

Contents

Introduction

Understanding REST

Creating the application stub

Handling GET requests

Handling POST requests

Handling PUT and DELETE requests

Adding authentication with route middleware

Supporting multiple-response formats

Conclusion

Downloadable resources

Related topics

Comments

Create REST applications with the Slim micro-framework

Use the Slim PHP micro-framework to efficiently prototype and deploy a REST API

**Vikram Vaswani**

Published on December 11, 2012



6

Until a few years ago, using a framework to develop PHP applications was the exception rather than the rule. Today, frameworks like [CakePHP](#), [Symfony](#), [CodeIgniter](#), and [Zend Framework](#) are widely used for PHP application development, each one offers unique features to streamline and simplify application development.

Often a full-fledged framework is overkill. Think about prototyping a web application, creating a quick and dirty CRUD front end, or deploying a basic REST API. You can do all these tasks with a traditional framework, but the time and effort involved in learning and using it often outweigh the benefits. Now consider micro-frameworks which enable rapid web application development and prototyping without the performance overhead and learning curve of full-fledged frameworks.

In this article, I introduce you to Slim, a PHP micro-framework that's designed for rapid development of web applications and APIs. Don't be fooled by the name: Slim comes with a sophisticated URL router and support for page templates, flash messages, encrypted cookies, and

middleware. It's also extremely easy to understand and use, and it comes with great documentation and an enthusiastic developer community.

This article walks you through the process of building a simple REST API with Slim. In addition to explaining how to implement the four basic REST methods with Slim's URL router, it demonstrates how Slim's unique features make it easy to add advanced features such as API authentication and multi-format support.

Understanding REST

First, refresh your understanding of REST, otherwise known as Representational State Transfer. REST differs from SOAP in that it is based on resources and actions, rather than on methods and data types. A resource is simply a URL referencing the object or entity on which you want to perform the actions—for example, `/users` or `/photos`—and an action is one of the four HTTP verbs:

- GET (retrieve)
- POST (create)
- PUT (update)
- DELETE (remove)

To better understand this, consider a simple example. Suppose that you have a file-sharing application and you need API methods for developers to remotely add new files to, or retrieve existing files from, the application data store. Under the REST approach, you expose a URL endpoint, such as `/files`, and examine the HTTP method used to access this URL to understand the action required. For example, you POST an HTTP packet to `/files` to create a new file, or send a request to GET `/files` to retrieve a list of available files.

This approach is much easier to understand, as it maps existing HTTP verbs to CRUD operations. It also consumes fewer resources, because no formal definition of data types of request/response headers is needed.

The typical REST conventions for URL requests, and what they mean, are:

- GET /items: Retrieve a list of items
- GET /items/123: Retrieve item 123
- POST /items: Create a new item
- PUT /items/123: Update item 123
- DELETE /items/123: Remove item 123

Slim's URL router helps developers "map resource URIs to callback functions for specific HTTP request methods," such as GET, POST, PUT, and DELETE. Defining a new REST API is simple as defining these callback methods and filling them with appropriate code. And that's exactly what you will do in the rest of this article.

Creating the application stub

Before you start to implement a REST API, let's review a few notes. Throughout this article, I assume that you have a working development environment (Apache, PHP, and MySQL) and that you're familiar with the basics of SQL and XML. I also assume that your Apache web server is configured to support virtual hosting, URL rewriting, and PUT and DELETE requests.

A REST API is designed to work with resources. The resources in this case are articles, each of which has a title, a URL, a date, and a unique identifier. These resources are stored in a MySQL database, and the example REST API developed in this article allows developers to retrieve, add, delete, and update these articles using normal REST conventions. The majority of this article assumes JSON request and response bodies. For more information on how to handle XML requests and responses, see [Supporting multiple-response formats](#) later in the article.

Step 1: Create the application directory structure

Change to the web server's document root directory (typically /usr/local/apache/htdocs on Linux® or c:\Program Files\Apache\htdocs on Windows®) and create a new subdirectory for the application. Name this directory slim/.

```
1 | shell> cd /usr/local/apache/htdocs
2 | shell> mkdir slim
```

This directory is referenced throughout this article as \$SLIM_ROOT.

Step 2: Download the Slim framework

Next, you need to add Slim. If you're using Composer, the PHP dependency manager, simply create a file at \$SLIM_ROOT/composer.json with the following content:

```
1 | {
2 |     "require": {
3 |         "slim/slim": "2.*"
4 |     }
5 | }
```

Install Slim using Composer with the command:

```
1 | shell> php composer.phar install
```

To load Slim, add this line to your application's 'index.php' file:

```
1 | <?php
2 | require 'vendor/autoload.php';
```

If you do not use Composer, you can manually download the Slim framework and extract the Slim directory within the download archive to a directory in your PHP include path, or to \$SLIM_ROOT/Slim.

