Manual de Silex

Release 0.0.0

Traducido por Nacho Pacheco

November 03, 2011

Índice general

Parte I

Introducción

Silex es una microplataforma *PHP* para *PHP* 5.3. Está construida sobre los hombros de *Symfony2* y *Pimple* además de inspirada en *Sinatra*.

Una microplataforma proporciona la base para construir aplicaciones simples de un solo archivo. *Silex* pretende ser:

- *Concisa*: *Silex* expone una *API* intuitiva, concisa y te divertirás usándola.
- *Extensible*: *Silex* cuenta con un sistema de extensión en torno al microcontenedor de servicios *Pimple* que lo hace aún más fácil de encajar con bibliotecas de terceros.
- *Probable*: *Silex* utiliza el `HttpKernel` de *Symfony2* el cual te abstrae de las peticiones y respuestas. Esto hace que sea muy fácil probar las aplicaciones y la propia plataforma. También respeta la especificación *HTTP* y alienta su uso adecuado.

En pocas palabras, tú defines controladores y asignas las rutas, en un solo paso.

¡Comencemos!

```
require_once __DIR__.'/silex.phar';

$app = new Silex\Application();

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello '.$app->escape($name);
});

$app->run();
```

Todo lo que necesitas para acceder a la plataforma es incluir `silex.phar`. Este archivo `phar` (archivo *PHP*) se encargará del resto.

A continuación defines una ruta a `/hello/{name}` que corresponde con las peticiones `GET`. Cuando la ruta coincide, se ejecuta la función y el valor de retorno se devuelve al cliente.

Por último, se ejecuta la aplicación. Visita `/hello/world` para ver el resultado. ¡Es así de fácil!

Instalar *Silex* es tan fácil como lo puedas obtener. ¡Descarga el archivo silex.phar y ya está!

Parte II

Usándola

Este capítulo describe cómo utilizar *Silex*.

CAPÍTULO 1

Arranque

Para incluir *Silex* todo lo que tienes que hacer es requerir el archivo `silex.phar` y crear una instancia de `Silex\Application`. Después de definir tu controlador, llama al método `run` en tu aplicación:

```
require_once __DIR__.'/silex.phar';

$app = new Silex\Application();

// definiciones

$app->run();
```

Otra cosa que tienes que hacer es configurar tu servidor web. Si estás usando *Apache* puedes utilizar un `.htaccess` para esto.

```
<IfModule mod_rewrite.c>
    Options -MultiViews

    RewriteEngine On
    #RewriteBase /ruta/a/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Nota: Si tu sitio no está a nivel raíz del servidor web, tienes que descomentar la declaración `RewriteBase` y ajustar la ruta para que apunte al directorio, relativo a la raíz del servidor web.

Truco: Cuando desarrollas un sitio web, posiblemente desees activar el modo de depuración para facilitar la corrección de errores:

```
$app['debug'] = true;
```

CAPÍTULO 2

Enrutado

En *Silex* defines una ruta y el controlador que se invocará cuando dicha ruta concuerde

Un patrón de ruta se compone de:

- *Pattern*: El patrón de ruta define una ruta que apunta a un recurso. El patrón puede incluir partes variables y tú podrás establecer los requisitos con expresiones regulares.
- *Method*: Uno de los siguientes métodos *HTTP*: `GET`, `POST`, `PUT` `DELETE`. Este describe la interacción con el recurso. Normalmente sólo se utilizan `GET` y `POST`, pero, también es posible utilizar los otros.

El controlador se define usando un cierre de esta manera:

```
function () {
    // hacer algo
}
```

Los cierres son funciones anónimas que pueden importar el estado desde fuera de su definición. Esto es diferente de las variables globales, porque el estado exterior no tiene que ser global. Por ejemplo, podrías definir un cierre en una función e importar variables locales desde esa función.

Nota: Los cierres que no importan el ámbito se conocen como lambdas. Debido a que en *PHP* todas las funciones anónimas son instancias de la clase `Closure`, no vamos a hacer una distinción aquí.

El valor de retorno del cierre se convierte en el contenido de la página.

También existe una forma alterna para definir controladores que utilizan un método de clase. La sintaxis para esto es **NombreClase**::**nombreMétodo**. También son posibles los métodos estáticos.

2.1 Ejemplo de ruta `GET`

He aquí un ejemplo de una definición de ruta `GET`:

```
$blogPosts = array(
    1 => array(
        'date'      => '2011-03-29',
        'author'    => 'igorw',
        'title'     => 'Using Silex',
        'body'      => '...',
```

```
    ),
);

$app->get('/blog', function () use ($blogPosts) {
    $output = '';
    foreach ($blogPosts as $post) {
        $output .= $post['title'];
        $output .= '<br />';
    }

    return $output;
});
```

Al visitar `/blog` devolverá una lista con los títulos de los comunicados en el `blog`. La declaración `use` significa algo diferente en este contexto. Esta instruye al cierre a importar la variable `comunicadosBLog` desde el ámbito externo. Esto te permite utilizarla dentro del cierre.

2.2 Enrutado dinámico

Ahora, puedes crear otro controlador para ver comunicados individuales del `blog`:

```
$app->get('/blog/show/{id}', function (Silex\Application $app, $id) use ($blogPosts) {
    if (!isset($blogPosts[$id])) {
        $app->abort(404, "Post $id does not exist.");
    }

    $post = $blogPosts[$id];

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

Esta definición de ruta tiene una parte variable `{id}` que se pasa al cierre.

Cuando el comunicado no existe, estamos usando `abort()` para detener la petición inicial. En realidad, se produce una excepción, la cual veremos cómo manejar más adelante.

2.3 Ejemplo de ruta `POST`

Las rutas `POST` denotan la creación de un recurso. Un ejemplo de esto es un formulario de comentarios. Vamos a utilizar la función `mail` para enviar un correo electrónico:

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

$app->post('/feedback', function (Request $request) {
    $message = $request->get('message');
    mail('feedback@yoursite.com', '[YourSite] Feedback', $message);

    return new Response('Thank you for your feedback!', 201);
});
```

Es bastante sencillo.

Nota: Hay un *SwiftmailerServiceProvider* (Página 93) incluido que puedes utilizar en lugar de `mail()`.

La `petición` actual es inyectada al `cierre` automáticamente por *Silex* gracias al indicador de tipo. Es una instancia de Request, por lo que puedes recuperar las variables usando el método `get` de la petición.

En lugar de devolver una cadena regresamos una instancia de Response. Esto nos permite fijar un código de estado *HTTP*, en este caso configurado a `201 Creado`.

Nota: *Silex* siempre utiliza internamente una `Respuesta`, la convierte a cadenas para respuestas con código de estado `200 OK`.

2.4 Otros métodos

Puedes crear controladores para la mayoría de los métodos *HTTP*. Sólo tienes que llamar uno de estos métodos en tu aplicación: `get`, `post`, `put`, `delete`. También puedes llamar `match`, el cual coincidirá con todos los métodos:

```
$app->match('/blog', function () {
    ...
});
```

Entonces puedes restringir los métodos permitidos a través del método `method`:

```
$app->match('/blog', function () {
    ...
})
->method('PARCHE');
```

Puedes sincronizar varios métodos con un controlador utilizando la sintaxis de expresiones regulares:

```
$app->match('/blog', function () {
    ...
})
->method('PUT|POST');
```

Nota: El orden en que definas las rutas es importante. La primera ruta que coincida se utilizará, por lo tanto coloca tus rutas más genéricas en la parte inferior.

2.5 Variables de ruta

Como mostramos antes, puedes definir partes variables en una ruta, como esta:

```
$app->get('/blog/show/{id}', function ($id) {
    ...
});
```

También es posible tener más de una parte variable, basta con que encierres los argumentos coincidentes con los nombres de las partes variables:

```
$app->get('/blog/show/{postId}/{commentId}', function ($postId, $commentId) {
    ...
});
```

Si bien no se sugiere, también lo puedes hacer (ten en cuenta la conmutación de los argumentos):

```
$app->get('/blog/show/{postId}/{commentId}', function ($commentId, $postId) {
    ...
});
```

También puedes consultar la `Petición` actual y el objeto `Aplicación`:

```
$app->get('/blog/show/{id}', function (Application $app, Request $request, $id) {
    ...
});
```

Nota: Ten en cuenta que para los objetos `Aplicación` y `Petición`, *Silex* hace la inyección basándose en el indicador de tipo y no en el nombre de la variable:

```
$app->get('/blog/show/{id}', function (Application $foo, Request $bar, $id) {
    ...
});
```

2.6 Convertidores de variables de ruta

Antes de inyectar las variables de ruta en el controlador, puedes aplicar algunos convertidores:

```
$app->get('/user/{id}', function ($id) {
    // ...
})->convert('id', function ($id) { return (int) $id; });
```

Esto es útil cuando quieres convertir las variables de ruta a objetos, ya que permite reutilizar el código de conversión entre diferentes controladores:

```
$userProvider = function ($id) {
    return new User($id);
};

$app->get('/user/{user}', function (User $user) {
    // ...
})->convert('user', $userProvider);

$app->get('/user/{user}/edit', function (User $user) {
    // ...
})->convert('user', $userProvider);
```

La retrollamada al convertidor también recibe la `Petición` como segundo argumento:

```
$callback = function ($post, Request $request) {
    return new Post($request->attributes->get('slug'));
};

$app->get('/blog/{id}/{slug}', function (Post $post) {
    // ...
})->convert('post', $callback);
```

2.7 Requisitos

En algunos casos es posible que sólo desees detectar ciertas expresiones. Puedes definir los requisitos usando expresiones regulares llamando a `assert` en el objeto `Controller`, que es devuelto por los métodos de enrutado.

Lo siguiente se asegurará de que el argumento `id` es numérico, ya que `\d+` coincide con cualquier cantidad de dígitos:

```
$app->get('/blog/show/{id}', function ($id) {
    ...
})
->assert('id', '\d+');
```

También puedes encadenar estas llamadas:

```
$app->get('/blog/show/{postId}/{commentId}', function ($postId, $commentId) {
    ...
})
->assert('postId', '\d+')
->assert('commentId', '\d+');
```

2.8 valores predeterminados

Puedes definir un valor predeterminado para cualquier variable de ruta llamando a `value` en el objeto `Controlador`:

```
$app->get('/{pageName}', function ($pageName) {
    ...
})
->value('pageName', 'index');
```

Esto te permitirá coincidir `/`, en cuyo caso la variable `nombrePagina` tendrá el valor de `index`.

2.9 Rutas con nombre

Algunos proveedores (como `UrlGeneratorProvider`) pueden usar rutas con nombre. De manera predeterminada *Silex* generará un nombre de ruta para ti, el cual, en realidad, no puedes utilizar. Puedes dar un nombre a una ruta llamando a `bind` en el objeto `Controlador` devuelto por los métodos de enrutado:

```
$app->get('/', function () {
    ...
})
->bind('homepage');

$app->get('/blog/show/{id}', function ($id) {
    ...
})
->bind('blog_post');
```

Nota: Sólo tiene sentido nombrar rutas si utilizas proveedores que usan la `RouteCollection`.

Filtros `before` y `after`

Silex te permite ejecutar código antes y después de cada petición. Esto ocurre a través de los filtros `before` y `after`. Todo lo que necesitas hacer es pasar un cierre:

```
$app->before(function () {
    // configurar
});

$app->after(function () {
    // destruir
});
```

El filtro `before` tiene acceso a la `Petición` actual, y puede provocar un cortocircuito en toda la reproducción, devolviendo una `Respuesta`:

```
$app->before(function (Request $request) {
    // redirige al usuario al formulario de acceso si el recurso accedido está protegido
    if (...) {
        return new RedirectResponse('/login');
    }
});
```

El filtro `after` tiene acceso a la `Petición` y a la `Respuesta`:

```
$app->after(function (Request $peticion, Response $respuesta) {
    // ajusta la respuesta
});
```

Nota: Los filtros sólo los ejecuta la `Petición` "maestra".

CAPÍTULO 4

Manipuladores de error

Si alguna parte de tu código produce una excepción de la que desees mostrar algún tipo de página de error al usuario. Esto es lo que hacen los manipuladores de error. También puedes utilizarlos para hacer cosas adicionales, tal como registrar sucesos.

Para registrar un manipulador de error, pasa un cierre al método `error` el cual toma un argumento `Exception` y devuelve una respuesta:

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

$app->error(function (\Exception $e, $code) {
    return new Response('We are sorry, but something went terribly wrong.', $code);
});
```

También puedes comprobar si hay errores específicos usando el argumento `$code`, y manejándolo de manera diferente:

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

$app->error(function (\Exception $e, $code) {
    switch ($code) {
        case 404:
            $message = 'The requested page could not be found.';
            break;
        default:
            $message = 'We are sorry, but something went terribly wrong.';
    }

    return new Response($message, $code);
});
```

Si deseas configurar el registro puedes utilizar un manipulador de errores independiente para eso. Sólo asegúrate de registrarlo antes que los manipuladores que responden a error, porque una vez que se devuelve una respuesta, se omiten los siguientes manipuladores.

Nota: *Silex* viene con un proveedor para Monolog el cual maneja el registro de errores. Échale un vistazo al capítulo *Proveedores* (Página 47) para más detalles.

Truco: *Silex* viene con un controlador de errores predeterminado que muestra un mensaje de error detallado con el seguimiento de la pila cuando **debug** es `true`, y de otra manera un mensaje de error simple. Los manipuladores de error registrados a través del método `error()` siempre tienen prioridad, pero puedes mantener agradables los mensajes de error de depuración cuando se enciende con algo como esto:

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

$app->error(function (\Exception $e, $code) use ($app) {
    if ($app['debug']) {
        return;
    }

    // lógica para manejar el error y devolver una respuesta
});
```

A los manipuladores de error también se les llama cuando utilizas `abort` para anular tempranamente una petición:

```
$app->get('/blog/show/{id}', function (Silex\Application $app, $id) use ($blogPosts) {
    if (!isset($blogPosts[$id])) {
        $app->abort(404, "Post $id does not exist.");
    }

    return new Response(...);
});
```

CAPÍTULO 5

Redirigiendo

Puedes redirigir a otra página devolviendo una respuesta de redirección, la cual puedes crear mediante una llamada al método `redirect`:

```
use Silex\Application;

$app->get('/', function (Silex\Application $app) {
    return $app->redirect('/hello');
});
```

Esto redirigirá de `/` a `/hello`.

CAPÍTULO 6

Seguridad

Asegúrate de proteger tu aplicación contra ataques.

6.1 Escapando

Cuando reproduces la entrada del usuario (ya sea en las variables `GET/POST` o en variables obtenidas desde la petición), tendrás que asegurarte de escaparlas correctamente, para evitar ataques que exploten vulnerabilidades del sistema.

- **Escapando HTML**: *PHP* proporciona la función `htmlspecialchars` para esto. *Silex* ofrece un atajo, el método `escape`:

  ```
  $app->get('/name', function (Silex\Application $app) {
      $name = $app['request']->get('name');
      return "You provided the name {$app->escape($name)}.";
  });
  ```

  Si utilizas el motor de plantillas *Twig* debes usar su escape o incluso mecanismos de autoescape.

- **Escapando JSON**: Si deseas proporcionar datos en formato *JSON* debes utilizar la función *PHP* `json_encode`:

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

  $app->get('/name.json', function (Silex\Application $app) {
      $name = $app['request']->get('name');
      return new Response(
          json_encode(array('name' => $name)),
          200,
          array('Content-Type' => 'application/json')
      );
  });
  ```

Consola

Silex incluye una consola ligera para actualizar a la última versión.

Para saber qué versión de *Silex* estás utilizando, invoca a `silex.phar` en la línea de ordenes con `version` como argumento:

```
$ php silex.phar version
Silex version 0a243d3 2011-04-17 14:49:31 +0200
```

Para comprobar si estás utilizando la última versión, ejecuta la orden `check`:

```
$ php silex.phar check
```

Para actualizar `silex.phar` a la última versión, invoca la orden `update`:

```
$ php silex.phar update
```

Esto descargará automáticamente un nuevo `silex.phar` desde `silex.sensiolabs.org` y sustituirá al actual.

Trampas

Hay algunas cosas que pueden salir mal. Aquí vamos a tratar de esbozar las más frecuentes.

8.1 Configuración de *PHP*

Ciertas distribuciones de *PHP*, de manera predeterminada tienen configuración `Phar` restrictiva. Ajustar lo siguiente puede ayudar.

