Slim Documentation

Get Started

Installation

System Requirements

- Web server with URL rewriting
- PHP 5.5 or newer

How to Install Slim

We recommend you install Slim with Composer (https://getcomposer.org/). Navigate into your project's root directory and execute the bash command shown below. This command downloads the Slim Framework and its third-party dependencies into your project's `vendor/` directory.

```
composer require slim/slim "^3.0"
```

Require the Composer autoloader into your PHP script, and you are ready to start using Slim.

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

How to Install Composer

Don't have Composer? It's easy to install by following the instructions on their download (https://getcomposer.org/download/) page.

Upgrade Guide

If you are upgrading from version 2 to version 3, these are the significant changes that you need to be aware of.

New PHP version

Slim 3 requires PHP 5.5+

Class \Slim\Slim renamed \Slim\App

Slim 3 uses `\Slim\App` for the Application (/docs/objects/application.html) object usually named `$app` .

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

New Route Function Signature

```
$app->get('/', function (Request $req,  Response $res, $args = []) {
    return $res->withStatus(400)->write('Bad Request');
});
```

Request and response objects are no longer accessible via the Application object

As mentioned above, Slim 3 passes the `Request` and `Response` objects as arguments to the route handling function. Since they are now accessible directly in the body of a route function, `request` and `response` are no longer properties of the `/Slim/App` (Application (/docs/objects/application.html) object) instance.

Getting _GET and _POST variables

```
$app->get('/', function (Request $req,  Response $res, $args = []) {
    $myvar1 = $req->getParam('myvar'); //checks both _GET and _POST [NOT PSR-7 Compliant]
    $myvar2 = $req->getParsedBody()['myvar']; //checks _POST  [IS PSR-7 compliant]
    $myvar3 = $req->getQueryParams()['myvar']; //checks _GET [IS PSR-7 compliant]
});
```

Hooks

Hooks are no longer part of Slim as of v3. You should consider reimplementing any functionality associated with the default hooks in Slim v2 (http://docs.slimframework.com/hooks/defaults/) as middleware (/docs/concepts/middleware.html) instead. If you need the ability to apply custom hooks at arbitrary points in your code (for example, within a route), you should consider a third-party package such as Symfony's EventDispatcher (http://symfony.com/doc/current/components/event_dispatcher/introduction.html) or Zend Framework's EventManager (https://zend-eventmanager.readthedocs.org/en/latest/).

Removal HTTP Cache

In Slim v3 we have removed the HTTP-Caching into its own module Slim\Http\Cache (https://github.com/slimphp/Slim-HttpCache).

Removal of Stop/Halt

Slim Core has removed Stop/Halt. In your applications, you should transition to using the withStatus() and withBody() methods.

Removal of autoloader

`Slim::registerAutoloader()` have been removed, we have fully moved to composer.

Changes to container

`$app->container->singleton(...)` is now `$container = $app->getContainer(); $container['...'] = function () {};` Please read Pimple docs for more info

Removal of configureMode()

`$app->configureMode(...)` has been removed in v3.

Removal of PrettyExceptions

PrettyExceptions cause lots of issues for many people, so these have been removed.

Route::setDefaultConditions(…) has been removed

We have switched routers which enable you to keep the default conditions regex inside of the route pattern.

Changes to redirect

In Slim v2.x one would use the helper function `$app->redirect();` to trigger a redirect request. In Slim v3.x one can do the same with using the Response class like so.

Example:

```
$app->get('/', function ($req, $res, $args) {
  return $res->withStatus(302)->withHeader('Location', 'your-new-uri');
});
```

Middleware Signature

The middleware signature has changed from a class to a function.

New signature:

```
use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

$app->add(function (Request $req,  Response $res, callable $next) {
    // Do stuff before passing along
    $newResponse = $next($req, $res);
    // Do stuff after route is rendered
    return $newResponse; // continue
});
```

You can still use a class:

```
namespace My;

use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

class Middleware
{
    function __invoke(Request $req,  Response $res, callable $next) {
        // Do stuff before passing along
        $newResponse = $next($req, $res);
        // Do stuff after route is rendered
        return $newResponse; // continue
    }
}


// Register
$app->add(new My\Middleware());
// or
$app->add(My\Middleware::class);
```

Middleware Execution

Application middleware is executed as Last In First Executed (LIFE).

Flash Messages

Flash messages are no longer a part of the Slim v3 core but instead have been moved to seperate Slim Flash (/docs/features/flash.html) package.

Cookies

In v3.0 cookies has been removed from core. See FIG Cookies (https://github.com/dflydev/dflydev-fig-cookies) for a PSR-7 compatible cookie component.

Removal of Crypto

In v3.0 we have removed the dependency for crypto in core.

New Router

Slim now utilizes FastRoute (https://github.com/nikic/FastRoute), a new, more powerful router!

This means that the specification of route patterns has changed with named parameters now in braces and square brackets used for optional segments:

```
// named parameter:
$app->get('/hello/{name}', /*...*/);

// optional segment:
$app->get('/news[/{year}]', /*...*/);
```

Route Middleware

The syntax for adding route middleware has changed slightly. In v3.0:

```
$app->get(…)->add($mw2)->add($mw1);
```

Getting the current route

The route is an attribute of the Request object in v3.0:

```
$request->getAttribute('route');
```

When getting the current route in middleware, the value for `determineRouteBeforeAppMiddleware` must be set to `true` in the Application configuration, otherwise the getAttribute call returns `null` .

urlFor() is now pathFor() in the router

`urlFor()` has been renamed `pathFor()` and can be found in the `router` object:

```
$app->get('/', function ($request, $response, $args) {
    $url = $this->router->pathFor('home');
    $response->write("<a href='$url'>Home</a>");
    return $response;
})->setName('home');
```

Also, `pathFor()` is base path aware.

Container and DI … Constructing

Slim uses Pimple as a Dependency Injection Container.

```
// index.php
$app = new Slim\App(
    new \Slim\Container(
        include '../config/container.config.php'
    )
);

// Slim will grab the Home class from the container defined below and execute its index method.
// If the class is not defined in the container Slim will still contruct it and pass the containe
r as the first arugment to the constructor!
$app->get('/', Home::class . ':index');


// In container.config.php
// We are using the SlimTwig here
return [
    'settings' => [
        'viewTemplatesDirectory' => '../templates',
    ],
    'twig' => [
        'title' => '',
        'description' => '',
        'author' => ''
    ],
    'view' => function ($c) {
        $view = new Twig(
            $c['settings']['viewTemplatesDirectory'],
            [
                'cache' => false // '../cache'
            ]
        );

        // Instantiate and add Slim specific extension
        $view->addExtension(
            new TwigExtension(
                $c['router'],
                $c['request']->getUri()
            )
        );

        foreach ($c['twig'] as $name => $value) {
            $view->getEnvironment()->addGlobal($name, $value);
        }

        return $view;
    },
    Home::class => function ($c) {
        return new Home($c['view']);
    }
];
```

PSR-7 Objects

Request, Response, Uri & UploadFile are immutable.

This means that when you change one of these objects, the old instance is not updated.

```
// This is WRONG. The change will not pass through.
$app->add(function (Request $request, Response $response, $next) {
    $request->withAttribute('abc', 'def');
    return $next($request, $response);
});

// This is correct.
$app->add(function (Request $request, Response $response, $next) {
    $request = $request->withAttribute('abc', 'def');
    return $next($request, $response);
});
```

Message bodies are streams

```
// ...
$image = __DIR__ . '/huge_photo.jpg';
$body = new Stream($image);
$response = (new Response())
     ->withStatus(200, 'OK')
     ->withHeader('Content-Type', 'image/jpeg')
     ->withHeader('Content-Length', filesize($image))
     ->withBody($body);
// ...
```

For text:

```
// ...
$response = (new Response())->getBody()->write('Hello world!')

// Or Slim specific: Not PSR-7 compliant.
$response = (new Response())->write('Hello world!');
// ...
```

Web Servers

It is typical to use the front-controller pattern to funnel appropriate HTTP requests received by your web server to a single PHP file. The instructions below explain how to tell your web server to send HTTP requests to your PHP front-controller file.

PHP built-in server

Run the following command in terminal to start localhost web server, assuming `./public/` is public-accessible directory with `index.php` file:

```
php -S localhost:8888 -t public public/index.php
```

If you are not using `index.php` as your entry point then change appropriately.

Apache configuration

Ensure your `.htaccess` and `index.php` files are in the same public-accessible directory. The `.htaccess` file should contain this code:

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

Make sure your Apache virtual host is configured with the `AllowOverride` option so that the `.htaccess` rewrite rules can be used:

```
AllowOverride All
```

Nginx configuration

This is an example Nginx virtual host configuration for the domain `example.com`. It listens for inbound HTTP connections on port 80. It assumes a PHP-FPM server is running on port 9000. You should update the `server_name`, `error_log`, `access_log`, and `root` directives with your own values. The `root` directive is the path to your application's public document root directory; your Slim app's `index.php` front-controller file should be in this directory.

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    error_log /path/to/example.error.log;
    access_log /path/to/example.access.log;
    root /path/to/public;

    location / {
        try_files $uri /index.php$is_args$args;
    }

    location ~ \.php {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
        fastcgi_index index.php;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

HipHop Virtual Machine

Your HipHop Virtual Machine configuration file should contain this code (along with other settings you may need). Be sure you change the `SourceRoot` setting to point to your Slim app's document root directory.

```
Server {
    SourceRoot = /path/to/public/directory
}

ServerVariables {
    SCRIPT_NAME = /index.php
}

VirtualHost {
    * {
        Pattern = .*
        RewriteRules {
                * {
                        pattern = ^(.*)$
                        to = index.php/$1
                        qsa = true
                }
        }
    }
}
```

IIS

Ensure the `Web.config` and `index.php` files are in the same public-accessible directory. The `Web.config` file should contain this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="slim" patternSyntax="Wildcard">
                    <match url="*" />
                    <conditions>
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
                    </conditions>
                    <action type="Rewrite" url="index.php" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

lighttpd

Your lighttpd configuration file should contain this code (along with other settings you may need).

This code requires lighttpd >= 1.4.24.

```
url.rewrite-if-not-file = ("(.*)" => "/index.php/$0")
```

This assumes that Slim's `index.php` is in the root folder of your project (www root).

Deployment

There are many ways to do this that are beyond the scope of this documentation. In this section, we provide some notes for various set-ups.

Disable error display in production

The first thing to do is to tweak your settings (`src/settings.php` in the skeleton application) and ensure that you do not display full error details to the public.

```
  'displayErrorDetails' => false, // set to false in production
```

You should also ensure that your PHP installation is configured to not display errors with the `php.ini` setting:

```
display_errors = 0
```

Deploying to your own server

If you control your server, then you should set up a deployment process using any one of the many deployment system such as:

- Deploybot
- Capistrano
- Script controlled with Phing, Make, Ant, etc.

Review the Web Servers (/docs/start/web-servers.html) documentation to configure your webserver.

Deploying to a shared server

If your shared server runs Apache, then you need to create a `.htaccess` file in your web server root directory (usually named `htdocs`, `public`, `public_html` or `www`) with the following content:

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule ^$ public/     [L]
   RewriteRule (.*) public/$1 [L]
</IfModule>
```

(replace ‘public’ with the correct name of)

Now upload all the files that make up your Slim project to the webserver. As you are on shared hosting, this is probably done via FTP and you can use any FTP client, such as Filezilla to do this.

Tutorial

First Application

If you're looking for a tour through all the ingredients for setting up a very simple Slim application (this one doesn't use Twig, but does use Monolog and a PDO database connection) then you're in the right place. Either walk through the tutorial to build the example application, or adapt each step for your own needs.

Before you start: There is also a skeleton project (https://github.com/slimphp/Slim-Skeleton) which will give you a quick-start for a sample application, so use that if you'd rather just have something working rather than exploring how all the moving parts work.

> *This tutorial walks through building an example application. The code for the application is available (https://github.com/slimphp/Tutor First-Application) if you want to refer to it.*

Getting Set Up

Start by making a folder for your project (mine is called `project`, because naming things is hard). I like to reserve the top level for things-that-are-not-code and then have a folder for source code, and a folder inside that which is my webroot, so my initial structure looks like this:

```
.
├── project
│   └── src
│       └── public
```

Installing Slim Framework

Composer (https://getcomposer.org) is the best way to install Slim Framework. If you don't have it already, you can follow the installation instructions (https://getcomposer.org/download/), in my project I've just downloaded the `composer.phar` into my `src/` directory and I'll use it locally. So my first command looks like this (I'm in the `src/` directory):

```
php composer.phar require slim/slim
```

This does two things:

- Add the Slim Framework dependency to `composer.json` (in my case it creates the file for me as I don't already have one, it's safe to run this if you do already have a `composer.json` file)
- Run `composer install` so that those dependencies are actually available to use in your application

If you look inside the project directory now, you'll see that you have a `vendor/` folder with all the library code in it. There are also two new files: `composer.json` and `composer.lock`. This would be a great time to get our source control setup correct as well: when working with composer, we always exclude the `vendor/` directory, but both `composer.json` and `composer.lock` should be included under source control. Since I'm using `composer.phar` in this directory I'm going to include it in my repo as well; you could equally install the `composer` command on all the systems that need it.

To set up the git ignore correctly, create a file called `src/.gitignore` and add the following single line to the file:

```
vendor/*
```

Now git won't prompt you to add the files in `vendor/` to the repository - we don't want to do this because we're letting composer manage these dependencies rather than including them in our source control repository.

Create The Application

There's a really excellent and minimal example of an `index.php` for Slim Framework on the project homepage (http://www.slimframework.com) so we'll use that as our starting point. Put the following code into `src/public/index.php`:

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Response $response) {
    $name = $request->getAttribute('name');
    $response->getBody()->write("Hello, $name");

    return $response;
});
$app->run();
```

We just pasted a load of code … let's take a look at what it does.

The `use` statements at the top of the script are just bringing the `Request` and `Response` classes into our script so we don't have to refer to them by their long-winded names. Slim framework supports PSR-7 which is the PHP standard for HTTP messaging, so you'll notice as you build your application that the `Request` and `Response` objects are something you see often. This is a modern and excellent approach to writing web applications.