Step 3: Initialize the example database

The next step is to initialize the application database. Create

a new MySQL table named "articles" to hold article records:

```
1 CREATE TABLE IF NOT EXISTS `articles` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `title` text NOT NULL,  
4   `url` text NOT NULL,  
5   `date` date NOT NULL,  
6   PRIMARY KEY (`id`)  
7 ) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

Seed this table with a few starter records:

```
1 INSERT INTO `articles` (`id`, `title`, `url`, `date`  
2 (1, 'Search and integrate Google+ activity streams w  
3 'http://www.ibm.com/developerworks/xml/library/x-goo  
4  
5 INSERT INTO `articles` (`id`, `title`, `url`, `date`  
6 (2, 'Getting Started with Zend Server CE',  
7 'http://devzone.zend.com/1389/getting-started-with-z
```



To interact with this MySQL database, you should also download RedBeanPHP, a low-footprint ORM library. This library is available as a single file that you can easily include in a PHP script. Remember to copy the file to a directory in your PHP include path or to \$SLIM_ROOT/RedBean.

Step 4: Define a virtual host

To make it easier to access the application, define a new virtual host and set it to the working directory. This optional step is recommended especially when you work on a development machine that holds multiple in-progress applications as it creates a closer replica of the target deployment environment. Slim also comes with a ".htaccess" file that lets you use pretty URLs that remove the "index.php" prefix.

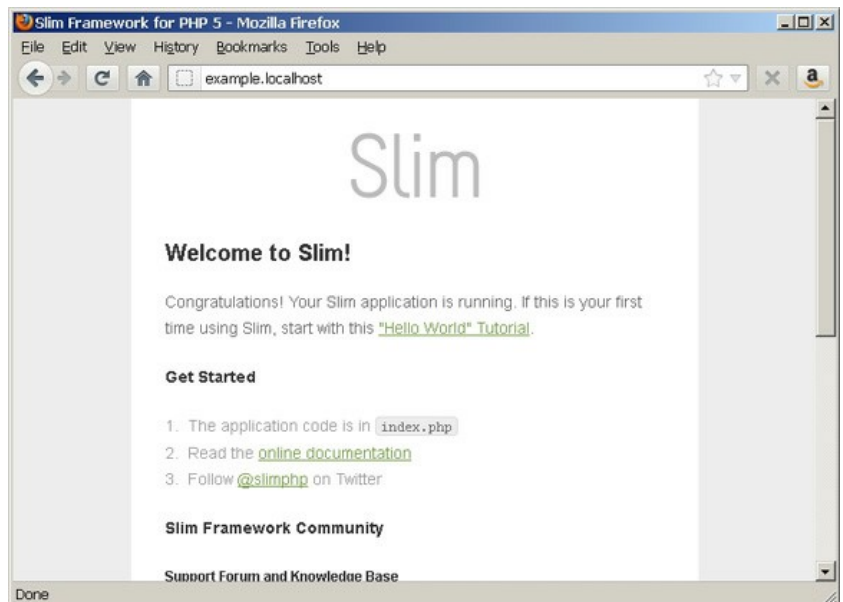
To set up a named virtual host for the application, open the Apache configuration file (httpd.conf or httpd-vhosts.conf) and add these lines to it:

```
1 NameVirtualHost 127.0.0.1  
2 <VirtualHost 127.0.0.1>  
3     DocumentRoot "/usr/local/apache/htdocs/slim"  
4     ServerName example.localhost  
5 </VirtualHost>
```

These lines define a new virtual host, `http://example.localhost/`, whose document root corresponds to the `$SLIM_ROOT`. Restart the web server to activate these new settings. Note that you might need to update your network's local DNS server to tell it know about the new host.

After you complete this task, point your browser to `http://example.localhost/`, and you should see something like [Figure 1](#).

Figure 1. The default Slim application index page



Handling GET requests

Slim works by defining router callbacks for HTTP methods and endpoints simply by calling the corresponding method—`get()` for GET requests, `post()` for POST requests, and so on—and passing the URL route to be matched as the first argument of the method. The final argument to the method is a function, which specifies the actions to take when the route is matched to an incoming request.

The typical REST API supports two types of GET requests, the first for a list of resources (GET `/articles`) and the second for a specific resource (GET `/articles/123`). Let's begin by writing code to handle the first scenario: responding to a GET `/articles` request with a list of all available article records.

Update your \$SLIM/index.php file with the code in [Listing 1](#):

Listing 1. The handler for multi-resource GET requests

```
1  <?php
2  // load required files
3  require 'Slim/Slim.php';
4  require 'RedBean/rb.php';
5
6  // register Slim auto-loader
7  \Slim\Slim::registerAutoloader();
8
9  // set up database connection
10 R::setup('mysql:host=localhost;dbname=appdata','use
11 R::freeze(true);
12
13 // initialize app
14 $app = new \Slim\Slim();
15
16 // handle GET requests for /articles
17 $app->get('/articles', function () use ($app) {
18     // query database for all articles
19     $articles = R::find('articles');
20
21     // send response header for JSON content type
22     $app->response()->header('Content-Type', 'applica
23
24     // return JSON-encoded response body with query r
25     echo json_encode(R::exportAll($articles));
26 });
27
28 // run
29 $app->run();
```

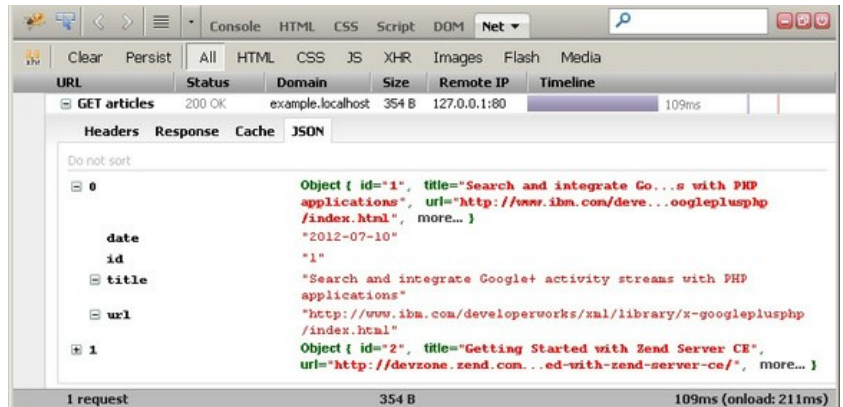
[Listing 1](#) loads the Slim and RedBean classes. It also registers the Slim auto-loader to ensure that other Slim classes are loaded as needed (remember that you don't need to require Slim or to register its auto-loader if you use Composer). Next, RedBeanPHP's `R::setup()` method opens a connection to the MySQL database by passing it the appropriate credentials, and its `R::freeze()` method locks the database schema against any RedBean-propagated changes. Finally, a new Slim application object is initialized; this object serves as the primary control point for defining router callbacks.

The next step is to define router callbacks for HTTP methods and endpoints. [Listing 1](#) calls the application object's `get()` method and passes it the URL route `/articles` as its first argument. The anonymous function passed as the final argument uses RedBeanPHP's `R::find()` method to retrieve all the records from the `'articles'` database table, turn them into a PHP array with the `R::exportAll()`

method, and return the array to the caller as a JSON-encoded response body. Slim's response object also exposes a `header()` method that you can use to set any response header; in [Listing 1](#), the `header()` method sets the 'Content-Type' header so the client knows to interpret it as a JSON response.

[Figure 2](#) shows the result of a request to this endpoint.

Figure 2. A GET request and response in JSON format



Note that you can use the `$app->contentType()` helper method instead of the `header()` method to directly access the request's content type, or treat Slim's response object as an array (it implements the `ArrayAccess` interface) and set headers.

Similarly, you can handle GET requests for specific resources, by adding a callback for the URL route `"/articles/:id"`. [Listing 2](#) illustrates.

Listing 2. The handler for single-resource GET requests


