell'angolo inferiore destro della barra di debug del web.




Cliccando sull'icona, si aprirà il profilatore, che mostrerà il numero esatto di query eseguite.

L'icona diventa gialla se ci sono più di 50 query nella pagina. Questo potrebbe indicare che qualcosa non va.

## Aggiornare un oggetto

Una volta che Doctrine ha recuperato un oggetto, il suo aggiornamento è facile. Supponiamo di avere una rotta che mappi un id di prodotto a un'azione di aggiornamento in un controllore:

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $product = $em->getRepository('AppBundle:Product')->find($id);

    if (!$product) {
        throw $this->createNotFoundException(
            'Nessun prodotto trovato per l\'id '.$id
        );
    }

    $product->setName('Nome del nuovo prodotto!');
```

```
    $em->flush();

    return $this->redirect($this->generateUrl('homepage'));
}
```

L'aggiornamento di un oggetto si svolge in tre passi:

1. recuperare l'oggetto da Doctrine;
2. modificare l'oggetto;
3. richiamare `flush()` sul gestore di entità

Si noti che non è necessario richiamare `$em->persist($product)`. Ricordiamo che questo metodo dice semplicemente a Doctrine di gestire o "osservare" l'oggetto `$product`. In questo caso, poiché l'oggetto `$product` è stato recuperato da Doctrine, è già gestito.

### Cancellare un oggetto

La cancellazione di un oggetto è molto simile, ma richiede una chiamata al metodo `remove()` del gestore delle entità:

```
$em->remove($product);
$em->flush();
```

Come ci si potrebbe aspettare, il metodo `remove()` rende noto a Doctrine che si vorrebbe rimuovere la data entità dalla base dati. Tuttavia, la query `DELETE` non viene realmente eseguita finché non si richiama il metodo `flush()`.

## Cercare gli oggetti

Abbiamo già visto come l'oggetto repository consenta di eseguire query di base senza alcuno sforzo:

```
$repository->find($id);

$repository->findOneByName('Pippo');
```

Ovviamente, Doctrine consente anche di scrivere query più complesse, usando Doctrine Query Language (DQL). DQL è simile a SQL, tranne per il fatto che bisognerebbe immaginare di stare cercando uno o più oggetti di una classe entità (p.e. `Product`) e non le righe di una tabella (p.e. `product`).

Durante una ricerca in Doctrine, si hanno due opzioni: scrivere direttamente query Doctrine, oppure usare il Query Builder di Doctrine.

### Cercare oggetti con DQL

Si immagini di voler cercare dei prodotti, ma solo quelli che costino più di `19.99`, ordinati dal più economico al più caro. Si può usare `QueryBuilder` di Doctrine, come segue:

```
$repository = $this->getDoctrine()
    ->getRepository('AppBundle:Product');

$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
```

```
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
// per ottenere un singolo risultato:
// $product = $query->setMaxResults(1)->getOneOrNullResult();
```

L'oggetto `QueryBuilder` contiene tutti i metodi necessari per costruire una query. Richiamando il metodo `getQuery()`, `QueryBuilder` restituisce un oggetto `Query`, che può essere usato per ottenere il risultato della query.

---

**Tip:** Prendere nota del metodo `setParameter()`. Interagendo con Doctrine, è sempre una buona idea impostare valori esterni tramite "segnaposto" (`:price` nell'esempio appena visto), per prevenire attacchi di tipo SQL injection.

---

Il metodo `getResult()` restituisce un array di risultati. Se si cerca un solo oggetto, si può usare invece il metodo `getSingleResult()` (che lancia un'eccezione se non ci sono risultati) o `getOneOrNullResult()`:

```
$product = $query->getOneOrNullResult();
```

Per maggiori informazioni su `QueryBuilder`, consultare la documetazione Query Builder di Doctrine.

## Cercare oggetti usando DQL

Invece di usare `QueryBuilder`, is possono scrivere query direttamente, usando DQL:

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p
    FROM AppBundle:Product p
    WHERE p.price > :price
    ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
// per ottenere un singolo risultato:
// $product = $query->setMaxResults(1)->getOneOrNullResult();
```

Se ci si trova a proprio agio con SQL, DQL dovrebbe sembrare molto naturale. La maggiore differenza è che occorre pensare in termini di "oggetti" invece che di righe di basi dati. Per questa ragione, si cerca *da* `AcmeStoreBundle:Product` e poi si usa `p` come suo alias (che è quello che stato fatto nella sezione precedente).

La sintassi DQL è incredibilmente potente e consente di fare join tra entità (l'argomento *relazioni* sarà affrontato successivamente), raggruppare, ecc. Per maggiori informazioni, vedere la documentazione ufficiale di Doctrine Doctrine Query Language.

## Classi repository personalizzate

Nelle sezioni precedenti, si è iniziato costruendo e usando query più complesse da dentro un controllore. Per isolare, testare e riusare queste query, è una buona idea creare una classe repository personalizzata per la propria entità e aggiungere metodi, come la propria logica di query, al suo interno.

Per farlo, aggiungere il nome della classe del repository alla propria definizione di mappatura.

Doctrine può generare la classe repository per noi, eseguendo lo stesso comando usato precedentemente per generare i metodi getter e setter mancanti:

```
$ php app/console doctrine:generate:entities AppBundle
```

Quindi, aggiungere un nuovo metodo, chiamato `findAllOrderedByName()`, alla classe repository appena generata. Questo metodo cercherà tutte le entità `Product`, ordinate alfabeticamente.

```
// src/AppBundle/Entity/ProductRepository.php
namespace AppBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM AppBundle:Product p ORDER BY p.name ASC'
            )
            ->getResult();
    }
}
```

**Tip:** Si può accedere al gestore di entità tramite `$this->getEntityManager()` da dentro il repository.

Si può usare il metodo appena creato proprio come i metodi predefiniti del repository:

```
$em = $this->getDoctrine()->getManager();
$products = $em->getRepository('AppBundle:Product')
    ->findAllOrderedByName();
```

**Note:** Quando si usa una classe repository personalizzata, si ha ancora accesso ai metodi predefiniti di ricerca, come `find()` e `findAll()`.

## Relazioni e associazioni tra entità

Supponiamo che i prodotti nella propria applicazione appartengano tutti a una "categoria". In questo caso, occorrerà un oggetto `Category` e un modo per per mettere in relazione un oggetto `Product` con un oggetto `Category`. Iniziamo creando l'entità `Category`. Sapendo che probabilmente occorrerà persistere la classe tramite Doctrine, lasciamo che sia Doctrine stesso a creare la classe.

```
$ php app/console doctrine:generate:entity \
    --entity="AppBundle:Category" \
    --fields="name:string(255)"
```

Questo task genera l'entità `Category`, con un campo `id`, un campo `name` e le relative funzioni getter e setter.

## Metadati di mappatura delle relazioni

Per correlare le entità `Category` e `Product`, iniziamo creando una proprietà `products` nella classe `Category`:

Primo, poiché un oggetto `Category` sarà collegato a diversi oggetti `Product`, va aggiunta una proprietà array `products`, per contenere questi oggetti `Product`. Di nuovo, non va fatto perché Doctrine ne abbia bisogno, ma perché ha senso nell'applicazione che ogni `Category` contenga un array di oggetti `Product`.

---

**Note:** Il codice nel metodo `__construct()` è importante, perché Doctrine esige che la proprietà `$products` sia un oggetto `ArrayCollection`. Questo oggetto sembra e si comporta quasi *esattamente* come un array, ma ha un po' di flessibilità in più. Se non sembra confortevole, niente paura. Si immagini solamente che sia un `array`.

---

---

**Tip:** Il valore `targetEntity`, usato in precedenza sul decoratore, può riferirsi a qualsiasi entità con uno spazio dei nomi valido, non solo a entità definite nella stessa classe. Per riferirsi a entità definite in classi diverse, inserire uno spazio dei nomi completo come `targetEntity`.

---

Poi, poiché ogni classe `Product` può essere in relazione esattamente con un oggetto `Category`, si deve aggiungere una proprietà `$category` alla classe `Product`:

Infine, dopo aver aggiunto una nuova proprietà sia alla classe `Category` che a quella `Product`, dire a Doctrine di generare i metodi mancanti getter e setter:

```
$ php app/console doctrine:generate:entities AppBundle
```

Ignoriamo per un momento i metadati di Doctrine. Abbiamo ora due classi, `Category` e `Product`, con una relazione naturale uno-a-molti. La classe `Category` contiene un array di oggetti `Product` e l'oggetto `Product` può contenere un oggetto `Category`. In altre parole, la classe è stata costruita in un modo che ha senso per le proprie necessità. Il fatto che i dati necessitino di essere persistiti su una base dati è sempre secondario.

Diamo ora uno sguardo ai metadati nella proprietà `$category` della classe `Product`. Qui le informazioni dicono a Doctrine che la classe correlata è `Category` e che dovrebbe memorizzare il valore `id` della categoria in un campo `category_id` della tabella `product`. In altre parole, l'oggetto `Category` correlato sarà memorizzato nella proprietà `$category`, ma dietro le quinte Doctrine persisterà questa relazione memorizzando il valore dell'id della categoria in una colonna `category_id` della tabella `product`.


I metadati della proprietà `$products` dell'oggetto `Category` sono meno importanti e dicono semplicemente a Doctrine di cercare la proprietà `Product.category` per sapere come mappare la relazione.

Prima di continuare, accertarsi di dire a Doctrine di aggiungere la nuova tabella `category` la nuova colonna `product.category_id` e la nuova chiave esterna:

```
$ php app/console doctrine:schema:update --force
```

---

**Note:** Questo task andrebbe usato solo durante lo sviluppo. Per un metodo più robusto di aggiornamento sistematico della propria base dati di produzione, vedere le migrazioni.

---

## Salvare le entità correlate

Vediamo ora il codice in azione. Immaginiamo di essere dentro un controllore:

```
// ...

use AppBundle\Entity\Category;
use AppBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Prodotti principali');

        $product = new Product();
        $product->setName('Pippo');
        $product->setPrice(19.99);
        $product->setDescription('Lorem ipsum dolor');
        // correlare questo prodotto alla categoria
        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Creati prodotto con id: '.$product->getId()
            .' e categoria con id: '.$category->getId()
        );
    }
}
```

Una riga è stata aggiunta alle tabelle `category` e `product`. La colonna `product.category_id` del nuovo prodotto è impostata allo stesso valore di `id` della nuova categoria. Doctrine gestisce la persistenza di tale relazione per noi.

## Recuperare gli oggetti correlati

Quando occorre recuperare gli oggetti correlati, il flusso è del tutto simile a quello precedente. Recuperare prima un oggetto `$product` e poi accedere alla sua `Category` correlata:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

In questo esempio, prima di cerca un oggetto `Product` in base al suo `id`. Questo implica una

query *solo* per i dati del prodotto e idrata l'oggetto `$product` con tali dati. Poi, quando si richiama `$product->getCategory()->getName()`, Doctrine effettua una seconda query, per trovare la `Category` correlata con il `Product`. Prepara l'oggetto `$category` e lo restituisce.


Quello che è importante è il fatto che si ha facile accesso al prodotto correlato con la categoria, ma i dati della categoria non sono recuperati finché la categoria non viene richiesta (processo noto come "lazy load").

Si può anche cercare nella direzione opposta:

```
public function showProductsAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AppBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

In questo caso succedono le stesse cose: prima si cerca un singolo oggetto `Category`, poi Doctrine esegue una seconda query per recuperare l'oggetto `Product` correlato, ma solo quando/se richiesto (cioè al richiamo di `->getProducts()`). La variabile `$products` è un array di tutti gli oggetti `Product` correlati con il dato oggetto `Category` tramite il loro valore `category_id`.

**Relazioni e classi proxy**

Questo "lazy load" è possibile perché, quando necessario, Doctrine restituisce un oggetto "proxy" al posto del vero oggetto. Guardiamo di nuovo l'esempio precedente:

```
$product = $this->getDoctrine()
    ->getRepository('AppBundle:Product')
    ->find($id);

$category = $product->getCategory();

// prints "Proxies\AppBundleEntityCategoryProxy"
echo get_class($category);
```

Questo oggetto proxy estende il vero oggetto `Category` e sembra e si comporta esattamente nello stesso modo. La differenza è che, usando un oggetto proxy, Doctrine può rimandare la query per i dati effettivi di `Category` fino a che non sia effettivamente necessario (cioè fino alla chiamata di `$category->getName()`).

Le classy proxy sono generate da Doctrine e memorizzate in cache. Sebbene probabilmente non si noterà mai che l'oggetto `$category` sia in realtà un oggetto proxy, è importante tenerlo a mente.

Nella prossima sezione, quando si recuperano i dati di prodotto e categoria in una volta sola (tramite una *join*), Doctrine restituirà il *vero* oggetto `Category`, poiché non serve alcun lazy load.

## Join di record correlati

Negli esempi precedenti, sono state eseguite due query: una per l'oggetto originale (p.e. una `Category`) e una per gli oggetti correlati (p.e. gli oggetti `Product`).

**Tip:** Si ricordi che è possibile vedere tutte le query eseguite durante una richiesta, tramite la barra di debug del web.

Ovviamente, se si sa in anticipo di aver bisogno di accedere a entrambi gli oggetti, si può evitare la seconda query, usando una join nella query originale. Aggiungere il seguente metodo alla classe `ProductRepository`:

```
// src/AppBundle/Entity/ProductRepository.php
public function findOneByIdJoinedToCategory($id)
{
    $query = $this->getEntityManager()
        ->createQuery(
            'SELECT p, c FROM AppBundle:Product p
            JOIN p.category c
            WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $query->getSingleResult();
    } catch (\Doctrine\ORM\NoResultException $e) {
        return null;
    }
}
```

Ora si può usare questo metodo nel controllore, per cercare un oggetto `Product` e la relativa `Category` con una sola query:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

## Ulteriori informazioni sulle associazioni

Questa sezione è stata un'introduzione a un tipo comune di relazione tra entità, la relazione uno-a-molti. Per dettagli ed esempi più avanzati su come usare altri tipi di relazioni (p.e. uno-a-uno, molti-a-molti), vedere la Association Mapping Documentation di Doctrine.

**Note:** Se si usano le annotazioni, occorrerà aggiungere a tutte le annotazioni il prefisso `ORM\` (p.e. `ORM\OneToMany`), che non si trova nella documentazione di Doctrine. Occorrerà anche includere l'istruzione `use Doctrine\ORM\Mapping as ORM;`, che *importa* il prefisso delle annotazioni `ORM`.

## Configurazione

Doctrine è altamente configurabile, sebbene probabilmente non si avrà nemmeno bisogno di preoccuparsi di gran parte delle sue opzioni. Per saperne di più sulla configurazione di Doctrine, vedere la sezione Doctrine del manuale di riferimento.

## Callback del ciclo di vita

A volte, occorre eseguire un'azione subito prima o subito dopo che un entità sia inserita, aggiornata o cancellata. Questi tipi di azioni sono noti come callback del "ciclo di vita", perché sono metodi callback che occorre eseguire durante i diversi stadi del ciclo di vita di un'entità (p.e. l'entità è inserita, aggiornata, cancellata, eccetera).

Se si usano le annotazioni per i metadati, iniziare abilitando i callback del ciclo di vita. Questo non è necessario se si usa YAML o XML per la mappatura:

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Si può ora dire a Doctrine di eseguire un metodo su uno degli eventi disponibili del ciclo di vita. Per esempio, supponiamo di voler impostare una colonna di data `createdAt` alla data attuale, solo quando l'entità è persistita la prima volta (cioè è inserita):

**Note:** L'esempio precedente presume che sia stata creata e mappata una proprietà `createdAt` (non mostrata qui).

Ora, appena prima che l'entità sia persistita per la prima volta, Doctrine richiamerà automaticamente questo metodo e il campo `created` sarà valorizzato con la data attuale.

Ci sono molti altri eventi del ciclo di vita, a cui ci si può agganciare. Per maggiori informazioni, vedere la documentazione di Doctrine Lifecycle Events documentation.

**Callback del ciclo di vita e ascoltatori di eventi**

Si noti che il metodo `setCreatedValue()` non riceve parametri. Questo è sempre il caso di callback del ciclo di vita ed è intenzionale: i callback del ciclo di vita dovrebbero essere metodi semplici, riguardanti la trasformazione interna di dati nell'entità (p.e. impostare un campo di creazione/aggiornamento, generare un valore per uno slug).

Se occorre un lavoro più pesante, come eseguire un log o inviare una email, si dovrebbe registrare una classe esterna come ascoltatore di eventi e darle accesso a qualsiasi risorsa necessaria. Per maggiori informazioni, vedere /cookbook/doctrine/event_listeners_subscribers.

## Riferimento sui tipi di campo di Doctrine

Doctrine ha un gran numero di tipi di campo a disposizione. Ognuno di questi mappa un tipo di dato PHP su un tipo specifico di colonna in qualsiasi base dati si utilizzi. Per ciascun tipo di campo, si può configurare ulteriormente

`Column`, impostando le opzioni `length`, `nullable`, `name` e altre ancora. Per una lista completa di tipi e per maggiori informazioni vedere la documentazione di Doctrine Mapping Types documentation.

## Riepilogo

Con Doctrine, ci si può concentrare sui propri oggetti e su come siano utili nella propria applicazione e preoccuparsi della persistenza su base dati in un secondo momento. Questo perché Doctrine consente di usare qualsiasi oggetto PHP per tenere i propri dati e si appoggia su metadati di mappatura per mappare i dati di un oggetto su una particolare tabella di base dati.

Sebbene Doctrine giri intorno a un semplice concetto, è incredibilmente potente, consentendo di creare query complesse e sottoscrivere eventi che consentono di intraprendere diverse azioni, mentre gli oggetti viaggiano lungo il loro ciclo di vita della persistenza.

### Saperne di più

Per maggiori informazioni su Doctrine, vedere la sezione *Doctrine* del ricettario, che include i seguenti articoli:

- /cookbook/doctrine/common_extensions
- /cookbook/doctrine/console
- DoctrineFixturesBundle
- DoctrineMongoDBBundle

CHAPTER 11

# Basi di dati e Propel

Ammettiamolo, uno dei compiti più comuni e impegnativi per qualsiasi applicazione implica la persistenza e la lettura di informazioni da una base dati. Symfony non è integrato nativamente con Propel, ma l'integrazione è alquanto semplice. Per iniziare, leggere Working With Symfony2.

## Un semplice esempio: un prodotto

In questa sezione, configureremo la nostra base dati, creeremo un oggetto `Product`, lo persisteremo nella base dati e lo recupereremo nuovamente.

### Configurare la base dati

Prima di iniziare, occorre configurare le informazioni sulla connessione alla base dati. Per convenzione, questa informazione solitamente è configurata in un file `app/config/parameters.yml`:

```
# app/config/parameters.yml
parameters:
    database_driver:   mysql
    database_host:     localhost
    database_name:     test_project
    database_user:     root
    database_password: password
    database_charset:  UTF8
```

I parametri definiti in `parameters.yml` possono essere inclusi nel file di configurazione (`config.yml`):

```
propel:
    dbal:
        driver:   "%database_driver%"
        user:     "%database_user%"
        password: "%database_password%"
        dsn:      "%database_driver%:host=%database_host%;dbname=%database_name%;
↪charset=%database_charset%"
```

**Note:** La definizione della configurazione tramite `parameters.yml` è una best practice di Symfony, ma si può usare qualsiasi metodo si ritenga appropriato.

Ora che Propel ha informazioni sulla base dati, si può fare in modo che crei la base dati al posto nostro:

```
$ php app/console propel:database:create
```

**Note:** In questo esempio, si ha una sola connessione configurata, di nome `default`. Se si vogliono configurare più connessioni, leggere la sezione configurazione di PropelBundle.

## Creare una classe del modello

Nel mondo di Propel, le classi ActiveRecord sono note come **modelli**, perché le classi generate da Propel contengono della logica di business.

**Note:** Per chi ha già usato Symfony con Doctrine2, i **modelli** sono equivalenti alle **entità**.

Si supponga di costruire un'applicazione in cui occorre mostrare dei prodotti. Innanzitutto, creare un file `schema.xml` nella cartella `Resources/config` del proprio AppBundle:

```
<!-- src/AppBundle/Resources/config/schema.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<database
    name="default"
    namespace="AppBundle\Model"
    defaultIdMethod="native">

    <table name="product">
        <column
            name="id"
            type="integer"
            required="true"
            primaryKey="true"
            autoIncrement="true" />

        <column
            name="name"
            type="varchar"
            primaryString="true"
            size="100" />
        <column
            name="price"
            type="decimal" />

        <column
            name="description"
            type="longvarchar" />
    </table>
</database>
```

## Costruire il modello

Dopo aver creato `schema.xml`, generare il modello, eseguendo:

```
$ php app/console propel:model:build
```

Questo comando genera ogni classe del modello, per sviluppare rapidamente un'applicazione, nella cartella `Model/` di AppBundle.

## Creare schema e tabelle della base dati

Ora si dispone di una classe `Product` e di tutto il necessario per poterla persistere. Ovviamente, non si ha ancora una corrispondente tabella `product` nella base dati. Per fortuna, Propel può creare automaticamente tutte le tabelle della base dati, per ciascun modello dell'applicazione. Per farlo, eseguire:

```
$ php app/console propel:sql:build
$ php app/console propel:sql:insert --force
```

La base dati ora ha una tabella `product`, con colonne corrispondenti allo schema creato in precedenza.

**Tip:** Si possono eseguire gli ultimi tre comandi in uno, usando il seguente comando:

```
$ php app/console propel:build --insert-sql
```

## Persistere oggetti nella base dati

Ora che si ha un oggetto `Product` e una tabella `product` corrispondente, si è pronti per persistere nella base dati. Da dentro un controllore, è molto facile. Aggiungere il seguente metodo a `ProductController` del bundle:

```
// src/AppBundle/Controller/ProductController.php

// ...
use AppBundle\Model\Product;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends Controller
{
    public function createAction()
    {
        $product = new Product();
        $product->setName('Un nome');
        $product->setPrice(19.99);
        $product->setDescription('Lorem ipsum dolor');

        $product->save();

        return new Response('Creato prodotto con id '.$product->getId());
    }
}
```

In questo pezzo di codice, è stato istanziato e usato un oggetto `$product`. Richiamando il suo metodo `save()`, lo si persiste nella base dati. Non occorre usare altri servizi, l'oggetto sa da solo come persistersi.

**Note:** Se si segue il codice di questo esempio, occorre creare una *rotta* che punti a questa azione.

## Recuperare oggetti dalla base dati

Recuperare oggetti dalla base dati è anche più semplice. Per esempio, si supponga di aver configurato una rotta per mostrare uno specifico `Product`, in base al valore del suo `id`:

```
// src/AppBundle/Controller/ProductController.php

// ...
use AppBundle\Model\ProductQuery;

class ProductController extends Controller
{
    // ...

    public function showAction($id)
    {
        $product = ProductQuery::create()->findPk($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'Nessun prodotto trovato con id '.$id
            );
        }

        // ... fare qualcosa, come passare l'oggetto $product a un template
    }
}
```

## Aggiornare un oggetto

Una volta recuperato un oggetto con Propel, aggiornarlo è facile. Si supponga di avere una rotta che mappi l'id di un prodotto all'azione di aggiornamento di un controllore:

```
// src/AppBundle/Controller/ProductController.php

// ...
use AppBundle\Model\ProductQuery;

class ProductController extends Controller
{
    // ...

    public function updateAction($id)
    {
        $product = ProductQuery::create()->findPk($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'Nessun prodotto trovato con id '.$id
            );
        }
```

```
        $product->setName('Nuovo nome del prodotto!');
        $product->save();

        return $this->redirect($this->generateUrl('homepage'));
    }
}
```

L'aggiornamento di un oggetto si esegue in tre passi:

1. recupero dell'oggetto da Propel;
2. modifica dell'oggetto;
3. salvataggio.

### Cancellare un oggetto

La cancellazione di un oggetto è molto simile, ma richiede una chiamata al metodo `delete()` dell'oggetto:

```
$product->delete();
```

## Cercare gli oggetti

Propel fornisce delle classi `Query`, per eseguire query, semplici o complesse, senza sforzo:

```
use AppBundle\Model\ProductQuery;
// ...

ProductQuery::create()->findPk($id);

ProductQuery::create()
    ->filterByName('Pippo')
    ->findOne();
```

Si immagini di voler cercare prodotti che costino più di 19.99, ordinati dal più economico al più costoso. Da dentro un controllore, fare come segue:

```
use AppBundle\Model\ProductQuery;
// ...

$products = ProductQuery::create()
    ->filterByPrice(array('min' => 19.99))
    ->orderByPrice()
    ->find();
```

In una sola riga, si ottengono i prodotti cercati in modo orientato agli oggetti. Non serve perdere tempo con SQL o simili, Symfony offre una programmazione completamente orientata agli oggetti e Propel rispetta la stessa filosofia, fornendo un incredibile livello di astrazione.

Se si vogliono riutilizzare delle query, si possono aggiungere i propri metodi alla classe `ProductQuery`:

```
// src/AppBundle/Model/ProductQuery.php

// ...
```

```
class ProductQuery extends BaseProductQuery
{
    public function filterByExpensivePrice()
    {
        return $this->filterByPrice(array(
            'min' => 1000,
        ));
    }
}
```

Ma si noti che Propel genera diversi metodi per noi e un semplice `findAllOrderedByName()` può essere scritto senza sforzi:

```
use AppBundle\Model\ProductQuery;
// ...

ProductQuery::create()
    ->orderByName()
    ->find();
```

## Relazioni/associazioni

Si supponga che tutti i prodotti dell'applicazione appartengano a una delle categorie. In questo caso, occorrerà un oggetto `Category` e un modo per correlare un oggetto `Product` a un oggetto `Category`.

Si inizi aggiungendo la definizione di `category` al file `schema.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<database
    name="default"
    namespace="AppBundle\Model"
    defaultIdMethod="native">

    <table name="product">
        <column
            name="id"
            type="integer"
            required="true"
            primaryKey="true"
            autoIncrement="true" />

        <column
            name="name"
            type="varchar"
            primaryString="true"
            size="100" />

        <column
            name="price"
            type="decimal" />

        <column
            name="description"
            type="longvarchar" />

        <column
```

```
            name="category_id"
            type="integer" />

        <foreign-key foreignTable="category">
            <reference local="category_id" foreign="id" />
        </foreign-key>
    </table>

    <table name="category">
        <column
            name="id"
            type="integer"
            required="true"
            primaryKey="true"
            autoIncrement="true" />

        <column
            name="name"
            type="varchar"
            primaryString="true"
            size="100" />
   </table>
</database>
```

Creare le classi:

```
$ php app/console propel:model:build
```

Ipotizziamo di avere già dei prodotti nella base dati e che non si voglia perderli. Grazie alle migrazioni, Propel sarà in grado di aggiornare la base dati, senza perdere alcun dato esistente.

```
$ php app/console propel:migration:generate-diff
$ php app/console propel:migration:migrate
```

La base dati è stata aggiornata, si può continuare nella scrittura dell'applicazione.

## Salvare oggetti correlati

Vediamo ora un po' di codice in azione. Immaginiamo di essere dentro un controllore:

```
// src/AppBundle/Controller/ProductController.php

// ...
use AppBundle\Model\Category;
use AppBundle\Model\Product;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Prodotti principali');

        $product = new Product();
        $product->setName('Pippo');
```

```
        $product->setPrice(19.99);
        // mette in relazione questo prodotto alla categoria
        $product->setCategory($category);

        // salva tutto
        $product->save();

        return new Response(
            'Creato prodotto con id: '.$product->getId().' e categoria con id: '.
↪$category->getId()
        );
    }
}
```

Una singola riga è stata aggiunta alle tabelle `category` e `product`. La colonna `product.category_id` del nuovo prodotto è stata impostata all'id della nuova categoria. Propel gestisce la persistenza di questa relazione al posto nostro.