Next we include the `vendor/autoload.php` file - this is created by Composer and allows us to refer to the Slim and other related dependencies we installed earlier. Look out that if you're using the same file structure as me then the `vendor/` directory is one level up from your `index.php` and you may need to adjust the path as I did above.

Finally we create the `$app` object which is the start of the Slim goodness. The `$app->get()` call is our first "route" - when we make a GET request to `/hello/someone` then this is the code that will respond to it. **Don't forget** you need that final `$app->run()` line to tell Slim that we're done configuring and it's time to get on with the main event.

Now we have an application, we'll need to run it. I'll cover two options: the built-in PHP webserver, and an Apache virtual host setup.

Run Your Application With PHP's Webserver

This is my preferred "quick start" option because it doesn't rely on anything else! From the `src/public` directory run the command:

```
php -S localhost:8080
```

This will make your application available at http://localhost:8080 (if you're already using port 8080 on your machine, you'll get a warning. Just pick a different port number, PHP doesn't care what you bind it to).

Note you'll get an error message about "Page Not Found" at this URL - but it's an error message **from** Slim, so this is expected. Try http://localhost:8080/hello/joebloggs instead :)

Run Your Application With Apache or nginx

To get this set up on a standard LAMP stack, we'll need a couple of extra ingredients: some virtual host configuration, and one rewrite rule.

The vhost configuration should be fairly straightforward; we don't need anything special here. Copy your existing default vhost configuration and set the `ServerName` to be how you want to refer to your project. For example you can set:

```
ServerName slimproject.dev

or for nginx:

server_name slimproject.dev;
```

Then you'll also want to set the `DocumentRoot` to point to the `public/` directory of your project, something like this (edit the existing line):

```
DocumentRoot    /home/lorna/projects/slim/project/src/public/

or for nginx:

root    /home/lorna/projects/slim/project/src/public/
```

Don't forget to restart your server process now you've changed the configuration!

I also have a `.htaccess` file in my `src/public` directory; this relies on Apache's rewrite module being enabled and simply makes all web requests go to index.php so that Slim can then handle all the routing for us. Here's my `.htaccess` file:

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule . index.php [L]
```

nginx does not use `.htaccess` files, so you will need to add the following to your server configuration in the `location` block:

```
if (!-e $request_filename){
    rewrite ^(.*)$ /index.php break;
}
```

NOTE: If you want your entry point to be something other than index.php you will need your config to change as well. `api.php` is also commonly used as an entry point, so your set up should match accordingly. This example assumes your are using index.php.

With this setup, just remember to use http://slimproject.dev instead of http://localhost:8080 in the other examples in this tutorial. The same health warning as above applies: you'll see an error page at http://slimproject.dev but crucially it's *Slim's* error page. If you go to http://slimproject.dev/hello/joebloggs then something better should happen.

Configuration and Autoloaders

Now we've set up the platform, we can start getting everything we need in place in the application itself.

Add Config Settings to Your Application

The initial example uses all the Slim defaults, but we can easily add configuration to our application when we create it. There are a few options but here I've just created an array of config options and then told Slim to take its settings from here when I create it.

First the configuration itself:

```
$config['displayErrorDetails'] = true;
$config['addContentLengthHeader'] = false;

$config['db']['host']   = "localhost";
$config['db']['user']   = "user";
$config['db']['pass']   = "password";
$config['db']['dbname'] = "exampleapp";
```

The first line is the most important! Turn this on in development mode to get information about errors (without it, Slim will at least log errors so if you're using the built in PHP webserver then you'll see them in the console output which is helpful). The second line allows the web server to set the Content-Length header which makes Slim behave more predictably.

The other settings here are not specific keys/values, they're just some data that I want to be able to access later.

Now to feed this into Slim, we need to *change* where we create the `Slim/App` object so that it now looks like this:

```
$app = new \Slim\App(["settings" => $config]);
```

We'll be able to access any settings we put into that `$config` array from our application later on.

Set up Autoloading for Your Own Classes

Composer can handle the autoloading of your own classes just as well as the vendored ones. For an in-depth guide, take a look at using Composer to manage autoloading rules (https://getcomposer.org/doc/04-schema.md#autoload).

My setup is pretty simple since I only have a few extra classes, they're just in the global namespace, and the files are in the `src/classes/` directory. So to autoload them, I add this `autoload` section to my `composer.json` file:

```
{
    "require": {
        "slim/slim": "^3.1",
        "slim/php-view": "^2.0",
        "monolog/monolog": "^1.17",
        "robmorgan/phinx": "^0.5.1"
    },
    "autoload": {
        "psr-4": {
            "": "classes/"
        }
    }
}
```

Add Dependencies

Most applications will have some dependencies, and Slim handles them nicely using a DIC (Dependency Injection Container) built on Pimple (http://pimple.sensiolabs.org/). This example will use both Monolog (https://github.com/Seldaek/monolog) and a PDO (http://php.net/manual/en/book.pdo.php) connection to MySQL.

The idea of the dependency injection container is that you configure the container to be able to load the dependencies that your application needs, when it needs them. Once the DIC has created/assembled the dependencies, it stores them and can supply them again later if needed.

To get the container, we can add the following after the line where we create `$app` and before we start to register the routes in our application:

```
$container = $app->getContainer();
```

Now we have the `Slim\Container` object, we can add our services to it.

Use Monolog In Your Application

If you're not already familiar with Monolog, it's an excellent logging framework for PHP applications, which is why I'm going to use it here. First of all, get the Monolog library installed via Composer:

```
php composer.phar require monolog/monolog
```

The dependency is named `logger` and the code to add it looks like this:

```
$container['logger'] = function($c) {
    $logger = new \Monolog\Logger('my_logger');
    $file_handler = new \Monolog\Handler\StreamHandler("../logs/app.log");
    $logger->pushHandler($file_handler);
    return $logger;
};
```

We're adding an element to the container, which is itself an anonymous function (the `$c` that is passed in is the container itself so you can acess other dependencies if you need to). This will be called when we try to access this dependency for the first time; the code here does the setup of the dependency. Next time we try to access the same dependence, the same object that was created the first time will be used the next time.

My Monolog config here is fairly light; just setting up the application to log all errors to a file called `logs/app.log` (remember this path is from the point of view of where the script is running, i.e. `index.php`).

With the logger in place, I can use it from inside my route code with a line like this:

```
    $this->logger->addInfo("Something interesting happened");
```

Having good application logging is a really important foundation for any application so I'd always recommend putting something like this in place. This allows you to add as much or as little debugging as you want, and by using the appropriate log levels with each message, you can have

as much or as little detail as is appropriate for what you're doing in any one moment.

Add A Database Connection

There are many database libraries available for PHP, but this example uses PDO - this is available in PHP as standard so it's probably useful in every project, or you can use your own libraries by adapting the examples below.

Exactly as we did for adding Monolog to the DIC, we'll add an anonymous function that sets up the dependency, in this case called `db`:

```
$container['db'] = function ($c) {
    $db = $c['settings']['db'];
    $pdo = new PDO("mysql:host=" . $db['host'] . ";dbname=" . $db['dbname'],
        $db['user'], $db['pass']);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
    return $pdo;
};
```

Remember the config that we added into our app earlier? Well, this is where we use it - the container knows how to access our settings, and so we can grab our configuration very easily from here. With the config, we create the `PDO` object (remember this will throw a `PDOException` if it fails and you might like to handle that here) so that we can connect to the database. I've included two `setAttribute()` calls that really aren't necessary but I find these two settings make PDO itself much more usable as a library so I left the settings in this example so you can use them too! Finally, we return our connection object.

Again, we can access our dependencies with just `$this->` and then the name of the dependency we want which in this case is `$this->db`, so there is code in my application that looks something like:

```
    $mapper = new TicketMapper($this->db);
```

This will fetch the `db` dependency from the DIC, creating it if necessary, and in this example just allows me to pass the `PDO` object straight into my mapper class.

Create Routes

“Routes” are the URL patterns that we’ll describe and attach functionality to. Slim doesn’t use any automatic mapping or URL formulae so you can make any route pattern you like map onto any function you like, it’s very flexible. Routes can be linked to a particular HTTP verb (such as GET or POST), or more than one verb.

As a first example, here’s the code for making a GET request to `/tickets` which lists the tickets in my bug tracker example application. It just spits out the variables since we haven’t added any views to our application yet:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $mapper = new TicketMapper($this->db);
    $tickets = $mapper->getTickets();

    $response->getBody()->write(var_export($tickets, true));
    return $response;
});
```

The use of `$app->get()` here means that this route is only available for GET requests; there’s an equivalent `$app->post()` call that also takes the route pattern and a callback for POST requests. There are also methods for other verbs (http://www.slimframework.com/docs/objects/router.html) - and also the `map()` function for situations where more than one verb should use the same code for a particular route.

Slim routes match in the order they are declared, so if you have a route which could overlap another route, you need to put the most specific one first. Slim will throw an exception if there’s a problem, for example in this application I have both `/ticket/new` and `/ticket/{id}` and they need to be declared in that order otherwise the routing will think that “new” is an ID!

In this example application, all the routes are in `index.php` but in practice this can make for a rather long and unwieldy file! It’s fine to refactor your application to put routes into a different file or files, or just register a set of routes with callbacks that are actually declared elsewhere.

All route callbacks accept three parameters (the third one is optional):

- Request: this contains all the information about the incoming request, headers, variables,

etc.

- Response: we can add output and headers to this and, once complete, it will be turned into the HTTP response that the client receives
- Arguments: the named placeholders from the URL (more on those in just a moment), this is optional and is usually omitted if there aren't any

This emphasis on Request and Response illustrates Slim 3 being based on the PSR-7 standard for HTTP Messaging. Using the Request and Response object also makes the application more testable as we don't need to make **actual** requests and responses, we can just set up the objects as desired.

Routes with Named Placeholders

Sometimes, our URLs have variables in them that we want to use in our application. In my bug tracking example, I want to have URLs like `/ticket/42` to refer to the ticket - and Slim has an easy way of parsing out the "42" bit and making it available for easy use in the code. Here's the route that does exactly that:

```
$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    $ticket_id = (int)$args['id'];
    $mapper = new TicketMapper($this->db);
    $ticket = $mapper->getTicketById($ticket_id);

    $response->getBody()->write(var_export($ticket, true));
    return $response;
});
```

Look at where the route itself is defined: we write it as `/ticket/{id}` . When we do this, the route will take the portion of the URL from where the `{id}` is declared, and it becomes available as `$args['id']` inside the callback.

Using GET Parameters

Since GET and POST send data in such different ways, then the way that we get that data from the Request object differs hugely in Slim.

It is possible to get all the query parameters from a request by doing `$request->getQueryParams()` which will return an associative array. So for the URL `/tickets?sort=date&order=desc` we'd get an associative array like:

```
["sort" => "date", "order" => "desc"]
```

These can then be used (after validating of course) inside your callback.

Working with POST Data

When working with incoming data, we can find this in the body. We've already seen how we can parse data from the URL and how to obtain the GET variables by doing `$request->getQueryParams()` but what about POST data? The POST request data can be found in the body of the request, and Slim has some good built in helpers to make it easier to get the information in a useful format.

For data that comes from a web form, Slim will turn that into an array. My tickets example application has a form for creating new tickets that just sends two fields: "title" and "description". Here is the first part of the route that receives that data, note that for a POST route use `$app->post()` rather than `$app->get()` :

```
$app->post('/ticket/new', function (Request $request, Response $response) {
    $data = $request->getParsedBody();
    $ticket_data = [];
    $ticket_data['title'] = filter_var($data['title'], FILTER_SANITIZE_STRING);
    $ticket_data['description'] = filter_var($data['description'], FILTER_SANITIZE_STRING);
    // ...
```

The call to `$request->getParsedBody()` asks Slim to look at the request and the `Content-Type` headers of that request, then do something smart and useful with the body. In this example it's just a form post and so the resulting `$data` array looks very similar to what we'd expect from `$_POST` - and we can go ahead and use the filter (http://php.net/manual/en/book.filter.php) extension to check the value is acceptable before we use it. A huge advantage of using the built in Slim methods is that we can test things by injecting different request objects - if we were to use `$_POST` directly, we aren't able to do that.

What's really neat here is that if you're building an API or writing AJAX endpoints, for example, it's super easy to work with data formats that arrive by POST but which aren't a web form. As long as the `Content-Type` header is set correctly, Slim will parse a JSON payload into an array and you can access it exactly the same way: by using `$request->getParsedBody()` .

Views and Templates

Slim doesn't have an opinion on the views that you should use, although there are some options that are ready to plug in. Your best choices are either Twig or plain old PHP. Both options have pros and cons: if you're already familiar with Twig then it offers lots of excellent features and functionality such as layouts - but if you're not already using Twig, it can be a large learning curve overhead to add to a microframework project. If you're looking for something dirt simple then the PHP views might be for you! I picked PHP for this example project, but if you're familiar with Twig then feel free to use that; the basics are mostly the same.

