- Algorithms
- BBS Discussion
- Calendars-Dates
- Databases
- File Management
- Games
- Graphics
- HTML
- HTTP
- Math Functions
- Money
- Networking
- Other
- Shopping Carts
- Voting

- TIPS
  - Application Architecture
  - Databases
  - HTML
  - News & Reviews
  - PHP Functions
  - Setup & Installation
  - Site Operation
  - Tricks & Hacks
  - Trick & Hack Articles
  - Search Engines
- NEWS
  - Application Architecture
  - Code Documentation
  - Databases
  - HTML
  - News & Reviews
  - PEAR
  - PHP Functions
  - PHP-GTK
  - Setup & Installation
  - Site Operation
  - Tools
  - Tricks & Hacks
- FORUM
- CONTRIBUTE

Articles Application Architecture

Building RESTful APIs with the Slim Microframework

by: Jason Gilmore | October 5, 2011 Tweet +1

In addition to embracing the very latest in PHP features, among them anonymous functions, many PHP frameworks also offer native support for building REST-driven Web services, an approach which explicitly embraces key characteristics of the HTTP protocol. This relatively recent step forward for the PHP community is pretty significant, in that embracing REST can not only further reduce the amount of code PHP developers have to write in an effort to produce non-REST-compliant workarounds, but also because REST's formalized approach to mapping CRUD (create, read, update, delete) actions to HTTP methods greatly improves code clarity.

Although a relatively new entrant in the PHP framework sweepstakes, I've been lately quite intrigued by Slim, a slick RESTful microframework modeled after Ruby's Sinatra, which is coincidentally by far my favorite microframework available for any programming language. In this article I'll show you just how easy it is to get started building a powerful RESTful API using this streamlined framework.

Installing Slim

Slim is hosted on GitHub, meaning the easiest way to obtain the code is by cloning the project using your locally installed Git client. Begin by cloning the project using the following command:

```
$ git clone https://github.com/codeguy/Slim.git
```

This will create a directory named `Slim` which contains the Slim library and a sample RESTful application. For the purposes of this tutorial I've renamed the parent `Slim` directory to like `slim.localhost.com` and then configured the directory to point to an aptly-named virtual host, allowing me to access the sample application via *http://slim.localhost.com*.

If you don't have a Git client handy, then you can download a Zip file by heading over to the project's repository page and pressing the `Downloads` button.

Once configured to point to a virtual host, you should be able to pull up the virtual host's associated domain within your browser and see the sample application's default home page (Figure 1). If you receive an error then check to ensure Apache's mod_rewrite module is enabled, since like many frameworks Slim redirects all requests to a front controller named `index.php` which is responsible for routing requests to the appropriate route handler.

