## Cheat Sheet



**SILEX**PHP micro-framework

## Install

\$ composer require silex/silex:~1.2

## Usage

```
// web/index.php
require_once __DIR__.'/../vendor/autoload.php';
$app = new Silex\Application();
$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello ' . $app->escape($name);
});
$app->run();
```

## Web Server Configuration

## **Apache**

Options -MultiViews

RewriteEngine On

#RewriteBase /path/to/app

(IfModule mod rewrite.c>)

RewriteBase /path/to/app RewriteCond %{REQUEST\_FILENAME} !-f RewriteRule ^ index.php [QSA,L]

If your site is not at the webroot level: uncomment the RewriteBase statement and adjust the path to point to your directory, relative from the webroot.

For Apache 2.2.16+, you can use the FallbackResource directive:

FallbackResource /index.php

## nginx

configure your vhost to forward non-existent resources to index.php:

```
server {
   *site root is redirected to the app boot script
  location = / {
     try_files @site @site;
   # all other locations try other files first and go to our
  # front controller if none of them exists
  location / {
     try_files $uri $uri/ @site;
   *return 404 for all php files as we do have a front controller
  location ~ \.php$ {
     return 404;
   location @site {
     fastcgi_pass unix:/var/run/php-fpm/www.sock;
     include fastcgi_params;
     fastcgi param SCRIPT FILENAME $document root/index.php;
     #uncomment when running via https
     #fastcgi_param HTTPS on;
```

## Lighttpd

#### simple-vhost:

#### IIS

```
web.config file:
```

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

## PHP 5.4 built-in webserver



Allows run silex without any configuration. However, in order to serve static files, you'll have to make sure your front controller returns false in that case (web/index.php):

To start the server from the command-line:

\$ php -S localhost:8080 -t web web/index.php

The application should be running at http://localhost:8080.

## **App Configuration**

## **Default Configuration**

Configuration	Default value
<pre>\$app['debug'] \$app['request.http_port'] \$app['request.https_port'] \$app['charset'] \$app['locale'] \$app['logger']</pre>	false 80 443 'UTF-8' 'en' null

Set	Get
<pre>\$app['debug'] = true; // turn on the debug mode</pre>	<pre>\$debug = \$app['debug'];</pre>

## **Custom Configuration (Parameters)**

#### Set

```
$app['asset_path'] = 'http://assets.examples.com';
$app['base_url'] = '/';
```

#### Get

\$assetPath = \$app['asset\_path'];

## Using in a template (Twig)

{{ app.asset\_path }}/css/styles.css

If your application is hosted behind a reverse proxy at address \$ip, and you want Silex to trust the X-Forwarded-For\* headers, you will need to run your application like this:

use Symfony\Component\HttpFoundation\Request;

Request::setTrustedProxies(array(\$ip));
\$app->run();

## **Global Configuration**

To apply a controller setting to all controllers (a converter, a middleware, a requirement, or a default value), configure it on \$app['controllers']:

```
$app['controllers']
->value('id', '1')
->assert('id', '\d+')
->requireHttps()
->method('get')
->convert('id', function () {/* ... */ })
->before(function () {/* ... */ });
```

Global configuration does not apply to controller providers you might mount as they have their own global configuration

## **Symfony2 Components**

Symfony2 components used by Silex:

HttpFoundation For Request and Response.

HttpKernel Because we need a heart.

Routing For matching defined routes.

EventDispatcher For hooking into the HttpKernel

## **App Helper Methods**

#### **Redirect**

\$app->redirect('/account');

#### **Forward**

```
$subRequest = Request::create('/hi', 'GET');
$app->handle($subRequest, HttpKernelInterface::SUB_REQUEST);
```

Generate the URI using UrlGeneratorProvider:

\$r = Request::create(\$app['url\_generator']->generate('hi'), 'GET');

## **JSON**

Return JSON data and apply correct escape

```
$app->json($error, 404);
$app->json($user);
```

#### Stream

```
Streaming response (when you cannot buffer the data being sent)
$app->stream($stream, 200, array('Content-Type' => 'image/png'));
```

To send chunks, make sure to call ob\_flush and flush after every chunk:

```
$stream = function () {
    $fh = fopen('http://www.example.com/', 'rb');

while (!feof($fh)) {
    echo fread($fh, 1024);
    ob_flush();
    flush();
}

fclose($fh);
};
```

#### **Abort**

Stop the request early. It actually throws an exception

```
$app->abort(404, "Book $id does not exist.");
```

## Sending a file HttpFoundation 2.2+

Create a BinaryFileResponse

#### **Escape**

Escape user input, to prevent Cross-Site-Scripting attacks \$app->escape(\$name)

## **App Events**

```
EARLY_EVENT = 512;
LATE_EVENT = -512;
```

## Routing

});

A route pattern consists of:

Pattern Defines a path that points to a resource.
Can include variable parts and you are able to set RegExp requirements for them

Method One of the HTTP methods:
GET, POST, PUT or DELETE.
Describes the interaction with the resource

```
method     pattern

$app->get('/blog', function () use ($posts) {
     $output = ";

     foreach ($posts as $post) {
          $output .= $post['title'];
          $output .= '<br />';
     }

     return $output;
});
```

#### **HTTP Methods**

## HTTP method override (for PUT, DELETE, and PATCH)

Forms in most web browsers do not directly support the use of other HTTP methods.
Use a special form field named \_method.