Since we'll be using the PHP views, we'll need to add this dependency to our project via Composer. The command looks like this (similar to the ones you've already seen):

```
php composer.phar require slim/php-view
```

In order to be able to render the view, we'll first need to create a view and make it available to our application; we do that by adding it to the DIC. The code we need goes with the other DIC additions near the top of `src/public/index.php` and it looks like this:

```
$container['view'] = new \Slim\Views\PhpRenderer("../templates/");
```

Now we have a `view` element in the DIC, and by default it will look for its templates in the `src/templates/` directory. We can use it to render templates in our actions - here's the ticket list route again, this time including the call to pass data into the template and render it:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $mapper = new TicketMapper($this->db);
    $tickets = $mapper->getTickets();

    $response = $this->view->render($response, "tickets.phtml", ["tickets" => $tickets]);
    return $response;
});
```

The only new part here is the penultimate line where we set the `$response` variable. Now that the `view` is in the DIC, we can refer to it as `$this->view`. Calling `render()` needs us to supply three arguments: the `$response` to use, the template file (inside the default templates directory), and any data we want to pass in. Response objects are *immutable* which means that the call to `render()` won't update the response object; instead it will return us a new object which is why it needs to be captured like this. This is always true when you operate on the response object.

When passing the data to templates, you can add as many elements to the array as you want to make available in the template. The keys of the array are the variables that the data will exist in once we get to the template itself.

As an example, here's a snippet from the template that displays the ticket list (i.e. the code from `src/templates/tickets.phtml` - which uses Pure.css (http://purecss.io/) to help cover my lack of frontend skills):

```
<h1>All Tickets</h1>

<p><a href="/ticket/new">Add new ticket</a></p>

<table class="pure-table">
    <tr>
        <th>Title</th>
        <th>Component</th>
        <th>Description</th>
        <th>Actions</th>
    </tr>

<?php foreach ($tickets as $ticket): ?>

    <tr>
        <td><?=$ticket->getTitle() ?></td>
        <td><?=$ticket->getComponent() ?></td>
        <td><?=$ticket->getShortDescription() ?> ...</td>
        <td>
            <a href="<?=$router->pathFor('ticket-detail', ['id' => $ticket->getId()])?>">view</a>
        </td>
    </tr>

<?php endforeach; ?>
</table>
```

In this case, `$tickets` is actually a `TicketEntity` class with getters and setters, but if you passed in an array, you'd be able to access it using array rather than object notation here.

Did you notice something fun going on with `$router->pathFor()` right at the end of the example? Let's talk about named routes next :)

Easy URL Building with Named Routes

When we create a route, we can give it a name by calling `->setName()` on the route object. In this case, I am adding the name to the route that lets me view an individual ticket so that I can quickly create the right URL for a ticket by just giving the name of the route, so my code now looks something like this (just the changed bits shown here):

```
$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    // ...
})->setName("ticket-detail");
```

To use this in my template, I need to make the router available in the template that's going to want to create this URL, so I've amended the `tickets/` route to pass a router through to the template by changing the render line to look like this:

```
    $response = $this->view->render($response, "tickets.phtml", ["tickets" => $tickets, "router"
=> $this->router]);
```

With the `/tickets/{id}` route having a friendly name, and the router now available in our template, this is what makes the `pathFor()` call in our template work. By supplying the `id` , this gets used as a named placeholder in the URL pattern, and the correct URL for linking to that route with those values is created. This feature is brilliant for readable template URLs and is even better if you ever need to change a URL format for any reason - no need to grep templates to see where it's used. This approach is definitely recomended, especially for links you'll use a lot.

Where Next?

This article gave a walkthrough of how to get set up with a simple application of your own, which I hope will let you get quickly started, see some working examples, and build something awesome.

From here, I'd recommend you take a look at the other parts of the project documentation for anything you need that wasn't already covered or that you want to see an alternative example of. A great next step would be to take a look at the Middleware (http://www.slimframework.com/docs/concepts/middleware.html) section - this technique is how we layer up our application and add functionality such as authentication which can be applied to multiple routes.

Concepts

PSR 7

Slim supports PSR-7 (https://github.com/php-fig/http-message) interfaces for its Request and Response objects. This makes Slim flexible because it can use *any* PSR-7 implementation. For example, a Slim application route does not *have* to return an instance of `\Slim\Http\Response` . It

could, for example, return an instance of `\GuzzleHttp\Psr7\CachingStream` or any instance returned by the `\GuzzleHttp\Psr7\stream_for()` function.

Slim provides its own PSR-7 implementation so that it works out of the box. However, you are free to replace Slim's default PSR 7 objects with a third-party implementation. Just override the application container's `request` and `response` services so they return an instance of `\Psr\Http\Message\ServerRequestInterface` and `\Psr\Http\Message\ResponseInterface`, respectively.

Value objects

Slim's Request and Response objects are *immutable value objects* (http://en.wikipedia.org/wiki/Value_object). They can be "changed" only by requesting a cloned version that has updated property values. Value objects have a nominal overhead because they must be cloned when their properties are updated. This overhead does not affect performance in any meaningful way.

You can request a copy of a value object by invoking any of its PSR 7 interface methods (these methods typically have a `with` prefix). For example, a PSR 7 Response object has a `withHeader($name, $value)` method that returns a cloned value object with the new HTTP header.

```
<?php
$app = new \Slim\App;
$app->get('/foo', function ($req, $res, $args) {
    return $res->withHeader(
        'Content-Type',
        'application/json'
    );
});
$app->run();
```

The PSR 7 interface provides these methods to transform Request and Response objects:

- `withProtocolVersion($version)`
- `withHeader($name, $value)`
- `withAddedHeader($name, $value)`
- `withoutHeader($name)`
- `withBody(StreamInterface $body)`

The PSR 7 interface provides these methods to transform Request objects:

- `withMethod($method)`
- `withUri(UriInterface $uri, $preserveHost = false)`
- `withCookieParams(array $cookies)`
- `withQueryParams(array $query)`
- `withUploadedFiles(array $uploadedFiles)`
- `withParsedBody($data)`
- `withAttribute($name, $value)`
- `withoutAttribute($name)`

The PSR 7 interface provides these methods to transform Response objects:

- `withStatus($code, $reasonPhrase = '')`

Refer to the PSR-7 documentation (http://www.php-fig.org/psr/psr-7/) for more information about these methods.

Middleware

You can run code *before* and *after* your Slim application to manipulate the Request and Response objects as you see fit. This is called *middleware*. Why would you want to do this? Perhaps you want to protect your app from cross-site request forgery. Maybe you want to authenticate requests before your app runs. Middleware is perfect for these scenarios.

What is middleware?

Technically speaking, a middleware is a *callable* that accepts three arguments:

1. `\Psr\Http\Message\ServerRequestInterface` - The PSR7 request object
2. `\Psr\Http\Message\ResponseInterface` - The PSR7 response object
3. `callable` - The next middleware callable

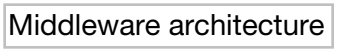
It can do whatever is appropriate with these objects. The only hard requirement is that a middleware **MUST** return an instance of `\Psr\Http\Message\ResponseInterface`. Each middleware **SHOULD** invoke the next middleware and pass it Request and Response objects as arguments.

How does middleware work?

Different frameworks use middleware differently. Slim adds middleware as concentric layers surrounding your core application. Each new middleware layer surrounds any existing middleware layers. The concentric structure expands outwardly as additional middleware layers are added.

The last middleware layer added is the first to be executed.

When you run the Slim application, the Request and Response objects traverse the middleware structure from the outside in. They first enter the outer-most middleware, then the next outer-most middleware, (and so on), until they ultimately arrive at the Slim application itself. After the Slim application dispatches the appropriate route, the resultant Response object exits the Slim application and traverses the middleware structure from the inside out. Ultimately, a final Response object exits the outer-most middleware, is serialized into a raw HTTP response, and is returned to the HTTP client. Here's a diagram that illustrates the middleware process flow:



How do I write middleware?

Middleware is a callable that accepts three arguments: a Request object, a Response object, and the next middleware. Each middleware **MUST** return an instance of

`\Psr\Http\Message\ResponseInterface` .

Closure middleware example.

This example middleware is a Closure.

```php
<?php
/**
 * Example middleware closure
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $request  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $response PSR7 response
 * @param  callable                                 $next     Next middleware
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
};
```

Invokable class middleware example

This example middleware is an invokable class that implements the magic `__invoke()` method.

```php
<?php
class ExampleMiddleware
{
    /**
     * Example middleware invokable class
     *
     * @param  \Psr\Http\Message\ServerRequestInterface $request  PSR7 request
     * @param  \Psr\Http\Message\ResponseInterface      $response PSR7 response
     * @param  callable                                 $next     Next middleware
     *
     * @return \Psr\Http\Message\ResponseInterface
     */
    public function __invoke($request, $response, $next)
    {
        $response->getBody()->write('BEFORE');
        $response = $next($request, $response);
        $response->getBody()->write('AFTER');

        return $response;
    }
}
```

To use this class as a middleware, you can use `->add( new ExampleMiddleware() );` function chain after the `$app`, `Route`, or `group()`, which in the code below, any one of these, could represent $subject.

```
$subject->add( new ExampleMiddleware() );
```

How do I add middleware?

You may add middleware to a Slim application, to an individual Slim application route or to a route group. All scenarios accept the same middleware and implement the same middleware interface.

Application middleware

Application middleware is invoked for every *incoming* HTTP request. Add application middleware with the Slim application instance's `add()` method. This example adds the Closure middleware example above:

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

$app->add(function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
});

$app->get('/', function ($request, $response, $args) {
    $response->getBody()->write(' Hello ');

    return $response;
});

$app->run();
```

This would output this HTTP response body:

```
BEFORE Hello AFTER
```

Route middleware

Route middleware is invoked *only if* its route matches the current HTTP request method and URI. Route middleware is specified immediately after you invoke any of the Slim application's routing methods (e.g., `get()` or `post()`). Each routing method returns an instance of `\Slim\Route`, and

this class provides the same middleware interface as the Slim application instance. Add middleware to a Route with the Route instance's `add()` method. This example adds the Closure middleware example above:

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

$mw = function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
};

$app->get('/', function ($request, $response, $args) {
    $response->getBody()->write(' Hello ');

    return $response;
})->add($mw);

$app->run();
```

This would output this HTTP response body:

```
BEFORE Hello AFTER
```

Group Middleware

In addition to the overall application, and standard routes being able to accept middleware, the `group()` multi-route definition functionality, also allows individual routes internally. Route group middleware is invoked *only if* its route matches one of the defined HTTP request methods and URIs from the group. To add middleware within the callback, and entire-group middleware to be set by chaining `add()` after the `group()` method.

Sample Application, making use of callback middleware on a group of url-handlers

```
<?php

require_once __DIR__.'/vendor/autoload.php';

$app = new \Slim\App();

$app->get('/', function ($request, $response) {
    return $response->getBody()->write('Hello World');
});

$app->group('/utils', function () use ($app) {
    $app->get('/date', function ($request, $response) {
        return $response->getBody()->write(date('Y-m-d H:i:s'));
    });
    $app->get('/time', function ($request, $response) {
        return $response->getBody()->write(time());
    });
})->add(function ($request, $response, $next) {
    $response->getBody()->write('It is now ');
    $response = $next($request, $response);
    $response->getBody()->write('. Enjoy!');

    return $response;
});
```

When calling the `/utils/date` method, this would output a string similar to the below

```
It is now 2015-07-06 03:11:01. Enjoy!
```

visiting `/utils/time` would output a string similar to the below

```
It is now 1436148762. Enjoy!
```

but visiting `/` *(domain-root)*, would be expected to generate the following output as no middleware has been assigned

```
Hello World
```

Passing variables from middleware

The easiest way to pass attributes from middleware is to use the request's attributes.

Setting the variable in the middleware:

```
$request = $request->withAttribute('foo', 'bar');
```

Getting the variable in the route callback:

```
$foo = $request->getAttribute('foo');
```

Dependency Container

Slim uses a dependency container to prepare, manage, and inject application dependencies. Slim supports containers that implement PSR-11 (http://www.php-fig.org/psr/psr-11/) or the Container-Interop (https://github.com/container-interop/container-interop) interface. You can use Slim's built-in container (based on Pimple (http://pimple.sensiolabs.org/)) or third-party containers like Acclimate (https://github.com/jeremeamia/acclimate-container) or PHP-DI (http://php-di.org/doc/frameworks/slim.html).

How to use the container

You don't *have* to provide a dependency container. If you do, however, you must inject the container instance into the Slim application's constructor.

```
$container = new \Slim\Container;
$app = new \Slim\App($container);
```

Add a service to Slim container:

```
$container = $app->getContainer();
$container['myService'] = function ($container) {
    $myService = new MyService();
    return $myService;
};
```

You can fetch services from your container explicitly or implicitly. You can fetch an explicit reference to the container instance from inside a Slim application route like this:

```
/**
 * Example GET route
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $req  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $res  PSR7 response
 * @param  array                                   $args Route parameters
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
$app->get('/foo', function ($req, $res, $args) {
    $myService = $this->get('myService');

    return $res;
});
```

You can implicitly fetch services from the container like this:

```
/**
 * Example GET route
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $req  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $res  PSR7 response
 * @param  array                                   $args Route parameters
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
$app->get('/foo', function ($req, $res, $args) {
    $myService = $this->myService;

    return $res;
});
```

To test if a service exists in the container before using it, use the `has()` method, like this:

```
/**
 * Example GET route
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $req  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $res  PSR7 response
 * @param  array                                   $args Route parameters
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
$app->get('/foo', function ($req, $res, $args) {
    if($this->has('myService')) {
        $myService = $this->myService;
    }

    return $res;
});
```

Slim uses `__get()` and `__isset()` magic methods that defer to the application's container for all properties that do not already exist on the application instance.

Required services

Your container MUST implement these required services. If you use Slim's built-in container, these are provided for you. If you choose a third-party container, you must define these required services on your own.

settings
Associative array of application settings, including keys:

- `httpVersion`
- `responseChunkSize`
- `outputBuffering`
- `determineRouteBeforeAppMiddleware` .
- `displayErrorDetails` .
- `addContentLengthHeader` .
- `routerCacheFile` .

environment
Instance of `\Slim\Interfaces\Http\EnvironmentInterface` .

request
Instance of `\Psr\Http\Message\ServerRequestInterface` .

response
Instance of `\Psr\Http\Message\ResponseInterface` .

router
Instance of `\Slim\Interfaces\RouterInterface` .

foundHandler
Instance of `\Slim\Interfaces\InvocationStrategyInterface` .

phpErrorHandler
Callable invoked if a PHP 7 Error is thrown. The callable **MUST** return an instance of `\Psr\Http\Message\ResponseInterface` and accept three arguments:

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. `\Error`

errorHandler
Callable invoked if an Exception is thrown. The callable **MUST** return an instance of `\Psr\Http\Message\ResponseInterface` and accept three arguments:

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. `\Exception`

notFoundHandler
Callable invoked if the current HTTP request URI does not match an application route. The callable **MUST** return an instance of `\Psr\Http\Message\ResponseInterface` and accept two arguments:

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`

notAllowedHandler
Callable invoked if an application route matches the current HTTP request path but not its method. The callable **MUST** return an instance of `\Psr\Http\Message\ResponseInterface` and accept three arguments:

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. Array of allowed HTTP methods

callableResolver
Instance of `\Slim\Interfaces\CallableResolverInterface`.

The Application

Overview

The Application, (or `Slim\App`) is the entry point to your Slim application and is used to register the routes that link to your callbacks or controllers.

