Getting Started

p://docs.slimframework.com/#Getting-rted)

Installation

Composer Install

Install composer in your project:

```
curl -s https://getcomposer.org/installer | php
```

Create a `composer.json` file in your project root:

```
{
    "require": {
        "slim/slim": "2.*"
    }
}
```

Install via composer:

```
php composer.phar install
```

Add this line to your application's `index.php` file:

```
<?php
require 'vendor/autoload.php';
```

Manual Install

Download and extract the Slim Framework into your project directory and `require` it in your application's `index.php` file. You'll also need to register Slim's autoloader.

```
<?php
require 'Slim/Slim.php';
\Slim\Slim::registerAutoloader();
```

Back to Top ↑

System Requirements

- PHP >= 5.3.0

The `mcrypt` extension is required *only* if you use encrypted cookies.

Back to Top ↑

Hello World

Instantiate a Slim application:

```
$app = new \Slim\Slim();
```

Define a HTTP GET route:

```
$app->get('/hello/:name', function ($name) {
    echo "Hello, $name";
});
```

Run the Slim application:

```
$app->run();
```

Back to Top ↑

Configuration

:s.slimframework.com/#Configuration)

Configuration Overview

There are two ways to apply settings to the Slim application. First during Slim application instantiation and second after instantiation. All settings can be applied at instantiation time by passing Slim's constructor an associative array. All settings can be retrieved and modified after instantiation, however some of them can not be done simply by using the config application instance method but will be demonstrated as necessary below. Before I list the available settings, I want to quickly explain how you may define and inspect settings with your Slim application.

During Instantiation

To define settings upon instantiation, pass an associative array into the Slim constructor.

```
<?php
$app = new Slim(array(
    'debug' => true
));
```

After Instantiation

To define settings after instantiation, the majority can use the config application instance method; the first argument is the setting name and the second argument is the setting value.

```
<?php
$app->config('debug', false);
```

You may also define multiple settings at once using an associative array:

```
<?php
$app->config(array(
    'debug' => true,
    'templates.path' => '../templates'
));
```

To retrieve the value of a setting, you also use the config application instance method; however, you only pass one argument - the name of the setting you wish to inspect. If the setting you request does not exist, `null` is returned.

```
<?php
$settingValue = $app->config('templates.path'); //returns "../templates"
```

You are not limited to the settings shown below; you may also define your own.

Back to Top ↑

Application Settings

mode

This is an identifier for the application's current mode of operation. The mode does not affect a Slim application's internal functionality. Instead, the mode is only for you to optionally invoke your own code for a given mode with the `configMode()` application method.

The application mode is declared during instantiation, either as an environment variable or as an argument to the Slim application constructor. It cannot be changed afterward. The mode may be anything you want — "development", "test", and "production" are typical, but you are free to use anything you want (e.g. "foo").

```
<?php
$app = new \Slim\Slim(array(
    'mode' => 'development'
));
```

Data Type
string
Default Value
"development"

debug

Heads Up! Slim converts errors into `ErrorException` instances.

If debugging is enabled, Slim will use its built-in error handler to display diagnostic information for uncaught Exceptions. If debugging is disabled, Slim will instead invoke your custom error handler, passing it the otherwise uncaught Exception as its first and only argument.

```
<?php
$app = new \Slim\Slim(array(
    'debug' => true
));
```

Data Type
boolean
Default Value
true

log.writer

Use a custom log writer to direct logged messages to the appropriate output destination. By default, Slim's logger will write logged messages to `STDERR`. If you use a custom log writer, it must implement this interface:

```
public write(mixed $message, int $level);
```

The `write()` method is responsible for sending the logged message (not necessarily a string) to the appropriate output destination (e.g. a text file, a database, or a remote web service).

To specify a custom log writer after instantiation you must access Slim's logger directly and use its `setWriter()` method:

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'log.writer' => new \My\LogWriter()
));

// After instantiation
$log = $app->getLog();
$log->setWriter(new \My\LogWriter());
```

Data Type
mixed
Default Value
\Slim\LogWriter

log.level

Heads Up! Use the constants defined in `\Slim\Log` instead of integers.

Slim has these log levels:

- \Slim\Log::EMERGENCY
- \Slim\Log::ALERT
- \Slim\Log::CRITICAL
- \Slim\Log::ERROR
- \Slim\Log::WARN
- \Slim\Log::NOTICE
- \Slim\Log::INFO
- \Slim\Log::DEBUG

The `log.level` application setting determines which logged messages will be honored and which will be ignored. For example, if the `log.level` setting is `\Slim\Log::INFO`, debug messages will be ignored while info, warn, error, and fatal messages will be logged.

To change this setting after instantiation you must access Slim's logger directly and use its `setLevel()` method.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'log.level' => \Slim\Log::DEBUG
));

// After instantiation
$log = $app->getLog();
$log->setLevel(\Slim\Log::WARN);
```

Data Type
integer
Default Value
\Slim\Log::DEBUG

log.enabled

This enables or disables Slim's logger. To change this setting after instantiation you need to access Slim's logger directly and use its `setEnabled()` method.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'log.enabled' => true
));

// After instantiation
$log = $app->getLog();
$log->setEnabled(true);
```

Data Type
boolean
Default Value
true

templates.path

The relative or absolute path to the filesystem directory that contains your Slim application's template files. This path is referenced by the Slim application's View to fetch and render templates.

To change this setting after instantiation you need to access Slim's view directly and use its `setTemplatesDirectory()` method.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'templates.path' => './templates'
));

// After instantiation
$view = $app->view();
$view->setTemplatesDirectory('./templates');
```

Data Type
string
Default Value
"./templates"

view

The View class or instance used by the Slim application. To change this setting after instantiation you need to use the Slim application's `view()` method.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'view' => new \My\View()
));

// After instantiation
$app->view(new \My\View());
```

Data Type
string|\Slim\View
Default Value
\Slim\View

cookies.encrypt

Determines if the Slim app should encrypt its HTTP cookies.

```
<?php
$app = new \Slim\Slim(array(
    'cookies.encrypt' => true
));
```

Data Type
boolean
Default Value
false

cookies.lifetime

Determines the lifetime of HTTP cookies created by the Slim application. If this is an integer, it must be a valid UNIX timestamp at which the cookie expires. If this is a string, it is parsed by the `strtotime()` function to extrapolate a valid UNIX timestamp at which the cookie expires.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.lifetime' => '20 minutes'
));

// After instantiation
$app->config('cookies.lifetime', '20 minutes');
```

Data Type
integer|string
Default Value
“20 minutes”

cookies.path

Determines the default HTTP cookie path if none is specified when invoking the Slim application’s `setCookie()` or `setEncryptedCookie()` methods.

```php
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.path' => '/'
));

// After instantiation
$app->config('cookies.path', '/');
```

Data Type
string
Default Value
"/"

cookies.domain

Determines the default HTTP cookie domain if none specified when invoking the Slim application's `setCookie()` or `setEncryptedCookie()` methods.

```php
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.domain' => 'domain.com'
));

// After instantiation
$app->config('cookies.domain', 'domain.com');
```

Data Type
string
Default Value
null

cookies.secure

Determines whether or not cookies are delivered only via HTTPS. You may override this setting when invoking the Slim application's `setCookie()` or `setEncryptedCookie()` methods.

```php
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.secure' => false
));

// After instantiation
$app->config('cookies.secure', false);
```

Data Type
boolean
Default Value
false

cookies.httponly

Determines whether cookies should be accessible through client side scripts (false = accessible). You may override this setting when invoking the Slim application's `setCookie()` or `setEncryptedCookie()` methods.

```php
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.httponly' => false
));

// After instantiation
$app->config('cookies.httponly', false);
```

Data Type
boolean
Default Value
false

cookies.secret_key

The secret key used for cookie encryption. You should change this setting if you use encrypted HTTP cookies in your Slim application.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.secret_key' => 'secret'
));

// After instantiation
$app->config('cookies.secret_key', 'secret');
```

Data Type
string
Default Value
“CHANGE_ME”

cookies.cipher

The mcrypt cipher used for HTTP cookie encryption. See available ciphers (http://php.net/manual/en/mcrypt.ciphers.php).

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.cipher' => MCRYPT_RIJNDAEL_256
));

// After instantiation
$app->config('cookies.cipher', MCRYPT_RIJNDAEL_256);
```

Data Type
integer
Default Value
MCRYPT_RIJNDAEL_256

cookies.cipher_mode

The mcrypt cipher mode used for HTTP cookie encryption. See available cipher modes (http://www.php.net/manual/en/mcrypt.constants.php).

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'cookies.cipher_mode' => MCRYPT_MODE_CBC
));

// After instantiation
$app->config('cookies.cipher_mode', MCRYPT_MODE_CBC);
```

Data Type
integer
Default Value
MCRYPT_MODE_CBC

http.version

By default, Slim returns an HTTP/1.1 response to the client. Use this setting if you need to return an HTTP/1.0 response. This is useful if you use PHPFog or an nginx server configuration where you communicate with backend proxies rather than directly with the HTTP client.

```
<?php
// During instantiation
$app = new \Slim\Slim(array(
    'http.version' => '1.1'
));

// After instantiation
$app->config('http.version', '1.1');
```

Data Type
string
Default Value
“1.1”

Back to Top ↑

Application Names and Scopes

When you build a Slim application you will enter various scopes in your code (e.g. global scope and function scope). You will likely need a reference to your Slim application in each scope. There are several ways to do this:

- Use application names with the Slim application’s `getInstance()` static method
- Curry an application instance into function scope with the `use` keyword

Application Names

Every Slim application may be given a name. **This is optional**. Names help you get a reference to a Slim application instance in any scope throughout your code. Here is how you set and get an application's name:

```
<?php
$app = new \Slim\Slim();
$app->setName('foo');
$name = $app->getName(); // "foo"
```

Scope Resolution

So how do you get a reference to your Slim application? The example below demonstrates how to obtain a reference to a Slim application within a route callback function. The `$app` variable is used in the global scope to define the HTTP GET route. But the `$app` variable is also needed within the route's callback scope to render a template.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () {
    $app->render('foo.php'); // <-- ERROR
});
```

This example fails because the `$app` variable is unavailable inside the route callback function.

