WellRESTed Documentation

Release 5.0.0

PJ Dietz

Dec 28, 2022

Contents

WellRESTed is a library for creating RESTful APIs and websites in PHP that provides abstraction for HTTP messages, a powerful handler and middleware system, and a flexible router.

CHAPTER 1

Features

1.1 PSR-7 HTTP Messages

Request and response messages are built to the interfaces standardized by PSR-7 making it easy to share code and use components from other libraries and frameworks.

The message abstractions facilitate working with message headers, status codes, variables extracted from the path, message bodies, and all the other aspects of requests and responses.

1.2 PSR-15 Handler Interfaces

WellRESTed can use handlers and middleware using the interfaces defined by the PSR-15 standard.

1.3 Router

The router allows you to define your endpoints using URI Templates like `/foo/{bar}/{baz}` that match patterns of paths and provide captured variables. You can also match exact paths for extra speed or regular expressions for extra flexibility.

WellRESTed's router automates responding to `OPTIONS` requests for each endpoint based on the methods you assign. `405 Method Not Allowed` responses come free of charge as well for any methods you have not implemented on a given endpoint.

1.4 Middleware

The middleware system allows you to build your Web service out of discrete, modular pieces. These pieces can be run in sequences where each has an opportunity to work with the request before handing it off to the next. For example, an authenticator can validate a request and forward it to a cache; the cache can check for a stored representation

and forward to another middleware if no cached representation is found, etc. All of this happens without any one middleware needing to know anything about where it is in the chain or which middleware comes before or after.

1.5 Lazy Loading

Handlers and middleware can be registered using factory functions so that they are only instantiated if needed. This way, a Web service with hundreds of handlers and middleware only creates instances required for the current request-response cycle.

1.6 Extensible

Most classes are coded to interfaces to allow you to provide your own implementations and use them in place of the built-in classes. For example, if your Web service needs to be able to dispatch middleware that implements a third-party interface, you can provide your own custom `DispatcherInterface` implementation.

CHAPTER 2

Example

Here's a customary "Hello, world!" example. This site will respond to requests for `GET /hello` with "Hello, world!" and provide custom responses for other paths (e.g., `GET /hello/Molly` will respond "Hello, Molly!").

The site will also provide an `X-example: hello world` using dedicated middleware, just to illustrate how middleware propagates.

```php
<?php

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
use Psr\Http\Server\RequestHandlerInterface;
use WellRESTed\Message\Response;
use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once 'vendor/autoload.php';

// Create a handler that will construct and return a response. We'll
// register this handler with a server and router below.
class HelloHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // Check for a "name" attribute which may have been provided as a
        // path variable. Use "world" as a default.
        $name = $request->getAttribute("name", "world");

        // Set the response body to the greeting and the status code to 200 OK.
        $response = (new Response(200))
            ->withHeader("Content-type", "text/plain")
            ->withBody(new Stream("Hello, $name!"));

        // Return the response.
```

(continues on next page)

(continued from previous page)

```
        return $response;
    }
}

// Create middleware that will add a custom header to every response.
class CustomerHeaderMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface {

        // Delegate to the next handler in the chain to obtain a response.
        $response = $handler->handle($request);

        // Add the header.
        $response = $response->withHeader("X-example", "hello world");

        // Return the altered response.
        return $response;
    }
}

// Create a server
$server = new Server();

// Add the header-adding middleware to the server first so that it will
// forward requests on to the router.
$server->add(new CustomerHeaderMiddleware());

// Create a router to map methods and endpoints to handlers.
$router = $server->createRouter();

$handler = new HelloHandler();
// Register a route to the handler without a variable in the path.
$router->register('GET', '/hello', $handler);
// Register a route that reads a "name" from the path.
// This will make the "name" request attribute available to the handler.
$router->register('GET', '/hello/{name}', $handler);
$server->add($router);

// Read the request from the client, dispatch, and output.
$server->respond();
```

CHAPTER 3

Contents

3.1 Overview

3.1.1 Installation

The recommended method for installing WellRESTed is to use Composer. Add an entry for WellRESTed in your project's `composer.json` file.

```
{
   "require": {
      "wellrested/wellrested": "^5"
   }
}
```

3.1.2 Requirements

- PHP 7.3

3.1.3 License

Licensed using the MIT license.

3.2 Getting Started

This page provides a brief introduction to WellRESTed. We'll take a tour of some of the features of WellRESTed without getting into too much depth.

To start, we'll make a "Hello, world!" to demonstrate the concepts of handlers and routing and show how to read variables from the request path.

3.2.1 Hello, World!

Let's start with a very basic "Hello, world!" Here, we will create a server. A `WellRESTed\Server` reads the incoming request from the client, dispatches a handler, and transmits a response back to the client.

Our handler will create and return a response with the status code set to `200` and the body set to "Hello, world!".

Example 1: Simple "Hello, world!"

```
<?php

use Psr\Http\Server\RequestHandlerInterface;
use WellRESTed\Message\Response;
use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once 'vendor/autoload.php';

// Define a handler implementing the PSR-15 RequestHandlerInterface interface.
class HelloHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        $response = (new Response(200))
            ->withHeader('Content-type', 'text/plain')
            ->withBody(new Stream('Hello, world!'));
        return $response;
    }
}

// Create a new server.
$server = new Server();

// Add this handler to the server.
```

(continues on next page)

(continued from previous page)

```
$server->add(new HelloHandler());

// Read the request sent to the server and use it to output a response.
$server->respond();
```

Note: The handler in this example provides a `Stream` as the body instead of a string. This is a feature or PSR-7 where HTTP message bodies are always represented by streams. This allows you to work with very large bodies without having to store the entire contents in memory.

WellRESTed provides `Stream` and `NullStream`, but you can use any implementation of `Psr\Http\Message\StreamInterface`.

3.2.2 Routing by Path

This is a good start, but it provides the same response to every request. Let's provide this response only when a client sends a request to `/hello`.

For this, we need a router. A router examines the request and sends the request through to the handler that matches the request's HTTP method and path.

Example 2: Routed "Hello, world!"

```
// Create a new server.
$server = new Server();

// Create a router to map methods and endpoints to handlers.
$router = $server->createRouter();
$router->register('GET', '/hello', new HelloHandler());
$server->add($router);

// Read the request sent to the server and use it to output a response.
$server->respond();
```

3.2.3 Reading Path Variables

Routes can be static (like the one above that matches only `/hello`), or they can be dynamic. Here's an example that uses a dynamic route to read a portion from the path to use as the greeting. For example, a request to `/hello/Molly` will respond "Hello, Molly", while a request to `/hello/Oscar` will respond "Hello, Oscar!"