```
// instantiate the App object
$app = new \Slim\App();

// Add route callbacks
$app->get('/', function ($request, $response, $args) {
    return $response->withStatus(200)->write('Hello World!');
});

// Run application
$app->run();
```

Application Configuration

The Application accepts just one argument. This can be either a Container (/docs/concepts/di.html) instance or an array to configure the default container that is created automatically.

There are also a number of settings that are used by Slim. These are stored in the `settings` configuration key. You can also add your application-specific settings.

For example, we can set the Slim setting `displayErrorDetails` to true and also configure Monolog like this:

```
$config = [
    'settings' => [
        'displayErrorDetails' => true,

        'logger' => [
            'name' => 'slim-app',
            'level' => Monolog\Logger::DEBUG,
            'path' => __DIR__ . '/../logs/app.log',
        ],
    ],
];
$app = new \Slim\App($config);
```

Retrieving Settings

As the settings are stored in the DI container so you can access them via the `settings` key in container factories. For example:

```
$loggerSettings = $container->get('settings')['logger'];
```

You can also access them in route callables via `$this`:

```
$app->get('/', function ($request, $response, $args) {
    $loggerSettings = $this->get('settings')['logger'];
    // ...
});
```

Updating Settings

If you need to add or update settings stored in the DI container *after* the container is initialized, you can use the `replace` method on the settings container. For example:

```
$settings = $container->get('settings');
$settings->replace([
        'displayErrorDetails' => true,
        'determineRouteBeforeAppMiddleware' => true,
        'debug' => true
    ]);
]);
```

Slim Default Settings

Slim has the following default settings that you can override:

`httpVersion`
The protocol version used by the Response (/docs/objects/response.html) object.
(Default: `'1.1'`)

`responseChunkSize`
Size of each chunk read from the Response body when sending to the browser.
(Default: `4096`)

`outputBuffering`
If `false`, then no output buffering is enabled. If `'append'` or `'prepend'`, then any `echo` or `print` statements are captured and are either appended or prepended to the Response returned from the route callable.
(Default: `'append'`)

`determineRouteBeforeAppMiddleware`
When true, the route is calculated before any middleware is executed. This means that you can inspect route parameters in middleware if you need to.
(Default: `false`)

`displayErrorDetails`
When true, additional information about exceptions are displayed by the default error handler (/docs/handlers/error.html).
(Default: `false`)

`addContentLengthHeader`
When true, Slim will add a `Content-Length` header to the response. If you are using a runtime analytics tool, such as New Relic, then this should be disabled.
(Default: `true`)

`routerCacheFile`
Filename for caching the FastRoute routes. Must be set to to a valid filename within a writeable directory. If the file does not exist, then it is created with the correct cache information on first

run.
Set to `false` to disable the FastRoute cache system.
(Default: `false`)

The Request

Overview

Your Slim app's routes and middleware are given a PSR 7 request object that represents the current HTTP request received by your web server. The request object implements the PSR 7 ServerRequestInterface (http://www.php-fig.org/psr/psr-7/#3-2-1-psr-http-message-serverrequestinterface) with which you can inspect and manipulate the HTTP request method, headers, and body.

How to get the Request object

The PSR 7 request object is injected into your Slim application routes as the first argument to the route callback like this:

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->get('/foo', function (ServerRequestInterface $request, ResponseInterface $response) {
    // Use the PSR 7 $request object

    return $response;
});
$app->run();
```

Figure 1: Inject PSR 7 request into application route callback.

The PSR 7 request object is injected into your Slim application *middleware* as the first argument of the middleware callable like this:

```php
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->add(function (ServerRequestInterface $request, ResponseInterface $response, callable $next)
{
    // Use the PSR 7 $request object

    return $next($request, $response);
});
// Define app routes...
$app->run();
```

Figure 2: Inject PSR 7 request into application middleware.

The Request Method

Every HTTP request has a method that is typically one of:

- GET
- POST
- PUT
- DELETE
- HEAD
- PATCH
- OPTIONS

You can inspect the HTTP request's method with the Request object method appropriately named `getMethod()` .

```php
$method = $request->getMethod();
```

Because this is a common task, Slim's built-in PSR 7 implementation also provides these proprietary methods that return `true` or `false` .

- `$request->isGet()`

- `$request->isPost()`
- `$request->isPut()`
- `$request->isDelete()`
- `$request->isHead()`
- `$request->isPatch()`
- `$request->isOptions()`

It is possible to fake or *override* the HTTP request method. This is useful if, for example, you need to mimic a `PUT` request using a traditional web browser that only supports `GET` or `POST` requests.

There are two ways to override the HTTP request method. You can include a `_METHOD` parameter in a `POST` request's body. The HTTP request must use the `application/x-www-form-urlencoded` content type.

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/x-www-form-urlencoded
Content-length: 22

data=value&_METHOD=PUT
```

Figure 3: Override HTTP method with _METHOD parameter.

You can also override the HTTP request method with a custom `X-Http-Method-Override` HTTP request header. This works with any HTTP request content type.

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/json
Content-length: 16
X-Http-Method-Override: PUT

{"data":"value"}
```

Figure 4: Override HTTP method with X-Http-Method-Override header.

You can fetch the *original* (non-overridden) HTTP method with the PSR 7 Request object's method named `getOriginalMethod()`.

The Request URI

Every HTTP request has a URI that identifies the requested application resource. The HTTP request URI has several parts:

- Scheme (e.g. `http` or `https`)
- Host (e.g. `example.com`)
- Port (e.g. `80` or `443`)
- Path (e.g. `/users/1`)
- Query string (e.g. `sort=created&dir=asc`)

You can fetch the PSR 7 Request object's URI object (http://www.php-fig.org/psr/psr-7/#3-5-psr-http-message-uriinterface) with its `getUri()` method:

```
$uri = $request->getUri();
```

The PSR 7 Request object's URI is itself an object that provides the following methods to inspect the HTTP request's URL parts:

- `getScheme()`
- `getAuthority()`
- `getUserInfo()`
- `getHost()`
- `getPort()`
- `getPath()`
- `getBasePath()`
- `getQuery()` (returns the full query string, e.g. `a=1&b=2`)
- `getFragment()`
- `getBaseUrl()`

You can get the query parameters as an associative array on the Request object using `getQueryParams()`.

You can also get a single query parameter value, with optional default value if the parameter is missing, using `getQueryParam($key, $default = null)`.

> **Base Path**
>
> If your Slim application's front-controller lives in a physical subdirectory beneath your document root directory, you can fetch the HTTP request's physical base path (relative to the document root) with the Uri object's `getBasePath()` method. This will be an empty string if the Slim application is installed in the document root's top-most directory.

The Request Headers

Every HTTP request has headers. These are metadata that describe the HTTP request but are not visible in the request's body. Slim's PSR 7 Request object provides several methods to inspect its headers.

Get All Headers

You can fetch all HTTP request headers as an associative array with the PSR 7 Request object's `getHeaders()` method. The resultant associative array's keys are the header names and its values are themselves a numeric array of string values for their respective header name.

```
$headers = $request->getHeaders();
foreach ($headers as $name => $values) {
    echo $name . ": " . implode(", ", $values);
}
```

Figure 5: Fetch and iterate all HTTP request headers as an associative array.

Get One Header

You can get a single header's value(s) with the PSR 7 Request object's `getHeader($name)` method. This returns an array of values for the given header name. Remember, *a single HTTP header may have more than one value!*

```
$headerValueArray = $request->getHeader('Accept');
```

Figure 6: Get values for a specific HTTP header.

You may also fetch a comma-separated string with all values for a given header with the PSR 7 Request object's `getHeaderLine($name)` method. Unlike the `getHeader($name)` method, this method returns a comma-separated string.

```
$headerValueString = $request->getHeaderLine('Accept');
```

Figure 7: Get single header's values as comma-separated string.

Detect Header

You can test for the presence of a header with the PSR 7 Request object's `hasHeader($name)` method.

```
if ($request->hasHeader('Accept')) {
    // Do something
}
```

Figure 8: Detect presence of a specific HTTP request header.

The Request Body

Every HTTP request has a body. If you are building a Slim application that consumes JSON or XML data, you can use the PSR 7 Request object's `getParsedBody()` method to parse the HTTP request body into a native PHP format. Slim can parse JSON, XML, and URL-encoded data out of the box.

```
$parsedBody = $request->getParsedBody();
```

Figure 9: Parse HTTP request body into native PHP format

- JSON requests are converted into associative arrays with `json_decode($input, true)`.
- XML requests are converted into a `SimpleXMLElement` with `simplexml_load_string($input)`.
- URL-encoded requests are converted into a PHP array with `parse_str($input)`.

For URL-encoded requests, you can also get a single parameter value, with optional default value if the parameter is missing, using `getParsedBodyParam($key, $default = null)`.

Technically speaking, Slim's PSR 7 Request object represents the HTTP request body as an instance of `\Psr\Http\Message\StreamInterface`. You can get the HTTP request body `StreamInterface` instance with the PSR 7 Request object's `getBody()` method. The `getBody()` method is preferable if the incoming HTTP request size is unknown or too large for available memory.

```
$body = $request->getBody();
```

Figure 10: Get HTTP request body

The resultant `\Psr\Http\Message\StreamInterface` instance provides the following methods to read and iterate its underlying PHP `resource`.

- `getSize()`
- `tell()`
- `eof()`
- `isSeekable()`

- `seek()`
- `rewind()`
- `isWritable()`
- `write($string)`
- `isReadable()`
- `read($length)`
- `getContents()`
- `getMetadata($key = null)`

Reparsing the body

When calling `getParsedBody` on the Request object multiple times, the body is only parsed once, even if the Request body is modified in the meantime.

To ensure the body is reparsed, the Request object's method `reparseBody` can be used.

Uploaded Files

The file uploads in `$_FILES` are available from the Request object's `getUploadedFiles()` method. This returns an array keyed by the name of the `<input>` element.

```
$files = $request->getUploadedFiles();
```

Figure 11: Get uploaded files

Each object in the `$files` array is a instance of `\Psr\Http\Message\UploadedFileInterface` and supports the following methods:

- `getStream()`
- `moveTo($targetPath)`
- `getSize()`
- `getError()`
- `getClientFilename()`

- `getClientMediaType()`

See the cookbook (/docs/cookbook/uploading-files.html) on how to upload files using a POST form.

Request Helpers

Slim's PSR 7 Request implementation provides these additional proprietary methods to help you further inspect the HTTP request.

Detect XHR requests

You can detect XHR requests with the Request object's `isXhr()` method. This method detects the presence of the `X-Requested-With` HTTP request header and ensures its value is `XMLHttpRequest`.

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/x-www-form-urlencoded
Content-length: 7
X-Requested-With: XMLHttpRequest

foo=bar
```

Figure 13: Example XHR request.

```
if ($request->isXhr()) {
    // Do something
}
```

Content Type

You can fetch the HTTP request content type with the Request object's `getContentType()` method. This returns the `Content-Type` header's full value as provided by the HTTP client.

```
$contentType = $request->getContentType();
```

Media Type

You may not want the complete `Content-Type` header. What if, instead, you only want the media type? You can fetch the HTTP request media type with the Request object's `getMediaType()` method.

```
$mediaType = $request->getMediaType();
```

You can fetch the appended media type parameters as an associative array with the Request object's `getMediaTypeParams()` method.

```
$mediaParams = $request->getMediaTypeParams();
```

Character Set

One of the most common media type parameters is the HTTP request character set. The Request object provides a dedicated method to retrieve this media type parameter.

```
$charset = $request->getContentCharset();
```

Content Length

You can fetch the HTTP request content length with the Request object's `getContentLength()` method.

```
$length = $request->getContentLength();
```

Request Parameter

To fetch single request parameter value, use methods: `getParam()`, `getQueryParam()`, `getParsedBodyParam()`, `getCookieParam()`, `getServerParam()`, counterparts of PSR-7's plural form get*Params() methods.

For example, to get a single Server Parameter:

```
$foo = $request->getServerParam('HTTP_NOT_EXIST', 'default_value_here');
```

Route Object

Sometimes in middleware you require the parameter of your route.

In this example we are checking first that the user is logged in and second that the user has permissions to view the particular video they are attempting to view.

```
    $app->get('/course/{id}', Video::class.":watch")->add(Permission::class)->add(Auth::class);

    //.. In the Permission Class's Invoke
    /** @var $route \Slim\Route */
    $route = $request->getAttribute('route');
    $courseId = $route->getArgument('id');
```

Media Type Parsers

Slim looks as the request's media type and if it recognises it, will parse it into structured data available via `$request->getParsedBody()`. This is usually an array, but is an object for XML media types.

The following media types are recognised and parsed:

- application/x-www-form-urlencoded
- application/json
- application/xml & text/xml

If you want Slim to parse content from a different media type then you need to either parse the raw body yourself or register a new media parser. Media parsers are simply callables that accept an `$input` string and return a parsed object or array.

Register a new media parser in an application or route middleware. Note that you must register the parser before you try to access the parsed body for the first time.

For example, to automatically parse JSON that is sent with a `text/javascript` content type, you register a media type parser in middleware like this:

```
// Add the middleware
$app->add(function ($request, $response, $next) {
    // add media parser
    $request->registerMediaTypeParser(
        "text/javascript",
        function ($input) {
            return json_decode($input, true);
        }
    );

    return $next($request, $response);
});
```

Attributes

With PSR-7 it is possible to inject objects/values into the request object for further processing. In your applications middleware often need to pass along information to your route closure and the way to do is it is to add it to the request object via an attribute.

Example, Setting a value on your request object.

```
$app->add(function ($request, $response, $next) {
    $request = $request->withAttribute('session', $_SESSION); //add the session storage to your r
equest as [READ-ONLY]
    return $next($request, $response);
});
```

Example, how to retrieve the value.

```
$app->get('/test', function ($request, $response, $args) {
    $session = $request->getAttribute('session'); //get the session from the request

    return $response->write('Yay, ' . $session['name']);
});
```

The request object also has bulk functions as well. `$request->getAttributes()` and `$request->withAttributes()`

The Response

Overview

Your Slim app's routes and middleware are given a PSR 7 response object that represents the current HTTP response to be returned to the client. The response object implements the PSR 7 ResponseInterface (http://www.php-fig.org/psr/psr-7/#3-2-1-psr-http-message-responseinterface) with which you can inspect and manipulate the HTTP response status, headers, and body.

How to get the Response object

The PSR 7 response object is injected into your Slim application routes as the second argument to the route callback like this:

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->get('/foo', function (ServerRequestInterface $request, ResponseInterface $response) {
    // Use the PSR 7 $response object

    return $response;
});
$app->run();
```

Figure 1: Inject PSR 7 response into application route callback.