Currying

We can inject the `$app` variable into the callback function with the `use` keyword:

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () use ($app) {
    $app->render('foo.php'); // <-- SUCCESS
});
```

Fetch by Name

You can use the Slim application's `getInstance()` static method, too:

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', 'foo');
function foo() {
    $app = Slim::getInstance();
    $app->render('foo.php');
}
```

Back to Top ↑

Application Modes

It is common practice to run web applications in a specific mode depending on the current state of the project. If you are developing the application, you will run the application in “development” mode; if you are testing the application, you will run the application in “test” mode; if you launch the application, you will run the application in “production” mode.

Slim supports the concept of modes in that you may define your own modes and prompt Slim to prepare itself appropriately for the current mode. For example, you may want to enable debugging in “development” mode but not in “production” mode. The examples below demonstrate how to configure Slim differently for a given mode.

What is a mode?

Technically, an application mode is merely a string of text - like “development” or “production” - that has an associated callback function used to prepare the Slim application appropriately. The application mode may be anything you like: “testing”, “production”, “development”, or even “foo”.

How do I set the application mode?

> **Heads Up!** The application mode may only be set during application instantiation. It may not be changed afterward.

Use an environment variable

If Slim sees an environment variable named “SLIM_MODE”, it will set the application mode to that variable’s value.

```
<?php
$_ENV['SLIM_MODE'] = 'production';
```

Use application setting

If an environment variable is not found, Slim will next look for the mode in the application settings.

```
<?php
$app = new \Slim\Slim(array(
    'mode' => 'production'
));
```

Default mode

If the environment variable and application setting are not found, Slim will set the application mode to “development”.

Configure for a Specific Mode

After you instantiate a Slim application, you may configure the Slim application for a specific mode with the Slim application's `configureMode()` method. This method accepts two arguments: the name of the target mode and a callable function to be immediately invoked if the first argument matches the current application mode.

Assume the current application mode is "production". Only the callable associated with the "production" mode will be invoked. The callable associated with the "development" mode will be ignored until the application mode is changed to "development".

```
<?php
// Set the current mode
$app = new \Slim\Slim(array(
    'mode' => 'production'
));

// Only invoked if mode is "production"
$app->configureMode('production', function () use ($app) {
    $app->config(array(
        'log.enable' => true,
        'debug' => false
    ));
});

// Only invoked if mode is "development"
$app->configureMode('development', function () use ($app) {
    $app->config(array(
        'log.enable' => false,
        'debug' => true
    ));
});
```

Back to Top ↑

Routing

›://docs.slimframework.com/#Routing)

Routing Overview

The Slim Framework helps you map resource URIs to callback functions for specific HTTP request methods (e.g. GET, POST, PUT, DELETE, OPTIONS or HEAD). A Slim application will invoke the first route that matches the current HTTP request's URI and method.

If the Slim application does not find routes with URIs that match the HTTP request URI and method, it will automatically return a **404 Not Found** response.

Back to Top ↑

GET Routes

Use the Slim application's `get()` method to map a callback function to a resource URI that is requested with the HTTP GET method.

```
<?php
$app = new \Slim\Slim();
$app->get('/books/:id', function ($id) {
    //Show book identified by $id
});
```

In this example, an HTTP GET request for "/books/1" will invoke the associated callback function, passing "1" as the callback's argument.

The first argument of the Slim application's `get()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an anonymous function (http://php.net/manual/en/functions.anonymous.php).

Back to Top ↑

POST Routes

Use the Slim application's `post()` method to map a callback function to a resource URI that is requested with the HTTP POST method.

```
<?php
$app = new \Slim\Slim();
$app->post('/books', function () {
    //Create book
});
```

In this example, an HTTP POST request for "/books" will invoke the associated callback function

The first argument of the Slim application's `post()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an anonymous function (http://php.net/manual/en/functions.anonymous.php).

Back to Top ↑

PUT Routes

Use the Slim application's `put()` method to map a callback function to a resource URI that is requested with the HTTP PUT method.

```
<?php
$app = new \Slim\Slim();
$app->put('/books/:id', function ($id) {
    //Update book identified by $id
});
```

In this example, an HTTP PUT request for "/books/1" will invoke the associated callback function, passing "1" as the callback function's argument.

The first argument of the Slim application's `put()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an anonymous function (http://php.net/manual/en/functions.anonymous.php).

Method Override

Unfortunately, modern browsers do not provide native support for HTTP PUT requests. To work around this limitation, ensure your HTML form's method attribute is "post", then add a method override parameter to your HTML form like this:

```
<form action="/books/1" method="post">
    ... other form fields here...
    <input type="hidden" name="_METHOD" value="PUT"/>
    <input type="submit" value="Update Book"/>
</form>
```

If you are using Backbone.js (http://documentcloud.github.com/backbone/) or a command-line HTTP client, you may also override the HTTP method by using the **X-HTTP-Method-Override** header.

Back to Top ↑

DELETE Routes

Use the Slim application's `delete()` method to map a callback function to a resource URI that is requested with the HTTP DELETE method.

```
<?php
$app = new \Slim\Slim();
$app->delete('/books/:id', function ($id) {
    //Delete book identified by $id
});
```

In this example, an HTTP DELETE request for “/books/1” will invoke the associated callback function, passing “1” as the callback function’s argument.

The first argument of the Slim application’s `delete()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an anonymous function (http://php.net/manual/en/functions.anonymous.php).

Method Override

Unfortunately, modern browsers do not provide native support for HTTP DELETE requests. To work around this limitation, ensure your HTML form’s method attribute is “post”, then add a method override parameter to your HTML form like this:

```
<form action="/books/1" method="post">
    ... other form fields here...
    <input type="hidden" name="_METHOD" value="DELETE"/>
    <input type="submit" value="Delete Book"/>
</form>
```

If you are using Backbone.js (http://documentcloud.github.com/backbone/) or a command-line HTTP client, you may also override the HTTP method by using the **X-HTTP-Method-Override** header.

Back to Top ↑

OPTIONS Routes

Use the Slim application’s `options()` method to map a callback function to a resource URI that is requested with the HTTP OPTIONS method.

```
<?php
$app = new \Slim\Slim();
$app->options('/books/:id', function ($id) {
    //Return response headers
});
```

In this example, an HTTP OPTIONS request for “/books/1” will invoke the associated callback function, passing “1” as the callback function’s argument.

The first argument of the Slim application’s `options()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an anonymous function (http://php.net/manual/en/functions.anonymous.php).

Method Override

Unfortunately, modern browsers do not provide native support for HTTP OPTIONS requests. To work around this limitation, ensure your HTML form’s method attribute is “post”, then add a method override parameter to your HTML form like this:

```
<form action="/books/1" method="post">
    ... other form fields here...
    <input type="hidden" name="_METHOD" value="OPTIONS"/>
    <input type="submit" value="Fetch Options For Book"/>
</form>
```

If you are using Backbone.js (http://documentcloud.github.com/backbone/) or a command-line HTTP client, you may also override the HTTP method by using the **X-HTTP-Method-Override** header.

Back to Top ↑

PATCH Routes

Use the Slim application's `patch()` method to map a callback function to a resource URI that is requested with the HTTP PATCH method.

```
<?php
$app = new \Slim\Slim();
$app->patch('/books/:id', function ($id) {
    // Patch book with given ID
});
```

In this example, an HTTP PATCH request for "/books/1" will invoke the associated callback function, passing "1" as the callback function's argument.

The first argument of the Slim application's `patch()` method is the resource URI. The last argument is anything that returns `true` for `is_callable()`. Typically, the last argument will be an [anonymous function][anon-func].

Back to Top ↑

Custom HTTP Methods

One route, multiple HTTP methods

Sometimes you may need a route to respond to multiple HTTP methods; sometimes you may need a route to respond to a custom HTTP method. You can accomplish both with the Route object's `via()` method. This example demonstrates how to map a resource URI to a callback that responds to multiple HTTP methods.

```
<?php
$app = new \Slim\Slim();
$app->map('/foo/bar', function() {
    echo "I respond to multiple HTTP methods!";
})->via('GET', 'POST');
$app->run();
```

The route defined in this example will respond to both GET and POST requests for the resource identified by "/foo/bar". Specify each appropriate HTTP method as a separate string argument to the Route object's `via()` method. Like other Route methods (e.g. `name()` and `conditions()`), the `via()` method is chainable:

```
<?php
$app = new \Slim\Slim();
$app->map('/foo/bar', function() {
    echo "Fancy, huh?";
})->via('GET', 'POST')->name('foo');
$app->run();
```

One route, custom http methods

The Route object's `via()` method is not limited to just GET, POST, PUT, DELETE, and OPTIONS methods. You may also specify your own custom HTTP methods (e.g. if you were responding to WebDAV HTTP requests). You can define a route that responds to a custom "FOO" HTTP method like this:

```
<?php
$app = new \Slim\Slim();
$app->map('/hello', function() {
    echo "Hello";
})->via('FOO');
$app->run();
```

Back to Top ↑

Route Parameters

You can embed parameters into route resource URIs. In this example, I have two parameters in my route URI, ":one" and ":two".

```
<?php
$app = new \Slim\Slim();
$app->get('/books/:one/:two', function ($one, $two) {
    echo "The first parameter is " . $one;
    echo "The second parameter is " . $two;
});
```

To create a URL parameter, prepend ":" to the parameter name in the route URI pattern. When the route matches the current HTTP request, the values for each route parameter are extracted from the HTTP request URI and are passed into the associated callback function in order of appearance.

