WellRESTed Documentation

Release 3.0.0

PJ Dietz

Aug 03, 2017

Contents

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

CHAPTER 1

Features

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.

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.

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 a chance to modify the response 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.

Most middleware is never autoloaded or instantiated until it is needed, so a Web service with hundreds of middleware still only creates instances required for the current request-response cycle.

You can register middleware directly, register callables that return middleware (e.g., dependency container services), or register strings containing the middleware class names to autoload and instantiate on demand.

Extensible

All 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 different 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 WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once "vendor/autoload.php";

// Build some middleware. We'll register these with a server below.
// We're using callables to fit this all in one example, but these
// could also be classes implementing WellRESTed\MiddlewareInterface.

// Set the status code and provide the greeting as the response body.
$hello = function ($request, $response, $next) {

    // 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 = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, $name!"));

    // Propagate to the next middleware, if any, and return the response.
    return $next($request, $response);

};

// Add a header to the response.
```

```php
$headerAdder = function ($request, $response, $next) {
    // Add the header.
    $response = $response->withHeader("X-example", "hello world");
    // Propagate to the next middleware, if any, and return the response.
    return $next($request, $response);
};

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

// Start each request-response cycle by dispatching the header adder.
$server->add($headerAdder);

// The header adder will propagate to this router, which will dispatch the
// $hello middleware, possibly with a {name} variable.
$server->add($server->createRouter()
    ->register("GET", "/hello", $hello)
    ->register("GET", "/hello/{name}", $hello)
);

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

CHAPTER 3

Contents

Overview

Installation

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

```
{
   "require": {
      "wellrested/wellrested": "~3.0"
   }
}
```

Requirements

- PHP 5.4.0

License

Licensed using the MIT license.

> The MIT License (MIT)
>
> Copyright (c) 2015 PJ Dietz
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

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 middleware and routing and show how to read variables from the request path.

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 some middleware, and transmits a response back to the client.

Our middleware is a function that returns a response with the status code set to `200` and the body set to "Hello, world!".

Example 1: Simple "Hello, world!"

```php
<?php

use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once "vendor/autoload.php";

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

// Add middleware to dispatch that will return a response.
// In this case, we'll use an anonymous function.
$server->add(function ($request, $response, $next) {
    // Update the response with the greeting, status, and content-type.
    $response = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, world!"));
    // Use $next to forward the request on to the next middleware, if any.
    return $next($request, $response);
});

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

Note: The middleware 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`.

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 is a special type of middleware that examines the request and routes the request through to the middleware that matches.

Example 2: Routed "Hello, world!"

```
<?php

use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once "vendor/autoload.php";

// Create a new server and use it to create a new router.
$server = new Server();
$router = $server->createRouter();

// Map middleware to an endpoint and method(s).
$router->register("GET", "/hello", function ($request, $response, $next) {
    // Update the response with the greeting, status, and content-type.
    $response = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, world!"));
    // Use $next to forward the request on to the next middleware, if any.
    return $next($request, $response);
});

// Add the router to the server.
$server->add($router);

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

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!"

```
<?php

use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once "vendor/autoload.php";

// Define middleware.
$hello = function ($request, $response, $next) {

    // Check for a "name" attribute which may have been provided as a
    // path variable. The second parameters allows us to set a default.
    $name = $request->getAttribute("name", "world");

    // Update the response with the greeting, status, and content-type.
    $response = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, $name!"));

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

Multiple Middleware

One thing we haven't seen yet is how middleware work together. For the next example, we'll use an additional middleware that sets an `X-example: hello world`.

```
<?php

use WellRESTed\Message\Stream;
use WellRESTed\Server;

require_once "vendor/autoload.php";

// Set the status code and provide the greeting as the response body.
$hello = function ($request, $response, $next) {

    // 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 = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new Stream("Hello, $name!"));

    // Propagate to the next middleware, if any, and return the response.
    return $next($request, $response);

};

// Add a header to the response.
$headerAdder = function ($request, $response, $next) {
    // Add the header.
    $response = $response->withHeader("X-example", "hello world");
    // Propagate to the next middleware, if any, and return the response.
    return $next($request, $response);
};

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

// Add $headerAdder to the server first to make it the first to run.
$server->add($headerAdder);

// When $headerAdder calls $next, it will dispatch the router because it is
// added to the server right after.
$server->add($server->createRouter()
    ->register("GET", "/hello", $hello)
    ->register("GET", "/hello/{name}", $hello)
);

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

Messages and PSR-7

WellRESTed uses PSR-7 as the 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 PSR-7.