### Recuperare oggetti correlati

Quando serve recuperare oggetti correlati, il flusso di lavoro assomiglia del tutto al precedente. Prima, recuperare un oggetto `$product` e quindi accedere alla `Category` relativa:

```
// src/AppBundle/Controller/ProductController.php

// ...
use AppBundle\Model\ProductQuery;

class ProductController extends Controller
{
    public function showAction($id)
    {
        $product = ProductQuery::create()
            ->joinWithCategory()
            ->findPk($id);

        $categoryName = $product->getCategory()->getName();

        // ...
    }
}
```

Si noti che, nell'esempio qui sopra, è stata eseguita una sola query.

### Maggior informazioni sulle associazioni

Si possono trovare maggiori informazioni sulle relazioni, leggendo il capitolo dedicato alle relazioni.

## Callback del ciclo di vita

A volte, occorre eseguire un'azione appena prima (o appena dopo) che l'oggetto sia inserito, aggiornato o cancellato. Questi tipi di azioni sono noti come "callback del ciclo di vita" oppure come "agganci", perché sono metodi callback

che occorre eseguire durante i diversi stadi del ciclo di vita di un oggetto (p.e. quando l'oggetto viene inserito, aggiornato, cancellato, eccetera).

Per aggiungere un aggancio, basta aggiungere un nuovo metodo alla classe:

```
// src/AppBundle/Model/Product.php

// ...
class Product extends BaseProduct
{
    public function preInsert(\PropelPDO $con = null)
    {
        // fare qualcosa prima che l'oggetto sia inserito
    }
}
```

Propel fornisce i seguenti agganci:

**`preInsert()`** codice eseguito prima dell'inserimento di un nuovo oggetto

**`postInsert()`** codice eseguito dopo l'inserimento di un nuovo oggetto

**`preUpdate()`** codice eseguito prima dell'aggiornamento di un oggetto esistente

**`postUpdate()`** codice eseguito dopo l'aggiornamento di un oggetto esistente

**`preSave()`** codice eseguito prima di salvare un oggetto (nuovo o esistente)

**`postSave()`** codice eseguito dopo il salvataggio di un oggetto (nuovo o esistente)

**`preDelete()`** codice eseguito prima di cancellare un oggetto

**`postDelete()`** codice eseguito dopo la cancellazione di un oggetto

## Comportamenti

Tutti i comportamenti distribuiti con Propel funzionano in Symfony. Per ottenere maggiori informazioni su come usare i comportamenti di Propel, fare riferimento alla sezione sui behavior.

## Comandi

Leggere la sezione dedicata ai comandi Propel in Symfony2.

CHAPTER 12

# Test

Ogni volta che si scrive una nuova riga di codice, si aggiungono potenzialmente nuovi bug. Per costruire applicazioni migliori e più affidabili, si dovrebbe sempre testare il codice, usando sia i test funzionali che quelli unitari.

## Il framework dei test PHPUnit

Symfony si integra con una libreria indipendente, chiamata PHPUnit, per fornire un ricco framework per i test. Questo capitolo non approfondisce PHPUnit stesso, che ha comunque un'eccellente documentazione.

**Note:** Symfony funziona con PHPUnit 3.5.11 o successivi, ma per testare il codice del nucleo di Symfony occorre la versione 3.6.4.

Ogni test, sia esso unitario o funzionale, è una classe PHP, che dovrebbe trovarsi in una sotto-cartella *Tests/* del bundle. Seguendo questa regola, si possono eseguire tutti i test di un'applicazione con il seguente comando:

```
# specifica la cartella di configurazione nella linea di comando
$ phpunit -c app/
```

L'opzione `-c` dice a PHPUnit di cercare nella cartella `app/` un file di configurazione. Chi fosse curioso di conoscere le opzioni di PHPUnit, può dare uno sguardo al file `app/phpunit.xml.dist`.

**Tip:** Si può generare la copertura del codice, con le opzioni `--coverage-*`, vedere le informazioni di aiuto mostrate usando `--help`.

## Test unitari

Un test unitario è solitamente un test di una specifica classe PHP, chiamata *unità*. Se si vuole testare il comportamento generale della propria applicazione, vedere la sezione dei *Test funzionali*.

La scrittura di test unitari in Symfony non è diversa dalla scrittura standard di test unitari in PHPUnit. Si supponga, per esempio, di avere una classe *incredibilmente* semplice, chiamata `Calculator`, nella cartella `Utility/` del bundle:

```
// src/AppBundle/Util/Calculator.php
namespace AppBundle\Util;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

Per testarla, creare un file `CalculatorTest` nella cartella `Tests/Util` del bundle:

```
// src/AppBundle/Tests/Util/CalculatorTest.php
namespace AppBundle\Tests\Util;

use AppBundle\Util\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // asserisce che il calcolatore aggiunga correttamente i numeri!
        $this->assertEquals(42, $result);
    }
}
```

**Note:** Per convenzione, si raccomanda di replicare la struttura di cartella di un bundle nella sua sotto-cartella `Tests/`. Quindi, se si testa una classe nella cartella `Util/` del bundle, mettere il test nella cartella `Tests/Util/`.

Proprio come per l'applicazione reale, l'autoloading è abilitato automaticamente tramite il file `bootstrap.php.cache` (come configurato in modo predefinito nel file `phpunit.xml.dist`).

Anche eseguire i test per un dato file o una data cartella è molto facile:

```
# eseguire tutti i test dell'applicazione
$ phpunit -c app

# eseguire i  test per la classe Calculator
$ phpunit -c app src/AppBundle/Tests/Util

# eseguire i test per la classe Calculator
$ phpunit -c app src/AppBundle/Tests/Util/CalculatorTest.php

# eseguire tutti i test per l'intero bundle
$ phpunit -c app src/AppBundle/
```

# Test funzionali

I test funzionali verificano l'integrazione dei diversi livelli di un'applicazione (dalle rotte alle viste). Non differiscono dai test unitari per quello che riguarda PHPUnit, ma hanno un flusso di lavoro molto specifico:

- Fare una richiesta;
- Testare la risposta;
- Cliccare su un collegamento o inviare un form;
- Testare la risposta;
- Ripetere.

## Un primo test funzionale

I test funzionali sono semplici file PHP, che tipicamente risiedono nella cartella `Tests/Controller` del bundle. Se si vogliono testare le pagine gestite dalla classe `PostController`, si inizi creando un file `PostControllerTest.php`, che estende una classe speciale `WebTestCase`.

Per esempio, un test può essere fatto in questo modo:

```
// src/AppBundle/Tests/Controller/PostControllerTest.php
namespace AppBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class PostControllerTest extends WebTestCase
{
    public function testShowPost()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/post/hello-world');

        $this->assertGreaterThan(
            0,
            $crawler->filter('html:contains("Hello World")')->count()
        );
    }
}
```

**Tip:** Per eseguire i test funzionali, la classe `WebTestCase` inizializza il kernel dell'applicazione. Nella maggior parte dei casi, questo avviene in modo automatico. Tuttavia, se il proprio kernel si trova in una cartella non standard, occorre modificare il file `phpunit.xml.dist` e impostare nella variabile d'ambiente `KERNEL_DIR` la cartella del kernel:

```
<?xml version="1.0" charset="utf-8" ?>
<phpunit>
    <php>
        <server name="KERNEL_DIR" value="/percorso/della/applicazione/" />
    </php>
    <!-- ... -->
</phpunit>
```

Il metodo `createClient()` restituisce un client, che è come un browser da usare per visitare un sito:

```
$crawler = $client->request('GET', '/post/hello-world');
```

Il metodo `request()` (vedere *di più sul metodo della richiesta*) restituisce un oggetto `Symfony\Component\DomCrawler\Crawler`, che può essere usato per selezionare elementi nella risposta, per cliccare su collegamenti e per inviare form.

**Tip:** Il crawler può essere usato solo se il contenuto della risposta è un documento XML o HTML. Per altri tipi di contenuto, richiamare `$client->getResponse()->getContent()`.

Cliccare su un collegamento, selezionandolo prima con il Crawler, usando o un'espressione XPath o un selettore CSS, quindi usando il Client per cliccarlo. Per esempio:

```
$link = $crawler
    ->filter('a:contains("Greet")') // trova i collegamenti con testo "Greet"
    ->eq(1) // seleziona il secondo collegamento della lista
    ->link() // e lo clicca
;

$crawler = $client->click($link);
```

Inviare un form è molto simile: selezionare il bottone di un form, eventualmente sovrascrivere alcuni valori del form e inviare il form corrispondente:

```
$form = $crawler->selectButton('submit')->form();

// impostare alcuni valori
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Bella per te!';

// inviare il form
$crawler = $client->submit($form);
```

**Tip:** Il form può anche gestire caricamenti di file e contiene metodi utili per riempire diversi tipi di campi (p.e. `select()` e `tick()`). Per maggiori dettagli, vedere la sezione *Form* più avanti.

Ora che si è in grado di navigare facilmente nell'applicazione, usare le asserzioni per testare che faccia effettivamente quello che ci si aspetta. Usare il Crawler per fare asserzioni sul DOM:

```
// Asserisce che la risposta corrisponda a un dato selettore CSS.
$this->assertGreaterThan(0, $crawler->filter('h1')->count());
```

Oppure, testare direttamente il contenuto della risposta, se si vuole solo asserire che il contenuto debba contenere del testo o se la risposta non è un documento XML/HTML:

```
$this->assertContains(
    'Hello World',
    $client->getResponse()->getContent()
);
```

**Asserzioni utili**

Per iniziare più rapidamente, ecco una lista delle asserzioni più utili e comuni:

```
use Symfony\Component\HttpFoundation\Response;

// ...

// Asserire che ci sia almeno un tag h2
// con la classe "subtitle"
$this->assertGreaterThan(
    0,
    $crawler->filter('h2.subtitle')->count()
);

// Asserire che ci sono esattamente 4 tag h2 nella pagina
$this->assertCount(4, $crawler->filter('h2'));

// Asserire che il "Content-Type" header sia "application/json"
$this->assertTrue(
    $client->getResponse()->headers->contains(
        'Content-Type',
        'application/json'
    )
);

// Asserire che la risposta contenga una stringa
$this->assertContains('foo', $client->getResponse()->getContent());
// Asserire che la risposta corrisponda a un'espressione regolare.
$this->assertRegExp('/pippo(pluto)?/', $client->getResponse()->getContent());

// Asserire che il codice di stato della risposta sia 2xx
$this->assertTrue($client->getResponse()->isSuccessful());
// Asserire che il codice di stato della risposta sia 404
$this->assertTrue($client->getResponse()->isNotFound());
// Asserire uno specifico codice di stato 200
$this->assertEquals(
    200, // o Symfony\Component\HttpFoundation\Response::HTTP_OK
    $client->getResponse()->getStatusCode()
);

// Asserire che il codice di stato della risposta sia un rinvio a /demo/contact
$this->assertTrue(
    $client->getResponse()->isRedirect('/demo/contact')
);
// o verificare semplicemente che la risposta sia un rinvio
$this->assertTrue($client->getResponse()->isRedirect());
```

New in version 2.4: Il supporto per le costanti dei codici di stato HTTP è stato aggiunto in Symfony 2.4.

## Lavorare con il client dei test

Il client dei test emula un client HTTP, come un browser, ed effettua richieste all'applicazione Symfony:

```
$crawler = $client->request('GET', '/hello/Fabien');
```

Il metodo `request()` accetta come parametri il metodo HTTP e un URL e restituisce un'istanza di `Crawler`.

**Tip:** Inserire gli URL a mano è preferibile per i test funzionali. Se un test generasse URL usando le rotte di Symfony, non si accorgerebbe di eventuali modifiche agli URL dell'applicazione, che potrebbero aver impatto sugli utenti finali.

**Di più sul metodo `request()`:**

La firma completa del metodo `request()` è:

```
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

L'array `server` contiene i valori grezzi che ci si aspetta di trovare normalmente nell'array superglobale $_SERVER di PHP. Per esempio, per impostare gli header HTTP `Content-Type`, `Referer` e `X-Requested-With`, passare i seguenti (ricordare il prefisso `HTTP_` per gli header non standard):

```
$client->request(
    'GET',
    '/post/hello-world',
    array(),
    array(),
    array(
        'CONTENT_TYPE'          => 'application/json',
        'HTTP_REFERER'          => '/foo/bar',
        'HTTP_X-Requested-With' => 'XMLHttpRequest',
    )
);
```

Usare il crawler per cercare elementi del DOM nella risposta. Questi elementi possono poi essere usati per cliccare su collegamenti e inviare form:

```
$link = $crawler->selectLink('Vai da qualche parte...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validare')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

I metodi `click()` e `submit()` restituiscono entrambi un oggetto `Crawler`. Questi metodi sono il modo migliore per navigare un'applicazione, perché si occupano di diversi dettagli, come il metodo HTTP di un form e il fornire un'utile API per caricare file.

**Tip:** Gli oggetti `Link` e `Form` nel crawler saranno approfonditi nella sezione *Crawler*, più avanti.

Il metodo `request()` può anche essere usato per simulare direttamente l'invio di form o per eseguire richieste più complesse:

```
// Invio diretto di form
$client->request('POST', '/submit', array('name' => 'Fabien'));

// Invio di una string JSON grezza nel corpo della richiesta
$client->request(
    'POST',
    '/submit',
    array(),
    array(),
    array('CONTENT_TYPE' => 'application/json'),
    '{"name":"Fabien"}'
);

// Invio di form di con caricamento di file
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/percorso/di/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Eseguire richieste DELETE e passare header HTTP
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

Infine, ma non meno importante, si può forzare l'esecuzione di ogni richiesta nel suo processo PHP, per evitare effetti collaterali quando si lavora con molti client nello stesso script:

```
$client->insulate();
```

## Browser

Il client supporta molte operazioni eseguibili in un browser reale:

```
$client->back();
$client->forward();
$client->reload();

// Pulisce tutti i cookie e la cronologia
$client->restart();
```

## Accesso agli oggetti interni

New in version 2.3: I metodi **:method:‘Symfony\\Component\\BrowserKit\\Client::getInternalRequest‘** e **:method:‘Symfony\\Component\\BrowserKit\\Client::getInternalResponse‘** sono stati aggiunti in Symfony 2.3.

Se si usa il client per testare la propria applicazione, si potrebbe voler accedere agli oggetti interni del client:

```
$history = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

I possono anche ottenere gli oggetti relativi all'ultima richiesta:

```
// l'istanza della richiesta HttpKernel
$request = $client->getRequest();

// l'istanza della richiesta BrowserKit
$request = $client->getInternalRequest();

// l'istanza della richiesta HttpKernel
$response = $client->getResponse();

// l'istanza della richiesta BrowserKit
$response = $client->getInternalResponse();

$crawler = $client->getCrawler();
```

Se le richieste non sono isolate, si può accedere agli oggetti `Container` e `Kernel`:

```
$container = $client->getContainer();
$kernel = $client->getKernel();
```

## Accesso al contenitore

È caldamente raccomandato che un test funzionale testi solo la risposta. Ma sotto alcune rare circostanze, si potrebbe voler accedere ad alcuni oggetti interni, per scrivere asserzioni. In questi casi, si può accedere al contenitore di dipendenze:

```
$container = $client->getContainer();
```

Attenzione, perché ciò non funziona se si isola il client o se si usa un livello HTTP. Per un elenco di servizi disponibili nell'applicazione, usare il comando `debug:container`.

New in version 2.6: Prima di ymfony 2.6, questo comando si chiamava `container:debug`.

**Tip:** Se l'informazione che occorre verificare è disponibile nel profilatore, si usi invece quest'ultimo.

## Accedere ai dati del profilatore

A ogni richiesta, il profilatore di Symfony raccoglie e memorizza molti dati, che riguardano la gestione interna della richiesta stessa. Per esempio, il profilatore può essere usato per verificare che una data pagina esegua meno di un certo numero di query alla base dati.

Si può ottenere il profilatore dell'ultima richiesta in questo modo:

```
// abilita il profilatore solo per la prossima richiesta
$client->enableProfiler();

$crawler = $client->request('GET', '/profiler');

// prende il profilatore
$profile = $client->getProfile();
```

Per dettagli specifici sull'uso del profilatore in un test, vedere la ricetta /cookbook/testing/profiling.

## Rinvii

Quando una richiesta restituisce una risposta di rinvio, il client la segue automaticamente. Se si vuole esaminare la risposta prima del rinvio, si può forzare il client a non seguire i rinvii, usando il metodo `followRedirect()`:

```
$crawler = $client->followRedirect();
```

Se si vuole che il client segua automaticamente tutti i rinvii, si può forzarlo con il metodo `followRedirects()`:

```
$client->followRedirects();
```

Se si passa `false` al metodo `followRedirects()`, i rinvii non saranno più seguiti:

```
$client->followRedirects(false);
```

## Il crawler

Un'istanza del crawler è creata automaticamente quando si esegue una richiesta con un client. Consente di attraversare i documenti HTML, selezionare nodi, trovare collegamenti e form.

## Attraversamento

Come jQuery, il crawler dispone di metodi per attraversare il DOM di documenti HTML/XML. Per esempio, per estrarre tutti gli elementi `input[type=submit]`, trovarne l'ultimo e quindi selezionare il suo genitore:

```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Ci sono molti altri metodi a disposizione:

**`filter('h1.title')`** Nodi corrispondenti al selettore CSS

**`filterXpath('h1')`** Nodi corrispondenti all'espressione XPath

**`eq(1)`** Nodi per l'indice specificato

**`first()`** Primo nodo

**`last()`** Ultimo nodo

**`siblings()`** Fratelli

**`nextAll()`** Tutti i fratelli successivi

**previousAll()** Tutti i fratelli precedenti

**parents()** Genitori

**children()** Figli

**reduce($lambda)** Nodi per cui la funzione non restituisce false

Si può iterativamente restringere la selezione del nodo, concatenando le chiamate ai metodi, perché ogni metodo restituisce una nuova istanza di Crawler per i nodi corrispondenti:

```
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i) {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first()
;
```

**Tip:** Usare la funzione `count()` per ottenere il numero di nodi memorizzati in un crawler: `count($crawler)`

## Estrarre informazioni

Il crawler può estrarre informazioni dai nodi:

```
// Restituisce il valore dell'attributo del primo nodo
$crawler->attr('class');

// Restituisce il valore del nodo del primo nodo
$crawler->text();

// Estrae un array di attributi per tutti i nodi
// (_text restituisce il valore del nodo)
// restituisce un array per ogni elemento nel crawler,
// ciascuno con valore e href
$info = $crawler->extract(array('_text', 'href'));

// Esegue una funzione lambda per ogni nodo e restituisce un array di risultati
$data = $crawler->each(function ($node, $i) {
    return $node->attr('href');
});
```

## Collegamenti

Si possono selezionare collegamenti coi metodi di attraversamento, ma la scorciatoia `selectLink()` è spesso più conveniente:

```
$crawler->selectLink('Clicca qui');
```

Seleziona i collegamenti che contengono il testo dato, oppure le immagini cliccabili per cui l'attributi `alt` contiene il testo dato. Come gli altri metodi filtro, restituisce un altro oggetto `Crawler`.

Una volta selezionato un collegamento, si ha accesso a uno speciale oggetto `Link`, che ha utili metodi specifici per i collegamenti (come `getMethod()` e `getUri()`). Per cliccare sul collegamento, usare il metodo `click()` di Client e passargli un oggetto `Link`:

```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

## Form

Come per i collegamenti, si possono selezionare i form col metodo `selectButton()`, come i link:

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

---

**Note:** Si noti che si selezionano i bottoni dei form e non i form stessi, perché un form può avere più bottoni; se si usa l'API di attraversamento, si tenga a mente che si deve cercare un bottone.

---

Il metodo `selectButton()` può selezionare i tag `button` e i tag `input` con attributo "submit". Ha diverse euristiche per trovarli:

- Il valore dell'attributo `value`;
- Il valore dell'attributo `id` o `alt` per le immagini;
- Il valore dell'attributo `id` o `name` per i tag `button`.

Quando si ha un nodo che rappresenta un bottone, richiamare il metodo `form()` per ottenere un'istanza `Form` per il form, che contiene il nodo bottone.

    $form = $buttonCrawlerNode->form();

Quando si richiama il metodo `form()`, si può anche passare un array di valori di campi, che sovrascrivano quelli predefiniti:

```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony spacca!',
));
```

Se si vuole emulare uno specifico metodo HTTP per il form, passarlo come secondo parametro:

```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

Il client puoi inviare istanze di `Form`:

```
$client->submit($form);
```

Si possono anche passare i valori dei campi come secondo parametro del metodo `submit()`:

```
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony spacca!',
));
```

Per situazioni più complesse, usare l'istanza di `Form` come un array, per impostare ogni valore di campo individualmente:

```
// Cambiare il valore di un campo
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony spacca!';
```

C'è anche un'utile API per manipolare i valori dei campi, a seconda del tipo:

```
// Selezionare un'opzione o un radio
$form['country']->select('France');

// Spuntare un checkbox
$form['like_symfony']->tick();

// Caricare un file
$form['photo']->upload('/percorso/di/lucas.jpg');
```

**Tip:** Se si vogliono selezionare apposta valori non validi per select o radio, si veda components-dom-crawler-invalid.

**Tip:** Si possono ottenere i valori che saranno inviati, richiamando il metodo `getValues()` sull'oggetto `Form`. I file caricati sono disponibili in un array separato, restituito dal metodo `getFiles()`. Anche i metodi `getPhpValues()` e `getPhpFiles()` restituiscono i valori inviati, ma nel formato di PHP (convertendo le chiavi con parentesi quadre, p.e. `my_form[subject]`, in array PHP).

## Configurazione dei test

Il client usato dai test funzionali crea un kernel che gira in uno speciale ambiente `test`. Siccome Symfony carica `app/config/config_test.yml` in ambiente `test`, si possono modificare le impostazioni dell'applicazione specificatamente per i test.

Per esempio, swiftmailer è configurato in modo predefinito per *non* inviare le email in ambiente `test`. Lo si può vedere sotto l'opzione di configurazione `swiftmailer`:

Si può anche cambiare l'ambiente predefinito (`test`) e sovrascrivere la modalità predefinita di debug (`true`) passandoli come opzioni al metodo `createClient()`:

```
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

Se la propria applicazione necessita di alcuni header HTTP, passarli come secondo parametro di `createClient()`:

```
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Si possono anche sovrascrivere gli header HTTP a ogni richiesta:

```
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

**Tip:** Il client dei test è disponibile come servizio nel contenitore, in ambiente `test` (o dovunque sia abilitata l'opzione framework.test). Questo vuol dire che si può ridefinire completamente il servizio, qualora se ne avesse la necessità.

## Configurazione di PHPUnit

Ogni applicazione ha la sua configurazione di PHPUnit, memorizzata nel file `app/phpunit.xml.dist`. Si può modificare tale file, per cambiare i parameteri predefiniti, oppure creare un file `app/phpunit.xml`, per adattare la configurazione per la propria macchina locale.

**Tip:** Inserire il file `phpunit.xml.dist` nel repository e ignorare il file `phpunit.xml`.

Per impostazione predefinita, solo i test memorizzati nelle cartelle "standard" sono eseguiti dal comando `phpunit` (per "standard" si intendono i test nelle cartelle `src/*/Bundle/Tests`, `src/*/Bundle/*Bundle/Tests` o `src/*Bundle/Tests`), come configurato nel file `app/phpunit.xml.dist`:

```
<!-- app/phpunit.xml.dist -->
<phpunit>
    <!-- ... -->
    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/*/Bundle/*Bundle/Tests</directory>
            <directory>../src/*Bundle/Tests</directory>
        </testsuite>
    </testsuites>
    <!-- ... -->
</phpunit>
```

Ma si possono facilmente aggiungere altri spazi dei nomi. Per esempio, la configurazione seguente aggiunge i test per la cartella `lib/tests`:

```
<!-- app/phpunit.xml.dist -->
<phpunit>
    <!-- ... -->
    <testsuites>
        <testsuite name="Project Test Suite">
            <!-- ... --->
            <directory>../lib/tests</directory>
        </testsuite>
    </testsuites>
    <!-- ... --->
</phpunit>
```

Per includere altre cartelle nella copertura del codice, modificare anche la sezione `<filter>`:

```
<!-- app/phpunit.xml.dist -->
<phpunit>
    <!-- ... -->
    <filter>
        <whitelist>
            <!-- ... -->
            <directory>../lib</directory>
```

```
            <exclude>
                <!-- ... -->
                <directory>../lib/tests</directory>
            </exclude>
        </whitelist>
    </filter>
    <!-- ... -->>
</phpunit>
```

## Saperne di più

- Il capitolo sui test nelle best practice
- /components/dom_crawler
- /components/css_selector
- /cookbook/testing/http_authentication
- /cookbook/testing/insulating_clients
- /cookbook/testing/profiling
- /cookbook/testing/bootstrap

CHAPTER 13

# Validazione

La validazione è un compito molto comune nella applicazioni web. I dati inseriti nei form hanno bisogno di essere validati. I dati hanno bisogno di essere validati anche prima di essere inseriti in una base dati o passati a un servizio web.

Symfony ha un componente Validator , che rende questo compito facile e trasparente. Questo componente è bastato sulle specifiche di validazione JSR303 Bean.

## Le basi della validazione

Il modo migliore per capire la validazione è quello di vederla in azione. Per iniziare, supponiamo di aver creato un classico oggetto PHP, da usare in qualche parte della propria applicazione:

```
// src/AppBundle/Entity/Author.php
namespace AppBundle\Entity;

class Author
{
    public $name;
}
```

Finora, questa è solo una normale classe, che ha una qualche utilità all'interno della propria applicazione. Lo scopo della validazione è dire se i dati di un oggetto siano validi o meno. Per poterlo fare, occorre configurare una lisa di regole (chiamate *vincoli*) che l'oggetto deve seguire per poter essere valido. Queste regole possono essere specificate tramite diversi formati (YAML, XML, annotazioni o PHP).

Per esempio, per garantire che la proprietà `$name` non sia vuota, aggiungere il seguente:

**Tip:** Anche le proprietà private e protette possono essere validate, così come i metodi "getter" (vedere *Obiettivi dei vincoli*).

## Usare il servizio `validator`

Successivamente, per validare veramente un oggetto `Author`, usare il metodo `validate` sul servizio `validator` (classe `Symfony\Component\Validator\Validator`). Il compito di `validator` è semplice: leggere i vincoli (cioè le regole) di una classe e verificare se i dati dell'oggetto soddisfino o no tali vincoli. Se la validazione fallisce, viene restituita una lista di errori (classe `Symfony\Component\Validator\ConstraintViolationList`). Prendiamo questo semplice esempio dall'interno di un controllore:

```
// ...
use Symfony\Component\HttpFoundation\Response;
use AppBundle\Entity\Author;

// ...
public function authorAction()
{
    $autore = new Author();

    // ... fare qualcosa con l'oggetto $autore

    $validator = $this->get('validator');
    $errori = $validator->validate($autore);

    if (count($errori) > 0) {
        /*
         * Usa un metodo a __toString sulla variabile $errors, che è un oggetto
         * ConstraintViolationList. Questo fornisce una stringa adatta
         * al debug
         */
        $errorsString = (string) $errori;

        return new Response($errorsString);
    }

    return new Response('L\'autore è valido! Sì!');
}
```

Se la proprietà `$name` è vuota, si vedrà il seguente messaggio di errore:

```
AppBundle\Author.name:
    This value should not be blank
```

Se si inserisce un valore per la proprietà `$name`, apparirà il messaggio di successo.

---

**Tip:** La maggior parte delle volte, non si interagirà direttamente con il servizio `validator`, né ci si dovrà occupare di stampare gli errori. La maggior parte delle volte, si userà indirettamente la validazione, durante la gestione di dati inviati tramite form. Per maggiori informazioni, vedere *Validazione e form*.

---

Si può anche passare un insieme di errori in un template:

```
if (count($errors) > 0) {
    return $this->render('author/validation.html.twig', array(
        'errors' => $errors,
    ));
}
```

Dentro al template, si può stampare la lista di errori, come necessario:

**Note:** Ogni errore di validazione (chiamato "violazione di vincolo") è rappresentato da un oggetto `Symfony\Component\Validator\ConstraintViolation`.

### Validazione e form

Il servizio `validator` può essere usato per validare qualsiasi oggetto. In realtà, tuttavia, solitamente si lavorerà con `validator` indirettamente, lavorando con i form. La libreria dei form di Symfony usa internamente il servizio `validator`, per validare l'oggetto sottostante dopo che i valori sono stati inviati e collegati. Le violazioni dei vincoli sull'oggetto sono convertite in oggetti `FieldError`, che possono essere facilmente mostrati con il proprio form. Il tipico flusso dell'invio di un form assomiglia al seguente, all'interno di un controllore:

```
// ...
use AppBundle\Entity\Author;
use AppBundle\Form\AuthorType;
use Symfony\Component\HttpFoundation\Request;

// ...
public function updateAction(Request $request)
{
    $author = new Author();
    $form = $this->createForm(new AuthorType(), $author);

    $form->handleRequest($request);

    if ($form->isValid()) {
        // validazione passata, fare qualcosa con l'oggetto $author

        return $this->redirect($this->generateUrl(...));
    }

    return $this->render('author/form.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

**Note:** Questo esempio usa una classe `AuthorType`, non mostrata qui.

Per maggiori informazioni, vedere il capitolo sui form.