The form's method attribute must be set to POST when using this field:

For Symfony Components 2.2+: explicitly enable method override use Symfony\Component\HttpFoundation\Request;

Request::enableHttpMethodParameterOverride();

\$app->run():

## **Dynamic Route (Route Variable)**

## **Default Values**

To define a default value for any route variable call value on the Controller object:

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

This will allow matching /, in which case the page variable will have the value index

## **Matching all Methods**

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

$app->match('/book', function () {
    // ...
})
->method('PATCH');

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

The order of the routes is significant.

The first matching route will be used (place more generic routes at the bottom)

## **Route Variables Converters**

Appling some converters before injecting route variables into the controller:

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

## Converting route variables to objects

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

The converter callback also receives the Reguest as its second argument:

```
$callback = function ($post, Request $req) {
  return new Post($req->attributes->get('slug'));
$app->get('/blog/{id}/{slug}', function (Post $post) {
})->convert('post', $callback);
```

#### Converter defined as a service

use Doctrine\Common\Persistence\ObjectManager;

E.g.: user converter based on Doctrine ObjectManager:

```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
class UserConverter
  private som:
  public function __construct(ObjectManager $om)
     $this->om = $om:
  public function convert($id)
     if (null === $user = $this->om->find('User', (int) $id)) {
       throw new NotFoundHttpException(
          sprintf('User %d does not exist', $id)
     return Suser:
  return new UserConverter();
```

The service will now be registered in the app, and the convert method will be used as converter:

```
$app['converter.user'] = $app->share(function () {
});
$app->get('/user/{user}', function (User $user) {
  // ...
                                                version 1.2
})->convert('user', 'converter.user:convert');
                                                (a:b notation)
```

#### mount() prefixes all routes with the given prefix **Group URLs (mount)** and merges them into the main Application

```
// define controllers for a blog
                                         // define controllers for a forum
$blog = $app['controllers_factory'];
                                         $forum = $app['controllers_factory'];
$blog->get('/', function() {
                                         $forum->get('/', function () {
  return 'Blog home page';
                                            return 'Forum home page';
```

\$app['controllers\_factory'] is a factory that returns a new instance of ControllerCollection.

When mounting a route collection under /blog, it is not possible to define a route for the /blog URL. The shortest possible URL is /blog/.

```
// define "global" controllers
// / map to the main home page
$app->get('/', function () {
  return 'Main home page';
});
// /blog/ map to the blog home page
$app->mount('/blog', $blog);
// /forum/ map to the forum home page
$app->mount('/forum', $forum);
```

## Requirements

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

### **Chained requirements**

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

#### **Named Routes**

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

It only makes sense to name routes if you use providers that make use of the RouteCollection

#### Controllers in Classes

```
$app->get('/', 'Acme\\Foo::bar');
use Silex\Application;
use Symfony\Component\HttpFoundation\Request:
namespace Acme
  class Foo
     public function bar(Request $request, Application $app)
       // ...
```

For an even stronger separation between Silex and your controllers, you can define your controllers as services

## **Error Handlers**

To register an error handler, pass a closure to the error method which takes an Exception argument and returns a response:

use Symfony\Component\HttpFoundation\Response:

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

Check for specific errors (using \$code):

use Symfony\Component\HttpFoundation\Response;

As Silex ensures that the Response status code is set to the most appropriate one depending on the exception, setting the status on the response won't work.

To overwrite the status code (which should not without a good reason), set the X-Status-Code header:

```
return new Response('Error', 404 /* ignored */,
array('X-Status-Code' => 200));
```

## **Restricting Error Handler**

To restrict an error handler to only handle some Exception classes set a more specific type hint for the Closure argument:

```
$app->error(function (\LogicException $e, $code) {
   // this handler will only handle \LogicException exceptions
   // and exceptions that extends \LogicException
});
```

## **Logging and Error Handler**

Use a separate error handler for logging. Make sure to register it before the response error handlers, because once a response is returned, the following handlers are ignored.