The PSR 7 response object is injected into your Slim application *middleware* as the second argument of the middleware callable like this:

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->add(function (ServerRequestInterface $request, ResponseInterface $response, callable $next)
{
    // Use the PSR 7 $response object

    return $next($request, $response);
});
// Define app routes...
$app->run();
```

Figure 2: Inject PSR 7 response into application middleware.

The Response Status

Every HTTP response has a numeric status code (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html). The status code identifies the *type* of HTTP response to be returned to the client. The PSR 7 Response object's default status code is `200` (OK). You can get the PSR 7 Response object's status code with the `getStatusCode()` method like this.

```
$status = $response->getStatusCode();
```

Figure 3: Get response status code.

You can copy a PSR 7 Response object and assign a new status code like this:

```
$newResponse = $response->withStatus(302);
```

Figure 4: Create response with new status code.

The Response Headers

Every HTTP response has headers. These are metadata that describe the HTTP response but are not visible in the response's body. Slim's PSR 7 Response object provides several methods to inspect and manipulate its headers.

Get All Headers

You can fetch all HTTP response headers as an associative array with the PSR 7 Response object's `getHeaders()` method. The resultant associative array's keys are the header names and its values are themselves a numeric array of string values for their respective header name.

```
$headers = $response->getHeaders();
foreach ($headers as $name => $values) {
    echo $name . ": " . implode(", ", $values);
}
```

Figure 5: Fetch and iterate all HTTP response headers as an associative array.

Get One Header

You can get a single header's value(s) with the PSR 7 Response object's `getHeader($name)` method. This returns an array of values for the given header name. Remember, *a single HTTP header may have more than one value!*

```
$headerValueArray = $response->getHeader('Vary');
```

Figure 6: Get values for a specific HTTP header.

You may also fetch a comma-separated string with all values for a given header with the PSR 7 Response object's `getHeaderLine($name)` method. Unlike the `getHeader($name)` method, this method returns a comma-separated string.

```
$headerValueString = $response->getHeaderLine('Vary');
```

Figure 7: Get single header's values as comma-separated string.

Detect Header

You can test for the presence of a header with the PSR 7 Response object's `hasHeader($name)` method.

```
if ($response->hasHeader('Vary')) {
    // Do something
}
```

Figure 8: Detect presence of a specific HTTP header.

Set Header

You can set a header value with the PSR 7 Response object's `withHeader($name, $value)` method.

```
$newResponse = $oldResponse->withHeader('Content-type', 'application/json');
```

Figure 9: Set HTTP header

> **Reminder**
>
> The Response object is immutable. This method returns a *copy* of the Response object that has the new header value. **This method is destructive**, and it *replaces* existing header values already associated with the same header name.

Append Header

You can append a header value with the PSR 7 Response object's `withAddedHeader($name, $value)` method.

```
$newResponse = $oldResponse->withAddedHeader('Allow', 'PUT');
```

Figure 10: Append HTTP header

> **Reminder**
>
> Unlike the `withHeader()` method, this method *appends* the new value to the set of values that already exist for the same header name. The Response object is immutable. This method returns a *copy* of the Response object that has the appended header value.

Remove Header

You can remove a header with the Response object's `withoutHeader($name)` method.

```
$newResponse = $oldResponse->withoutHeader('Allow');
```

Figure 11: Remove HTTP header

> **Reminder**
>
> The Response object is immutable. This method returns a *copy* of the Response object that has the appended header value.

The Response Body

An HTTP response typically has a body. Slim provides a PSR 7 Response object with which you can inspect and manipulate the eventual HTTP response's body.

Just like the PSR 7 Request object, the PSR 7 Response object implements the body as an instance of `\Psr\Http\Message\StreamInterface`. You can get the HTTP response body `StreamInterface` instance with the PSR 7 Response object's `getBody()` method. The `getBody()`

method is preferable if the outgoing HTTP response length is unknown or too large for available memory.

```
$body = $response->getBody();
```

Figure 12: Get HTTP response body

The resultant `\Psr\Http\Message\StreamInterface` instance provides the following methods to read from, iterate, and write to its underlying PHP `resource`.

- `getSize()`
- `tell()`
- `eof()`
- `isSeekable()`
- `seek()`
- `rewind()`
- `isWritable()`
- `write($string)`
- `isReadable()`
- `read($length)`
- `getContents()`
- `getMetadata($key = null)`

Most often, you'll need to write to the PSR 7 Response object. You can write content to the `StreamInterface` instance with its `write()` method like this:

```
$body = $response->getBody();
$body->write('Hello');
```

Figure 13: Write content to the HTTP response body

You can also *replace* the PSR 7 Response object's body with an entirely new `StreamInterface`

instance. This is particularly useful when you want to pipe content from a remote destination (e.g. the filesystem or a remote API) into the HTTP response. You can replace the PSR 7 Response object's body with its `withBody(StreamInterface $body)` method. Its argument **MUST** be an instance of `\Psr\Http\Message\StreamInterface`.

```
$newStream = new \GuzzleHttp\Psr7\LazyOpenStream('/path/to/file', 'r');
$newResponse = $oldResponse->withBody($newStream);
```

Figure 14: Replace the HTTP response body

> **Reminder**
>
> The Response object is immutable. This method returns a *copy* of the Response object that contains the new body.

Returning JSON

Slim's Response object has a custom method `withJson($data, $status, $encodingOptions)` to help simplify the process of returning JSON data.

The `$data` parameter contains the data structure you wish returned as JSON. `$status` is optional, and can be used to return a custom HTTP code. `$encodingOptions` is optional, and are the same encoding options used for `json_encode()` (http://php.net/manual/en/function.json-encode.php).

In it's simplest form, JSON data can be returned with a default 200 HTTP status code.

```
$data = array('name' => 'Bob', 'age' => 40);
$newResponse = $oldResponse->withJson($data);
```

Figure 15: Returning JSON with a 200 HTTP status code.

We can also return JSON data with a custom HTTP status code.

```
$data = array('name' => 'Rob', 'age' => 40);
$newResponse = $oldResponse->withJson($data, 201);
```

Figure 16: Returning JSON with a 201 HTTP status code.

The `Content-Type` of the Response is automatically set to `application/json;charset=utf-8`.

If there is a problem encoding the data to JSON, a `\RuntimeException($message, $code)` is thrown containing the values of `json_last_error_msg()` (http://php.net/manual/en/function.json-last-error-msg.php) as the `$message` and `json_last_error()` (http://php.net/manual/en/function.json-last-error.php) as the `$code`.

Reminder

The Response object is immutable. This method returns a *copy* of the Response object that has a new Content-Type header. **This method is destructive**, and it *replaces* the existing Content-Type header. The Status is also replaced if a $status was passed when `withJson()` was called.

Routing

Overview

The Slim Framework's router is built on top of the nikic/fastroute (https://github.com/nikic/FastRoute) component, and it is remarkably fast and stable.

How to create routes

You can define application routes using proxy methods on the `\Slim\App` instance. The Slim Framework provides methods for the most popular HTTP methods.

GET Route

You can add a route that handles only `GET` HTTP requests with the Slim application's `get()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->get('/books/{id}', function ($request, $response, $args) {
    // Show book identified by $args['id']
});
```

POST Route

You can add a route that handles only `POST` HTTP requests with the Slim application's `post()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->post('/books', function ($request, $response, $args) {
    // Create new book
});
```

PUT Route

You can add a route that handles only `PUT` HTTP requests with the Slim application's `put()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->put('/books/{id}', function ($request, $response, $args) {
    // Update book identified by $args['id']
});
```

DELETE Route

You can add a route that handles only `DELETE` HTTP requests with the Slim application's `delete()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->delete('/books/{id}', function ($request, $response, $args) {
    // Delete book identified by $args['id']
});
```

OPTIONS Route

You can add a route that handles only `OPTIONS` HTTP requests with the Slim application's `options()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->options('/books/{id}', function ($request, $response, $args) {
    // Return response headers
});
```

PATCH Route

You can add a route that handles only `PATCH` HTTP requests with the Slim application's `patch()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->patch('/books/{id}', function ($request, $response, $args) {
    // Apply changes to book identified by $args['id']
});
```

Any Route

You can add a route that handles all HTTP request methods with the Slim application's `any()` method. It accepts two arguments:

1. The route pattern (with optional named placeholders)
2. The route callback

```
$app = new \Slim\App();
$app->any('/books/[{id}]', function ($request, $response, $args) {
    // Apply changes to books or book identified by $args['id'] if specified.
    // To check which method is used: $request->getMethod();
});
```

Note that the second parameter is a callback. You could specify a Class (which need a `__invoke()` implementation) instead of a Closure. You can then do the mapping somewhere else:

```
$app->any('/user', 'MyRestfulController');
```

Custom Route

You can add a route that handles multiple HTTP request methods with the Slim application's `map()` method. It accepts three arguments:

1. Array of HTTP methods
2. The route pattern (with optional named placeholders)
3. The route callback

```
$app = new \Slim\App();
$app->map(['GET', 'POST'], '/books', function ($request, $response, $args) {
    // Create new book or list all books
});
```

Route callbacks

Each routing method described above accepts a callback routine as its final argument. This argument can be any PHP callable, and by default it accepts three arguments.

Request
The first argument is a `Psr\Http\Message\ServerRequestInterface` object that represents the current HTTP request.

Response
The second argument is a `Psr\Http\Message\ResponseInterface` object that represents the current HTTP response.

Arguments
The third argument is an associative array that contains values for the current route's named placeholders.

Writing content to the response

There are two ways you can write content to the HTTP response. First, you can simply `echo()` content from the route callback. This content will be appended to the current HTTP response object. Second, you can return a `Psr\Http\Message\ResponseInterface` object.

Closure binding

If you use a `Closure` instance as the route callback, the closure's state is bound to the `Container` instance. This means you will have access to the DI container instance *inside* of the Closure via the `$this` keyword:

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
    // Use app HTTP cookie service
    $this->get('cookies')->set('name', [
        'value' => $args['name'],
        'expires' => '7 days'
    ]);
});
```

Route strategies

The route callback signature is determined by a route strategy. By default, Slim expects route callbacks to accept the request, response, and an array of route placeholder arguments. This is called the RequestResponse strategy. However, you can change the expected route callback signature by simply using a different strategy. As an example, Slim provides an alternative strategy called RequestResponseArgs that accepts request and response, plus each route placeholder as a separate argument. Here is an example of using this alternative strategy; simply replace the `foundHandler` dependency provided by the default `\Slim\Container`:

```
$c = new \Slim\Container();
$c['foundHandler'] = function() {
    return new \Slim\Handlers\Strategies\RequestResponseArgs();
};

$app = new \Slim\App($c);
$app->get('/hello/{name}', function ($request, $response, $name) {
    return $response->write($name);
});
```

You can provide your own route strategy by implementing the `\Slim\Interfaces\InvocationStrategyInterface`.

Route placeholders

Each routing method described above accepts a URL pattern that is matched against the current HTTP request URI. Route patterns may use named placeholders to dynamically match HTTP request URI segments.

Format

A route pattern placeholder starts with a `{`, followed by the placeholder name, ending with a `}`. This is an example placeholder named `name`:

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
    echo "Hello, " . $args['name'];
});
```

Optional segments

To make a section optional, simply wrap in square brackets:

```
$app->get('/users[/{id}]', function ($request, $response, $args) {
    // responds to both `/users` and `/users/123`
    // but not to `/users/`
});
```

Multiple optional parameters are supported by nesting:

```
$app->get('/news[/{year}[/{month}]]', function ($request, $response, $args) {
    // reponds to `/news`, `/news/2016` and `/news/2016/03`
});
```

For “Unlimited” optional parameters, you can do this:

```
$app->get('/news[/{params:.*}]', function ($request, $response, $args) {
    $params = explode('/', $request->getAttribute('params'));

    // $params is an array of all the optional segments
});
```

In this example, a URI of `/news/2016/03/20` would result in the `$params` array containing three elements: `['2016', '03', '20']` .

Regular expression matching

By default the placeholders are written inside `{}` and can accept any values. However, placeholders can also require the HTTP request URI to match a particular regular expression. If the current HTTP request URI does not match a placeholder regular expression, the route is not invoked. This is an example placeholder named `id` that requires one or more digits.

```
$app = new \Slim\App();
$app->get('/users/{id:[0-9]+}', function ($request, $response, $args) {
    // Find user identified by $args['id']
});
```

Route names

Application routes can be assigned a name. This is useful if you want to programmatically generate a URL to a specific route with the router's `pathFor()` method. Each routing method described above returns a `\Slim\Route` object, and this object exposes a `setName()` method.

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
    echo "Hello, " . $args['name'];
})->setName('hello');
```

You can generate a URL for this named route with the application router's `pathFor()` method.

```
echo $app->getContainer()->get('router')->pathFor('hello', [
    'name' => 'Josh'
]);
// Outputs "/hello/Josh"
```

The router's `pathFor()` method accepts two arguments:

1. The route name
2. Associative array of route pattern placeholders and replacement values

Route groups

To help organize routes into logical groups, the `\Slim\App` also provides a `group()` method. Each group's route pattern is prepended to the routes or groups contained within it, and any placeholder arguments in the group pattern are ultimately made available to the nested routes:

```
$app = new \Slim\App();
$app->group('/users/{id:[0-9]+}', function () {
    $this->map(['GET', 'DELETE', 'PATCH', 'PUT'], '', function ($request, $response, $args) {
        // Find, delete, patch or replace user identified by $args['id']
    })->setName('user');
    $this->get('/reset-password', function ($request, $response, $args) {
        // Route for /users/{id:[0-9]+}/reset-password
        // Reset the password for user identified by $args['id']
    })->setName('user-password-reset');
});
```

Note inside the group closure, `$this` is used instead of `$app`. Slim binds the closure to the application instance for you, just like it is the case with route callback binds with container instance.

- inside group closure, `$this` is bound to the instance of `Slim\App`
- inside route closure, `$this` is bound to the instance of `Slim\Container`

Route middleware

You can also attach middleware to any route or route group. Learn more (/docs/concepts/middleware.html).

