

Simple, fast and yet powerful PHP router that is easy to get integrated and in any project. Heavily inspired by the Laravel router.

[#router](#) [#php](#) [#cms](#) [#library](#) [#module](#) [#php-router](#) [#php-router-standalone](#) [#csrf-protection](#) [#middleware](#) [#rewrite-urls](#) [#easy-to-use](#) [#lightweight](#) [#input-handler](#) [#request-handler](#) [#url-handler](#) [#routing-engine](#) [#routing](#)

681 commits

9 branches

190 releases

3 contributors

Branch: master ▾

New pull request

Create new file

Upload files

Find file

Clone or download ▾

skipperbent committed 29 days ago Merge pull request #353 from skipperbent/v3 ... Latest commit 14360b6 29 days ago

src/Pecee	Fixed Input::all when php://input is not available	29 days ago
test	Fixed parameter default-value being overwritten (issue: #347).	a month ago
.gitignore	Development	7 months ago
README.md	Development	3 months ago
composer.json	Updated composer.json tags.	7 months ago
helpers.php	Optimisations	3 months ago

README.md

Simple PHP router

Simple, fast and yet powerful PHP router that is easy to get integrated and in any project. Heavily inspired by the way Laravel handles routing, with both simplicity and expandability in mind.

Note: this documentation is currently work-in-progress. Feel free to contribute.

Notes

The goal of this project is to create a router that is more or less 100% compatible with the Laravel documentation, while remaining as simple as possible, and as easy to integrate and change without compromising either speed or complexity. Being lightweight is the #1 priority.

Feedback and development

If you are missing a feature, experience problems or have ideas or feedback that you want us to hear, please feel free to create an issue.

Issues guidelines

- Please be as detailed as possible in the description when creating a new issue. This will help others to more easily understand- and solve your issue. For example: if you are experiencing issues, you should provide the necessary steps to reproduce the error within your description.
- We love to hear out any ideas or feedback to the library.

[Create a new issue here](#)

Contribution development guidelines

- Please try to follow the PSR-2 codestyle guidelines.
- Please create your pull requests to the development base that matches the version number you want to change. For example when pushing changes to version 3, the pull request should use the `v3-development` base/branch.
- Create detailed descriptions for your commits, as these will be used in the changelog for new releases.
- When changing existing functionality, please ensure that the unit-tests working.

- When adding new stuff, please remember to add new unit-tests for the functionality.

Table of Contents

- [Getting started](#)
 - [Requirements](#)
 - [Notes](#)
 - [Features](#)
 - [Installation](#)
 - [Setting up Apache](#)
 - [Setting up Nginx](#)
 - [Setting up IIS](#)
 - [Configuration](#)
 - [Helper functions](#)
- [Routes](#)
 - [Basic routing](#)
 - [Available methods](#)
 - [Multiple HTTP-verbs](#)
 - [Route parameters](#)
 - [Required parameters](#)
 - [Optional parameters](#)
 - [Regular expression constraints](#)
 - [Regular expression route-match](#)
 - [Custom regex for matching parameters](#)
 - [Named routes](#)
 - [Generating URLs To Named Routes](#)
 - [Router groups](#)
 - [Middleware](#)
 - [Namespaces](#)
 - [Subdomain-routing](#)
 - [Route prefixes](#)
 - [Partial groups](#)
 - [Form Method Spoofing](#)
 - [Accessing The Current Route](#)
 - [Other examples](#)
- [CSRF-protection](#)
 - [Adding CSRF-verifier](#)
 - [Getting CSRF-token](#)
 - [Custom CSRF-verifier](#)
 - [Custom Token-provider](#)
- [Middlewares](#)
 - [Example](#)
- [ExceptionHandlerers](#)
 - [Handling 404, 403 and other errors](#)
 - [Using custom exception handlers](#)
- [Urls](#)
 - [Get by name \(single route\)](#)
 - [Get by name \(controller route\)](#)

- [Get by class](#)
- [Get by custom names for methods on a controller/resource route](#)
- [Getting REST/resource controller urls](#)
- [Get the current url](#)
- [Input & parameters](#)
 - [Using the Input class to manage parameters](#)
 - [Get single parameter value](#)
 - [Get parameter object](#)
 - [Managing files](#)
 - [Get all parameters](#)
- [Advanced](#)
 - [Url rewriting](#)
 - [Rewrite using callback](#)
 - [Rewrite using url](#)
 - [Bootmanager: loading routes dynamically](#)
 - [Adding routes manually](#)
 - [Parameters](#)
 - [Extending](#)
- [Credits](#)
 - [Sites](#)
 - [License](#)

Getting started

Add the latest version of Simple PHP Router running this command.

```
composer require pecee/simple-router
```

Requirements

- PHP 5.5 or greater

Notes

We've included a simple demo project for the router which can be found in the `demo-project` folder. This project should give you a basic understanding of how to setup and use simple-php-router project.