## **Debug and Error Handler**

Silex comes with a default error handler that displays a detailed error message with the stack trace when debug is true, and a simple error message otherwise.

Error handlers registered via error() method always take precedence. To keep the nice error messages when debug is on use this:

```
$app->error(function (\Exception $e, $code) use ($app) {
   if ($app['debug']) {
      return;
   }
   // ... logic to handle the error and return a Response
});
```

The error handlers are also called when you use abort to abort a request early:



Define shortcut methods.

Almost all built-in service providers have some corresponding PHP traits.

To use them, define your own Application class and include the traits you want:

```
use Silex\Application;

class MyApplication extends Application {
    use Application\TwigTrait;
    use Application\SecurityTrait;
    use Application\FormTrait;
    use Application\UrlGeneratorTrait;
    use Application\SwiftmailerTrait;
    use Application\MonologTrait;
    use Application\TranslationTrait;
}
```

To define your own Route class and use some traits:

```
use Silex\Route;
class MyRoute extends Route
{
   use Route\SecurityTrait;
}
```

To use the newly defined route, override the \$app['route\_class'] setting:

```
$app['route_class'] = 'MyRoute';
```

Silex documentation, examples, and download: http://silex.sensiolabs.org/

## **Middlewares**

## Application Middlewares

Only run for "master" request

#### **Before**

Event registered on the Symfony request event.

Run after the routing and security and before the controller.

```
$app->before(function (Request $req, Application $app) {
    // ...
}):
```

To run the middleware even if an exception is thrown early on (on a 404 or 403 error for instance), register it as an early event:

```
$app->before(function (Request $req, Application $app) {
    // ...
}, Application::EARLY_EVENT);
```

#### After

Event registered on the Symfony response event.

Run before the Response is sent to the client.

```
$app->after(function (Request $req, Response $resp) {
    // ...
});
```

#### **Terminate**

Event registered on the Symfony terminate event.

Run after the Response has been sent to the client (like sending emails or logging).

```
$app->finish(function (Request $req, Response $resp) {
    // ...
    // Warning: modifications to the Request or Response
    // will be ignored
}):
```

#### **Route Middlewares**

Added to routes or route collections.

Only triggered when the corresponding route is matched.
You can also stack them:

```
$app->get('/somewhere', function () {
    // ...
})
->before($before1)
->before($before2)
->after($after1)
->after($after2)
;
```

#### **Before**

Fired before the route callback, but after the before application middlewares:

```
$before = function (Request $req, Application $app) {
    // ...
};

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

#### **After**

Fired after the route callback, but before the application after application middlewares:

## **Middlewares Priority**

The middlewares are triggered in the same order they are added.

To control the priority of the middleware pass an additional argument to the registration methods:

```
$app->before(function (Request $req) {
   // ...
}, 32);
```

As a convenience, two constants allow you to register an event as early as possible or as late as possible:

```
$app->before(function (Request $req) {
    // ...
}, Application::EARLY_EVENT);
$app->before(function (Request $req) {
    // ...
}, Application::LATE_EVENT);
```

## **Short-circuiting the Controller**

If a before middleware returns a Response object, the Request handling is short-circuited (the next middlewares won't be run, nor the route callback), and the Response is passed to the after middlewares right away:

```
$app->before(function (Request $req) {
   // redirect the user to the login screen if access
   // to the Resource is protected

if (...) {
   return new RedirectResponse('/login');
   }
});
```

If a before middleware does not return a Response or null, a RuntimeException is thrown.



#### **Definition**

Define the service:

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

Retrieve the service:

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

Every time \$app['some\_service'] is called,
a new instance of the service is created.

#### **Shared Services**

Use the same instance of a service across all of your code. Create the service on first invocation, and then return the existing instance on any subsequent access.

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

E.g. 2:
$app['asset_path'] = $app->share(function () {
    // logic to determine the asset path
    return 'http://assets.examples.com';
});

E.g. 3:
$app['user.persist_path'] = '/tmp/users';

$app['user.persister'] = $app->share(function ($app) {
    return new JsonUserPersister($app['user.persist_path']);
});
```

## **Access Container from Closure**

Access the service container from within a service definition closure.

For example when fetching services the current service depends on.

# protected closures do not get access to the container Protected Closures

## The container sees closures as factories for services, so

it will always execute them when reading them.

In some cases you will however want to store a closure as a parameter, so that you can fetch it and execute it yourself -- with your own arguments.

