

Slim

a micro framework for PHP

Rob Allen, October 2019

A 3D animated character of Mr. Topham Hatt, a portly man with a large nose, wearing a black top hat, a dark suit, a white shirt, and a yellow tie. He is standing in front of a brick wall with stone accents. The text "The C in MVC" is overlaid in the center in a bright green font.

The C in MVC

Slim Framework

- Created by Josh Lockhart (phptherightway.com)
- PSR-7 Request and Response objects
- PSR-15 Middleware and Request Handlers
- PSR-11 DI container support



PHP-FIG

Hello world

use ...

```
$app = AppFactory::create();
```

```
$app->get(  
    '/ping',  
    function (Request $request, Response $response) {  
        $response->getBody()->write(json_encode(['ack' => time()]));  
        return $response->withHeader('Content-Type', 'application/json');  
    });
```

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


Hello world

```
use ...
```

```
$app = AppFactory::create(); // Init
```

```
$app->get(
    '/ping',
    function (Request $request, Response $response) {
        $response->getBody()->write(json_encode(['ack' => time()]));
        return $response->withHeader('Content-Type', 'application/json');
    });
```

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

Hello world

```
use ...
```

```
$app = AppFactory::create();
```

```
$app->get(                                                    // Method  
    '/ping',  
    function (Request $request, Response $response) {  
        $response->getBody()->write(json_encode(['ack' => time()]));  
        return $response->withHeader('Content-Type', 'application/json');  
    });
```

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

Hello world

```
use ...
```

```
$app = AppFactory::create();
```

```
$app->get(  
    '/ping', // Pattern  
    function (Request $request, Response $response) {  
        $response->getBody()->write(json_encode(['ack' => time()]));  
        return $response->withHeader('Content-Type', 'application/json');  
    });
```

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

Hello world

```
use ...
```

```
$app = AppFactory::create();
```

```
$app->get(  
    '/ping',  
    function (Request $request, Response $response) { // Handler  
        $response->getBody()->write(json_encode(['ack' => time()]));  
        return $response->withHeader('Content-Type', 'application/json');  
    });
```

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

Hello world

```
$ curl -i http://localhost:8888/ping
HTTP/1.1 200 OK
Host: localhost:8888
Connection: close
Content-Type: application/json
```

```
{
  "ack":1570046120
}
```



PSR-7 is the foundation

It's all about HTTP

Request:

{METHOD} {URI} HTTP/1.1

Header: value1,value2

Another-Header: value

Message body



It's all about HTTP

Response:

HTTP/1.1 {STATUS_CODE} {REASON_PHRASE}

Header: value

Message body



PSR 7

OO interfaces to model HTTP

- `RequestInterface` (& `ServerRequestInterface`)
- `ResponseInterface`
- `UriInterface`
- `UploadedFileInterface`

Key feature 1: Immutability

Request, Response, Uri & UploadFile are *immutable*

```
$uri = new Uri('https://api.joind.in/v2.1/events');  
$uri2 = $uri->withQuery('?filter=upcoming');  
$uri3 = $uri->withQuery('?filter=cfp');
```

Key feature 2: Streams

Message bodies are *streams*

```
$body = new Stream();  
$body->write('<p>Hello');  
$body->write('World</p>');  
  
$response = (new Response())  
    ->withStatus(200, 'OK')  
    ->withHeader('Content-Type', 'application/header')  
    ->withBody($body);
```

API

Device $\leftrightarrow$ Server

/api/route

PUT title, waypoint[] $\rightarrow$ routeID

/api/route/id

I write APIs

GET $\rightarrow$ Route, Waypoint[], Skin

/api/route/updates

POST $\rightarrow$ [" , " , "]

/api/route/browse*

GET $\rightarrow$ [Route, Skin]

*Optional geo data

A good ~~API~~ framework

- Understands HTTP methods
- Can receive data in various formats
- Has useful error reporting

~~hätte...~~

~~sollte...~~

~~würde...~~

~~könnte...~~

machen !!!