Obtaining Instances

When working with middleware, you generally will not need to create requests and responses yourself, as these are passed into the middleware when it is dispatched.

In Getting Started, we saw that middleware looks like this:

```
/**
 * @param Psr\Http\Message\ServerRequestInterface $request
 * @param Psr\Http\Message\ResponseInterface $response
 * @param callable $next
 * @return Psr\Http\Message\ResponseInterface
 */
function ($request, $response, $next) { }
```

When middleware is called, it receives a `Psr\Http\Message\ServerRequestInterface` instance representing the client's request and a `Psr\Http\Message\ResponseInterface` instance that serves as a starting place for the response to output to the client. These instances are created by the `WellRESTed\Server` when you call `WellRESTed\Server::respond`.

Note: If you want to provide your own custom request and response (either to adjust the initial settings or to use a different implementation), you can do so by passing request and response instances as the first and second parameters to `WellRESTed\Server::respond`.

Requests

The `$request` variable passed to middleware represents the request message sent by the client. Middleware 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/ HTTP/1.1
Host: example.com
Cache-control: no-cache
```

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

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

    $path = $request->getRequestTarget();
    // "/cats/?color=orange"

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

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

}
```

This example middleware 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

Let's move on to some more interesting features.

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.

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

    // 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 name 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.

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

When the request contains form fields (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:

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

    $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 like this:

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

    $cat = json_decode((string) $request->getBody());
    /*
        stdClass Object
        (
            [name] => Molly
            [color] => calico
        )
    */

    $request = $request->withParsedBody($cat);

}
```

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

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

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

    // 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 = $rqst->getBody();
    while (!$body->eof()) {
        fwrite($h, $body->read($chunkSize));
    }
    fclose($h);
```

```
}
```

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

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`:

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

    $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.

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

    // 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 middleware. For example, an authorization middleware could obtain an object representing a user and store is as the "user" attribute which later middleware could read.

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

```
    try {
        $user = readUserFromCredentials($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);

    // Call $next, passing the request with the added attribute.
    return $next($request, $response);

};

$subsequent = function ($request, $response, $next) {

    // Read the "user" attribute added by a previous middleware.
    $user = $request->getAttribute("user");

    // Do something with $user

}

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

Finally, attributes provide a nice way to provide a dependency injection container for to your middleware.

Responses

Initial Response

When you call `WellRESTed\Server::respond`, the server creates a "blank" response instance to pass to dispatched middleware. This response will have a `500 Internal Server Error` status, no headers, and an empty body.

You may wish to start each request-response cycle with a response with a different initial state, for example to include a custom header with all responses or to assume success and only change the status code on a failure (or non-`200` success). Here are two ways to provide this starting response:

Provide middleware as the first middleware that set the default conditions.

```
$initialResponsePrep = function ($rqst, $resp, $next) {
    // Set initial response and forward to subsequent middleware.
    $resp = $resp
        ->withStatus(200)
        ->withHeader("X-powered-by", "My Super Cool API v1.0.2")
    return $next($rqst, $resp);
};

$server = new \WellRESTed\Server();
$server->add($initialResponsePrep);
```

```
// ...add other middleware...
$server->respond();
```

Alternatively, instantiate a response and provide it to `WellRESTed\Server::respond`.

```
// Create an initial response. This can be any instance implementing
// Psr\Http\Message\ResponseInterface.
$response = new \WellRESTed\Message\Response(200, [
    "X-powered-by" => ["My Super Cool API v1.0.2"]]);

$server = new \WellRESTed\Server();
// ...add middleware middleware...
// Pass the response to respond()
$server->respond(null, $response);
```

Modifying

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 = $response
    ->withStatus(200)
    ->withHeader("Content-type", "text/plain")
    ->withBody(new \WellRESTed\Message\Stream("Hello, world!);
```

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"
```

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 that 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
```

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).

```
function ($rqst, $resp, $next) {

    // 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.
    $resp = $resp
        ->withStatus(200)
        ->withBody($body);

    // Forward to the next middleware.
    return $next($rqst, $resp);

}
```