Please note that the demo-project only covers how to integrate the `simple-php-router` in a project without an existing framework. If you are using a framework in your project, the implementation might vary.

You can find the demo-project here: <https://github.com/skipperbent/simple-router-demo>

What we won't cover:

- How to setup a solution that fits your need. This is a basic demo to help you get started.
- Understanding of MVC; including Controllers, Middlewares or ExceptionHandlers.
- How to integrate into third party frameworks.

What we cover:

- How to get up and running fast - from scratch.
- How to get ExceptionHandlers, Middlewares and Controllers working.
- How to setup your webservers.

Features

- Basic routing (`GET` , `POST` , `PUT` , `PATCH` , `UPDATE` , `DELETE`) with support for custom multiple verbs.
- Regular Expression Constraints for parameters.
- Named routes.
- Generating url to routes.
- Route groups.
- Middleware (classes that intercepts before the route is rendered).
- Namespaces.
- Route prefixes.
- CSRF protection.
- Optional parameters
- Sub-domain routing
- Custom boot managers to rewrite urls to "nicer" ones.
- Input manager; easily manage `GET` , `POST` and `FILE` values.

Installation

1. Navigate to your project folder in terminal and run the following command:

```
composer require pecee/simple-router
```

Setting up Nginx

If you are using Nginx please make sure that url-rewriting is enabled.

You can easily enable url-rewriting by adding the following configuration for the Nginx configuration-file for the demo-project.

```
location / {  
    try_files $uri $uri/ /index.php?$query_string;  
}
```

Setting up Apache

Nothing special is required for Apache to work. We've include the `.htaccess` file in the `public` folder. If rewriting is not working for you, please check that the `mod_rewrite` module (htaccess support) is enabled in the Apache configuration.

.htaccess example

Below is an example of an working `.htaccess` file used by simple-php-router.

Simply create a new `.htaccess` file in your projects `public` directory and paste the contents below in your newly created file. This will redirect all requests to your `index.php` file (see Configuration section below).

```
RewriteEngine on  
RewriteCond %{SCRIPT_FILENAME} !-f  
RewriteCond %{SCRIPT_FILENAME} !-d  
RewriteCond %{SCRIPT_FILENAME} !-l  
RewriteRule ^(.*)$ index.php/$1
```

Setting up IIS

On IIS you have to add some lines your `web.config` file in the `public` folder or create a new one. If rewriting is not working for you, please check that your IIS version have included the `url rewrite` module or download and install them from Microsoft web site.

web.config example

Below is an example of an working `web.config` file used by simple-php-router.

Simply create a new `web.config` file in your projects `public` directory and paste the contents below in your newly created file. This will redirect all requests to your `index.php` file (see Configuration section below). If the `web.config` file already exists, add the `<rewrite>` section inside the `<system.webServer>` branch.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <!-- Remove slash '/' from the en of the url -->
        <rule name="RewriteRequestsToPublic">
          <match url="^(.*)$" />
          <conditions logicalGrouping="MatchAll" trackAllCaptures="false">
          </conditions>
          <action type="Rewrite" url="{R:0}" />
        </rule>

        <!-- When requested file or folder don't exists, will request again through index.php -->
        <rule name="Imported Rule 1" stopProcessing="true">
          <match url="^(.*)$" ignoreCase="true" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
          </conditions>
          <action type="Rewrite" url="/index.php/{R:1}" appendQueryString="true" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
</configuration>
```

Troubleshooting

If you do not have a `favicon.ico` file in your project, you can get a `NotFoundException` (404 - not found). To add `favicon.ico` to the IIS ignore-list, add the following line to the `<conditions>` group:

```
<add input="{REQUEST_FILENAME}" negate="true" pattern="favicon.ico" ignoreCase="true" />
```

You can also make one exception for files with some extensions:

```
<add input="{REQUEST_FILENAME}" pattern="\.ico|\.png|\.css|\.jpg" negate="true" ignoreCase="true" />
```

If you are using `$_SERVER['ORIG_PATH_INFO']`, you will get `\index.php\` as part of the returned value. For example:

```
/index.php/test/mypage.php
```

Configuration

Create a new file, name it `routes.php` and place it in your library folder. This will be the file where you define all the routes for your project.

WARNING: NEVER PLACE YOUR ROUTES.PHP IN YOUR PUBLIC FOLDER!

In your `index.php` require your newly-created `routes.php` and call the `SimpleRouter::start()` method. This will trigger and do the actual routing of the requests.

It's not required, but you can set `SimpleRouter::setDefaultNamespace('Demo\Controllers');` to prefix all routes with the namespace to your controllers. This will simplify things a bit, as you won't have to specify the namespace for your controllers on each route.

This is an example of a basic `index.php` file:

```
<?php
use Pecee\SimpleRouter\SimpleRouter;

/* Load external routes file */
require_once 'routes.php';
```

```

/**
 * The default namespace for route-callbacks, so we don't have to specify it each time.
 * Can be overwritten by using the namespace config option on your routes.
 */

SimpleRouter::setDefaultNamespace('Demo\Controllers');

// Start the routing
SimpleRouter::start();

```

Helper functions

We recommend that you add these helper functions to your project. These will allow you to access functionality of the router more easily.

To implement the functions below, simply copy the code to a new file and require the file before initializing the router or copy the `helpers.php` we've included in this library.

```

use Pecee\SimpleRouter\SimpleRouter as Router;

/**
 * Get url for a route by using either name/alias, class or method name.
 *
 * The name parameter supports the following values:
 * - Route name
 * - Controller/resource name (with or without method)
 * - Controller class name
 *
 * When searching for controller/resource by name, you can use this syntax "route.name@method".
 * You can also use the same syntax when searching for a specific controller-class "MyController@home".
 * If no arguments is specified, it will return the url for the current loaded route.
 *
 * @param string|null $name
 * @param string|array|null $parameters
 * @param array|null $getParams
 * @return string
 */
function url($name = null, $parameters = null, $getParams = null)
{
    return Router::getUrl($name, $parameters, $getParams);
}

/**
 * @return \Pecee\Http\Response
 */
function response()
{
    return Router::response();
}

/**
 * @return \Pecee\Http\Request
 */
function request()
{
    return Router::request();
}

/**
 * Get input class
 * @param string|null $index Parameter index name
 * @param string|null $defaultValue Default return value
 * @param string|array|null $methods Default method
 * @return \Pecee\Http\Input\Input|string
 */
function input($index = null, $defaultValue = null, $methods = null)
{
    if ($index !== null) {
        return request()->getInput()->get($index, $defaultValue, $methods);
    }

    return request()->getInput();
}

```

```
function redirect($url, $code = null)
{
    if ($code !== null) {
        response()->httpCode($code);
    }

    response()->redirect($url);
}

/**
 * Get current csrf-token
 * @return string|null
 */
function csrf_token()
{
    $baseVerifier = Router::router()->getCsrVerifier();
    if ($baseVerifier !== null) {
        return $baseVerifier->getToken();
    }

    return null;
}
```

Routes

Remember the `routes.php` file you required in your `index.php`? This file be where you place all your custom rules for routing.

Basic routing

Below is a very basic example of setting up a route. First parameter is the url which the route should match - next parameter is a `Closure` or callback function that will be triggered once the route matches.

```
SimpleRouter::get('/', function() {
    return 'Hello world';
});
```

Available methods

Here you can see a list over all available routes:

```
SimpleRouter::get($url, $callback, $settings);
SimpleRouter::post($url, $callback, $settings);
SimpleRouter::put($url, $callback, $settings);
SimpleRouter::patch($url, $callback, $settings);
SimpleRouter::delete($url, $callback, $settings);
SimpleRouter::options($url, $callback, $settings);
```

Multiple HTTP-verbs

Sometimes you might need to create a route that accepts multiple HTTP-verbs. If you need to match all HTTP-verbs you can use the `any` method.

```
SimpleRouter::match(['get', 'post'], '/', function() {
    // ...
});

SimpleRouter::any('foo', function() {
    // ...
});
```

We've created a simple method which matches `GET` and `POST` which is most commonly used:

```
SimpleRouter::form('foo', function() {
    // ...
```

```
});
```

Route parameters

Required parameters

You'll properly wondering by know how you parse parameters from your urls. For example, you might want to capture the users id from an url. You can do so by defining route-parameters.

```
SimpleRouter::get('/user/{id}', function ($userId) {  
    return 'User with id: ' . $userId;  
});
```

You may define as many route parameters as required by your route:

```
SimpleRouter::get('/posts/{post}/comments/{comment}', function ($postId, $commentId) {  
    // ...  
});
```

Note: Route parameters are always encased within {} braces and should consist of alphabetic characters. Route parameters may not contain a - character. Use an underscore (_) instead.

Optional parameters

Occasionally you may need to specify a route parameter, but make the presence of that route parameter optional. You may do so by placing a ? mark after the parameter name. Make sure to give the route's corresponding variable a default value:

```
SimpleRouter::get('/user/{name?}', function ($name = null) {  
    return $name;  
});  
  
SimpleRouter::get('/user/{name?}', function ($name = 'Simon') {  
    return $name;  
});
```

Regular expression constraints

You may constrain the format of your route parameters using the where method on a route instance. The where method accepts the name of the parameter and a regular expression defining how the parameter should be constrained:

```
SimpleRouter::get('/user/{name}', function ($name) {  
    //  
})->where('name', '[A-Za-z]+');  
  
SimpleRouter::get('/user/{id}', function ($id) {  
    //  
})->where('id', '[0-9]+');  
  
SimpleRouter::get('/user/{id}/{name}', function ($id, $name) {  
    //  
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Regular expression route-match

You can define a regular-expression match for the entire route if you wish.