HTTP verbs



HTTP methods

Method	Used for	Idempotent?
GET	Retrieve data	Yes
PUT	Change data	Yes
DELETE	Delete data	Yes
POST	Change data	No
PATCH	Update data	No



Routing HTTP methods in Slim

```
// specific HTTP methods map to methods on $app
```

```
$app->get('/games', ListGamesHandler::class);
```

```
$app->post('/games', CreateGameHandler::class);
```

```
// routing multiple HTTP methods
```

```
$app->any('/games', GamesHandler::class);
```

```
$app->map(['GET', 'POST'], '/games', GamesHandler::class);
```


Dynamic routes

```
$app->get('/games/{id}',  
  function($request, $response) {  
    $id = $request->getAttribute('id');  
    $games = $this->gameRepository->loadById($id);  
  
    $body = json_encode(['game' => $game]);  
    $response->getBody()->write($body);  
  
    $response = $response->withHeader(  
      'Content-Type', 'application/json');  
    return $response;  
  });
```

It's just Regex

// numbers only

```
$app->get('/games/{id:\d+}', $callable);
```

// optional segments

```
$app->get('/games[/{\id:\d+}]/', $callable);
```

```
$app->get('/news[/{\y:\d{4}}[/{\m:\d{2}}]]', $callable);
```

Invalid HTTP request methods

If the *HTTP method* is not supported, return the 405 status code

```
$ http --json PUT http://localhost:8888/ping
```

```
HTTP/1.1 405 Method Not Allowed
```

```
Allow: GET
```

```
Connection: close
```

```
Content-Length: 53
```

```
Content-type: application/json
```

```
Host: localhost:8888
```

```
{  
  "message": "Method not allowed. Must be one of: GET"  
}
```



Incoming data

Content-Type handling

The *Content-Type* header specifies the format of the incoming data

```
$ curl http://localhost:8888/games \  
  -H "Content-Type: application/json" \  
  -d '{"player1": "Rob", "player2": "Jon"}'
```

Read with `getBody()`

```
$app->post('/games',  
  function ($request, $response) {  
    $data = $request->getBody();  
    $response->getBody()->write(print_r($data, true));  
    return $response;  
  }  
);
```

Output:

```
'{"player1": "Rob", "player2": "Jon"}'
```

Read with getParsedBody()

Add Slim's body-parsing middleware to your app:

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

Use in your handler:

```
$app->post( '/games',  
  function ($request, $response) {  
    $data = $request->getParsedBody();  
    return $response->write(print_r($data, true));  
  }  
);
```

Read with getParsedBody()

```
$ curl -H "Content-Type: application/json" \  
  -H "Content-Type: application/json" \  
  -d '{"player1": "Rob", "player2": "Jon"}'
```

Output:

```
Array  
(  
  [player1] => Rob,  
  [player2] => Jon  
)
```


This also works with XML

```
$ curl "http://localhost:8888/games" \  
  -H "Content-Type: application/xml" \  
  -d "<game><player1>Rob</player1><player2>Jon</player2></game>"
```

Output:

Array

```
(  
  [player1] => Rob,  
  [player2] => Jon  
)
```

And form data

```
curl "http://localhost:8888/games" \  
  -H "Content-Type: application/x-www-form-urlencoded" \  
  -d "player1=Rob" -d "player2=Jon"
```

Output:

Array

```
(  
  [player1] => Rob,  
  [player2] => Jon  
)
```

addBodyParsingMiddleware() ?