Wildcard route parameters

You may also use wildcard route parameters. These will capture one or many URI segments that correspond to the route pattern's wildcard parameter into an array. A wildcard parameter is identified by a "+" suffix; it otherwise acts the same as normal route parameters shown above. Here's an example:

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:name+', function ($name) {
    // Do something
});
```

When you invoke this example application with a resource URI "/hello/Josh/T/Lockhart", the route callback's `$name` argument will be equal to `array('Josh', 'T', Lockhart')`.

Optional route parameters

Heads Up! Optional route segments are experimental. They should only be used in the manner demonstrated below.

You may also have optional route parameters. These are ideal for using one route for a blog archive. To declare optional route parameters, specify your route pattern like this:

```
<?php
$app = new Slim();
$app->get('/archive(/:year(/:month(/:day)))', function ($year = 2010, $month = 12, $day = 05) {
    echo sprintf('%s-%s-%s', $year, $month, $day);
});
```

Each subsequent route segment is optional. This route will accept HTTP requests for:

- /archive
- /archive/2010
- /archive/2010/12
- /archive/2010/12/05

If an optional route segment is omitted from the HTTP request, the default values in the callback signature are used instead.

Currently, you can only use optional route segments in situations like the example above where each route segment is subsequently optional. You may find this feature unstable when used in scenarios different from the example above.

Back to Top ↑

Route Names

Slim lets you assign a name to a route. Naming a route enables you to dynamically generate URLs using the urlFor helper method. When you use the Slim application's `urlFor()` method to create application URLs, you can freely change route patterns without breaking your application. Here is an example of a named route:

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:name', function ($name) {
    echo "Hello, $name!";
})->name('hello');
```

You may now generate URLs for this route using the `urlFor()` method, described later in this documentation. The route `name()` method is also chainable:

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:name', function ($name) {
    echo "Hello, $name!";
})->name('hello')->conditions(array('name' => '\w+'));
```

Back to Top ↑

Route Conditions

Slim lets you assign conditions to route parameters. If the specified conditions are not met, the route is not run. For example, if you need a route with a second segment that must be a valid 4-digit year, you could enforce this condition like this:

```
<?php
$app = new \Slim\Slim();
$app->get('/archive/:year', function ($year) {
    echo "You are viewing archives from $year";
})->conditions(array('year' => '(19|20)\d\d'));
```

Invoke the Route object's `conditions()` method. The first and only argument is an associative array with keys that match any of the route's parameters and values that are regular expressions.

Application-wide route conditions

If many of your Slim application Routes accept the same parameters and use the same conditions, you can define default application-wide Route conditions like this:

```
<?php
\Slim\Route::setDefaultConditions(array(
    'firstName' => '[a-zA-Z]{3,}'
));
```

Define application-wide route conditions before you define application routes. When you define a route, the route will automatically be assigned any application-wide Route conditions defined with `\Slim\Route::setDefaultConditions()`. If for whatever reason you need to get the application-wide default route conditions, you can fetch them with `\Slim\Route::getDefaultConditions()`. This static method returns an array exactly as the default route conditions were defined.

You may override a default route condition by redefining the route's condition when you define the route, like this:

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:firstName', $callable)
    ->conditions(array('firstName' => '[a-z]{10,}'));
```

You may append new conditions to a given route like this:

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:firstName/:lastName', $callable)
    ->conditions(array('lastName' => '[a-z]{10,}'));
```

Back to Top ↑

Route Middleware

Slim enables you to associate middleware with a specific application route. When the given route matches the current HTTP request and is invoked, Slim will first invoke the associated middleware in the order they are defined.

What is route middleware?

Route middleware is anything that returns `true` for `is_callable`. Route middleware will be invoked in the sequence defined before its related route callback is invoked.

How do I add route middleware?

When you define a new application route with the Slim application's `get()`, `post()`, `put()`, or `delete()` methods you must define a route pattern and a callable to be invoked when the route matches an HTTP request.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () {
    //Do something
});
```

In the example above, the first argument is the route pattern. The last argument is the callable to be invoked when the route matches the current HTTP request. The route pattern must always be the first argument. The route callable must always be the last argument.

You can assign middleware to this route by passing each middleware as a separate interior or… (ahem) middle… argument like this:

```
<?php
function mw1() {
    echo "This is middleware!";
}
function mw2() {
    echo "This is middleware!";
}
$app = new \Slim\Slim();
$app->get('/foo', 'mw1', 'mw2', function () {
    //Do something
});
```

When the /foo route is invoked, the `mw1` and `mw2` functions will be invoked in sequence before the route's callable is invoked.

Suppose you wanted to authenticate the current user against a given role for a specific route. You could use some closure magic like this:

```
<?php
$authenticateForRole = function ( $role = 'member' ) {
    return function () use ( $role ) {
        $user = User::fetchFromDatabaseSomehow();
        if ( $user->belongsToRole($role) === false ) {
            $app = \Slim\Slim::getInstance();
            $app->flash('error', 'Login required');
            $app->redirect('/login');
        }
    };
};
$app = new \Slim\Slim();
$app->get('/foo', $authenticateForRole('admin'), function () {
    //Display admin control panel
});
```

What arguments are passed into each route middleware callable?

Each middleware callable is invoked with one argument, the currently matched `\Slim\Route` object.

```
<?php
$aBitOfInfo = function (\Slim\Route $route) {
    echo "Current route is " . $route->getName();
};

$app->get('/foo', $aBitOfInfo, function () {
    echo "foo";
});
```

Back to Top ↑

Route Groups

Slim lets you group related routes. This is helpful when you find yourself repeating the same URL segments for multiple routes. This is best explained with an example. Let’s pretend we are building an API for books.

```php
<?php
$app = new \Slim\Slim();

// API group
$app->group('/api', function () use ($app) {

    // Library group
    $app->group('/library', function () use ($app) {

        // Get book with ID
        $app->get('/books/:id', function ($id) {

        });

        // Update book with ID
        $app->put('/books/:id', function ($id) {

        });

        // Delete book with ID
        $app->delete('/books/:id', function ($id) {

        });

    });

});
```

The routes defined above would be accessible at, respectively:

```
GET    /api/library/books/:id
PUT    /api/library/books/:id
DELETE /api/library/books/:id
```

Route groups are very useful to group related routes and avoid repeating common URL segments for each route definition.

Back to Top ↑

Route Helpers

Slim provides several helper methods (exposed via the Slim application instance) that will help you control the flow of your application.

Please be aware that the following application instance method helpers `halt()`, `pass()`, `redirect()` and `stop()` are implemented using Exceptions. Each will throw a `\Slim\Exception\Stop` or `\Slim\Exception\Pass` exception. Throwing the Exception in these cases is a simple way to stop user code from processing, have the framework take over, and immediately send the necessary response to the client. This behavior can be surprising if unexpected. Take a look at the following code.

```
<?php
$app->get('/', function() use ($app, $obj) {
    try {
        $obj->thisMightThrowException();
        $app->redirect('/success');
    } catch(\Exception $e) {
        $app->flash('error', $e->getMessage());
        $app->redirect('/error');
    }
});
```

If `$obj->thisMightThrowException()` does throw an Exception the code will run as expected. However, if no exception is thrown the call to $app->redirect() will throw a `\Slim\Exception\Stop` Exception that will be caught by the user `catch` block rather than by the framework redirecting the browser to the “/error” page. Where possible in your own application you should use typed Exceptions so your `catch` blocks are more targeted rather than swallowing all Exceptions. In some situations the `thisMightThrowException()` might be an external component call that you don’t control, in which case typing all exceptions thrown may not be feasible. For these instances we can adjust our code slightly by moving the success `$app->redirect()` after the try/catch block to fix the issues. Since processing will stop on the error redirect this code will now execute as expected.

```
<?php
$app->get('/', function() use ($app, $obj) {
    try {
        $obj->thisMightThrowException();
    } catch(Exception $e) {
        $app->flash('error', $e->getMessage());
        $app->redirect('/error');
    }
    $app->redirect('/success');
});
```

Halt

The Slim application’s `halt()` method will immediately return an HTTP response with a given status code and body. This method accepts two arguments: the HTTP status code and an optional message. Slim will immediately halt the current application and send an HTTP response to the client with the specified status and optional message (as the response body). This will override the existing `\Slim\Http\Response` object.

```
<?php
$app = new \Slim\Slim();

//Send a default 500 error response
$app->halt(500);

//Or if you encounter a Balrog...
$app->halt(403, 'You shall not pass!');
```

If you would like to render a template with a list of error messages, you should use the Slim application's `render()` method instead.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () use ($app) {
    $errorData = array('error' => 'Permission Denied');
    $app->render('errorTemplate.php', $errorData, 403);
});
$app->run();
```

The `halt()` method may send any type of HTTP response to the client: informational, success, redirect, not found, client error, or server error.

Pass

A route can tell the Slim application to continue to the next matching route with the Slim application's `pass()` method. When this method is invoked, the Slim application will immediately stop processing the current matching route and invoke the next matching route. If no subsequent matching route is found, a **404 Not Found** response is sent to the client. Here is an example. Assume an HTTP request for "GET /hello/Frank".

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/Frank', function () use ($app) {
    echo "You won't see this...";
    $app->pass();
});
$app->get('/hello/:name', function ($name) use ($app) {
    echo "But you will see this!";
});
$app->run();
```

Redirect

It is easy to redirect the client to another URL with the Slim application's `redirect()` method. This method accepts two arguments: the first argument is the URL to which the client will redirect; the second optional argument is the HTTP status code. By default the `redirect()` method will send a **302 Temporary Redirect**

response.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () use ($app) {
    $app->redirect('/bar');
});
$app->run();
```

Or if you wish to use a permanent redirect, you must specify the destination URL as the first parameter and the HTTP status code as the second parameter.