Example 3: Personalized "Hello, world!"

```
class HelloHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // Check for a "name" attribute which may have been provided as a
        // path variable. Use "world" as a default.
        $name = $request->getAttribute("name", "world");
```

(continues on next page)

(continued from previous page)

```
        // Set the response body to the greeting and the status code to 200 OK.
        $response = (new Response(200))
            ->withHeader("Content-type", "text/plain")
            ->withBody(new Stream("Hello, $name!"));

        // Return the response.
        return $response;
    }
}

// Create the server and router.
$server = new Server();
$router = $server->createRouter();

// Register the middleware for an exact match to /hello
$router->register("GET", "/hello", $hello);
// Register to match a pattern with a variable.
$router->register("GET", "/hello/{name}", $hello);
$server->add($router);

$server->respond();
```

3.2.4 Middleware

In addition to handlers, which provide responses directly, WellRESTed also supports middleware to act on the requests and then pass them on for other middleware or handlers to work with.

Middleware allows you to compose your application in multiple pieces. In the example, we'll use middleware to add a header to every response, regardless of which handler is called.

```
// This middleware will add a custom header to every response.
class CustomHeaderMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface {

        // Delegate to the next handler in the chain to obtain a response.
        $response = $handler->handle($request);

        // Add the header to the response we got back from upstream.
        $response = $response->withHeader("X-example", "hello world");

        // Return the altered response.
        return $response;
    }
}

// Create a server
$server = new Server();

// Add the header-adding middleware to the server first so that it will
// forward requests on to the router.
```

(continues on next page)

(continued from previous page)

```
$server->add(new CustomHeaderMiddleware());

// Create a router to map methods and endpoints to handlers.
$router = $server->createRouter();

$handler = new HelloHandler();
// Register a route to the handler without a variable in the path.
$router->register('GET', '/hello', $handler);
// Register a route that reads a "name" from the path.
// This will make the "name" request attribute available to the handler.
$router->register('GET', '/hello/{name}', $handler);
$server->add($router);

// Read the request from the client, dispatch, and output.
$server->respond();
```

3.3 Messages and PSR-7

WellRESTed uses the PSR-7 interfaces for HTTP messages. This section provides an introduction to working with these interfaces and the implementations provided with WellRESTed. For more information, please read about PSR-7.

3.3.1 Requests

The `$request` variable passed to handlers and middleware represents the request message sent by the client. You can inspect this variable to read information such as the request path, method, query, headers, and body.

Let's start with a very simple GET request to the path `/cats/?color=orange`.

```
GET /cats/?color=orange HTTP/1.1
Host: example.com
Cache-control: no-cache
```

You can read information from the request in your handler like this:

```
class MyHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        $path = $request->getRequestTarget();
        // "/cats/?color=orange"

        $method = $request->getMethod();
        // "GET"

        $query = $request->getQueryParams();
        /*
            Array
            (
                [color] => orange
            )
        */
    }
}
```

This example shows that you can use:

- `getRequestTarget()` to read the path and query string for the request
- `getMethod()` to read the HTTP verb (e.g., GET, POST, OPTIONS, DELETE)
- `getQueryParams()` to read the query as an associative array

3.3.1.1 Headers

The request above also included a `Cache-control: no-cache` header. You can read this header a number of ways. The simplest way is with the `getHeaderLine($name)` method.

Call `getHeaderLine($name)` and pass the case-insensitive name of a header. The method will return the value for the header, or an empty string if the header is not present.

```
class MyHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // This message contains a "Cache-control: no-cache" header.
        $cacheControl = $request->getHeaderLine("cache-control");
        // "no-cache"

        // This message does not contain any authorization headers.
        $authorization = $request->getHeaderLine("authorization");
        // ""
    }
}
```

Note: All methods relating to headers treat header field names case insensitively.

Because HTTP messages may contain multiple headers with the same field name, `getHeaderLine($name)` has one other feature: If multiple headers with the same field name are present in the message, `getHeaderLine($name)` returns a string containing all of the values for that field, concatenated by commas. This is more common with responses, particularly with the `Set-cookie` header, but is still possible for requests.

You may also use `hasHeader($name)` to test if a header exists, `getHeader($name)` to receive an array of values for this field name, and `getHeaders()` to receive an associative array of headers where each key is a field name and each value is an array of field values.

3.3.1.2 Body

PSR-7 provides access to the body of the request as a stream and—when possible—as a parsed object or array. Let's start by looking at a request with form fields made available as an array.

Parsed Body

For POST requests for forms (i.e., the `Content-type` header is either `application/x-www-form-urlencoded` or `multipart/form-data`), the request makes the form fields available via the `getParsedBody` method. This provides access to the fields without needing to rely on the `$_POST` superglobal.

Given this request:

```
POST /cats/ HTTP/1.1
Host: example.com
Content-type: application/x-www-form-urlencoded
Content-length: 23

name=Molly&color=Calico
```

We can read the parsed body like this:

```
class MyHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        $cat = $request->getParsedBody();
        /*
            Array
            (
                [name] => Molly
                [color] => calico
            )
        */
    }
}
```

Body Stream

For other content types, use the `getBody` method to get a stream containing the contents of request entity body.

Using a JSON representation of our cat, we can make a request like this:

```
POST /cats/ HTTP/1.1
Host: example.com
Content-type: application/json
Content-length: 46

{
    "name": "Molly",
    "color": "Calico"
}
```

We can read and parse the JSON body, and even provide it as the parsedBody for later middleware or handlers like this:

```
class JsonParser implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface
    {
        // Parse the body.
        $cat = json_decode((string) $request->getBody());
        /*
            stdClass Object
            (
                [name] => Molly
```

(continues on next page)

(continued from previous page)

```
                [color] => calico
            )
        */
        // Add the parsed JSON to the request.
        $request = $request->withParsedBody($cat);
        // Send the request to the next handler.
        return $handler->handle($request);
    }
}
```

Because the entity body of a request or response can be very large, PSR-7 represents bodies as streams using the `Psr\Http\Message\StreamInterface` (see PSR-7 Section 1.3).

The JSON example casts the stream to a string, but we can also do things like copy the stream to a local file:

```
// Store the body to a temp file.
$chunkSize = 2048; // Number of bytes to read at once.
$localPath = tempnam(sys_get_temp_dir(), "body");
$h = fopen($localPath, "wb");
$body = $request->getBody();
while (!$body->eof()) {
    fwrite($h, $body->read($chunkSize));
}
fclose($h);
```

3.3.1.3 Parameters

PSR-7 eliminates the need to read from many of the superglobals. We already saw how `getParsedBody` takes the place of reading directly from `$_POST` and `getQueryParams` replaces reading from `$_GET`. Here are some other `ServerRequestInterface` methods with brief descriptions. Please see PSR-7 for full details, particularly for `getUploadedFiles`.