Router caching

It's possible to enable router cache by setting valid filename in default Slim settings. Learn more (/docs/objects/application.html#slim-default-settings).

Container Resolution

You are not limited to defining a function for your routes. In Slim there are a few different ways to define your route action functions.

In addition to a function, you may use:

- `container_key:method`
- `Class:method`
- An invokable class
- `container_key`

This functionality is enabled by Slim's Callable Resolver Class. It translates a string entry into a function call. Example:

```
$app->get('/', '\HomeController:home');
```

Alternatively, you can take advantage of PHP's `::class` operator which works well with IDE lookup systems and produces the same result:

```
$app->get('/', \HomeController::class . ':home');
```

In this code above we are defining a `/` route and telling Slim to execute the `home()` method on the `HomeController` class.

Slim first looks for an entry of `HomeController` in the container, if it's found it will use that instance otherwise it will call it's constructor with the container as the first argument. Once an instance of the class is created it will then call the specified method using whatever Strategy you have defined.

Registering a controller with the container

Create a controller with the `home` action method. The constructor should accept the dependencies that are required. For example:

```
class HomeController
{
    protected $view;

    public function __construct(\Slim\Views\Twig $view) {
        $this->view = $view;
    }
    public function home($request, $response, $args) {
      // your code here
      // use $this->view to render the HTML
      return $response;
    }
}
```

Create a factory in the container that instantiates the controller with the dependencies:

```
$container = $app->getContainer();
$container['HomeController'] = function($c) {
    $view = $c->get("view"); // retrieve the 'view' from the container
    return new HomeController($view);
};
```

This allows you to leverage the container for dependency injection and so you can inject specific dependencies into the controller.

Allow Slim to instantiate the controller

Alternatively, if the class does not have an entry in the container, then Slim will pass the container's instance to the constructor. You can construct controllers with many actions instead of an invokable class which only handles one action.

```
class HomeController
{
   protected $container;

   // constructor receives container instance
   public function __construct(ContainerInterface $container) {
       $this->container = $container;
   }

   public function home($request, $response, $args) {
        // your code
        // to access items in the container... $this->container->get('');
        return $response;
   }

   public function contact($request, $response, $args) {
        // your code
        // to access items in the container... $this->container->get('');
        return $response;
   }
}
```

You can use your controller methods like so.

```
$app->get('/', \HomeController::class . ':home');
$app->get('/contact', \HomeController::class . ':contact');
```

Using an invokable class

You do not have to specify a method in your route callable and can just set it to be an invokable class such as:

```
class HomeAction
{
   protected $container;

   public function __construct(ContainerInterface $container) {
       $this->container = $container;
   }

   public function __invoke($request, $response, $args) {
        // your code
        // to access items in the container... $this->container->get('');
        return $response;
   }
}
```

You can use this class like so.

```
$app->get('/', \HomeAction::class);
```

Again, as with controllers, if you register the class name with the container, then you can create a factory and inject just the specific dependencies that you require into your action class.

Error Handling

Error Handlers

Things go wrong. You can't predict errors, but you can anticipate them. Each Slim Framework application has an error handler that receives all uncaught PHP exceptions. This error handler also receives the current HTTP request and response objects, too. The error handler must prepare and return an appropriate Response object to be returned to the HTTP client.

Default error handler

The default error handler is very basic. It sets the Response status code to `500`, it sets the Response content type to `text/html`, and appends a generic error message into the Response body.

This is *probably* not appropriate for production applications. You are strongly encouraged to implement your own Slim application error handler.

The default error handler can also include detailed error diagnostic information. To enable this you need to set the `displayErrorDetails` setting to true:

```
$configuration = [
    'settings' => [
        'displayErrorDetails' => true,
    ],
];
$c = new \Slim\Container($configuration);
$app = new \Slim\App($c);
```

Custom error handler

A Slim Framework application's error handler is a Pimple service. You can substitute your own error handler by defining a custom Pimple factory method with the application container.

There are two ways to inject handlers:

Pre App

```
$c = new \Slim\Container();
$c['errorHandler'] = function ($c) {
    return function ($request, $response, $exception) use ($c) {
        return $c['response']->withStatus(500)
                             ->withHeader('Content-Type', 'text/html')
                             ->write('Something went wrong!');
    };
};
$app = new \Slim\App($c);
```

Post App

```
$app = new \Slim\App();
$c = $app->getContainer();
$c['errorHandler'] = function ($c) {
    return function ($request, $response, $exception) use ($c) {
        return $c['response']->withStatus(500)
                             ->withHeader('Content-Type', 'text/html')
                             ->write('Something went wrong!');
    };
};
```

In this example, we define a new `errorHandler` factory that returns a callable. The returned callable accepts three arguments:

1. A `\Psr\Http\Message\ServerRequestInterface` instance
2. A `\Psr\Http\Message\ResponseInterface` instance
3. A `\Exception` instance

The callable **MUST** return a new `\Psr\Http\Message\ResponseInterface` instance as is appropriate for the given exception.

Class-based error handler

Error handlers may also be defined as an invokable class.

```
class CustomHandler {
   public function __invoke($request, $response, $exception) {
        return $response
            ->withStatus(500)
            ->withHeader('Content-Type', 'text/html')
            ->write('Something went wrong!');
   }
}
```

and attached like so:

```
$app = new \Slim\App();
$c = $app->getContainer();
$c['errorHandler'] = function ($c) {
    return new CustomHandler();
};
```

This allows us to define more sophisticated handlers or extend/override the built-in `Slim\Handlers\*` classes.

Handling other errors

Please note: The following four types of exceptions will not be handled by a custom `errorHandler`:

- `Slim\Exception\MethodNotAllowedException`: This can be handled via a custom `notAllowedHandler` (/docs/handlers/not-allowed.html).
- `Slim\Exception\NotFoundException`: This can be handled via a custom `notFoundHandler` (/docs/handlers/not-found.html).
- Runtime PHP errors (PHP 7+ only): This can be handled via a custom `phpErrorHandler` (/docs/handlers/php-error.html).
- `Slim\Exception\SlimException`: This type of exception is internal to Slim, and its handling cannot be overridden.

Disabling

To completely disable Slim's error handling, simply remove the error handler from the container:

```
unset($app->getContainer()['errorHandler']);
unset($app->getContainer()['phpErrorHandler']);
```

You are now responsible for handling any exceptions that occur in your application as they will not be handled by Slim.

404 Not Found

If your Slim Framework application does not have a route that matches the current HTTP request URI, the application invokes its Not Found handler and returns a `HTTP/1.1 404 Not Found` response to the HTTP client.

Default Not Found handler

Each Slim Framework application has a default Not Found handler. This handler sets the Response status to `404`, it sets the content type to `text/html`, and it writes a simple explanation to the Response body.

Custom Not Found handler

A Slim Framework application's Not Found handler is a Pimple service. You can substitute your own Not Found handler by defining a custom Pimple factory method with the application container.

```
$c = new \Slim\Container(); //Create Your container

//Override the default Not Found Handler
$c['notFoundHandler'] = function ($c) {
    return function ($request, $response) use ($c) {
        return $c['response']
            ->withStatus(404)
            ->withHeader('Content-Type', 'text/html')
            ->write('Page not found');
    };
};

//Create Slim
$app = new \Slim\App($c);

//... Your code
```

In this example, we define a new `notFoundHandler` factory that returns a callable. The returned callable accepts two arguments:

1. A `\Psr\Http\Message\ServerRequestInterface` instance
2. A `\Psr\Http\Message\ResponseInterface` instance

The callable **MUST** return an appropriate `\Psr\Http\Message\ResponseInterface` instance.

405 Not Allowed

If your Slim Framework application has a route that matches the current HTTP request URI **but NOT the HTTP request method**, the application invokes its Not Allowed handler and returns a `HTTP/1.1 405 Not Allowed` response to the HTTP client.

Default Not Allowed handler

Each Slim Framework application has a default Not Allowed handler. This handler sets the Response status to `405`, it sets the content type to `text/html`, it adds a `Allowed:` HTTP header with a comma-delimited list of allowed HTTP methods, and it writes a simple explanation to the Response body.

Custom Not Allowed handler

A Slim Framework application's Not Allowed handler is a Pimple service. You can substitute your own Not Allowed handler by defining a custom Pimple factory method with the application container.

```
// Create Slim
$app = new \Slim\App();
// get the app's di-container
$c = $app->getContainer();
$c['notAllowedHandler'] = function ($c) {
    return function ($request, $response, $methods) use ($c) {
        return $c['response']
            ->withStatus(405)
            ->withHeader('Allow', implode(', ', $methods))
            ->withHeader('Content-type', 'text/html')
            ->write('Method must be one of: ' . implode(', ', $methods));
    };
};
```

> ***N.B*** *Check out Not Found (/docs/handlers/not-found.html) docs for pre-slim creation method using a new instance of* `\Slim\Container`

In this example, we define a new `notAllowedHandler` factory that returns a callable. The returned callable accepts three arguments:

1. A `\Psr\Http\Message\ServerRequestInterface` instance
2. A `\Psr\Http\Message\ResponseInterface` instance
3. A numeric array of allowed HTTP method names

The callable **MUST** return an appropriate `\Psr\Http\Message\ResponseInterface` instance.

PHP Runtime Error

If your Slim Framework application throws a PHP Runtime error (http://php.net/manual/en/class.error.php) (PHP 7+ only), the application invokes its PHP Error handler and returns a `HTTP/1.1 500 Internal Server Error` response to the HTTP client.

Default PHP Error handler

Each Slim Framework application has a default PHP Error handler. This handler sets the Response status to `500`, it sets the content type to `text/html`, and it writes a simple explanation to the Response body.

Custom PHP Error handler

A Slim Framework application's PHP Error handler is a Pimple service. You can substitute your own PHP Error handler by defining a custom Pimple factory method with the application container.

```
// Create Slim
$app = new \Slim\App();
// get the app's di-container
$c = $app->getContainer();
$c['phpErrorHandler'] = function ($c) {
    return function ($request, $response, $error) use ($c) {
        return $c['response']
            ->withStatus(500)
            ->withHeader('Content-Type', 'text/html')
            ->write('Something went wrong!');
    };
};
```

__N.B__ Check out Not Found (/docs/handlers/not-found.html) docs for pre-slim creation method using a new instance of `\Slim\Container`

In this example, we define a new `phpErrorHandler` factory that returns a callable. The returned callable accepts three arguments:

1. A `\Psr\Http\Message\ServerRequestInterface` instance
2. A `\Psr\Http\Message\ResponseInterface` instance
3. A `\Throwable` instance

The callable **MUST** return a new `\Psr\Http\Message\ResponseInterface` instance as is appropriate for the given error.

Cook book

Trailing / in routes

Slim treats a URL pattern with a trailing slash as different to one without. That is, `/user` and `/user/` are different and so can have different callbacks attached.

For GET requests a permanent redirect is fine, but for other request methods like POST or PUT the browser will send the second request with the GET method. To avoid this you simply need to remove the trailing slash and pass the manipulated url to the next middleware.

If you want to redirect/rewrite all URLs that end in a `/` to the non-trailing `/` equivalent, then you can add this middleware:

```
use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

$app->add(function (Request $request, Response $response, callable $next) {
    $uri = $request->getUri();
    $path = $uri->getPath();
    if ($path != '/' && substr($path, -1) == '/') {
        // permanently redirect paths with a trailing slash
        // to their non-trailing counterpart
        $uri = $uri->withPath(substr($path, 0, -1));

        if($request->getMethod() == 'GET') {
            return $response->withRedirect((string)$uri, 301);
        }
        else {
            return $next($request->withUri($uri), $response);
        }
    }

    return $next($request, $response);
});
```

Alternatively, consider oscarotero/psr7-middlewares' TrailingSlash (//github.com/oscarotero/psr7-middlewares#trailingslash) middleware which also allows you to force a trailing slash to be appended to all URLs:

```
use Psr7Middlewares\Middleware\TrailingSlash;

$app->add(new TrailingSlash(true)); // true adds the trailing slash (false removes it)
```

Retrieving IP address

The best way to retrieve the current IP address of the client is via middleware using a component such as rka-ip-address-middleware (https://github.com/akrabat/rka-ip-address-middleware).

This component can be installed via composer:

```
composer require akrabat/rka-ip-address-middleware
```

To use it, register the middleware with the `App`, providing a list of trusted proxies (e.g. varnish servers) if you are using them.:

```
$checkProxyHeaders = true;
$trustedProxies = ['10.0.0.1', '10.0.0.2'];
$app->add(new RKA\Middleware\IpAddress($checkProxyHeaders, $trustedProxies));

$app->get('/', function ($request, $response, $args) {
    $ipAddress = $request->getAttribute('ip_address');

    return $response;
});
```

The middleware stores the client's IP address in a request attribute, so access is via `$request->getAttribute('ip_address')` .

Retrieving Current Route

If you ever need to get access to the current route within your application all you have to do is call the request class' `getAttribute` method with an argument of `'route'` and it will return the current route, which is an instance of the `Slim\Route` class.

From there you can get the route's name by using `getName()` or get the methods supported by this route via `getMethods()` , etc.

Note: If you need to access the route from within your app middleware you must set `'determineRouteBeforeAppMiddleware'` to true in your configuration otherwise `getAttribute('route')` will return null. Also `getAttribute('route')` will return null on non existent routes.

Example:

```
use Slim\App;
use Slim\Exception\NotFoundException;
use Slim\Http\Request;
use Slim\Http\Response;

$app = new App([
    'settings' => [
        // Only set this if you need access to route within middleware
        'determineRouteBeforeAppMiddleware' => true
    ]
]);

// routes...
$app->add(function (Request $request, Response $response, callable $next) {
    $route = $request->getAttribute('route');

    // return NotFound for non existent route
    if (empty($route)) {
        throw new NotFoundException($request, $response);
    }

    $name = $route->getName();
    $groups = $route->getGroups();
    $methods = $route->getMethods();
    $arguments = $route->getArguments();

    // do something with that information

    return $next($request, $response);
});
```

Using Eloquent with Slim

You can use a database ORM such as Eloquent (https://laravel.com/docs/5.1/eloquent) to connect your SlimPHP application to a database.