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.

```
function ($rqst, $resp, $next) {

    // 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.
    $resp = $resp
        ->withStatus(200)
        ->withBody($body);
```

```
    // Forward to the next middleware.
    return $next($rqst, $resp);

}
```

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.

```
function ($rqst, $resp, $next) {

    // Set the body and status code.
    $resp = $resp
        ->withStatus(304)
        ->withBody(new \WellRESTed\Message\NullStream());

    // Forward to the next middleware.
    return $next($rqst, $resp);

}
```

Middleware

Okay, so what exactly **is** middleware? It's a nebulous term, and it's a bit reminiscent of the Underpants gnomes.

- Phase 1: Request
- Phase 2: ???
- Phase 3: Response

Middleware is indeed Phase 2. It's something (a callable or object) that takes a request and a response as inputs, does something with the response, and sends the altered response back out.

A Web service can built from many, many pieces of middleware, with each piece managing a specific task such as authentication or parsing representations. When each middleware runs, it is responsible for propagating the request through to the next middleware in the sequence—or deciding not to.

So what's it look like? In essence, a single piece of middleware looks something like this:

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

    // Update the response.
    /* $response = ... */

    // Determine if any other middleware should be called after this.
    if (/* Stop now without calling more middleware? */) {
        // Return the response without calling any other middleware.
        return $response;
    }

    // Let the next middleware work on the response. This propagates "up"
```

```
    // the chain of middleware, and will eventually return a response.
    $response = $next($request, $response);

    // Possibly update the response some more.
    /* $response = ... */

    // Return the response.
    return $response;

}
```

Defining Middleware

Middleware can be a callable (as in the Getting Started) or an implementation of the `WellRESTed\MiddlewareInterface` (which implements `__invoke` so is technically a callable, too).

Callable

```
/**
 * @param Psr\Http\Message\ServerRequestInterface $request
 * @param Psr\Http\Message\ResponseInterface $response
 * @param callable $next
 * @return Psr\Http\Message\ResponseInterface
 */
function ($request, $response, $next) { }
```

MiddlewareInterface

```
<?php

namespace WellRESTed;

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

interface MiddlewareInterface
{
    /**
     * @param ServerRequestInterface $request
     * @param ResponseInterface $response
     * @param callable $next
     * @return ResponseInterface
     */
    public function __invoke(ServerRequestInterface $request, ResponseInterface
↪$response, $next);
}
```

Using Middleware

Methods that accept middleware (e.g., `Server::add`, `Router::register`) allow you to provide middleware in a number of ways. For example, you can provide a string containing a class name, a middleware callable, a factory

callable, or even an array containing a sequence of middleware.

Fully Qualified Class Name (FQCN)

Assume your Web service has an autoloadable class named `Webservice\Widgets\WidgetHandler`. You can register it with a router by passing a string containing the fully qualified class name (FQCN):

```
$router->register("GET,PUT,DELETE", "/widgets/{id}", 'Webservice\Widgets\WidgetHandler
↪');
```

The class is not loaded, and no instances are created, until the route is matched and dispatched. Even for a router with 100 routes, no middleware registered by string name is loaded, except for the one that matches the request.

Factory Callable

You can also use a callable to instantiate and return a `MiddlewareInterface` instance or middleware callable.

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

This still delays instantiation, but gives you some added flexibility. For example, you could define middleware that receives some dependencies upon construction.

```
$container = new MySuperCoolDependencyContainer();

$router->add("GET,PUT,DELETE", "/widgets/{id}", function ()  use ($container) {
    return new \Webservice\Widgets\WidgetHandler($container["foo"], $container["baz
↪"]);
});
```

This is one approach to dependency injection.

Middleware Callable

Use a middleware callable directly.

```
$router->add("GET,PUT,DELETE", "/widgets/{id}", function ($request, $response, $next)
↪{
    $response = $response->withStatus(200)
        ->withHeader("Content-type", "text/plain")
        ->withBody(new \WellRESTed\Message\Stream("It's a bunch of widgets!"));
    return $next($request, $response);
});
```

Because `WellRESTed\MiddlewareInterface` has an `__invoke` method, implementing instances are also middleware callables. Assuming `WidgetHandler` implements `MiddelewareInterface`, you can do this:

```
$router->add("GET,PUT,DELETE", "/widgets/{id}", new␣
↪\Webservice\Widgets\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.