Method	Replaces	Note
getServerParams	$_SERVER	Data related to the request environment
getCookieParams	$_COOKIE	Compatible with the structure of $_COOKIE
getQueryParams	$_GET	Deserialized query string arguments, if any
getParsedBody	$_POST	Request body as an object or array
getUploadedFiles	$_FILES	Normalized tree of file upload data

3.3.1.4 Attributes

`ServerRequestInterface` provides another useful feature called "attributes". Attributes are key-value pairs associated with the request that can be, well, pretty much anything.

The primary use for attributes in WellRESTed is to provide access to path variables when using template routes or regex routes.

For example, the template route `/cats/{name}` matches routes such as `/cats/Molly` and `/cats/Oscar`. When the route is dispatched, the router takes the portion of the actual request path matched by `{name}` and provides it as an attribute.

For a request to `/cats/Rufus`:

```
$name = $request->getAttribute("name");
// "Rufus"
```

When calling `getAttribute`, you can optionally provide a default value as the second argument. The value of this argument will be returned if the request has no attribute with that name.

```
// Request has no attribute "dog"
$name = $request->getAttribute("dog", "Bear");
// "Bear"
```

Middleware can also use attributes as a way to provide extra information to subsequent handlers. For example, an authorization middleware could obtain an object representing a user and store is as the "user" attribute which later middleware could read.

```
class AuthorizationMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface

        try {
            $user = $this->readUserFromCredentials($request);
        } catch (NoCredentialsSupplied $e) {
            return $response->withStatus(401);
        } catch (UserNotAllowedHere $e) {
            return $response->withStatus(403);
        }

        // Store this as an attribute.
        $request = $request->withAttribute("user", $user);

        // Delegate to the handler, passing the request with the "user" attribute.
        return $handler->handle($request);
    }
};

class SecureHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // Read the "user" attribute added by a previous middleware.
        $user = $request->getAttribute("user");

        // Do something with $user ...
    }
}

$server = new \WellRESTed\Server();
$server->add(new AuthorizationMiddleware());
$server->add(new SecureHandler()); // Must be added AFTER authorization to get "user"
$server->respond();
```

3.3.2 Responses

PSR-7 messages are immutable, so you will not be able to alter values of response properties. Instead,

`with*` methods provide ways to get a copy of the current message with updated properties. For example, `ResponseInterface::withStatus` returns a copy of the original response with the status changed.

```
// The original response has a 500 status code.
$response->getStatusCode();
// 500

// Replace this instance with a new instance with the status updated.
$response = $response->withStatus(200);
$response->getStatusCode();
// 200
```

Note: PSR-7 requests are immutable as well, and we used `withAttribute` and `withParsedBody` in a few of the examples in the Requests section.

Chain multiple `with` methods together fluently:

```
// Get a new response with updated status, headers, and body.
$response = (new Response())
    ->withStatus(200)
    ->withHeader("Content-type", "text/plain")
    ->withBody(new \WellRESTed\Message\Stream("Hello, world!");
```

3.3.2.1 Status

Provide the status code for your response with the `withStatus` method. When you pass a standard status code to this method, the WellRESTed response implementation will provide an appropriate reason phrase for you. For a list of reason phrases provided by WellRESTed, see the IANA HTTP Status Code Registry.

Note: The "reason phrase" is the text description of the status that appears in the status line of the response. The "status line" is the very first line in the response that appears before the first header.

Although the PSR-7 `ResponseInterface::withStatus` method accepts the reason phrase as an optional second parameter, you generally shouldn't pass anything unless you are using a non-standard status code. (And you probably shouldn't be using a non-standard status code.)

```
// Set the status and view the reason phrase provided.

$response = $response->withStatus(200);
$response->getReasonPhrase();
// "OK"

$response = $response->withStatus(404);
$response->getReasonPhrase();
// "Not Found"
```

3.3.2.2 Headers

Use the `withHeader` method to add a header to a response. `withHeader` will add the header if not already set, or replace the value of an existing header with the same name.

```
// Add a "Content-type" header.
$response = $response->withHeader("Content-type", "text/plain");
$response->getHeaderLine("Content-type");
// "text/plain"

// Calling withHeader a second time updates the value.
$response = $response->withHeader("Content-type", "text/html");
$response->getHeaderLine("Content-type");
// "text/html"
```

To set multiple values for a given header field name (e.g., for `Set-cookie` headers), call `withAddedHeader`. `withAddedHeader` adds the new header without altering existing headers with the same name.

```
$response = $response
    ->withHeader("Set-cookie", "cat=Molly; Path=/cats; Expires=Wed, 13 Jan 2021␣
↪22:23:01 GMT;")
    ->withAddedHeader("Set-cookie", "dog=Bear; Domain=.foo.com; Path=/; Expires=Wed,␣
↪13 Jan 2021 22:23:01 GMT;")
    ->withAddedHeader("Set-cookie", "hamster=Fizzgig; Domain=.foo.com; Path=/;␣
↪Expires=Wed, 13 Jan 2021 22:23:01 GMT;");
```

To check if a header exists or to remove a header, use `hasHeader` and `withoutHeader`.

```
// Check if a header exists.
$response->hasHeader("Content-type");
// true

// Clone this response without the "Content-type" header.
$response = $response->withoutHeader("Content-type");

// Check if a header exists.
$response->hasHeader("Content-type");
// false
```

3.3.2.3 Body

To set the body for the response, pass an instance implementing `Psr\Http\Message\Stream` to the `withBody` method.

```
$stream = new \WellRESTed\Message\Stream("Hello, world!");
$response = $response->withBody($stream);
```

WellRESTed provides two `Psr\Http\Message\Stream` implementations. You can use these, or any other implementation.

Stream

`WellRESTed\Message\Stream` wraps a file pointer resource and is useful for responding with a string or file.

When you pass a string to the constructor, the Stream instance uses php://temp as the file pointer resource. The string passed to the constructor is automatically stored to `php://temp`, and you can write more content to it using the `StreamInterface::write` method.