This is why Pimple allows to protect closures from being executed, by using the protect method:

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

// will not execute the closure
$add = $app['closure_parameter'];

// calling it now
echo $add(2, 3);
```

## Core Services

request Current request object (instance of Request).

\_ are shared

It gives access to GET, POST parameters

and lots more! E.g.:

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

Only available when a request is being served, you can only access it from within a controller, an application before/after

middlewares, or an error handler.

routes RouteCollection that is used internally.

You can add, modify, and read routes.

controllers Silex\ControllerCollection that is used

internally.

dispatcher EventDispatcher that is used internally.

It is the core of the Symfony2 system

and is used quite a bit by Silex.

resolver ControllerResolver that is used internally.

It takes care of executing the controller

with the right arguments.

kernel HttpKernel that is used internally. Is the

heart of Symfony2, it takes a Request as input and returns a Response as output

request\_context Simplified representation of the

request that is used by the Router and

the UrlGenerator

exception handler Default handler that is used when

you don't register one via the error()
method or if the handler does not return

a Response. Disable it with

\$app['exception\_handler']->disable().

logger Psr\Log\LoggerInterface instance.

By default, logging is disabled as the value is set to null. To enable logging either use the MonologServiceProvider or define your own logger service that conforms to the PSR logger interface. In versions of Silex before 1.1 this must be a Symfony\Component\HttpKernel\

Log\LoggerInterface.

## **Providers**

Allow to reuse parts of an application into another one

#### **Service Providers**

### Loading providers

In order to load and use a service provider, register it on the application:

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

It is possible to provide some parameters as a second argument. These will be set after the provider is registered, but before it is booted:

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

## Creating a provider

Providers must implement the Silex\ServiceProviderInterface:

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

Just create a new class that implements the two methods:

```
register()

define services on the application which then may make use of other services and parameters

boot()

configure the application, just before it handles a request
```

#### E.g.:

#### Using the provider:

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

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

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

#### **Controller Providers**

### **Loading providers**

To load and use a controller provider, "mount" its controllers under a path:

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

All controllers defined by the provider will now be available under the /blog path.

### Creating a provider

```
Providers must implement the Silex\ControllerProviderInterface:
interface ControllerProviderInterface
{
   public function connect(Application $app);
}
```

The connect method must return an instance of ControllerCollection. ControllerCollection is the class where all controller related methods are defined (like get, post, match, ...).

```
namespace Acme;
use Silex\Application;
use Silex\ControllerProviderInterface;

class HelloControllerProvider implements ControllerProviderInterface {
    public function connect(Application $app)
    {
        // creates a new controller based on the default route
        $controllers = $app['controllers_factory'];
    }
}
```

return \$app->redirect('/hello');

});

return \$controllers;
}

#### Using:

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

\$controllers->get('/', function (Application \$app) {

#### **Included Providers**

Providers available in the Silex\Provider namespace (https://github.com/silexphp/Silex-Providers):

DoctrineServiceProvider MonologServiceProvider SessionServiceProvider SerializerServiceProvider SwiftmailerServiceProvider TwigServiceProvider

TranslationServiceProvider
UrlGeneratorServiceProvider
ValidatorServiceProvider
HttpCacheServiceProvider

FormServiceProvider

SecurityServiceProvider

RememberMeServiceProvider

ServiceControllerServiceProvider



Integration with the Doctrine DBAL for easy database access

#### **Parameters**

db.options Array of Doctrine DBAL options.
These options are available:

driver The database driver to use.

Can be: pdo\_mysql, pdo\_sqlite, pdo\_pgsql, pdo\_oci, oci8, ibm\_db2,

pdo\_ibm, pdo\_sqlsrv (default: pdo\_mysql)

dbname Name of the database

host Host of the database.

(default: localhost)

user User of the database. (default: root)

password of the database

charset Only relevant for pdo\_mysql, and

pdo\_oci/oci8, specifies the charset used

when connecting to the database

Only relevant for pdo\_sqlite, specifies

the path to the SQLite database

port Specifies the port of the database.

Only relevant for pdo\_mysql, pdo\_pgsql,

and pdo\_oci/oci8

#### **Services**

path

db The database connection, instance of

Doctrine\DBAL\Connection

db.config Configuration object for Doctrine.

(default: empty Doctrine\DBAL\Configuration)

db.event\_manager Event Manager for Doctrine

## Registering

#### Usage

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

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

#### Using multiple databases