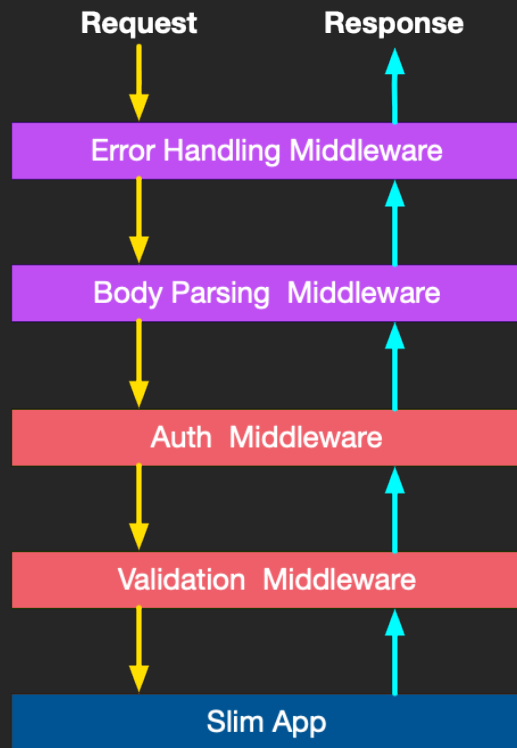
Middleware

Middleware is code that exists between the request and response, and which can take the incoming request, perform actions based on it, and either complete the response or pass delegation on to the next middleware in the queue.

Matthew Weier O'Phinney

Middleware

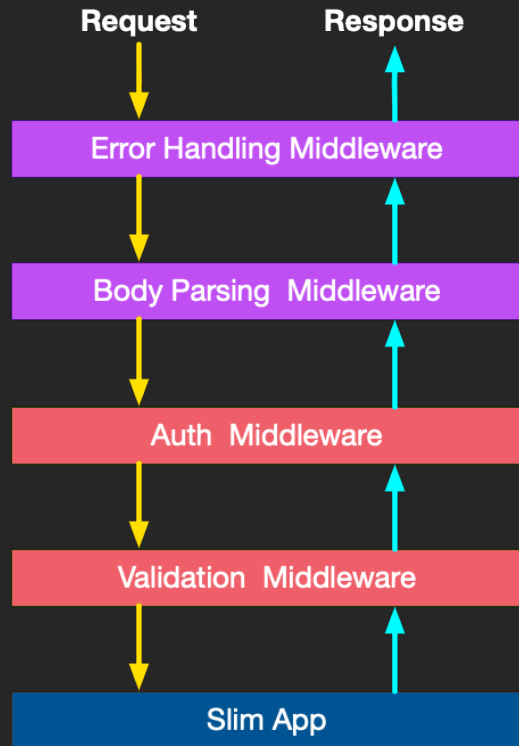
Take a request, return a response



Middleware

LIFO stack:

```
$app->add(ValidationMiddleware::class);  
$app->add(AuthMiddleware::class);  
$app->add(AuthMiddleware::class);  
$app->addBodyParsingMiddleware();  
$app->addErrorMiddleware(true, true, true);
```



PSR-15 MiddlewareInterface

```
namespace Psr\Http\Server;
```

```
use Psr\Http\Message\ResponseInterface;
```

```
use Psr\Http\Message\ServerRequestInterface;
```

```
interface MiddlewareInterface
```

```
{
```

```
    public function process(  
        ServerRequestInterface $request,  
        RequestHandlerInterface $handler  
    ) : ResponseInterface;
```

```
}
```

PSR-15 MiddlewareInterface

```
namespace Psr\Http\Server;  
  
use Psr\Http\Message\ResponseInterface;  
use Psr\Http\Message\ServerRequestInterface;  
  
interface MiddlewareInterface  
{  
    public function process(  
        ServerRequestInterface $request,  
        RequestHandlerInterface $handler  
    ) : ResponseInterface;  
}
```


PSR-15 MiddlewareInterface

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ) : ResponseInterface;
}
```

PSR-15 MiddlewareInterface

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ) : ResponseInterface;
}
```

TimerMiddleware

```
class TimerMiddleware implements MiddlewareInterface
{
    public function process($request, $handler)
    {
        $start = microtime(true);

        $response = $handler->handle($request);

        $taken = microtime(true) - $start;

        return $response->withHeader('Time-Taken', $taken);
    }
}
```