Note: `php://temp` stores the contents to memory, but switches to a temporary file once the amount of data stored hits a predefined limit (the default is 2 MB).

```
// Pass the beginning of the contents to the constructor as a string.
$body = new \WellRESTed\Message\Stream("Hello ");

// Append more contents.
$body->write("world!");

// Set the body and status code.
$response = (new Response())
    ->withStatus(200)
    ->withBody($body);
```

To respond with the contents of an existing file, use `fopen` to open the file with read access and pass the pointer to the constructor.

```
// Open the file with read access.
$resource = fopen("/home/user/some/file", "rb");

// Pass the file pointer resource to the constructor.
$body = new \WellRESTed\Message\Stream($resource);

// Set the body and status code.
$response = (new Response())
    ->withStatus(200)
    ->withBody($body);
```

NullStream

Each PSR-7 message MUST have a body, so there's no `withoutBody` method. You also cannot pass `null` to `withBody`. Instead, use a `WellRESTed\Messages\NullStream` to provide a very simple, zero-length, no-content body.

```
$response = (new Response())
    ->withStatus(200)
    ->withBody(new \WellRESTed\Message\NullStream());
```

3.4 Handlers and Middleware

WellRESTed allows you to define and use your handlers and middleware in a number of ways.

3.4.1 Defining Handlers and Middleware

3.4.1.1 PSR-15 Interfaces

The preferred method is to use the interfaces standardized by PSR-15. This standard includes two interfaces, `Psr\Http\Server\RequestHandlerInterface` and `Psr\Http\Server\MiddlewareInterface`.

Use `RequestHandlerInterface` for individual components that generate and return responses.

```
class HelloHandler implements RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // Check for a "name" attribute which may have been provided as a
        // path variable. Use "world" as a default.
        $name = $request->getAttribute("name", "world");

        // Set the response body to the greeting and the status code to 200 OK.
        $response = (new Response(200))
            ->withHeader("Content-type", "text/plain")
            ->withBody(new Stream("Hello, $name!"));

        // Return the response.
        return $response;
    }
}
```

Use `MiddlewareInterface` for classes that interact with other middleware and handlers. For example, you may have middleware that attempts to retrieve a cached response and delegates to other handlers on a cache miss.

```
class CacheMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface
    {

        // Inspect the request to see if there is a representation on hand.
        $representation = $this->getCachedRepresentation($request);
        if ($representation !== null) {
            // There is already a cached representation.
            // Return it without delegating to the next handler.
            return (new Response())
                ->withStatus(200)
                ->withBody($representation);
        }

        // No representation exists. Delegate to the next handler.
        $response = $handler->handle($request);

        // Attempt to store the response to the cache.
        $this->storeRepresentationToCache($response);

        return $response
    }

    private function getCachedRepresentation(ServerRequestInterface $request)
    {
        // Look for a cached representation. Return null if not found.
        // ...
    }

    private function storeRepresentationToCache(ResponseInterface $response)
    {
        // Ensure the response contains a success code, a valid body,
```

(continues on next page)

(continued from previous page)

```
        // headers that allow caching, etc. and store the representation.
        // ...
    }
}
```

3.4.1.2 Legacy Middleware Interface

Prior to PSR-15, WellRESTed's recommended handler interface was `WellRESTed\MiddlewareInterface`. This interface is still supported for backwards compatibility.

This interface serves for both handlers and middleware. It differs from the `Psr\Http\Server\MiddlewareInterface` in that is expects an incoming `$response` parameter which you may use to generate the returned response. It also expected a `$next` parameter which is a `callable` with this signature:

```
function next($request, $response): ResponseInterface
```

Call `$next` and pass `$request` and `$response` to forward the request to the next handler. `$next` will return the response from the handler. Here's the cache example above as a `WellRESTed\MiddlewareInterface`.

```
class CacheMiddleware implements WellRESTed\MiddlewareInterface
{
    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response,
        $next
    ) {

        // Inspect the request to see if there is a representation on hand.
        $representation = $this->getCachedRepresentation($request);
        if ($representation !== null) {
            // There is already a cached representation.
            // Return it without delegating to the next handler.
            return $response
                ->withStatus(200)
                ->withBody($representation);
        }

        // No representation exists. Delegate to the next handler.
        $response = $next($request, $response);

        // Attempt to store the response to the cache.
        $this->storeRepresentationToCache($response);

        return $response
    }

    private function getCachedRepresentation(ServerRequestInterface $request)
    {
        // Look for a cached representation. Return null if not found.
        // ...
    }

    private function storeRepresentationToCache(ResponseInterface $response)
    {
```

(continues on next page)

(continued from previous page)

```
        // Ensure the response contains a success code, a valid body,
        // headers that allow caching, etc. and store the representation.
        // ...
    }
}
```

3.4.1.3 Callables

You may also use a `callable` similar to the legacy `WellRESTed\MiddlewareInterface`. The signature of the callable matches the signature of `WellRESTed\MiddlewareInterface::__invoke`.

```
$handler = function ($request, $response, $next) {

    // Delegate to the next handler.
    $response = $next($request, $response);

    return $response
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, $name!"));
}
```

3.4.2 Using Handlers and Middleware

Methods that accept handlers and middleware (e.g., `Server::add`, `Router::register`) allow you to provide them in a number of ways. For example, you can provide an instance, a `callable` that returns an instance, or an `array` of middleware to use in sequence. The following examples will demonstrate all of the ways you can register handlers and middleware.

3.4.2.1 Factory Functions

The best method is to use a function that returns an instance of your handler. The main benefit of this approach is that no handlers are instantiated until they are needed.

```
$router->register("GET,PUT,DELETE", "/widgets/{id}",
    function () { return new App\WidgetHandler() }
);
```

If you're using `Pimple`, a popular dependency injection container for PHP, you may have code that looks like this:

```
// Create a DI container.
$c = new Container();
// Store a function to the container that will create and return the handler.
$c['widgetHandler'] = $c->protect(function () use ($c) {
    return new App\WidgetHandler();
});

$router->register("GET,PUT,DELETE", "/widgets/{id}", $c['widgetHandler']);
```

3.4.2.2 Instance

WellRESTed also allows you to pass an instance of a handler directly. This may be useful for smaller handlers that don't require many dependencies, although the factory function approach is better in most cases.

```
$widgetHandler = new App\WidgetHandler();

$router->register("GET,PUT,DELETE", "/widgets/{id}", $widgetHandler);
```

> **Warning:** This is simple, but has a significant disadvantage over the other options because each middleware used this way will be loaded and instantiated, even if it's not needed for a given request-response cycle. You may find this approach useful for testing, but avoid if for production code.