## Configurazione

La validazione in Symfony è abilitata per configurazione predefinita, ma si devono abilitare esplicitamente le annotazioni, se le si usano per specificare i vincoli:

# Vincoli

Il servizio `validator` è progettato per validare oggetti in base a *vincoli* (cioè regole). Per poter validare un oggetto, basta mappare uno o più vincoli alle rispettive classi e quindi passarli al servizio `validator`.

Dietro le quinte, un vincolo è semplicemente un oggetto PHP che esegue un'istruzione assertiva. Nella vita reale, un vincolo potrebbe essere "la torta non deve essere bruciata". In Symfony, i vincoli sono simili: sono asserzioni sulla verità di una condizione. Dato un valore, un vincolo dirà se tale valore sia aderente o meno alle regole del vincolo.

## Vincoli supportati

Symfony dispone di un gran numero dei vincoli più comunemente necessari:

Si possono anche creare i propri vincoli personalizzati. L'argomento è discusso nell'articolo "/cookbook/validation/custom_constraint" del ricettario.

## Configurazione dei vincoli

Alcuni vincoli, come NotBlank, sono semplici, mentre altri, come Choice, hanno diverse opzioni di configurazione disponibili. Supponiamo che la classe `Author` abbia un'altra proprietà, `gender`, che possa valere solo "M", "F" o "altro": Le opzioni di un vincolo possono sempre essere passate come array. Alcuni vincoli, tuttavia, consentono anche di passare il valore di una sola opzione, *predefinita*, al posto dell'array. Nel caso del vincolo `Choice`, l'opzione `choices` può essere specificata in tal modo.

Questo ha il solo scopo di rendere la configurazione delle opzioni più comuni di un vincolo più breve e rapida.

Se non si è sicuri di come specificare un'opzione, verificare la documentazione delle API per il vincolo relativo, oppure andare sul sicuro passando sempre un array di opzioni (il primo metodo mostrato sopra).

# Traduzione dei messaggi dei vincoli

Per informazioni sulla traduzione dei messaggi dei vincoli, vedere *Tradurre i messaggi dei vincoli*.

# Obiettivi dei vincoli

I vincoli possono essere applicati alle proprietà di una classe (p.e. `$name`) oppure a un metodo getter pubblico (p.e. `getFullName`). Il primo è il modo più comune e facile, ma il secondo consente di specificare regole di validazione più complesse.

## Proprietà

La validazione delle proprietà di una classe è la tecnica di base. Symfony consente di validare proprietà private, protette o pubbliche. L'elenco seguente mostra come configurare la proprietà `$firstName` di una classe `Author`, per avere almeno 3 caratteri.

### Getter

I vincoli si possono anche applicare ai valori restituiti da un metodo. Symfony2 consente di aggiungere un vincolo a qualsiasi metodo il cui nome inizi per "get", "is" o "has". In questa guida, si fa riferimento a questi tipi di metodi come "getter".

New in version 2.5: Il supporto per metodi che iniziano per `has` è stato introdotto in Symfony 2.5.

Il vantaggio di questa tecnica è che consente di validare gli oggetti dinamicamente. Per esempio, supponiamo che ci si voglia assicurare che un campo password non corrisponda al nome dell'utente (per motivi di sicurezza). Lo si può fare creando un metodo `isPasswordLegal` e asserendo che tale metodo debba restituire `true`:

Creare ora il metodo `isPasswordLegal()` e includervi la logica necessaria:

```
public function isPasswordLegal()
{
    return $this->firstName !== $this->password;
}
```

**Note:** I lettori più attenti avranno notato che il prefisso del getter ("get" o "is") viene omesso nella mappatura. Questo consente di spostare il vincolo su una proprietà con lo stesso nome, in un secondo momento (o viceversa), senza dover cambiare la logica di validazione.

### Classi

Alcuni vincoli si applicano all'intera classe da validare. Per esempio, il vincolo Callback è un vincolo generico, che si applica alla classe stessa. Quando tale classe viene validata, i metodi specifici di questo vincolo vengono semplicemente eseguiti, in modo che ognuno possa fornire una validazione personalizzata.

## Gruppi di validazione

Finora, è stato possibile aggiungere vincoli a una classe e chiedere se tale classe passasse o meno tutti i vincoli definiti. In alcuni casi, tuttavia, occorre validare un oggetto solo per *alcuni* vincoli della sua classe. Per poterlo fare, si può organizzare ogni vincolo in uno o più "gruppi di validazione" e quindi applicare la validazione solo su un gruppo di vincoli.

Per esempio, si supponga di avere una classe `User`, usata sia quando un utente si registra che quando aggiorna successivamente le sue informazioni:

Con questa configurazione, ci sono tre gruppi di validazione:

**`Default`** Contiene i vincoli, nella classe corrente e in tutte le classi referenziate, che non appartengono ad altri gruppi.

**`User`** Equivalente a tutti i i vincoli dell'oggetto `User` nel gruppo `Default`. È sempre il nome della classe. La differenza tra questo e `Default` è spiegato più avanti.

**`registration`** Contiene solo i vincoli sui campi `email` e `password`.

Per dire al validatore di usare uno specifico gruppo, passare uno o più nomi di gruppo come secondo parametro del metodo `validate()`:

```
// Se si usa la nuova API di validazione 2.5 (è probabile)
$errors = $validator->validate($author, null, array('registration'));
```

```
// Se si usa la vecchia API di validazione 2.4
// $errors = $validator->validate($author, array('registration'));
```

Se non si specifica alcun gruppo, saranno applicati tutti i vincoli che appartengono al gruppo `Default`.

Ovviamente, di solito si lavorerà con la validazione in modo indiretto, tramite la libreria dei form. Per informazioni su come usare i gruppi di validazione dentro ai form, vedere *Gruppi di validatori*.

## Sequenza di gruppi

A volte si vogliono validare i gruppi in passi separati. Lo si può fare, usando `GroupSequence`. In questo caso, un oggetto definisce una sequenza di gruppi e i gruppi in tale sequenza sono validati in ordine.

Per esempio, si supponga di avere una classe `User` e di voler validare che nome utente e password siano diversi, solo se le altre validazioni passano (per evitare messaggi di errore multipli).

In questo esempio, prima saranno validati i vincoli del gruppo `User` (che corrispondono a quelli del gruppo `Default`). Solo se tutti i vincoli in tale gruppo sono validi, sarà validato il secondo gruppo, `Strict`.

**Caution:** Come già visto nella precedente sezione, il gruppo `Default` e il gruppo contenente il nome della classe (p.e. `User`) erano identici. Tuttavia, quando si usando le sequenza di gruppo, non lo sono più. Il gruppo `Default` farà ora riferimento alla sequenza digruppo, al posto di tutti i vincoli che non appartengono ad alcun gruppo.

Questo vuol dire che si deve usare il gruppo `{NomeClasse}` (p.e. `User`) quando si specifica una sequenza di gruppo. Quando si usa `Default`, si avrà una ricorsione infinita (poiché il gruppo `Default` si riferisce alla sequenza di gruppo, che contiene il gruppo `Default`, che si riferisce alla stessa sequenza di gruppo, ecc...).

### Fornitori di sequenza di gruppo

Si immagini un'entità `User`, che potrebbe essere un utente normale oppure premium. Se è premium, necessita di alcuni vincoli aggiuntivi (p.e. dettagli sulla carta di credito). Per determinare in modo dinamico quali gruppi attivare, si può creare un Group Sequence Provider. Creare prima l'entità e aggiungere un nuovo gruppo di vincoli, chiamato `Premium`:

Cambiare ora la classe `User` per implementare `Symfony\Component\Validator\GroupSequenceProviderInterface` e aggiungere **:method:'Symfony\\Component\\Validator\\GroupSequenceProviderInterface::getGroupSequence'**, che deve restituire un array di gruppi da usare:

```
// src/AppBundle/Entity/User.php
namespace AppBundle\Entity;

// ...
use Symfony\Component\Validator\GroupSequenceProviderInterface;

class User implements GroupSequenceProviderInterface
{
    // ...

    public function getGroupSequence()
    {
        $groups = array('User');
```

```
        if ($this->isPremium()) {
            $groups[] = 'Premium';
        }

        return $groups;
    }
}
```

Infine, occorre notificare al componente Validator che la classe `User` fornisce una sequenza di gruppi da validare:

## Validare valori e array

Finora abbiamo visto come si possono validare oggetti interi. Ma a volte si vuole validare solo un semplice valore, come verificare che una stringa sia un indirizzo email valido. Lo si può fare molto facilmente. Da dentro a un controllore, assomiglia a questo:

```
// ...
use Symfony\Component\Validator\Constraints as Assert;

// ...
public function addEmailAction($email)
{
    $emailConstraint = new Assert\Email();
    // tutte le opzioni sui vincoli possono essere impostate in questo modo
    $emailConstraint->message = 'Indirizzo email non valido';

    // usa il validatore per validare il valore
    // Se si usa la nuova API di validazione 2.5 (è probabile)
    $errorList = $this->get('validator')->validate(
        $email,
        $emailConstraint
    );

    // Se si usa la vecchia API di validazione 2.4
    /*
    $errorList = $this->get('validator')->validateValue(
        $email,
        $emailConstraint
    );
    */

    if (0 === count($errorList)) {
        // è un indirizzo email valido, fare qualcosa
    } else {
        // *non* è un indirizzo email valido
        $errorMessage = $errorList[0]->getMessage();

        // ... fare qualcosa con l'errore
    }

    // ...
}
```

Richiamando `validateValue` sul validatore, si può passare un valore grezzo e l'oggetto vincolo su cui si vuole validare tale valore. Una lista completa di vincoli disponibili, così come i nomi completi delle classi per ciascun vincolo, è disponibile nella sezione riferimento sui vincoli.

Il metodo `validateValue` restituisce un oggetto `Symfony\Component\Validator\ConstraintViolationList`, che si comporta come un array di errori. Ciascun errore della lista è un oggetto `Symfony\Component\Validator\ConstraintViolation`, che contiene il messaggio di errore nel suo metodo `getMessage`.

## Considerazioni finali

`validator` di Symfony è uno strumento potente, che può essere sfruttato per garantire che i dati di qualsiasi oggetto siano validi. La potenza dietro alla validazione risiede nei "vincoli", che sono regole da applicare alle proprietà o ai metodi getter del proprio oggetto. Sebbene la maggior parte delle volte si userà il framework della validazione indirettamente, usando i form, si ricordi che può essere usato ovunque, per validare qualsiasi oggetto.

## Imparare di più con le ricette

- /cookbook/validation/custom_constraint

CHAPTER 14

# Form

L'utilizzo dei form HTML è una delle attività più comuni e stimolanti per uno sviluppatore web. Symfony integra un componente Form che permette di gestire facilmente i form. Con l'aiuto di questo capitolo si potrà creare da zero un form complesso, e imparare le caratteristiche più importanti della libreria dei form.

**Note:** Il componente form di Symfony è una libreria autonoma che può essere usata al di fuori dei progetti Symfony. Per maggiori informazioni, vedere la documentazione del componente Form su GitHub.

## Creazione di un form semplice

Supponiamo che si stia costruendo un semplice applicazione "elenco delle cose da fare" che dovrà visualizzare le "attività". Poiché gli utenti avranno bisogno di modificare e creare attività, sarà necessario costruire un form. Ma prima di iniziare, si andrà a vedere la generica classe `Task` che rappresenta e memorizza i dati di una singola attività:

```
// src/AppBundle/Entity/Task.php
namespace AppBundle\Entity;

class Task
{
    protected $task;
    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }

    public function setTask($task)
    {
        $this->task = $task;
    }
```

```
    public function getDueDate()
    {
        return $this->dueDate;
    }

    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Questa classe è un "vecchio-semplice-oggetto-PHP", perché finora non ha nulla a che fare con Symfony o qualsiasi altra libreria. È semplicemente un normale oggetto PHP, che risolve un problema direttamente dentro la *propria* applicazione (cioè la necessità di rappresentare un task nella propria applicazione). Naturalmente, alla fine di questo capitolo, si sarà in grado di inviare dati all'istanza di un `Task` (tramite un form HTML), validare i suoi dati e persisterli nella base dati.

## Costruire il Form

Ora che la classe `Task` è stata creata, il prossimo passo è creare e visualizzare il form HTML. In Symfony, lo si fa costruendo un oggetto form e poi visualizzandolo in un template. Per ora, lo si può fare all'interno di un controllore:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use AppBundle\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // crea un task fornendo alcuni dati fittizi per questo esempio
        $task = new Task();
        $task->setTask('Scrivere un post sul blog');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->add('save', 'submit', array('label' => 'Crea post'))
            ->getForm();

        return $this->render('default/new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

**Tip:** Questo esempio mostra come costruire il form direttamente nel controllore. Più tardi, nella sezione "*Creare classi per i form*", si imparerà come costruire il form in una classe autonoma, metodo consigliato perché in questo modo il form diventa riutilizzabile.

La creazione di un form richiede relativamente poco codice, perché gli oggetti form di Symfony sono costruiti con un "costruttore di form". Lo scopo del costruttore di form è quello di consentire di scrivere una semplice "ricetta" per il form e fargli fare tutto il lavoro pesante della costruzione del form.

In questo esempio sono stati aggiunti due campi al form, `task` e `dueDate`, corrispondenti alle proprietà `task` e `dueDate` della classe `Task`. È stato anche assegnato un "tipo" ciascuno (ad esempio `text`, `date`), che, tra le altre cose, determina quale tag form HTML viene utilizzato per tale campo.

Infine, è stato aggiunto un bottone submit, con un'etichetta personalizzata, per l'invio del form.

New in version 2.3: Il supporto per i bottoni submit è stato aggiunto in Symfony 2.3. Precedentemente, era necessario aggiungere i bottoni manualmente nel codice HTML.

Symfony ha molti tipi predefiniti che verranno trattati a breve (see *Tipi di campo predefiniti*).

## Visualizzare il Form

Ora che il modulo è stato creato, il passo successivo è quello di visualizzarlo. Questo viene fatto passando uno speciale oggetto form "view" al template (notare il `$form->createView()` nel controllore sopra) e utilizzando una serie di funzioni aiutanti per i form:

images/book/form-simple.png

**Note:** Questo esempio presuppone che sia stata creata una rotta chiamata `task_new` che punta al controllore `AcmeTaskBundle:Default:new` che era stato creato precedentemente.

Questo è tutto! Bastano tre righe per rendere completamente il form:

**`form_start(form)`** Rende il tag iniziale del form, incluso l'attributo `enctype`, se si usa un caricamento di file;

**`form_widget(form)`** Rende tutti i campi, inclusi l'elemento stesso, un'etichetta ed eventuali messaggi di errori;

**`form_end(form)`** Rende il tag finale del form e ogni campo che non sia ancora stato reso, nel caso in cui i campi siano stati resti singolarmante a mano. È utile per rendere campi nascosti e sfruttare la *protezione CSRF* automatica.

**See also:**

Pur essendo facile, non è (ancora) flessibile. Di solito, si vorranno rendere i singoli campi, in modo da poter controllare l'aspetto del form. Si vedrà come fare nella sezione "*Rendere un form in un template*".

Prima di andare avanti, notare come il campo input `task` reso ha il value della proprietà `task` dall'oggetto `$task` (ad esempio "Scrivere un post sul blog"). Questo è il primo compito di un form: prendere i dati da un oggetto e tradurli in un formato adatto a essere reso in un form HTML.

**Tip:** Il sistema dei form è abbastanza intelligente da accedere al valore della proprietà protetta `task` attraverso i metodi `getTask()` e `setTask()` della classe `Task`. A meno che una proprietà non sia privata, *deve* avere un metodo "getter" e uno "setter", in modo che il componente form possa ottenere e mettere dati nella proprietà. Per una proprietà booleana, è possibile utilizzare un metodo "isser" o "hasser" (per esempio `isPublished()` o `hasReminder`) invece di un getter (per esempio `getPublished()` o `getReminder()`).

## Gestione dell'invio del form

Il secondo compito di un form è quello di tradurre i dati inviati dall'utente alle proprietà di un oggetto. Affinché ciò avvenga, i dati inviati dall'utente devono essere associati al form. Aggiungere le seguenti funzionalità al controllore:

```
// ...
use Symfony\Component\HttpFoundation\Request;

public function newAction(Request $request)
{
    // crea un nuovo oggetto $task (rimuove i dati fittizi)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->add('save', 'submit', array('label' => 'Crea post'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // esegue alcune azioni, come ad esempio salvare il task nella base dati

        return $this->redirect($this->generateUrl('task_success'));
    }

    // ...
}
```

New in version 2.3: Il metodo **:method:'Symfony\\Component\\Form\\FormInterface::handleRequest'** è stato aggiunto in Symfony 2.3. In precedenza, veniva passata `$request` al metodo `submit`, una strategia deprecata, che sarà rimossa in Symfony 3.0. Per dettagli sul metodo, vedere cookbook-form-submit-request.

Questo controllore segue uno schema comune per gestire i form e ha tre possibili percorsi:

1. Quando in un browser inizia il caricamento di una pagina, il form viene creato e reso. **:method:'Symfony\\Component\\Form\\FormInterface::handleRequest'** capisce che il form non è stato inviato e non fa nulla. **:method:'Symfony\\Component\\Form\\FormInterface::isValid'** restituisce `false` se il form non è stato inviato.
2. Quando l'utente invia il form, **:method:'Symfony\\Component\\Form\\FormInterface::handleRequest'** lo capisce e scrive immediatamente i dati nelle proprietà `task` e `dueDate` dell'oggetto `$task`. Quindi tale oggetto viene validato. Se non è valido (la validazione è trattata nella prossima sezione), **:method:'Symfony\\Component\\Form\\FormInterface::isValid'** restituisce `false` di nuovo, quindi il form viene reso insieme agli errori di validazione;

   **Note:** Si può usare il metodo **:method:'Symfony\\Component\\Form\\FormInterface::isSubmitted'** per verificare se il form sia stato inviato, indipendentemente dal fatto che i dati inviati siano validi o meno.

3. Quando l'utente invia il form con dati validi, i dati inviati sono scritti nuovamente nel form, ma stavolta **:method:'Symfony\\Component\\Form\\FormInterface::isValid'** restituisce `true`. Ora si ha la possibilità di eseguire alcune azioni usando l'oggetto `$task` (ad esempio persistendolo nella base dati) prima di rinviare l'utente a un'altra pagina (ad esempio una pagina "thank you" o "success").

**Note:** Reindirizzare un utente dopo aver inviato con successo un form impedisce l'utente di essere in grado di premere

il tasto “aggiorna” del suo browser e reinviare i dati.

**See also:**

Se occorre maggior controllo su quando esattamente il form è inviato o su quali dati siano passati, si può usare il metodo **:method:‘Symfony\\Component\\Form\\FormInterface::submit‘**. Si può approfondire nel ricettario.

### Inviare form con bottoni di submit multipli

New in version 2.3: Il supporto per i bottoni nei form è stato aggiunto in Symfony 2.3.

Quando un form contiene più di un bottone di submit, si vuole sapere quale dei bottoni sia stato cliccato, per adattare il flusso del controllore. Aggiungiamo un secondo bottone “Salva e aggiungi” al form:

```
$form = $this->createFormBuilder($task)
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->add('save', 'submit', array('label' => 'Crea post'))
    ->add('saveAndAdd', 'submit', array('label' => 'Salva e aggiungi'))
    ->getForm();
```

Nel controllore, usare il metodo **:method:‘Symfony\\Component\\Form\\ClickableInterface::isClicked‘** del bottone per sapere se sia stato cliccato il bottone “Salva e aggiungi”:

```
if ($form->isValid()) {
    // ... eseguire un'azione, come salvare il task nella base dati

    $nextAction = $form->get('saveAndAdd')->isClicked()
        ? 'task_new'
        : 'task_success';

    return $this->redirect($this->generateUrl($nextAction));
}
```

## Validare un form

Nella sezione precedente, si è appreso come un form può essere inviato con dati validi o invalidi. In Symfony, la validazione viene applicata all’oggetto sottostante (per esempio `Task`). In altre parole, la questione non è se il “form” è valido, ma se l’oggetto `$task` è valido o meno dopo che al form sono stati applicati i dati inviati. La chiamata di `$form->isValid()` è una scorciatoia che chiede all’oggetto `$task` se ha dati validi o meno.

La validazione è fatta aggiungendo di una serie di regole (chiamate vincoli) a una classe. Per vederla in azione, verranno aggiunti vincoli di validazione in modo che il campo `task` non possa essere vuoto e il campo `dueDate` non possa essere vuoto e debba essere un oggetto DateTime valido.

Questo è tutto! Se si re-invia il form con i dati non validi, si vedranno i rispettivi errori visualizzati nel form.

**Validazione HTML5**

Dall’HTML5, molti browser possono nativamente imporre alcuni vincoli di validazione sul lato client. La validazione più comune è attivata con la resa di un attributo `required` sui campi che sono obbligatori. Per i browser che supportano HTML5, questo si tradurrà in un messaggio nativo del browser che verrà visualizzato se l’utente tenta di inviare il form con quel campo vuoto.

I form generati traggono il massimo vantaggio di questa nuova funzionalità con l'aggiunta di appropriati attributi HTML che verifichino la convalida. La convalida lato client, tuttavia, può essere disabilitata aggiungendo l'attributo `novalidate` al tag `form` o `formnovalidate` al tag submit. Ciò è particolarmente utile quando si desidera testare i propri vincoli di convalida lato server, ma viene impedito dal browser, per esempio, inviando campi vuoti.

La validazione è una caratteristica molto potente di Symfony e dispone di un proprio capitolo dedicato.

## Gruppi di validatori

Se un oggetto si avvale dei *gruppi di validatori*, occorrerà specificare quali gruppi di convalida deve usare il form:

```
$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registrazione'),
))->add(...);
```

Se si stanno creando *classi per i form* (una buona pratica), allora si avrà bisogno di aggiungere quanto segue al metodo `setDefaultOptions()`:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('registrazione'),
    ));
}
```

In entrambi i casi, *solo* il gruppo di validazione `registrazione` verrà utilizzato per validare l'oggetto sottostante.

## Disabilitare la validazione

New in version 2.3: La possibilità di impostare `validation_groups` a `false` è stata aggiunta in Symfony 2.3.

A volte è utile sopprimere la validazione per un intero form. Per questi casi, si può impostare l'opzione `validation_groups` a `false`:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => false,
    ));
}
```

Notare che in questo caso il form eseguirà comunque alcune verifiche basilari di integrità, per esempio se un file caricato è troppo grande o se dei campi non esistenti sono stati inviati. Se si vuole sopprimere completamente la validazione, si può usare l'evento POST_SUBMIT.

## Gruppi basati su dati inseriti

Se si ha bisogno di una logica avanzata per determinare i gruppi di validazione (p.e. basandosi sui dati inseriti), si può impostare l'opzione `validation_groups` a un callback o a una `Closure`:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array(
            'AppBundle\Entity\Client',
            'determineValidationGroups',
        ),
    ));
}
```

Questo richiamerà il metodo statico `determineValidationGroups()` della classe `Client`, dopo il bind del form ma prima dell'esecuzione della validazione. L'oggetto Form è passato come parametro del metodo (vedere l'esempio successivo). Si può anche definire l'intera logica con una Closure:

```
use AppBundle\Entity\Client;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function (FormInterface $form) {
            $data = $form->getData();
            if (Client::TYPE_PERSON == $data->getType()) {
                return array('person');
            }

            return array('company');
        },
    ));
}
```

L'uso dell'opzione `validation_groups` sovrascrive il gruppo di validazione predefinito in uso. Se si vogliono validare anche i vincoli predefiniti dell'entità, si deve cambiare l'opzione in questo modo:

```
use AppBundle\Entity\Client;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function (FormInterface $form) {
            $data = $form->getData();
            if (Client::TYPE_PERSON == $data->getType()) {
                return array('Default', 'person');
            }

            return array('Default', 'company');
        },
    ));
}
```

Si possono trovare maggiori informazioni su come funzionino i gruppi di validazione e i vincoli predefiniti nella sezione del libro relativa ai *gruppi di validazione*.

### Gruppi basati sul bottone cliccato

New in version 2.3: Il supporto per i bottoni nei form è stato aggiunto in Symfony 2.3.

Se un form contiene più bottoni submit, si può modificare il gruppo di validazione, a seconda di quale bottone sia stato usato per inviare il form. Per esempi, consideriamo un form in sequenza, in cui si può avanzare al passo successivo o tornare al passo precedente. Ipotizziamo anche che, quando si torna al passo precedente, i dati del form debbano essere salvati, ma non validati.

Prima di tutto, bisogna aggiungere i due bottoni al form:

```
$form = $this->createFormBuilder($task)
    // ...
    ->add('nextStep', 'submit')
    ->add('previousStep', 'submit')
    ->getForm();
```

Quindi, occorre configurare il bottone che torna al passo precedente per eseguire specifici gruppi di validazione. In questo esempio, vogliamo sopprimere la validazione, quindi impostiamo l'opzione `validation_groups` a `false`:

```
$form = $this->createFormBuilder($task)
    // ...
    ->add('previousStep', 'submit', array(
        'validation_groups' => false,
    ))
    ->getForm();
```

Ora il form salterà i controlli di validazione. Validerà comunque i vincoli basilari di integrità, come il controllo se un file caricato sia troppo grande o se si sia tentato di inserire del testo in un campo numerico.

## Tipi di campo predefiniti

Symfony dispone di un folto gruppo di tipi di campi che coprono tutti i campi più comuni e i tipi di dati di cui necessitano i form:

È anche possibile creare dei tipi di campi personalizzati. Questo argomento è trattato nell'articolo "/cookbook/form/create_custom_field_type" del ricettario.

### Opzioni dei tipi di campo

Ogni tipo di campo ha un numero di opzioni che può essere utilizzato per la configurazione. Ad esempio, il campo `dueDate` è attualmente reso con 3 menu select. Tuttavia, il campo data può essere configurato per essere reso come una singola casella di testo (in cui l'utente deve inserire la data nella casella come una stringa):

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```


Ogni tipo di campo ha un numero di opzioni differente che possono essere passate a esso. Molte di queste sono specifiche per il tipo di campo e i dettagli possono essere trovati nella documentazione di ciascun tipo.

**L'opzione `required`**

L'opzione più comune è l'opzione `required`, che può essere applicata a qualsiasi campo. Per impostazione predefinita, l'opzione `required` è impostata a `true` e questo significa che i browser che interpretano l'HTML5 applicheranno la validazione lato client se il campo viene lasciato vuoto. Se non si desidera questo comportamento, impostare l'opzione `required` del campo a `false` oppure *disabilitare la validazione HTML5*.