```
phar.readonly = Off
phar.require_hash = Off
detect_unicode = Off
```

Si estás en `Suhosin` también tendrás que fijar lo siguiente:

```
suhosin.executor.include.whitelist = phar
```

Nota: El *PHP* de *Ubuntu* viene con *Suhosin*, así que si estás usando *Ubuntu*, necesitarás este cambio.

8.2 Fallo `Phar-Stub`

Algunas instalaciones de *PHP* tienen un error que arroja una `PharException` cuando tratas de incluir el `Phar`. También te dirá que `Silex\Application` no se pudo encontrar. Una solución es usar la siguiente línea:

```
require_once 'phar://'.__DIR__.'/silex.phar/autoload.php';
```

La causa exacta de esta emisión no se ha podido determinar todavía.

8.3 Fallo en el cargador de `ioncube`

El cargador de `Ioncube` es una extensión que puede decodificar archivos *PHP* codificados. Desafortunadamente, las versiones antiguas (anteriores a la versión 4.0.9) no están funcionando bien con archivos `phar`. Debes actualizar tu `Ioncube Loder` a la versión 4.0.9 o más reciente o desactivarla comentando o eliminando esta línea en tu archivo `php.ini`:

```
zend_extension = /usr/lib/php5/20090626+lfs/ioncube_loader_lin_5.3.so
```

Configuración *IIS*

Si estás utilizando el `Internet Information Services` de *Windows*, puedes utilizar de ejemplo este archivo `web.config`:

```
<?xml version="1.0"?>
<configuration>
    <system.webServer>
        <defaultDocument>
            <files>
                <clear />
                <add value="index.php" />
            </files>
        </defaultDocument>
        <rewrite>
            <rules>
                <rule name="Silex Front Controller" stopProcessing="true">
                    <match url="^(.*)$" ignoreCase="false" />
                    <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" ignoreCase="false" negate=
                    </conditions>
                    <action type="Rewrite" url="index.php" appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Parte III

Servicios

Silex no es sólo una microplataforma. También es un microcontenedor de servicios. Esto se logra extendiendo a Pimple que ofrece el bondadoso servicio en sólo 44 *NCLOC*.

Inyección de dependencias

Nota: Puedes omitir esto si ya sabes lo que es la inyección de dependencias.

La inyección de dependencias es un patrón de diseño dónde pasas las dependencias a servicios en lugar de crearlas desde dentro del servicio o depender de variables globales. Esto generalmente lleva a un código disociado, reutilizable, flexible y fácil de probar.

He aquí un ejemplo de una clase que toma un objeto `Usuario` y lo guarda como un archivo en formato *JSON*:

```
class JsonUserPersister
{
    private $basePath;

    public function __construct($basePath)
    {
        $this->basePath = $basePath;
    }

    public function persist(User $user)
    {
        $data = $user->getAttributes();
        $json = json_encode($data);
        $filename = $this->basePath.'/'.$user->id.'.json';
        file_put_contents($filename, $json, LOCK_EX);
    }
}
```

En este sencillo ejemplo la dependencia es la propiedad `basePath`. Esta se pasa al constructor. Esto significa que puedes crear varias instancias independientes con diferentes rutas base. Por supuesto, las dependencias no tienen que ser simples cadenas de texto. Muy a menudo estas, de hecho, se encuentran en otros servicios.

10.1 Contenedor

Un `ID` o contenedor de servicio es responsable de crear y almacenar los servicios. Este puede crear recurrentemente dependencias de los servicios solicitados e inyectarlos. Lo hace de manera diferida, lo cual significa que un servicio sólo se crea cuando realmente se necesita.

La mayoría de los contenedores son muy complejos y se configuran a través de archivos *XML* o *YAML*.

Pimple es diferente

CAPÍTULO 11

Pimple

Pimple, probablemente, es el más simple contenedor de servicios que hay. Se usan exhaustivamente los cierres que implementan la interfaz `ArrayAccess`.

Vamos a empezar por crear una nueva instancia de *Pimple* – y puesto que `Silex\application` extiende a *Pimple* todo esto se aplica a *Silex* también:

```
$contenedor = new Pimple();
```

o:

```
$app = new Silex\Application();
```

11.1 Parámetros

Puedes establecer los parámetros (los cuales suelen ser cadenas) estableciendo una clave en el arreglo del contenedor:

```
$app['algún_parámetro'] = 'valor';
```

La clave del arreglo puede ser cualquier cosa, por convención se utilizan puntos para denominar los espacios de nombres:

```
$app['activo.anfitrión'] = 'http://cdn.misitio.com/';
```

Es posible leer valores de parámetros con la misma sintaxis:

```
echo $app['algún_parámetro'];
```

11.2 Definiendo servicios

La definición de servicios no es diferente de la definición de parámetros. Sólo tienes que establecer una clave en el arreglo del contenedor a un cierre. Sin embargo, cuando recuperes el servicio, se ejecuta el cierre. Esto permite la creación diferida de servicios:

```
$app['some_service'] = function () {
    return new Service();
};
```

Y para recuperar el servicio, utiliza:

```
$service = $app['some_service'];
```

Cada vez que llames a `$app['some_service']`, se crea una nueva instancia del servicio.

11.3 Servicios compartidos

Posiblemente desees utilizar la misma instancia de un servicio a través de todo el código. A fin de que puedas hacer *compartido* un servicio:

```
$app['some_service'] = $app->share(function () {
    return new Service();
});
```

Esto creará el servicio en la primera invocación, y luego devolverá la instancia existente en cualquier acceso posterior.

11.4 Accediendo al contenedor desde un cierre

En muchos casos, desearás acceder al contenedor de servicios dentro de un cierre en la definición del servicio. Por ejemplo, al recuperar servicios de los que depende el servicio actual.

Debido a esto, el contenedor se pasa al cierre como argumento:

```
$app['some_service'] = function ($app) {
    return new Service($app['some_other_service'], $app['some_service.config']);
};
```

Aquí puedes ver un ejemplo de Inyección de dependencias. `some_service` depende de `some_other_service` y toma `some_service.config` como opciones de configuración. La dependencia sólo se crea cuando accedes a `some_service`, y es posible reemplazar cualquiera de las dependencias simplemente remplazando esas definiciones.

Nota: Esto también trabaja para los servicios compartidos.

11.5 Cierres protegidos

Debido a que el contenedor ve los cierres cómo fábricas de servicios, siempre se deben ejecutar cuando los leas.

En algunos casos, sin embargo, deseas almacenar un cierre como un parámetro, de modo que lo puedas recuperar y ejecutar tú mismo – con tus propios argumentos.

Esta es la razón por la cual *Pimple* te permite proteger tus cierres de ser ejecutados, usando el método `protect`:

```
$app['closure_parameter'] = $app->protect(function ($a, $b) {
    return $a + $b;
});

// no debe ejecutar el cierre
$add = $app['closure_parameter'];

// llamándolo ahora
echo $add(2, 3);
```

Ten en cuenta que los cierres protegidos no tienen acceso al contenedor.

CAPÍTULO 12

Servicios básicos

Silex define una serie de servicios que puedes utilizar o reemplazar. Es probable que no quieras meterte con la mayoría de ellos.

- **request**: Contiene el objeto `Petición` actual, el cual es una instancia de Request. ¡Este proporciona acceso a los parámetros `GET`, `POST` y mucho más!

  Ejemplo de uso:

  ```
  $id = $app['request']->get('id');
  ```

  Este sólo está disponible cuando se está sirviendo la petición, sólo puedes acceder a él desde dentro de un controlador, antes del filtro, después del filtro o al manejar algún error.

- **autoloader**: Este servicio te proporciona un UniversalClassLoader que ya está registrado. Puedes registrar prefijos y espacios de nombres en él.

  Ejemplo de uso, autocargando clases *Twig*:

  ```
  $app['autoloader']->registerPrefix('Twig_', $app['twig.class_path']);
  ```

  Para más información, consulta la documentación del autocargador de Symfony2.

- **routes**: El RouteCollection utilizado internamente. Puedes agregar, modificar y leer rutas.

- **controllers**: La `Silex\ControllerCollection` utilizada internamente. Consulta el capítulo *Funcionamiento interno* (Página 73) para más información.

- **dispatcher**: El EventDispatcher utilizado internamente. Es el núcleo del sistema *Symfony2* y se utiliza un poco en *Silex*.

- **resolver**: El ControllerResolver utilizado internamente. Se encarga de ejecutar el controlador con los argumentos adecuados.

- **kernel**: El HttpKernel utilizado internamente. El `HttpKernel` es el corazón de *Symfony2*, este toma una Petición como entrada y devuelve una Respuesta como salida.

- **request_context**: El contexto de la petición es una representación simplificada de la petición que utilizan el `Router` y el `UrlGenerator`.

- **exception_handler**: El controlador de excepciones es el controlador predeterminado que se utiliza cuando no registras uno a través del método `error()` o si el controlador no devuelve una Respuesta. Lo puedes desactivar con `unset($app['exception_handler'])`.

Nota: Todos estos servicios básicos de *Silex* son compartidos.

Parámetros básicos

- **request.http_port** (opcional): Te permite redefinir el puerto predeterminado para direcciones que no sean *HTTPS*. Si la petición actual es *HTTP*, siempre utiliza el puerto actual.

  El predeterminado es 80.

  Este parámetro lo puede utilizar el `UrlGeneratorProvider`.

- **request.https_port** (opcional): Te permite redefinir el puerto predeterminado para direcciones que no sean *HTTPS*. Si la petición actual es *HTTPS*, siempre usará el puerto actual.

  Predeterminado a 443.

  Este parámetro lo puede utilizar el `UrlGeneratorProvider`.

- **debug** (opcional): Indica si o no se ejecuta la aplicación en modo de depuración.