TimerMiddleware

```
class TimerMiddleware implements MiddlewareInterface
{
    public function process($request, $handler)
    {
        $start = microtime(true);

        $response = $handler->handle($request);

        $taken = microtime(true) - $start;

        return $response->withHeader('Time-Taken', $taken);
    }
}
```

TimerMiddleware

```
class TimerMiddleware implements MiddlewareInterface
{
    public function process($request, $handler)
    {
        $start = microtime(true);

        $response = $handler->handle($request);

        $taken = microtime(true) - $start;

        return $response->withHeader('Time-Taken', $taken);
    }
}
```

TimerMiddleware

```
class TimerMiddleware implements MiddlewareInterface
{
    public function process($request, $handler)
    {
        $start = microtime(true);

        $response = $handler->handle($request);

        $taken = microtime(true) - $start;

        return $response->withHeader('Time-Taken', $taken);
    }
}
```

TimerMiddleware

```
class TimerMiddleware implements MiddlewareInterface
{
    public function process($request, $handler)
    {
        $start = microtime(true);

        $response = $handler->handle($request);

        $taken = microtime(true) - $start;

        return $response->withHeader('Time-Taken', $taken);
    }
}
```

Route Handlers

The other half of PSR-15!



Route Handlers

Any callable!

```
$app->add('/', function ($request, $response) { ... });  
$app->add('/', ['RootController', 'aStaticFunction']);  
$app->add('/', [new RootController(), 'aFunction']);  
$app->add('/', RootController::class.':pingAction');  
$app->add('/', RootHandler::class);
```

Use this one

```
$app->add( '/', RootHandler::class);
```

PSR-15 RouteHandlerInterface

```
namespace Psr\Http\Server;
```

```
use Psr\Http\Message\ResponseInterface;
```

```
use Psr\Http\Message\ServerRequestInterface;
```

```
interface RequestHandlerInterface
```

```
{
```

```
    public function handle(  
        ServerRequestInterface $request  
    ): ResponseInterface;
```

```
}
```

PSR-15 RouteHandlerInterface

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface RequestHandlerInterface
{
    public function handle(
        ServerRequestInterface $request
    ): ResponseInterface;
}
```

PSR-15 RouteHandlerInterface

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface RequestHandlerInterface
{
    public function handle(
        ServerRequestInterface $request
    ): ResponseInterface;
}
```

PSR-15 RouteHandlerInterface

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface RequestHandlerInterface
{
    public function handle(
        ServerRequestInterface $request
    ): ResponseInterface;
}
```

RootHandler

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Server\RequestHandlerInterface;
use Slim\Psr7\Response;

class RootHandler implements RequestHandlerInterface
{
    public function handle(Request $request): ResponseInterface
    {
        $response = new Response();
        $response->getBody()->write(json_encode(['msg' => 'hello world']));
        return $response;
    }
}
```

RootHandler

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Server\RequestHandlerInterface;
use Slim\Psr7\Response;

class RootHandler implements RequestHandlerInterface
{
    public function handle(Request $request): ResponseInterface
    {
        $response = new Response();
        $response->getBody()->write(json_encode(['msg' => 'hello world']));
        return $response;
    }
}
```


RootHandler

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Server\RequestHandlerInterface;
use Slim\Psr7\Response;

class RootHandler implements RequestHandlerInterface
{
    public function handle(Request $request): ResponseInterface
    {
        $response = new Response();
        $response->getBody()->write(json_encode(['msg' => 'Hello world']));
        return $response;
    }
}
```

RootHandler

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Server\RequestHandlerInterface;
use Slim\Psr7\Response;

class RootHandler implements RequestHandlerInterface
{
    public function handle(Request $request): ResponseInterface
    {
        $response = new Response();
        $response->getBody()->write(json_encode(['msg' => 'hello world']));
        return $response;
    }
}
```

When things go wrong



Error handling

- Internal logging
- Rendering for output

Injecting a logger

More PSRs! 11 & 3

Configure PHP-DI