Adding Eloquent to your application

```
composer require illuminate/database "~5.1"
```

Figure 1: Add Eloquent to your application.

Configure Eloquent

Add the database settings to Slim’s settings array.

```
<?php
return [
    'settings' => [
        // Slim Settings
        'determineRouteBeforeAppMiddleware' => false,
        'displayErrorDetails' => true,
        'db' => [
            'driver' => 'mysql',
            'host' => 'localhost',
            'database' => 'database',
            'username' => 'user',
            'password' => 'password',
            'charset'   => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix'    => '',
        ]
    ],
];
```

Figure 2: Settings array.

In your `dependencies.php` or wherever you add your Service Factories:

```
// Service factory for the ORM
$container['db'] = function ($container) {
    $capsule = new \Illuminate\Database\Capsule\Manager;
    $capsule->addConnection($container['settings']['db']);

    $capsule->setAsGlobal();
    $capsule->bootEloquent();

    return $capsule;
};
```

Figure 3: Configure Eloquent.

Pass a controller an instance of your table

```
$container[App\WidgetController::class] = function ($c) {
    $view = $c->get('view');
    $logger = $c->get('logger');
    $table = $c->get('db')->table('table_name');
    return new \App\WidgetController($view, $logger, $table);
};
```

Figure 4: Pass table object into a controller.

Query the table from a controller

```
<?php

namespace App;

use Slim\Views\Twig;
use Psr\Log\LoggerInterface;
use Illuminate\Database\Query\Builder;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

class WidgetController
{
    private $view;
    private $logger;
    protected $table;

    public function __construct(
        Twig $view,
        LoggerInterface $logger,
        Builder $table
    ) {
        $this->view = $view;
        $this->logger = $logger;
        $this->table = $table;
    }

    public function __invoke(Request $request, Response $response, $args)
    {
        $widgets = $this->table->get();

        $this->view->render($response, 'app/index.twig', [
            'widgets' => $widgets
        ]);

        return $response;
    }
}
```

Figure 5: Sample controller querying the table.

Query the table with where

```
...
$records = $this->table->where('name', 'like', '%foo%')->get();
...
```

Figure 6: Query searching for names matching foo.

Query the table by id

```
...
$record = $this->table->find(1);
...
```

Figure 7: Selecting a row based on id.

More information

Eloquent (https://laravel.com/docs/5.1/eloquent) Documentation

Enabling CORS

CORS - Cross origin resource sharing

A good flowchart for implementing CORS support Reference:

CORS server flowchart (http://www.html5rocks.com/static/images/cors_server_flowchart.png)

You can test your CORS Support here: http://www.test-cors.org/

You can read the specification here: https://www.w3.org/TR/cors/

The simple solution

For simple CORS requests, the server only needs to add the following header to its response:

```
Access-Control-Allow-Origin: <domain>, ... | *
```

The following code should enable lazy CORS.

```
$app->options('/{routes:.+}', function ($request, $response, $args) {
    return $response;
});

$app->add(function ($req, $res, $next) {
    $response = $next($req, $res);
    return $response
            ->withHeader('Access-Control-Allow-Origin', 'http://mysite')
            ->withHeader('Access-Control-Allow-Headers', 'X-Requested-With, Content-Type, Accept,
 Origin, Authorization')
            ->withHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
});
```

Access-Control-Allow-Methods

The following middleware can be used to query Slim's router and get a list of methods a particular pattern implements.

Here is a complete example application:

```
require __DIR__ . "/vendor/autoload.php";

// This Slim setting is required for the middleware to work
$app = new Slim\App([
    "settings"  => [
        "determineRouteBeforeAppMiddleware" => true,
    ]
]);

// This is the middleware
// It will add the Access-Control-Allow-Methods header to every request

$app->add(function($request, $response, $next) {
    $route = $request->getAttribute("route");

    $methods = [];

    if (!empty($route)) {
        $pattern = $route->getPattern();

        foreach ($this->router->getRoutes() as $route) {
            if ($pattern === $route->getPattern()) {
                $methods = array_merge_recursive($methods, $route->getMethods());
            }
        }
        //Methods holds all of the HTTP Verbs that a particular route handles.
    } else {
        $methods[] = $request->getMethod();
    }

    $response = $next($request, $response);


    return $response->withHeader("Access-Control-Allow-Methods", implode(",", $methods));
});

$app->get("/api/{id}", function($request, $response, $arguments) {
});

$app->post("/api/{id}", function($request, $response, $arguments) {
});

$app->map(["DELETE", "PATCH"], "/api/{id}", function($request, $response, $arguments) {
});

// Pay attention to this when you are using some javascript front-end framework and you are using
 groups in slim php
$app->group('/api', function () {
    // Due to the behaviour of browsers when sending PUT or DELETE request, you must add the OPTI
ONS method. Read about preflight.
    $this->map(['PUT', 'OPTIONS'], '/{user_id:[0-9]+}', function ($request, $response, $arguments
) {
        // Your code here...
    });
});

$app->run();
```

A big thank you to tuupola (https://github.com/tuupola) for coming up with this!

Getting and Mocking the Environment

The Environment object encapsulates the `$_SERVER` superglobal array and decouples the Slim application from the PHP global environment. Decoupling the Slim application from the PHP global environment lets us create HTTP requests that may (or may not) resemble the global environment. This is particuarly useful for unit testing and initiating sub-requests. You can fetch the current Environment object anywhere in your Slim application like this:

```
$container = $app->getContainer();
$environment = $container['environment'];
```

Environment Properties

Each Slim application has an Environment object with various properties that determine application behavior. Many of these properties mirror those found in the `$_SERVER` superglobal array. Some properties are required. Other properties are optional.

Required Properties

REQUEST_METHOD
The HTTP request method. This must be one of “GET”, “POST”, “PUT”, “DELETE”, “HEAD”, “PATCH”, or “OPTIONS”.

SCRIPT_NAME
The absolute path name to the front-controller PHP script relative to your document root, disregarding any URL rewriting performed by your web server.

REQUEST_URI
The absolute path name of the HTTP request URI, including any URL rewriting changes performed by your web server.

QUERY_STRING
The part of the HTTP request’s URI path after, but not including, the “?”. This may be an empty string if the current HTTP request does not specify a query string.

SERVER_NAME
The name of the server host under which the current script is executing. If the script is running on a virtual host, this will be the value defined for that virtual host.

SERVER_PORT
The port on the server machine being used by the web server for communication. For default setups, this will be ‘80’; using SSL, for instance, will change this to whatever your defined secure HTTP port is.

HTTPS
Set to a non-empty value if the script was queried through the HTTPS protocol.

Optional Properties

CONTENT_TYPE
The HTTP request content type (e.g., `application/json;charset=utf8`)

CONTENT_LENGTH
The HTTP request content length. This must be an integer if present.

HTTP_*
The HTTP request headers sent by the client. These values are identical to their counterparts in the `$_SERVER` superglobal array. If present, these values must retain the “HTTP_” prefix.

PHP_AUTH_USER
The HTTP `Authentication` header’s decoded username.

PHP_AUTH_PW
The HTTP `Authentication` header’s decoded password.

PHP_AUTH_DIGEST
The raw HTTP `Authentication` header as sent by the HTTP client.

AUTH_TYPE
The HTTP `Authentication` header’s authentication type (e.g., “Basic” or “Digest”).

Mock Environments

Each Slim application instantiates an Environment object using information from the current global environment. However, you may also create mock environment objects with custom information. Mock Environment objects are only useful when writing unit tests.

```
$env = \Slim\Http\Environment::mock([
    'REQUEST_METHOD' => 'PUT',
    'REQUEST_URI' => '/foo/bar',
    'QUERY_STRING' => 'abc=123&foo=bar',
    'SERVER_NAME' => 'example.com',
    'CONTENT_TYPE' => 'application/json;charset=utf8',
    'CONTENT_LENGTH' => 15
]);
```

Uploading Files using POST forms

Files that are uploaded using forms in POST requests can be retrieved with the `getUploadedFiles` (/docs/objects/request.html#uploaded-files) method of the `Request` object.

When uploading files using a POST request, make sure your file upload form has the attribute `enctype="multipart/form-data"` otherwise `getUploadedFiles()` will return an empty array.

If multiple files are uploaded for the same input name, add brackets after the input name in the HTML, otherwise only one uploaded file will be returned for the input name by `getUploadedFiles()`.

Below is an example HTML form that contains both single and multiple file uploads.

```
<!-- make sure the attribute enctype is set to multipart/form-data -->
<form method="post" enctype="multipart/form-data">
    <!-- upload of a single file -->
    <p>
        <label>Add file (single): </label><br/>
        <input type="file" name="example1"/>
    </p>

    <!-- multiple input fields for the same input name, use brackets -->
    <p>
        <label>Add files (up to 2): </label><br/>
        <input type="file" name="example2[]"/><br/>
        <input type="file" name="example2[]"/>
    </p>

    <!-- one file input field that allows multiple files to be uploaded, use brackets -->
    <p>
        <label>Add files (multiple): </label><br/>
        <input type="file" name="example3[]" multiple="multiple"/>
    </p>

    <p>
        <input type="submit"/>
    </p>
</form>
```

Figure 1: Example HTML form for file uploads

Uploaded files can be moved to a directory using the `moveTo` method. Below is an example application that handles the uploaded files of the HTML form above.

```
<?php

require_once __DIR__ . '/vendor/autoload.php';

use Slim\Http\Request;
use Slim\Http\Response;
use Slim\Http\UploadedFile;

$app = new \Slim\App();

$container = $app->getContainer();
$container['upload_directory'] = __DIR__ . '/uploads';

$app->post('/', function(Request $request, Response $response) {
    $directory = $this->get('upload_directory');
```

```
    $uploadedFiles = $request->getUploadedFiles();

    // handle single input with single file upload
    $uploadedFile = $uploadedFiles['example1'];
    if ($uploadedFile->getError() === UPLOAD_ERR_OK) {
        $filename = moveUploadedFile($directory, $uploadedFile);
        $response->write('uploaded ' . $filename . '<br/>');
    }


    // handle multiple inputs with the same key
    foreach ($uploadedFiles['example2'] as $uploadedFile) {
        if ($uploadedFile->getError() === UPLOAD_ERR_OK) {
            $filename = moveUploadedFile($directory, $uploadedFile);
            $response->write('uploaded ' . $filename . '<br/>');
        }
    }

    // handle single input with multiple file uploads
    foreach ($uploadedFiles['example3'] as $uploadedFile) {
        if ($uploadedFile->getError() === UPLOAD_ERR_OK) {
            $filename = moveUploadedFile($directory, $uploadedFile);
            $response->write('uploaded ' . $filename . '<br/>');
        }
    }

    // handle single input with multiple file uploads
    foreach ($uploadedFiles['example3'] as $uploadedFile) {
        if ($uploadedFile->getError() === UPLOAD_ERR_OK) {
            $filename = moveUploadedFile($directory, $uploadedFile);
            $response->write('uploaded ' . $filename . '<br/>');
        }
    }
});

/**
 * Moves the uploaded file to the upload directory and assigns it a unique name
 * to avoid overwriting an existing uploaded file.
 *
 * @param string $directory directory to which the file is moved
 * @param UploadedFile $uploaded file uploaded file to move
 * @return string filename of moved file
 */
function moveUploadedFile($directory, UploadedFile $uploadedFile)
{
    $extension = pathinfo($uploadedFile->getClientFilename(), PATHINFO_EXTENSION);
    $basename = bin2hex(random_bytes(8)); // see http://php.net/manual/en/function.random-bytes.p
hp
    $filename = sprintf('%s.%0.8s', $basename, $extension);

    $uploadedFile->moveTo($directory . DIRECTORY_SEPARATOR . $filename);

    return $filename;
}

$app->run();
```

Figure 2: Example Slim applcation to handle the uploaded files

See also

- https://akrabat.com/psr-7-file-uploads-in-slim-3/ (https://akrabat.com/psr-7-file-uploads-in-slim-3/)

Action-Domain-Responder with Slim

In this post, I'll show how to refactor the Slim tutorial application to more closely follow the Action-Domain-Responder (http://pmjones.io/adr) pattern.

One nice thing about Slim (and most other HTTP user interface frameworks (http://paul-m-jones.com/archives/6627)) is that they are already "action" oriented. That is, their routers do not presume a controller class with many action methods. Instead, they presume an action closure or a single-action invokable class.

So the Action part of Action-Domain-Responder already exists for Slim. All that is needed is to pull extraneous bits out of the Actions, to more clearly separate the Action behaviors from Domain and the Responder behaviors.