  El valor predeterminado es `false`.

- **charset** (opcional): El juego de caracteres a usar para las Respuestas.

  Por omisión es UTF-8.

Parte IV

Proveedores

Los proveedores permiten al desarrollador reutilizar partes de una aplicación en otra. Silex ofrece dos tipos de proveedores definidos por dos interfaces: `ServiceProviderInterface` para los servicios y `ControllerProviderInterface` para los controladores.

CAPÍTULO 14

Proveedores de servicios

14.1 Cargando proveedores

Con el fin de cargar y usar un proveedor de servicios, lo debes registrar en la aplicación:

```
$app = new Silex\Application();

$app->register(new Acme\DatabaseServiceProvider());
```

También puedes proporcionar algunos parámetros como segundo argumento. Estos se deben establecer **antes** de registrar al proveedor:

```
$app->register(new Acme\DatabaseServiceProvider(), array(
    'database.dsn'      => 'mysql:host=localhost;dbname=myapp',
    'database.user'     => 'root',
    'database.password' => 'secret_root_password',
));
```

14.2 Convenciones

Necesitas tener cuidado con el orden en que haces ciertas cosas cuando interactúas con proveedores. Sólo sigue estas reglas:

- Rutas de clase (para el cargador automático) las debes definir **antes** de registrar al proveedor. Pasar esta como segundo argumento a `Application::register` también califica, ya que en primer lugar establece los parámetros pasados.

  Razón: El proveedor configurará el cargador automático al momento de registrar al proveedor. Si no estableces la ruta de la clase en ese punto, no puedes registrar un cargador automático.

- La redefinición de servicios existentes debe ocurrir **después** de haber registrado al proveedor.

  Razón: Si los servicios ya existen, el proveedor los sobrescribirá.

- Puedes configurar los parámetros en cualquier momento antes de acceder al servicio.

Asegúrate de que te adhieres a este comportamiento al crear tus propios proveedores.

14.3 Proveedores integrados

Hay algunos proveedores que obtienes fuera de la caja. Todos estos están dentro del espacio de nombres `Silex\Provider`.

- *DoctrineServiceProvider* (Página 85)
- *MonologServiceProvider* (Página 89)
- *SessionServiceProvider* (Página 91)
- *SwiftmailerServiceProvider* (Página 93)
- *SymfonyBridgesServiceProvider* (Página 95)
- *TwigServiceProvider* (Página 101)
- *TranslationServiceProvider* (Página 97)
- *UrlGeneratorServiceProvider* (Página 103)
- *ValidatorServiceProvider* (Página 105)
- *HttpCacheServiceProvider* (Página 107)

14.4 Creando un proveedor

Los proveedores deben implementar el `Silex\ServiceProviderInterface`:

```
interface ServiceProviderInterface
{
    function register(Application $app);
}
```

Esto es muy sencillo, basta con crear una nueva clase que implemente el método `register`. En este método debes definir los servicios de la aplicación que pueden usar otros servicios y parámetros.

He aquí un ejemplo de tal proveedor:

```
namespace Acme;

use Silex\Application;
use Silex\ServiceProviderInterface;

class HelloServiceProvider implements ServiceProviderInterface
{
    public function register(Application $app)
    {
        $app['hello'] = $app->protect(function ($name) use ($app) {
            $default = $app['hello.default_name'] ? $app['hello.default_name'] : '';
            $name = $name ?: $default;

            return 'Hello '.$app->escape($name);
        });
    }
}
```

Esta clase proporciona un servicio, `hello`, el cual es un cierre protegido. Este toma un argumento nombre y devolverá `hello.default_name` si no se da un nombre. Si además falta el predeterminado, utilizará una cadena vacía.

Ahora puedes utilizar este proveedor de la siguiente manera:

```
$app = new Silex\Application();

$app->register(new Acme\HelloServiceProvider(), array(
    'hello.default_name' => 'Igor',
));

$app->get('/hello', function () use ($app) {
    $name = $app['request']->get('name');

    return $app['hello']($name);
});
```

En este ejemplo estamos obteniendo el parámetro `name` de la cadena de consulta, por lo que la ruta de la petición tendría que ser `/hello?name=Fabien`.

14.5 Cargando clases

Los proveedores son ideales para atarlos en bibliotecas externas como puedes ver mirando a `MonologServiceProvider` y `TwigServiceProvider`. Si la biblioteca es decente y sigue el Estándar de nomenclatura PSR-0 o la convención de nomenclatura *PEAR*, es posible cargar automáticamente las clases con el `UniversalClassLoader`.

Como se describe en el capítulo *Servicios*, hay un *cargador automático* de servicios que puedes utilizar para esto.

He aquí un ejemplo de cómo usarlo (en base a Buzz):

```
namespace Acme;

use Silex\Application;
use Silex\ServiceProviderInterface;

class BuzzServiceProvider implements ServiceProviderInterface
{
    public function register(Application $app)
    {
        $app['buzz'] = $app->share(function () { ... });

        if (isset($app['buzz.class_path'])) {
            $app['autoloader']->registerNamespace('Buzz', $app['buzz.class_path']);
        }
    }
}
```

Esto te permite proporcionar sólo la ruta de clases como una opción al registrar el proveedor:

```
$app->register(new BuzzServiceProvider(), array(
    'buzz.class_path' => __DIR__.'/vendor/buzz/lib',
));
```

Nota: Para las bibliotecas que no usan el espacio de nombres de *PHP* 5.3 pueden utilizar `RegisterPrefix` en lugar de `registerNamespace`, el cual utilizará un guión como delimitador de directorio.

Proveedores de controladores

15.1 Cargando proveedores

Con el fin de cargar y usar un controlador del proveedor, debes "montar" tus controladores en una ruta:

```
$app = new Silex\Application();

$app->mount('/blog', new Acme\BlogControllerProvider());
```

Todos los controladores definidos por el proveedor ahora estarán disponibles bajo la ruta */blog*.

15.2 Creando un proveedor

Los proveedores deben implementar la `Silex\ControllerProviderInterface`:

```
interface ControllerProviderInterface
{
    function connect(Application $app);
}
```

He aquí un ejemplo de tal proveedor:

```
namespace Acme;

use Silex\Application;
use Silex\ControllerProviderInterface;
use Silex\ControllerCollection;

class HelloControllerProvider implements ControllerProviderInterface
{
    public function connect(Application $app)
    {
        $controllers = new ControllerCollection();

        $controllers->get('/', function (Application $app) {
            return $app->redirect('/hello');
        });
```

```
        return $controllers;
    }
}
```

El método `connect` debe regresar una instancia de `ControllerCollection`. `ControllerCollection` es la clase donde todos los métodos controladores relacionados están definidos (como `get`, `post`, `match`, ...).

Truco: La clase `Application` de hecho actúa en un delegado para estos métodos.

Ahora puedes utilizar este proveedor de la siguiente manera:

```
$app = new Silex\Application();

$app->mount('/blog', new Acme\HelloControllerProvider());
```

En este ejemplo, la ruta `/blog/` ahora hace referencia al controlador definido en el proveedor.

Truco: También puedes definir un proveedor que implemente ambos el servicio y la interfaz del proveedor del controlador y envasar en la misma clase los servicios necesarios para hacer que tu controlador trabaje.

Parte V

Probando

Debido a que *Silex* está construido en la cima de *Symfony2*, es muy fácil escribir pruebas funcionales para tu aplicación. Las pruebas funcionales son pruebas automatizadas de software que garantizan que el código funciona correctamente. Estas van a través de la interfaz de usuario, utilizando un navegador simulado, e imitan las acciones que un usuario podría llevar a cabo.

CAPÍTULO 16

¿Por qué?

Si no estás familiarizado con las pruebas de software, puedes preguntarte por qué tendrías que necesitarlas. Cada vez que haces un cambio a tu aplicación, tienes que probarlo. Esto significa recorrer todas las páginas y asegurarte de que todavía están trabajando. Las pruebas funcionales te ahorran un montón de tiempo, ya que te permiten probar la aplicación en general, en menos de un segundo ejecutando una única orden.

Para más información sobre las pruebas funcionales, pruebas unitarias y pruebas automatizadas de software en general, consulta PHPUnit y Bulat Shakirzyanov habla en código limpio.

PHPUnit

PHPUnit Es de facto la plataforma de pruebas estándar para *PHP*. Fue construido para escribir pruebas unitarias, pero también lo puedes utilizar para pruebas funcionales. Escribes tus pruebas creando una nueva clase, que extienda a `PHPUnit_Framework_TestCase`. Tus casos de prueba son los métodos prefijados con `test`:

```
class ContactFormTest extends PHPUnit_Framework_TestCase
{
    public function testInitialPage()
    {
        ...
    }
}
```

En tus casos de prueba, haces afirmaciones sobre el estado de lo que estás probando. En este caso estamos probando un formulario de contacto, por lo tanto se quiere acertar que la página se ha cargado correctamente y contiene nuestro formulario:

```
public function testInitialPage()
{
    $statusCode = ...
    $pageContent = ...

    $this->assertEquals(200, $statusCode);
    $this->assertContains('Contact us', $pageContent);
    $this->assertContains('<form', $pageContent);
}
```

Aquí puedes ver algunas de las aserciones disponibles. Hay una lista completa en la sección Escribiendo pruebas para PHPUnit de la documentación de *PHPUnit*.