Array

Why use one middleware when you can use more?

Provide a sequence of middleware as an array. Each component of the array can be any of the varieties listed in this section.

When dispatched, the middleware in the array will run in order, with each calling the one following via the `$next` parameter.

```
$router->add("GET", "/widgets/{id}", ['Webservice\Auth', $jsonParser,
↪$widgetHandler]);
```

Chaining Middleware

Chaining middleware together allows you to build your Web service in a discrete, modular pieces. Each middleware in the chain makes the decision to either move the request up the chain by calling `$next`, or stop propagation by returning a response without calling `$next`.

Propagating Up the Chain

Imagine we want to add authorization to the `/widgets/{id}` endpoint. We can do this without altering the existing middleware that deals with the widget itself.

What we do is create an additional middleware that performs just the authorization task. This middleware will inspect the incoming request for authorization headers, and either move the request on up the chain to the next middleware if all looks good, or send a request back out with an appropriate status code.

Here's an example authorization middleware using pseudocode.

```
namespace Webservice;

class Authorization implements \WellRESTed\MiddlewareInterface
{
    public function __invoke(ServerRequestInterface $request, ResponseInterface
↪$response, $next)
    {
        // Validate the headers in the request.
        try {
            $validateUser($request);
        } catch (InvalidHeaderException $e) {
            // User did not supply the right headers.
            // Respond with a 401 Unauthorized status.
            return $response->withStatus(401);
        } catch (BadUserException $e) {
            // User is not permitted to access this resource.
            // Respond with a 403 Forbidden status.
            return $response->withStatus(403);
        }
```

```
        // No exception was thrown, so propagate to the next middleware.
        return $next($request, $response);
    }
}
```

We can add authorization for just the `/widgets/{id}` endpoint like this:

```
$server = new \WellRESTed\Server();
$server->add($server->createRouter()
    ->register("GET,PUT,DELETE",  "/widgets/{id}", [
        'Webservice\Authorization',
        'Webservice\Widgets\WidgetHandler'
    ])
    ->respond();
```

Or, if you wanted to use the authorization for the entire service, you can add it to the `Server` in front of the `Router`.

```
$server = new \WellRESTed\Server();
$server
    ->add('Webservice\Authorization')
    ->add($server->createRouter()
        ->register("GET,PUT,DELETE", "/widgets/{id}",
↪'Webservice\Widgets\WidgetHandler')
    )
    ->respond();
```

Moving Back Down the Chain

The authorization example returned `$next($request, $response)` immediately, but you can do some interesting things by working with the response that comes back from `$next`. Think of the request as taking a round trip on the subway with each middleware being a stop along the way. Each of the stops you go through going up the chain, you also go through on the way back down.

We could add a caching middleware in front of `GET` requests for a specific widget. This middleware will check if a cached representation exists for the resource the client requested. If it exists, it will send it out to the client without ever bothering the `WidgetHandler`. If there's no representation cached, it will call `$next` to propagate the request up the chain. On the return trip (when the call to `$next` finishes), the caching middleware will inspect the response and store the body to the cache for next time.

Here's a pseudocode example:

```
namespace Webservice;

class Cache implements \WellRESTed\MiddlewareInterface
{
    public function dispatch(ServerRequestInterface $request, ResponseInterface
↪$response, $next)
    {
        // Inspect the request path to see if there is a representation on
        // hand for this resource.
        $representation = $this->getCachedRepresentation($request);
        if ($representation !== null) {
            // There is already a cached representation. Send it out
            // without propagating.
            return $response
                ->withStatus(200)
```

```
                ->withBody($representation);
        }

        // No representation exists. Propagate to the next middleware.
        $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)
    {
        // Ensure the response contains a success code, a valid body,
        // headers that allow caching, etc. and store the representation.
        // ...
    }
}
```

We can add this caching middleware in the chain between the authorization middleware and the Widget.

```
$router->register("GET,PUT,DELETE",  "/widgets/{id}", [
        'Webservice\Authorization',
        'Webservice\Cache',
        'Webservice\Widgets\WidgetHandler'
    ]);
```

Or, if you wanted to use the authorization and caching middleware for the entire service, you can add them to the `Server` in front of the `Router`.

```
$server = new \WellRESTed\Server();
$server
    ->add('Webservice\Authorization')
    ->add('Webservice\Cache')
    ->add($server->createRouter()
        ->register("GET,PUT,DELETE", "/widgets/{id}",
↪'Webservice\Widgets\WidgetHandler')
    )
    ->respond();
```

Router

A router is a type of middleware that organizes the components of a site by associating URI paths with other 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 middleware. The dispatched middleware is then responsible for reacting to the request and providing a response.

