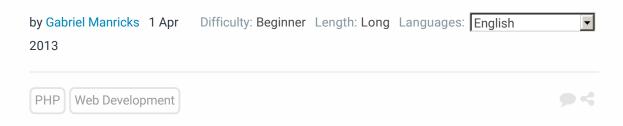
CODE > PHP

# Taming Slim 2.0



Slim is a lightweight framework that packs a lot of punch for its tiny footprint. It has an incredible routing system, and offers a solid base to work from without getting in your way. Let me show you!

But that's not to say that Slim doesn't has some issues; it's one-file setup becomes cluttered as your application grows. In this article, we'll review how to structure a Slim application to not only sustain, but improve its functionality and keep things neat and systematic.

## Vanilla Slim

Let's begin by looking at some common Slim code to identify the problem. After you've install Slim through Composer, you need to create an instance of the slim object and define your routes:

```
08
09  $app->get('/testPage', function() use ($app) {
10     $app->render('testpage.php');
11  });
12
13  $app->run();
```

## Let's turn the Slim object into the "controller."

The first method call sets a new route for the root URI (/), and connects the given function to that route. This is fairly verbose, yet easy to setup. The second method call defines a route for the URI testPage. Inside the supplied method, we use Slim's render() method to render a view.

Here lies the first problem: this function (a closure) is not called in the current context and has no way of accessing Slim's features. This is why we need to use the use keyword to pass the reference to the Slim app.

The second issue stems from Slim's architecture; it's meant to be defined all in one file. Of course, you can outsource the variable to another file, but it just gets messy. Ideally, we want the ability to add controllers to modularize the framework into individual components. As a bonus, it would be nice if these controllers offered native access to Slim's features, removing the need to pass references into the closures.

# **A Little Reverse Engineering**

It's debatable whether reading source code from an open-source project is considered reverse engineering, but it's the term I'll stick with. We understand how to use Slim, but what goes on under the hood? Let's look at a more complicated route to get to the root of this question:

```
$app->get('/users/:name', function($name){
ceho "Hello" . $name;
});
```

This route definition uses a colon with the word, <code>name</code>. This is a placeholder, and the value used in its place is passed to the function. For example, <code>/users/gabriel</code> matches this route, and 'gabriel' is passed to the function. The route, <code>/users</code>, on the other hand, is not a match because it is missing the parameter.

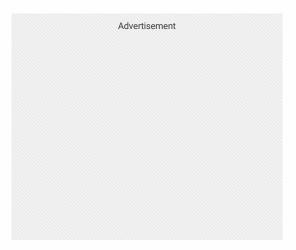
If you think about it logically, there are a number of steps that must complete in order to process a route.

- Step One: check if the route matches the current URI.
- Step Two: extract all parameters from the URI.
- **Step Three**: call the connected closure and pass the extracted parameters.

To better optimize the process, Slim — using regex callbacks and groups — stores the placeholders as it checks for matches. This combines two steps into one, leaving only the need to execute the connected function when Slim is ready. It becomes clear that the route object is self-contained, and frankly, all that is needed.

In the previous example, we had access to Slim's features when parsing the routes, but we needed to pass a Slim object reference because it would otherwise be unavailable within the function's execution context. That's all you need for most applications, as your application's logic should occur in the controller.

With that in mind, let's extract the "routing" portion into a class and turn the Slim object into the "controller."



# **Getting Started**

To begin, let's download and install "vanilla Slim" if you haven't done so already. I'm going to assume that you have Composer installed, but if not, follow the steps .

Within a new directory, create a file named composer.json, and append the following:

In a terminal window, navigate to said directory and type composer install. I'll walk you through these packages, if this is you're first time using Slim.

- slim/slim the actual Slim framework.
- slim/extras a set of optional classes to extend Slim.
- twig/twig the Twig templating engine.

You technically don't need the Slim extras or Twig for this tutorial, but I like using Twig instead of standard PHP templates. If you use Twig, however, you need the Slim extras because it provides an interface between Twig and Slim.

Now lets add our custom files, and we'll start by adding a directory to the vendors folder. I'll name mine Nettuts, but feel free to name yours whatever you wish. If you are still in the terminal, ensure that your terminal window is in the project's directory and type the following:

```
1 | mkdir vendor/Nettuts
```

Now, edit | composer.json | by adding the reference to this new folder:

We want our app to automatically load classes from the <code>Nettuts</code> namespace, so this tells Composer to map all requests for <code>Nettuts</code> to the PSR-0 standard

starting from the vendor folder.