CAPÍTULO 18

WebTestCase

Symfony2 proporciona una clase `WebTestCase` que puedes utilizar para escribir pruebas funcionales. La versión `Silex` de esta clase es `Silex\WebTestCase`, y la puedes utilizar haciendo que tu prueba la extienda:

```
use Silex\WebTestCase;

class FormularioDeContactoTest extends WebTestCase
{
    ...
}
```

Nota: Para hacer comprobable tu aplicación, es necesario asegurarte de que sigue las instrucciones de "reutilización de aplicaciones" de *Usándola* (Página 7).

Para tu `WebTestCase`, tendrás que implementar un método `createApplication`, el cual devuelve tu aplicación. Este, probablemente, se verá así:

```
public function createApplication()
{
    return require __DIR__.'/ruta/a/app.php';
}
```

Asegúrate de **no** usar `require_once` aquí, ya que este método se ejecutará antes de cada prueba.

Truco: De manera predeterminada, la aplicación se comporta de la misma manera que cuando se utiliza desde un navegador. Pero cuando se produce un error, a veces es más fácil obtener excepciones en lugar de páginas *HTML*. Es bastante simple si ajustas la configuración de la aplicación en el método `createApplication()` como sigue:

```
public function createApplication()
{
    $app = require __DIR__.'/ruta/a/app.php';
    $app['debug'] = true;
    unset($app['exception_handler']);

    return $app;
}
```

Truco: Si tu aplicación usa sesiones, tienes que usar `FilesystemSessionStorage` para guardar las sesiones:

```
// ...
use Symfony\Component\HttpFoundation\SessionStorage\FilesystemSessionStorage;
// ...

public function createApplication()
{
    // ...
    $this->app['session.storage'] = $this->app->share(function() {
        return new FilesystemSessionStorage(sys_get_temp_dir());
    });
    // ...
}
```

El `WebTestCase` proporciona un método `createClient`. Un cliente actúa como un navegador, y te permite interactuar con tu aplicación. Así es como funciona:

```
public function testInitialPage()
{
    $client = $this->createClient();
    $crawler = $client->request('GET', '/');

    $this->assertTrue($client->getResponse()->isOk());
    $this->assertEquals(1, count($crawler->filter('h1:contains("Contact us")')));
    $this->assertEquals(1, count($crawler->filter('form')));
    ...
}
```

Aquí suceden varias cosas. Tienes tanto un `Cliente` como un `Rastreador`.

También puedes acceder a la aplicación a través de `$this->app`.

CAPÍTULO 19

Cliente

El cliente representa un navegador. Este mantiene tu historial de navegación, `cookies` y mucho más. El método `request` te permite hacer una petición a una página en tu aplicación.

Nota: Puedes encontrar alguna documentación para esto en la sección cliente del capítulo de pruebas de la documentación de Symfony2.

CAPÍTULO 20

Rastreador

El `rastreador` te permite inspeccionar el contenido de una página. Lo puedes filtrar usando expresiones *CSS* y mucho más.

Nota: Puedes encontrar alguna documentación para este en la sección rastreador del capítulo de pruebas de la documentación de Symfony2.

Configurando

La forma sugerida para configurar *PHPUnit* es crear un archivo `phpunit.xml.dist`, un directorio `tests` y tus pruebas en `tests/TuAp/Tests/TuPruebaTest.php`. El archivo `phpunit.xml.dist` debe tener este aspecto:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
>
    <testsuites>
        <testsuite name="YourApp Test Suite">
            <directory>./tests/</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

También puedes configurar un archivo de arranque para cargar tus clases y listas blancas automáticamente para los informes de cobertura de código.

Tu `tests/YourApp/Tests/YourTest.php` debería tener este aspecto:

```
namespace YourApp\Tests;

use Silex\WebTestCase;

class YourTest extends WebTestCase
{
    public function createApplication()
    {
        return require __DIR__.'/../../../app.php';
    }

    public function testFooBar()
    {
```

```
            ...
    }
}
```

Ahora, cuando ejecutes `phpunit` en la línea de ordenes, se deben ejecutar tus pruebas.

Parte VI

Funcionamiento interno

Este capítulo te dirá un poco sobre cómo funciona *Silex* internamente.

CAPÍTULO 22

Silex

22.1 Aplicación

La aplicación es la interfaz principal de *Silex*. Implementa la HttpKernelInterface de *Symfony2*, por lo tanto puedes pasar una Petición al método `handle` y devolverá una Respuesta.

Esta extiende el contenedor de servicios *Pimple*, lo cual permite flexibilidad tanto en el exterior cómo en el interior. puedes sustituir cualquier servicio, y también serás capaz de leerlo.

La aplicación usa exhaustivamente el EventDispatcher para engancharse a los eventos `HttpKernel` de *Symfony2*. Esto te permite recuperar la `Petición`, convertir las cadenas de respuestas en objetos `Respuesta` y manipular excepciones. También lo usamos para despachar algunos eventos personalizados como los filtros `before/after` y errores.

22.2 Controlador

El Enrutador de *Symfony2* en realidad es bastante potente. Puedes nombrar rutas, lo cual te permite generar *URL*. También puedes tener requisitos para las partes variables. A fin de permitirte una interfaz agradable a través de estos ajustes en el método `match` (que es utilizado por `get`, `post`, etc.) devolviendo una instancia del `Controller`, que envuelve una ruta.

22.3 `ControllerCollection`

Uno de los objetivos de exponer el RouteCollection era hacerlo mutable, para que los proveedores le puedan agregar cosas. Aquí, el reto es el hecho de que las rutas no saben nada acerca de su nombre. El nombre sólo tiene sentido en el contexto de la `RouteCollection` y no se puede cambiar.

Para resolver este desafío, se nos ocurrió tener una área de estacionamiento para las rutas. La `ControllerCollection` tiene los controladores hasta que se llama a `flush`, momento en el cual las rutas se añaden a la `RouteCollection`. Además, los controladores se congelan. Esto significa que ya no se pueden modificar y se produce una excepción si se intenta hacerlo.

Desafortunadamente no se ha podido encontrar una buena manera para lavar implícitamente, por lo que el lavado ahora siempre es explícito. La aplicación se debe vaciar, pero si quieres leer la `ControllerCollection` antes de que se lleve a cabo la petición, tendrás que llamar a `flush` tú mismo.

La `Appication` proporciona un atajo al método `flush` para el lavado del `ControllerCollection`.

Symfony2

Silex utiliza los siguientes componentes de *Symfony2*:

- **ClassLoader**: Para cargar clases automáticamente.
- **HttpFoundation**: Para `Petición` y `Respuesta`.
- **HttpKernel**: Porque necesitamos un corazón.
- **Routing**: Para concordancia de las rutas definidas.
- **EventDispatcher**: Para conectar al `HttpKernel`.

Para más información, visita el sitio web de Symfony.

Parte VII

Colaborando

Estamos abiertos a las aportaciones al código de *Silex*. Si encuentras un error o quieres contribuir con un proveedor, sólo tienes que seguir estos pasos.

- Bifurca el repositorio de Silex en *github*.
- Agrega tu característica o corrección de fallo.
- Añade las pruebas para ello. Esto es importante para que no se rompa involuntariamente en una futura versión.
- Envía una solicitud de atracción. Explicando el propósito para el tema de la rama.

Si tienes un gran cambio o te gustaría hablar de algo, por favor, únete a nuestra lista de correo http://groups.google.com/group/silex-php.

Nota: Cualquier aportación de código se debe autorizar bajo la Licencia *MIT*.

Parte VIII

Silex

DoctrineServiceProvider

El `DoctrineServiceProvider` proporciona integración con *Doctrine* DBAL para acceder fácilmente a la base de datos.

Nota: Sólo hay una `DBAL` de *Doctrine*. **No** se suministra un servicio `ORM`.

24.1 Parámetros

- **db.options**: Arreglo de opciones para `DBAL` de *Doctrine*.