3.4.2.3 Fully Qualified Class Name (FQCN)

For handlers that do not require any arguments passed to the constructor, you may pass the fully qualified class name of your handler as a `string`. You can do that like this:

```
$router->register('GET,PUT,DELETE', '/widgets/{id}', App\WidgetHandler::class);
// ... or ...
$router->register('GET,PUT,DELETE', '/widgets/{id}', 'App\\WidgetHandler');
```

The class is not loaded, and no instances are created, until the route is matched and dispatched. However, the drawback to this approach is the there is no way to pass any arguments to the constructor.

3.4.2.4 Array

The final approach is to provide a sequence of middleware and a handler as an `array`.

For example, imagine if we had a Pimple container with these services:

```
$c['authMiddleware'] // Ensures the user is logged in
$c['cacheMiddleware'] // Provides a cached response if able
$c['widgetHandler']  // Provides a widget representation
```

We could provide these as a sequence by using an `array`.

```
$router->register('GET', '/widgets/{id}', [
    $c['authMiddleware'],
    $c['cacheMiddleware'],
    $c['widgetHandler']
]);
```

3.5 Router

A router is a type of middleware that organizes the components of a site by associating HTTP methods and paths with handlers and middleware. When the router receives a request, it examines the path components of the request's URI, determines which "route" matches, and dispatches the associated handler. The dispatched handler is then responsible for reacting to the request and providing a response.

3.5.1 Basic Usage

Typically, you will want to use the `WellRESTed\Server::createRouter` method to create a `Router`.

```
$server = new WellRESTed\Server();
$router = $server->createRouter();
```

Suppose `$catHandler` is a handler that you want to dispatch whenever a client makes a `GET` request to the path `/cats/`. Use the `register` method map it to that path and method.

```
$router->register("GET", "/cats/", $catHandler);
```

The `register` method is fluent, so you can add multiple routes in either of these styles:

```
$router->register("GET", "/cats/", $catReader);
$router->register("POST", "/cats/", $catWriter);
$router->register("GET", "/cats/{id}", $catItemReader);
$router->register("PUT,DELETE", "/cats/{id}", $catItemWriter);
```

. . . Or. . .

```
$router
    ->register("GET", "/cats/", $catReader)
    ->register("POST", "/cats/", $catWriter)
    ->register("GET", "/cats/{id}", $catItemReader)
    ->register("PUT,DELETE", "/cats/{id}", $catItemWriter);
```

3.5.2 Paths

A router can map a handler to an exact path, or to a pattern of paths.

3.5.2.1 Static Routes

The simplest type of route is called a "static route". It maps a handler to an exact path.

```
$router->register("GET", "/cats/", $catHandler);
```

This route will map a request to `/cats/` and only `/cats/`. It will **not** match requests to `/cats` or `/cats/molly`.

3.5.2.2 Prefix Routes

The next simplest type of route is a "prefix route". A prefix route matches requests by the beginning of the path.

To create a "prefix handler", include `*` at the end of the path. For example, this route will match any request that begins with `/cats/`.

```
$router->register("GET", "/cats/*", $catHandler);
```

3.5.2.3 Template Routes

Template routes allow you to provide patterns for paths with one or more variables (sections surrounded by curly braces) that will be extracted.

For example, this template will match requests to `/cats/12`, `/cats/molly`, etc.,

```
$router->register("GET", "/cats/{cat}", $catHandler);
```

When the router dispatches a route matched by a template route, it provides the extracted variables as request attributes. To access a variable, call the request object's `getAttribute` method and pass the variable's name.

For a request to `/cats/molly`:

```
$name = $request->getAttribute("cat");
// "molly"
```

Template routes are very powerful, and this only scratches the surface. See URI Templates for a full explanation of the syntax supported.

3.5.2.4 Regex Routes

You can also use regular expressions to describe route paths.

```
$router->register("GET", "~cats/(?<name>[a-z]+)-(?<number>[0-9]+)~", $catHandler);
```

When using regular expression routes, the attributes will contain the captures from preg_match.

For a request to `/cats/molly-90`:

```
$vars = $request->getAttributes();
/*
Array
(
    [0] => cats/molly-12
    [name] => molly
    [1] => molly
    [number] => 12
    [2] => 12
)
*/
```

3.5.2.5 Route Priority

A router will often contain many routes, and sometimes more than one route will match for a given request. When the router looks for a matching route, it performs these checks in order.

1. If there is a static route with exact match to path, dispatch it.
2. If one prefix route matches the beginning of the path, dispatch it.
3. If multiple prefix routes match, dispatch the longest matching prefix route.
4. Inspect each pattern route (template and regular expression) in the order in which they were added to the router. Dispatch the first route that matches.
5. If no pattern routes match, return a response with a `404 Not Found` status. (**Note:** This is the default behavior. To configure a router to delegate to the next middleware when no route matches, call the router's `continueOnNotFound()` method.)

Static vs. Prefix

Consider these routes:

```
$router
    ->register("GET", "/cats/", $static);
    ->register("GET", "/cats/*", $prefix);
```

The router will dispatch a request for `/cats/` to `$static` because the static route `/cats/` has priority over the prefix route `/cats/*`.

The router will dispatch a request to `/cats/maine-coon` to `$prefix` because it is not an exact match for `/cats/`, but it does begin with `/cats/`.

Prefix vs. Prefix

Given these routes:

```
$router
    ->register("GET", "/dogs/*", $short);
    ->register("GET", "/dogs/sporting/*", $long);
```

A request to `/dogs/herding/australian-shepherd` will be dispatched to `$short` because it matches `/dogs/*`, but does not match `/dogs/sporting/*`

A request to `/dogs/sporing/flat-coated-retriever` will be dispatched to `$long` because it matches both routes, but `/dogs/sporting` is longer.

Prefix vs. Pattern

Given these routes:

```
$router
    ->register("GET", "/dogs/*", $prefix);
    ->register("GET", "/dogs/{group}/{breed}", $pattern);
```

`$pattern` will **never** be dispatched because any route that matches `/dogs/{group}/{breed}` also matches `/dogs/*`, and prefix routes have priority over pattern routes.

Pattern vs. Pattern

When multiple pattern routes match a path, the first one that was added to the router will be the one dispatched. **Be careful to add the specific routes before the general routes.** For example, say you want to send traffic to two similar looking URIs to different handlers based whether the variables were supplied as numbers or letters—`/dogs/102/132` should be dispatched to `$numbers`, while `/dogs/herding/australian-shepherd` should be dispatched to `$letters`.