This is useful if you for example are creating a model-box which loads urls from ajax.

The example below is using the following regular expression: `/ajax/([\w]+)/?([0-9]+)?/?` which basically just matches `/ajax/` and expects the next parameter to be a string - and the next to be a number (but optional).

Matches: `/ajax/abc/` , `/ajax/abc/123/`

Won't match: `/ajax/`

Match groups specified in the regex will be passed on as parameters:

```
SimpleRouter::all('/ajax/abc/123', function($param1, $param2) {
    // param1 = abc
    // param2 = 123
})->setMatch('/\Vajax\V([\w]+)\V?([0-9]+)?\V?/is');
```

Custom regex for matching parameters

By default simple-php-router uses the `\w` regular expression when matching parameters. This decision was made with speed and reliability in mind, as this match will match both letters, number and most of the used symbols on the internet.

However, sometimes it can be necessary to add a custom regular expression to match more advanced characters like `-` etc.

Instead of adding a custom regular expression to all your parameters, you can simply add a global regular expression which will be used on all the parameters on the route.

Note: If you the regular expression to be available across, we recommend using the global parameter on a group as demonstrated in the examples below.

Example

This example will ensure that all parameters use the `[\w\-\-]+` regular expression when parsing.

```
SimpleRouter::get('/path/{parameter}', 'VideoController@home', ['defaultParameterRegex' => '[\w\-\-]+']);
```

You can also apply this setting to a group if you need multiple routes to use your custom regular expression when parsing parameters.

```
SimpleRouter::group(['defaultParameterRegex' => '[\w\-\-]+'], function() {

    SimpleRouter::get('/path/{parameter}', 'VideoController@home');

});
```

Named routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the name method onto the route definition:

```
SimpleRouter::get('/user/profile', function () {
    // Your code here
})->name('profile');
```

You can also specify names for Controller-actions:

```
SimpleRouter::get('/user/profile', 'UserController@profile')->name('profile');
```

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via the global `url` helper-function (see helpers section):

```
// Generating URLs...
$url = url('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the `url` function. The given parameters will automatically be inserted into the URL in their correct positions:

```
SimpleRouter::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');
```

```
$url = url('profile', ['id' => 1]);
```

For more information on urls, please see the [Urls](#) section.

Router groups

Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route. Shared attributes are specified in an array format as the first parameter to the `SimpleRouter::group` method.

Middleware

To assign middleware to all routes within a group, you may use the middleware key in the group attribute array. Middleware are executed in the order they are listed in the array:

```
SimpleRouter::group(['middleware' => \Demo\Middleware\Auth::class], function () {  
    SimpleRouter::get('/', function () {  
        // Uses Auth Middleware  
    });  
  
    SimpleRouter::get('/user/profile', function () {  
        // Uses Auth Middleware  
    });  
});
```

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers using the `namespace` parameter in the group array:

Note

Group namespaces will only be added to routes with relative callbacks. For example if your route has an absolute callback like `\Demo\Controller\DefaultController@home`, the namespace from the route will not be prepended. To fix this you can make the callback relative by removing the `\` in the beginning of the callback.

```
SimpleRouter::group(['namespace' => 'Admin'], function () {  
    // Controllers Within The "App\Http\Controllers\Admin" Namespace  
});
```

Sub domain-routing

Route groups may also be used to handle sub-domain routing. Sub-domains may be assigned route parameters just like route URIs, allowing you to capture a portion of the sub-domain for usage in your route or controller. The sub-domain may be specified using the `domain` key on the group attribute array:

```
SimpleRouter::group(['domain' => '{account}.myapp.com'], function () {  
    SimpleRouter::get('/user/{id}', function ($account, $id) {  
        //  
    });  
});
```

Route prefixes

The `prefix` group attribute may be used to prefix each route in the group with a given URI. For example, you may want to prefix all route URIs within the group with `admin`:

```
SimpleRouter::group(['prefix' => '/admin'], function () {  
    SimpleRouter::get('/users', function () {  
        // Matches The "/admin/users" URL  
    });  
});
```

Partial groups

Partial router groups has the same benefits as a normal group, but supports parameters and are only rendered once the url has matched.

This can be extremely useful in situations, where you only want special routes to be added, when a certain criteria or logic has been met.

NOTE: Use partial groups with caution as routes added within are only rendered and available once the url of the partial-group has matched. This can cause `url()` not to find urls for the routes added within.

Example:

```
SimpleRouter::partialGroup('/admin/{applicationId}', function ($applicationId) {

    SimpleRouter::get('/', function($applicationId) {

        // Matches The "/admin/applicationId" URL

    });

});
```

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

```
<input type="hidden" name="_method" value="PUT" />
```

Accessing The Current Route

You can access information about the current route loaded by using the following method:

```
SimpleRouter::request()->getLoadedRoute();
request()->getLoadedRoute();
```

Other examples

You can find many more examples in the `routes.php` example-file below:

```
<?php
use Pecee\SimpleRouter\SimpleRouter;

/* Adding custom csrfVerifier here */
SimpleRouter::csrfVerifier(new \Demo\Middlewares\CsrfVerifier());

SimpleRouter::group(['middleware' => \Demo\Middlewares\Site::class, 'exceptionHandler' => \Demo\Handlers\CustomExcept

SimpleRouter::get('/answers/{id}', 'ControllerAnswers@show', ['where' => ['id' => '[0-9]+']]);

/**
 * Restful resource (see IRestController interface for available methods)
 */

SimpleRouter::resource('/rest', ControllerResource::class);

/**
 * Load the entire controller (where url matches method names - getIndex(), postIndex(), putIndex()).
 * The url paths will determine which method to render.
 *
 * For example:
```

```

*
* GET /animals      => getIndex()
* GET /animals/view => getView()
* POST /animals/save => postSave()
*
* etc.
*/

SimpleRouter::controller('/animals', ControllerAnimals::class);

});

SimpleRouter::get('/page/404', 'ControllerPage@notFound', ['as' => 'page.notfound']);

```

CSRF Protection

Any forms posting to `POST`, `PUT` or `DELETE` routes should include the CSRF-token. We strongly recommend that you enable CSRF-verification on your site to maximize security.

You can use the `BaseCsrfVerifier` to enable CSRF-validation on all request. If you need to disable verification for specific urls, please refer to the "Custom CSRF-verifier" section below.

By default simple-php-router will use the `CookieTokenProvider` class. This provider will store the security-token in a cookie on the clients machine. If you want to store the token elsewhere, please refer to the "Creating custom Token Provider" section below.

Adding CSRF-verifier

When you've created your CSRF-verifier you need to tell simple-php-router that it should use it. You can do this by adding the following line in your `routes.php` file:

```
Router::csrfVerifier(new \Demo\Middlewares\CsrfVerifier());
```

Getting CSRF-token

When posting to any of the urls that has CSRF-verification enabled, you need post your CSRF-token or else the request will get rejected.

You can get the CSRF-token by calling the helper method:

```
csrf_token();
```

You can also get the token directly:

```
return Router::router()->getCsrfVerifier()->getTokenProvider()->getToken();
```

The default name/key for the input-field is `csrf_token` and is defined in the `POST_KEY` constant in the `BaseCsrfVerifier` class. You can change the key by overwriting the constant in your own CSRF-verifier class.

Example:

The example below will post to the current url with a hidden field "`csrf_token`".

```

<form method="post" action="<?= url(); ?>">
  <input type="hidden" name="csrf_token" value="<?= csrf_token(); ?>">
  <!-- other input elements here -->
</form>

```

Custom CSRF-verifier

Create a new class and extend the `BaseCsrfVerifier` middleware class provided by default with the simple-php-router library.

Add the property `except` with an array of the urls to the routes you want to exclude/whitelist from the CSRF validation. Using `*` at the end for the url will match the entire url.

Here's a basic example on a CSRF-verifier class:

```
namespace Demo\Middlewares;

use Pecee\Http\Middleware\BaseCsrfVerifier;

class CsrfVerifier extends BaseCsrfVerifier
{
    /**
     * CSRF validation will be ignored on the following urls.
     */
    protected $except = ['/api/*'];
}
```

Custom Token Provider

By default the `BaseCsrfVerifier` will use the `CookieTokenProvider` to store the token in a cookie on the clients machine.

If you need to store the token elsewhere, you can do that by creating your own class and implementing the `ITokenProvider` class.