  Estas opciones están disponibles:

  - **driver**: El controlador de la base de datos a utilizar, por omisión es `pdo_mysql`. Puede ser alguno de entre: `pdo_mysql`, `pdo_sqlite`, `pdo_pgsql`, `pdo_oci`, `oci8`, `ibm_db2`, `pdo_ibm`, `pdo_sqlsrv`.
  - **dbname**: El nombre de la base de datos a la cual conectarse.
  - **host**: El servidor de la base de datos a la cual conectarse. Por omisión es `localhost`.
  - **user**: El usuario de la base de datos con el cual conectarse. Por omisión es `root`.
  - **password**: La contraseña de la base de datos con la cual conectarse.
  - **path**: Sólo es relevante para `pdo_sqlite`, especifica la ruta a la base de datos `SQLite`.

  Estas y otras opciones se describen en detalle en la documentación de configurando el DBAL de Doctrine.
- **db.dbal.class_path** (opcional): Ruta a dónde se encuentra el `DBAL` de *Doctrine*.
- **db.common.class_path** (opcional): Ruta a dónde se encuentra *Doctrine Common*.

24.2 Servicios

- **db**: La conexión de base de datos, instancia de `Doctrine\DBAL\Connection`.
- **db.config**: Objeto de configuración para *Doctrine*. El valor predeterminado es una `Doctrine\DBAL\Configuration` vacía.

- **db.event_manager**: Gestor de eventos para *Doctrine*.

24.3 Registrando

Asegúrate de colocar una copia del `DBAL` de *Doctrine* en `vendor/doctrine-dbal` y *Doctrine Common* en `vendor/doctrine-common`:

```
$app->register(new Silex\Provider\DoctrineServiceProvider(), array(
    'db.options'            => array(
        'driver'    => 'pdo_sqlite',
        'path'      => __DIR__.'/app.db',
    ),
    'db.dbal.class_path'    => __DIR__.'/vendor/doctrine-dbal/lib',
    'db.common.class_path'  => __DIR__.'/vendor/doctrine-common/lib',
));
```

24.4 Uso

El proveedor *Doctrine* proporciona un servicio `db`. Aquí está un ejemplo de uso:

```
$app->get('/blog/show/{id}', function ($id) use ($app) {
    $sql = "SELECT * FROM posts WHERE id = ?";
    $post = $app['db']->fetchAssoc($sql, array((int) $id));

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

24.5 Utilizando múltiples bases de datos

El proveedor *Doctrine* te permite acceder a múltiples bases de datos. Para configurar tus fuentes de datos, sustituye `db.options` con `dbs.options`. `dbs.options` es una matriz de configuraciones donde las claves son los nombres de las conexiones y los valores son las opciones:

```
$app->register(new Silex\Provider\DoctrineServiceProvider(), array(
    'dbs.options' => array (
        'mysql_read' => array(
            'driver'    => 'pdo_mysql',
            'host'      => 'mysql_read.someplace.tld'
            'dbname'    => 'mi_base_de_datos',
            'user'      => 'mi_nombredeusuario',
            'password'  => 'mi_pase',
        ),
        'mysql_write' => array(
            'driver'    => 'pdo_mysql',
            'host'      => 'mysql_write.someplace.tld'
            'dbname'    => 'mi_base_de_datos',
            'user'      => 'mi_nombredeusuario',
            'password'  => 'mi_pase',
        ),
    ),
    'db.dbal.class_path'    => __DIR__.'/vendor/doctrine-dbal/lib',
```

```
    'db.common.class_path'  => __DIR__.'/vendor/doctrine-common/lib',
));
```

La primer conexión registrada es la predeterminada y puedes acceder a ella como lo harías si hubiera una sola conexión. Teniendo en cuenta la configuración anterior, estas dos líneas son equivalentes:

```
$app['db']->fetchAssoc('SELECT * FROM table');

$app['dbs']['mysql_read']->fetchAssoc('SELECT * FROM table');
```

Usando múltiples conexiones:

```
$app->get('/blog/show/{id}', function ($id) use ($app) {
    $sql = "SELECT * FROM posts WHERE id = ?";
    $post = $app['dbs']['mysql_read']->fetchAssoc($sql, array((int) $id));

    $sql = "UPDATE posts SET value = ? WHERE id = ?";
    $app['dbs']['mysql_write']->execute($sql, array('newValue', (int) $id));

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

Para más información, consulta la Documentación DBAL de Doctrine.

CAPÍTULO 25

MonologServiceProvider

El proveedor `MonologServiceProvider` proporciona un mecanismo de registro predeterminado a través de la biblioteca Monolog de Jordi Boggiano's.

Esta registrará las peticiones y errores y te permite añadir a tu aplicación el registro de depuración, para que no tengas que usar `var_dump` mucho más. Puedes utilizar la versión madura llamada `tail-f`.

25.1 Parámetros

- **monolog.logfile**: Archivo donde escribir los registros.
- **monolog.class_path** (opcional): Ruta a la biblioteca donde se encuentra *Monolog*.
- **monolog.level** (opcional): El nivel de registro por omisión es `DEBUG`. Debe ser uno de `Logger::DEBUG`, `Logger::INFO`, `Logger::WARNING`, `Logger::ERROR`. `DEBUG` registra todo, `INFO` registrará todo excepto `DEBUG`, etc.
- **monolog.name** (opcional): Nombre del canal de *Monplog*, por omisión es `myapp`.

25.2 Servicios

- **monolog**: La instancia del notario de *Monolog*.

  Ejemplo de uso:

  ```
  $app['monolog']->addDebug('Probando el notario de Monolog.');
  ```

- **monolog.configure**: Cierre protegido que toma al notario como argumento. Lo puedes modificar si no deseas el comportamiento por omisión.

25.3 Registrando

Asegúrate de colocar una copia de *Monolog* en el directorio `vendor/monolog`:

```
$app->register(new Silex\Provider\MonologServiceProvider(), array(
    'monolog.logfile'       => __DIR__.'/development.log',
    'monolog.class_path'    => __DIR__.'/vendor/monolog/src',
));
```

Nota: *Monolog* no está compilado en el archivo `silex.phar`. Tienes que añadir tu propia copia de *Monolog* a tu aplicación.

25.4 Uso

El proveedor `MonologServiceProvider` ofrece un servicio `monolog`. Lo puedes utilizar para agregar entradas al registro para cualquier nivel de registro a través de `addDebug()`, `addInfo()`, `addWarning()` y `addError()`.

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

$app->post('/user', function () use ($app) {
    // ...

    $app['monolog']->addInfo(sprintf("User '%s' registered.", $username));

    return new Response('', 201);
});
```

Para más información, consulta la documentación de Monolog.

SessionServiceProvider

El proveedor `SessionServiceProvider` ofrece un servicio para almacenar datos persistentes entre peticiones.

26.1 Parámetros

- **session.default_locale**: La región usada por omisión en la sesión.
- **session.storage.options**: Un arreglo de opciones que se pasa al constructor del servicio `session.storage`.

  En caso del predeterminado `NativeSessionStorage`, las opciones posibles son:

  - **name**: El nombre de la `cookie` (por omisión `_SESS`)
  - **id**: El `id` de la sesión (por omisión `null`)
  - **lifetime**: Tiempo de vida de la `cookie`
  - **path**: Ruta a la `cookie`
  - **domain**: Dominio de la `cookie`
  - **secure**: `cookie` segura (*HTTPS*)
  - **httponly**: Cuando la `cookie` únicamente es *http*

  Sin embargo, todas estas son opcionales. Las sesiones duran por siempre, mientras el navegador permanezca abierto. Para evitar esto, establece la opción `lifetime`.

26.2 Servicios

- **session**: Una instancia de la Session de *Symfony2*.
- **session.storage**: Un servicio que se utiliza para persistir los datos de sesión. Por omisión es NativeSessionStorage.

26.3 Registrando

```
$app->register(new Silex\Provider\SessionServiceProvider());
```

26.4 Uso

El proveedor `Session` proporciona un servicio `session`. He aquí un ejemplo que autentica a un usuario y crea una sesión para él:

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

$app->get('/login', function () use ($app) {
    $username = $app['request']->server->get('PHP_AUTH_USER', false);
    $password = $app['request']->server->get('PHP_AUTH_PW');

    if ('igor' === $username && 'password' === $password) {
        $app['session']->set('user', array('username' => $username));
        return $app->redirect('/account');
    }

    $response = new Response();
    $response->headers->set('WWW-Authenticate', sprintf('Basic realm="%s"', 'site_login'));
    $response->setStatusCode(401, 'Please sign in.');
    return $response;
});

$app->get('/account', function () use ($app) {
    if (null === $user = $app['session']->get('user')) {
        return $app->redirect('/login');
    }

    return "Welcome {$user['username']}!";
});
```

SwiftmailerServiceProvider

El proveedor `SwiftmailerServiceProvider` ofrece un servicio para enviar correo electrónico a través de la biblioteca de correo Swift Mailer.

Puedes utilizar el servicio `mailer` (cliente de correo) para enviar mensajes fácilmente. Por omisión, este tratará de enviar el correo electrónico a través de *SMTP*.