This will work:

```
// Matches only when the variables are digits.
$router->register("GET", "~/dogs/([0-9]+)/([0-9]+)", $numbers);
// Matches variables with any unreserved characters.
$router->register("GET", "/dogs/{group}/{breed}", $letters);
```

This will **NOT** work:

```
// Matches variables with any unreserved characters.
$router->register("GET", "/dogs/{group}/{breed}", $letters);
// Matches only when the variables are digits.
$router->register("GET", "~/dogs/([0-9]+)/([0-9]+)", $numbers);
```

This is because `/dogs/{group}/{breed}` will match both `/dogs/102/132` **and** `/dogs/herding/australian-shepherd`. If it is added to the router before the route for `$numbers`, it will be dispatched before the route for `$numbers` is ever evaluated.

3.5.3 Methods

When you register a route, you can provide a specific method, a list of methods, or a wildcard to indicate any method.

3.5.3.1 Registering by Method

Specify a specific handler for a path and method by including the method as the first parameter.

```
// Dispatch $dogCollectionReader for GET requests to /dogs/
$router->register("GET", "/dogs/", $dogCollectionReader);

// Dispatch $dogCollectionWriter for POST requests to /dogs/
$router->register("POST", "/dogs/", $dogCollectionWriter);
```

3.5.3.2 Registering by Method List

Specify the same handler for multiple methods for a given path by proving a comma-separated list of methods as the first parameter.

```
// Dispatch $catCollectionHandler for GET and POST requests to /cats/
$router->register("GET,POST", "/cats/", $catCollectionHandler);

// Dispatch $catItemReader for GET requests to /cats/12, /cats/12, etc.
$router->register("GET", "/cats/{id}", $catItemReader);

// Dispatch $catItemWriter for PUT, and DELETE requests to /cats/12, /cats/12, etc.
$router->register("PUT,DELETE", "/cats/{id}", $catItemWriter);
```

3.5.3.3 Registering by Wildcard

Specify a handler for all methods for a given path by proving a * wildcard.

```
// Dispatch $guineaPigHandler for all requests to /guinea-pigs/, regardless of method.
$router->register("*", "/guinea-pigs/", $guineaPigHandler);

// Use $hamstersHandler by default for requests to /hamsters/
$router->register("*", "/hamsters/", $hamstersHandler);

// Provide a specific handler for POST /hamsters/
$router->register("POST", "/hamsters/", $hamstersPostOnly);
```

Note: The wildcard `*` can be useful, but be aware that the associated middleware will need to manage `HEAD` and `OPTIONS` requests, whereas this is done automatically for non-wildcard routes.

3.5.3.4 HEAD

Any route that supports `GET` requests will automatically support `HEAD`. You don't need to provide any specific middleware for `HEAD`, and you usually shouldn't. (Although you can if you want.)

For most cases, just implement `GET`, and the webserver will manage suppressing the response body for you.

3.5.3.5 OPTIONS, 405 Responses, and Allow Headers

When you add routes to a router by method, the router automatically provides responses for `OPTIONS` requests. For example, given this route:

```
// Dispatch $catItemReader for GET requests to /cats/12, /cats/12, etc.
$router->register("GET", "/cats/{id}", $catItemReader);

// Dispatch $catItemWriter for PUT, and DELETE requests to /cats/12, /cats/12, etc.
$router->register("PUT,DELETE", "/cats/{id}", $catItemWriter);
```

An `OPTIONS` request to `/cats/12` will provide a response like:

```
HTTP/1.1 200 OK
Allow: GET,PUT,DELETE,HEAD,OPTIONS
```

Likewise, a request to an unsupported method will return a `405 Method Not Allowed` response with a descriptive `Allow` header.

A `POST` request to `/cats/12` will provide:

```
HTTP/1.1 405 Method Not Allowed
Allow: GET,PUT,DELETE,HEAD,OPTIONS
```

3.5.4 Error Responses

Then a router is able to locate a route that matches the path, but that route doesn't support the request's method, the router will respond `405 Method Not Allowed`.

When a router is unable to match the route, it will delegate to the next middleware.

Note: When no route matches, the Router will delegate to the next middleware in the server. This is a change from previous versions of WellRESTed where there Router would return a 404 Not Found response. This new behaviour allows a servers to have multiple routers.

3.5.5 Router-specific Middleware

WellRESTed allows a Router to have a set of middleware to dispatch whenever it finds a route that matches. This middleware runs before the handler for the matched route, and only when a route matches.

This feature allows you to build a site where some sections use certain middleware and other do not. For example, suppose your site has a public section that does not require authentication and a private section that does. We can use a different router for each section, and provide authentication middleware on only the router for the private area.

```
$server = new Server();

// Add the "public" router.
$public = $server->createRouter();
$public->register('GET', '/', $homeHandler);
$public->register('GET', '/about', $homeHandler);
// Set the router call the next middleware when no route matches.
$public->continueOnNotFound();
$server->add($public);

// Add the "private" router.
$private = $server->createRouter();
// Authorization middleware checks for an Authorization header and
// responds 401 when the header is missing or invalid.
$private->add($authorizationMiddleware);
$private->register('GET', '/secret', $secretHandler);
$private->register('GET', '/members-only', $otherHandler);
$server->add($private);

$server->respond();
```

3.5.6 Nested Routers

For large Web services with large numbers of endpoints, a single, monolithic router may not to optimal. To avoid having each request test every pattern-based route, you can break up a router into a hierarchy of routers.

Here's an example where all of the traffic beginning with `/cats/` is sent to one router, and all the traffic for endpoints beginning with `/dogs/` is sent to another.

```
$server = new Server();

$catRouter = $server->createRouter()
    ->register("GET", "/cats/", $catReader)
    ->register("POST", "/cats/", $catWriter)
    // ... many more endpoints starting with /cats/
    ->register("POST", "/cats/{cat}/photo/{gallery}/{width}x{height}.{extension}",
↪$catImageHandler);

$dogRouter = $server->createRouter()
    ->register("GET,POST", "/dogs/", $dogHandler)
    // ... many more endpoints starting with /dogs/
    ->register("POST", "/dogs/{dog}/photo/{gallery}/{width}x{height}.{extension}",
↪$dogImageHandler);

$server->add($server->createRouter()
    ->register("*", "/cats/*", $catRouter)
    ->register("*", "/dogs/*", $dogRouter)
);

$server->respond();
```

3.6 URI Templates

WellRESTed allows you to register handlers with a router using URI Templates, based on the URI Templates defined in RFC 6570. These templates include variables (enclosed in curly braces) which are extracted and made available to the dispatched middleware.

3.6.1 Reading Variables

3.6.1.1 Basic Usage

Register a handler with a URI Template by providing a path that include at least one section enclosed in curly braces. The curly braces define variables for the template.

```
$router->register("GET", "/widgets/{id}", $widgetHandler);
```

The router will match requests for paths like `/widgets/12` and `/widgets/mega-widget` and dispatch `$widgetHandler` with the extracted variables made available as request attributes.