```

1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  class ResourceNotFoundException extends Exception {
8
9  // handle GET requests for /articles/:id
10 $app->get('/articles/:id', function ($id) use ($app)
11     try {
12         // query database for single article
13         $article = R::findOne('articles', 'id=?', array
14
15             if ($article) {
16                 // if found, return JSON response
17                 $app->response()->header('Content-Type', 'app
18                 echo json_encode(R::exportAll($article));
19             } else {
20                 // else throw exception
21                 throw new ResourceNotFoundException();
22             }
23         } catch (ResourceNotFoundException $e) {
24             // return 404 server error
25             $app->response()->status(404);
26         } catch (Exception $e) {
27             $app->response()->status(400);
28             $app->response()->header('X-Status-Reason', $e-
29         }
30     });
31
32 // run
33 $app->run();

```



Slim can automatically extract route parameters from URLs. [Listing 2](#) illustrates this as it extracts the `:id` parameter from the GET request and passes it to the callback function. Within the function, the `R::findOne()` method retrieves the corresponding resource record. This record is returned to the client as a JSON-encoded response body. In the event that the provided identifier is invalid and no matching resource is found, the callback function throws a custom `ResourceNotFoundException`. The try- catch block then converts this into a 404 Not Found server response.

[Figure 3](#) displays the result of a successful request.

Figure 3. A GET request and response (successful) in JSON format

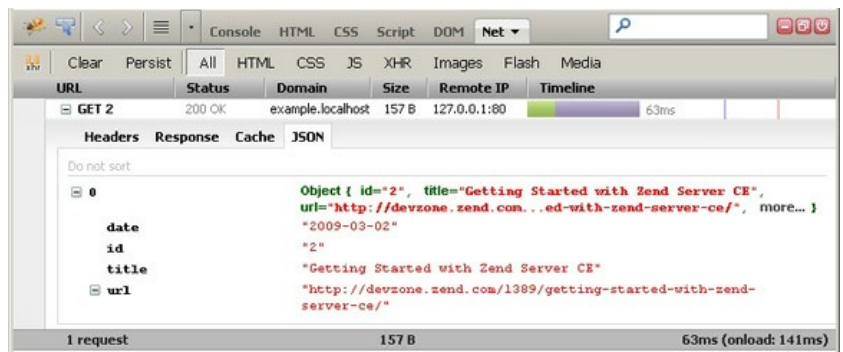
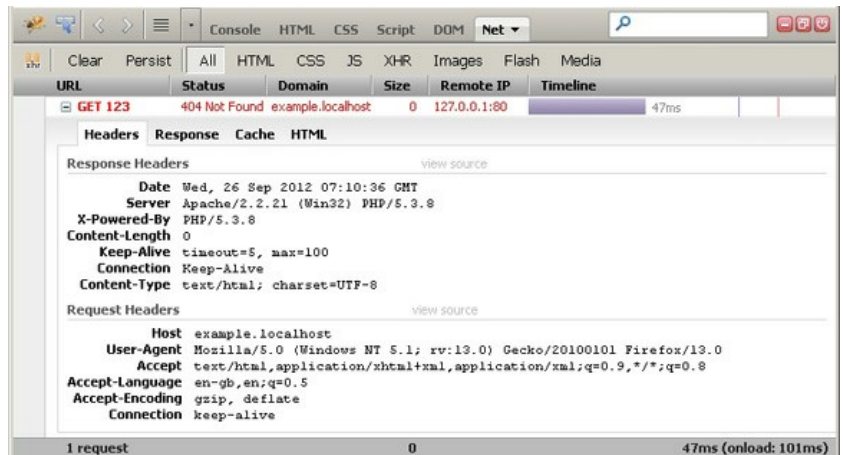


Figure 4 displays the result of an unsuccessful one.

Figure 4. A GET request and response (unsuccessful) in JSON format



Slim also supports route conditions, allowing developers to specify regular expressions for route parameters. If these parameters are not met, the route callback will not execute. You can specify conditions on a per-route basis, as shown in this code:

```
1 <?php
2 $app->get('/articles/:id', function ($id) use ($app)
3 // callback code
4 }->conditions(array('id' => '([0-9]{1,})'));
```



Or you can specify conditions globally using the `setDefaultConditions()` method, as shown here:

```
1 <?php
2 // set default conditions for route parameters
3 \Slim\Route::setDefaultConditions(array(
4 'id' => '([0-9]{1,})',
5 ));
```

A Slim application also exposes a `notFound()` method, which you can use to define custom code for scenarios where no route match is possible. This method is an

alternative to throwing a custom exception and manually setting the 404 server response code.

Handling POST requests

Handling POST requests is a little more involved. The usual REST convention is that a POST request creates a new resource, with the request body containing all the necessary inputs (in this case, the author and title) for the resource. The callback function needs to decode the request body, convert it into a RedBean object, and save it to the database, then return a JSON-encoded representation of the newly-created resource.

[Listing 3](#) displays the code to handle POST requests.

Listing 3. The handler for POST requests

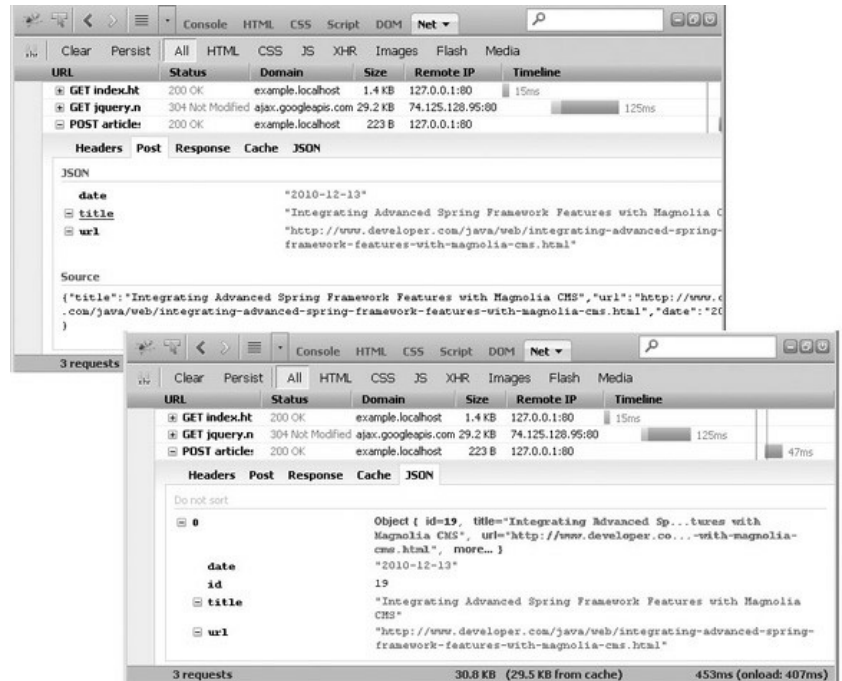
```
1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // handle POST requests to /articles
8  $app->post('/articles', function () use ($app) {
9      try {
10         // get and decode JSON request body
11         $request = $app->request();
12         $body = $request->getBody();
13         $input = json_decode($body);
14
15         // store article record
16         $article = R::dispense('articles');
17         $article->title = (string)$input->title;
18         $article->url = (string)$input->url;
19         $article->date = (string)$input->date;
20         $id = R::store($article);
21
22         // return JSON-encoded response body
23         $app->response()->header('Content-Type', 'appli
24         echo json_encode(R::exportAll($article));
25     } catch (Exception $e) {
26         $app->response()->status(400);
27         $app->response()->header('X-Status-Reason', $e-
28     }
29     .});
30
31 // run
32 $app->run();
```

In [Listing 3](#), the callback function retrieves the body of the request (which is assumed to be a JSON-encoded packet) using the request object's `getBody()` and converts it to a

PHP object using the `json_decode()` function. Next, a new RedBeanPHP article object is initialized with the properties from the PHP object and stored in the database using the `R::store()` method. The new object is then exported to an array and returned to the client as a JSON packet.

Figure 5 displays an example POST request and response.

Figure 5. A POST request and response in JSON format



Handling PUT and DELETE requests

PUT requests are used to indicate a modification to an existing resource and, as such, include a resource identifier in the request string. A successful PUT implies that the existing resource has been replaced with the resource specified in the PUT request body. The response to a successful PUT can be either status code 200 (OK), with the response body containing a representation of the updated resource, or status code 204 (No Content) with an empty response body.

Listing 4 displays the code to handle PUT requests.

Listing 4. The handler for PUT requests

```

1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // handle PUT requests to /articles/:id
8  $app->put('/articles/:id', function ($id) use ($app)
9  {
10     try {
11         // get and decode JSON request body
12         $request = $app->request();
13         $body = $request->getBody();
14         $input = json_decode($body);
15
16         // query database for single article
17         $article = R::findOne('articles', 'id=?', array
18         (
19             // store modified article
20             // return JSON-encoded response body
21             if ($article) {
22                 $article->title = (string)$input->title;
23                 $article->url = (string)$input->url;
24                 $article->date = (string)$input->date;
25                 R::store($article);
26                 $app->response()->header('Content-Type', 'app
27                 echo json_encode(R::exportAll($article));
28             } else {
29                 throw new ResourceNotFoundException();
30             }
31         } catch (ResourceNotFoundException $e) {
32             $app->response()->status(404);
33         } catch (Exception $e) {
34             $app->response()->status(400);
35             $app->response()->header('X-Status-Reason', $e-
36             });
37         }
38     }
39     // run
40     $app->run();