27.1 Parámetros

- **swiftmailer.options**: Una matriz de opciones para la configuración predeterminada basada en *SMTP*.

  Puedes ajustar las siguientes opciones:

  - **host**: nombre del servidor *SMTP*, predeterminado a `'localhost'`.
  - **port**: puerto *SMTP*, el predeterminado es 25.
  - **username**: nombre del usuario *SMTP*, por omisión es una cadena vacía.
  - **password**: contraseña *SMTP*, de manera predeterminada es una cadena vacía.
  - **encryption**: cifrado *SMTP*, predeterminado a `null`.
  - **auth_mode**: modo de autenticación *SMTP*, predeterminado a `null`.
- **swiftmailer.class_path** (opcional): Ruta a donde está ubicada la librería `Swift Mailer`.

27.2 Servicios

- **mailer**: La instancia del cliente de correo.

  Ejemplo de uso:

  ```
  $message = \Swift_Message::newInstance();

  // ...

  $app['mailer']->send($message);
  ```

- **swiftmailer.transport**: El transporte usado para entregar el correo electrónico. Predeterminado a `Swift_Transport_EsmtpTransport`.
- **swiftmailer.transport.buffer**: El `StreamBuffer` usado por el transporte.
- **swiftmailer.transport.authhandler**: Controlador de autenticación usado por el transporte. De manera predeterminada intentará con los siguientes: CRAM-MD5, login, plaintext.
- **swiftmailer.transport.eventdispatcher**: Despachador de eventos interno usado por `Swiftmailer`.

27.3 Registrando

Asegúrate de colocar una copia de *Swift Mailer* en el directorio `vendor/SwiftMailer`. Asegúrate de apuntar la ruta de tu clase a `/lib/classes`.

```
$app->register(new Silex\Provider\SwiftmailerServiceProvider(), array(
    'swiftmailer.class_path'  => __DIR__.'/vendor/swiftmailer/lib/classes',
));
```

Nota: `Swift Mailer` no está compilado en el archivo `silex.phar`. Tienes que añadir tu propia copia de `Swift Mailer` para tu aplicación.

27.4 Uso

El proveedor `Swiftmailer` proporciona un servicio `mailer`.

```
$app->post('/feedback', function () use ($app) {
    $request = $app['request'];

    $message = \Swift_Message::newInstance()
        ->setSubject('[YourSite] Feedback')
        ->setFrom(array('noreply@yoursite.com'))
        ->setTo(array('feedback@yoursite.com'))
        ->setBody($request->get('message'));

    $app['mailer']->send($message);

    return new Response('Thank you for your feedback!', 201);
});
```

Para más información, consulta la Documentación de Swift Mailer.

CAPÍTULO 28

SymfonyBridgesServiceProvider

El *SymfonyBridgesServiceProvider* proporciona integración adicional entre componentes y bibliotecas de *Symfony2*.

28.1 Parámetros

- **symfony_bridges.class_path** (opcional): Ruta a la ubicación donde están localizados los puentes de *Symfony2*.

28.2 *Twig*

Cuando está activado el `SymfonyBridgesServiceProvider`, el `TwigServiceProvider` te proporcionará capacidades adicionales:

- **UrlGeneratorServiceProvider**: Si estás usando el `UrlGeneratorServiceProvider`, recibirás los ayudantes `path` y `url` para *Twig*. Puedes encontrar más información en la documentación de enrutado de Symfony2.
- **TranslationServiceProvider**: Si estás usando el `TranslationServiceProvider`, recibirás los ayudantes `trans` y `transchoice` para traducir en las plantillas `Twig`. Puedes encontrar más información en la documentación de traducción de Symfony2.
- **FormServiceProvider**: Si estás usando el `FormServiceProvider`, recibirás un conjunto de ayudantes para trabajar con formularios en plantillas. Puedes encontrar más información en la referencia de formularios de Symfony2.

28.3 Registrando

Asegúrate de colocar una copia del puente *Symfony2* en `vendor/symfony/src` bien clonando Symfony2 o `vendor/symfony/src/Symfony/Bridge/Twig` clonando TwigBridge (el último consume menos recursos).

Luego, registra el proveedor vía:

```
$app->register(new Silex\Provider\SymfonyBridgesServiceProvider(), array(
    'symfony_bridges.class_path'  => __DIR__.'/vendor/symfony/src',
));
```

TranslationServiceProvider

El *TranslationServiceProvider* provee un servicio para traducir tu aplicación a diferentes idiomas.

29.1 Parámetros

- **translator.messages**: Una asignación de regiones para los arreglos de mensajes. Este parámetro contiene los datos de traducción en todos los idiomas.
- **locale** (opcional): La región para el traductor. Lo más probable es que desees establecer este basándote en algún parámetro de la petición. Predeterminado a `en`.
- **locale_fallback** (opcional): La región de reserva para el traductor. Esta se utiliza cuando la configuración regional actual no tiene ningún conjunto de mensajes.
- **translation.class_path** (opcional): Ruta a donde se encuentra el componente `Translation` de *Symfony2*.

29.2 Servicios

- **translator**: Una instancia de Translator, utilizada para traducir.
- **translator.loader**: Una instancia de una implementación de la traducción LoaderInterface, predeterminada a un ArrayLoader.
- **translator.message_selector**: Una instancia de MessageSelector.

29.3 Registrando

Asegúrate de colocar una copia del componente de traducción de *Symfony2* en `vendor/symfony/src`. Puedes simplemente clonar todo *Symfony2* en `vendor`:

```
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'locale_fallback'           => 'en',
    'translation.class_path'    => __DIR__.'/vendor/symfony/src',
));
```

29.4 Uso

El proveedor `Translation` ofrece un servicio `traductor` y usa el parámetro `translator.messages`:

```
$app['translator.messages'] = array(
    'en' => array(
        'hello'     => 'Hello %name%',
        'goodbye'   => 'Goodbye %name%',
    ),
    'de' => array(
        'hello'     => 'Hallo %name%',
        'goodbye'   => 'Tschüss %name%',
    ),
    'fr' => array(
        'hello'     => 'Bonjour %name%',
        'goodbye'   => 'Au revoir %name%',
    ),
);

$app->before(function () use ($app) {
    if ($locale = $app['request']->get('locale')) {
        $app['locale'] = $locale;
    }
});

$app->get('/{locale}/{message}/{name}', function ($message, $name) use ($app) {
    return $app['translator']->trans($message, array('%name%' => $name));
});
```

El ejemplo anterior se traducirá en las siguientes rutas:

- `/en/hello/igor` regresará `Hello igor`.
- `/de/hello/igor` regresará `Hallo igor`.
- `/fr/hello/igor` regresará `Bonjour igor`.
- `/it/hello/igor` regresará `Hello igor` (debido a la reserva).

29.5 Recetas

29.5.1 Archivos de idioma basados en *YAML*

Tener tu traducción en archivos *PHP* puede ser un inconveniente. Esta receta te muestra cómo cargar traducciones de archivos *YAML* externos.

En primer lugar necesitas los componentes `Config` y `Yaml` de *Symfony2*. Además, asegúrate de registrarlos en el cargador automático. Puedes clonar el repositorio de *Symfony2* completo en `vendor/symfony`:

```
$app['autoloader']->registerNamespace('Symfony', __DIR__.'/vendor/symfony/src');
```

A continuación, debes crear las asignaciones de idioma en los archivos *YAML*. Un nombre que puedes utilizar es `locales/en.yml`. Sólo haz la asignación en este archivo de la siguiente manera:

```
hello: Hello %name%
goodbye: Goodbye %name%
```

Repite esto para todos tus idiomas. A continuación, configura el `translator.messages` para asignar archivos a los idiomas:

```
$app['translator.messages'] = array(
    'en' => __DIR__.'/locales/en.yml',
    'de' => __DIR__.'/locales/de.yml',
    'fr' => __DIR__.'/locales/fr.yml',
);
```

Finalmente sobrescribe el `translator.loader` para utilizar `YamlFileLoader` en lugar del `ArrayLoader` predeterminado:

```
$app['translator.loader'] = new Symfony\Component\Translation\Loader\YamlFileLoader();
```

Y eso es todo lo que necesitas para cargar traducciones desde archivos *YAML*.

TwigServiceProvider

El `TwigServiceProvider` proporciona integración con el motor de plantillas Twig.

30.1 Parámetros

- **twig.path** (opcional): Ruta al directorio que contiene archivos de plantilla *Twig* (también puede ser un arreglo de rutas).
- **twig.templates** (opcional): Se trata de una matriz asociativa de nombres de plantilla para el contenido de la plantilla. Usa esta opción si deseas definir tus plantillas en línea.
- **twig.options** (opcional): Una matriz asociativa de opciones *Twig*. Echa un vistazo a la documentación de *Twig* para más información.
- **twig.class_path** (opcional): Ruta a dónde se encuentra la biblioteca *Twig*.
- **twig.form.templates** (opcional): Una matriz de plantillas utilizada para reproducir formularios (disponible únicamente cuando está habilitado el `FormServiceProvider`).