To read a path variable, router inspects the request attribute named `"id"`, since `id` is what appears inside curly braces in the URI template.

```
// For a request to /widgets/12
$id = $request->getAttribute("id");
// "12"

// For a request to /widgets/mega-widget
$id = $request->getAttribute("id");
// "mega-widget"
```

Note: Request attributes are a feature of the `ServerRequestInterface` provided by PSR-7.

3.6.1.2 Multiple Variables

The example above included one variable, but URI Templates may include multiple. Each variable will be provided as a request attribute, so be sure to give your variables unique names.

Here's an example with a handful of variables. Suppose we have a template describing the path for a user's avatar image. The image is identified by a username and the image dimensions.

```
$router->register("GET", "/avatars/{username}-{width}x{height}.jpg", $avatarHandlers);
```

A request for `GET /avatars/zoidberg-100x150.jpg` will provide these request attributes:

```
// Read the variables extracted form the path.
$username = $request->getAttribute("username");
// "zoidberg"
$width = $request->getAttribute("width");
// "100"
$height = $request->getAttribute("height");
// "150"
```

3.6.1.3 Arrays

You may also match a comma-separated series of values as an array using a URI Template by providing a * at the end of the variable name.

```
$router->register("GET", "/favorite-colors/{colors*}", $colorsHandler);
```

A request for `GET /favorite-colors/red,green,blue` will provide an array as the value for the `"colors"` request attribute.

```
$colorsHandler = function ($request, $response, $next) {
    // Read the variable extracted form the path.
    $colorsList = $request->getAttribute("colors");
    /*  Array
        (
            [0] => red
            [1] => green
            [2] => blue
        )
    */
};
```

3.6.2 Matching Characters

3.6.2.1 Unreserved Characters

By default, URI Template variables will match only "unreserved" characters. RFC 3968 Section 2.3 defines unreserved characters as alphanumeric characters, `-`, `.`, `_`, and `~`. All other characters must be percent encoded to be matched by a default template variable.

Note: Percent-encoded characters matched by template variables are automatically decoded when provided as request attributes.

Given the template `/users/{user}`, the following paths provide these values for `getAttribute("user")`:

Table 1: Paths and Values for the Template `/users/{user}`

Path	Value
/users/123	"123"
/users/zoidberg	"zoidberg"
/users/zoidberg%40planetexpress.com	"zoidberg@planetexpress.com"

A request for `GET /uses/zoidberg@planetexpress.com` will **not** match this template, because `@` is a reserved character and is not percent encoded.

3.6.2.2 Reserved Characters

If you need to match a non-percent-encoded reserved character like `@` or `/`, use the + operator at the beginning of the variable name.

Using the template `/users/{+user}`, we can match all of the paths above, plus `/users/zoidberg@planetexpress.com`.

Reserved matching also allows matching unencoded slashes (`/`). For example, given this template:

```
$router->register("GET", "/my-favorite-path{+path}", $pathHandler);
```

The router will dispatch `$pathHandler` with for a request to `GET /my-favorite-path/has/a/few/slashes.jpg`

```
$path = $request->getAttribute("path");
// "/has/a/few/slashes.jpg"
```

Note: Combine the `+` operator and `*` modifier to match reserved characters as an array. For example, the template `/{+vars*}` will match the path `/c@t,d*g`, providing the array `["c@t", "d*g"]`.

3.7 URI Templates (Advanced)

In URI Templates, we looked at the most common ways to use URI Templates. Here, we'll look at some of the extended syntaxes that URI Templates provide.

3.7.1 Path Components

To match a path component, include a slash `/` at the beginning of the variable expression. This instructs the template to match the variable if it:

- Begins with `/`
- Contains only unreserved and percent-encoded characters

You may also use the explode (`*`) modifier to match a variable number of path components and provide them as an array. When using the explode (`*`) modifier to match paths components, the `/` character serves as the delimiter instead of a comma.

Table 2: Matching path components

Template	Path	Match?	Attributes
{/path}	/hello.html	Yes	**path** "hello.html"
{/path}	/too/many/parts.jpg	No	
{/one}{/two}{/three}	/just/enough/parts.jpg	Yes	**one** "just" **two** "enough" **three** "parts.jpg"
{/path*}	/any/number/of/parts.jpg	Yes	**path** ["any", "number", "of", "parts.jpg"]
/image{/image*}.jpg	/image/with/any/path.jpg	Yes	**image** ["with", "any", "path"]

Note: The template `{/path}` fails to match the path `/too/many/parts.jpg`. Although the path does begin with a slash, the subsequent slashes are reserved characters, and therefore the match fails. To match a variable number of path components, use the explode `*` modifier (e.g, `{/paths*}`), or use the reserved (+) operator (e.g., `/{+paths}`).

3.7.2 Dot Prefixes

Dot prefixes work similarly to matching path components, but a dot `.` is the prefix character in place of a slash. This may be useful for file extensions, etc.

Including a dot `.` at the beginning of the variable expression instructs the template to match the variable if it:

- Begins with `.`
- Contains only unreserved (including `.`) and percent-encoded characters

You may also use the explode (`*`) modifier to match a variable number of dot-prefixed segments and store them to an array. When using the explode (`*`) modifier to match paths components, the `.` character serves as the delimiter instead of a comma.

Table 3: Matching dot prefixes

Template	Path	Match?	Attributes
/file{.ext}	/file.jpg	Yes	**ext** `"jpg"`
/file{.ext}	/file.tar.gz	Yes	**ext** `"tar.gz"`
/file{.ext1}{.ext2}	/file.tar.gz	Yes	**ext1** `"tar"` **ext2** `"gz"`
/file{.ext*}	/file.tar.gz	Yes	**ext** `["tar", "gz"]`

Note: Because `.` is an unreserved character, the template `/file{.ext}` matches the path `/file.tar.gz` and provides the value `"tar.gz"`. This is different from the behavior of the slash prefix, where an unexpected slash causes the match to fail.

3.7.3 Multiple-variable Expressions

An expression in a URI template may contain more than one variable. For example, the template `/aliases/{one},{two},{three}` can be written as `/aliases/{one,two,three}`.