```
<?php
$app = new \Slim\Slim();
$app->get('/old', function () use ($app) {
    $app->redirect('/new', 301);
});
$app->run();
```

This method will automatically set the Location: header. The HTTP redirect response will be sent to the HTTP client immediately.

Stop

The Slim application's `stop()` method will stop the Slim application and send the current HTTP response to the client as is. No ifs, ands, or buts.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () use ($app) {
    echo "You will see this...";
    $app->stop();
    echo "But not this";
});
$app->run();
```

URL For

The Slim applications' `urlFor()` method lets you dynamically create URLs for a named route so that, were a route pattern to change, your URLs would update automatically without breaking your application. This example demonstrates how to generate URLs for a named route.

```
<?php
$app = new \Slim\Slim();

//Create a named route
$app->get('/hello/:name', function ($name) use ($app) {
    echo "Hello $name";
})->name('hello');

//Generate a URL for the named route
$url = $app->urlFor('hello', array('name' => 'Josh'));
```

In this example, $url is “/hello/Josh”. To use the `urlFor()` method, you must first assign a name to a route. Next, invoke the `urlFor()` method. The first argument is the name of the route, and the second argument is an associative array used to replace the route’s URL parameters with actual values; the array’s keys must match parameters in the route’s URI and the values will be used as substitutions.

Back to Top ↑

Route URL Rewriting

I strongly encourage you to use a web server that supports URL rewriting; this will let you enjoy clean, human-friendly URLs with your Slim application. To enable URL rewriting, you should use the appropriate tools provided by your web server to forward all HTTP requests to the PHP file in which you instantiate and run your Slim application. The following are sample, bare minimum, configurations for Apache with mod_php and nginx. These are not meant to be production ready configurations but should be enough to get you up and running. To read more on server deployment in general you can continue reading http://www.phptherightway.com (http://www.phptherightway.com).

Apache and mod_rewrite

Here is an example directory structure:

```
/path/www.mysite.com/
    public_html/ <-- Document root!
        .htaccess
        index.php <-- I instantiate Slim here!
    lib/
        Slim/ <-- I store Slim lib files here!
```

The **.htaccess** file in the directory structure above contains:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [QSA,L]
```

You also need a directory directive to enable **.htaccess** files and allow the **RewriteEngine** directive to be used. This is sometimes done globally in the **httpd.conf** file, but its generally a good idea to limit the directive to just your virtual host by enclosing it in your **VirtualHost** configuration block. This is generally setup in your configuration in the form of:

```
<VirtualHost *:80>
    ServerAdmin me@mysite.com
    DocumentRoot "/path/www.mysite.com/public_html"
    ServerName mysite.com
    ServerAlias www.mysite.com

    #ErrorLog "logs/mysite.com-error.log"
    #CustomLog "logs/mysite.com-access.log" combined

    <Directory "/path/www.mysite.com/public_html">
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

As a result, Apache will send all requests for non-existent files to my **index.php** script in which I instantiate and run my Slim application. With URL rewriting enabled and assuming the following Slim application is defined in **index.php**, you can access the application route below at “/foo” rather than “/index.php/foo”.

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () {
    echo "Foo!";
});
$app->run();
```

nginx

We will use the same example directory structure as before, but with nginx our configuration will go into **nginx.conf**.

```
/path/www.mysite.com/
    public_html/ <-- Document root!
        index.php <-- I instantiate Slim here!
    lib/
        Slim/ <-- I store Slim lib files here!
```

Here is a snippet of a **nginx.conf** in which we use the **try_files** directive to serve the file if it exists, good for static files (images, css, js etc), and otherwise forward it on to the **index.php** file.

```
server {
    listen       80;
    server_name  www.mysite.com mysite.com;
    root         /path/www.mysite.com/public_html;

    try_files $uri /index.php;

    # this will only pass index.php to the fastcgi process which is generally safer but
    # assumes the whole site is run via Slim.
    location /index.php {
        fastcgi_connect_timeout 3s;     # default of 60s is just too long
        fastcgi_read_timeout 10s;       # default of 60s is just too long
        include fastcgi_params;
        fastcgi_pass 127.0.0.1:9000;    # assumes you are running php-fpm locally on port 9000
    }
}
```

Most installations will have a default **fastcgi_params** file setup that you can just include as shown above. Some configurations don't include the **SCRIPT_FILENAME** parameter. You must ensure you include this parameter otherwise you might end up with a No input file specified error from the fastcgi process. This can be done directly in the location block or simply added to the **fastcgi_params** file. Either way it looks like this:

```
fastcgi_param  SCRIPT_FILENAME  $document_root$fastcgi_script_name;
```

Without URL Rewriting

Slim will work without URL rewriting. In this scenario, you must include the name of the PHP file in which you instantiate and run the Slim application in the resource URI. For example, assume the following Slim application is defined in **index.php** at the top level of your virtual host's document root:

```
<?php
$app = new \Slim\Slim();
$app->get('/foo', function () {
    echo "Foo!";
});
$app->run();
```

You can access the defined route at "/index.php/foo". If the same application is instead defined in **index.php** inside of the physical subdirectory blog/, you can access the defined route at /blog/index.php/foo.

Back to Top ↑

Environment

cs.slimframework.com/#Environment)

Environment Overview

The Slim Framework implements a derivation of the Rack protocol (http://rack.rubyforge.org/doc/files/SPEC.html). When you instantiate a Slim application, it immediately inspects the `$_SERVER` superglobal and derives a set of environment variables that dictate application behavior.

What is the Environment?

A Slim application's "environment" is an associative array of settings that are parsed once and made accessible to the Slim application and its middleware. You are free to modify the environment variables during runtime; changes will propagate immediately throughout the application.

When you instantiate a Slim application, the environment variables are derived from the `$_SERVER` superglobal; you do not need to set these yourself. However, you are free to modify or supplement these variables in Slim middleware.

These variables are fundamental to determining how your Slim application runs: the resource URI, the HTTP method, the HTTP request body, the URL query parameters, error output, and more. Middleware, described later, gives you the power to - among other things - manipulate environment variables before and/or after the Slim application is run.

Environment Variables

The following text respectfully borrows the same information originally available at http://rack.rubyforge.org/doc/files/SPEC.html (http://rack.rubyforge.org/doc/files/SPEC.html). The environment array must include these variables:

REQUEST_METHOD
: The HTTP request method. This is required and may never be an empty string.

SCRIPT_NAME
: The initial portion of the request URI's "path" that corresponds to the physical directory in which the Slim application is installed — so that the application knows its virtual "location". This may be an empty string if the application is installed in the top-level of the public document root directory. This will never have a trailing slash.

PATH_INFO
: The remaining portion of the request URI's "path" that determines the "virtual" location of the HTTP request's target resource within the Slim application context. This will always have a leading slash; it may or may not have a trailing slash.

QUERY_STRING
: The part of the HTTP request's URI after, but not including, the "?". This is required but may be an empty string.

SERVER_NAME
: When combined with `SCRIPT_NAME` and `PATH_INFO`, this can be used to create a fully qualified URL to an

application resource. However, if `HTTP_HOST` is present, that should be used instead of this. This is required and may never be an empty string.

SERVER_PORT
When combined with `SCRIPT_NAME` and `PATH_INFO`, this can be used to create a fully qualified URL to any application resource. This is required and may never be an empty string.

HTTP_*
Variables matching the HTTP request headers sent by the client. The existence of these variables correspond with those sent in the current HTTP request.

slim.url_scheme
Will be “http” or “https” depending on the HTTP request URL.

slim.input
Will be a string representing the raw HTTP request body. If the HTTP request body is empty (e.g. with a GET request), this will be an empty string.

slim.errors
Must always be a writable resource; by default, this is a write-only resource handle to `php://stderr`.

The Slim application can store its own data in the environment, too. The environment array’s keys must contain at least one dot, and should be prefixed uniquely (e.g. “prefix.foo”). The prefix **slim.** is reserved for use by Slim itself and must not be used otherwise. The environment must not contain the keys `HTTP_CONTENT_TYPE` or `HTTP_CONTENT_LENGTH` (use the versions without HTTP_). The CGI keys (named without a period) must have String values. There are the following restrictions:

- slim.url_scheme must either be “http” or “https”.
- `slim.input` must be a string.
- There must be a valid, writable resource in `slim.errors`.
- The `REQUEST_METHOD` must be a valid token.
- The `SCRIPT_NAME`, if non-empty, must start with “/”
- The `PATH_INFO`, if non-empty, must start with “/”
- The `CONTENT_LENGTH`, if given, must consist of digits only.
- One of `SCRIPT_NAME` or `PATH_INFO` must be set. `PATH_INFO` should be “/” if `SCRIPT_NAME` is empty. `SCRIPT_NAME` never should be “/”, but instead be an empty string.

Back to Top ↑

Request

://docs.slimframework.com/#Request)

Request Overview

Each Slim application instance has one request object. The request object is an abstraction of the current HTTP request and allows you to easily interact with the Slim application's environment variables. Although each Slim application includes a default request object, the `\Slim\Http\Request` class is idempotent; you may instantiate the class at will (in middleware or elsewhere in your Slim application) without affecting the application as a whole. You can obtain a reference to the Slim application's request object like this:

```
<?php
// Returns instance of \Slim\Http\Request
$request = $app->request;
```

Back to Top ↑

Request Method

Every HTTP request has a method (e.g. GET or POST). You can obtain the current HTTP request method via the Slim application's request object:

```
/**
 * What is the request method?
 * @return string (e.g. GET, POST, PUT, DELETE)
 */
$app->request->getMethod();

/**
 * Is this a GET request?
 * @return bool
 */
$app->request->isGet();

/**
 * Is this a POST request?
 * @return bool
 */
$app->request->isPost();

/**
 * Is this a PUT request?
 * @return bool
 */
$app->request->isPut();

/**
 * Is this a DELETE request?
 * @return bool
 */
$app->request->isDelete();
```

```
/**
 * Is this a HEAD request?
 * @return bool
 */
$app->request->isHead();

/**
 * Is this a OPTIONS request?
 * @return bool
 */
$app->request->isOptions();

/**
 * Is this a PATCH request?
 * @return bool
 */
$app->request->isPatch();

/**
 * Is this a XHR/AJAX request?
 * @return bool
 */
$app->request->isAjax();
```

Back to Top ↑

Request Headers

A Slim application will automatically parse all HTTP request headers. You can access the request headers using the request object's public `headers` property. The `headers` property is an instance of `\Slim\Helper\Set`, meaning it provides a simple, standardized interface to interactive with the HTTP request headers.