```
class SessionTokenProvider implements ITokenProvider
{
    /**
     * Refresh existing token
     */
    public function refresh()
    {
        // Implement your own functionality here...
    }

    /**
     * Validate valid CSRF token
     *
     * @param string $token
     * @return bool
     */
    public function validate($token)
    {
        // Implement your own functionality here...
    }
}
```

Next you need to set your custom `ITokenProvider` implementation on your `BaseCsrfVerifier` class in your routes file:

```
$verifier = new \dscuz\Middleware\CsrfVerifier();
$verifier->setTokenProvider(new SessionTokenProvider());

Router::csrfVerifier($verifier);
```

Middlewares

Middlewares are classes that loads before the route is rendered. A middleware can be used to verify that a user is logged in - or to set parameters specific for the current request/route. Middlewares must implement the `IMiddleware` interface.

Example

```

namespace Demo\Middlewares;

use Pecee\Http\Middleware\IMiddleware;
use Pecee\Http\Request;

class CustomMiddleware implements Middleware {

    public function handle(Request $request) {

        // Authenticate user, will be available using request()->user
        $request->user = User::authenticate();

        // If authentication failed, redirect request to user-login page.
        if($request->user === null) {
            $request->setRewriteUrl(url('user.login'));
            return $request;
        }
    }
}

```

ExceptionHandler

ExceptionHandler are classes that handles all exceptions. ExceptionsHandlers must implement the `ExceptionHandler` interface.

Handling 404, 403 and other errors

If you simply want to catch a 404 (page not found) etc. you can use the `Router::error($callback)` static helper method.

This will add a callback method which is fired whenever an error occurs on all routes.

The basic example below simply redirect the page to `/not-found` if an `NotFoundHttpException` (404) occurred. The code should be placed in the file that contains your routes.

```

Router::get('/not-found', 'PageController@notFound');

Router::error(function(Request $request, \Exception $exception) {
    if($exception instanceof NotFoundHttpException && $exception->getStatusCode() == 404) {
        response()->redirect('/not-found');
    }
});

```

Using custom exception handlers

This is a basic example of an ExceptionHandler implementation (please see "[Easily overwrite route about to be loaded](#)" for examples on how to change callback).

```

namespace Demo\Handlers;

use Pecee\Handlers\ExceptionHandler;
use Pecee\Http\Request;
use Pecee\SimpleRouter\Exceptions\NotFoundHttpException;

class CustomExceptionHandler implements ExceptionHandler
{
    public function handleError(Request $request, \Exception $error)
    {
        /* You can use the exception handler to format errors depending on the request and type. */

        if (stripos($request->getUri()->getPath(), '/api') !== false) {

            response()->json([
                'error' => $error->getMessage(),
                'code' => $error->getStatusCode(),
            ]);
        }
    }
}

```

```

    }

    /* The router will throw the NotFoundHttpException on 404 */
    if($error instanceof NotFoundHttpException) {

        // Render custom 404-page

        $request->setRewriteCallback('Demo\Controllers\PageController@notFound');
        return $request;

    }

    throw $error;

}

}
}

```

Urls

By default all controller and resource routes will use a simplified version of their url as name.

Get by name (single route)

```

SimpleRouter::get('/product-view/{id}', 'ProductsController@show', ['as' => 'product']);

url('product', ['id' => 22], ['category' => 'shoes']);
url('product', null, ['category' => 'shoes']);

# output
# /product-view/22/?category=shoes
# /product-view/?category=shoes

```

Get by name (controller route)

```

SimpleRouter::controller('/images', ImagesController::class, ['as' => 'picture']);

url('picture@getView', null, ['category' => 'shoes']);
url('picture', 'getView', ['category' => 'shoes']);
url('picture', 'view');

# output
# /images/view/?category=shows
# /images/view/?category=shows
# /images/view/

```

Get by class

```

SimpleRouter::get('/product-view/{id}', 'ProductsController@show', ['as' => 'product']);
SimpleRouter::controller('/images', 'ImagesController');

url('ProductsController@show', ['id' => 22], ['category' => 'shoes']);
url('ImagesController@getImage', null, ['id' => 22]);

# output
# /product-view/22/?category=shoes
# /images/image/?id=22

```

Using custom names for methods on a controller/resource route

```

SimpleRouter::controller('gadgets', GadgetsController::class, ['names' => ['getIphoneInfo' => 'iphone']]);

url('gadgets.iphone');

```

```
# output
# /gadgets/iphoneinfo/
```

Getting REST/resource controller urls

```
SimpleRouter::resource('/phones', PhonesController::class);

url('phones');
url('phones.index');
url('phones.create');
url('phones.edit');

// etc..

# output
# /phones/
# /phones/create/
# /phones/edit/
```

Get the current url

```
url();
url(null, null, ['q' => 'cars']);

# output
# /CURRENT-URL/
# /CURRENT-URL/?q=cars
```

Input & parameters

Using the Input class to manage parameters

We've added the `Input` class to easy access and manage parameters from your Controller-classes.

Get single parameter value:

If items is grouped in the html, it will return an array of items.

Note: `get` will automatically trim the value and ensure that it's not empty. If it's empty the `$defaultValue` will be returned.