The delimiter between the matched variables is the same as when matching with the explode (`*`) modifier:

Type	Delimiter
Simple String	Comma `,`
Reserved	Comma `,`
Path Components	Slash `/`
Dot Prefix	Dot `.`

Table 4: Multiple-variable expressions

Template	Path	Attributes
/{one,two,three}	/fry,leela,bender	**one** `"fry"` **two** `"leela"` **three** `"bender"`
/{one,two,three}	/fry,leela,Nixon%27s%20head	**one** `"fry"` **two** `"leela"` **three** `"Nixon's head"`
/{+one,two,three}	/fry,leela,Nixon's+head	**one** `"fry"` **two** `"leela"` **three** `"Nixon's head"`
/{/one,two,three}	/fry/leela/bender	**one** `"fry"` **two** `"leela"` **three** `"bender"`
/file{.one,two,three}	/file.fry.leela.bender	**one** `"fry"` **two** `"leela"` **three** `"bender"`

3.8 Extending and Customizing

WellRESTed is designed with customization in mind. This section describes some common scenarios for customization, starting with using a handler that implements a different interface.

3.8.1 Custom Handlers and Middleware

Imagine you found a handler class from a third party that does exactly what you need. The only problem is that it implements a different interface.

Here's the interface for the third-party handler:

```
interface OtherHandlerInterface
{
    /**
     * @param ServerRequestInterface $request
     * @return ResponseInterface
     */
    public function run(ResponseInterface $response);
}
```

3.8.1.1 Wrapping

One solution is to wrap an instance of this handler inside of a `Psr\Http\Server\RequestHandlerInterface` instance.

```
/**
 * Wraps an instance of OtherHandlerInterface
 */
class OtherHandlerWrapper implements RequestHandlerInterface
{
    private $handler;

    public function __construct(OtherHandlerInterface $handler)
    {
        $this->handler = $handler;
    }

    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        return $this->handler->run($request);
    }
}
```

3.8.1.2 Custom Dispatcher

Wrapping works well when you have one or two handlers implementing a third-party interface. If you want to integrate a lot of classes that implement a given third-party interface, you're might consider customizing the dispatcher.

The dispatcher is an instance that unpacks your handlers and middleware and sends the request and response through it. A default dispatcher is created for you when you use your `WellRESTed\Server`.

If you need the ability to dispatch other types of middleware, you can create your own by implementing `WellRESTed\Dispatching\DispatcherInterface`. The easiest way to do this is to subclass `WellRESTed\Dispatching\Dispatcher`. Here's an example that extends `Dispatcher` and adds support for `OtherHandlerInterface`:

```
/**
 * Dispatcher with support for OtherHandlerInterface
 */
class CustomDispatcher extends \WellRESTed\Dispatching\Dispatcher
{
    public function dispatch(
        $dispatchable,
        ServerRequestInterface $request,
        ResponseInterface $response,
        $next
    ) {
        try {
            // Use the dispatch method in the parent class first.
            $response = parent::dispatch($dispatchable, $request, $response, $next);
        } catch (\WellRESTed\Dispatching\DispatchException $e) {
            // If there's a problem, check if the handler or middleware
            // (the "dispatchable") implements OtherHandlerInterface.
            // Dispatch it if it does.
            if ($dispatchable instanceof OtherHandlerInterface) {
                $response = $dispatchable->run($request);
```

(continues on next page)

(continued from previous page)

```
            } else {
                // Otherwise, re-throw the exception.
                throw $e;
            }
        }
        return $response;
    }
}
```

To use this dispatcher, create an instance implementing `WellRESTed\Dispatching\DispatcherInterface` and pass it to the server's `setDispatcher` method.

```
$server = new WellRESTed\Server();
$server->setDispatcher(new MyApi\CustomDispatcher());
```

Warning: When you supply a custom Dispatcher, be sure to call `Server::setDispatcher` before you create any routers with `Server::createRouter` to allow the `Server` to pass you customer `Dispatcher` on to the newly created `Router`.

3.9 Dependency Injection

WellRESTed strives to play nicely with other code and not force developers into using any specific libraries or frameworks. As such, WellRESTed does not provide a dependency injection container, nor does it require you to use a specific container (or any).

This section describes the recommended way of using WellRESTed with Pimple, a common dependency injection container for PHP.

Imaging we have a `FooHandler` that depends on a `BarInterface`, and `BazInterface`. Our handler looks something like this:

```
class FooHandler implements RequestHandlerInterface
{
    private $bar;
    private $baz;

    public function __construct(BarInterface $bar, BazInterface $baz)
    {
        $this->bar = $bar;
        $this->baz = $baz;
    }

    public function handle(ServerRequestInterface $request): ResponseInterface
    {
        // Do something with the bar and baz and return a response...
        // ...
    }
}
```

We can register the handler and these dependencies in a Pimple service provider.

```
class MyServiceProvider implements ServiceProviderInterface
{
    public function register(Container $c)
    {
        // Register the Bar and Baz as services.
        $c['bar'] = function ($c) {
            return new Bar();
        };
        $c['baz'] = function ($c) {
            return new Baz();
        };

        // Register the Handler as a protected function. When you use
        // protect, Pimple returns the function itself instead of the return
        // value of the function.
        $c['fooHandler'] = $c->protect(function () use ($c) {
            return new FooHandler($c['bar'], $c['baz']);
        });
    }
}
```

To register this handler with a router, we can pass the service:

```
$router->register('GET', '/foo', $c['fooHandler']);
```

By "protecting" the `fooHandler` service, we are delaying the instantiation of the `FooHandler`, the `Bar`, and the `Baz` until the handler needs to be dispatched. This works because we're not passing instance of `FooHandler` when we register this with a router, we're passing a function to it that does the instantiation on demand.

3.10 Additional Components

The core WellRESTed library is designed to be very small and limited in scope. It should do only what's needed, and no more. One of WellRESTed's main goals is to stay small, and not force anything on consumers.

That being said, there are a number or situations that come up that warrant solutions. For that, WellRESTed also provides a (growing) number of companion packages that you may find useful, depending on the project.

HTTP Exceptions A collection of Exception classes that correspond to common HTTP error status codes.

Error Handling Classes to facilitate error handling including

Test Components Test cases and doubles for use with WellRESTed

Or, see WellRESTed on GitHub.

3.11 Web Server Configuration

You will typically want to have all traffic on your site directed to a single script that creates a `WellRESTed\Server` and calls `respond`. Here are basic setups for doing this in *Nginx* and *Apache*.

3.11.1 Nginx

```
server {

    listen 80;
    server_name your.hostname.here;
    root /your/sites/document/root;
    index index.php index.html;
    charset utf-8;

    # Attempt to serve actual files first.
    # If no file exists, send to /index.php
    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
        include fastcgi_params;
    }

}
```

3.11.2 Apache

```
RewriteEngine on
RewriteBase /

# Send all requests to non-regular files and directories to index.php
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^.+$ index.php [L,QSA]
```