```
$app->register(new Silex\Provider\DoctrineServiceProvider(),
               array(
                  'dbs.options' => array (
                    'mysql read' => array(
                       'driver' => 'pdo_mysql',
                       'host' => 'mysql_read.someplace.tld',
                       'dbname' => 'mv database'.
                       'user' => 'my_username',
                       'password' => 'my_password',
                       'charset' => 'utf8'.
                    'mysql_write' => array(
                       'driver' => 'pdo_mysql',
                       'host' => 'mysql_write.someplace.tld',
                       'dbname' => 'my_database',
                       'user' => 'my_username',
                       'password' => 'my password',
                       'charset' => 'utf8',
         ));
```

The first registered connection is the default, so the lines below are equivalent:

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

#### Using multiple connections:

## **Monolog Service Provider**

Log requests and errors and allow to add logging to app

#### **Parameters**

monolog.logfile File where logs are written to

\* monolog.bubble Whether the messages that are handled

can bubble up the stack or not

\* monolog.permission File permissions default, nothing change

Level of logging, defaults to DEBUG. \* monolog.level

Must be one of:

Logger::DEBUG (log everything), Logger::INFO (log everything except

DEBUG).

Logger::WARNING, Logger::ERROR.

In addition to the Logger:: constants, it is also possible to supply the level in string form, e.g.: "DEBUG", "INFO",

"WARNING". "ERROR"

\* monolog.name Name of the monolog

channel

#### Services

monolog The monolog logger instance

\$app['monolog']->addDebug('Test');

An event listener to log requests, monolog.listener

responses and errors

## Registering

```
$app->register(new Silex\Provider\MonologServiceProvider(),
                    array(
                       'monolog.logfile' => __DIR___.'/dev.log',
```

#### Usage

```
$app->post('/user', function () use ($app) {
  $app['monolog']->addInfo(sprintf("User '%s' registered.",
                                    $username));
  return new Response(", 201);
}):
```

#### Customization

It is possible to configure Monolog (like adding or changing the handlers) before using it by extending the monolog service:

\$app['monolog'] = \$app->share(\$app->extend('monolog', function(\$monolog, \$app) { \$monolog->pushHandler(...);

> return \$monolog; }));

#### **Traits**

log Logs a message

\$app->log(sprintf("User '%s' registered.", \$username));

## **Session Service Provider**

Service for storing data persistently between requests

#### **Parameters**

\* session.storage.save\_path (default: value of

Path for the NativeFileSessionHandler.

session.storage.options

An array of options that is passed to the constructor of the session.storage service In case of the default NativeSessionStorage, the most useful options are:

name The cookie name. (default: \_SESS)

id The session id. (default: null)

cookie lifetime Cookie lifetime cookie path Cookie path cookie domain Cookie domain

cookie secure Cookie secure (HTTPS)

cookie\_httponly Whether the cookie is http only

All of these are optional. Default Sessions life time is 1800 seconds (30 minutes). To override, set the lifetime option.

session.test Whether to simulate sessions or not

(useful when writing functional tests).

#### Services

Instance of Symfony2's Session session

Service that is used for session.storage

persistence of the session data

session.storage.handler

Service that is used by the session.storage for data access

NativeFileSessionHandler)

## Registering

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

### Usage

```
$app['session']->set('user', array('username' => $username));
```

\$user = \$app['session']->get('user'))

### **Custom Session Configurations**

When using a custom session configuration is necessary to disable the NativeFileSessionHandler by setting session.storage.handler to null and configure the session.save path ini setting yourself

\$app['session.storage.handler'] = null;

#### Swiftmailer Service Provider

Service for sending email through the Swift Mailer library

#### **Parameters**

swiftmailer.use\_spool A boolean to specify whether or not to use the memory spool.

swiftmailer.options

An array of options for the default SMTP-based configuration. The following options can be set:

host SMTP hostname.

port SMTP port. (default: 25)

username SMTP username.

(default: empty string)

SMTP password. password

(default: empty string)

encryption SMTP encryption.

auth mode SMTP authentication

mode. (default: null)

\$app['swiftmailer.options'] = array( 'host' => 'host'. 'port' => '25', 'username' => 'username', 'password' => 'password', 'encryption' => null, 'auth mode' => null

#### Services

The mailer instance mailer

```
$message = \Swift_Message::newInstance();
$app['mailer']->send($message):
```

swiftmailer.transport

StreamBuffer used by the

Transport used for e-mail

transport

delivery.

swiftmailer.transport.authhandler

swiftmailer.transport.buffer

Authentication handler used by the transport. Try by default: CRAM-MD5, login, plaintext

swiftmailer.transport.eventdispatcher Internal event dispatcher

used by Swiftmailer

## Registering

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

#### Usage

```
$app->post('/feedback', function () use ($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!', 201);
});
```

#### **Traits**

Sends an email mail

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

#### **Translation Service Provider**

Service for translating your app into different languages

#### **Parameters**

\* translator.domains Mapping of domains/locales/messages. Contains the translation data for all

languages and domains

\* locale

Locale for the translator. Generally set this based on some request parameter

\* locale\_fallbacks

Fallback locales for the translator. Used when the current locale has no

messages set

#### Services

translator

Instance of Translator, that is used for translation

translator.loader (default: ArrayLoader) Instance of an implementation of the translation

LoaderInterface

#### Registering