```
$value = input($index, $defaultValue, $methods);
```

Get parameter object

Will return an instance of `InputItem` or `InputFile` depending on the type.

You can use this in your html as it will render the value of the item. However if you want to compare value in your if statements, you have to use the `getValue` or use the `input()` instead.

If items is grouped in the html, it will return an array of items.

Note: `getObject` will only return `$defaultValue` if the item doesn't exist. If you want `$defaultValue` to be returned if the item is empty, please use `input()` instead.

```
$object = input()->getObject($index, $defaultValue = null, $methods = null);
```

Return specific GET parameter (where name is the name of your parameter):

```
# -- match any (default) --

/*
 * This is the recommended way to go for normal usage
 * as it will strip empty values, ensuring that
```



```

* $defaultValue is returned if the value is empty.
*/

$id = input()->get($index, $defaultValue, $method);

# -- shortcut to above --

$id = input($index, $defaultValue, $method);

# -- match specific --

$object = input($index, $defaultValue, 'get');
$object = input($index, $defaultValue, 'post');
$object = input($index, $defaultValue, 'file');

# -- or --

$object = input()->findGet($index, $defaultValue);
$object = input()->findPost($index, $defaultValue);
$object = input()->findFile($index, $defaultValue);

```

Managing files

```

/**
 * In this small example we loop through a collection of files
 * added on the page like this
 * <input type="file" name="images[]" />
 */

/* @var $image \Pecee\Http\Input\InputFile */
foreach(input('images', []) as $image)
{
    if($image->getMime() === 'image/jpeg') {

        $destinationFilename = sprintf('%s.%s', uniqid(), $image->getExtension());

        $image->move('/uploads/' . $destinationFilename);

    }
}

```

Get all parameters

```

// Get all
$values = input()->all();

// Only match certain keys
$values = input()->all([
    'company_name',
    'user_id'
]);

```

All `input` implements the `IInputItem` interface and will always contain these methods:

- `getIndex()` - returns the index/key of the input.
- `getName()` - returns a human friendly name for the input (company_name will be Company Name etc).
- `getValue()` - returns the value of the input.

`InputFile` has the same methods as above along with some other file-specific methods like:

- `getFilename()` - get the filename.
- `getTmpName()` - get file temporary name.
- `getSize()` - get file size.
- `move($destination)` - move file to destination.
- `getContents()` - get file content.
- `getType()` - get mime-type for file.
- `getError()` - get file upload error.
- `hasError()` - returns `bool` if an error occurred while uploading (if `getError` is not 0).

- `toArray()` - returns raw array

Below example requires you to have the helper functions added. Please refer to the helper functions section in the documentation.

```
/* Get parameter site_id or default-value 2 from either post-value or query-string */
$siteId = input('site_id', 2, ['post', 'get']);
```

Advanced

Url rewriting

Sometimes it can be useful to manipulate the route about to be loaded. `simple-php-router` allows you to easily change the route about to be executed. All information about the current route is stored in the `\Pecee\SimpleRouter\Router` instance's `loadedRoute` property.

For easy access you can use the shortcut method `\Pecee\SimpleRouter\SimpleRouter::router()`.

```
use Pecee\SimpleRouter;
$request = SimpleRouter::request();
$request->setRewriteCallback('Example\MyCustomClass@hello');

// -- or you can rewrite by url --

$request->setRewriteUrl('/my-rewrite-url');
```

Note: It's only possible to change the route BEFORE the route has initially been rendered. You can use the `Request` object to manipulate the route which are about to be loaded.

Rewrite using callback

This method is most efficient, as it will render the route immediately.

This method is useful for rendering 404-pages etc.

You can also change the callback by modifying the `$route` parameter. This is perfect if you just want to display a view quickly - or change the callback depending on some criteria's for the request.

The callback below will fire immediately after the `Middleware` or `ExceptionHandler` has been loaded, as they are loaded before the route is rendered. If you wish to change the callback from outside, please have this in mind.

The example below will render `DefaultController@notFound` regardless of the url.

NOTE: Use this method if you want to load another controller. No additional middlewares or rules will be loaded.

Middleware example

```
namespace Demo\Middlewares;

use Pecee\Http\Middleware\IMiddleware;
use Pecee\Http\Request;

class CustomMiddleware implements IMiddleware {

    public function handle(Request $request) {

        $request->setRewriteCallback('Demo\Controllers\DefaultController@notFound');
        return $request;