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 middleware 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);
```

Paths

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

Static Routes

The simplest type of route is called a "static route". It maps middleware 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`.

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);
```

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 an associative array. To access a variable, call the request object's `getAttribute` method method and pass the variable's name.

For a request to `/cats/molly`:

```
$catHandler = function ($request, $response, $next) {
    $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.

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`:

```
$catHandler = function ($request, $response, $next) {
    $vars = $request->getAttributes();
    /*
    Array
    (
        [0] => cats/molly-12
        [name] => molly
        [1] => molly
        [number] => 12
        [2] => 12
        ... Plus any other attributes that were set ...
    )
    */
    ...
}
```

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:

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 added. Dispatch the first route that matches.

5. If no pattern routes match, return a response with a `404 Not Found` status.

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 middleware 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.

Methods

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

Registering by Method

Specify a specific middleware 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);
```

Registering by Method List

Specify the same middleware 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);
```

Registering by Wildcard

Specify middleware 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.

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.

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
```

Error Responses

When a router is unable to dispatch a route because either the path or method does not match a defined route, it will provide an appropriate error response code—either `404 Not Found` or `405 Method Not Allowed`.

The router always checks the path first. If route for that path matches, the router responds `404 Not Found`.

If the 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`.

Given this router:

```
$router
    ->register("GET", "/cats/", $catReader)
    ->register("POST", "/cats/", $catWriter)
    ->register("GET", "/dogs/", $catItemReader)
```

The following requests wil provide these responses:

Method	Path	Response
GET	/hamsters/	404 Not Found
PUT	/cats/	405 Method Not Allowed

Note: When the router fails to dispatch a route, or when it responds to an `OPTIONS` request, is will stop propagation, and any middleware that comes after the router will not be dispatched.

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 sub-routers.

This works because a `Router` is type of middleware, and can be used wherever middleware can be used.

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();
```

URI Templates

WellRESTed allows you to register middleware 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.

Reading Variables

Basic Usage

Register middleware 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, the `$widgetHandler` middleware inspects the request attribute named `"id"`, since `id` is what appears inside curly braces in the URI template.

```
$widgetHandler = function ($request, $response, $next) {
    // Read the variable extracted form the path.
    $id = $request->getAttribute("id");
};
```

When the request path is `/widgets/12`, the value returned by `$request->getAttribute("id")` is `"12"`. For `/widgets/mega-widget`, the value is `"mega-widget"`.

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

Multiple Variables

The example above included one variable, but URI Templates may include multiple variables. 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:

```
$avatarHandlers = function ($request, $response, $next) {
    // Read the variables extracted form the path.
    $username = $request->getAttribute("username");
    // "zoidberg"
    $width = $request->getAttribute("width");
    // "100"
    $height = $request->getAttribute("height");
    // "150"
};
```

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
        )
    */
};
```

Matching Characters

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 3.1: 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 not an unreserved character and is not percent encoded.

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`

```
$pathHandler = function ($request, $response, $next) {
    // Read the variable extracted form the path.
    $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"]`.

URI Templates (Advanced)

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

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 3.2: 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}`).

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 3.3: 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.

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 3.4: 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"`

Extending and Customizing

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

Custom Middleware

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

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

```
interface OtherMiddlewareInterface
{
    /**
     * @param \Psr\Http\Message\ServerRequestInterface $request
     * @param \Psr\Http\Message\ResponseInterface $response
     * @return \Psr\Http\Message\ResponseInterface
     */
    public function run(
        \Psr\Http\Message\ServerRequestInterface $request,
        \Psr\Http\Message\ResponseInterface $response
    );
}
```

Wrapping

One solution is to wrap an instance of this middleware inside of a `WellRESTed\MiddlewareInterface` instance.