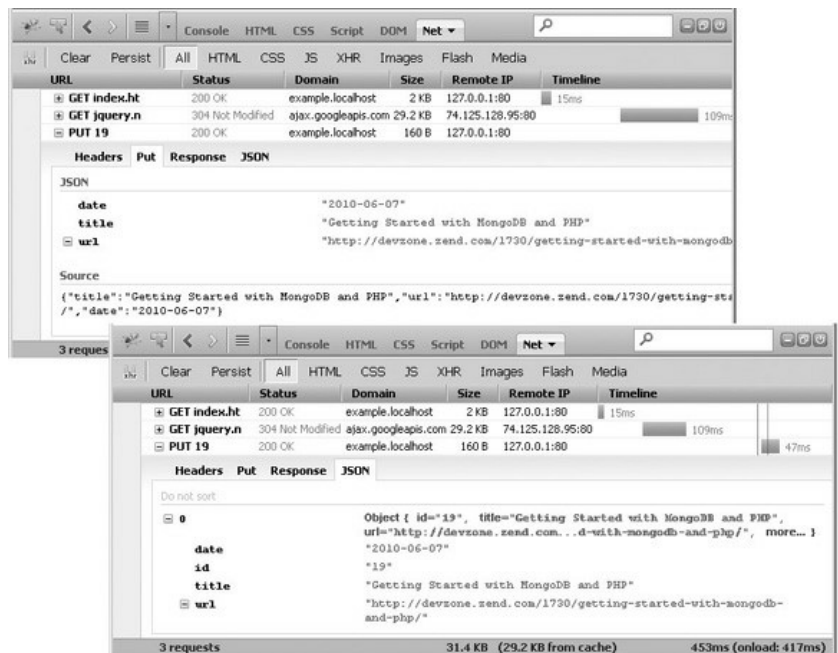
```



In [Listing 4](#), the identifier provided as part of the URL route is used to retrieve the corresponding resource from the database as a RedBeanPHP object. The body of the request (which is assumed to be a JSON-encoded packet) is converted to a PHP object with the `json_decode()` function, and its properties are used to overwrite the properties of the RedBeanPHP object. The modified object is then saved back to the database, and a JSON representation returned to the caller with a 200 server response code. In the event that the identifier provided does not match an existing resource, a custom exception is thrown and a 404 server error response sent back to the client.

[Figure 6](#) displays an example PUT request and response.

Figure 6. A PUT request and response in JSON format



You also can write a callback to handle DELETE requests that remove the specified resource from the data store.

[Listing 5](#) displays the necessary code.

Listing 5. The handler for DELETE requests

```

1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // handle DELETE requests to /articles/:id
8  $app->delete('/articles/:id', function ($id) use ($
9      try {
10         // query database for article
11         $request = $app->request();
12         $article = R::findOne('articles', 'id=?', array
13
14         // delete article
15         if ($article) {
16             R::trash($article);
17             $app->response()->status(204);
18         } else {
19             throw new ResourceNotFoundException();
20         }
21     } catch (ResourceNotFoundException $e) {
22         $app->response()->status(404);
23     } catch (Exception $e) {
24         $app->response()->status(400);
25         $app->response()->header('X-Status-Reason', $e-
26     }
27 });
28
29 // run
30 $app->run();

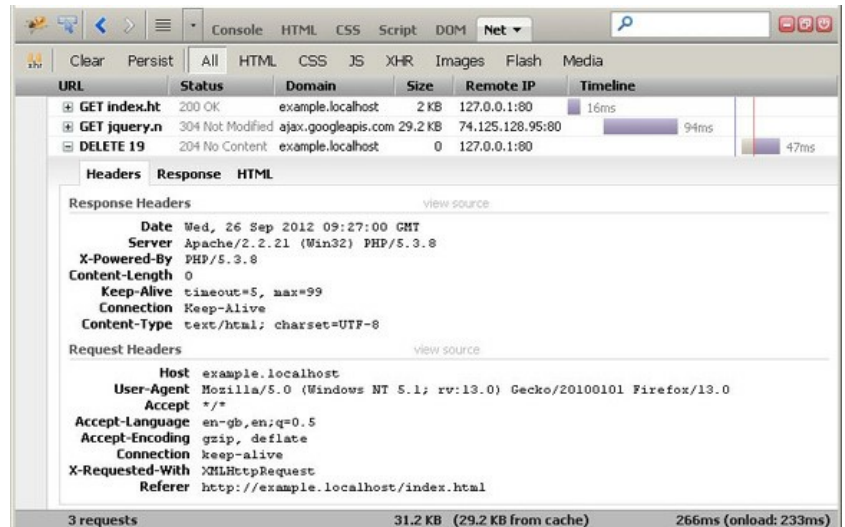
```

As with [Listing 4](#), [Listing 5](#) uses the resource identifier from the URL to retrieve the corresponding resource from the database as an object and then uses the `R::trash()`

method to permanently delete it. The response to a successful DELETE request can be a 200 (OK) with the status in the response body or a 204 (No Content) with an empty response body; [Listing 5](#) does the latter.

[Figure 7](#) displays an example DELETE request and the API response.

Figure 7. A DELETE request and response in JSON format



Adding authentication with route middleware

In addition to flexible routing, Slim also supports so-called "route middleware." In its simplest terms, middleware is one or more custom functions that are invoked before the callback function for the corresponding route. Middleware makes it possible to perform custom processing on a request; as an Ephemera blog post on "Rack Middleware Use Case Examples" explains, middleware "...can manipulate the request, the response, it can halt the processing altogether, or do completely unrelated stuff, like logging."

To illustrate how Slim's middleware architecture works, consider a common API requirement: authentication. With middleware, it's possible to ensure that API requests are properly authenticated, by passing the request through a custom authentication method before allowing them to be processed.

[Listing 6](#) illustrates a simple authentication function and adds this function as middleware to the GET request handler:

Listing 6. Authentication middleware