```
<?php
$app = new \Slim\Slim();

// Get request headers as associative array
$headers = $app->request->headers;

// Get the ACCEPT_CHARSET header
$charset = $app->request->headers->get('ACCEPT_CHARSET');
```

The HTTP specification states that HTTP header names may be uppercase, lowercase, or mixed-case. Slim is smart enough to parse and return header values whether you request a header value using upper, lower, or mixed case header name, with either underscores or dashes. So use the naming convention with which

you are most comfortable.

Back to Top ↑

Request Body

Use the request object's `getBody()` method to fetch the raw HTTP request body sent by the HTTP client. This is particularly useful for Slim application's that consume JSON or XML requests.

```
<?php
$app = new \Slim\Slim();
$body = $app->request->getBody();
```

Back to Top ↑

Request Variables

An HTTP request may have associated variables (not to be confused with route variables). The GET, POST, or PUT variables sent with the current HTTP request are exposed via the Slim application's request object.

If you want to quickly fetch a request variable value without considering its type, use the request object's `params()` method:

```
<?php
$app = new \Slim\Slim();
$paramValue = $app->request->params('paramName');
```

The `params()` method will first search PUT variables, then POST variables, then GET variables. If no variables are found, `null` is returned. If you only want to search for a specific type of variable, you can use these methods instead:

```
<?php
//GET variable
$paramValue = $app->request->get('paramName');

//POST variable
$paramValue = $app->request->post('paramName');

//PUT variable
$paramValue = $app->request->put('paramName');
```

If a variable does not exist, each method above will return `null`. You can also invoke any of these functions without an argument to obtain an array of all variables of the given type:

```
<?php
$allGetVars = $app->request->get();
$allPostVars = $app->request->post();
$allPutVars = $app->request->put();
```

Back to Top ↑

Request Cookies

Get Cookies

A Slim application will automatically parse cookies sent with the current HTTP request. You can fetch cookie values with the Slim application's `getCookie()` helper method like this:

```
<?php
$app = new \Slim\Slim();
$foo = $app->getCookie('foo');
```

Only Cookies sent with the current HTTP request are accessible with this method. If you set a cookie during the current request, it will not be accessible with this method until the subsequent request. Bear in mind the \Slim\Slim object's `getCookie()` method is a convenience. You may also retrieve the complete set of HTTP request cookies directly from the \Slim\Http\Request object like this:

```
<?php
$cookies = $app->request->cookies;
```

This will return an instance of \Slim\Helper\Set so you can use its simple, standardized interface to inspect the request's cookies.

Cookie Encryption

You can optionally choose to encrypt all cookies stored on the HTTP client with the Slim app's `cookies.encrypt` setting. When this setting is `true`, all cookies will be encrypted using your designated secret key and cipher.

It's really that easy. Cookies will be encrypted automatically before they are sent to the client. They will also be decrypted on-demand when you request them with `\Slim\Slim::getCookie()` during subsequent requests.

Back to Top ↑

Request Paths

Every HTTP request received by a Slim application will have a root URI and a resource URI.

Root URI

The **root URI** is the physical URL path of the directory in which the Slim application is instantiated and run. If a Slim application is instantiated in **index.php** within the top-most directory of the virtual host's document root, the root URI will be an empty string. If a Slim application is instantiated and run in **index.php** within a physical subdirectory of the virtual host's document root, the root URI will be the path to that subdirectory with a leading slash and without a trailing slash.

Resource URI

The **resource URI** is the virtual URI path of an application resource. The resource URI will be matched to the Slim application's routes.

Assume the Slim application is installed in a physical subdirectory **/foo** beneath your virtual host's document root. Also assume the full HTTP request URL (what you'd see in the browser location bar) is **/foo/books/1**. The root URI is **/foo** (the path to the physical directory in which the Slim application is instantiated) and the resource URI is **/books/1** (the path to the application resource).

You can get the HTTP request's root URI and resource URI with the request object's `getRootUri()` and `getResourceUri()` methods:

```
<?php
$app = new \Slim\Slim();

// Get request object
$req = $app->request;

//Get root URI
$rootUri = $req->getRootUri();

//Get resource URI
$resourceUri = $req->getResourceUri();
```

Back to Top ↑

XMLHttpRequest

When using a Javascript framework like MooTools or jQuery to execute an XMLHttpRequest, the XMLHttpRequest will usually be sent with a **X-Requested-With** HTTP header. The Slim application will detect the HTTP request's **X-Requested-With** header and flag the request as such. If for some reason an XMLHttpRequest cannot be sent with the **X-Requested-With** HTTP header, you can force the Slim application to assume an HTTP request is an XMLHttpRequest by setting a GET, POST, or PUT parameter in the HTTP request named "isajax" with a truthy value.

Use the request object's `isAjax()` or `isXhr()` method to tell if the current request is an XHR/Ajax request:

```php
<?php
$isXHR = $app->request->isAjax();
$isXHR = $app->request->isXhr();
```

Back to Top ↑

Request Helpers

The Slim application's request object provides several helper methods to fetch common HTTP request information:

Content Type

Fetch the request's content type (e.g. "application/json;charset=utf-8"):

```php
<?php
$req = $app->request;
$req->getContentType();
```

Media Type

Fetch the request's media type (e.g. "application/json"):

```php
<?php
$req = $app->request;
$req->getMediaType();
```

Media Type Params

Fetch the request's media type parameters (e.g. [charset => "utf-8"]):

```php
<?php
$req = $app->request;
$req->getMediaTypeParams();
```

Content Charset

Fetch the request's content character set (e.g. "utf-8"):

```php
<?php
$req = $app->request;
$req->getContentCharset();
```

Content Length

Fetch the request's content length:

```
<?php
$req = $app->request;
$req->getContentLength();
```

Host

Fetch the request's host (e.g. "slimframework.com"):

```
<?php
$req = $app->request;
$req->getHost();
```

Host with Port

Fetch the request's host with port (e.g. "slimframework.com:80"):

```
<?php
$req = $app->request;
$req->getHostWithPort();
```

Port

Fetch the request's port (e.g. 80):

```
<?php
$req = $app->request;
$req->getPort();
```

Scheme

Fetch the request's scheme (e.g. "http" or "https"):

```
<?php
$req = $app->request;
$req->getScheme();
```

Path

Fetch the request's path (root URI + resource URI):

```php
<?php
$req = $app->request;
$req->getPath();
```

URL

Fetch the request's URL (scheme + host [+ port if non-standard]):

```php
<?php
$req = $app->request;
$req->getUrl();
```

IP Address

Fetch the request's IP address:

```php
<?php
$req = $app->request;
$req->getIp();
```

Referer

Fetch the request's referrer:

```php
<?php
$req = $app->request;
$req->getReferrer();
```

User Agent

Fetch the request's user agent string:

```php
<?php
$req = $app->request;
$req->getUserAgent();
```

Back to Top ↑

Response

/docs.slimframework.com/#Response)

Response Overview

Each Slim application instance has one response object. The response object is an abstraction of your Slim application's HTTP response that is returned to the HTTP client. Although each Slim application includes a default response object, the `\Slim\Http\Response` class is idempotent; you may instantiate the class at will (in middleware or elsewhere in your Slim application) without affecting the application as a whole. You can obtain a reference to the Slim application's response object with:

```
<?php
$app = new \Slim\Slim();
$app->response;
```

An HTTP response has three primary properties:

- Status
- Header
- Body

The response object provides helper methods, described next, that help you interact with these HTTP response properties. The default response object will return a **200 OK** HTTP response with the **text/html** content type.

Back to Top ↑

Response Status

The HTTP response returned to the client will have a status code indicating the response's type (e.g. 200 OK, 400 Bad Request, or 500 Server Error). You can use the Slim application's response object to set the HTTP response's status like this:

```
<?php
$app->response->setStatus(400);
```

You only need to set the response object's status if you intend to return an HTTP response that *does not* have a 200 OK status. You can just as easily fetch the response object's current HTTP status by invoking the same method without an argument, like this:

```
<?php
$status = $app->response->getStatus();
```

Back to Top ↑

Response Headers

The HTTP response returned to the HTTP client will have a header. The HTTP header is a list of keys and values that provide metadata about the HTTP response. You can use the Slim application's response object to set the HTTP response's header. The response object has a public property `headers` that is an instance of `\Slim\Helper\Set`; this provides a simple, standardized interface to manipulate the HTTP response headers.

```
<?php
$app = new \Slim\Slim();
$app->response->headers->set('Content-Type', 'application/json');
```

You may also fetch headers from the response object's `headers` property, too:

```
<?php
$contentType = $app->response->headers->get('Content-Type');
```

If a header with the given name does not exist, `null` is returned. You may specify header names with upper, lower, or mixed case with dashes or underscores. Use the naming convention with which you are most comfortable.

Back to Top ↑

Response Body

The HTTP response returned to the client will have a body. The HTTP body is the actual content of the HTTP response delivered to the client. You can use the Slim application's response object to set the HTTP response's body:

```
<?php
$app = new \Slim\Slim();

// Overwrite response body
$app->response->setBody('Foo');

// Append response body
$app->response->write('Bar');
```

When you overwrite or append the response object's body, the response object will automatically set the **Content-Length** header based on the bytesize of the new response body.

You can fetch the response object's body like this:

```
<?php
$body = $app->response->getBody();
```

Usually, you will never need to manually set the response body with the `setBody()` or `write()` methods; instead, the Slim app will do this for you. Whenever you `echo()` content inside a route's callback function, the `echo()`'d content is captured in an output buffer and appended to the response body before the HTTP response is returned to the client.

Back to Top ↑

Response Cookies

The Slim application provides helper methods to send cookies with the HTTP response.

Set Cookie

This example demonstrates how to use the Slim application's `setCookie()` method to create an HTTP cookie to be sent with the HTTP response:

```
<?php
$app->setCookie('foo', 'bar', '2 days');
```

This creates an HTTP cookie with the name "foo" and value "bar" that expires two days from now. You may also provide additional cookie properties, including its path, domain, secure, and httponly settings. The Slim application's `setCookie()` method uses the same signature as PHP's native `setCookie()` function.