Si noti inoltre che l'impostazione dell'opzione `required` a `true` **non** farà applicare la validazione lato server. In altre parole, se un utente invia un valore vuoto per il campo (sia con un browser vecchio o un servizio web, per esempio), sarà accettata come valore valido a meno che si utilizzi il vincolo di validazione `NotBlank` o `NotNull`.

In altre parole, l'opzione `required` è "bella", ma la vera validazione lato server dovrebbe *sempre* essere utilizzata.

**L'opzione `label`**

La label per il campo del form può essere impostata con l'opzione `label`, applicabile a qualsiasi campo:

```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

La label per un campo può anche essere impostata nel template che rende il form, vedere sotto. Se non occorre alcuna label, la si può disabilitare impostandone il valore a `false`.

## Indovinare il tipo di campo

Ora che sono stati aggiunti i metadati di validazione alla classe `Task`, Symfony sa già un po' dei campi. Se lo si vuole permettere, Symfony può "indovinare" il tipo del campo e impostarlo al posto vostro. In questo esempio, Symfony può indovinare dalle regole di validazione che il campo `task` è un normale campo `text` e che il campo `dueDate` è un campo `date`:

```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->add('save', 'submit')
        ->getForm();
}
```

Questa funzionalità si attiva quando si omette il secondo parametro del metodo `add()` (o se si passa `null` a esso). Se si passa un array di opzioni come terzo parametro (fatto sopra per `dueDate`), queste opzioni vengono applicate al campo indovinato.

**Caution:** Se il form utilizza un gruppo specifico di validazione, la funzionalità che indovina il tipo di campo prenderà ancora in considerazione *tutti* i vincoli di validazione quando andrà a indovinare i tipi di campi (compresi i vincoli che non fanno parte del processo di convalida dei gruppi in uso).

### Indovinare le opzioni dei tipi di campo

Oltre a indovinare il "tipo" di un campo, Symfony può anche provare a indovinare i valori corretti di una serie di opzioni del campo.

**Tip:** Quando queste opzioni vengono impostate, il campo sarà reso con speciali attributi HTML che forniscono la validazione HTML5 lato client. Tuttavia, non genera i vincoli equivalenti lato server (ad esempio `Assert\MaxLength`). E anche se si ha bisogno di aggiungere manualmente la validazione lato server, queste opzioni dei tipi di campo possono essere ricavate da queste informazioni.

**`required`** L'opzione `required` può essere indovinata in base alle regole di validazione (cioè se il campo è `NotBlank` o `NotNull`) o dai metadati di Doctrine (vale a dire se il campo è `nullable`). Questo è molto utile, perché la validazione lato client corrisponderà automaticamente alle vostre regole di validazione.

**`max_length`** Se il campo è un qualche tipo di campo di testo, allora l'opzione `max_length` può essere indovinata dai vincoli di validazione (se viene utilizzato `Length` o `Range`) o dai metadati Doctrine (tramite la lunghezza del campo).

**Note:** Queste opzioni di campi vengono indovinate *solo* se si sta usando Symfony per ricavare il tipo di campo (ovvero omettendo o passando `null` nel secondo parametro di `add()`).

Se si desidera modificare uno dei valori indovinati, è possibile sovrascriverlo passando l'opzione nell'array di opzioni del campo:

```
->add('task', null, array('max_length' => 4))
```

## Rendere un form in un template

Finora si è visto come un intero form può essere reso con una sola linea di codice. Naturalmente, solitamente si ha bisogno di molta più flessibilità:

Abbiamo già visto le funzioni `form_start()` e `form_end()`, ma cosa fanno le altre funzioni?

**`form_errors(form)`** Rende eventuali errori globali per l'intero modulo (gli errori specifici dei campi vengono visualizzati accanto a ciascun campo);

**`form_row(form.dueDate)`** Rende l'etichetta, eventuali errori e il widget HTML del form per il dato campo (p.e. `dueDate`) all'interno, per impostazione predefinita, di un elemento `div`;

La maggior parte del lavoro viene fatto dall'helper `form_row`, che rende l'etichetta, gli errori e i widget HTML del form di ogni campo all'interno di un tag `div` per impostazione predefinita. Nella sezione *Temi con i form*, si apprenderà come l'output di `form_row` possa essere personalizzato su diversi livelli.

**Tip:** Si può accedere ai dati attuali del form tramite `form.vars.value`:

### Rendere manualmente ciascun campo

L'aiutante `form_row` è utile, perché si può rendere ciascun campo del form molto facilmente (e il markup utilizzato per la "riga" può essere personalizzato a piacere). Ma poiché la vita non è sempre così semplice, è anche possibile rendere ogni campo interamente a mano. Il risultato finale del codice che segue è lo stesso di quando si è utilizzato l'aiutante `form_row`:

Se la label auto-generata di un campo non è giusta, si può specificarla esplicitamente:

Alcuni tipi di campi hanno opzioni di resa aggiuntive che possono essere passate al widget. Queste opzioni sono documentate con ogni tipo, ma un'opzione comune è `attr`, che permette di modificare gli attributi dell'elemento form. Di seguito viene aggiunta la classe `task_field` al resa del campo casella di testo:

Se occorre rendere dei campi "a mano", si può accedere ai singoli valori dei campi, come `id`, `name` e `label`. Per esempio, per ottenere `id`:

Per ottenere il valore usato per l'attributo nome dei campi del form, occorre usare il valore `full_name`:

### Riferimento alle funzioni del template Twig

Se si utilizza Twig, un riferimento completo alle funzioni di resa è disponibile nel manuale di riferimento. Leggendolo si può sapere tutto sugli helper disponibili e le opzioni che possono essere usate con ciascuno di essi.

## Cambiare azione e metodo di un form

Finora, è stato usato l'helper `form_start()` per rendere il tag di aperture del form, ipotizzando che ogni form sia inviato allo stesso URL in POST. A volte si vogliono cambiare questi parametri. Lo si può fare in modi diversi. Se si costruisce il form nel controllore, si può usare `setAction()` e `setMethod()`:

```
$form = $this->createFormBuilder($task)
    ->setAction($this->generateUrl('target_route'))
    ->setMethod('GET')
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->add('save', 'submit')
    ->getForm();
```

**Note:** Questo esempio ipotizza la presenza di una rotta `target_route`, che punti al controllore che processerà il form.

In *Creare classi per i form*, vedremo come spostare il codice di costruzione del form in una classe separata. Quando si usa una classe form esterna nel controllore, si possono passare azione e metodo come opzioni:

```
$form = $this->createForm(new TaskType(), $task, array(
    'action' => $this->generateUrl('target_route'),
    'method' => 'GET',
));
```

Infine, si possono sovrascrivere azione e metodo nel template, passandoli all'aiutante `form()` o `form_start()`:

**Note:** Se il metodo del form non è GET o POST, ma PUT, PATCH o DELETE, Symfony inserirà un campo nascosto chiamato "_method", per memorizzare il metodo. Il form sarà inviato in POST, ma il router di Symfony's è in grado di rilevare il parametro "_method" e interpretare la richiesta come PUT, PATCH o DELETE. Si veda la ricetta "/cookbook/routing/method_parameters" per maggiori informazioni.

## Creare classi per i form

Come si è visto, un form può essere creato e utilizzato direttamente in un controllore. Tuttavia, una pratica migliore è quella di costruire il form in una apposita classe PHP, che può essere riutilizzata in qualsiasi punto dell'applicazione. Creare una nuova classe che ospiterà la logica per la costruzione del form task:

```
// src/AppBundle/Form/Type/TaskType.php
namespace AppBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('task')
            ->add('dueDate', null, array('widget' => 'single_text'))
            ->add('save', 'submit')
        ;
    }

    public function getName()
    {
        return 'task';
    }
}
```

Questa nuova classe contiene tutte le indicazioni necessarie per creare il form task (notare che il metodo `getName()` dovrebbe restituire un identificatore univoco per questo "tipo" di form). Può essere usato per costruire rapidamente un oggetto form nel controllore:

```
// src/AppBundle/Controller/DefaultController.php

// add this new use statement at the top of the class
use AppBundle\Form\Type\TaskType;

public function newAction()
{
    $task = ...;
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Porre la logica del form in una classe a parte significa che il form può essere facilmente riutilizzato in altre parti del progetto. Questo è il modo migliore per creare form, ma la scelta in ultima analisi, spetta allo sviluppatore.

**Impostare `data_class`**

Ogni form ha bisogno di sapere il nome della classe che detiene i dati sottostanti (ad esempio `AppBundle\Entity\Task`). Di solito, questo viene indovinato in base all'oggetto passato al secondo parametro di `createForm` (vale a dire `$task`). Dopo, quando si inizia a incorporare i form, questo non sarà più sufficiente. Così, anche se non sempre necessario, è in genere una buona idea specificare esplicitamente l'opzione `data_class` aggiungendo il codice seguente alla classe del tipo di form:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'AppBundle\Entity\Task',
    ));
}
```

**Tip:** Quando si mappano form su oggetti, tutti i campi vengono mappati. Ogni campo nel form che non esiste nell'oggetto mappato causerà il lancio di un'eccezione.

Nel caso in cui servano campi extra nel form (per esempio, un checkbox "accetto i termini"), che non saranno mappati nell'oggetto sottostante, occorre impostare l'opzione `mapped` a `false`:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('task')
        ->add('dueDate', null, array('mapped' => false))
        ->add('save', 'submit')
    ;
}
```

Inoltre, se ci sono campi nel form che non sono inclusi nei dati inviati, tali campi saranno impostati esplicitamente a `null`.

Si può accedere ai dati del campo in un controllore con:

```
$form->get('dueDate')->getData();
```

Inoltre, anche i dati di un campo non mappato si possono modificare direttamente:

```
$form->get('dueDate')->setData(new \DateTime());
```

## Definire i form come servizi

La definizione dei form type come servizi è una buona pratica e li rende riusabili facilmente in un'applicazione.

**Note:** I servizi e il contenitore di servizi saranno trattati più avanti nel libro. Le cose saranno più chiaro dopo aver letto quel capitolo.

Ecco fatto! Ora si può usare il form type direttamente in un controllore:

```
// src/AppBundle/Controller/DefaultController.php
// ...

public function newAction()
{
    $task = ...;
    $form = $this->createForm('task', $task);

    // ...
}
```

o anche usarlo in un altro form:

```
// src/AppBundle/Form/Type/ListType.php
// ...

class ListType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...

        $builder->add('someTask', 'task');
    }
}
```

Si veda form-cookbook-form-field-service per maggiori informazioni.

## I form e Doctrine

L'obiettivo di un form è quello di tradurre i dati da un oggetto (ad esempio `Task`) a un form HTML e quindi tradurre i dati inviati dall'utente indietro all'oggetto originale. Come tale, il tema della persistenza dell'oggetto `Task` nella base dati è interamente non correlato al tema dei form. Ma, se la classe `Task` è stata configurata per essere salvata attraverso Doctrine (vale a dire che per farlo si è aggiunta la *mappatura dei metadati*), allora si può salvare dopo l'invio di un form, quando il form stesso è valido:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($task);
    $em->flush();

    return $this->redirect($this->generateUrl('task_success'));
}
```

Se, per qualche motivo, non si ha accesso all'oggetto originale `$task`, è possibile recuperarlo dal form:

```
$task = $form->getData();
```

Per maggiori informazioni, vedere il capitolo ORM Doctrine.

La cosa fondamentale da capire è che quando il form viene riempito, i dati inviati vengono trasferiti immediatamente all'oggetto sottostante. Se si vuole persistere i dati, è sufficiente persistere l'oggetto stesso (che già contiene i dati inviati).

## Incorporare form

Spesso, si vuole costruire form che includono campi provenienti da oggetti diversi. Ad esempio, un form di registrazione può contenere dati appartenenti a un oggetto `User` così come a molti oggetti `Address`. Fortunatamente, questo è semplice e naturale con il componente per i form.

### Incorporare un oggetto singolo

Supponiamo che ogni `Task` appartenga a un semplice oggetto `Category`. Si parte, naturalmente, con la creazione di un oggetto `Category`:

```
// src/AppBundle/Entity/Category.php
namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Poi, aggiungere una nuova proprietà `category` alla classe `Task`:

```
// ...

class Task
{
    // ...

    /**
     * @Assert\Type(type="AppBundle\Entity\Category")
     * @Assert\Valid()
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

**Tip:** Il vincolo `Valid` è stato aggiunto alla proprietà `category`. In questo modo si valida a cascata l'entità corrispondente. Se si omette tale vincolo, l'entità figlia non sarà validata.

Ora che l'applicazione è stata aggiornata per riflettere le nuove esigenze, creare una classe di form in modo che l'oggetto `Category` possa essere modificato dall'utente:

```
// src/AppBundle/Form/Type/CategoryType.php
namespace AppBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('name');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Category',
        ));
    }

    public function getName()
    {
        return 'category';
    }
}
```

L'obiettivo finale è quello di far si che la `Category` di un `Task` possa essere correttamente modificata all'interno dello stesso form task. Per farlo, aggiungere il campo `category` all'oggetto `TaskType`, il cui tipo è un'istanza della nuova classe `CategoryType`:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('category', new CategoryType());
}
```

I campi di `CategoryType` ora possono essere resi accanto a quelli della classe `TaskType`.

Rendere i campi di `Category` allo stesso modo dei campi `Task` originali:

Quando l'utente invia il form, i dati inviati con i campi `Category` sono utilizzati per costruire un'istanza di `Category`, che viene poi impostata sul campo `category` dell'istanza `Task`.

L'istanza `Category` è accessibile naturalmente attraverso `$task->getCategory()` e può essere memorizzata nella base dati o utilizzata quando serve.

### Incorporare un insieme di form

È anche possibile incorporare un insieme di form in un form (si immagini un form `Category` con tanti sotto-form `Product`). Lo si può fare utilizzando il tipo di campo `collection`.

Per maggiori informazioni, vedere la ricetta "/cookbook/form/form_collections" e il riferimento al tipo collection.

## Temi con i form

Ogni parte nel modo in cui un form viene reso può essere personalizzata. Si è liberi di cambiare come ogni "riga" del form viene resa, modificare il markup utilizzato per rendere gli errori, o anche personalizzare la modalità con cui un tag `textarea` dovrebbe essere rappresentato. Nulla è off-limits, e personalizzazioni differenti possono essere utilizzate in posti diversi.

Symfony utilizza i template per rendere ogni singola parte di un form, come ad esempio i tag `label`, i tag `input`, i messaggi di errore e ogni altra cosa.

In Twig, ogni "frammento" di form è rappresentato da un blocco Twig. Per personalizzare una qualunque parte di come un form è reso, basta sovrascrivere il blocco appropriato.

In PHP, ogni "frammento" è reso tramite un file template individuale. Per personalizzare una qualunque parte del modo in cui un form viene reso, basta sovrascrivere il template esistente creandone uno nuovo.

Per capire come funziona, cerchiamo di personalizzare il frammento `form_row` e aggiungere un attributo class all'elemento `div` che circonda ogni riga. Per farlo, creare un nuovo file template per salvare il nuovo codice:

Il frammento di form `field_row` è utilizzato per rendere la maggior parte dei campi attraverso la funzione `form_row`. Per dire al componente form di utilizzare il nuovo frammento `field_row` definito sopra, aggiungere il codice seguente all'inizio del template che rende il form:

Il tag `form_theme` (in Twig) "importa" i frammenti definiti nel dato template e li usa quando deve rendere il form. In altre parole, quando la funzione `form_row` è successivamente chiamata in questo template, utilizzerà il blocco `field_row` dal tema personalizzato (al posto del blocco predefinito `field_row` fornito con Symfony).

Non è necessario che il tema personalizzato sovrascriva tutti i blocchi. Quando viene reso un blocco non sovrascrritto nel tema personalizzato, il sistema dei temi userà il tema globale (definito a livello di bundle).

Se vengono forniti più temi personalizzati, saranno analizzati nell'ordine elencato, prima di usare il tema globale.

Per personalizzare una qualsiasi parte di un form, basta sovrascrivere il frammento appropriato. Sapere esattamente qual è il blocco o il file da sovrascrivere è l'oggetto della sezione successiva.

Per una trattazione più ampia, vedere /cookbook/form/form_customization.

### Nomi per i frammenti di form

In Symfony, ogni parte di un form che viene reso (elementi HTML del form, errori, etichette, ecc.) è definito in un tema base, che in Twig è una raccolta di blocchi e in PHP una collezione di file template.

In Twig, ogni blocco necessario è definito in un singolo file template (p.e. form_div_layout.html.twig) che si trova all'interno di Twig Bridge. Dentro questo file, è possibile ogni blocco necessario alla resa del form e ogni tipo predefinito di campo.

In PHP, i frammenti sono file template individuali. Per impostazione predefinita sono posizionati nella cartella *Resources/views/Form* del bundle framework (vedere su GitHub).

Ogni nome di frammento segue lo stesso schema di base ed è suddiviso in due pezzi, separati da un singolo carattere di sottolineatura (_). Alcuni esempi sono:

- `field_row` - usato da `form_row` per rendere la maggior parte dei campi;
- `textarea_widget` - usato da `form_widget` per rendere un campo di tipo `textarea`;
- `field_errors` - usato da `form_errors` per rendere gli errori di un campo;

Ogni frammento segue lo stesso schema di base: `type_part`. La parte `type` corrisponde al campo *type* che viene reso (es. `textarea`, `checkbox`, `date`, ecc) mentre la parte `part` corrisponde a *cosa* si sta rendendo (es. `label`, `widget`, `errors`, ecc). Per impostazione predefinita, ci sono 4 possibili *parti* di un form che possono essere rese:

| | | |
|---|---|---|
| `label` | (es. `form_label`) | rende l'etichetta dei campi |
| `widget` | (es. `form_widget`) | rende la rappresentazione HTML dei campi |
| `errors` | (es. `form_errors`) | rende gli errori dei campi |
| `row` | (es. `form_row`) | rende l'intera riga del campo (etichetta, widget ed errori) |

**Note:** In realtà ci sono altre 3 *parti* (`rows`, `rest` e `enctype`), ma raramente c'è la necessità di sovrascriverle.

Conoscendo il tipo di campo (ad esempio `textarea`) e che parte si vuole personalizzare (ad esempio `widget`), si può costruire il nome del frammento che deve essere sovrascritto (esempio `textarea_widget`).

## Ereditarietà dei frammenti di template

In alcuni casi, il frammento che si vuole personalizzare sembrerà mancare. Ad esempio, non c'è nessun frammento `textarea_errors` nei temi predefiniti forniti con Symfony. Quindi dove sono gli errori di un campo textarea che deve essere reso?

La risposta è: nel frammento `field_errors`. Quando Symfony rende gli errori per un tipo textarea, prima cerca un frammento `textarea_errors`, poi cerca un frammento `form_errors`. Ogni tipo di campo ha un tipo *genitore* (il tipo genitore di `textarea` è `text`) e Symfony utilizza il frammento per il tipo del genitore se il frammento di base non esiste.

Quindi, per ignorare gli errori dei *soli* campi `textarea`, copiare il frammento `form_errors`, rinominarlo in `textarea_errors` e personalizzarlo. Per sovrascrivere la resa degli errori predefiniti di *tutti* i campi, copiare e personalizzare direttamente il frammento `form_errors`.

**Tip:** Il tipo "genitore" di ogni tipo di campo è disponibile per ogni tipo di campo in form type reference

## Temi globali per i form

Nell'esempio sopra, è stato utilizzato l'helper `form_theme` (in Twig) per "importare" i frammenti personalizzati *solo* in quel form. Si può anche dire a Symfony di importare personalizzazioni del form nell'intero progetto.

### Twig

Per includere automaticamente i blocchi personalizzati del template `fields.html.twig` creato in precedenza, in *tutti* i template, modificare il file della configurazione dell'applicazione:

Tutti i blocchi all'interno del template `fields.html.twig` vengono ora utilizzati a livello globale per definire l'output del form.

**Personalizzare tutti gli output del form in un singolo file con Twig**

Con Twig, si può anche personalizzare il blocco di un form all'interno del template in cui questa personalizzazione è necessaria:

```
{% extends 'base.html.twig' %}

{# import "_self" as the form theme #}
{% form_theme form _self %}

{# make the form fragment customization #}
{% block form_row %}
    {# custom field row output #}
{% endblock form_row %}

{% block content %}
    {# ... #}

    {{ form_row(form.task) }}
{% endblock %}
```

Il tag `{% form_theme form _self %}` consente ai blocchi del form di essere personalizzati direttamente all'interno del template che utilizzerà tali personalizzazioni. Utilizzare questo metodo per creare velocemente personalizzazioni del form che saranno utilizzate solo in un singolo template.

**Caution:** La funzionalità `{% form_theme form _self %}` funziona *solo* se un template estende un altro. Se un template non estende, occorre far puntare `form_theme` a un template separato.

#### PHP

Per includere automaticamente i template personalizzati dalla cartella `app/Resources/views/Form` creata in precedenza in *tutti* i template, modificare il file con la configurazione dell'applicazione:

Ogni frammento all'interno della cartella `app/Resources/views/Form` è ora usato globalmente per definire l'output del form.

## Protezione da CSRF

CSRF, o Cross-site request forgery, è un metodo mediante il quale un utente malintenzionato cerca di fare inviare inconsapevolmente agli utenti legittimi dati che non vorrebbero inviare. Fortunatamente, gli attacchi CSRF possono essere prevenuti, utilizzando un token CSRF all'interno dei form.

La buona notizia è che, per impostazione predefinita, Symfony integra e convalida i token CSRF automaticamente. Questo significa che è possibile usufruire della protezione CSRF, senza dover far nulla. Infatti, ogni form in questo capitolo sfrutta la protezione CSRF!

La protezione CSRF funziona con l'aggiunta al form di un campo nascosto, il cui nome predefinito è `_token`, che contiene un valore noto solo allo sviluppatore e all'utente. Questo garantisce che proprio l'utente, e non qualcun altro, stia inviando i dati. Symfony valida automaticamente la presenza e l'esattezza di questo token.

Il campo `_token` è un campo nascosto e sarà reso automaticamente se si include la funzione `form_end()` nel template, perché questa assicura che tutti i campi non ancora resi vengano visualizzati.

Il token CSRF può essere personalizzato specificatamente per ciascun form. Per esempio:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TaskType extends AbstractType
{
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class'      => 'AppBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // una chiave univoca per generare il token
            'intention'       => 'task_item',
        ));
    }

    // ...
}
```

Per disabilitare la protezione CSRF, impostare l'opzione `csrf_protection` a `false`. Si può anche personalizzare a livello globale nel progetto. Per ulteriori informazioni, vedere la sezione riferimento della configurazione dei form.

**Note:** L'opzione `intention` è facoltativa, ma migliora notevolmente la sicurezza del token generato, rendendolo diverso per ogni modulo.

**Caution:** I token CSRF sono pensati per essere diversi per ciascun utente. Per questo motivo, occorre cautela nel provare a mettere in cache pagine con form che includano questo tipo di protezione. Per maggiori informazioni, vedere /cookbook/cache/form_csrf_caching.

## Usare un form senza una classe

Nella maggior parte dei casi, un form è legato a un oggetto e i campi del form prendono i loro dati dalle proprietà di tale oggetto. Questo è quanto visto finora in questo capitolo, con la classe *Task*.

A volte, però, si vuole solo usare un form senza classi, per ottenere un array di dati inseriti. Lo si può fare in modo molto facile:

```
// assicurarsi di aver importato lo spazio dei nomi Request all'inizio della classe
use Symfony\Component\HttpFoundation\Request
// ...

public function contactAction(Request $request)
{
    $defaultData = array('message' => 'Type your message here');
    $form = $this->createFormBuilder($defaultData)
        ->add('name', 'text')
        ->add('email', 'email')
        ->add('message', 'textarea')
        ->add('send', 'submit')
        ->getForm();
```

```
    $form->handleRequest($request);

    if ($form->isValid()) {
        // data è un array con "name", "email", e "message" come chiavi
        $data = $form->getData();
    }

    // ... rendere il form
}
```

Per impostazione predefinita, un form ipotizza che si voglia lavorare con array di dati, invece che con oggetti. Ci sono due modi per modificare questo comportamento e legare un form a un oggetto:

1. Passare un oggetto alla creazione del form (come primo parametro di `createFormBuilder` o come secondo parametro di `createForm`);
2. Dichiarare l'opzione `data_class` nel form.

Se *non* si fa nessuna di queste due cose, il form restituirà i dati come array. In questo esempio, poiché `$defaultData` non è un oggetto (e l'opzione `data_class` è omessa), `$form->getData()` restituirà un array.

---

**Tip:** Si può anche accedere ai valori POST ("name", in questo caso) direttamente tramite l'oggetto `Request`, in questo modo:

```
$this->get('request')->request->get('name');
```

Tuttavia, si faccia attenzione che in molti casi l'uso del metodo `getData()` è preferibile, poiché restituisce i dati (solitamente un oggetto) dopo che sono stati manipolati dal sistema dei form.

---

## Aggiungere la validazione

L'ultima parte mancante è la validazione. Solitamente, quando si richiama `$form->isValid()`, l'oggetto viene validato dalla lettura dei vincoli applicati alla classe. Se il form è legato a un oggetto (cioè se si sta usando l'opzione `data_class` o passando un oggetto al form), questo è quasi sempre l'approccio desiderato. Vedere /book/validation per maggiori dettagli. Ma se il form non è legato a un oggetto e invece si sta recuperando un semplice array di dati inviati, come si possono aggiungere vincoli al form?

La risposta è: impostare i vincoli in modo autonomo e passarli al form. L'approccio generale è spiegato meglio nel *capitolo sulla validazione*, ma ecco un breve esempio:

```
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;

$builder
   ->add('firstName', 'text', array(
       'constraints' => new Length(array('min' => 3)),
   ))
   ->add('lastName', 'text', array(
       'constraints' => array(
           new NotBlank(),
           new Length(array('min' => 3)),
       ),
   ))
;
```

**Tip:** Se si usano i gruppi di validazione, occorre fare riferimento al gruppo `Default` quando si crea il form, oppure impostare il gruppo corretto nel vincolo che si sta aggiungendo.