```
1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // route middleware for simple API authentication
8  function authenticate(\Slim\Route $route) {
9      $app = \Slim\Slim::getInstance();
10     if (validateUserKey() === false) {
11         $app->halt(401);
12     }
13 }
14
15 function validateUserKey($uid, $key) {
16     // insert your (hopefully complex) validation routine
17 }
18
19 // handle GET requests for /articles
20 $app->get('/articles', 'authenticate', function () {
21     // query database for all articles
22     $articles = R::find('articles');
23
24     // send response header for JSON content type
25     $app->response()->header('Content-Type', 'application/json');
26
27     // return JSON-encoded response body with query results
28     echo json_encode(R::exportAll($articles));
29 });
30
31 // run
32 $app->run();
```

The example `authenticate()` function in [Listing 6](#) is referenced from within the GET route handler as middleware, and it automatically receives the route object as an argument. It can also access the Slim application object using the `\Slim::getInstance()` method.

Now, when the user requests the URL `/articles`, the `authenticate()` function will be invoked before the request is processed. This function uses a custom validation routine to authenticate the request and allows further processing only if the validation routine returns true. In case validation fails, the `authenticate()` function halts processing and sends the client a 401 Authorization Required response.

[Listing 7](#) provides a more concrete example of how you might implement the `authenticate()` method, by checking

for cookies with the user identifier "demo" and API key "demo." These cookies are set by calling a new API method, /demo, that allows a client temporary access to the API for a five-minute period.

Listing 7. A simple implementation of authentication middleware

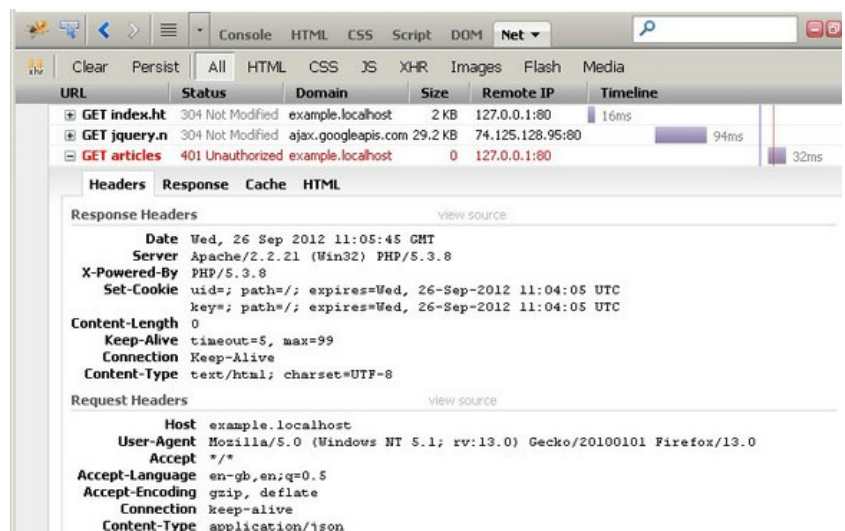
```
1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // route middleware for simple API authentication
8  function authenticate(\Slim\Route $route) {
9      $app = \Slim\Slim::getInstance();
10     $uid = $app->getEncryptedCookie('uid');
11     $key = $app->getEncryptedCookie('key');
12     if (validateUserKey($uid, $key) === false) {
13         $app->halt(401);
14     }
15 }
16
17 function validateUserKey($uid, $key) {
18     // insert your (hopefully more complex) validation
19     if ($uid == 'demo' && $key == 'demo') {
20         return true;
21     } else {
22         return false;
23     }
24 }
25
26 // handle GET requests for /articles
27 $app->get('/articles', 'authenticate', function ()
28     // query database for all articles
29     $articles = R::find('articles');
30
31     // send response header for JSON content type
32     $app->response()->header('Content-Type', 'application/
33
34     // return JSON-encoded response body with query results
35     echo json_encode(R::exportAll($articles));
36 });
37
38 // generates a temporary API key using cookies
39 // call this first to gain access to protected API
40 $app->get('/demo', function () use ($app) {
41     try {
42         $app->setEncryptedCookie('uid', 'demo', '5 minutes');
43         $app->setEncryptedCookie('key', 'demo', '5 minutes');
44     } catch (Exception $e) {
45         $app->response()->status(400);
46         $app->response()->header('X-Status-Reason', $e-
47     }
48 });
49
50 // run
51 $app->run();
```

In this case, the client first sends a request to the API

endpoint `/demo`", which sets encrypted cookies with the "demo" credentials valid for five minutes. These cookies will automatically accompany any subsequent request to the same host. The `authenticate()` middleware function, which is attached to the GET handler for the `/articles` URL endpoint, checks each incoming GET request for these cookies. It then uses the `validateUserKey()` function to verify the user's credentials (in this example, simply by checking for the value "demo"). After the cookies expire, the `validateUserKey()` function returns false, and the `authenticate()` middleware terminates the request and disallows access to the `/article` URL endpoint with a 401 server error.

Figure 8 displays the server error on an unauthenticated GET request.

Figure 8. An unauthenticated GET request and response in JSON format



The accompanying code archive uses this same middleware to authenticate POST, PUT, and DELETE requests. For obvious reasons, this system isn't particularly secure and should never be used in a production environment; it's included in this article purely for illustrative purposes.

As this example illustrates, route middleware comes in handy to perform request pre-processing; it can also be used for other tasks, such as request filtering or logging.

Supporting multiple-response formats

The previous sections illustrated how to set up a simple JSON-based REST API. XML is also a popular data-exchange format, and it's often necessary to support XML request and response bodies in your REST API.

Slim provides full access to request headers. One of the easiest ways to meet this requirement is to use the "Content-Type" request header to determine the data format used in the transaction. Consider [Listing 8](#), which revises the callback function for GET requests to return either XML or JSON response bodies, based on the "Content-Type" header:

Listing 8. The handler for GET requests, with XML and JSON format support