Now execute:

```
1 composer dump-autoload
```

This recompiles the autoloader to include the new reference. Next, create a file, named Router.php, within the Nettuts directory, and enter the following:

```
1  <?php
2
3  namespace Nettuts;
4
5  Class Router
6  {
7  }</pre>
```

We saw that each route object has a self-contained function that determines if it matches the provided URI. So, we want an array of routes and a function to parse through them. We'll also need another function to add new routes, and a way to retrieve the URI from the current HTTP request.

Let's begin by adding some member variables and the constructor:

We set the routes variable to contain the routes, and the request variable to store the Slim Request object. Next, we need the ability to add routes. To stick with best practices, I will break this into two steps:

```
$\func = \func = \func \( \func \);

$\func = \func \( \func \);

$\f
```

This public function accepts an associative array of routes in the format of route => path, where route is a standard Slim route and path is a string with the following convention:

Optionally, you can leave out certain parameters to use a default value. For example, the class name will be replaced with Main if you leave it out, index is the default for omitted function names, and the default for the HTTP method is any. Of course, any is not a real HTTP method, but it is a value that Slim uses to match all HTTP method types.

The addRoutes function starts with a foreach loop that cycles through the routes. Next, we set the default HTTP method, optionally overriding it with the provided method if the graph symbol is present. Then we pass the remainder of the path to a function to retrieve a callback, and attach it to a route. Finally, we add the route to the array.

Now let's look at the processCallback() function:

```
protected function processCallback($path)

{
    $class = "Main";

    if (strpos($path, ":") !== false) {
        list($class, $path) = explode(":", $path);
    }

    $function = ($path != "") ? $path : "index";

    $func = function () use ($class, $function) {
        $class = '\Controllers\\' . $class;
        $class = new $class();

        $args = func_get_args();

        return call_user_func_array(array($class, $function), $args);
};

return $func;
```

We first set the default class to Main, and override that class if the colon symbol is found. Next, we determine if a function is defined and use the default method index if necessary. We then pass the class and function names to a closure and return it to the route.

Inside the closure, we prepend the class name with the namespace. We then create a new instance of the specified class and retrieve the list of arguments passed to this function. If you remember, while Slim checks if a route matches, it slowly builds a list of parameters based on wildcards from the route. This function (func\_get\_args()) can be used to get the passed parameters in an array. Then, using the call\_user\_func\_array() method enables us to specify the class and function, while passing the parameters to the controller.

It's not a very complicated function once you understand it, but it is a very good example of when closures come in handy.

To recap, we added a function to our Router that allows you to pass an associative array containing routes and paths that map to classes and functions. The last step is to process the routes and execute any that match. Keeping with the Slim naming convention, let's call it run:

```
public function run()
02 {
        $display404 = true;
        $uri = $this->request->getResourceUri();
        $method = $this->request->getMethod();
        foreach ($this->routes as $i => $route) {
            if ($route->matches($uri)) {
                if ($route->supportsHttpMethod($method) || $route->supportsHt
                    call user func array($route->getCallable(), array values
                    $display404 = false;
                }
            }
14
        }
        if ($display404) {
            echo "404 - route not found";
        }
                                                                            •
```

We begin by setting the <code>display404</code> variable, representing no routes found, to <code>true</code>. If we find a matching route, we'll set this to <code>false</code> and bypass the error message. Next, we use Slim's request object to retrieve the current URI and HTTP method.

We'll use this information to cycle through and find matches from our array.

Once the route object's <code>matches()</code> function executes, you are able to call <code>getParams()</code> to retrieve the parsed parameters. Using that function and the <code>getCallable()</code> method, we are able to execute the closure and pass the necessary parameters. Finally, we display a 404 message if no route matched the current URI.

Let's create the controller class that holds the callbacks for these routes. If you have been following along, then you may have realized that we never forced a protocol or class type. If you don't want to create a controller class, then any class will work fine.

So why are create a controller class? The short answer is we still haven't really used Slim! We used parts of Slim for the HTTP request and routes, but the whole point of this was to have easy access to all of Slim's properties. Our controller class will extend the actual Slim class, gaining access to all of Slim's methods.