```
new NotBlank(array('groups' => array('create', 'update'))
```

## Considerazioni finali

Ora si è a conoscenza di tutti i mattoni necessari per costruire form complessi e funzionali per la propria applicazione. Quando si costruiscono form, bisogna tenere presente che il primo obiettivo di un form è quello di tradurre i dati da un oggetto (`Task`) a un form HTML in modo che l'utente possa modificare i dati. Il secondo obiettivo di un form è quello di prendere i dati inviati dall'utente e ri-applicarli all'oggetto.

Ci sono altre cose da imparare sul potente mondo dei form, ad esempio come gestire il caricamento di file con Doctrine o come creare un form dove un numero dinamico di sub-form possono essere aggiunti (ad esempio una todo list in cui è possibile continuare ad aggiungere più campi tramite Javascript prima di inviare). Vedere il ricettario per questi argomenti. Inoltre, assicurarsi di basarsi sulla documentazione di riferimento sui tipi di campo, che comprende esempi di come usare ogni tipo di campo e le relative opzioni.

## Saperne di più con il ricettario

- /cookbook/doctrine/file_uploads
- Riferimento del tipo di campo file
- Creare tipi di campo personalizzati
- /cookbook/form/form_customization
- /cookbook/form/dynamic_form_modification
- /cookbook/form/data_transformers
- /cookbook/security/csrf_in_login_form
- /cookbook/cache/form_csrf_caching

CHAPTER 15

# Sicurezza

Il sistema di sicurezza di Symfony è incredibilmente potente, ma può anche essere difficile da configurare. In questo capitolo si vedrà come impostare passo-passo la sicurezza di un'applicazione, dalla configurazione del firewall a come caricare utenti per negare l'accesso e recuperare un oggetto utente. A seconda dei bisogni, a volte la prima configurazione potrebbe essere difficoltosa. Ma, una volta a posto, il sistema di sicurezza di Symfony sarà flessibile e (speriamo) divertente.

Questa guida è divisa in alcune sezioni:

1. Preparazione di `security.yml` (*autenticazione*);
2. Negare l'accesso all'applicazione (*autorizzazione*);
3. Recuperare l'oggetto corrispondente all'utente corrente

Successivamente ci saranno un certo numero di piccole (ma interessanti) sezioni, come *logout* e *codifica delle password*.

## 1) Preparazione di security.yml (autenticazione)

Il sistema di sicurezza è configurato in `app/config/security.yml`. La configurazione predefinita è simile a questa:

La voce `firewalls` è il *cuore* della configurazione della sicurezza. Il firewall `dev` non è importante, serve solo ad assicurarsi che gli strumenti di sviluppo di Symfony, che si trovano sotto URL come `/_profiler` e `/_wdt`, non siano bloccati.

---

**Tip:** Si può anche far corrispondere la richiesta ad altri dettagli (p.e. l'host). Per maggiori informazioni ed esempi, leggere /cookbook/security/firewall_restriction.

---

Tutti gli altri URL saranno gestiti dal firewall `default` (l'assenza della chiave `pattern` vuol dire che corrisponde a *ogni* URL). Si può pensare al firewall come il proprio sistema di sicurezza e quindi solitamente ha senso avere un singolo firewall. Ma questo non vuol dire che ogni URL richieda autenticazione e quindi la voce `anonymous` si

occupa di questo. In effetti, se ora si apre l'homepage, si potrà accedere e si vedrà che si è "autenticati" come `anon.`. Non lasciarsi ingannare dalla parola "Yes" vicino ad "Authenticated", si è ancora un utente anonimo:


Più avanti si vedrà come negare l'accesso ad alcuni URL o controllori.

**Tip:** La sicurezza è *altamente* configurabile e c'è una guida di riferimento alla configurazione della sicurezza, che mostra tutte le opzioni, con spiegazioni aggiuntive.

## A) Configurare il modo in cui gli utenti si autenticano

Il lavoro principale di un firewall è quello di configurare il *modo* in cui gli utenti si autenticheranno. Useranno un form? Http Basic? Il token di un'API? Tutti questi metodi insieme?

Iniziamo con Http Basic (il caro vecchio popup). Per attivarlo, aggiungere la voce `http_basic` nel firewall:

Facile! Per fare una prova, si deve richiedere che un utente sia connesso per poter vedere una pagina. Per rendere le cose interessanti, creare una nuova pagina su `/admin`. Per esempio, se si usano le annotazioni, creare qualcosa come questo:

```
// src/AppBundle/Controller/DefaultController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
     * @Route("/admin")
     */
    public function adminAction()
    {
        return new Response('Pagina admin!');
    }
}
```

Quindi, aggiungere a `security.yml` una voce `access_control` che richieda all'utente di essere connesso per poter accedere a tale URL:

**Note:** La questione `ROLE_ADMIN` e l'accesso negato saranno analizzati più avanti, nella sezione *2) Accesso negato, ruoli e altre autorizzazioni*.

Ottimo! Ora, se si va su `/admin`, si vedrà il popup HTTP Basic:


Ma come si può entrare? Da dove vengono gli utenti?

---

**Tip:** E se invece si volesse un form di login tradizionoale? Nessun problema! Vedere /cookbook/security/form_login_setup. Che altri metodi sono supportati? Vedere riferimento sulla configurazione oppure costruire un proprio.

---

## B) Configurare come vengono caricati gli utenti

Quando si inserisce il proprio nome utente, Symfony deve caricare le informazioni da qualche parte. Questo viene chiamato "fornitore di utenti" ed è compito dello sviluppatore configurarlo. Symfony ha un modo predefinito di caricare utenti dalla base dati, ma si può anche creare il proprio fornitore di utenti.

Il modo più facile (ma anche più limitato) è di configurare Symfony per caricare utenti inseriti direttamente nel file `security.yml`. Questo fornitore è chiamato "in memoria", ma è meglio pensare a esso come fornitore "in configurazione":

Come per i `firewalls`, si possono avere più `providers`, ma probabilmente ne basterà uno solo. Se si ha bisogno di più fornitori, si può configurare il fornitore usato dal firewall, sotto la voce `provider` (p.e. `provider: in_memory`).

Provare a entrare con nome utente `admin` e password `kitten`. Si dovrebbe vedere un errore!

> No encoder has been configured for account "SymfonyComponentSecurityCoreUserUser"

Per risolvere, aggiungere una chiave `encoders`:

I fornitori di utenti caricano le informazioni dell'utente e le inseriscono in un oggetto `User`. Se si caricano utenti dalla base dati o da altre sorgenti, si userà una propria classe personalizzata. Se invece si usa il fornitore "in memoria", esso restituirà un oggetto `Symfony\Component\Security\Core\User\User`.

Qualunque sia la classe di `User`, si deve dire a Symfony quale algoritmo sia stato usato per codificare le password. In questo caso, le password sono in chiaro, ma tra un attimo faremo in modo di usare `bcrypt`.

Se ora si aggiorna, ci si troverà dentro! La barra di debug del web fornirà informazioni sul nome dell'utente e sui suoi ruoli:


Poiché questo URL richiede `ROLE_ADMIN`, se si prova a entrare come `ryan` ci si vedrà negato l'accesso. Lo vedremo più avanti (*Proteggere schemi di URL (access_control)*).

#### Caricare utenti dalla base dati

Se si vogliono caricare gli utenti usando l'ORM di Doctrine, è facile! Vedere /cookbook/security/entity_provider per tutti i dettagli.

### C) Codifica delle password

Che gli utenti siano dentro a `security.yml`, in una base dati o da qualsiasi altra parte, se ne vorranno codificare le password. Il miglior algoritmo da usare è `bcrypt`:

Ovviamente, sarà ora necessario codificare le password con tale algoritmo. Per gli utenti inseriti a mano, si può usare uno strumento online, che restituirà qualcosa del genere:

Tutto funzionerà come prima. Ma se si hanno utenti dinamici (p.e. da base dati), come si fa a codificare ogni password prima dell'inserimento? Nessun problema, vedere *Codifica dinamica di una password* per i dettagli.

---

**Tip:** Gli algoritmi supportati dipendono dalla versione di PHP, ma includono gli algoritmi restituiti dalla funzione **:phpfunction:'hash_algos'**, più alcuni altri (come bcrypt). Vedere la voce `encoders` nella sezione riferimento della sicurezza per degli esempi.

Si possono anche usare algoritmi differenti per singolo utente. Vedere /cookbook/security/named_encoders per maggiori dettagli.

---

### D) Configurazione conclusa!

Congratulazioni! Ora di dispone un sistema di autenticazione funzionante, che usa Http Basic e carica utenti dal file `security.yml`.

I prossimi passi possono variare:

- Configurare un modo diverso per il login, come un *form di login* o qualcosa di completamente personalizzato;
- Caricare utenti da un'altra sorgente, come la base dati o un'altra sorgente;
- Imparare come negare l'accesso, caricare l'oggetto `User` e trattare con i ruoli nella sezione *autorizzazione*.

## 2) Accesso negato, ruoli e altre autorizzazioni

Ora gli utenti possono accedere all'applicazione usando `http_basic` o un altro metodo. Ottimo! Ora, occorre imparare come negare l'accesso e lavorare con l'oggetto `User`. Questo processo prende il nome di **autorizzazione** e spetta a esso decidere se un utente possa accedere a una determinata risorsa (un URL, un oggetto del modello, una chiamata a un metodo, ...).

Il processo di autorizzazione ha due lati:

1. L'utente riceve uno specifico insieme di ruoli, quando entra (p.e. `ROLE_ADMIN`).
2. Si aggiunge codice in modo che una risorsa (come un URL o un controllore) richieda uno specifico "attributo" (solitamente un ruolo, come `ROLE_ADMIN`) per potervi accedere.

---

**Tip:** Oltre ai ruoli (come `ROLE_ADMIN`), si può proteggere una risorsa tramite altri attributi/stringhe (come `EDIT`) e usare i votanti o il sistema ACL di Symfony per dar loro un significato. Questo può essere utile nel caso serva

verificare se l'utente A possa modificare un oggetto B (p.e. un prodotto con un determinato id). Vedere *Access Control List (ACL): proteggere singoli oggetti della base dati*.

## Ruoli

Quando un utente entra, riceve un insieme di ruoli (p.e. `ROLE_ADMIN`). Nell'esempio precedente, tali ruoli sono scritti a mano in `security.yml`. Se si caricano utenti dalla base dati, probabilmente saranno memorizzati in una colonna della tabella.

**Caution:** Tutti i ruoli assegnati **devono** avere il prefisso `ROLE_`. In caso contrario, non possono essere gestiti dal sistema di sicurezza di Symfony (a meno che non si faccia qualcosa di avanzato, assegnare un ruolo come `PIPPO` a un utente e poi verificare `PIPPO`, come descritto *successivamente* non funzionerà).

I ruoli sono semplici e sono di base stringhe inventate e usate come necessario. Per esempio, per poter iniziare a limitare l'accesso alla sezione amministrativa di un blog, si può proteggere tale sezione usando un ruolo `ROLE_BLOG_ADMIN`. Non occorre definire tale ruolo in altri posti, basta iniziare a usarlo.

**Tip:** Assicurarsi che ciascun utente abbia almeno *un* ruolo, altrimenti sembrerà che l'utente non sia autenticato. Una convenzione tipica consiste nell'assegnare a *ogni* utente `ROLE_USER`.

Si può anche specificare una *gerarchia di ruoli*, in cui determinati ruoli ne hanno automaticamente anche altri.

## Aggiungere codice per negare l'accesso

Ci sono **due** modi per negare accesso a qualcosa:

1. *access_control in security.yml* consente di proteggere schemi di URL (p.e. `/admin/*`). È facile, ma meno flessibile;
2. *nel codice, tramite il servizio security.context*.

### Proteggere schemi di URL (access_control)

Il modo più semplice per proteggere parti di un'applicazione è proteggere un intero schema di URL. L'abbiamo visto in precedenza, quando abbiamo richiesto che ogni URL corrispondente all'espressione regolare `^/admin` richieda `ROLE_ADMIN`:

Questo va benissimo per proteggere intere sezioni, ma si potrebbero anche voler *proteggere singoli controllori*.

Si possono definire quanti schemi di URL si vuole, ciascuno con un'espressione regolare. Tuttavia, solo **uno** di questi avrà una corrispondenza. Symfony inizierà cercando dalla cima e si fermerà non appena troverà una voce di `access_control` che corrisponda all'URL.

Aggiungendo un `^` iniziale, solo gli URL *che iniziano* con lo schema corrisponderanno. Per esempio, un percorso `/admin` (senza `^`) corrisponderebbe ad `/admin/pippo`, ma anche a URL come `/pippo/admin`.

**Capire come funziona `access_control`**

La sezione `access_control` è molto potente, ma può anche essere pericolosa (perché si tratta di sicurezza), se non ci comprende *come* funzioni. Oltre all'URL, `access_control` può corrispondere un indirizzo IP, un nome di host e metodi HTTP. Può anche essere usato per rinviare un utente alla versione `https` di uno schema di URL.

Per approfondire questi argomenti, vedere /cookbook/security/access_control.

### Proteggere controllori e altro codice

Si può negare accesso da dentro un controllore:

```
// ...

public function helloAction($name)
{
    // Il secondo parametro si usa per specificare l'oggetto su cui si testa il ruolo
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Non si può accedere a questa␣
↪pagina!');

    // Vecchio modo :
    // if (false === $this->get('security.authorization_checker')->isGranted('ROLE_
↪ADMIN')) {
    //     throw $this->createAccessDeniedException('Non si può accedere a questa␣
↪pagina!');
    // }

    // ...
}
```

New in version 2.6: Il metodo `denyAccessUnlessGranted()` è stato introdotto in Symfony 2.6. In precedenza (ma anche ora), si poteva verificare l'accesso direttamente e sollevare `AccessDeniedException`, come mostrato nell'esempio preedente).

New in version 2.6: Il servizio `security.authorization_checker` è stato introdotto in Symfony 2.6. Prima di Symfony 2.6, si doveva usare il metodo `isGranted()` del servizio `security.context`.

Il metodo **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::createAccessDeniedException'** crea uno speciale oggetto `Symfony\Component\Security\Core\Exception\AccessDeniedException`, che alla fine lancia una risposta HTTP 403 in Symfony.

Ecco fatto! Se l'utente non è ancora loggato, gli sarà richiesto il login (p.e. rinviato alla pagina di login). Se invece *è* loggato, gli sarà mostrata una pagina di errore 403 (che si può personalizzare). Grazie a SensioFrameworkExtraBundle, si può anche proteggere un controllore tramite annotazioni:

```
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Security("has_role('ROLE_ADMIN')")
 */
public function helloAction($name)
{
    // ...
}
```

Per maggiori informazioni, vedere la documentazione di FrameworkExtraBundle.

### Controllo degli accessi nei template

Se si vuole verificare in un template che l'utente corrente abbia un ruolo, usare la funzione aiutante predefinita:

Se si usa questa funzione *non* essendo dietro a un firewall, sarà lanciata un'ecceezione. È quindi sempre una buona idea avere almeno un firewall principale, che copra tutti gli URL (come mostrato in questo capitolo).

**Caution:** Prestare attenzione nel layout e nelle pagine di errore! A causa di alcuni dettagli interno di Symfony, per evitare di rompere le pagine di errore in ambiente `prod`, verificare prima se sia definito `app.user`:

```
{% if app.user and is_granted('ROLE_ADMIN') %}
```

### Proteggere altri servizi

In Symfony, ogni cosa può essere protetta facendo qualcosa di simile a questo. Per esempio, si supponga di avere un servizio (cioè una classe PHP), il cui compito è inviare email. Si può restringere l'uso di questa classe, non importa dove venga usata, solo ad alcuni utenti.

Per maggiori informazioni, vedere /cookbook/security/securing_services.

## Verificare se un utente sia connesso (IS_AUTHENTICATED_FULLY)

Finora, i controlli sugli accessi sono stati basati su ruoli, stringhe che iniziano con `ROLE_` e assegnate agli utenti. Se invece si vuole *solo* verificare se un utente sia connesso (senza curarsi dei ruoli), si può usare `IS_AUTHENTICATED_FULLY`:

```
// ...

public function helloAction($name)
{
    if (!$this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_
↪FULLY')) {
        throw $this->createAccessDeniedException();
    }

    // ...
}
```

**Tip:** Si può usare questo metodo anche in `access_control`.

`IS_AUTHENTICATED_FULLY` non è un ruolo, ma si comporta come tale ed è assegnato a ciascun utente che sia sia connesso. In effetti, ci sono tre attributi speciali di questo tipo:

- `IS_AUTHENTICATED_REMEMBERED`: Assegnato a *tutti* gli utenti connessi, anche se si sono connessi tramite un cookie "ricordami". Anche se non si usa la funzionalità "ricordami", lo si può usare per verificare se l'utente sia connesso.
- `IS_AUTHENTICATED_FULLY`: Simile a `IS_AUTHENTICATED_REMEMBERED`, ma più forte. Gli utenti connessi tramite un cookie "ricordami" avranno `IS_AUTHENTICATED_REMEMBERED`, ma non `IS_AUTHENTICATED_FULLY`.
- `IS_AUTHENTICATED_ANONYMOUSLY`: Assegnato a *tutti* gli utenti (anche quelli anonimi). Utile per mettere URL in una *lista bianca* per garantire accesso, alcuni dettagli sono in /cookbook/security/access_control.

Si possono anche usare espressioni nei template:

Per maggiori dettagli su espressioni e sicurezza, vedere book-security-expressions.

### Access Control List (ACL): proteggere singoli oggetti della base dati

Si immagini di progettare un blog in cui gli utenti possono commentare i post. Si vuole anche che un utente sia in grado di modificare i propri commenti, ma non quelli di altri utenti. Inoltre, come utente amministratore, si vuole essere in grado di modificare *tutti* i commenti.

Per la realizzazione, si hanno due opzioni:

- I votanti consentono di usare logica di business (p.e. l'utente può modificare i suoi commenti perché ne è il creatore) per stabilire l'accesso. Probabilmente si userà questa opzione, è abbastanza flessibile per risolvere la situazione.
- Le ACL consentono di creare una struttura di base dati in cui si può assegnare *qualsiasi* accesso (p.e. EDIT, VIEW) a *quasliasi* utente su *qualsiasi* oggetto del sistema. Usarla se si ha bisogno che l'utente amministratore possa garantire accessi personalizzati nel sistema, tramite una qualche interfaccia di amministrazione.

In entrambi i casi, occorre comunque negare l'accesso usando metodi simili a quelli visti in precedenza.

## Recuperare l'oggetto utente

New in version 2.6: Il servizio `security.token_storage` è stato introdotti in Symfony 2.6. Prima di Symfony 2.6, si doveva usare il metodo `getToken()` del servizio `security.context`.

Dopo l'autenticazione, si può accedere all'oggetto `User` dell'uente corrente tramite il servizio `security.context`. Da dentro un controllore, Sarà una cosa simile:

```
public function indexAction()
{
    if (!$this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_
↪FULLY')) {
        throw $this->createAccessDeniedException();
    }

    $user = $this->getUser();

    // il precedente è una scorciatoia per questo
    $user = $this->get('security.token_storage')->getToken()->getUser();
}
```

**Tip:** L'oggetto e la classe dell'utente dipenderanno dal proprio *fornitore di utenti*.

Ora si possono chiamare i metodi desiderati sul *proprio* oggetto utente. Per esempio, se il proprio oggetto utente ha un metodo `getFirstName()`, lo si può usare:

```
use Symfony\Component\HttpFoundation\Response;

public function indexAction()
{
    // ...
```

```
    return new Response('Ciao '.$user->getFirstName());
}
```

### Verificare sempre se l'utente è connesso

È importante verificare prima se l'utente sia autenticato. Se non lo è, `$user` sarà `null` oppure la stringa `anon.`. Come? Esatto, c'è una stranezza. Se non si è leggati, l'utente è tecnicamente la stringa `anon.`, anche se la scorciatoia `getUser()` del controllore la converte in `null` per convenienza.

Il punto è questo: verificare sempre se l'utente sia connesso, prima di usare l'oggetto `User` e usare il metodo `isGranted` (o *access_control*) per farlo:

```
// Usare questo per vedere se l'utente sia connesso
if (!$this->get('security.context')->isGranted('IS_AUTHENTICATED_FULLY')) {
    throw new AccessDeniedException();
}

// Non verificare mai l'oggetto User per verdere se l'utente sia connesso
if ($this->getUser()) {

}
```

### Recuperare l'utente in un template

In un template Twig, si può accedere all'oggetto tramite **:ref:'app.user <reference-twig-global-app>'_**:

## Logout

Solitamente, si desidera che gli utenti possano eseguire un logout. Per fortuna, il firewall può gestirlo automaticamente, se si attiva il parametro `logout` nella configurazione:

Quindi, si deve creare una rotta per tale URL (non serve invece un controllore):

Ecco fatto! Se l'utente va su `/logout` (o sull'URL configurato in `path`), Symfony disconnetterà l'utente corrente.

Una volta eseguito il logout, l'utente sarà rinviato al percorso definito nel parametro `target` (p.e. su `homepage`).

---

**Tip:** Se si ha bisogno di fare qualcosa d'altro dopo il logout, si può specificare un gestore di logout, aggiungendo la voce `success_handler` e puntandola a un servizio, che implementi `Symfony\Component\Security\Http\Logout\LogoutSuccessHandlerInterface`. Vedere Security Configuration Reference.

---

## Codifica dinamica di una password

Se, per esempio, gli utenti sono memorizzati in una base dati, occorrerà codificare le loro password, prima di inserirle. Non importa quale algoritmo sia configurato per l'oggetto utente, l'hash della password può sempre essere determinato nel modo seguente, in un controllore:

```
// qualunque sia il *proprio* oggetto User
$user = new AppBundle\Entity\User();
$plainPassword = 'ryanpass';
$encoder = $this->container->get('security.password_encoder');
$encoded = $encoder->encodePassword($user, $plainPassword);

$user->setPassword($encoded);
```

New in version 2.6: Il servizio `security.password_encoder` è stato introdotto in Symfony 2.6.

Per poter funzionare, assicurarsi di avere un codificatore per la classe utente (p.e. `AppBundle\Entity\User`) configurato sotto la voce `encoders` in `app/config/security.yml`.

L'oggetto `$encoder` ha anche un metodo `isPasswordValid`, che accetta l'oggetto `User` come primo parametro e la password in chiaro, da verificare, come secondo parametro.

**Caution:** Quando si consente a un utente di inviare una password in chiaro (p.e. un form di registrazione, un form di cambio password), si *deve* avere una validazione che garantisca una lunghezza massima della password di 4096 caratteri. Maggiori dettagli su implementare una semplice form di registrazione.

## Gerarchia dei ruoli

Invece di associare molti ruoli agli utenti, si possono definire regole di ereditarietà dei ruoli, creando una gerarchia:

In questa configurazione, gli utenti con il ruolo `ROLE_ADMIN` avranno anche il ruolo `ROLE_USER`. Il ruolo `ROLE_SUPER_ADMIN` ha `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` e `ROLE_USER` (ereditato da `ROLE_ADMIN`).

## Autenticazione senza stato

Symfony si appoggia a un cookie (la sessione) per persistere il contesto di sicurezza dell'utente. Se però si usano certificati o autenticazione HTTP, per esempio, non serve persistenza, perché le credenziali sono disponibili a ogni richiesta. In tal caso, e non si ha bisogno di memorizzare altro tra una richiesta e l'altro, si può attivare l'autenticazione senza stato (che vuol dire che Symfony non creerà alcun cookie):

**Note:** Se si usa un form di login, Symfony creerà un cookie anche se si imposta `stateless` a `true`.

### Verificare vulnerabilità note nelle dipendenze

New in version 2.5: Il comando `security:check` è stato introdotto in Symfony 2.5. Questo comando è incluso in `SensioDistributionBundle`, bundle che va registrato nell'applicazione per consentire l'utilizzo del comando stesso.

Quando si hanno molte dipendenze in progetti Symfony, alcune potrebbero contenere delle vulnerabilità. Per questo motivo, Symfony include un comando chiamato `security:check`, che verifica il file `composer.lock` e trova eventuali vulnerabilità nelle dipendenze installate:

```
$ php app/console security:check
```

Una buona pratica di sicurezza consiste nell'eseguire regolarmente questo comando, per poter aggiornare o sostituire dipendenze compromesse il prima possibile. Internamente, questo comando usa la base dati degli avvisi di sicurezza pubblicato dall'organizzazione FriendsOfPHP.

**Tip:** Il comando `security:check` esce con un codice diverso da zero, se alcune dipendenze sono afflitte da problemi noti di sicurezza. Quindi, si può facilmente integrare in un processo di build.

## Considerazioni finali

Ora sappiamo più di qualche base sulla sicurezza. Le parti più difficili coinvolgono i requisiti personalizzati: una strategia di autenticazione personalizzata (p.e. token API), logica di autorizzazione complessa e molte altre cose (perché la sicurezza è complessa!).

Fortunatamente, ci sono molte ricette sulla sicurezza, che descrivono molte di queste situazioni. Inoltre, vedere la sezione di riferimento della sicurezza. Molte opzioni non hanno dettagli specifici, ma analizzare l'intero albero di configurazione potrebbe essere utile.

Buona fortuna!

## Saperne di più con il ricettario

- Forzare HTTP/HTTPS
- Impersonare un utente
- /cookbook/security/voters_data_permission
- Access Control List (ACL)
- /cookbook/security/remember_me
- /cookbook/security/multiple_user_providers

CHAPTER 16

# Cache HTTP

Le applicazioni web sono dinamiche. Non importa quanto efficiente possa essere un'applicazione, ogni richiesta conterrà sempre overhead rispetto a quando si serve un file statico.

Per la maggior parte delle applicazioni, questo non è un problema. Symfony è molto veloce e, a meno che non si stia facendo qualcosa di veramente molto pesante, ogni richiesta sarà gestita rapidamente, senza stressare troppo il server.

Man mano che il sito cresce, però, quell'overhead può diventare un problema. Il processo normalmente seguito a ogni richiesta andrebbe fatto una volta sola. Questo è proprio lo scopo che si prefigge la cache.

## La cache sulle spalle dei giganti

Il modo più efficace per migliorare le prestazioni di un'applicazione è mettere in cache l'intero output di una pagina e quindi aggirare interamente l'applicazione a ogni richiesta successiva. Ovviamente, questo non è sempre possibile per siti altamente dinamici, oppure sì? In questo capitolo, mostreremo come funziona il sistema di cache di Symfony e perché pensiamo che sia il miglior approccio possibile.

Il sistema di cache di Symfony è diverso, perché si appoggia sulla semplicità e sulla potenza della cache HTTP, definita nelle specifiche HTTP. Invece di inventare un altro metodo di cache, Symfony abbraccia lo standard che definisce la comunicazione di base sul web. Una volta capiti i fondamenti dei modelli di validazione e scadenza della cache HTTP, si sarà in grado di padroneggiare il sistema di cache di Symfony.

Per poter imparare come funziona la cache in Symfony, procederemo in quattro passi:

1. **Passo 1**: Un *gateway cache*, o reverse proxy, è un livello indipendente che si situa davanti all'applicazione. Il reverse proxy mette in cache le risposte non appena sono restituite dall'applicazione e risponde alle richieste con risposte in cache, prima che arrivino all'applicazione. Symfony fornisce il suo reverse proxy, ma se ne può usare uno qualsiasi.
2. **Passo 2**: Gli header di *cache HTTP* sono usati per comunicare col gateway cache e con ogni altra cache tra l'applicazione e il client. Symfony fornisce impostazioni predefinite appropriate e una potente interfaccia per interagire con gli header di cache.
3. **Passo 3**: La *scadenza e la validazione* HTTP sono due modelli usati per determinare se il contenuto in cache è *fresco* (può essere riusato dalla cache) o *vecchio* (andrebbe rigenerato dall'applicazione):

4. **Passo 4**: Gli *Edge Side Include* (ESI) consentono alla cache HTTP di essere usata per mettere in cache frammenti di pagine (anche frammenti annidati) in modo indipendente. Con ESI, si può anche mettere in cache una pagina intera per 60 minuti, ma una barra laterale interna per soli 5 minuti.

Poiché la cache con HTTP non è esclusiva di Symfony, esistono già molti articoli a riguardo. Se si è nuovi con la cache HTTP, raccomandiamo *caldamente* l'articolo di Ryan Tomayko Things Caches Do. Un'altra risorsa importante è il Cache Tutorial di Mark Nottingham.

## Cache con gateway cache

Quando si usa la cache con HTTP, la *cache* è completamente separata dall'applicazione e si trova in mezzo tra applicazione e client che effettua la richiesta.

Il compito della cache è accettare le richieste dal client e passarle all'applicazione. La cache riceverà anche risposte dall'applicazione e le girerà al client. La cache è un "uomo in mezzo" nella comunicazione richiesta-risposta tra il client e l'applicazione.

Lungo la via, la cache memorizzerà ogni risposta ritenuta "cacheable" (vedere *Introduzione alla cache HTTP*). Se la stessa risorsa viene richiesta nuovamente, la cache invia la risposta in cache al client, ignorando completamente l'applicazione.

Questo tipo di cache è nota come HTTP gateway cache e ne esistono diverse, come Varnish, Squid in modalità reverse proxy e il reverse proxy di Symfony.

### Tipi di cache

Ma il gateway cache non è l'unico tipo di cache. Infatti, gli header HTTP di cache inviati dall'applicazione sono analizzati e interpretati da tre diversi tipi di cache:

- *Cache del browser*: Ogni browser ha la sua cache locale, usata principalmente quando si clicca sul pulsante "indietro" per immagini e altre risorse. La cache del browser è una cache *privata*, perché le risorse in cache non sono condivise con nessun altro.
- *Proxy cache*: Un proxy è una cache *condivisa*, perché molte persone possono stare dietro a un singolo proxy. Solitamente si trova nelle grandi aziende e negli ISP, per ridurre la latenza e il traffico di rete.
- *Gateway cache*: Come il proxy, anche questa è una cache *condivisa*, ma dalla parte del server. Installata dai sistemisti di rete, rende i siti più scalabili, affidabili e performanti.

**Tip:** Le gateway cache sono a volte chiamate reverse proxy cache, cache surrogate o anche acceleratori HTTP.

**Note:** I significati di cache *privata* e *condivisa* saranno più chiari quando si parlerà di mettere in cache risposte che contengono contenuti specifici per un singolo utente (p.e. informazioni sull'account).

Ogni risposta dall'applicazione probabilmente attraverserà una o più cache dei primi due tipi. Queste cache sono fuori dal nostro controllo, ma seguono le indicazioni di cache HTTP impostate nella risposta.

### Il reverse proxy di Symfony

Symfony ha un suo reverse proxy (detto anche gateway cache) scritto in PHP. Abilitandolo, le risposte in cache dall'applicazione inizieranno a essere messe in cache. L'installazione è altrettanto facile. Ogni una applicazione Symfony ha la cache già configurata in `AppCache`, che estende `AppKernel`. Il kernel della cache *è* il reverse proxy.

Per abilitare la cache, modificare il codice di un front controller, per usare il kernel della cache:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// inserisce AppKernel all'interno di AppCache
$kernel = new AppCache($kernel);

$request = Request::createFromGlobals();

$response = $kernel->handle($request);
$response->send();

$kernel->terminate($request, $response);
```

Il kernel della cache agirà immediatamente da reverse proxy, mettendo in cache le risposte dell'applicazione e restituendole al client.