```
/**
 * Wraps an instance of OtherMiddlewareInterface
 */
class OtherWrapper implements \WellRESTed\MiddlewareInterface
{
    private $middleware;

    public function __construct(OtherMiddlewareInterface $middleware)
    {
        $this->middleware = $middleware;
    }

    public function __invoke(
        \Psr\Http\Message\ServerRequestInterface $request,
        \Psr\Http\Message\ResponseInterface $response,
        $next
    ) {
        // Run the wrapped middleware.
        $response = $this->middleware->run($request, $response);
        // Pass the middleware's response to $next and return the result.
        return $next($request, $myResponse);
    }
}
```

Note: `OtherMiddlewareInterface` doesn't provide any information about how to propagate the request and response through a chain of middleware, so I chose to call `$next` every time. If there's a sensible way to tell that you should stop propagating, your wrapper class could return a response without calling `$next` under those circumstances. It's up to you and the middleware you're wrapping.

To use this wrapped middleware, you can do something like this:

```
// The class we need to wrap; implements OtherMiddlewareInterface
$other = new OtherMiddleware();

// The wrapper class; implements WellRESTed\MiddlewareInterface
$otherWrapper = new OtherWrapper($other)

$server = new WellRESTed\Server();
$server->add($otherWrapper);
```

Custom Dispatcher

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

The dispatcher is an instance that unpacks your middleware and sends the request and response through it. A default dispatcher is created for you when you instantiate your `WellRESTed\Server` (without passing the second argument). The server instantiates a `WellRESTed\Dispatching\Dispatcher` which is capable of running middleware provided as a callable, a string containing the fully qualified class name of a middleware, or an array of middleware. (See Using Middleware for a description of what a default dispatcher can dispatch.)

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 `OtherMiddlewareInterface`:

```
namespace MyApi;

/**
 * Dispatcher with support for OtherMiddlewareInterface
 */
class CustomDispatcher extends \WellRESTed\Dispatching\Dispatcher
{
    public function dispatch(
        $middleware,
        \Psr\Http\Message\ServerRequestInterface $request,
        \Psr\Http\Message\ResponseInterface $response,
        $next
    ) {
        try {
            // Use the dispatch method in the parent class first.
            $response = parent::dispatch($middleware, $request, $response, $next);
        } catch (\WellRESTed\Dispatching\DispatchException $e) {
            // If there's a problem, check if the middleware implements
            // OtherMiddlewareInterface. Dispatch it if it does.
            if ($middleware instanceof OtherMiddlewareInterface) {
                $response = $middleware->run($request, $response);
                $response = $next($request, $response);
            } else {
                // Otherwise, re-throw the exception.
                throw $e;
            }
        }
        return $response;
    }
}
```

To use this dispatcher, pass it to the constructor of `WellRESTed\Server` as the second argument. (The first argument is a hash array to use as request attributes.)

```
// Create an instance of your custom dispatcher.
$dispatcher = new MyApi\CustomDispatcher;

// Pass this dispatcher to the server.
$server = new WellRESTed\Server(null, $dispatcher);

// Now, you can add any middleware implementing OtherMiddlewareInterface
$other = new OtherMiddleware();
$server->add($other);

// Registering OtherMiddlewareInterface middleware by FQCN will work, too.
```

Message Customization

In the example above, we passed a custom dispatcher to the server. You can also customize your server in other ways. For example, if you have a different implementation of PSR-7 messages that you prefer, you can pass them into the `Server::respond` method:

```
// Represents the request submitted by the client.
$request = new ThirdParty\Request();
// A "blank" response.
$response = new ThirdParty\Response();

$server = new WellRESTed\Server();
// ...add middleware...

// Pass your request and response to Server::respond
$server->response($request, $response);
```

Even if you don't want to use a different implementation, you may still find a reason to provide you're own messages. For example, the default response status code for a `WellRESTed\Message\Response` is 500. If you wanted to make the default 200 instead, you could do something like this:

```
// The first argument is the status code.
$response = new \WellRESTed\Message\Response(200);

$server = new \WellRESTed\Server();
// ...add middleware...