```
$containerBuilder = new ContainerBuilder();

$containerBuilder->addDefinitions([
    // Factory for a PSR-3 logger
    LoggerInterface::class => function (ContainerInterface $c) {
        $logger = new Logger($settings['name']);
        $logger->pushHandler(new ErrorLogHandler());
        return $logger;
    },
]);

AppFactory::setContainer($containerBuilder->build());
$app = AppFactory::create();
```

Configure PHP-DI

```
$containerBuilder = new ContainerBuilder();

$containerBuilder->addDefinitions([
    // Factory for a PSR-3 logger
    LoggerInterface::class => function (ContainerInterface $c) {
        $logger = new Logger($settings['name']);
        $logger->pushHandler(new ErrorLogHandler());
        return $logger;
    },
]);

AppFactory::setContainer($containerBuilder->build());
$app = AppFactory::create();
```

Configure PHP-DI

```
$containerBuilder = new ContainerBuilder();

$containerBuilder->addDefinitions([
    // Factory for a PSR-3 logger
    LoggerInterface::class => function (ContainerInterface $c) {
        $logger = new Logger($settings['name']);
        $logger->pushHandler(new ErrorLogHandler());
        return $logger;
    },
]);

AppFactory::setContainer($containerBuilder->build());
$app = AppFactory::create();
```


Configure PHP-DI

```
$containerBuilder = new ContainerBuilder();

$containerBuilder->addDefinitions([
    // Factory for a PSR-3 logger
    LoggerInterface::class => function (ContainerInterface $c) {
        $logger = new Logger($settings['name']);
        $logger->pushHandler(new ErrorLogHandler());
        return $logger;
    },
]);

AppFactory::setContainer($containerBuilder->build());
$app = AppFactory::create();
```

Configure PHP-DI

```
$containerBuilder = new ContainerBuilder();

$containerBuilder->addDefinitions([
    // Factory for a PSR-3 logger
    LoggerInterface::class => function (ContainerInterface $c) {
        $logger = new Logger($settings['name']);
        $logger->pushHandler(new ErrorLogHandler());
        return $logger;
    },
]);

AppFactory::setContainer($containerBuilder->build());
$app = AppFactory::create();
```

PHP-DI autowiring

Just type-hint your constructor!

```
class GetGameHandler implements RequestHandlerInterface
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
    ...
}
```

Logging

```
class GetGameHandler implements RequestHandlerInterface
{
    public function handle(Request $request): ResponseInterface
    {
        this->logger->info("Fetching game", ['id' => $id]);

        try {
            $games = $this->gameRepository->loadById($id);
        } catch (NotFoundException $e) {
            throw new HttpNotFoundException($request, 'Game not found', $e);
        }
        ...
    }
}
```

this is the first time
start your computer. If
ese steps:

eck for viruses on your
rd drives or hard drive
make sure it is proper
n CHKDSK /F to check for
start your computer.

Rendering errors

CLEAR CHANNEL

1219

57 Street Station

Slim's error handling

Add Slim's error handling middleware to render exceptions

```
$displayDetails = true;  
$logErrors = true;  
$logErrorDetails = true;
```

```
$app->addErrorMiddleware($displayDetails, $logErrors, $logErrorDetails);
```



Error rendering

```
$ http -j DELETE http://localhost:8888/game
HTTP/1.1 405 Method Not Allowed
Allow: GET
Content-type: application/json
{
  "exception": [
    {
      "code": 405,
      "file": ".../Slim/Middleware/RoutingMiddleware.php",
      "message": "Method not allowed: Must be one of: GET",
      "type": "Slim\\Exception\\HttpMethodNotAllowedException"
    }
  ],
  "message": "Method not allowed: Must be one of: GET"
}
```

Raise your own