**Caution:** Se si usa l'opzione framework.http_method_override per leggere il metodo HTTP da un parametro `_method`, vedere il collegamento precedente per un trucco da applicare.

---

**Tip:** Il kernel della cache ha uno speciale metodo `getLog()`, che restituisce una rappresentazione in stringa di ciò che avviene a livello di cache. Nell'ambiente di sviluppo, lo si può usare per il debug e la verifica della strategia di cache:

```
error_log($kernel->getLog());
```

---

L'oggetto `AppCache` una una configurazione predefinita adeguata, ma può essere regolato tramite un insieme di opzioni impostabili sovrascrivendo il metodo **:method:'Symfony\\Bundle\\FrameworkBundle\\HttpCache\\HttpCache::getOptions'**:

```
// app/AppCache.php
use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
{
    protected function getOptions()
    {
        return array(
            'debug'                  => false,
            'default_ttl'            => 0,
            'private_headers'        => array('Authorization', 'Cookie'),
            'allow_reload'           => false,
            'allow_revalidate'       => false,
            'stale_while_revalidate' => 2,
            'stale_if_error'         => 60,
        );
    }
```

```
}
```

**Tip:** A meno che non sia sovrascritta in `getOptions()`, l'opzione `debug` sarà impostata automaticamente al valore di debug di `AppKernel` circostante.

Ecco una lista delle opzioni principali:

**`default_ttl`** Il numero di secondi per cui un elemento in cache va considerato fresco, quando nessuna informazione esplicita sulla freschezza viene fornita in una risposta. Header espliciti `Cache-Control` o `Expires` sovrascrivono questo valore (predefinito: `0`);

**`private_headers`** Insieme di header di richiesta che fanno scattare il comportamento "privato" `Cache-Control` sulle risposte che non stabiliscono esplicitamente il loro stato di `public` o `private`, tramite una direttiva `Cache-Control`. (predefinito: `Authorization` e `Cookie`);

**`allow_reload`** Specifica se il client possa forzare un ricaricamento della cache includendo una direttiva `Cache-Control` "no-cache" nella richiesta. Impostare a `true` per aderire alla RFC 2616 (predefinito: `false`);

**`allow_revalidate`** Specifica se il client possa forzare una rivalidazione della cache includendo una direttiva `Cache-Control` "max-age=0" nella richiesta. Impostare a `true` per aderire alla RFC 2616 (predefinito: false);

**`stale_while_revalidate`** Specifica il numero predefinito di secondi (la granularità è il secondo, perché la precisione del TTL della risposta è un secondo) durante il quale la cache può restituire immediatamente una risposta vecchia mentre si rivalida in background (predefinito: `2`); questa impostazione è sovrascritta dall'estensione `stale-while-revalidate` `Cache-Control` di HTTP (vedere RFC 5861);

**`stale_if_error`** Specifica il numero predefinito di secondi (la granularità è il secondo) durante il quale la cache può servire una risposta vecchia quando si incontra un errore (predefinito: `60`). Questa impostazione è sovrascritta dall'estensione `stale-if-error` `Cache-Control` di HTTP (vedere RFC 5861).

Se `debug` è `true`, Symfony aggiunge automaticamente un header `X-Symfony-Cache` alla risposta, con dentro informazioni utili su hit e miss della cache.

**Cambiare da un reverse proxy a un altro**

Il reverse proxy di Symfony è un grande strumento da usare durante lo sviluppo di un sito oppure quando il deploy di un sito è su un host condiviso, dove non si può installare altro che codice PHP. Ma, essendo scritto in PHP, non può essere veloce quando un proxy scritto in C. Per questo si raccomanda caldamente di usare Varnish o Squid sul server di produzione, se possibile. La buona notizia è che il cambio da un proxy a un altro è facile e trasparente, non implicando alcuna modifica al codice dell'applicazione. Si può iniziare semplicemente con il reverse proxy di Symfony e aggiornare successivamente a Varnish, quando il traffico aumenta.

Per maggiori informazioni sull'uso di Varnish con Symfony, vedere la ricetta usare Varnish.

**Note:** Le prestazioni del reverse proxy di Symfony non dipendono dalla complessità dell'applicazione. Questo perché il kernel dell'applicazione parte solo quando ha una richiesta a cui deve essere rigirato.

# Introduzione alla cache HTTP

Per sfruttare i livelli di cache disponibili, un'applicazione deve poter comunicare quale risposta può essere messa in cache e le regole che stabiliscono quando e come tale cache debba essere considerata vecchia. Lo si può fare impostando gli header di cache HTTP nella risposta.

---

**Tip:** Si tenga a mente che "HTTP" non è altro che il linguaggio (un semplice linguaggio testuale) usato dai client web (p.e. i browser) e i server web per comunicare tra loro. La cache HTTP è la parte di tale linguaggio che consente a client e server di scambiarsi informazioni riguardo alla cache.

---

HTTP specifica quattro header di cache per la risposta di cui ci occupiamo:

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

L'header più importante e versatile è l'header `Cache-Control`, che in realtà è un insieme di varie informazioni sulla cache.

---

**Note:** Ciascun header sarà spiegato in dettaglio nella sezione *Scadenza e validazione HTTP*.

---

## L'header Cache-Control

L'header `Cache-Control` è unico, perché non contiene una, ma vari pezzi di informazione sulla possibilità di una risposta di essere messa in cache. Ogni pezzo di informazione è separato da una virgola:

```
Cache-Control: private, max-age=0, must-revalidate

Cache-Control: max-age=3600, must-revalidate
```

Symfony fornisce un'astrazione sull'header `Cache-Control`, per rendere la sua creazione più gestibile:

```
// ...

use Symfony\Component\HttpFoundation\Response;

$response = new Response();

// segna la risposta come pubblica o privata
$response->setPublic();
$response->setPrivate();

// imposta max age privata o condivisa
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

// imposta una direttiva personalizzata Cache-Control
$response->headers->addCacheControlDirective('must-revalidate', true);
```

**Tip:** Se occorre impostare header di cache per molte azioni diverse, si potrebbe usare FOSHttpCacheBundle. Questo bundle fornisce un modo per definire header di cache in base allo schema dell'URL e ad altre proprietà della richiesta.

## Risposte pubbliche e risposte private

Sia la gateway cache che la proxy cache sono considerate cache "condivise", perché il contenuto della cache è condiviso da più di un utente. Se una risposta specifica per un utente venisse per errore inserita in una cache condivisa, potrebbe successivamente essere restituita a diversi altri utenti. Si immagini se delle informazioni su un account venissero messe in cache e poi restituite a ogni utente successivo che richiede la sua pagina dell'account!

Per gestire questa situazione, ogni risposta può essere impostata a pubblica o privata:

***pubblica*** Indica che la risposta può essere messa in cache sia da che private che da cache condivise;

***privata*** Indica che tutta la risposta, o una sua parte, è per un singolo utente e quindi non deve essere messa in una cache condivisa.

Symfony è conservativo e ha come predefinita una risposta privata. Per sfruttare le cache condivise (come il reverse proxy di Symfony), la risposta deve essere impostata esplicitamente come pubblica.

## Metodi sicuri

La cache HTTP funziona solo per metodi HTTP "sicuri" (come GET e HEAD). Essere sicuri vuol dire che lo stato dell'applicazione sul server non cambia mai quando si serve la richiesta (si può, certamente, memorizzare un'informazione sui log, mettere in cache dati, eccetera). Questo ha due conseguenze molto ragionevoli:

- Non si dovrebbe *mai* cambiare lo stato dell'applicazione quando si risponde a una richiesta GET o HEAD. Anche se non si usa una gateway cache, la presenza di proxy cache vuol dire che ogni richiesta GET o HEAD potrebbe arrivare al server, ma potrebbe anche non arrivare.
- Non aspettarsi la cache dei metodi PUT, POST o DELETE. Questi metodi sono fatti per essere usati quando si cambia lo stato dell'applicazione (p.e. si cancella un post di un blog). Metterli in cache impedirebbe ad alcune richieste di arrivare all'applicazione o di modificarla.

## Regole e valori predefiniti della cache

HTTP 1.1 consente per impostazione predefinita la cache di tutto, a meno che non ci sia un header esplicito `Cache-Control`. In pratica, la maggior parte delle cache non fanno nulla quando la richiesta ha un cookie, un header di autorizzazione, usa un metodo non sicuro (PUT, POST, DELETE) o quando la risposta ha un codice di stato di rinvio.

Symfony imposta automaticamente un header `Cache-Control` conservativo, quando nessun header è impostato dallo sviluppatore, seguendo queste regole:

- Se non è definito nessun header di cache (`Cache-Control`, `Expires`, `ETag` o `Last-Modified`), `Cache-Control` è impostato a `no-cache`, il che vuol dire che la risposta non sarà messa in cache;
- Se `Cache-Control` è vuoto (ma uno degli altri header di cache è presente), il suo valore è impostato a `private, must-revalidate`;
- Se invece almeno una direttiva `Cache-Control` è impostata e nessuna direttiva `public` o `private` è stata aggiunta esplicitamente, Symfony aggiunge automaticamente la direttiva `private` (tranne quando è impostato `s-maxage`).

# Scadenza e validazione HTTP

Le specifiche HTTP definiscono due modelli di cache:

- Con il modello a scadenza, si specifica semplicemente quanto a lungo una risposta debba essere considerata "fresca", includendo un header `Cache-Control` e/o uno `Expires`. Le cache che capiscono la scadenza non faranno di nuovo la stessa richiesta finché la versione in cache non raggiunge la sua scadenza e diventa "vecchia".
- Quando le pagine sono molto dinamiche (cioè quando la loro rappresentazione varia spesso), il modello a validazione è spesso necessario. Con questo modello, la cache memorizza la risposta, ma chiede al serve a ogni richiesta se la risposta in cache sia ancora valida o meno. L'applicazione usa un identificatore univoco per la risposta (l'header `Etag`) e/o un timestamp (come l'header `Last-Modified`) per verificare se la pagina sia cambiata da quanto è stata messa in cache.

Lo scopo di entrambi i modelli è quello di non generare mai la stessa risposta due volte, appoggiandosi a una cache per memorizzare e restituire risposte "fresche". Per ottenere tempi di cache lunghi, ma fornire comunque contenuti aggiornati immediatamente, a volte si usa l'*invalidazione della cache*.

**Leggere le specifiche HTTP**

Le specifiche HTTP definiscono un linguaggio semplice, ma potente, in cui client e server possono comunicare. Come sviluppatori web, il modello richiesta-risposta delle specifiche domina il nostro lavoro. Sfortunatamente, il documento delle specifiche, la RFC 2616, può risultare di difficile lettura.

C'è uno sforzo in atto (HTTP Bis) per riscrivere la RFC 2616. Non descrive una nuova versione di HTTP, ma per lo più chiarisce le specifiche HTTP originali. Anche l'organizzazione è migliore, essendo le specifiche separate in sette parti; tutto ciò che riguarda la cache HTTP si trova in due parti dedicate (P4 - Richieste condizionali e P6 - Cache: Browser e cache intermedie).

Come sviluppatori web, dovremmo leggere tutti le specifiche. Possiedono un chiarezza e una potenza, anche dopo oltre dieci anni dalla creazione, inestimabili. Non ci si spaventi dalle apparenze delle specifiche, il contenuto è molto più bello della copertina.

## Scadenza

Il modello a scadenza è il più efficiente e il più chiaro dei due modelli di cache e andrebbe usato ogni volta che è possibile. Quando una risposta è messa in cache con una scadenza, la cache memorizzerà la risposta e la restituirà direttamente, senza arrivare all'applicazione, finché non scade.

Il modello a scadenza può essere implementato con l'uso di due header HTTP, quasi identici: `Expires` o `Cache-Control`.

## Scadenza con l'header `Expires`

Secondo le specifiche HTTP, "l'header `Expires` dà la data e l'ora dopo la quale la risposta è considerata vecchia". L'header `Expires` può essere impostato con il metodo `setExpires()` di `Response`. Accetta un'istanza di `DateTime` come parametro:

```
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

Il risultante header HTTP sarà simile a questo:

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

**Note:** Il metodo `setExpires()` converte automaticamente la data al fuso orario GMT, come richiesto dalle specifiche.

Si noti che, nelle versioni di HTTP precedenti alla 1.1, non era richiesto al server di origine di inviare l'header `Date`. Di conseguenza, la cache (p.e. il browser) potrebbe aver bisogno di appoggiarsi all'orologio locale per valutare l'header `Expires`, rendendo il calcolo del ciclo di vita vulnerabile a difformità di ore. L'header `Expires` soffre di un'altra limitazione: le specifiche stabiliscono che "i server HTTP/1.1 non dovrebbero inviare header `Expires` oltre un anno nel futuro."

## Scadenza con l'header `Cache-Control`

A causa dei limiti dell'header `Expires`, la maggior parte delle volte si userà al suo posto l'header `Cache-Control`. Si ricordi che l'header `Cache-Control` è usato per specificare molte differenti direttive di cache. Per la scadenza, ci sono due direttive, `max-age` e `s-maxage`. La prima è usata da tutte le cache, mentre la seconda viene considerata solo dalla cache condivise:

```
// Imposta il numero di secondi dopo cui la risposta
// non dovrebbe più essere considerata fresca
$response->setMaxAge(600);

// Come sopra, ma solo per cache condivise
$response->setSharedMaxAge(600);
```

L'header `Cache-Control` avrebbe il seguente formato (potrebbe contenere direttive aggiuntive):

```
Cache-Control: max-age=600, s-maxage=600
```

## Validazione

Quando una risorsa ha bisogno di essere aggiornata non appena i dati sottostanti subiscono una modifica, il modello a scadenza non raggiunge lo scopo. Con il modello a scadenza, all'applicazione non sarà chiesto di restituire la risposta aggiornata, finché la cache non diventa vecchia.

Il modello a validazione si occupa di questo problema. Con questo modello, la cache continua a memorizzare risposte. La differenza è che, per ogni richiesta, la cache chiede all'applicazione se la risposta in cache è ancora valida. Se la cache *è* ancora valida, l'applicazione dovrebbe restituire un codice di stato 304 e nessun contenuto. Questo dice alla cache che è va bene restituire la risposta in cache.

Con questo modello, si risparmiare solo CPU, se si è in grado di determinare che la risposta in cache sia ancora valida, facendo *meno* lavoro rispetto alla generazione dell'intera pagina (vedere sotto per un esempio di implementazione).

**Tip:** Il codice di stato 304 significa "non modificato". È importante, perché questo codice di stato *non* contiene il vero contenuto richiesto. La risposta è invece un semplice e leggero insieme di istruzioni che dicono alla cache che dovrebbe usare la sua versione memorizzata.

Come per la scadenza, ci sono due diversi header HTTP che possono essere usati per implementare il modello a validazione: `ETag` e `Last-Modified`.

## Validazione con header `ETag`

L'header `ETag` è un header stringa (chiamato "tag entità") che identifica univocamente una rappresentazione della risorsa in questione. È interamente generato e impostato dall'applicazione, quindi si può dire, per esempio, se la risorsa `/about` che è in cache sia aggiornata con ciò che l'applicazione restituirebbe. Un `ETag` è come un'impronta digitale ed è usato per confrontare rapidamente se due diverse versioni di una risorsa siano equivalenti. Come le impronte digitali, ogni `ETag` deve essere univoco tra tutte le rappresentazioni della stessa risorsa.

Ecco una semplice implementazione, che genera l'ETag come un md5 del contenuto:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function homepageAction(Request $request)
    {
        $response = $this->render('static/homepage.html.twig');
        $response->setETag(md5($response->getContent()));
        $response->setPublic(); // assicurarsi che la risposta sia pubblica
        $response->isNotModified($request);

        return $response;
    }
}
```

Il metodo **:method:'Symfony\\Component\\HttpFoundation\\Response::isNotModified'** confronta l'`ETag` inviato con la `Request` con quello impostato nella `Response`. Se i due combaciano, il metodo imposta automaticamente il codice di stato della `Response` a 304.

---

**Note:** L'header `If-None-Match` della richiesta corrisponde all'header `ETag` dell'ultima risposta inviata al client per una particolare risorsa. In questo modo il client e il server comunicano a vicenda e decidono se la risorsa sia stata aggiornata o meno, rispetto a quando è stata messa in cache.

---

Questo algoritmo è abbastanza semplice e molto generico, ma occorre creare l'intera `Response` prima di poter calcolare l'ETag, che non è ottimale. In altre parole, fa risparmiare banda, ma non cicli di CPU.

Nella sezione *Ottimizzare il codice con la validazione*, mostreremo come si possa usare la validazione in modo più intelligente, per determinare la validità di una cache senza dover fare tanto lavoro.

---

**Tip:** Symfony supporta anche gli ETag deboli, passando `true` come secondo parametro del metodo **:method:'Symfony\\Component\\HttpFoundation\\Response::setETag'**.

---

## Validazione col metodo `Last-Modified`

L'header `Last-Modified` è la seconda forma di validazione. Secondo le specifiche HTTP, "l'header `Last-Modified` indica la data e l'ora in cui il server di origine crede che la rappresentazione sia stata modificata l'ultima volta". In altre parole, l'applicazione decide se il contenuto in cache sia stato modificato o meno, in base al fatto se sia stato aggiornato o meno da quando la risposta è stata messa in cache.

Per esempio, si può usare la data di ultimo aggiornamento per tutti gli oggetti necessari per calcolare la rappresentazione della risorsa come valore dell'header `Last-Modified`:

```
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

// ...
use Symfony\Component\HttpFoundation\Request;
use AppBundle\Entity\Article;

class ArticleController extends Controller
{
    public function showAction(Article $article, Request $request)
    {
        $author = $article->getAuthor();

        $articleDate = new \DateTime($article->getUpdatedAt());
        $authorDate = new \DateTime($author->getUpdatedAt());

        $date = $authorDate > $articleDate ? $authorDate : $articleDate;

        $response->setLastModified($date);
        // imposta la risposta come pubblica. Altrimenti, è privata come valore␣
↪predefinito.
        $response->setPublic();

        if ($response->isNotModified($request)) {
            return $response;
        }

        // ... fare qualcosa per popolare la risposta con il contenuto completo

        return $response;
    }
}
```

Il metodo method:*Symfony\Component\HttpFoundation\Response::isNotModified* confronta l'header `If-Modified-Since` inviato dalla richiesta con l'header `Last-Modified` impostato nella risposta. Se sono equivalenti, la `Response` sarà impostata a un codice di stato 304.

---

**Note:** L'header della richiesta `If-Modified-Since` equivale all'header `Last-Modified` dell'ultima risposta inviata al client per una determinata risorsa. In questo modo client e server comunicano l'uno con l'altro e decidono se la risorsa sia stata aggiornata o meno da quando è stata messa in cache.

---

## Ottimizzare il codice con la validazione

Lo scopo principale di ogni strategia di cache è alleggerire il carico dell'applicazione. In altre parole, meno un'applicazione fa per restituire una risposta 304, meglio è. Il metodo `Response::isNotModified()` fa esattamente questo, esponendo uno schema semplice ed efficiente:

```
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

// ...
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
```

```
class ArticleController extends Controller
{
    public function showAction($articleSlug, Request $request)
    {
        // Prende l'informazione minima per calcolare
        // l'ETag o o il valore di Last-Modified
        // (in base alla Request, i dati sono recuperati da una
        // base dati o da una memoria chiave-valore, per esempio)
        $article = ...;

        // crea una Response con un ETag e/o un header Last-Modified
        $response = new Response();
        $response->setETag($article->computeETag());
        $response->setLastModified($article->getPublishedAt());

        // imposta la risposta come pubblica. Altrimenti, è privata come valore␣
↪predefinito.
        $response->setPublic();

        // Verifica che la Response non sia modificata per la Request data
        if ($response->isNotModified($request)) {
            // restituisce subito la Response 304
            return $response;
        }

        // qui fare qualcosa, come recuperare altri dati
        $comments = ...;

        // o rendere un template con la $response già iniziata
        return $this->render('article/show.html.twig', array(
            'article' => $article,
            'comments' => $comments
        ), $response);
    }
}
```

Quando la `Response` non è stata modificata, `isNotModified()` imposta automaticamente il codice di stato della risposta a `304`, rimuove il contenuto e rimuove alcuni header che non devono essere presenti in una risposta `304` (vedere **:method:'Symfony\\Component\\HttpFoundation\\Response::setNotModified'**).

## Variare la risposta

Finora abbiamo ipotizzato che ogni URI avesse esattamente una singola rappresentazione della risorsa interessata. Per impostazione predefinita, la cache HTTP usa l'URI della risorsa come chiave. Se due persone richiedono lo stesso URI di una risorsa che si può mettere in cache, la seconda persona riceverà la versione in cache.

A volte questo non basta e diverse versioni dello stesso URI hanno bisogno di stare in cache in base a uno più header di richiesta. Per esempio, se si comprimono le pagine per i client che supportano per la compressione, ogni URI ha due rappresentazioni: una per i client col supporto e l'altra per i client senza supporto. Questo viene determinato dal valore dell'header di richiesta `Accept-Encoding`.

In questo caso, occorre mettere in cache sia una versione compressa che una non compressa della risposta di un particolare URI e restituirle in base al valore `Accept-Encoding` della richiesta. Lo si può fare usando l'header di risposta `Vary`, che è una lista separata da virgole dei diversi header i cui valori causano rappresentazioni diverse della risorsa richiesta:

```
Vary: Accept-Encoding, User-Agent
```

**Tip:** Questo particolare header `Vary` fa mettere in cache versioni diverse di ogni risorsa in base all'URI, al valore di `Accept-Encoding` e all'header di richiesta `User-Agent`.

L'oggetto `Response` offre un'interfaccia pulita per la gestione dell'header `Vary`:

```
// imposta un header Vary
$response->setVary('Accept-Encoding');

// imposta diversi header Vary
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

Il metodo `setVary()` accetta un nome di header o un array di nomi di header per i quali la risposta varia.

## Scadenza e validazione

Si può ovviamente usare sia la validazione che la scadenza nella stessa `Response`. Poiché la scadenza vince sulla validazione, si può beneficiare dei vantaggi di entrambe. In altre parole, usando sia la scadenza che la validazione, si può istruire la cache per servire il contenuto in cache, controllando ogni tanto (la scadenza) per verificare che il contenuto sia ancora valido.

**Tip:** Si possono anche definire header HTTP per la scadenza e la validazione della cache usando le annotazioni. Vedere la documentazione di FrameworkExtraBundle.

## Altri metodi della risposta

La classe `Response` fornisce molti altri metodi per la cache. Ecco alcuni dei più utili:

```
// Segna la risposta come vecchia
$response->expire();

// Forza la risposta a restituire un 304 senza contenuti
$response->setNotModified();
```

Inoltre, la maggior parte degli header HTTP relativi alla cache può essere impostata tramite il singolo metodo **:method:'Symfony\\Component\\HttpFoundation\\Response::setCache'**:

```
// Imposta le opzioni della cache in una sola chiamata
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

## Invalidazione della cache

"Ci sono solo due cose difficili in informatica: invalidazione della cache e nomi delle cose." – Phil Karlton

Una volta che un URL è memorizzato in una gateway cache, la cache non chiederà più tale contenuto all'applicazione. Ciò consente alla cache di fornire risposte veloci e ridurre il carico sull'applicazione. Tuttavia, si rischia di fornire contenuti obsoleti. Un modo per uscire da questo dilemma è usare tempi di cache lunghi, notificando alla gateway cache quando il contenuto cambia. I reverse proxy solitamente forniscono un canale per ricevere tali notifiche, tipicamente tramite speciali richieste HTTP.

**Caution:** L'invalidazione della cache è potente, ma va evitata quando possibile. Se non si riesce a invalidare qualcosa, le cache obsolete saranno servite per un tempo potenzialmente molto lungo. Invece, usare tempi di cache brevi o usare il modello a validazione e adattare i controllori per eseguire validazioni efficienti, come spiegato in *Ottimizzare il codice con la validazione*.

Inoltre, essendo l'invalidazione un argomento specifico di ciascun tipo di reverse proxy, il suo utilizzo lega a un reverse proxy specifico oppure richiede sforzi aggiuntivi per supportare proxy diversi.

A volte, tuttavia, si ha bisogno di quelle prestazioni in più che si possono ottenere invalidando esplicitamente. Per l'invalidazione, l'applicazione ha bisogno di individuare quando il contenuto cambia e riferire alla cache di rimuovere gli URL che contengono tali dati in cache.

**Tip:** Se si vuole usare l'invalidazione della cache, si potrebbe usare FOSHttpCacheBundle. Questo bundle fornisce servizi per l'aiuto in vari concetti di invalidazione della cache e documenta la configurazione per un paio di proxy comunemente usati.

Se un contenuto corrisponde a un URL, il modello `PURGE` funziona molto bene. Si manda una richiesta al proxy con il metodo HTTP `PURGE` (l'uso del verbo "PURGE" è una convenzione, tecnicamente si può usare una stringa qualsiasi) al posto di `GET` e fare in modo che il proxy lo individui e rimuova i dati dall cache, invece di passare a Symfony per avere una risposta.

Ecco come si può configurare il reverse proxy di Symfony per supportare il metodo HTTP `PURGE`:

```
// app/AppCache.php

use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
// ...

class AppCache extends HttpCache
{
    protected function invalidate(Request $request, $catch = false)
    {
        if ('PURGE' !== $request->getMethod()) {
            return parent::invalidate($request, $catch);
        }

        if ('127.0.0.1' !== $request->getClientIp()) {
            return new Response(
                'Invalid HTTP method',
                Response::HTTP_BAD_REQUEST
            );
        }
```

```
        $response = new Response();
        if ($this->getStore()->purge($request->getUri())) {
            $response->setStatusCode(200, 'Purged');
        } else {
            $response->setStatusCode(200, 'Not found');
        }

        return $response;
    }
}
```

**Caution:** Occorre proteggere in qualche modo il metodo HTTP `PURGE`, per evitare che qualcuno pulisca casualmente i dati in cache.

**Purge** dice alla cache di eliminare una risorsa in *tutte le sue varianti* (in accordo con l'header `Vary`, vedere sopra). Un'alternativa a Purge è il **refresh**. Refresh vuol dire che al proxy viene chiesto di scartare la sua cache locale e recuperare nuovamente il contenuto. In questo modo, il nuovo contenuto è già disponibile nella cache. Il difetto del refresh è che le varianti non vengono invalidate.

In molte applicazioni, lo stesso bit di contenuto è usato su varie pagine con URL diversi. In questo caso, esistono concetti più flessibili:

- Il *ban** invalida risposte che corrispondono a espressioni sull'URL o ad altri criteri;
- Il **tag** consente di marcare ciascun contenuto usato in una risposta, in modo da poter invalidare tutti gli URL che includono un determinato contenuto.