```
<?php
$app->setCookie(
    $name,
    $value,
    $expiresAt,
    $path,
    $domain,
    $secure,
    $httponly
);
```

Set Encrypted Cookie

You can tell Slim to encrypt the response cookies by setting the app's `cookies.encrypt` setting to `true`. When this setting is `true`, Slim will encrypt the response cookies automatically before they are returned to the HTTP client.

Here are the available Slim app settings used for cookie encryption:

```
<?php
$app = new \Slim\Slim(array(
    'cookies.encrypt' => true,
    'cookies.secret_key' => 'my_secret_key',
    'cookies.cipher' => MCRYPT_RIJNDAEL_256,
    'cookies.cipher_mode' => MCRYPT_MODE_CBC
));
```

Delete Cookie

You can delete a cookie using the Slim application's `deleteCookie()` method. This will remove the cookie from the HTTP client before the next HTTP request. This method accepts the same signature as the Slim application's `setCookie()` instance method, *without* the `$expires` argument. Only the first argument is required.

```
<?php
$app->deleteCookie('foo');
```

If you need to also specify the path and domain:

```
<?php
$app->deleteCookie('foo', '/', 'foo.com');
```

You may also further specify the secure and httponly properties:

```
<?php
$app->deleteCookie('foo', '/', 'foo.com', true, true);
```

Back to Top ↑

Response Helpers

The response object provides helper methods to inspect and interact with the underlying HTTP response.

Finalize

The response object's `finalize()` method returns a numeric array of `[status, header, body]`. The status is an integer; the header is an iterable data structure; and the body is a string. Were you to create a new `\Slim\Http\Response` object in your Slim application or its middleware, you would call the response object's `finalize()` method to produce the status, header, and body for the underlying HTTP response.

```php
<?php
/**
 * Prepare new response object
 */
$res = new \Slim\Http\Response();
$res->setStatus(400);
$res->write('You made a bad request');
$res->headers->set('Content-Type', 'text/plain');

/**
 * Finalize
 * @return [
 *     200,
 *     ['Content-type' => 'text/plain'],
 *     'You made a bad request'
 * ]
 */
$array = $res->finalize();
```

Redirect

The response object's `redirect()` method will set the response status and its **Location:** header needed to return a **3xx Redirect** response.

```php
<?php
$app->response->redirect('/foo', 303);
```

Status Introspection

The response object provides other helper methods to inspect its current status. All of the following methods return a boolean value:

```php
<?php
$res = $app->response;

//Is this an informational response?
$res->isInformational();

//Is this a 200 OK response?
$res->isOk();

//Is this a 2xx successful response?
$res->isSuccessful();

//Is this a 3xx redirection response?
$res->isRedirection();

//Is this a specific redirect response? (301, 302, 303, 307)
$res->isRedirect();

//Is this a forbidden response?
$res->isForbidden();

//Is this a not found response?
$res->isNotFound();

//Is this a client error response?
$res->isClientError();

//Is this a server error response?
$res->isServerError();
```

Back to Top ↑

View

http://docs.slimframework.com/#View)

View Overview

A Slim application delegates rendering of templates to its view object. A Slim application view is a subclass of `\Slim\View` that implements this interface:

```
<?php
public render(string $template);
```

The view object's `render` method must return the rendered content of the template specified by its `$template` argument.

Back to Top ↑

Rendering

You can use the Slim application's `render()` method to ask the current view object to render a template with a given set of variables. The Slim application's `render()` method will `echo()` the output returned from the view object to be captured by an output buffer and appended automatically to the response object's body. This assumes nothing about how the template is rendered; that is delegated to the view object.

```
<?php
$app = new \Slim\Slim();
$app->get('/books/:id', function ($id) use ($app) {
    $app->render('myTemplate.php', array('id' => $id));
});
```

If you need to pass data from the route callback into the view object, you must explicitly do so by passing an array as the second argument of the Slim application's `render()` method like this:

```
<?php
$app->render(
    'myTemplate.php',
    array( 'name' => 'Josh' )
);
```

You can also set the HTTP response status when you render a template:

```
<?php
$app->render(
    'myTemplate.php',
    array( 'name' => 'Josh' ),
    404
);
```

Back to Top ↑

Custom Views

A Slim application delegates rendering of templates to its view object. A custom view is a subclass of `\Slim\View` that implements this interface:

```
<?php
public render(string $template);
```

The view object's `render` method must return the rendered content of the template specified by its `$template` argument. When the custom view's render method is invoked, it is passed the desired template pathname (relative to the Slim application's "templates.path" setting) as its argument. Here's an example custom view:

```
<?php
class CustomView extends \Slim\View
{
    public function render($template)
    {
        return 'The final rendered template';
    }
}
```

The custom view can do whatever it wants internally so long as it returns the template's rendered output as a string. A custom view makes it easy to integrate popular PHP template systems like Twig or Smarty.

Heads Up! A custom view may access data passed to it by the Slim application's `render()` method with `$this->data`.

You can browse ready-to-use custom views that work with popular PHP template engines in the Slim-Extras repository on GitHub.

Example View

```
<?php
class CustomView extends \Slim\View
{
    public function render($template)
    {
        // $template === 'show.php'
        // $this->data['title'] === 'Sahara'
    }
}
```

Example Integration

If the custom view is not discoverable by a registered autoloader, it must be required before the Slim application is instantiated.

```
<?php
require 'CustomView.php';

$app = new \Slim\Slim(array(
    'view' => new CustomView()
));

$app->get('/books/:id', function ($id) use ($app) {
    $app->render('show.php', array('title' => 'Sahara'));
});

$app->run();
```

Back to Top ↑

View Data

Heads Up! Rarely will you set or append data directly on the view object. Usually, you pass data to the view with the Slim application's `render()` method. See Rendering Templates (/pages/view-rendering-templates).

The view object's `setData()` and `appendData()` methods inject data into the view object; the injected data is available to view templates. View data is stored internally as a key-value array.

Setting Data

The view object's `setData()` instance method will overwrite existing view data. You may use this method to set a single variable to a given value:

```
<?php
$app->view->setData('color', 'red');
```

The view's data will now contain a key "color" with value "red". You may also use the view's `setData()` method to batch assign an entire array of data:

```
<?php
$app->view->setData(array(
    'color' => 'red',
    'size' => 'medium'
));
```

Remember, the view's `setData()` method will replace all previous data.

Appending Data

The view object also has a `appendData()` method that appends data to the view's existing data. This method accepts an array as its one and only argument:

```
<?php
$app->view->appendData(array(
    'foo' => 'bar'
));
```

Back to Top ↑

HTTP Caching

ttp://docs.slimframework.com/#HTTP-aching)

HTTP Caching Overview

A Slim application provides built-in support for HTTP caching with its `etag()`, `lastModified()`, and `expires()` helper methods. It is best to use one of `etag()` or `lastModified()` - in conjunction with `expires()` - per route; never use both `etag()` and `lastModified()` together in the same route callback.

The `etag()` and `lastModified()` methods should be invoked in a route callback before other code; this allows Slim to check conditional GET requests before processing the route callback's remaining code.

Both `etag()` and `lastModified()` instruct the HTTP client to store the resource response in a client-side cache. The `expires()` method indicates to the HTTP client when the client-side cache should be considered stale.

Back to Top ↑

ETag

A Slim application provides built-in support for HTTP caching using ETags. An ETag is a unique identifier for a resource URI. When an ETag header is set with the Slim application's `etag()` method, the HTTP client will send an **If-None-Match** header with each subsequent HTTP request of the same resource URI. If the ETag value for the resource URI matches the **If-None-Match** HTTP request header, the Slim application will return a **304 Not Modified** HTTP response that will prompt the HTTP client to continue using its cache; this also prevents the Slim application from serving the entire markup for the resource URI, saving bandwidth and response time.

Setting an ETag with Slim is very simple. Invoke the Slim application's `etag()` method in your route callback, passing it a unique ID as the first and only argument.

```
<?php
$app->get('/foo', function () use ($app) {
    $app->etag('unique-id');
    echo "This will be cached after the initial request!";
});
```

That's it. Make sure the ETag ID is unique for the given resource. Also make sure the ETag ID changes as your resource changes; otherwise, the HTTP client will continue serving its outdated cache.

Back to Top ↑

Last Modified

A Slim application provides built-in support for HTTP caching using the resource's last modified date. When you specify a last modified date, Slim tells the HTTP client the date and time the current resource was last modified. The HTTP client will then send a **If-Modified-Since** header with each subsequent HTTP request for the given resource URI. If the last modification date you specify matches the **If-Modified-Since** HTTP request header, the Slim application will return a **304 Not Modified** HTTP response that will prompt the HTTP client to use its cache; this also prevents the Slim application from serving the entire markup for the resource URI saving bandwidth and response time.

Setting a last modified date with Slim is very simple. You only need to invoke the Slim application's `lastModified()` method in your route callback passing in a UNIX timestamp of the last modification date for the given resource. Be sure the `lastModified()` method's timestamp updates along with the resource's last modification date; otherwise, the browser client will continue serving its outdated cache.

```
<?php
$app->get('/foo', function () use ($app) {
    $app->lastModified(1286139652);
    echo "This will be cached after the initial request!";
});
```

Back to Top ↑

Expires

Used in conjunction with the Slim application's `etag()` or `lastModified()` methods, the `expires()` method sets an **Expires** header on the HTTP response informing the HTTP client when its client-side cache for the current resource should be considered stale. The HTTP client will continue serving from its client-side cache until the expiration date is reached, at which time the HTTP client will send a conditional GET request to the Slim application.

The `expires()` method accepts one argument: an integer UNIX timestamp, or a string to be parsed with `strtotime()`.