You can just as easily skip this and subclass Slim directly from your controllers.

# **Building the Controller**

This controller basically allows you to modify Slim while still keeping it vanilla. Name the file <code>controller.php</code>, and write the following code:

```
if (isset($settings['model'])) {
    $this->data = $settings['model'];
}

parent::__construct($settings);
}
```

When you initialize Slim, you can pass in a variety of settings, ranging from the application's debug mode to the templating engine. Instead of hard coding any values in the constructor, I load them from a file named settings.php and pass that array into the parent's constructor.

Because we are extending Slim, I thought it would be cool to add a 'model' setting, allowing people to hook their data object directly into the controller.

That's the section you can see in the middle of the above code. We check if the model setting has been set and assign it to the controller's data property if necessary.

Now create a file named settings.php in the root of your project (the folder with the composer.json file), and enter the following:

These are standard Slim settings with the exception of the model. Whatever value is assigned to the model property is passed to the data variable; this could be an array, another class, a string, etc... I set it to an object because I like using the -> notation instead of the bracket (array) notation.

We can now test the system. If you remember in the Router class, we prepend the class name with the "Controller" namespace. Open up Composer.json add the following directly after the psr-0 definition for the Nettuts namespace:

```
01 {
02    "name": "nettuts/slim_advanced",
03    "require": {
```

Then like before, just dump the autoloader:

```
1 composer dump-autoload
```

If we just set the base path to the root directory, then the namespace Controller will map to a folder named "Controller" in the root of our app. So create that folder:

```
1 | mkdir Controller
```

Inside this folder, create a new file named Main.php. Inside the file, we need to declare the namespace and create a class that extends our Controller base class:

This is not complicated, but let's take it in moderation. In this class, we define two functions; their names don't matter because we will map them to routes later. It's important to notice that I directly access properties from the controller (i.e. the model) in the first function, and in fact, you will have full access to all of Slim's commands.

Let's now create the actual public file. Create a new directory in the root of your

project and name it <code>public</code>. As its name implies, this is were all the public stuff will reside. Inside this folder, create a file called <code>index.php</code> and enter the following:

We include Composer's autoloading library and create a new instance of our router. Then we define two routes, add them to the router object and execute it.

You also need to turn on mod\_rewrite in Apache (or the equivalent using a different web server). To set this up, create a file named .htaccess inside the public directory and fill it with the following:

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

Now all requests to this folder (that do not match an actual file) will be transferred to index.php.

In your browser, navigate to your public directory, and you should see a page that says "Hello World". Navigate to "/test", and you should see the message "Test Page". It's not terribly exciting, but we have successfully moved all the logic code into individual controllers.

## **Round Two**

Slim is not Codelgniter, it's not Symfony and it's not Laravel.

So we have basic functionality, but there are a few rough edges. Let's start with the router.

As of right now, we display a simple error message if a route doesn't exist. In a real application, we want the same functionality as loading a regular page. We want to take advantage of Slim's ability to load views, as well as set the response's error code.

Let's add a new class variable that holds an optional path (just like the other routes). At the top of the file, add the following line directly after the request object definition:

```
1 protected $errorHandler;
```

Next, let's create a function that accepts a path and assigns it a callback function. This is relatively simple because we already abstracted this functionality:

```
public function set404Handler($path)

{
    $this->errorHandler = $this->processCallback($path);
}
```

Now let's adjust the run command to optionally execute the callback instead of just displaying the error message:

```
if ($display404) {
    if (is_callable($this->errorHandler)) {
        call_user_func($this->errorHandler);
    } else {
        echo "404 - route not found";
}
```

Open the controller class. This is where you can adjust Slim's functionality to your own personal preferences. For example, I would like the option to omit the file extension when loading views. So instead of writing sthis-

```
public function render($name, $data = array(), $status = null)

if (strpos($name, ".php") === false) {
    $name = $name . ".php";
}

parent::render($name, $data, $status);
}
```

We accept the same parameters as the parent function, but we check if the file extension is provided and add it if necessary. After this modification, we pass the file to the parent method for processing.

This is just a single example, but we should put any other changes here in the render() method. For example, if you load the same header and footer pages on all your documents, you can add a function renderPage(). This function would load the passed view between the calls to load the regular header and footer.