## Usare Edge Side Include

Le gateway cache sono un grande modo per rendere un sito più prestante. Ma hanno una limitazione: possono mettere in cache solo pagine intere. Se non si possono mettere in cache pagine intere o se le pagine hanno più parti dinamiche, non vanno bene. Fortunatamente, Symfony fornisce una soluzione a questi casi, basata su una tecnologia chiamata ESI, o Edge Side Includes. Akamaï ha scritto le specifiche quasi dieci anni fa, consentendo a determinate parti di una pagina di avere differenti strategie di cache rispetto alla pagina principale.

Le specifiche ESI descrivono dei tag che si possono inserire nelle proprie pagine, per comunicare col gateway cache. L'unico tag implementato in Symfony è `include`, poiché è l'unico utile nel contesto di Akamaï:

```
<!DOCTYPE html>
<html>
    <body>
        <!-- ... del contenuto -->

        <!-- Inserisce qui il contenuto di un'altra pagina -->
        <esi:include src="http://..." />

        <!-- ... dell'altro contenuto -->
    </body>
</html>
```

**Note:** Si noti nell'esempio che ogni tag ESI ha un URL pienamente qualificato. Un tag ESI rappresenta un frammento

di pagina che può essere recuperato tramite l'URL fornito.

Quando gestisce una richiesta, il gateway cache recupera l'intera pagina dalla sua cache oppure la richiede dall'applicazione di backend. Se la risposta contiene uno o più tag ESI, questi vengono processati nello stesso modo. In altre parole, la gateway cache o recupera il frammento della pagina inclusa dalla sua cache oppure richiede il frammento di pagina all'applicazione di backend. Quando tutti i tag ESI sono stati risolti, il gateway cache li fonde nella pagina principale e invia il contenuto finale al client.

Tutto questo avviene in modo trasparente a livello di gateway cache (quindi fuori dall'applicazione). Come vedremo, se si scegli di avvalersi dei tag ESI, Symfony rende quasi senza sforzo il processo di inclusione.

## Usare ESI in Symfony

Per usare ESI, assicurarsi prima di tutto di abilitarlo nella configurazione dell'applicazione:

Supponiamo ora di avere una pagina relativamente statica, tranne per un elenco di news in fondo al contenuto. Con ESI, si può mettere in cache l'elenco di news indipendentemente dal resto della pagina.

```
// src/AppBundle/Controller/DefaultController.php

// ...
class DefaultController extends Controller
{
    public function aboutAction()
    {
        $response = $this->render('static/about.html.twig');
        // imposta il tempo massimo condiviso, il che rende la risposta pubblica
        $response->setSharedMaxAge(600);

        return $response;
    }
}
```

In questo esempio, abbiamo dato alla cache della pagina intera un tempo di vita di dieci minuti. Successivamente, includiamo l'elenco di news nel template, includendolo in un'azione. Possiamo farlo grazie all'aiutante `render` (vedere *Inserire controllori* per maggiori dettagli).

Poiché il contenuto incluso proviene da un'altra pagina (o da un altro controllore), Symfony usa l'aiutante `render` per configurare i tag ESI:

Usando l'opzione `esi` (che usa a sua volta la funzione Twig `render_esi`), si dice a Symfony che l'azione va resa come tag ESI. Ci si potrebbe chiedere perché voler usare un aiutante invece di scrivere direttamente il tag ESI. Il motivo è che un aiutante fa funzionare l'applicazione anche se non ci sono gateway per la cache installati.

**Tip:** Come si vedrà più avanti, la variabile passata `maxPerPage` è disponibile come parametro del controllore (come `$maxPerPage`). Le variabili passate tramite `render_esi` diventano ugualmente parte della chiave di cache, in modo da avere cache uniche per ogni combinazione di variabili e valori.

Quando si usa la funzione `render` predefinita (o si usa l'opzione `inline`), Symfony fonde il contenuto della pagina inclusa in quello principale, prima di inviare la risposta al client. Se invece si usa l'opzione `esi` (che richiama `render_esi`) *e* se Symfony capisce che sta parlando a un gateway per la cache che supporti ESI, genera un tag ESI. Ma se non c'è alcun gateway per la cache o se ce n'è uno che non supporta ESI, Symfony fonderà il contenuto della pagina inclusa in quello principale, come se fosse state usata `render`.

**Note:** Symfony individua se una gateway cache supporta ESI tramite un'altra specifica di Akamaï, che è supportata nativamente dal reverse proxy di Symfony.

L'azione inclusa ora può specificare le sue regole di cache, indipendentemente dalla pagina principale.

```
// src/AppBundle/Controller/NewsController.php
namespace AppBundle\Controller;

// ...
class NewsController extends Controller
{
    public function latestAction($maxPerPage)
    {
        // ...
        $response->setSharedMaxAge(60);

        return $response;
    }
}
```

Con ESI, la cache dell'intera pagina sarà valida per 600 secondi, mentre il componente delle news avrà una cache che dura per soli 60 secondi.

Quando si fa riferimento a un controllore, il tag ESI dovrebbe far riferimento all'azione inclusa con un URL accessibile, in modo che il gateway della cache possa recuperarla indipendentemente dal resto della pagina. Symfony si occupa di generare un URL univoco per ogni riferimento a controllori ed è in grado di puntare correttamente le rotte, grazie all'ascoltatore `Symfony\Component\HttpKernel\EventListener\FragmentListener`, che va abilitato nella configurazione:

Un grosso vantaggio di questa strategia di cache è che si può rendere l'applicazione tanto dinamica quanto necessario e, allo stesso tempo, mantenere gli accessi al minimo.

**Tip:** L'ascoltatore risponde solo agli indirizzi IP locali o ai proxy fidati.

**Note:** Una volta iniziato a usare ESI, si ricordi di usare sempre la direttiva `s-maxage` al posto di `max-age`. Poiché il browser riceve la risorsa aggregata, non ha visibilità sui sotto-componenti, quindi obbedirà alla direttiva `max-age` e metterà in cache l'intera pagina. E questo non è quello che vogliamo.

L'aiutante `render_esi` supporta due utili opzioni:

**`alt`** usato come attributo `alt` nel tag ESI, che consente di specificare un URL alternativo da usare, nel caso in cui `src` non venga trovato;

**`ignore_errors`** se impostato a `true`, un attributo `onerror` sarà aggiunto a ESI con il valore di `continue`, a indicare che, in caso di fallimento, la gateway cache semplicemente rimuoverà il tag ESI senza produrre errori.

## Riepilogo

Symfony è stato progettato per seguire le regole sperimentate della strada: HTTP. La cache non fa eccezione. Padroneggiare il sistema della cache di Symfony vuol dire acquisire familiarità con i modelli di cache HTTP e usarli in modo efficace. Vuol dire anche che, invece di basarsi solo su documentazione ed esempi di Symfony, si ha accesso al mondo della conoscenza relativo alla cache HTTP e a gateway cache come Varnish.

## Imparare di più con le ricette

- /cookbook/cache/varnish

CHAPTER 17

# Traduzioni

Il termine "internazionalizzazione" si riferisce al processo di astrazione delle stringhe e altri pezzi specifici dell'applicazione che variano in base al locale, in uno strato dove possono essere tradotti e convertiti in base alle impostazioni internazionali dell'utente (ad esempio lingua e paese). Per il testo, questo significa che ognuno viene avvolto con una funzione capace di tradurre il testo (o "messaggio") nella lingua dell'utente:

```
// il testo verrà *sempre* stampato in inglese
echo 'Hello World';

// il testo può essere tradotto nella lingua dell'utente finale o
// restare in inglese
echo $translator->trans('Hello World');
```

**Note:** Il termine *locale* si riferisce all'incirca al linguaggio dell'utente e al paese. Può essere qualsiasi stringa che l'applicazione utilizza poi per gestire le traduzioni e altre differenze di formati (ad esempio il formato di valuta). Si consiglia di utilizzare il codice di *lingua* ISO 639-1, un carattere di sottolineatura (_), poi il codice di *paese* ISO 3166-1 alpha-2 (per esempio `fr_FR` per francese/Francia).

In questo capitolo si imparerà a usare il componente Translation nel framework Symfony. Si può leggere la documentazione del componente Translation per saperne di più. Nel complesso, il processo ha diverse fasi:

1. *Abilitare e configurare* il servizio translation di Symfony;
2. Astrarre le stringhe (i. "messaggi") avvolgendoli nelle chiamate al `Translator` ("*Traduzione di base*");
3. *Creare risorse di traduzione* per ogni lingua supportata che traducano tutti i messaggio dell'applicazione;
4. Determinare, *impostare e gestire le impostazioni locali* dell'utente per la richiesta e, facoltativamente, sull'intera sessione.

## Configurazione

Le traduzioni sono gestire da un servizio `translator`, che utilizza i locale dell'utente per cercare e restituire i messaggi tradotti. Prima di utilizzarlo, abilitare `translator` nella configurazione:

Vedere *Fallback e locale predefinito* per dettagli sulla voce `fallbacks` e su cosa faccia Symfony quando non trova una traduzione.

Il locale usato nelle traduzioni è quello memorizzato nella richiesta. Tipicamente, è impostato tramite un attributo `_locale` in una rotta (vedere *Il locale e gli URL*).

## Traduzione di base

La traduzione del testo è fatta attraverso il servizio `translator` (`Symfony\Component\Translation\Translator`). Per tradurre un blocco di testo (chiamato *messaggio*), usare il metodo **:method:'Symfony\\Component\\Translation\\Translator::trans'**. Supponiamo, ad esempio, che stiamo traducendo un semplice messaggio all'interno del controllore:

```
// ...
use Symfony\Component\HttpFoundation\Response;

public function indexAction()
{
    $translated = $this->get('translator')->trans('Symfony is great');

    return new Response($translated);
}
```

Quando questo codice viene eseguito, Symfony tenterà di tradurre il messaggio "Symfony is great" basandosi sul locale dell'utente. Perché questo funzioni, bisogna dire a Symfony come tradurre il messaggio tramite una "risorsa di traduzione", che è una raccolta di traduzioni dei messaggi per un dato locale. Questo "dizionario" delle traduzioni può essere creato in diversi formati, ma XLIFF è il formato raccomandato:

Per informazioni sulla posizione di questi file, vedere *Sedi per le traduzioni e convenzioni sui nomi*.

Ora, se la lingua del locale dell'utente è il francese (per esempio `fr_FR` o `fr_BE`), il messaggio sarà tradotto in `J'aime Symfony`. Si può anche tradurre il messaggio da un *template*.

### Il processo di traduzione

Per tradurre il messaggio, Symfony utilizza un semplice processo:

- Viene determinato il `locale` dell'utente corrente, che è memorizzato nella richiesta;
- Un catalogo di messaggi tradotti viene caricato dalle risorse di traduzione definite per il `locale` (ad es. `fr_FR`). Vengono anche caricati i messaggi dal *locale predefinito* e aggiunti al catalogo, se non esistono già. Il risultato finale è un grande "dizionario" di traduzioni;
- Se il messaggio si trova nel catalogo, viene restituita la traduzione. Se no, il traduttore restituisce il messaggio originale.

Quando si usa il metodo `trans()`, Symfony cerca la stringa esatta all'interno del catalogo dei messaggi e la restituisce (se esiste).

### Segnaposto per i messaggi

A volte, un messaggio da tradurre contiene una variabile:

```
use Symfony\Component\HttpFoundation\Response;

public function indexAction($name)
{
    $translated = $this->get('translator')->trans('Hello '.$name);

    return new Response($translated);
}
```

Tuttavia, la creazione di una traduzione per questa stringa è impossibile, poiché il traduttore proverà a cercare il messaggio esatto, includendo le parti con le variabili (per esempio "Hello Ryan" o "Hello Fabien").

Per dettagli su come gestire questa situazione, vedere component-translation-placeholders nella documentazione del componente. Per i template, vedere *Template Twig*.

## Pluralizzazione

Un'ulteriore complicazione si presenta con traduzioni che possono essere plurali o meno, in base a una qualche variabile:

```
There is one apple.
There are 5 apples.
```

Per poterlo gestire, usare il metodo **:method:'Symfony\\Component\\Translation\\Translator::transChoice'** del tag o del filtro `transchoice` nel *template*.

Per ulteriori e approfondite informazioni, vedere component-translation-pluralization nella documentazione del componente Translation.

## Traduzioni nei template

Le traduzioni avvengono quasi sempre all'interno di template. Symfony fornisce un supporto nativo sia per i template Twig che per quelli PHP.

### Template Twig

Symfony fornisce tag specifici per Twig (`trans` e `transchoice`), che aiutano nella traduzioni di messaggi di *blocchi statici di testo*:

```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Il tag `transchoice` prende in automatico la variabile `%count%` dal contesto e la passa al traduttore. Questo meccanismo funziona solo usando un segnaposto che segue lo schema `%variabile%`.

**Caution:** La notazione `%variabile%` dei segnaposti è obbligatoria quando si traduce in un template Twig usando il tag.

**Tip:** Se si deve usare un simbolo di percentuale (`%`) in una stringa, occorre raddoppiarlo: `{% trans %}Percent: %percent%%%{% endtrans %}`

Si può anche specificare il dominio del messaggio e passare variabili aggiuntive:

```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% trans with {'%name%': 'Fabien'} from "app" into "fr" %}Hello %name%{% endtrans %}

{% transchoice count with {'%name%': 'Fabien'} from "app" %}
    {0} %name%, there are no apples|{1} %name%, there is one apple|]1,Inf] %name%,␣
↪there are %count% apples
{% endtranschoice %}
```

I filtri `trans` e `transchoice` possono essere usati per tradurre *testi variabili* ed espressioni complesse:

```
{{ message|trans }}

{{ message|transchoice(5) }}

{{ message|trans({'%name%': 'Fabien'}, "app") }}

{{ message|transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

**Tip:** L'uso dei tag o dei filtri di traduzione ha il medesimo effetto, ma con una sottile differenza: l'escape automatico si applica solo alla traduzione che usa un filtro. In altre parole, se ci si deve assicurare che il testo tradotto *non* abbia escape, occorre applicare il filtro `raw` dopo il filtro di traduzione:

```
{# il testo tra tag non subisce escape #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# stringhe e variabili tradotte con filtro subiscono escape #}
{{ message|trans|raw }}
{{ '<h3>bar</h3>'|trans|raw }}
```

**Tip:** Si può impostare il dominio di un intero template Twig con un semplice tag:

```
{% trans_default_domain "app" %}
```

Notare che questo influenza solo in template attuale, non i template "inclusi" (per evitare effetti collaterali).

### Template PHP

Il servizio di traduzione è accessibile nei template PHP attraverso l'aiutante `translator`:

```
<?php echo $view['translator']->trans('Symfony is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

## Sedi per le traduzioni e convenzioni sui nomi

Symfony cerca i file dei messaggi (ad esempio le traduzioni) in due sedi:

- la cartella `app/Resources/translations`;
- la cartella `app/Resources/<nome bundle>/translations`;
- la cartella `Resources/translations/` del bundle.

I posti sono elencati in ordine di priorità. Quindi, si possono sovrascrivere i messaggi di traduzione di un bundle in una qualsiasi delle due cartelle superiori.

Il meccanismo di priorità si basa sulle chiavi: occorre dichiarare solamente le chiavi da sovrascrivere in un file di messaggi a priorità superiore. Se una chiave non viene trovata in un file di messaggi, il traduttore si appoggerà automaticamente ai file di messaggi a priorità inferiore.

È importante anche il nome del file con le traduzioni: ogni file con i messaggi deve essere nominato secondo il seguente schema: `dominio.locale.caricatore`:

- **dominio**: Un modo opzionale per organizzare i messaggi in gruppi (ad esempio `admin`, `navigation` o il predefinito `messages`, vedere "using-message-domains");
- **locale**: Il locale per cui sono state scritte le traduzioni (ad esempio `en_GB`, `en`, ecc.);
- **caricatore**: Come Symfony dovrebbe caricare e analizzare il file (ad esempio `xliff`, `php` o `yml`).

Il caricatore può essere il nome di un qualunque caricatore registrato. Per impostazione predefinita, Symfony fornisce i seguenti caricatori:

- `xliff`: file XLIFF;
- `php`: file PHP;
- `yml`: file YAML.

La scelta di quali caricatori utilizzare è interamente a carico dello sviluppatore ed è una questione di gusti. L'opzione raccomandata è il formato `xliff`. Per altre opzioni, vedere component-translator-message-catalogs.

---

**Note:** È anche possibile memorizzare le traduzioni in una base dati o in qualsiasi altro mezzo, fornendo una classe personalizzata che implementa l'interfaccia `Symfony\Component\Translation\Loader\LoaderInterface`. Vedere dic-tags-translation-loader per maggiori informazioni.

---

**Caution:** Ogni volta che si crea una *nuova* risorsa di traduzione (o si installa un bundle che include risorse di traduzioni), assicurarsi di pulire la cache, in modo che Symfony possa rilevare le nuove risorse:

```
$ php app/console cache:clear
```

### Fallback e locale predefinito

Ipotizzando che il locale dell'utente sia `fr_FR` e che si stia traducendo la chiave `Symfony is great`. Per trovare la traduzione francese, Symfony verifica le risorse di traduzione di vari locale:

1. Prima, Symfony cerca la traduzione in una risorsa di traduzione `fr_FR` (p.e. `messages.fr_FR.xfliff`);
2. Se non la trova, Symfony cerca una traduzione per una risorsa di traduzione `fr` (p.e. `messages.fr.xliff`);
3. Se non trova nemeno questa, Symfony usa il parametro di configurazione `fallback`, che ha come valore predefinito `en` (vedere *Configurazione*).

New in version 2.6: La possibilità di scrivere nei log le traduzioni mancanti è stata introdotta in Symfony 2.6.

**Note:** Quando Symfony non trova una traduzione per il locale dato, aggiungerà la traduzione mancante al file di log. Per dettagli, vedere reference-framework-translator-logging.

## Gestire il locale dell'utente

Il locale dell'utente attuale è memorizzato nella richiesta e accessibile tramite l'oggetto `request`:

```
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $locale = $request->getLocale();

    $request->setLocale('en_US');
}
```

**Tip:** Leggere /cookbook/session/locale_sticky_session per approfondimenti sull'argomento.

Vedere la sezione seguente, *Il locale e gli URL*, per impostare il locale tramite rotte.

### Il locale e gli URL

Dal momento che si può memorizzare il locale dell'utente nella sessione, si può essere tentati di utilizzare lo stesso URL per visualizzare una risorsa in più lingue in base al locale dell'utente. Per esempio, `http://www.example.com/contact` può mostrare contenuti in inglese per un utente e in francese per un altro. Purtroppo questo viola una fondamentale regola del web: un particolare URL deve restituire la stessa risorsa indipendentemente dall'utente. Inoltre, quale versione del contenuto dovrebbe essere indicizzata dai motori di ricerca?

Una politica migliore è quella di includere il locale nell'URL. Questo è completamente supportato dal sistema delle rotte utilizzando il parametro speciale `_locale`:

Quando si utilizza il parametro speciale *_locale* in una rotta, il locale corrispondente verrà *automaticamente impostato sulla richiesta* e potrà essere recuperate tramite il metodo **:method:'Symfony\\Component\\HttpFoundation\\Request::getLocale'**. In altre parole, se un utente visita l'URI `/fr/contact`, il locale `fr` viene impostato automaticamente come locale per la richiesta corrente.

È ora possibile utilizzare il locale dell'utente per creare rotte ad altre pagine tradotte nell'applicazione.

---

**Tip:** Leggere /cookbook/routing/service_container_parameters per imparare come evitare di inserire manualmente il requisito `_locale` in ogni rotta.

---

## Impostare un locale predefinito

Che fare se non si è in grado di determinare il locale dell'utente? Si può garantire che un locale sia impostato a ogni richiesta, definendo un `default_locale` per il framework:

## Tradurre i messaggi dei vincoli

Se si usano i vincoli di validazione dei form, la traduzione dei messaggi di errore è facile: basta creare una risorsa di traduzione per il dominio `validators`.

Per iniziare, supponiamo di aver creato un oggetto PHP, necessario da qualche parte in un'applicazione:

```
// src/AppBundle/Entity/Author.php
namespace AppBundle\Entity;

class Author
{
    public $name;
}
```

Aggiungere i vincoli tramite uno dei metodi supportati. Impostare l'opzione del messaggio al testo sorgente della traduzione. Per esempio, per assicurarsi che la proprietà `$name` non sia vuota, aggiungere il seguente:

Creare un file di traduzione sotto il catalogo `validators` per i messaggi dei vincoli, tipicamente nella cartella `Resources/translations/` del bundle.

## Tradurre contenuti della base dati

La traduzione di contenuti della base dati andrebbe affidata a Doctrine, tramite l'estensione Translatable o il behavior Translatable (per PHP 5.4+). Per maggiori informazioni, vedere la documentazione di queste librerie.

## Debug delle traduzioni

New in version 2.5: Il comando `translation:debug` è stato introdotto in Symfony 2.5.

New in version 2.6: Prima di Symfony 2.6, questo comando si chiamava `translation:debug`.

Durante la manutenzione di un bundle, si potrebbe usare o disabilitare un messaggio di traduzioni, senza aggiornare tutti i cataloghi dei messaggi. Il comando `translation:debug` aiuta a trovare questi messaggi di traduzioni

mancanti o inutilizzati, per un locale dato. Mostra una tabella con i risultati della traduzione del messaggio nel locale fornito e il risultato quando viene usato il fallback. Inoltre, mostra quando le traduzioni sono uguali all traduzione fallback (potrebbe indicare che il messaggio non sia stato tradotto correttamente).

Grazie agli estrattori di messaggi, il comando troverà il tag di traduzione o l'uso di filtri nei template Twig:

```
{% trans %}Symfony2 is great{% endtrans %}

{{ 'Symfony2 is great'|trans }}

{{ 'Symfony2 is great'|transchoice(1) }}

{% transchoice 1 %}Symfony2 is great{% endtranschoice %}
```

Individuerà anche i seguenti utilizzi di traduzione in template PHP:

```
$view['translator']->trans("Symfony2 is great");

$view['translator']->trans('Symfony2 is great');
```

**Caution:** Gli estrattori non sono in grado di ispezionare i messaggi tradoti fuori dai template, il che vuol dire che gli utilizzi di traduzioni in label di form o dentro a controllori non saranno individuati. Traduzioni dinamiche, che coinvolgano variabili o espressioni, non sono individuate nei template, il che vuol dire che questo esempio non sarà analizzato:

```
{% set message = 'Symfony2 is great' %}
{{ message|trans }}
```

Si supponga che il locale predefinito sia `fr` e di aver configurato `en` come locale di fallback (vedere *Configurazione* e *Fallback e locale predefinito* su come configurarli). Si supponga inoltre di aver già preparato alcune traduzioni per il locale `fr` in un AcmeDemoBundle:

e per il locale `en`:

Per individuare tutti i messaggi nel locale `fr` per AcmeDemoBundle, eseguire:

```
$ php app/console debug:translation fr AcmeDemoBundle
```

Si otterrà questo output:

images/book/translation/debug_1.png

Indica che il messaggio `Symfony2 is great` è inutilizzato, perché è stato tradotto, ma viene usato.

Ora, se si traduce il messaggio in uno dei template, si otterrà questo output:

images/book/translation/debug_2.png

Lo stato è vuoto, che vuol dire che il messaggio è stato tradotto nel locale `fr` e usato in un template.

Se si cancella il messaggio `Symfony2 is great` dal file di traduzione per il locale `fr` e si esegue il comando, si otterrà:


Lo stato indica che il messaggio è mancante, perché non è tradotto nel locale `fr`, ma è usato in un template. Inoltre, il messaggio nel locale `fr` è uguale al messaggio nel locale `en`. Questo è caso particolare, perché il messaggio non tradotto ha lo stesso id della sua traduzione nel locale `en`.

Se si copia il contenuto del file di traduzione del locale `en` nel file di traduzione del locale `fr` e si esegue il comando, si otterrà:


Si può vedere che le traduzioni del messaggio sono identiche nei locale `fr` ed `en`, che vuol dire che questo messaggio è stato probabilmente copiato da francese a inglese e forse ci si è dimenticati di tradurlo.

L'ispezione predefinita avviene su tutti i domini, ma si può specificare un singolo dominio:

```
$ php app/console debug:translation en AcmeDemoBundle --domain=messages
```

Quando i bundle hanno molti messaggi, è utile mostrare solo quelli non usati oppure solo quelli mancanti, usando le opzioni `--only-unused` o `--only-missing`:

```
$ php app/console debug:translation en AcmeDemoBundle --only-unused
$ php app/console debug:translation en AcmeDemoBundle --only-missing
```

## Riepilogo

Con il componente Translation di Symfony, la creazione e l'internazionalizzazione di applicazioni non è più un processo doloroso e si riduce solo a pochi semplici passi:

- Astrarre i messaggi dell'applicazione avvolgendoli utilizzando i metodi **:method:'Symfony\\Component\\Translation\\Translator::trans'** o **:method:'Symfony\\Component\\Translation\\Translato** (vedere anche "/components/translation/usage");
- Tradurre ogni messaggio in più locale creando dei file con i messaggi per la traduzione. Symfony scopre ed elabora ogni file perché i suoi nomi seguono una specifica convenzione;
- Gestire il locale dell'utente, che è memorizzato nella richiesta, ma può anche essere memorizzato nella sessione.

CHAPTER 18

# Contenitore di servizi

Una moderna applicazione PHP è piena di oggetti. Un oggetto può facilitare la consegna dei messaggi di posta elettronica, mentre un altro può consentire di salvare le informazioni in una base dati. Nell'applicazione, è possibile creare un oggetto che gestisce l'inventario dei prodotti, o un altro oggetto che elabora i dati da un'API di terze parti. Il punto è che una moderna applicazione fa molte cose ed è organizzata in molti oggetti che gestiscono ogni attività.

In questo capitolo si parlerà di un oggetto speciale PHP presente in Symfony che aiuta a istanziare, organizzare e recuperare i tanti oggetti della propria applicazione. Questo oggetto, chiamato contenitore di servizi, permetterà di standardizzare e centralizzare il modo in cui sono costruiti gli oggetti nell'applicazione. Il contenitore rende la vita più facile, è super veloce ed evidenzia un'architettura che promuove codice riusabile e disaccoppiato. E poiché tutte le classi del nucleo di Symfony utilizzano il contenitore, si apprenderà come estendere, configurare e usare qualsiasi oggetto in Symfony. In gran parte, il contenitore dei servizi è il più grande contributore riguardo la velocità e l'estensibilità di Symfony.

Infine, la configurazione e l'utilizzo del contenitore di servizi è semplice. Entro la fine di questo capitolo, si sarà in grado di creare i propri oggetti attraverso il contenitore e personalizzare gli oggetti da un bundle di terze parti. Si inizierà a scrivere codice che è più riutilizzabile, testabile e disaccoppiato, semplicemente perché il contenitore di servizi consente di scrivere facilmente del buon codice.

**Tip:** Per un approfondimento successivo alla lettura di questo capitolo, dare un'occhiata alla documentazione del componente Dependency Injection.

## Cos'è un servizio?

In parole povere, un servizio è un qualsiasi oggetto PHP che esegue una sorta di compito "globale". È un nome volutamente generico utilizzato in informatica per descrivere un oggetto che è stato creato per uno scopo specifico (ad esempio spedire email). Ogni servizio è utilizzato in tutta l'applicazione ogni volta che si ha bisogno delle funzionalità specifiche che fornisce. Non bisogna fare nulla di speciale per creare un servizio: è sufficiente scrivere una classe PHP con del codice che realizza un compito specifico. Congratulazioni, si è appena creato un servizio!

**Note:** Come regola generale, un oggetto PHP è un servizio se viene utilizzato a livello globale nell'applicazione. Un singolo servizio `Mailer` è usato globalmente per inviare messaggi email mentre i molti oggetti `Message` che spedisce *non* sono servizi. Allo stesso modo, un oggetto `Product` non è un servizio, ma un oggetto che persiste oggetti `Product` su una base dati *è* un servizio.

Qual è il discorso allora? Il vantaggio dei "servizi" è che si comincia a pensare di separare ogni "pezzo di funzionalità" dell'applicazione in una serie di servizi. Dal momento che ogni servizio fa solo un lavoro, si può facilmente accedere a ogni servizio e utilizzare le sue funzionalità ovunque ce ne sia bisogno. Ogni servizio può anche essere più facilmente testato e configurato essendo separato dalle altre funzionalità dell'applicazione. Questa idea si chiama architettura orientata ai servizi e non riguarda solo Symfony o il PHP. Strutturare la propria applicazione con una serie di classi indipendenti di servizi è una nota best practice della programmazione a oggetti. Queste conoscenze sono fondamentali per essere un buon sviluppatore in quasi tutti i linguaggi.

## Cos'è un contenitore di servizi?

Un contenitore di servizi (o *contenitore di dependency injection*) è semplicemente un oggetto PHP che gestisce l'istanza di servizi (cioè gli oggetti).

Per esempio, supponiamo di avere una semplice classe PHP che spedisce messaggi email. Senza un contenitore di servizi, bisogna creare manualmente l'oggetto ogni volta che se ne ha bisogno:

```
use Acme\HelloBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ...);
```

Questo è abbastanza facile. La classe immaginaria `Mailer` permette di configurare il metodo utilizzato per inviare i messaggi email (per esempio `sendmail`, `smtp`, ecc). Ma cosa succederebbe se volessimo utilizzare il servizio mailer da qualche altra parte? Certamente non si vorrebbe ripetere la configurazione del mailer *ogni* volta che si ha bisogno dell'oggetto `Mailer`. Cosa succederebbe se avessimo bisogno di cambiare il `transport` da `sendmail` a `smtp` in ogni punto dell'applicazione? Avremo bisogno di cercare ogni posto in cui si crea un servizio `Mailer` e cambiarlo.

## Creare/Configurare servizi nel contenitore

Una soluzione migliore è quella di lasciare che il contenitore di servizi crei l'oggetto `Mailer` per noi. Affinché questo funzioni, bisogna *insegnare* al contenitore come creare il servizio `Mailer`. Questo viene fatto tramite la configurazione, che può essere specificata in YAML, XML o PHP:

**Note:** Durante l'inizializzazione di Symfony, viene costruito il contenitore di servizi utilizzando la configurazione dell'applicazione (per impostazione predefinita `app/config/config.yml`). Il file esatto che viene caricato è indicato dal metodo `AppKernel::registerContainerConfiguration()`, che carica un file di configurazione specifico per l'ambiente (ad esempio `config_dev.yml` per l'ambiente `dev` o `config_prod.yml` per `prod`).

Un'istanza dell'oggetto `Acme\HelloBundle\Mailer` è ora disponibile tramite il contenitore di servizio. Il contenitore è disponibile in qualsiasi normale controllore di Symfony in cui è possibile accedere ai servizi del contenitore attraverso il metodo scorciatoia `get()`:

```
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ...);
    }
}
```

Quando si chiede il servizio `my_mailer` del contenitore, il contenitore costruisce l'oggetto e lo restituisce. Questo è un altro grande vantaggio che si ha utilizzando il contenitore di servizi. Questo significa che un servizio non è *mai* costruito fino a che non ce n'è bisogno. Se si definisce un servizio e non lo si usa mai su una richiesta, il servizio non verrà mai creato. Ciò consente di risparmiare memoria e aumentare la velocità dell'applicazione. Questo significa anche che c'è un calo di prestazioni basso o inesistente quando si definiscono molti servizi. I servizi che non vengono mai utilizzati non sono mai costruite.

Come bonus aggiuntivo, il servizio `Mailer` è creato una sola volta e ogni volta che si chiede per il servizio viene restituita la stessa istanza. Questo è quasi sempre il comportamento di cui si ha bisogno (è più flessibile e potente), ma si imparerà come configurare un servizio che ha istanze multiple nella ricetta "/cookbook/service_container/scopes".

---

**Note:** In questo esempio, il controllore estende quello base di Symfony, il quale fornisce accesso al contenitore di servizi. Si può quindi usare il metodo `get` per recuperare il servizio `my_mailer` dal contenitore. Si possono anche definire i controllori come servizi. Questo è un po' più avanzato e non sempre necessario, ma consente di iniettare solo i servizi che serviranno nel controllore.

---

## I parametri del servizio

La creazione di nuovi servizi (cioè oggetti) attraverso il contenitore è abbastanza semplice. Con i parametri si possono definire servizi più organizzati e flessibili:

Il risultato finale è esattamente lo stesso di prima, la differenza è solo nel *come* è stato definito il servizio. Circondando le stringhe `my_mailer.class` e `my_mailer.transport` con il segno di percentuale (`%`), il contenitore sa di dover cercare per parametri con questi nomi. Quando il contenitore è costruito, cerca il valore di ogni parametro e lo usa nella definizione del servizio.

---

**Note:** Se si vuole usare una stringa che inizi con il simbolo `@` come valore di un parametro (p.e. una password) in un file yaml, occorre un escape tramite un ulteriore simbolo `@` (si applica solo al formato YAML):

```
# app/config/parameters.yml
parameters:
    # Questo valore sarà analizzato come "@passwordsicura"
    mailer_password: "@@passwordsicura"
```

---

**Note:** Il simbolo di percentuale dentro a un parametro, come parte della stringa, deve subire un escape tramite un ulteriore simbolo di percentuale:

```
<argument type="string">http://symfony.com/?pippo=%%s&pluto=%%d</argument>
```

Lo scopo dei parametri è quello di inserire informazioni dei servizi. Naturalmente non c'è nulla di sbagliato a definire il servizio senza l'uso di parametri. I parametri, tuttavia, hanno diversi vantaggi:

- separazione e organizzazione di tutte le "opzioni" del servizio sotto un'unica chiave `parameters`;
- i valori dei parametri possono essere utilizzati in molteplici definizioni di servizi;
- la creazione di un servizio in un bundle (lo mostreremo a breve), usando i parametri consente al servizio di essere facilmente personalizzabile nell'applicazione.

La scelta di usare o non usare i parametri è personale. I bundle di alta qualità di terze parti li utilizzeranno *sempre*, perché rendono i servizi memorizzati nel contenitore più configurabili. Per i servizi della propria applicazione, tuttavia, potrebbe non essere necessaria la flessibilità dei parametri.

### Parametri array

I parametri possono anche contenere array. Vedere component-di-parameters-array.

## Importare altre risorse di configurazione del contenitore

**Tip:** In questa sezione, si farà riferimento ai file di configurazione del servizio come *risorse*. Questo per sottolineare il fatto che, mentre la maggior parte delle risorse di configurazione saranno file (ad esempio YAML, XML, PHP), Symfony è così flessibile che la configurazione potrebbe essere caricata da qualunque parte (per esempio in una base dati o tramite un servizio web esterno).

Il contenitore dei servizi è costruito utilizzando una singola risorsa di configurazione (per impostazione predefinita `app/config/config.yml`). Tutte le altre configurazioni di servizi (comprese le configurazioni del nucleo di Symfony e dei bundle di terze parti) devono essere importate da dentro questo file in un modo o nell'altro. Questo dà una assoluta flessibilità sui servizi dell'applicazione.

La configurazione esterna di servizi può essere importata in due modi differenti. Il primo, e più comune, è la direttiva `imports`. Nella sezione seguente, si introdurrà il secondo metodo, che è il metodo più flessibile e privilegiato per importare la configurazione di servizi in bundle di terze parti.

### Importare la configurazione con `imports`

Finora, si è messo la definizione di contenitore del servizio `my_mailer` direttamente nel file di configurazione dell'applicazione (ad esempio `app/config/config.yml`). Naturalmente, poiché la classe `Mailer` stessa vive all'interno di `AcmeHelloBundle`, ha più senso mettere la definizione `my_mailer` del contenitore dentro il bundle stesso.

In primo luogo, spostare la definizione `my_mailer` del contenitore, in un nuovo file risorse del contenitore in `AcmeHelloBundle`. Se le cartelle `Resources` o `Resources/config` non esistono, crearle.

Non è cambiata la definizione, solo la sua posizione. Naturalmente il servizio contenitore non conosce il nuovo file di risorse. Fortunatamente, si può facilmente importare il file risorse utilizzando la chiave `imports` nella configurazione dell'applicazione.

La direttiva `imports` consente all'applicazione di includere risorse di configurazione per il contenitore di servizi da qualsiasi altro posto (in genere da bundle). La locazione `resource`, per i file, è il percorso assoluto al file risorse.

La speciale sintassi `@AcmeHelloBundle` risolve il percorso della cartella del bundle AcmeHelloBundle. Questo aiuta a specificare il percorso alla risorsa senza preoccuparsi in seguito, se si sposta AcmeHelloBundle in una cartella diversa.

## Importare la configurazione attraverso estensioni del contenitore

Quando si sviluppa in Symfony, si usa spesso la direttiva `imports` per importare la configurazione del contenitore dai bundle che sono stati creati appositamente per l'applicazione. Le configurazioni dei contenitori di bundle di terze parti, includendo i servizi del nucleo di Symfony, di solito sono caricati utilizzando un altro metodo che è più flessibile e facile da configurare nell'applicazione.

Ecco come funziona. Internamente, ogni bundle definisce i propri servizi in modo molto simile a come si è visto finora. Un bundle utilizza uno o più file di configurazione delle risorse (di solito XML) per specificare i parametri e i servizi del bundle. Tuttavia, invece di importare ciascuna di queste risorse direttamente dalla configurazione dell'applicazione utilizzando la direttiva `imports`, si può semplicemente richiamare una *estensione del contenitore di servizi* all'interno del bundle che fa il lavoro per noi. Un'estensione del contenitore dei servizi è una classe PHP creata dall'autore del bundle con lo scopo di realizzare due cose:

- importare tutte le risorse del contenitore dei servizi necessarie per configurare i servizi per il bundle;
- fornire una semplice configurazione semantica in modo che il bundle possa essere configurato senza interagire con i parametri "piatti" della configurazione del contenitore dei servizi del bundle.

In altre parole, una estensione dei contenitore dei servizi configura i servizi del il bundle per lo sviluppatore. E, come si vedrà tra poco, l'estensione fornisce un'interfaccia comoda e ad alto livello per configurare il bundle.

Si prenda FrameworkBundle, il bundle del nucleo del framework Symfony, come esempio. La presenza del seguente codice nella configurazione dell'applicazione invoca l'estensione del contenitore dei servizi all'interno di FrameworkBundle:

Quando viene analizzata la configurazione, il contenitore cerca un'estensione che sia in grado di gestire la direttiva di configurazione `framework`. L'estensione in questione, che si trova in FrameworkBundle, viene invocata e la configurazione del servizio per FrameworkBundle viene caricata. Se si rimuove del tutto la chiave `framework` dal file di configurazione dell'applicazione, i servizi del nucleo di Symfony non vengono caricati. Il punto è che è tutto sotto controllo: il framework Symfony non contiene nessuna magia e non esegue nessuna azione su cui non si abbia il controllo.

Naturalmente è possibile fare molto di più della semplice "attivazione" dell'estensione del contenitore dei servizi di FrameworkBundle. Ogni estensione consente facilmente di personalizzare il bundle, senza preoccuparsi di come i servizi interni siano definiti.

In questo caso, l'estensione consente di personalizzare la configurazione di `error_handler`, `csrf_protection`, `router` e di molte altre. Internamente, FrameworkBundle usa le opzioni qui specificate per definire e configurare i servizi a esso specifici. Il bundle si occupa di creare tutte i necessari `parameters` e `services` per il contenitore dei servizi, pur consentendo di personalizzare facilmente gran parte della configurazione. Come bonus aggiuntivo, la maggior parte delle estensioni dei contenitori di servizi sono anche sufficientemente intelligenti da eseguire la validazione, notificando le opzioni mancanti o con un tipo di dato sbagliato.

Durante l'installazione o la configurazione di un bundle, consultare la documentazione del bundle per vedere come devono essere installati e configurati i suoi servizi. Le opzioni disponibili per i bundle del nucleo si possono trovare all'interno della guida di riferimento.

---

**Note:** Nativamente, il contenitore dei servizi riconosce solo le direttive `parameters`, `services` e `imports`. Ogni altra direttiva è gestita dall'estensione del contenitore dei servizi.

---

Se si vogliono esporre in modo amichevole le configurazioni dei propri bundle, leggere la ricetta "/cookbook/bundles/extension".

## Referenziare (iniettare) servizi

Finora, il servizio `my_mailer` è semplice: accetta un solo parametro nel suo costruttore, che è facilmente configurabile. Come si vedrà, la potenza reale del contenitore viene fuori quando è necessario creare un servizio che dipende da uno o più altri servizi nel contenitore.

Cominciamo con un esempio. Supponiamo di avere un nuovo servizio, `NewsletterManager`, che aiuta a gestire la preparazione e la spedizione di un messaggio email a un insieme di indirizzi. Naturalmente il servizio `my_mailer` è già capace a inviare messaggi email, quindi verrà usato all'interno di `NewsletterManager` per gestire la spedizione effettiva dei messaggi. Questa classe potrebbe essere qualcosa del genere:

```
// src/Acme/HelloBundle/Newsletter/NewsletterManager.php
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Senza utilizzare il contenitore di servizi, si può creare abbastanza facilmente un nuovo `NewsletterManager` dentro a un controllore:

```
use Acme\HelloBundle\Newsletter\NewsletterManager;

// ...

public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new NewsletterManager($mailer);
    // ...
}
```

Questo approccio va bene, ma cosa succede se più avanti si decide che la classe `NewsletterManager` ha bisogno di un secondo o terzo parametro nel costruttore? Che cosa succede se si decide di rifattorizzare il codice e rinominare la classe? In entrambi i casi si avrà bisogno di cercare ogni posto in cui viene istanziata `NewsletterManager` e fare le modifiche. Naturalmente, il contenitore dei servizi fornisce una soluzione molto migliore:

In YAML, la sintassi speciale `@my_mailer` dice al contenitore di cercare un servizio chiamato `my_mailer` e di passare l'oggetto nel costruttore di `NewsletterManager`. In questo caso, tuttavia, il servizio specificato `my_mailer` deve esistere. In caso contrario, verrà lanciata un'eccezione. È possibile contrassegnare le proprie dipendenze come opzionali (sarà discusso nella prossima sezione).

L'utilizzo di riferimenti è uno strumento molto potente che permette di creare classi di servizi indipendenti con dipen-

denze ben definite. In questo esempio, il servizio `newsletter_manager` ha bisogno del servizio `my_mailer` per poter funzionare. Quando si definisce questa dipendenza nel contenitore dei servizi, il contenitore si prende cura di tutto il lavoro di istanziare degli oggetti.

## Usare Expression Language

Il contenitore di servizi supporta anche un"'espressione", che consente di iniettare valori molto specifici in un servizio.

Per esempio, su supponga di avere un terzo servizio (non mostrato qui), chiamato `mailer_configuration`, che ha un metodo `getMailerMethod()`, che restituisce una stringa come `sendmail` a seconda di una qualche configurazione. Si ricordi che il primo parametro del servizio `my_mailer` è la semplice stringa `sendmail`:

Invece di scrivere direttamente la stringa, come si può ottenere tale valore da `getMailerMethod()` del servizio `mailer_configuration`? Un possibile modo consiste nell'usare un'espressione:

Per approfondire la sintassi di Expression Language, vedere /components/expression_language/syntax.

In questo contesto, si ha accesso a due funzioni:

**`service`** restituisce un servizio dato (vedere l'esempio precedente);

**`parameter`** restituisce un parametro specifico (la sintassi è proprio come `service`)

Si ha anche accesso a `Symfony\Component\DependencyInjection\ContainerBuilder`, tramite una variabile `container`. Ecco un altro esempio:

Si possono usare espressioni in `arguments`, `properties`, come parametri con `configurator` e come parametri di `calls` (chiamate di metodi).

## Dipendenze opzionali: iniettare i setter

Iniettare dipendenze nel costruttore è un eccellente modo per essere sicuri che la dipendenza sia disponibile per l'uso. Se per una classe si hanno dipendenze opzionali, allora l"'iniezione dei setter" può essere una scelta migliore. Significa iniettare la dipendenza utilizzando una chiamata di metodo al posto del costruttore. La classe sarà simile a questa:

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Iniettare la dipendenza con il metodo setter, necessita solo di un cambio di sintassi:

**Note:** Gli approcci presentati in questa sezione sono chiamati "iniezione del costruttore" e "iniezione del setter". Il contenitore dei servizi di Symfony supporta anche "iniezione di proprietà".

### Iniettare la richiesta

A partire da Symfony 2.4, invece di iniettare il servizio `request`, si dovrebbe iniettare il servizio `request_stack` e accedere alla richiesta con il metodo **:method:'Symfony\\Component\\HttpFoundation\\RequestStack::getCurrentRequest'**:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\HttpFoundation\RequestStack;

class NewsletterManager
{
    protected $requestStack;

    public function __construct(RequestStack $requestStack)
    {
        $this->requestStack = $requestStack;
    }

    public function anyMethod()
    {
        $request = $this->requestStack->getCurrentRequest();
        // ... fare qualcosa con la richiesta
    }

    // ...
}
```

Ora, basta iniettare `request_stack`, che si comporta come un normale servizio:

---

**Tip:** Se si definisce un controllore come servizio, si può ottenere l'oggetto `Request` senza iniettare il contenitore, passandolo come parametro di un metodo azione. Vedere *La Request come parametro del controllore* per maggiori dettagli.

---

## Rendere opzionali i riferimenti

A volte, uno dei servizi può avere una dipendenza opzionale, il che significa che la dipendenza non è richiesta al fine di fare funzionare correttamente il servizio. Nell'esempio precedente, il servizio `my_mailer` *deve* esistere, altrimenti verrà lanciata un'eccezione. Modificando la definizione del servizio `newsletter_manager`, è possibile rendere questo riferimento opzionale. Il contenitore inietterà se esiste e in caso contrario non farà nulla:

In YAML, la speciale sintassi `@?` dice al contenitore dei servizi che la dipendenza è opzionale. Naturalmente, `NewsletterManager` deve essere scritto per consentire una dipendenza opzionale:

```
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

## Servizi del nucleo di Symfony e di terze parti

Dal momento che Symfony e tutti i bundle di terze parti configurano e recuperano i loro servizi attraverso il contenitore, si possono accedere facilmente o addirittura usarli nei propri servizi. Per mantenere le cose semplici, Symfony per impostazione predefinita non richiede che i controllori siano definiti come servizi. Inoltre Symfony inietta l'intero contenitore dei servizi nel controllore. Ad esempio, per gestire la memorizzazione delle informazioni su una sessione utente, Symfony fornisce un servizio `session`, a cui è possibile accedere dentro a un controllore standard, come segue:

```
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

In Symfony, si potranno sempre utilizzare i servizi forniti dal nucleo di Symfony o dai bundle di terze parti per eseguire funzionalità come la resa di template (`templating`), l'invio di email (`mailer`), o l'accesso a informazioni sulla richiesta (`request`).

Questo possiamo considerarlo come un ulteriore passo in avanti con l'utilizzo di questi servizi all'interno di servizi che si è creato per l'applicazione. Andiamo a modificare `NewsletterManager` per usare il reale servizio `mailer` di Symfony (al posto del finto `my_mailer`). Si andrà anche a far passare il servizio con il motore dei template al `NewsletterManager` in modo che possa generare il contenuto dell'email tramite un template:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Templating\EngineInterface;

class NewsletterManager
{
    protected $mailer;

    protected $templating;

    public function __construct(
        \Swift_Mailer $mailer,
        EngineInterface $templating
    ) {
        $this->mailer = $mailer;
        $this->templating = $templating;
    }

    // ...
}
```

La configurazione del contenitore dei servizi è semplice:

Il servizio `newsletter_manager` ora ha accesso ai servizi del nucleo `mailer` e `templating`. Questo è un modo comune per creare servizi specifici all'applicazione, in grado di sfruttare la potenza di numerosi servizi presenti nel framework.

---

**Tip:** Assicurarsi che la voce `swiftmailer` appaia nella configurazione dell'applicazione. Come è stato accennato in *Importare la configurazione attraverso estensioni del contenitore*, la chiave `swiftmailer` invoca l'estensione del servizio da `SwiftmailerBundle`, il quale registra il servizio `mailer`.

---

### I tag

Allo stesso modo con cui il post di un blog su web viene etichettato con cose tipo "Symfony" o "PHP", anche i servizi configurati nel contenitore possono essere etichettati. Nel contenitore dei servizi, un tag implica che si intende utilizzare il servizio per uno scopo specifico. Si prenda il seguente esempio:

Il tag `twig.extension` è un tag speciale che `TwigBundle` utilizza durante la configurazione. Dando al servizio il tag `twig.extension`, il bundle sa che il servizio `foo.twig.extension` dovrebbe essere registrato come estensione Twig. In altre parole, Twig cerca tutti i servizi etichettati con `twig.extension` e li registra automaticamente come estensioni.

I tag, quindi, sono un modo per dire a Symfony o a un altro bundle di terze parti che il servizio dovrebbe essere registrato o utilizzato in un qualche modo speciale dal bundle.

Per una lista completa dei tag disponibili in Symfony, dare un'occhiata a /reference/dic_tags. Ognuno di essi ha un differente effetto sul servizio e molti tag richiedono parametri aggiuntivi (oltre al solo `name` del parametro).

## Debug dei servizi

Si può sapere quali servizi sono registrati nel contenitore, usando la console. Per mostrare tutti i servizi e le relative classi, eseguire:

```
$ php app/console debug:container
```

New in version 2.6: Prima di Symfony 2.6, questo comando si chiamava `container:debug`.

Vengono mostrati solo i servizi pubblici, ma si possono vedere anche quelli privati:

```
$ php app/console container:debug --show-private
```

---

**Note:** Se un servizio privato è usato solo come parametro di *un solo* altro servizio, non sarà mostrato dal comando `container:debug`, anche usando l'opzione `--show-private`. vedere servizi privati in linea per maggiori dettagli.

---

Si possono ottenere informazioni più dettagliate su un singolo servizio, specificando il suo id:

```
$ php app/console container:debug my_mailer
```

## Saperne di più

- /components/dependency_injection/parameters
- /components/dependency_injection/compilation
- /components/dependency_injection/definitions
- /components/dependency_injection/factories
- /components/dependency_injection/parentservices
- /components/dependency_injection/tags
- /cookbook/controller/service
- /cookbook/service_container/scopes

- /cookbook/service_container/compiler_passes
- /components/dependency_injection/advanced

CHAPTER 19

# Prestazioni

Symfony è veloce, senza alcuna modifica. Ovviamente, se occorre maggiore velocità, ci sono molti modi per rendere Symfony ancora più veloce. In questo capitolo saranno esplorati molti dei modi più comuni e potenti per rendere un'applicazione Symfony più veloce.

## Usare una cache bytecode (p.e. APC)

Una delle cose migliori (e più facili) che si possono fare per migliorare le prestazioni è quella di usare una cache bytecode. L'idea di una cache bytecode è di rimuove l'esigenza di dover ricompilare ogni volta il codice sorgente PHP. Ci sono numerose cache bytecode disponibili, alcune delle quali open source. Dalla versione 5.5, PHP include OPcache. Per versioni precedenti, la cache più usata è probabilmente APC.

Usare una cache bytecode non ha alcun effetto negativo, e Symfony è stato progettato per avere prestazioni veramente buone in questo tipo di ambiente.

### Ulteriori ottimizzazioni

Le cache bytecode solitamente monitorano i cambiamenti dei file sorgente. Questo assicura che, se la sorgente del file cambia, il bytecode sia ricompilato automaticamente. Questo è molto conveniente, ma ovviamente ha un costo.

Per questa ragione, alcune cache bytecode offrono un'opzione per disabilitare questi controlli. Ovviamente, quando si disabilitano i controlli, sarà compito dell'amministratore del server assicurarsi che la cache sia svuotata a ogni modifica dei file sorgente. Altrimenti, gli aggiornamenti eseguiti non saranno mostrati.

Per esempio, per disabilitare questi controlli in APC, aggiungere semplicemente `apc.stat=0` al file di configurazione php.ini.

## Usare un autoloader con cache (p.e. `ApcUniversalClassLoader`)

Per impostazione predefinita, Symfony standard edition usa `UniversalClassLoader` nel file autoloader.php. Questo autoloader è facile da usare, perché troverà automaticamente ogni nuova classe inserita nelle cartelle registrate.

Sfortunatamente, questo ha un costo, perché il caricatore itera tutti gli spazi dei nomi configurati per trovare un particolare file, richiamando `file_exists` finché non trova il file cercato.

La soluzione più semplice è dire a Composer di costruire una "mappa di classi" (cioè un grosso array con le posizioni di tutte le classi). Lo si può fare da linea di comando e potrebbe diventare parte del processo di deploy:

```
$ composer dump-autoload --optimize
```

Internamente, costruisce un grosso array di mappature delle classi in `vendor/composer/autoload_classmap.php`.

## Cache dell'autoloader con APC

Un'altra soluzione è mettere in cache la posizione di ogni classe, dopo che è stata trovata per la prima volta. Symfony dispone di una classe, `Symfony\Component\ClassLoader\ApcClassLoader`, che si occupa proprio di questo. Per usarla, basta adattare il file del front controller. Se si usa la Standard Edition, il codice è già disponibile nel file, ma commentato:

```
// app.php
// ...

$loader = require_once __DIR__.'/../app/bootstrap.php.cache';

// Usa APC per aumentare le prestazioni dell'auto-caricamento
// Cambiare 'sf2' con il prefisso desiderato
// per prevenire conflitti di chiavi con altre applicazioni
/*
$loader = new ApcClassLoader('sf2', $loader);
$loader->register(true);
*/

// ...
```

Per maggiori dettagli, vedere /components/class_loader/cache_class_loader.

---

**Note:** Quando si usa l'autoloader APC, se si aggiungono nuove classi, saranno trovate automaticamente e tutto funzionerà come prima (cioè senza motivi per "pulire" la cache). Tuttavia, se si cambia la posizione di un particolare spazio dei nomi o prefisso, occorrerà pulire la cache di APC. Altrimenti, l'autoloader cercherà ancora la classe nella vecchia posizione per tutte le classi in quello spazio dei nomi.

---

## Usare i file di avvio

Per assicurare massima flessibilità e riutilizzo del codice, le applicazioni Symfony sfruttano una varietà di classi e componenti di terze parti. Ma il caricamento di tutte queste classi da diversi file a ogni richiesta può risultate in un overhead. Per ridurre tale overhead, Symfony Standard Edition fornisce uno script per generare i cosiddetti file di avvio, che consistono in definizioni di molte classi in un singolo file. Includendo questo file (che contiene una copia di molte classi del nucleo), Symfony non avrà più bisogno di includere alcuno dei file sorgente contenuti nelle classi stesse. Questo riduce un po' la lettura/scrittura su disco.

Se si usa Symfony Standard Edition, probabilmente si usa già un file di avvio. Per assicurarsene, aprire il front controller (solitamente `app.php`) e verificare che sia presente la seguente riga:

```
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Si noti che ci sono due svantaggi nell'uso di un file di avvio:

- il file deve essere rigenerato ogni volta che cambia una delle sorgenti originali (p.e. quando si aggiorna il sorgente di Symfony o le librerie dei venditori);
- durante il debug, occorre inserire i breakpoint nel file di avvio.

Se si usa Symfony Standard Edition, il file di avvio è ricostruito automaticamente dopo l'aggiornamento delle librerie dei venditori, tramite il comando `composer install`.

## File di avvio e cache bytecode

Anche usando una cache bytecode, le prestazioni aumenteranno con l'uso di un file di avvio, perché ci saranno meno file da monitorare per i cambiamenti. Certamente, se questa caratteristica è disabilitata nella cache bytecode (p.e. con `apc.stat=0` in APC), non c'è più ragione di usare un file di avvio.

# Index

## V