```
$app->register(new Silex\Provider\TranslationServiceProvider(),
                    arrav(
                         'locale_fallbacks' => array('en'),
               ));
```

#### Usage

```
$app['translator.domains'] = array(
  '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%',
  'validators' => array(
    'fr' => array(
       'This value should be a valid number.' =>
       'Cette valeur doit être un nombre.'.
$app->get('/{_locale}/{message}/{name}',
          function ($message, $name) use ($app) {
            return $app['translator']->trans(
                      $message.
                      array('%name%' => $name)
     });
```

The above example will result in following routes:

- · /en/hello/igor will return Hello igor
- · /de/hello/igor will return Hallo igor
- /fr/hello/igor will return Bonjour igor
- /it/hello/igor will return Hello igor (fallback)

#### Traits

trans

Translates the given message

transChoice

Translates the given choice message by choosing a translation according to a number

E.g.:

\$app->trans('Hello World'); \$app->transChoice('Hello World');

## Twig Service Provider

Provide integration with the Twig template engine

#### **Parameters**

Path to the directory containing twig \* twig.path

template files (it can also be an

array of paths)

\* twig.templates Associative array of template names

to template contents. Use this if you want to define your templates inline

\* twig.options Associative array of twig options

\* twig.form.templates An array of templates used to render

forms (only available when the FormServiceProvider is enabled)

```
$app['twig.path']
                    = array(__DIR__.'/../templates');
$app['twig.options'] = array(
                        'cache' => __DIR___.'/../var/cache/twig'
```

#### Services

Twig Environment instance. Main way twig

of interacting with Twig

The loader for Twig templates which uses the twig.loader

twig.path and the twig.templates options. The loader can be replaced completely

#### Registering

```
$app->register(new Silex\Provider\TwigServiceProvider(),
                     arrav(
                        'twig.path' => ___DIR___.'/views',
               ));
```

#### Usage

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

#### app variable

In any Twig template, the app variable refers to the Application object. So you can access any service from within your view.

```
E.g.: to access $app['request']->getHost(),
just put this in your template:
{{app.request.host}}
```

#### render function

A render function is also registered to help render another controller from a template:

```
{{render(app.request.baseUrl ~ '/sidebar') }}
{ # or if using the UrlGeneratorServiceProvider # }
{ render(url('sidebar')) } }
```

#### **Traits**

render Renders a view with the given parameters and returns a Response object.

```
return $app->render('index.html', ['name' => 'Fabien']);
```

#### Customization

You can configure the Twig environment before using it by extending the twig service:

```
$app['twig'] = $app->share(
               $app->extend('twig', function($twig, $app) {
                 $twig->addGlobal('pi', 3.14);
                 $twig->addFilter('levenshtein',
                    new \Twig_Filter_Function('levenshtein'));
                 return $twig;
              }));
```

\* - optional

#### **UrlGenerator Service Provider**

Service for generating URLs for named routes

#### Services

url\_generator An instance of UrlGenerator, using the RouteCollection that is provided through the routes service. It has a generate method. which takes the route name as an argument, followed by an array of route parameters

### Registering

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

## Usage

```
$app->get('/', function () {
  return 'welcome to the homepage';
->bind('home');
$app->get('/navigation', function () use ($app) {
  return 'ka href="'.
         $app['url_generator']->generate('home').
         ">Home</a>';
});
```

When using Twig, the service can be used like this:

```
{{ app.url generator.generate('home') }}
```

if you have twig-bridge as a Composer dep, you will have access to the path() and url() functions:

```
{{ path('home') }}
{{ url('home') }}
{ # generates the absolute url http://example.org/ # }
```

#### **Traits**

```
path
                                     $app->path('home');
        Generates a path
                                    $app->url('home');
        Generates an absolute URL
url
```

#### **Validator Service Provider**

Service for validating data. It is most useful when used with the FormServiceProvider, but can also be used standalone

#### Services

validator

Instance of Validator

validator.mapping. class metadata factory Factory for metadata loaders, which can read validation constraint information from classes. Defaults to StaticMethodLoader--ClassMetadataFactory.

This means you can define a static loadValidatorMetadata method on your data class, which takes a ClassMetadata argument. Then you can set constraints on this ClassMetadata instance

validator.validator\_factory Factory for ConstraintValidators.

Defaults to a standard ConstraintValidatorFactory. Mostly used internally by the Validator

## Registering

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

#### Usage

#### Validating Values

use Symfony\Component\Validator\Constraints as Assert;

```
$app->get('/validate/{email}', function ($email) use ($app) {
  $errors = $app['validator']->validateValue($email,
                                         new Assert\Email());
  if (count($errors) > 0) {
     return (string) $errors;
  return 'The email is valid':
}):
```

#### **Validating Objects**