```

1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  // handle GET requests for /articles
8  $app->get('/articles', function () use ($app) {
9      try {
10         // query database for articles
11         $articles = R::find('articles');
12
13         // check request content type
14         // format and return response body in specified
15         $mediaType = $app->request()->getMediaType();
16         if ($mediaType == 'application/xml') {
17             $app->response()->header('Content-Type', 'app
18             $xml = new SimpleXMLElement('<root/>');
19             $result = R::exportAll($articles);
20             foreach ($result as $r) {
21                 $item = $xml->addChild('item');
22                 $item->addChild('id', $r['id']);
23                 $item->addChild('title', $r['title']);
24                 $item->addChild('url', $r['url']);
25                 $item->addChild('date', $r['date']);
26             }
27             echo $xml->asXml();
28         } else if (($mediaType == 'application/json'))
29             $app->response()->header('Content-Type', 'app
30             echo json_encode(R::exportAll($articles));
31         }
32         catch (Exception $e) {
33             $app->response()->status(400);
34             $app->response()->header('X-Status-Reason', $e-
35         }
36     });
37
38     // run
39     $app->run();

```

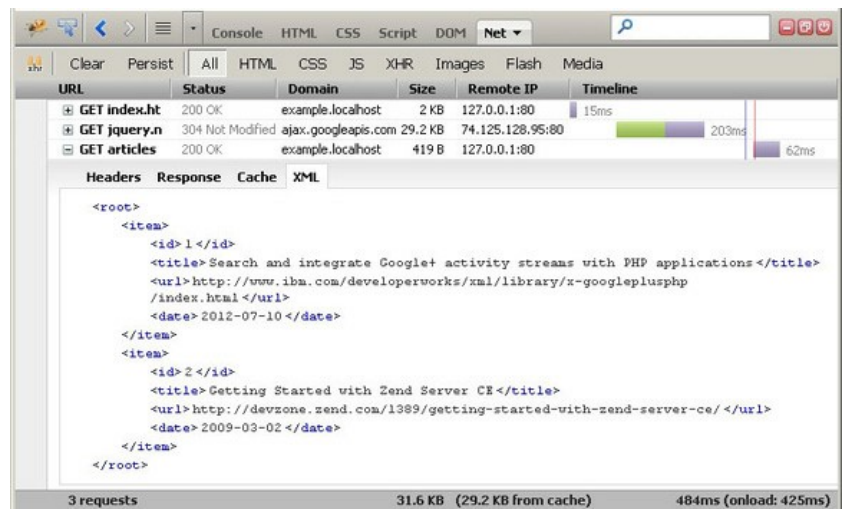


In [Listing 8](#), the request object's `getMediaType()` method is used to retrieve the content type of the request.

- For requests where the content type is "application/json", the list of articles is returned as before, in JSON format.
- For requests where the content type is 'application/xml', the list of articles is formatted as an XML document with SimpleXML and returned in XML format.

[Figure 9](#) illustrates the XML response to a GET request:

Figure 9. A GET request and response in XML format



Why stop there? [Listing 9](#) adds XML support for POST requests as well:

Listing 9. The handler for POST requests, with XML and JSON format support

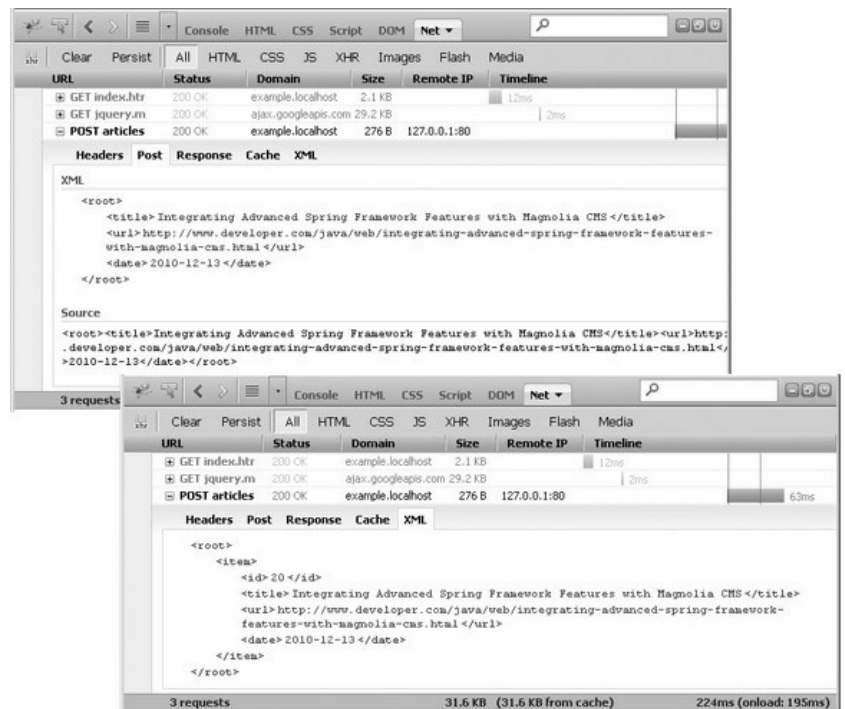
```