```
<?php
$app->get('/foo', function () use ($app) {
    $app->etag('unique-resource-id');
    $app->expires('+1 week');
    echo "This will be cached client-side for one week";
});
```

Back to Top ↑

Middleware

ocs.slimframework.com/#Middleware)

Middleware Overview

The Slim Framework implements a version of the Rack protocol. As a result, a Slim application can have middleware that may inspect, analyze, or modify the application environment, request, and response before and/or after the Slim application is invoked.

Middleware Architecture

Think of a Slim application as the core of an onion. Each layer of the onion is middleware. When you invoke the Slim application's `run()` method, the outer-most middleware layer is invoked first. When ready, that middleware layer is responsible for optionally invoking the next middleware layer that it surrounds. This process steps deeper into the onion - through each middleware layer - until the core Slim application is invoked. This stepped process is possible because each middleware layer, and the Slim application itself, all implement a public `call()` method. When you add new middleware to a Slim application, the added middleware will become a new outer layer and surround the previous outer middleware layer (if available) or the Slim application itself.

Application Reference

The purpose of middleware is to inspect, analyze, or modify the application environment, request, and response before and/or after the Slim application is invoked. It is easy for each middleware to obtain references to the primary Slim application, its environment, its request, and its response:

```php
<?php
class MyMiddleware extends \Slim\Middleware
{
    public function call()
    {
        //The Slim application
        $app = $this->app;

        //The Environment object
        $env = $app->environment;

        //The Request object
        $req = $app->request;

        //The Response object
        $res = $app->response;
    }
}
```

Changes made to the environment, request, and response objects will propagate immediately throughout the application and its other middleware layers. This is possible because every middleware layer is given a reference to the same Slim application object.

Next Middleware Reference

Each middleware layer also has a reference to the next inner middleware layer with `$this->next`. It is each middleware's responsibility to optionally call the next middleware. Doing so will allow the Slim application to complete its full lifecycle. If a middleware layer chooses **not** to call the next inner middleware layer, further inner middleware and the Slim application itself will not be run, and the application response will be returned to the HTTP client as is.

```php
<?php
class MyMiddleware extends \Slim\Middleware
{
    public function call()
    {
        //Optionally call the next middleware
        $this->next->call();
    }
}
```

Back to Top ↑

How to Use Middleware

Use the Slim application's `add()` instance method to add new middleware to a Slim application. New middleware will surround previously added middleware, or the Slim application itself if no middleware has yet been added.

Example Middleware

This example middleware will capitalize the Slim application's HTTP response body.

```
<?php
class AllCapsMiddleware extends \Slim\Middleware
{
    public function call()
    {
        // Get reference to application
        $app = $this->app;

        // Run inner middleware and application
        $this->next->call();

        // Capitalize response body
        $res = $app->response;
        $body = $res->getBody();
        $res->setBody(strtoupper($body));
    }
}
```

Add Middleware

```
<?php
$app = new \Slim\Slim();
$app->add(new \AllCapsMiddleware());
$app->get('/foo', function () use ($app) {
    echo "Hello";
});
$app->run();
```

The Slim application's `add()` method accepts one argument: a middleware instance. If the middleware instance requires special configuration, it may implement its own constructor so that it may be configured before it is added to the Slim application.

When the example Slim application above is run, the HTTP response body will be an enthusiastic "HELLO";

Back to Top ↑

How to Write Middleware

Slim application middleware must subclass `\Slim\Middleware` and implement a public `call()` method. The `call()` method does not accept arguments. Middleware may implement its own constructor, properties, and methods. I encourage you to look at Slim's built-in middleware for working examples (e.g. Slim/Middleware/ContentTypes.php or Slim/Middleware/SessionCookie.php).

This example is the most simple implementation of Slim application middleware. It extends `\Slim\Middleware`, implements a public `call()` method, and calls the next inner middleware.

```
<?php
class MyMiddleware extends \Slim\Middleware
{
    public function call()
    {
        $this->next->call();
    }
}
```

Back to Top ↑

Hooks

tp://docs.slimframework.com/#Hooks)

Hooks Overview

A Slim application provides a set of hooks to which you can register your own callbacks.

What is a hook?

A "hook" is a moment in the Slim application lifecycle at which a priority list of callables assigned to the hook will be invoked. A hook is identified by a string name.

A "callable" is anything that returns `true` for `is_callable()`. A callable is assigned to a hook and is invoked when the hook is called. If multiple callables are assigned to a single hook, each callable is invoked in the order assigned.

Back to Top ↑

How to Use Hooks

A callable is assigned to a hook using the Slim application's `hook()` method:

```
<?php
$app = new \Slim\Slim();
$app->hook('the.hook.name', function () {
    //Do something
});
```

The first argument is the hook name, and the second argument is the callable. Each hook maintains a priority list of registered callables. By default, each callable assigned to a hook is given a priority of 10. You can give your callable a different priority by passing an integer as the third parameter of the `hook()` method:

```
<?php
$app = new \Slim\Slim();
$app->hook('the.hook.name', function () {
    //Do something
}, 5);
```

The example above assigns a priority of 5 to the callable. When the hook is called, it will sort all callables assigned to it by priority (ascending). A callable with priority 1 will be invoked before a callable with priority 10.

Hooks do not pass arguments to their callables. If a callable needs to access the Slim application, you can inject the application into the callback with the `use` keyword or with the Slim application's static `getInstance()` method:

```
<?php
$app = new \Slim\Slim();
$app->hook('the.hook.name', function () use ($app) {
    // Do something
});
```

Back to Top ↑

Default Hooks

These are the default hooks always invoked in a Slim application.

slim.before
: This hook is invoked before the Slim application is run and before output buffering is turned on. This hook is invoked once during the Slim application lifecycle.

slim.before.router
: This hook is invoked after output buffering is turned on and before the router is dispatched. This hook is invoked once during the Slim application lifecycle.

slim.before.dispatch
: This hook is invoked before the current matching route is dispatched. Usually this hook is invoked only once during the Slim application lifecycle; however, this hook may be invoked multiple times if a matching route chooses to pass to a subsequent matching route.

slim.after.dispatch

This hook is invoked after the current matching route is dispatched. Usually this hook is invoked only once during the Slim application lifecycle; however, this hook may be invoked multiple times if a matching route chooses to pass to a subsequent matching route.

slim.after.router

This hook is invoked after the router is dispatched, before the Response is sent to the client, and after output buffering is turned off. This hook is invoked once during the Slim application lifecycle.

slim.after

This hook is invoked after output buffering is turned off and after the Response is sent to the client. This hook is invoked once during the Slim application lifecycle.

Back to Top ↑

Custom Hooks

Custom hooks may be created and invoked in a Slim application. When a custom hook is invoked with `applyHook()`, it will invoke all callables assigned to that hook. This is exactly how the Slim application's default hooks work. In this example, I apply a custom hook called "my.hook.name". All callables previously registered for this hook will be invoked.

```
<?php
$app = new \Slim\Slim();
$app->applyHook('my.hook.name');
```

When you run the above code, any callables previously assigned to the hook "my.hook.name" will be invoked in order of priority (ascending).

You should register callables to a hook before the hook is applied. Think of it this way: when you invoke the Slim application's `applyHook()` method, you are asking Slim to invoke all callables already registered for that hook name.

Back to Top ↑

Flash Messages

ttp://docs.slimframework.com/#Flash-essages)

Flash Messaging Overview

Heads Up! Flash messages require sessions. If you do not use the

`\Slim\Middleware\SessionCookie` middleware, you must start a native PHP session yourself.

Slim supports flash messaging much like Rails and other larger web frameworks. Flash messaging allows you to define messages that will persist until the next HTTP request but no further. This is helpful to display messages to the user after a given event or error occurs.

As shown below, the Slim application's `flash()` and `flashNow()` methods accept two arguments: a key and a message. The key may be whatever you want and defines how the message will be accessed in the view templates. For example, if I invoke the Slim application's `flash('foo', 'The foo message')` method with those arguments, I can access that message in the next request's templates with `flash['foo']`.

Flash messages are persisted with sessions; sessions are required for flash messages to work. Flash messages are stored in `$_SESSION['slim.flash']`.

Back to Top ↑

Flash Next

The Slim application's `flash()` method sets a message that will be available in the next request's view templates. The message in this example will be available in the template variable `flash['error']`.

```
<?php
$app->flash('error', 'User email is required');
```

Back to Top ↑

Flash Now

The Slim application's `flashNow()` method sets a message that will be available in the current request's view templates. Messages set with the `flashNow()` application instance method will not be available in the next request. The message in the example below will be available in the template variable `flash['info']`.

```
<?php
$app->flashNow('info', 'Your credit card is expired');
```

Back to Top ↑

Flash Keep

This method tells the Slim application to keep existing flash messages set in the previous request so they will be available to the next request. This method is helpful for persisting flash messages across HTTP redirects.