Extract Domain

Let's begin by extracting the Domain logic. In the original tutorial, the Actions use two data-source mappers directly, and embed some business logic as well. We can create a Service Layer class called `TicketService` and move those operations from the Actions into the Domain. Doing so gives us this class:

```
class TicketService
{
    protected $ticket_mapper;
    protected $component_mapper;

    public function __construct(
        TicketMapper $ticket_mapper,
        ComponentMapper $component_mapper
    ) {
        $this->ticket_mapper = $ticket_mapper;
        $this->component_mapper = $component_mapper;
    }

    public function getTickets()
    {
        return $this->ticket_mapper->getTickets();
    }

    public function getComponents()
    {
        return $this->component_mapper->getComponents();
    }

    public function getTicketById($ticket_id)
    {
        $ticket_id = (int) $ticket_id;
        return $this->ticket_mapper->getTicketById($ticket_id);
    }

    public function createTicket($data)
    {
        $component_id = (int) $data['component'];
        $component = $this->component_mapper->getComponentById($component_id);

        $ticket_data = [];
        $ticket_data['title'] = filter_var($data['title'], FILTER_SANITIZE_STRING);
        $ticket_data['description'] = filter_var($data['description'], FILTER_SANITIZE_STRING);
        $ticket_data['component'] = $component->getName();

        $ticket = new TicketEntity($ticket_data);
        $this->ticket_mapper->save($ticket);
        return $ticket;
    }
}
```

We create a container object for it in `index.php` like so:

```
$container['ticket_service'] = function ($c) {
    return new TicketService(
        new TicketMapper($c['db']),
        new ComponentMapper($c['db'])
    );
};
```

And now the Actions can use the `TicketService` instead of performing domain logic directly:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $tickets = $this->ticket_service->getTickets();
    $response = $this->view->render(
        $response,
        "tickets.phtml",
        ["tickets" => $tickets, "router" => $this->router]
    );
    return $response;
});

$app->get('/ticket/new', function (Request $request, Response $response) {
    $components = $this->ticket_service->getComponents();
    $response = $this->view->render(
        $response,
        "ticketadd.phtml",
        ["components" => $components]
    );
    return $response;
});

$app->post('/ticket/new', function (Request $request, Response $response) {
    $data = $request->getParsedBody();
    $this->ticket_service->createTicket($data);
    $response = $response->withRedirect("/tickets");
    return $response;
});

$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    $ticket = $this->ticket_service->getTicketById($args['id']);
    $response = $this->view->render(
        $response,
        "ticketdetail.phtml",
        ["ticket" => $ticket]
    );
    return $response;
})->setName('ticket-detail');
```

One benefit here is that we can now test the domain activities separately from the actions. We can begin to do something more like integration testing, even unit testing, instead of end-to-end system testing.

Extract Responder

In the case of the tutorial application, the presentation work is so straightforward as to not require a separate Responder for each action. A relaxed variation of a Responder layer is perfectly suitable in this simple case, one where each Action uses a different method on a common Responder.

Extracting the presentation work to a separate Responder, so that response-building is completely removed from the Action, looks like this:

```
use Psr\Http\Message\ResponseInterface as Response;
use Slim\Views\PhpRenderer;

class TicketResponder
{
    protected $view;

    public function __construct(PhpRenderer $view)
    {
        $this->view = $view;
    }

    public function index(Response $response, array $data)
    {
        return $this->view->render(
            $response,
            "tickets.phtml",
            $data
        );
    }

    public function detail(Response $response, array $data)
    {
        return $this->view->render(
            $response,
            "ticketdetail.phtml",
            $data
        );
    }

    public function add(Response $response, array $data)
    {
        return $this->view->render(
            $response,
            "ticketadd.phtml",
            $data
        );
    }

    public function create(Response $response)
    {
        return $response->withRedirect("/tickets");
    }
}
```

We can then add the `TicketResponder` object to the container in `index.php`:

```
$container['ticket_responder'] = function ($c) {
    return new TicketResponder($c['view']);
};
```

And finally we can refer to the Responder, instead of just the template system, in the Actions:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $tickets = $this->ticket_service->getTickets();
    return $this->ticket_responder->index(
        $response,
        ["tickets" => $tickets, "router" => $this->router]
    );
});

$app->get('/ticket/new', function (Request $request, Response $response) {
    $components = $this->ticket_service->getComponents();
    return $this->ticket_responder->add(
        $response,
        ["components" => $components]
    );
});

$app->post('/ticket/new', function (Request $request, Response $response) {
    $data = $request->getParsedBody();
    $this->ticket_service->createTicket($data);
    return $this->ticket_responder->create($response);
});

$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    $ticket = $this->ticket_service->getTicketById($args['id']);
    return $this->ticket_responder->detail(
        $response,
        ["ticket" => $ticket]
    );
})->setName('ticket-detail');
```

Now we can test the response-building work separately from the domain work.

Some notes:

Putting all the response-building in a single class with multiple methods, especially for simple cases like this tutorial, is fine to start with. For ADR, is not strictly necessary to have one Responder for each Action. What *is* necessary is to extract the response-building concerns out of the Action.

But as the presentation logic complexity increases (content-type negotiation? status headers? etc.), and as dependencies become different for each kind of response being built, you will want to have a Responder for each Action.

Alternatively, you might stick with a single Responder, but reduce its interface to a single method. In that case, you may find that using a Domain Payload (http://paul-m-jones.com/archives/6043) (instead of “naked” domain results) has some significant benefits.

Conclusion

At this point, the Slim tutorial application has been converted to ADR. We have separated the domain logic to a `TicketService`, and the presentation logic to a `TicketResponder`. And it's easy to see how each Action does pretty much the same thing:

- Marshals input and passes it into the Domain
- Gets back a result from the Domain and passes it to the Responder
- Invokes the Responder so it can build and return the Response

Now, for a simple case like this, using ADR (or even webbishy MVC) might seem like overkill. But simple cases become complex quickly, and this simple case shows how the ADR separation-of-concerns can be applied as a Slim-based application increases in complexity.

Add Ons

Templates

Slim does not have a view layer like traditional MVC frameworks. Instead, Slim's "view" *is the HTTP response*. Each Slim application route is responsible for preparing and returning an appropriate PSR 7 response object.

> *Slim's "view" is the HTTP response.*

That being said, the Slim project provides the Twig-View and PHP-View components to help you render templates to a PSR7 Response object.

The slim/twig-view component

The Twig-View (https://github.com/slimphp/Twig-View) PHP component helps you render Twig (http://twig.sensiolabs.org/) templates in your application. This component is available on Packagist, and it's easy to install with Composer like this:

```
composer require slim/twig-view
```

Figure 1: Install slim/twig-view component.

Next, you need to register the component as a service on the Slim app's container like this:

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

// Get container
$container = $app->getContainer();

// Register component on container
$container['view'] = function ($container) {
    $view = new \Slim\Views\Twig('path/to/templates', [
        'cache' => 'path/to/cache'
    ]);

    // Instantiate and add Slim specific extension
    $basePath = rtrim(str_ireplace('index.php', '', $container['request']->getUri()->getBasePath(
)), '/');
    $view->addExtension(new Slim\Views\TwigExtension($container['router'], $basePath));

    return $view;
};
```

Figure 2: Register slim/twig-view component with container.

Note : "cache" could be set to false to disable it, see also 'auto_reload' option, useful in development environment. For more information, see Twig environment options (http://twig.sensiolabs.org/doc/2.x/api.html#environment-options)

Now you can use the `slim/twig-view` component service inside an app route to render a template and write it to a PSR 7 Response object like this:

```
// Render Twig template in route
$app->get('/hello/{name}', function ($request, $response, $args) {
    return $this->view->render($response, 'profile.html', [
        'name' => $args['name']
    ]);
})->setName('profile');

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

Figure 3: Render template with slim/twig-view container service.

In this example, `$this->view` invoked inside the route callback is a reference to the `\Slim\Views\Twig` instance returned by the `view` container service. The `\Slim\Views\Twig` instance's `render()` method accepts a PSR 7 Response object as its first argument, the Twig template path as its second argument, and an array of template variables as its final argument. The `render()` method returns a new PSR 7 Response object whose body is the rendered Twig template.

The path_for() method

The `slim/twig-view` component exposes a custom `path_for()` function to your Twig templates. You can use this function to generate complete URLs to any named route in your Slim application. The `path_for()` function accepts two arguments:

1. A route name
2. A hash of route placeholder names and replacement values

The second argument's keys should correspond to the selected route's pattern placeholders. This is an example Twig template that draws a link URL for the "profile" named route shown in the example Slim application above.

```
{% extends "layout.html" %}

{% block body %}
<h1>User List</h1>
<ul>
    <li><a href="{{ path_for('profile', { 'name': 'josh' }) }}">Josh</a></li>
</ul>
{% endblock %}
```

The slim/php-view component

The PHP-View (https://github.com/slimphp/PHP-View) PHP component helps you render PHP templates. This component is available on Packagist and can be installed using Composer like this:

```
composer require slim/php-view
```

Figure 4: Install slim/php-view component.

To register this component as a service on Slim App's container, do this:

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

// Get container
$container = $app->getContainer();

// Register component on container
$container['view'] = function ($container) {
    return new \Slim\Views\PhpRenderer('path/to/templates/with/trailing/slash/');
};
```

Figure 5: Register slim/php-view component with container.

Use the view component to render a PHP view like this:

```
// Render PHP template in route
$app->get('/hello/{name}', function ($request, $response, $args) {
    return $this->view->render($response, 'profile.html', [
        'name' => $args['name']
    ]);
})->setName('profile');

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

Figure 6: Render template with slim/php-view container service.

Other template systems

You are not limited to the `Twig-View` and `PHP-View` components. You can use any PHP template system provided that you ultimately write the rendered template output to the PSR 7 Response object's body.

HTTP Caching

Slim 3 uses the optional standalone slimphp/Slim-HttpCache (https://github.com/slimphp/Slim-HttpCache) PHP component for HTTP caching. You can use this component to create and return responses that contain `Cache` , `Expires` , `ETag` , and `Last-Modified` headers that control when and how long application output is retained by client-side caches. You may have to set your php.ini setting "session.cache_limiter" to an empty string in order to get this working without interferences.

Installation

Execute this bash command from your project's root directory:

```
composer require slim/http-cache
```

Usage

The `slimphp/Slim-HttpCache` component contains a service provider and an application middleware. You should add both to your application like this:

```
// Register service provider with the container
$container = new \Slim\Container;
$container['cache'] = function () {
    return new \Slim\HttpCache\CacheProvider();
};

// Add middleware to the application
$app = new \Slim\App($container);
$app->add(new \Slim\HttpCache\Cache('public', 86400));

// Create your application routes...

// Run application
$app->run();
```

ETag

Use the service provider's `withEtag()` method to create a Response object with the desired `ETag` header. This method accepts a PSR7 response object, and it returns a cloned PSR7 response with the new HTTP header.

```
$app->get('/foo', function ($req, $res, $args) {
    $resWithEtag = $this->cache->withEtag($res, 'abc');

    return $resWithEtag;
});
```

Expires

Use the service provider's `withExpires()` method to create a Response object with the desired `Expires` header. This method accepts a PSR7 response object, and it returns a cloned PSR7 response with the new HTTP header.

```
$app->get('/bar',function ($req, $res, $args) {
    $resWithExpires = $this->cache->withExpires($res, time() + 3600);

    return $resWithExpires;
});
```

Last-Modified

Use the service provider's `withLastModified()` method to create a Response object with the desired `Last-Modified` header. This method accepts a PSR7 response object, and it returns a cloned PSR7 response with the new HTTP header.

```
//Example route with LastModified
$app->get('/foobar',function ($req, $res, $args) {
    $resWithLastMod = $this->cache->withLastModified($res, time() - 3600);

    return $resWithLastMod;
});
```

CSRF Protection

Slim 3 uses the optional standalone slimphp/Slim-Csrf (https://github.com/slimphp/Slim-Csrf) PHP component to protect your application from CSRF (cross-site request forgery). This component generates a unique token per request that validates subsequent POST requests from client-side HTML forms.

Installation

Execute this bash command from your project's root directory:

```
composer require slim/csrf
```

Usage

The `slimphp/Slim-Csrf` component contains an application middleware. Add it to your application like this:

```
// Add middleware to the application
$app = new \Slim\App;
$app->add(new \Slim\Csrf\Guard);

// Create your application routes...

// Run application
$app->run();
```

Fetch the CSRF token name and value

The latest CSRF token's name and value are available as attributes on the PSR7 request object. The CSRF token name and value are unique for each request. You can fetch the current CSRF token name and value like this.

```
$app->get('/foo', function ($req, $res, $args) {
    // CSRF token name and value
    $nameKey = $this->csrf->getTokenNameKey();
    $valueKey = $this->csrf->getTokenValueKey();
    $name = $req->getAttribute($nameKey);
    $value = $req->getAttribute($valueKey);

    // Render HTML form which POSTs to /bar with two hidden input fields for the
    // name and value:
    // <input type="hidden" name="<?= $nameKey ?>" value="<?= $name ?>">
    // <input type="hidden" name="<?= $valueKey ?>" value="<?= $value ?>">
});

$app->post('/bar', function ($req, $res, $args) {
    // CSRF protection successful if you reached
    // this far.
});
```

You should pass the CSRF token name and value to the template so they may be submitted with HTML form POST requests. They are often stored as a hidden field with HTML forms.

For more use cases and documentation please check slimphp/Slim-Csrf (https://github.com/slimphp/Slim-Csrf)'s page.

Flash Messages

Install

Via Composer

```
$ composer require slim/flash
```

Requires Slim 3.0.0 or newer.

Usage

```
// Start PHP session
session_start(); //by default requires session storage

$app = new \Slim\App();

// Fetch DI Container
$container = $app->getContainer();

// Register provider
$container['flash'] = function () {
    return new \Slim\Flash\Messages();
};

$app->get('/foo', function ($req, $res, $args) {
    // Set flash message for next request
    $this->flash->addMessage('Test', 'This is a message');

    // Redirect
    return $res->withStatus(302)->withHeader('Location', '/bar');
});

$app->get('/bar', function ($req, $res, $args) {
    // Get flash messages from previous request
    $messages = $this->flash->getMessages();
    print_r($messages);
});

$app->run();
```

Please note that a message could be a string, object or array. Please check what your storage can handle.

Contributing

Branching Strategy

The Slim Framework uses a simple branching strategy. There is a `3.x` branch, and the `3.x` branch `HEAD` reference points to the latest unstable code. Each stable release is denoted with a numeric tag (e.g., `3.0.0`).

Guidelines

I encourage everyone to contribute to the Slim Framework project. You can find the latest code on GitHub at https://github.com/slimphp/Slim (https://github.com/slimphp/Slim).

Issue Tracker

You can find outstanding issues on the GitHub Issue Tracker (https://github.com/slimphp/Slim/issues). If you intend to work on a specific issue, leave a comment on the appropriate thread to inform other project contributors.

Pull Requests

- Each pull request should contain only one new feature or improvement.
- Pull requests should be submitted to the `master` branch

Code Style

All pull requests must use the PSR-2 (http://www.php-fig.org/psr/psr-2/) code style.

- Code MUST use the PSR-1 (http://www.php-fig.org/psr/psr-1/) code style.
- Code MUST use 4 spaces for indenting, not tabs.
- There MUST NOT be a hard limit on line length; the soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.
- There MUST be one blank line after the namespace declaration, and there MUST be one blank line after the block of use declarations.
- Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.
- Opening braces for methods MUST go on the next line, and closing braces MUST go on the next line after the body.
- Visibility MUST be declared on all properties and methods; abstract and final MUST be declared before the visibility; static MUST be declared after the visibility.
- Control structure keywords MUST have one space after them; method and function calls MUST NOT.
- Opening braces for control structures MUST go on the same line, and closing braces MUST go on the next line after the body.
- Opening parentheses for control structures MUST NOT have a space after them, and closing parentheses for control structures MUST NOT have a space before.

Index

Add Ons

Contributing