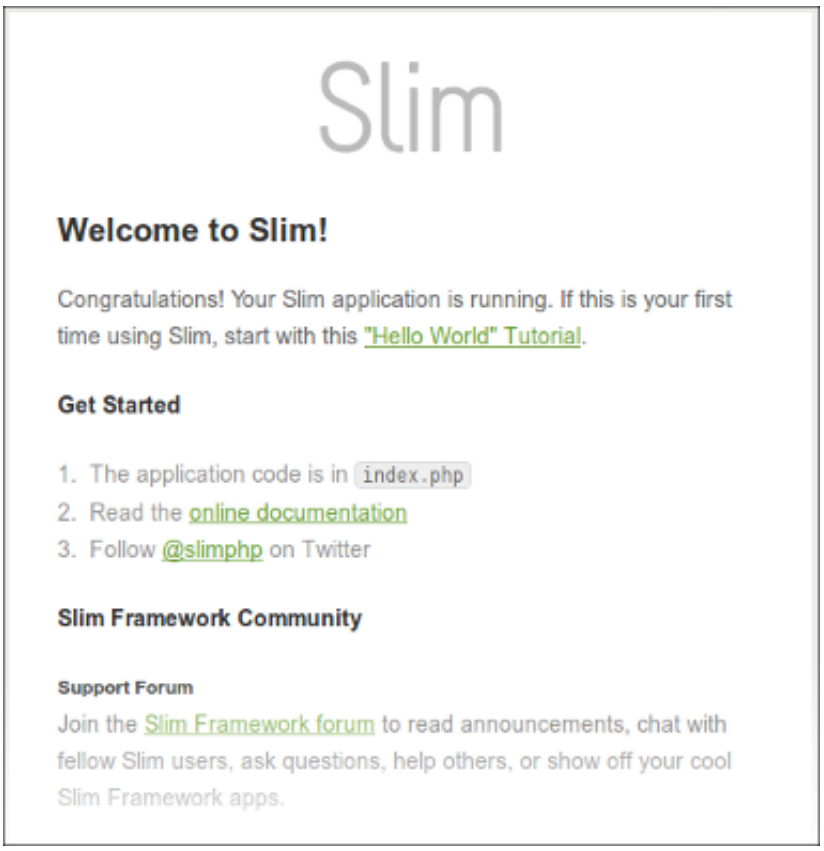

Figure 1. The Sample Slim Application

Creating Your First RESTful Route

After confirming that the sample application's index route is properly responding and presenting the default home page (Figure 1), I suggest deleting or renaming the `index.php` file and starting anew by creating your own route handlers from scratch. We'll start with an easy one, creating a handler which responds to requests sent to the API's home route using HTTP's `GET` method:

```
<?php

  // Include the Slim library
  require 'Slim/Slim.php';

  // Instantiate the Slim class
  $app = new Slim();

  // Create a GET-based route
  $app->get('/', function () {
    echo "HELLO SLIM";
  });

  // Ready the routes and run the application
  $app->run();
?>
```

Reload the home route in your browser (which in my case is *http://slim.localhost.com*) and you should see the `HELLO SLIM` output to the page.

Returning Dynamic Content

When building RESTful APIs, you'll regularly use HTTP's GET method to retrieve data from the server and pass it to whatever client has implemented the API interface. RESTful routes tend to follow a

rigorous structure which allows the developer to easily interpret their purpose. For instance, the route `/games` usually indicates that a list of games will be returned in response to a GET method sent to that endpoint. The following example demonstrates how an array of games will be returned in JSON format to any GET request sent to `/games`:

```
$app->get('/games', function () {

  $games = array (
    array('id' => '1', 'title' => 'Homefront'),
    array('id' => '2', 'title' => 'Perfect Dark Zero'),
    array('id' => '3', 'title' => 'Devil May Cry 4'),
    array('id' => '4', 'title' => 'Pro Evolution Soccer 2011'),
  );

  echo json_encode($games);

});
```

Of course, there's nothing stopping you from adhering fully to the norms dictated by the MVC architecture, and encapsulating all data-related actions within a data layer. The following example is a refactorization of the previous, introducing a hypothetical `Game` model which encapsulates the data retrieval and JSON formatting:

```
...
require_once 'models/Game.php';
...

$app->get('/games', function () {

  $game = new Game();

  echo $games->findAll()->toJson();

});
```

Creating a New Game

Even if you've never heard of RESTful routing before, everything discussed so far should make perfect sense. So it seems an opportune time to introduce a bit of confusion into the picture. The last example demonstrated how a RESTful route defined as `/games` would be configured in order to retrieve a list of games. What would you say to the idea that `/games` would *also* be used to create a new game? Indeed this is the case, which doesn't make any sense until you take the HTTP method into account! Remember that RESTful routes are defined by both their endpoints and the HTTP method used to perform the request, therefore `GET /games` and `POST /games` are considered to be two entirely unique routes! Let's configure a route which will use the hypothetical `Game` model to add a game to the database:

```
...
require_once 'models/Game.php';
...

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

  $game = new Game();

  $game->setTitle($app->request()->post('title'));

  $game->save();

  $id = $game->getId();
```

```
  $app->redirect("/games/{$id}");

});
...
$app->get('/games/:id', function ($id) {
  // Retrieve game associated with an ID of $id
});
```

Notice how we're incorporating another useful feature which results in the client being redirected to another route which will result in the game associated with the ID `$id` being displayed. Although there are several ways one could conceivably handle the post-save behavior, I thought demonstrating this approach would be useful in that it would not only show how Slim is capable of redirecting the client, but also how parameterized routes can be incorporated into your API.

Modifying Existing Games

So far you've learned how to retrieve and create games, but what about updating them? To update a game you'll use the PUT method in conjunction with a route which identifies the game you'd like to update: `PUT /games/:id`. I'll demonstrate this feature, again using the hypothetical `Game` model:

```
$app->put('/games/:id', function ($id) use ($app) {
  $game = new Game();
  $game->find($id);
  $game->setTitle($app->request()->put('title'));
  $game->save();
});
```

The first three CRUD components (create, retrieve and update) have been demonstrated; try continuing this exercise by learning more about how Slim implements the DELETE route!

What's Next for Slim

Although Slim is already quite capable of building simple REST APIs, at the time of this writing it's missing a few key features, including notably the ability to conveniently respond accordingly to requests asking for different content types (JSON and XML, for instance). I'm pleased to report that according to the development roadmap this feature is already in the works and slated for inclusion in the next point release (1.5.2).

If you plan to explore more frameworks, I've written a great deal about PHP frameworks large and small, having recently covered DooPHP, Fat-Free, Fuel and Yii, to name a few.

About the Author

Jason Gilmore is founder of the publishing, training, and consulting firm WJGilmore.com. He is the author of several popular books, including "Easy PHP Websites with the Zend Framework", "Easy PayPal with PHP", and "Beginning PHP and MySQL, Fourth Edition". Follow him on Twitter at @wjgilmore.

Email:

Comment:

(Maximum characters: 1200). You have 1200 characters left.

Type the text

Privacy & Terms