30.2 Servicios

- **twig**: La instancia de `Twig_Environment`. La principal forma de interactuar con *Twig*.
- **twig.configure**: Cierre protegido que toma el entorno *Twig* como argumento. Lo puedes utilizar para agregar más globales personalizadas.
- **twig.loader**: El cargador de plantillas *Twig* que utilizan las opciones `twig.path` y `twig.templates`. También puedes reemplazar el cargador completamente.

30.3 Registrando

Asegúrate de colocar una copia de *Twig* en el directorio `vendor/twig`:

```
$app->register(new Silex\Provider\TwigServiceProvider(), array(
    'twig.path'       => __DIR__.'/views',
    'twig.class_path' => __DIR__.'/vendor/twig/lib',
));
```

Nota: *Twig* no está compilado en el archivo `silex.phar`. Tienes que añadir tu propia copia de *Twig* a tu aplicación.

30.4 Uso

El proveedor *Twig* ofrece un servicio `twig`:

```
$app->get('/hello/{name}', function ($name) use ($app) {
    return $app['twig']->render('hello.twig', array(
        'name' => $name,
    ));
});
```

Esto reproducirá un archivo llamado `views/hola.twig`.

En cualquier plantilla de *Twig*, la variable `app` se refiere al objeto `Aplicación`. Para que puedas acceder a cualquier servicio desde tu vista. Por ejemplo, para acceder a `$app['request']->getHost()`, sólo tienes que poner esto en tu plantilla:

```
{{ app.request.host }}
```

También se registra una función `render` para ayudarte a reproducir otro controlador desde una plantilla:

```
{{ render('/sidebar') }}

{# o si también estás usando UrlGeneratorServiceProvider #}
{{ render(path('sidebar')) }}
```

Para más información, consulta la documentación de Twig.

UrlGeneratorServiceProvider

El `UrlGeneratorServiceProvider` ofrece un servicio para generar *URL* para las rutas con nombre.

31.1 Parámetros

Ninguno.

31.2 Servicios

- **url_generator**: Una instancia del UrlGenerator, usando la RouteCollection proporcionada a través del servicio `routes`. Tiene un método `generate`, el cual toma el nombre de la ruta como argumento, seguido por un arreglo de parámetros de la ruta.

31.3 Registrando

```
$app->register(new Silex\Provider\UrlGeneratorServiceProvider());
```

31.4 Uso

El proveedor `UrlGenerator` ofrece un servicio `url_generator`:

```
$app->get('/', function () {
    return 'welcome to the homepage';
})
->bind('homepage');

$app->get('/hello/{name}', function ($name) {
    return "Hello $name!";
})
->bind('hello');
```

```
$app->get('/navigation', function () use ($app) {
    return '<a href="'.$app['url_generator']->generate('homepage').'">Home</a>'.
           ' | '.
           '<a href="'.$app['url_generator']->generate('hello', array('name' => 'Igor')).'">Hello Igo
});
```

ValidatorServiceProvider

El `ValidatorServiceProvider` ofrece un servicio de validación de datos. Es más útil cuando lo utilizas con `FormServiceProvider`, pero también se puede utilizar de manera independiente.

32.1 Parámetros

- **validator.class_path** (opcional): Ruta a donde se encuentra el componente `Validator` de *Symfony2*.

32.2 Servicios

- **validator**: Una instancia del Validator.
- **validator.mapping.class_metadata_factory**: Fábrica de cargadores de metadatos, que pueden leer la información de validación desde la restricción de las clases. El valor predeterminado es `StaticMethodLoader--ClassMetadataFactory`.

  Esto significa que puedes definir un método estático `loadValidatorMetadata` en tu clase de datos, que tenga un argumento `ClassMetadata`. Entonces puedes establecer restricciones en esta instancia de `ClassMetadata`.
- **validator.validator_factory**: Fábrica de `ConstraintValidators`. De manera predeterminada a un `ConstraintValidatorFactory` estándar. Generalmente lo utiliza internamente el validador.

32.3 Registrando

Asegúrate de colocar una copia del componente `Validator` de *Symfony2* en `vendor/symfony/src`. Puedes simplemente clonar todo *Symfony2* en `vendor`:

```
$app->register(new Silex\Provider\ValidatorServiceProvider(), array(
    'validator.class_path'    => __DIR__.'/vendor/symfony/src',
));
```

32.4 Uso

El proveedor `Validator` proporciona un servicio `validator`.

32.4.1 Validando valores

Puedes validar directamente los valores usando el método de validación `validateValue`:

```
use Symfony\Component\Validator\Constraints;

$app->get('/validate-url', function () use ($app) {
    $violations = $app['validator']->validateValue($app['request']->get('url'), new Constraints\Url()
    return $violations;
});
```

Esto está limitado relativamente.

32.4.2 Validando propiedades de objeto

Si deseas añadir validaciones a una clase, puedes implementar un método estático `loadValidatorMetadata` como se describe en *Servicios*. Esto te permite definir las restricciones para las propiedades de tu objeto. También trabaja con captadores:

```
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints;

class Post
{
    public $title;
    public $body;

    static public function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('title', new Constraints\NotNull());
        $metadata->addPropertyConstraint('title', new Constraints\NotBlank());
        $metadata->addPropertyConstraint('body', new Constraints\MinLength(array('limit' => 10)));
    }
}

$app->post('/posts/new', function () use ($app) {
    $post = new Post();
    $post->title = $app['request']->get('title');
    $post->body = $app['request']->get('body');

    $violations = $app['validator']->validate($post);
    return $violations;
});
```

Tendrás que manipular la presentación de estas violaciones tú mismo. No obstante, puedes utilizar el `FormServiceProvider` el cual puede usar el `ValidatorServiceProvider`.

Para más información, consulta la documentación de validación de Symfony2.

HttpCacheServiceProvider

El proveedor `HttpCacheProvider` proporciona compatibilidad para el delegado inverso de *Symfony2*.

33.1 Parámetros

- **http_cache.cache_dir**: El directorio de caché para almacenar los datos de la caché *HTTP*.
- **http_cache.options** (opcional): Un arreglo de opciones para el constructor de HttpCache.

33.2 Servicios

- **http_cache**: Una instancia de HttpCache,

33.3 Registrando

```
$app->register(new Silex\Provider\HttpCacheServiceProvider(), array(
    'http_cache.cache_dir' => __DIR__.'/cache/',
));
```

33.4 Uso

Silex, fuera de la caja, ya es compatible con cualquier delegado inverso como *Varnish* ajustando las cabeceras de caché *HTTP* de la Respuesta:

```
$app->get('/', function() {
    return new Response('Foo', 200, array(
        'Cache-Control' => 's-maxage=5',
    ));
});
```

Este proveedor te permite utilizar el delegado inverso nativo de *Symfony2* con aplicaciones *Silex* usando el servicio `http_cache`:

```
$app['http_cache']->run();
```

El proveedor también proporciona apoyo *ESI*:

```
$app->get('/', function() {
    return new Response(<<<EOF
<html>
    <body>
        Hello
        <esi:include src="/included" />
    </body>
</html>

EOF
    , 200, array(
        'Cache-Control' => 's-maxage=20',
        'Surrogate-Control' => 'content="ESI/1.0"',
    ));
});

$app->get('/included', function() {
    return new Response('Foo', 200, array(
        'Cache-Control' => 's-maxage=5',
    ));
});

$app['http_cache']->run();
```

Para más información, consulta la documentación de la caché HTTP de Symfony2.

Parte IX

Registro de cambios

Este registro de cambios refiere todas las incompatibilidades con versiones anteriores conforme se presentaron:

- **2011-09-22**: `ExtensionInterface` se le cambió el nombre a `ServiceProviderInterface`. Todas las extensiones integradas se han renombrado consecuentemente (por ejemplo, `Silex\Extension\TwigExtension` ha cambiado el nombre de `Silex\Provider\TwigServiceProvider`)
- **2011-09-22**: La forma de trabajar de las aplicaciones reutilizables ha cambiado. El método `mount()` ahora toma una instancia de `ControllerCollection` en lugar de una `Application`.

  Antes:

  ```
  $app = new Application();
  $app->get('/bar', function() { return 'foo'; });

  return $app;
  ```

  Después:

  ```
  $app = new ControllerCollection();
  $app->get('/bar', function() { return 'foo'; });

  return $app;
  ```

- **2011-08-08**: La configuración del método controlador ahora se hace en el propio `Controller`

  Antes:

  ```
  $app->match('/', function () { echo 'foo'; }, 'GET|POST');
  ```

  Después:

  ```
  $app->match('/', function () { echo 'foo'; })->method('GET|POST');
  ```