```
<?php
$app->flashKeep();
```

Back to Top ↑

Sessions

//docs.slimframework.com/#Sessions)

Native Session Store

A Slim application does not presume anything about sessions. If you prefer to use a PHP session, you must configure and start a native PHP session with `session_start()` before you instantiate the Slim application.

You should also disable PHP's session cache limiter so that PHP does not send conflicting cache expiration headers with the HTTP response. You can disable PHP's session cache limiter with:

```
<?php
session_cache_limiter(false);
session_start();
```

Back to Top ↑

Cookie Session Store

You may also use the `\Slim\Middleware\SessionCookie` middleware to persist session data in encrypted, hashed HTTP cookies. To enable the session cookie middleware, add the `\Slim\Middleware\SessionCookie` middleware to your Slim application:

```
<?php
$app = new Slim();
$app->add(new \Slim\Middleware\SessionCookie(array(
    'expires' => '20 minutes',
    'path' => '/',
    'domain' => null,
    'secure' => false,
    'httponly' => false,
    'name' => 'slim_session',
    'secret' => 'CHANGE_ME',
    'cipher' => MCRYPT_RIJNDAEL_256,
    'cipher_mode' => MCRYPT_MODE_CBC
)));
```

The second argument is optional; it is shown here so you can see the default middleware settings. The session cookie middleware will work seamlessly with the `$_SESSION` superglobal so you can easily migrate to this session storage middleware with zero changes to your application code.

If you use the session cookie middleware, you DO NOT need to start a native PHP session. The `$_SESSION` superglobal will still be available, and it will be persisted into an HTTP cookie via the middleware layer rather than with PHP's native session management.

Remember, HTTP cookies are inherently limited to only 4 kilobytes of data. If your encrypted session data will exceed this length, you should instead rely on PHP's native sessions or an alternate session store.

> **PLEASE NOTE:** Client-side storage of session data is not recommended if you are dealing with sensitive information, even when using Slim's encrypted session cookie middleware. If you need to store sensitive information, you should encrypt and store the session information on your server.

Back to Top ↑

Logging

://docs.slimframework.com/#Logging)

Logging Overview

A Slim application provides a log object that writes data to a specific output. The actual writing of data is delegated to a log writer.

How to log data

To log data in a Slim application, get a reference to the log object:

```
<?php
$log = $app->log;
```

The log object provides the following PSR-3 interface

```
$app->log->debug(mixed $object);
$app->log->info(mixed $object);
$app->log->notice(mixed $object);
$app->log->warning(mixed $object);
$app->log->error(mixed $object);
$app->log->critical(mixed $object);
$app->log->alert(mixed $object);
$app->log->emergency(mixed $object);
```

Each log object method accepts one mixed argument. The argument is usually a string, but the argument can be anything. The log object will pass the argument to its log writer. It is the log writer's responsibility to write arbitrary input to the appropriate destination.

Back to Top ↑

Activate Logging

The Slim application's log object provides the following public methods to enable or disable logging during runtime.

```
<?php
//Enable logging
$app->log->setEnabled(true);

//Disable logging
$app->log->setEnabled(false);
```

You may enable or disable the log object during application instantiation like this:

```
<?php
$app = new Slim(array(
    'log.enabled' => true
));
```

If logging is disabled, the log object will ignore all logged messages until it is enabled.

Back to Top ↑

Log Levels

Heads Up! Use the `\Slim\Log` constants when setting the log level instead of using raw integers.

The Slim application's log object will respect or ignore logged messages based on its log level setting. When you invoke the log objects's methods, you are inherently assigning a level to the logged message. The available log levels are:

\Slim\Log::EMERGENCY
  Level 1
\Slim\Log::ALERT
  Level 2
\Slim\Log::CRITICAL
  Level 3
\Slim\Log::ERROR
  Level 4
\Slim\Log::WARN
  Level 5
\Slim\Log::NOTICE
  Level 6
\Slim\Log::INFO
  Level 7
\Slim\Log::DEBUG
  Level 8

Only messages that have a level less than the current log object's level will be logged. For example, if the log object's level is `\Slim\Log::WARN` (5), the log object will ignore `\Slim\Log::DEBUG` and `\Slim\Log::INFO` messages but will accept `\Slim\Log::WARN`, `\Slim\Log::ERROR`, and `\Slim\Log::CRITICAL` messages.

How to set the log level

```
<?php
$app->log->setLevel(\Slim\Log::WARN);
```

You can set the log object's level during application instantiation, too:

```
<?php
$app = new \Slim\Slim(array(
    'log.level' => \Slim\Log::WARN
));
```

Back to Top ↑

Log Writers

The Slim application's log object has a log writer. The log writer is responsible for sending a logged message to the appropriate output (e.g. STDERR, a log file, a remote web service, Twitter, or a database). Out of the box, the Slim application's log object has a log writer of class `\Slim\LogFileWriter`; this log writer directs output to the resource handle referenced by the application environment's **slim.errors** key (by default, this is "php://stderr"). You may also define and use a custom log writer.

How to use a custom log writer

A custom log writer must implement the following public interface:

```
<?php
public function write(mixed $message);
```

You must tell the Slim application's log object to use your custom log writer. You can do so in your application's settings during instantiation like this:

```
<?php
$app = new \Slim\Slim(array(
    'log.writer' => new MyLogWriter()
));
```

You may also set a custom log writer with middleware like this:

```
<?php
class CustomLogWriterMiddleware extends \Slim\Middleware
{
    public function call()
    {
        //Set the new log writer
        $this->app->log->setWriter(new \MyLogWriter());

        //Call next middleware
        $this->next->call();
    }
}
```

You can set the log writer similarly in an application hook or route callback like this:

```
<?php
$app->hook('slim.before', function () use ($app) {
    $app->log->setWriter(new \MyLogWriter());
});
```

If you only need to redirect error output to a different resource handle, use the Slim application's default log writer; it writes log messages to a resource handle. All you need to do is set the **slim.errors** environment variable to a valid resource handle.

Back to Top ↑

Error Handling

ttp://docs.slimframework.com/#Error-andling)

Error Handling Overview

Let's face it: sometimes things go wrong. It is important to intercept errors and respond to them appropriately. A Slim application provides helper methods to respond to errors and exceptions.

Important Notes

- A Slim application respects your existing `error_reporting` setting;
- A Slim application only handles errors and exceptions generated inside the Slim application;
- A Slim application converts errors into `ErrorException` objects and throws them;
- A Slim application uses its built-in error handler if its `debug` setting is true; otherwise, it uses the custom error handler.

Back to Top ↑

Error Handler

You may use the Slim application's `error()` method to specify a custom error handler to be invoked when an error or exception occurs. Custom error handlers are only invoked if application debugging is disabled.

A custom error handler should render a user-friendly message that mitigates user confusion. Similar to the Slim application's `notFound()` method, the `error()` method acts as both a getter and a setter.

Set custom error handler

You may set a custom error handler by passing a callable into the Slim application's `error()` method as its first and only argument.

```
<?php
$app = new \Slim\Slim();
$app->error(function (\Exception $e) use ($app) {
    $app->render('error.php');
});
```

In this example, the custom error handler accepts the caught Exception as its argument. This allows you to respond appropriately to different exceptions.

Invoke custom error handler

Usually, the Slim application will automatically invoke the error handler when an exception or error occurs. However, you may also manually invoke the error handler with the Slim application's `error()` method (without an argument).

Back to Top ↑

Not Found Handler

It is an inevitability that someone will request a page that does not exist. The Slim application lets you easily define a custom Not Found handler with the Slim application's `notFound()` method. The Not Found handler will be invoked when a matching route is not found for the current HTTP request. This method acts as both a getter and a setter.

Set not found handler

If you invoke the Slim application's `notFound()` method and specify a callable object as its first and only argument, this method will register the callable object as the Not Found handler. However, the registered handler will not be invoked.

```
<?php
$app = new \Slim\Slim();
$app->notFound(function () use ($app) {
    $app->render('404.html');
});
```

Invoke not found handler

If you invoke the Slim application's `notFound()` method without any arguments, this method will invoke the previously registered Not Found handler.

```
<?php
$app = new \Slim\Slim();
$app->get('/hello/:name', function ($name) use ($app) {
    if ( $name === 'Waldo' ) {
        $app->notFound();
    } else {
        echo "Hello, $name";
    }
});
```

Back to Top ↑

Debugging

You can enable debugging during application instantiation with this setting:

```
<?php
$app = new \Slim\Slim(array(
    'debug' => true
));
```

You may also enable debugging during runtime with the Slim application's `config()` instance method:

```
<?php
$app = new \Slim\Slim();

//Enable debugging (on by default)
$app->config('debug', true);

//Disable debugging
$app->config('debug', false);
```

If debugging is enabled and an exception or error occurs, a diagnostic screen will appear with the error description, the affected file, the file line number, and a stack trace. If debugging is disabled, the custom Error handler will be invoked instead.

Back to Top ↑

Output Redirection

The Slim application's environment will always contain a key **slim.errors** with a value that is a writable resource to which log and error messages may be written. The Slim application's log object will write log messages to **slim.errors** whenever an Exception is caught or the log object is manually invoked.

If you want to redirect error output to a different location, you can define your own writable resource by modifying the Slim application's environment settings. I recommend you use middleware to update the environment:

```
<?php
class CustomErrorMiddleware extends \Slim\Middleware
{
    public function call()
    {
        // Set new error output
        $env = $this->app->environment;
        $env['slim.errors'] = fopen('/path/to/output', 'w');

        // Call next middleware
        $this->next->call();
    }
}
```

Remember, **slim.errors** does not have to point to a file; it can point to any valid writable resource.

Back to Top ↑

Dependency Injection

ocs.slimframework.com/#Dependency-)

DI Overview

Slim has a built-in resource locator, providing an easy way to inject objects into a Slim app, or to override any of the Slim app's internal objects (e.g. Request, Response, Log).

Injecting simple values

If you want to use Slim as a simple key-value store, it is as simple as this:

```
<?php
$app = new \Slim\Slim();
$app->foo = 'bar';
```

Now, you can fetch this value anywhere with `$app->foo` and get its value `bar`.

Using the resource locator

You can also use Slim as a resource locator by injecting closures that define how your desired objects will be constructed. When the injected closure is requested, it will be invoked and the closure's return value will be returned.

```
<?php
$app = new \Slim\Slim();

// Determine method to create UUIDs
$app->uuid = function () {
    return exec('uuidgen');
};

// Get a new UUID
$uuid = $app->uuid;
```

Singleton resources

Sometimes, you may want your resource definitions to stay the same each time they are requested (i.e. they should be singletons within the scope of the Slim app). This is easy to do:

```
<?php
$app = new \Slim\Slim();

// Define log resource
$app->container->singleton('log', function () {
    return new \My\Custom\Log();
});

// Get log resource
$log = $app->log;
```

Every time you request the log resource with `$app->log`, it will return the same instance.

Closure resources

What if you want to literally store a closure as the raw value and not have it invoked? You can do that like this:

```
<?php
$app = new \Slim\Slim();

// Define closure
$app->myClosure = $app->container->protect(function () {});

// Return raw closure without invoking it
$myClosure = $app->myClosure;
```

Back to Top ↑