1  <?php
2  // do initial application and database setup
3
4  // initialize app
5  $app = new \Slim\Slim();
6
7  $app->post('articles', function () use ($app) {
8      try {
9          // check request content type
10         // decode request body in JSON or XML format
11         $request = $app->request();
12         $mediaType = $request->getMediaType();
13         $body = $request->getBody();
14         if ($mediaType == 'application/xml') {
15             $input = simplexml_load_string($body);
16         } elseif ($mediaType == 'application/json') {
17             $input = json_decode($body);
18         }
19
20         // create and store article record
21         $article = R::dispense('articles');
22         $article->title = (string)$input->title;
23         $article->url = (string)$input->url;
24         $article->date = (string)$input->date;
25         $id = R::store($article);
26
27         // return JSON/XML response
28         if ($mediaType == 'application/xml') {
29             $app->response()->header('Content-Type', 'app
30             $xml = new SimpleXMLElement('<root/>');
31             $result = R::exportAll($article);
32             foreach ($result as $r) {
33                 $item = $xml->addChild('item');
34                 $item->addChild('id', $r['id']);
35                 $item->addChild('title', $r['title']);
36                 $item->addChild('url', $r['url']);
37                 $item->addChild('date', $r['date']);
38             }
39             echo $xml->asXml();
40         } elseif ($mediaType == 'application/json') {
41             $app->response()->header('Content-Type', 'app
42             echo json_encode(R::exportAll($article));
43         }
44     } catch (Exception $e) {
45         $app->response()->status(400);
46         $app->response()->header('X-Status-Reason', $e-
47     }
48 });
49
50 // run
51 $app->run();

```

In [Listing 9](#), depending on whether the content type is "application/json" or "application/xml," the request body is converted to a PHP object using `json_decode()` or `simplexml_load_string()`. The object is then saved to the database and a JSON or XML representation of the resource is returned to the client. [Figure 10](#) illustrates the XML POST request and response:

Figure 10. A POST request and response in XML format




Conclusion

Slim provides a powerful, extensible framework for building a REST API, making it easy for application developers to allow third-party access to application functions using an intuitive architectural pattern. Slim's URL matching and routing capabilities, coupled with its lightweight API and support for various HTTP methods, make it ideal for rapid API prototyping and implementation.

See [Downloads](#) for all the code implemented in this article, together with a simple jQuery-based test script that you can use to perform GET, POST, PUT and DELETE requests on the example API. I recommend that you get the code, start playing with it, and maybe try to add new things to it. I guarantee you won't break anything, and it will definitely add to your learning.

Downloadable resources

 [PDF of this content](#)

 [Archive of example application](#) (example-app-slim-rest-api.zip | 45KB)

Related topics

[Slim Framework](#)

[Slim Framework repository](#)

[Rack Middleware Use Case Examples](#)

Comments

 [Subscribe me to comment notifications](#)

Sign in or register to add and subscribe to comments.

Newest

Oldest

Popular



adnansandhila

20 Sep 2017

hi php composer.phar install gives me an error that it could not open the file composer.phar for input, does that file needs to exist before?



StevieG780

19 Mar 2016

On Step 4, Apache no longer needs a you NameVirtualHost

this:

```
NameVirtualHost 127.0.0.1
```

```
<VirtualHost 127.0.0.1>
```

```
DocumentRoot "/usr/local/apache/htdocs/slim"
```

```
ServerName example.localhost
```

```
</VirtualHost>
```

should be this:

```
<VirtualHost 127.0.0.1>
```

```
DocumentRoot "/usr/local/apache/htdocs/slim"
```

```
ServerName example.localhost
```

```
</VirtualHost>
```

and if you're working with Linux AMI

DocumentRoot will be "/home/ec2-user/vendor/slim/slim"



phpsharma

17 Nov 2013

Nice Article..I worked on another Rest frame work called Tonic. I used this knoweldge to

implement redmine in
Tonic.\n\nThanks,\nsharma



davss

01 Jun 2013

Slim is cool and lightweight framework but decided to go for another with even more cleaner syntax and built-in functionalities such as user authentication. Nevertheless it's a great piece of code so thanks for this tutorial - will give it another try one day.\n\nIf you are looking for PHP RESTful frameworks there is a quick list with a poll at the below link so feel free to vote:\n\n<http://davss.com/tech/php-rest-api-frameworks/>



fakeid

05 Jan 2013

Good article!\n\nTried your examples, and Firebug not displaying a JSON tab, reduced to the opening code and listings 1 and 2. Still no JSON tab. Results are displaying unformatted in browser (as HTML 3.2). Tried same browser window with different site using JSON -- tab appeared, so no Firefox\Firebug.\n\nIdeas?



fakeid

16 Dec 2012

A good article about an important Domain-Specific Language (DSL) or micro-framework.\n\nIt is important to note that Slim has been inspired by the Ruby-based DSL Sinatra: <http://www.sinatrarb.com/> In point of fact, to all intents and purposes Slim is a PHP-clone of Sinatra. Which is good, since the Sinatra DSL pattern is well established.\n\nIn Sinatra, one of the strengths is the ease of integrating a database, which means that it, like Slim, is useful for more than just creating static websites or API prototyping.

IBM **Developer**

About

Site Feedback &
FAQ

Select a language

English

中文

Code Patterns

Articles

Tutorials

Videos

Newsletters

Events

[Submit content](#)[Report abuse](#)[Third-party notice](#)[Follow us](#)[日本語](#)[Русский](#)[Português \(Brasil\)](#)[Español](#)[한글](#)[Recipes](#)[Open Source
Projects](#)[Cities](#)[Developer
Answers](#)[Contact](#)[Privacy](#)[Terms of use](#)[Accessibility](#)[Feedback](#)[Cookie Preferences](#)[United States - English](#)