```
use App\Model\GameNotFoundException;
use Slim\Exception\HttpNotFoundException;
use Slim\Exception\HttpInternalServerErrorException;

public function handle(Request $request): ResponseInterface
{
    $id = $request->getAttribute('id');
    try {
        $game = $this->gameRepository->loadById($id);
    } catch (GameNotFoundException $e) {
        throw new HttpNotFoundException($request, 'Game not found', $e);
    } catch (\Exception $e) {
        throw new HttpInternalServerErrorException($request,
            'An unknown error occurred', $e);
    }
}
```


Not found error

```
use App\Model\GameNotFoundException;
use Slim\Exception\HttpNotFoundException;
use Slim\Exception\HttpInternalServerErrorException;

public function handle(Request $request): ResponseInterface
{
    $id = $request->getAttribute('id');
    try {
        $games = $this->gameRepository->loadById($id);
    } catch (GameNotFoundException $e) {
        throw new HttpNotFoundException($request, 'Game not found', $e);
    } catch (\Exception $e) {
        throw new HttpInternalServerErrorException($request,
            'An unknown error occurred', $e);
    }
}
```

Not found error

```
$ http -j http://localhost:8888/games/1234  
HTTP/1.1 404 Not Found  
Content-type: application/json
```

```
{  
  "message": "Game not found"  
}
```

(With `$displayDetails = false`)

Generic error

```
use App\Model\GameNotFoundException;
use Slim\Exception\HttpNotFoundException;
use Slim\Exception\HttpInternalServerErrorException;

public function handle(Request $request): ResponseInterface
{
    $id = $request->getAttribute('id');
    try {
        $games = $this->gameRepository->loadById($id);
    } catch (GameNotFoundException $e) {
        throw new HttpNotFoundException($request, 'Game not found', $e);
    } catch (\Exception $e) {
        throw new HttpInternalServerErrorException($request,
            'An unknown error occurred', $e);
    }
}
```

Generic error

```
$ http -j http://localhost:8888/games/abcd
HTTP/1.1 500 Internal Server Error
Content-type: application/json
```

```
{
  "exception": [
    {
      "code": 500,
      "file": ".../src/Handler/GetGameHandler.php",
      "line": 43,
      "message": "An unknown error occurred",
      "type": "Slim\\Exception\\HttpInternalServerErrorException"
    },
  ],
}
```

```
{
    "code": 40,
    "file": ".../lib/Assert/Assertion.php",
    "line": 2752,
    "message": "Value \"abcd\" is not a valid integer.",
    "type": "Assert\\InvalidArgumentException"
},
"message": "An unknown error occurred"
}
```

(With \$displayDetails = true)

A photograph of a space shuttle launching at dusk or dawn. The shuttle is ascending vertically, leaving a large, bright, billowing plume of white smoke and fire. Three tall, slender service towers are visible in the background, flanking the launch pad. The sky is a deep, dark blue, and the horizon shows a faint glow of light. The overall scene is dramatic and captures the power of the launch.

To sum up

Resources

- <http://slimframework.com>
- <https://akrabat.com/category/slim-framework/>
- <https://github.com/akrabat/slim4-rps-api>
- <https://github.com/akrabat/slim4-starter>



Thank you!

Rob Allen - <http://akrabat.com> - @akrabat

Photo credits

- The Fat Controller: HiT Entertainment
- Foundation: <https://www.flickr.com/photos/armchairbuilder/6196473431>
- APIs: <https://www.flickr.com/photos/ebothy/15723500675>
- Verbs: <https://www.flickr.com/photos/160866001@N07/45904136621/>
- Incoming Data: <https://www.flickr.com/photos/natspressoffice/13085089605>
- Computer code: <https://www.flickr.com/photos/n3wjack/3856456237/>
- Road sign: <https://www.flickr.com/photos/ell-r-brown/6804246004>
- Car crash: EuroNCAP
- Writing: <https://www.flickr.com/photos/froderik/9355085596/>
- Error screen: <https://www.flickr.com/photos/thirdrail/18126260>
- Rocket launch: <https://www.flickr.com/photos/gsfc/16495356966>
- Stars: <https://www.flickr.com/photos/gsfc/19125041621>