Next, let's take a look at loading some views. In the root of your project create a folder named "views" (the location and name can be adjusted in the settings.php file). Let's just create two views named test.php and error.php.

Inside test.php , add the following:

And inside the error.php file, enter this:

Also, modify the  $_{\text{Main}}$  controller by changing the  $_{\text{index}()}$  function to the following:

Here, we render the test view that we just made and pass it data to display. Next, let's try a route with parameters. Change the <code>test()</code> function to the following:

```
public function test($title)
{
    $this->render("test", array("title" => $title, "name" => "Test"));
}
```

Here, we take it one step further by retrieving the page's title from the URI itself. Last, but not least, let's add a function for the 404 page:

We use the render () function's third optional parameter, which sets the response's HTTP status code.

Our final edit is in <code>index.php</code> to incorporate our new routes:

```
$\text{states} = \text{array}(\\ \ \'/\ => \', \\ \'/\ \text{states} = \'\mathreat{Get'}'
);

\text{states} = \text{array}(\\ \'/\ \'' => \'', \\ \'/\ \'' \'' => \'\mathreat{Main:test@get'}'
);

\text{states} \\
\text{stat
```

You should now be able to navigate to the three routes and see their respective views.

## **Conclusion**

With everything that we accomplished, you sure have a few questions about why Slim does not already offer these modifications. They seem logical, they don't stray from Slim's implementation too far, and they make a lot of sense.

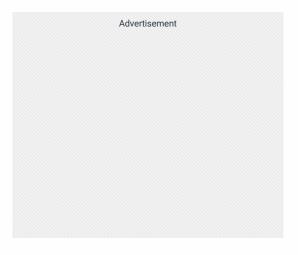
Josh Lockhart (Slim's creator) put it best:

"Slim is not Codelgniter, it's not Symfony, and it's not Laravel. Slim is Slim. It was built to be light-weight and fun, while still able to solve about 80% of the most common problems. Instead of worrying about the edge cases, it focuses on being simple and having an easy-to-read codebase."

Sometimes, as developers, we get so caught up covering crazy scenarios that we forget about what's really important: the code. Mods, like the one in this tutorial, are only possible because of the code's simplicity and verbosity. So yes,

there may be some edge cases that need special attention, but you get an active community, which in my opinion, heavily out-weighs the costs.

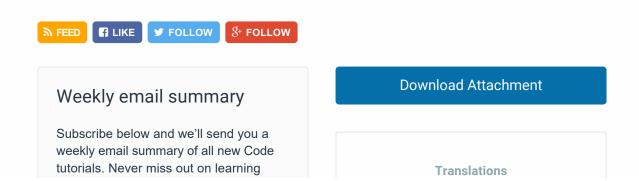
I hope you enjoyed this article. If you have any questions or comments, leave a message down below. You can also contact me through IRC channel on Freenode at the #nettuts channel.

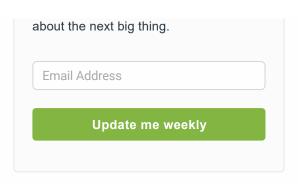




### **Gabriel Manricks**

I'm a freelance web developer with experience spanning the full stack of application development and a senior writer here at NetTuts+. Besides for that I spend my time writing books for Packt or working on open source projects I find intriguing . You can find me on Twitter @gabrielmanricks or visit my site to see all the things I'm working on gabrielmanricks.com.





Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

Advertisement

Advertisement

LOOKING FOR SOMETHING TO HELP KICK START YOUR NEXT PROJECT?

Envato Market has a range of items for sale to help get you started.

WordPress Plugins
From \$5

PHP Scripts

From \$5

### **Unlimited Downloads** From \$16.50/month

Get access to over 400,000 creative assets on Envato Elements.

#### **Over 9 Million Digital Assets**

Everything you need for your next creative project.

+ QUICK LINKS - Explore popular categories

**ENVATO TUTS+ JOIN OUR COMMUNITY HELP** 

Teach at Envato Tuts+ About Envato Tuts+ FAQ Terms of Use Translate for Envato Tuts+ Help Center

Advertise Forums

**●envato**tuts+

27,426 1,224 39,585

Courses Translations

#### Envato.com Our products Careers Sitemap

© 2019 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

Follow Envato Tuts+