```
use Symfony\Component\Validator\Constraints as Assert;
$author = new Author();
$author->first name = 'Fabien';
$author->last name = 'Potencier';
$book = new Book():
$book->title = 'My Book';
$book->author = $author:
$metadata = $app['validator.mapping.class metadata factory']
                        ->getMetadataFor('Author');
$metadata->addPropertyConstraint('first_name',
                        new Assert\NotBlank()):
$metadata->addPropertyConstraint('first_name',
                        new Assert\Length(array('min' => 10)));
$metadata->addPropertyConstraint('last name'.
                        new Assert\Length(array('min' => 10)));
$metadata = $app['validator.mapping.class_metadata_factory']
                        ->getMetadataFor('Book');
$metadata->addPropertyConstraint('title',
                        new Assert\Length(array('min' => 10)));
$metadata->addPropertyConstraint('author'.
                        new Assert\Valid());
$errors = $app['validator']->validate($book);
if (count($errors) > 0) {
  foreach ($errors as $error) {
     echo $error->getPropertvPath().''.
          $error->getMessage()."\n";
```

#### **Translation**

echo 'The author is valid';

}else {

```
$app['translator.domains'] = array(
  'validators' => array(
     'fr' => arrav(
        'This value should be a valid number.' =>
        'Cette valeur doit être un nombre.',
 ));
```

#### Form Service Provider

Service for building forms with the Symfony2 Form component

#### **Parameters**

form.secret md5(\_\_DIR\_\_)) This secret value is used for generating and validating the CSRF token for a specific page. It is very important to set this value to a static randomly generated value, to prevent hijacking of your forms

#### Services

form.factory

Instance of FormFactory, that is used for build a form

form.csrf provider

Instance of an implementation of the DefaultCsrfProvider) CsrfProviderInterface

### Registering

\$app->register(new FormServiceProvider());

#### Usage

```
$app->match('/form', function (Reguest $reg) use ($app) {
  // default data for when the form is displayed the first time
  $data = array(
     'name' => 'Your name',
     'email' => 'Your email'.
  $form = $app['form.factory']->createBuilder('form', $data)
     ->add('name')
     ->add('email')
     ->add('gender', 'choice', array(
           'choices' => array(1 => 'male', 2 => 'female'),
           'expanded' => true,
     ->getForm();
```

```
$form->handleRequest($req);
  if ($form->isValid()) {
     $data = $form->getData();
     // do something with the data
     // redirect somewhere
     return $app->redirect('...');
  // display the form
  return $app['twig']->render('index.twig', array(
                   'form' => $form->createView()
});
```

#### **Traits**

form

Creates a FormBuilder instance

\$app->form(\$data):

## HttpCache Service Provider

Provides support for the Symfony2 Reverse Proxy

#### **Parameters**

http\_cache.cache\_dir

Cache directory to store the HTTP

cache data

\* http\_cache.options

An array of options for the HttpCache constructor

#### Services

http\_cache

Instance of HttpCache

http\_cache.esi

Instance of Esi, that implements the ESI capabilities to Request and Response

instances

http cache.store

Instance of Store, that implements all the logic for storing cache metadata

(Request and Response headers)

#### Registering

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

#### Usage

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

If you want Silex to trust the X-Forwarded-For\* headers from your reverse proxy at address \$ip, you will need to whitelist it as documented in Trusting Proxies. If you would be running Varnish in front of your app on the same machine:

Request::setTrustedProxies(array('127.0.0.1', '::1')); \$app->run();

## Using Symfony2 reverse proxy natively (with http\_cache service)

The Symfony2 reverse proxy acts much like any other proxy would, so whitelist it:

```
Request::setTrustedProxies(array('127.0.0.1'));
$app['http cache']->run();
```

### **Disabling ESI**