// Pass the response to respond()
$server->respond(null, $response);
```

Server Customization

As an alternative to passing you preferred request and response instances into `Server::respond`, you can extend `Server` to obtain default values from a different source.

Classes such as `Server` that create dependencies as defaults keep the instantiation isolated in easy-to-override methods. For example, `Server` has a protected method `getResponse` that instantiates and returns a new response. You can easily replace this method with your own that returns the default response of your choice.

For example, imagine you have a dependency container that provides the starting messages for you. You can subclass `Server` to obtain and use these messages as defaults like this:

```
class CustomerServer extends WellRESTed\Server
{
    /** @var A dependency container */
    private $container;

    public function __construct(
        $container,
        array $attributes = null,
        DispatcherInterface $dispatcher = null,
        $pathVariablesAttributeName = null
    ) {
        // Call the parent constructor with the expected parameters.
        parent::__construct($attributes, $dispatcher, $pathVariablesAttributeName);
        // Store the container.
        $this->container = $container;
```

```
    }

    /**
     * Redefine this method, which is called in Server::respond when
     * the caller does not provide a request.
     */
    protected function getRequest()
    {
        // Return a request obtained from the container.
        return $this->container["request"];
    }

    /**
     * Redefine this method, which is called in Server::respond when
     * the caller does not provide a response.
     */
    protected function getResponse()
    {
        // Return a response obtained from the container.
        return $this->container["response"];
    }
}
```

In addition to the messages, you can do similar customization for other `Server` dependencies such as the dispatcher (see above), the transmitter (which writes the response out to the client), and the routers that are created with `Server::createRouter`. These dependencies are instantiated in isolated methods as with the request and response to make this sort of customization easy, and other classes such as `Router` use this pattern as well. See the source code, and don't hesitated to subclass.

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 a handful of ways of making dependencies available to middleware.

Request Attribute

`Psr\Http\Message\ServerRequestInterface` provides "attributes" that allow you attach arbitrary data to a request. You can use this to make your dependency container available to any dispatched middleware.

When you instantiate a `WellRESTed\Server`, you can provide an array of attributes that the server will add to the request.

```
$container = new MySuperCoolDependencyContainer();

$server = new WellRESTed\Server(["container" => $container]);
// ... Add middleware, routes, etc. ...
```

When the server dispatches middleware, the middleware will be able to read the container as the "container" attribute.

```
function ($request, $response, $next) {
    $container = $request->getAttribute("container");
```

```
    // It's a super cool dependency container!
}
```

Note: This approach is technically more of a service locator pattern. It's easy to implement, and it allows you the most flexibility in how you assign middleware.

It has some drawbacks as well, though. For example, your middleware is now dependent on your container, and describing which items needs to be **in** the container provides its own challenge.

If your interested in a truer dependency injection approach, read on to the next section where we look at registering middleware factories.

Middleware Factories

Another approach is to use a factory function that returns middleware, usually in the form of a `MiddlewareInterface` instance. This approach provides the opportunity to pass dependencies to your middleware's constructor, while still delaying instantiation until the middleware is used.

Imagine a middleware `FooHandler` that depends on a `BarInterface`, and `BazInterface`.

```
Class FooHandler implements WellRESTed\MiddlewareInterface
{
    private $bar;
    private $baz;

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

    public function __invoke(ServerRequestInterface $request, ResponseInterface
↪$response, $next);
    {
        // Do something with the bar and baz and update the response.
        // ...
        return $response;
    }
}
```

When you add the middleware to the server or register it with a router, you can use a callable that passes appropriate instances into the constructor.

```
// Assume $bar and $baz exist in this scope.
$fooHandlerFactory = function () use ($bar, $bar) {
    return new FooHandler($bar, $baz);
}

$server = new Server();
$server->add(
    $server->createRoute()
        ->register("GET", "/foo/{id}", $fooHandlerFactory)
    );
$server->respond();
```

You can combine this approach with a dependency container. Here's an example using Pimple).

```
$c = new Pimple\Container();
$c["bar"] = /* Return a BarInterface */
$c["baz"] = /* Return a BazInterface */
$c["fooHandler"] = $c->protect(function () use ($c) {
    return new FooHandler($c["bar"], $c["baz"]);
});

$server = new Server();
$server->add(
    $server->createRoute()
        ->register("GET", "/foo/{id}", $c["fooHandler"])
    );
$server->respond();
```

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

Or, see WellRESTed on GitHub.

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*.

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;
    }

}
```

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]
```