    }

}
```

ExceptionHandler example

```

namespace Demo\Handlers;

use Pecee\Handlers\ExceptionHandler;
use Pecee\Http\Request;
use Pecee\SimpleRouter\Exceptions\NotFoundException;

class CustomExceptionHandler implements ExceptionHandler
{
    public function handleError(Request $request, \Exception $error)
    {
        /* The router will throw the NotFoundException on 404 */
        if($error instanceof NotFoundException) {

            /*
             * Render your own custom 404-view, rewrite the request to another route,
             * or simply return the $request object to ignore the error and continue on rendering the route.
             *
             * The code below will make the router render our page.notfound route.
             */

            $request->setRewriteCallback('Demo\Controllers\DefaultController@notFound');
            return $request;

        }

        throw $error;
    }
}

```

Rewrite using url

The example below will cause the router to reload the request and reinitialize all the routes. This method is slower, but will ensure that all middlewares and rules for the route is loaded.

This method is useful if you want to redirect a url to another-url which is dependent on a middleware. You can also add a custom rule by calling `$request->setRewriteRoute($route)` if you want to customize request-methods or use another route-type like `RouteController` etc.

We are using the `url()` helper function to get the uri to another route added in the `routes.php` file.

NOTE: Use this method if you want to fully load another route using it's settings (request method, middlewares etc).

Middleware example

The example below will redirect the request to the `home` -route.

```

namespace Demo\Middlewares;

use Pecee\Http\Middleware\IMiddleware;
use Pecee\Http\Request;

class CustomMiddleware implements IMiddleware {

    public function handle(Request $request) {

        $request->setRewriteUrl(url('home'));
        return $request;

    }
}

```

Bootmanager: loading routes dynamically

Sometimes it can be necessary to keep urls stored in the database, file or similar. In this example, we want the url `/my-cat-is-beautiful` to load the route `/article/view/1` which the router knows, because it's defined in the `routes.php` file.

To interfere with the router, we create a class that implements the `IRouterBootManager` interface. This class will be loaded before any other rules in `routes.php` and allow us to "change" the current route, if any of our criteria are fulfilled (like coming from the url `/my-cat-is-beautiful`).

```
use Pecee\Http\Request;
use Pecee\SimpleRouter\IRouterBootManager;

class CustomRouterRules implements IRouterBootManager {

    public function boot(Request $request) {

        $rewriteRules = [
            '/my-cat-is-beautiful' => '/article/view/1',
            '/horses-are-great' => '/article/view/2',
        ];

        foreach($rewriteRules as $url => $rule) {

            // If the current uri matches the url, we use our custom route

            if($request->getUri()->getPath() === $url) {
                $request->setRewriteUrl($rule);
                return $request;
            }
        }
    }
}
```

The above should be pretty self-explanatory and can easily be changed to loop through urls store in the database, file or cache.

What happens is that if the current route matches the route defined in the index of our `$rewriteRules` array, we set the route to the array value instead.

By doing this the route will now load the url `/article/view/1` instead of `/my-cat-is-beautiful`.

The last thing we need to do, is to add our custom boot-manager to the `routes.php` file. You can create as many bootmanagers as you like and easily add them in your `routes.php` file.

```
SimpleRouter::addBootManager(new CustomRouterRules());
```

Adding routes manually

The `SimpleRouter` class referenced in the previous example, is just a simple helper class that knows how to communicate with the `Router` class. If you are up for a challenge, want the full control or simply just want to create your own `Router` helper class, this example is for you.

```
use \Pecee\SimpleRouter\Router;
use \Pecee\SimpleRouter\Route\RouteUrl;

/* Create new Router instance */
$router = new Router();

$route = new RouteUrl('/answer/1', function() {

    die('this callback will match /answer/1');

});

$route->addMiddleware(\Demo\Middlewares\AuthMiddleware::class);
$route->setNamespace('\Demo\Controllers');
$route->setPrefix('v1');

/* Add the route to the router */
$router->addRoute($route);
```

Parameters

This section contains advanced tips & tricks on extending the usage for parameters.

Extending

This is a simple example of an integration into a framework.

The framework has it's own `Router` class which inherits from the `SimpleRouter` class. This allows the framework to add custom functionality like loading a custom `routes.php` file or add debugging information etc.

```
namespace Demo;

use Pecee\SimpleRouter\SimpleRouter;

class Router extends SimpleRouter {

    public static function start() {

        // change this to whatever makes sense in your project
        require_once 'routes.php';

        // change default namespace for all routes
        parent::setDefaultNamespace('\Demo\Controllers');

        // Do initial stuff
        parent::start();

    }

}
```

Credits

Sites

This is some sites that uses the simple-router project in production.

- [holla.dk](#)
- [ninjaimg.com](#)
- [bookandbegin.com](#)
- [dscuz.com](#)

License

The MIT License (MIT)

Copyright (c) 2016 Simon Sessingø / simple-php-router

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:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.