```
$app->register(new Silex\Provider\HttpCacheServiceProvider(),
               arrav(
                  'http cache.cache dir' => DIR .'/cache/',
                  'http cache.esi' => null,
               ));
```

\* - optional

Symfony 2.4+

## HttpFragment Service Provider

Allows to embed fragments of HTML in a template

#### **Parameters**

fragment.path Path to use for the URL generated for ESI and HInclude URLs

/\_fragment)

uri\_signer.secret Secret to use for the URI signer

service (used for the HInclude

renderer)

fragment.renderers. Content or Twig template to use hinclude.global\_template for the default content when using

the Hinclude renderer

#### Services

fragment.handler Instance of FragmentHandler

fragment.renderers Array of fragment renderers

(by default, the inline, ESI, and HInclude

renderers are pre-configured)

### Registering

\$app->register(new Silex\Provider\HttpFragmentServiceProvider()

#### Usage

Using Twig for your templates:

The main page content. {{ render('/foo') }}

The main page content resumes here.

## **Security Service Provider**

Manages authentication and authorization for apps

#### **Parameters**

Defines whether to hide user \* security.hide\_user\_not\_found not found exception or not

#### **Services**

Main entry point for the security security provider. Use it to get the current

user token

Instance of security.

authentication\_manager AuthenticationProviderManager,

responsible for authentication

Instance of security.access\_manager

> AccessDecisionManager, responsible for authorization

security.session\_strategy Define the session strategy used

for authentication (default to a

migration strategy)

security.user checker Checks user flags after

authentication

Returns the last authentication security.last error

errors when given a Request object

security.encoder\_factory Defines the encoding strategies

for user passwords (default to use a digest algorithm for all users)

security.encoder.digest The encoder to use by default for

all users

#### Registering

```
$app->register(new Silex\Provider\SecurityServiceProvider(),
                      'security.firewalls' => // see below
               ));
```

The security features are only available after the Application has been booted. So, to use it outside of the handling of a request, call boot() first:

\$app->boot();

#### Usage

// Current user

```
$token = $app['security']->getToken();
if (null !== $token) {
   $user = $token->getUser();
// Securing a Path with HTTP Authentication
$app['security.firewalls'] = array(
   'admin' => arrav(
     'pattern' => '^/admin',
     'http' => true.
     'users' => array(
        // raw password is foo
        'admin' => array('ROLE ADMIN',
         '5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsYQ8t+a8...'),
// Find the encoder for a UserInterface instance
$encoder = $app['security.encoder factory']->getEncoder($user);
// Compute the encoded password for foo
$password = $encoder->encodePassword('foo', $user->getSalt());
// Checking User Roles
if ($app['security']->isGranted('ROLE_ADMIN')) {
  // ...
// Allowing Anonymous Users
$app['security.firewalls'] = array(
   'unsecured' => array(
     'anonymous' => true.
     // ...
```

#### **Traits**

user Returns the current user encodePassword Encode a given password

secure Secures a controller for the given roles

```
$user = $app->user();
$encoded = $app->encodePassword($user, 'foo');

$app->get('/', function () {
    // do something but only for admins
})->secure('ROLE_ADMIN');
```

#### RememberMe Service Provider

Adds "Remember-Me" authentication to the SecurityServiceProvider

## Registering

## **Options**

key Secret key to generate tokens (you should

generate a random string)

name Cookie name (default: REMEMBERME)

lifetime Cookie lifetime (default: 315 36000 ~ 1 year)

path Cookie path (default: /)

domain Cookie domain (default: null = request domain)

secure Cookie is secure (default: false)

httponly Cookie is HTTP only (default: true)

always remember me Enable remember me (default: false)

remember\_me\_parameter Name of the request parameter

enabling remember\_me on login.

#### Serializer Service Provider

Provides a serializer service

#### **Services**

serializer Symfony\Component\Serializer\Serializer

serializer.encoders Symfony\Component\Serializer\

Encoder\JsonEncoder and
Symfony\Component\Serializer\

Encoder\XmlEncoder

serializer.normalizers Symfony\Component\Serializer\

Normalizer\CustomNormalizer and Symfony\Component\Serializer\ Normalizer\GetSetMethodNormalizer

#### Registering

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

#### Usage

```
$app = new Application();
$app->register(new SerializerServiceProvider());
```

\$app->get("/pages/{id}.{\_format}", function (\$id) use (\$app) {
 // assume a page\_repository service exists that returns Page
 // objects. Object returned has getters/setters exposing state
 \$page = \$app['page\_repository']->find(\$id);

```
$format = $app['request']->getRequestFormat();
if (!$page instanceof Page) {
    $app->abort("No page found for id: $id");
}

return new Response(
    $app['serializer']->serialize($page, $format), 200,
    array(
    "Content-Type" => $app['request']->getMimeType($format)
));
})->assert("_format", "xml|json")
->assert("id", "\d+");
```

#### ServiceController Service Provider

Controllers can be created as services, providing the full power of dependency injection and lazy loading

#### Registering

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

#### Usage

/posts.json route will use a controller that is defined as a service

```
use Silex\Application;
use Demo\Repository\PostRepository;

$app = new Application();
$app['posts.repository'] = $app->share(function() {
    return new PostRepository;
});

$app->get('/posts.json', function() use ($app) {
    return $app->json($app['posts.repository']->findAll());
});
```

#### Define the controller as a service:

```
$app['posts.controller'] = $app->share(function() use ($app) {
    return new PostController($app['posts.repository']);
});

$app->get('/posts.json', "posts.controller:indexJsonAction");
```