# Marmalade

Marmalade is a PHP micro-framework designed to be easy to set up and use while being very flexible and very fast.

GET MARMALADE

**User Guide:**

## Features:

### API READY

Marmalade is API ready with a flip of a boolean. Marmalade provides a robust framework to build APIs from including strong API request security and versioning.

### MVC

Marmalade is designed using MVC (the model, view, controller pattern). MVC allows for separation of concerns, meaning your application logic is separate from your presentation. This allows you to update your UI without changing any backend code or logic.

### NAMESPACES

Marmalade is fully namespaced and supports user namespacing as well. Namespacing is a nice way of organizing code while also preventing naming collisions with other libraries.

### CACHING

Marmalade comes with native APC and XCache support. If you don't want to use caching, you don't have to. If you decide to use caching and then disable it one day, your caching system won't break. Marmalade will automatically fall back to a simple datastore to pass data around from different places in your app.

### SECURITY

Marmalade comes with strong security out of the box. Securing your API or web app is important and Marmalade provides strong nonce creation and validation along with API signature verification so you don't have to worry as much about unsavory characters spoofing or repeating requests.

### AUTOLOADING

Marmalade comes with autoloading out of the box. There are two ways autoloading can be used: in a PSR-4 way with namespaces, or by simply calling a class.

### ROUTER

Marmalade comes with a powerful router giving you complete control over how your URLs are defined and where they go.

## Installation:

With Marmalade, installation is very straightforward.

GET MARMALADE

1. Download Marmalade with the link above.
2. Extract the folder into your project directory.
3. Install the appropriate configuration based on your web server (i.e. Apache or Nginx). The example configurations are found in *app/Config/examples*
4. Start your web server.
5. Navigate to your server's URL. If you are developing on localhost you may navigate to http://localhost.

If you have completed these steps, you should see Marmalade's installation helper page. This page will provide you the final steps to get your application up and running.

#### Server Configuration

In order for Marmalade's routing to work properly, you must configure your web server to send all requests through *index.php*. This is called a front-controller pattern. There are example configuration files for Apache and Nginx located in the */app/Config/examples/* folder.

#### Keys

Marmalade will generate some encryption keys for you to place in your *\App\Config\Constants.php* file. These are important as they will provide security for your application when using nonces or encrypting objects.

#### Database Install (optional)

Installation of the database is optional in Marmalade unless you are using Marmalade's API signature authentication or one-time-use nonces. To install the database, set the database constants in *\App\Config\Constants.php* and refresh Marmalade's install page. If successful, the install page will tell you the database is connected and if the schema installed.

#### Cron-tab installation (optional)

Installing a Cron-tab will allow Marmalade to execute tasks on a regular schedule automatically. Installation of cron jobs may differ across different servers. Most Linux based web servers provide out-of-box cron job support. To learn more, read the cron jobs section.

## Folder Structure:

Before getting into the details of configuration, we should explain some of the folder structure you will see upon extracting Marmalade into your project directory. This guide will only cover folders who's purpose might not be immediately apparent.

### /app

The */app* folder is where you will spend almost all of your time. It consists of all of your controllers, models, and views. This folder also contains the assets folder which is where you will store all public assets (i.e. CSS and JavaScript files).

### /app/Config

This is where all user configuration will take place. *Constants.php* is for defining all constants such as database connectivity and the way the framework should operate. The *Routes.php* file is where you will define all of the routes for your app. Lastly, the *Hooks.php* file is for defining any runtime hook you want to apply to customize your web app.

### /assets

The */app/assets* folder contains all publicly available assets your app will use or provide. This will contain CSS, JavaScript, custom fonts, or any user-downloadable content.

***Note: PHP files should NOT be stored here unless you want users to be able to download them.***

### /marmalade

This folder is Marmalade's core. Many user defined classes will extend classes from this directory. It may be important to reference these classes to see how they work and what their namespace declarations are.

## Configuration:

Configuration in Marmalade is simple. The only things you need to edit to get started are *Constants.php*, *Hooks.php*, and *Routes.php* located in */app/Config*.

### Constants.php

This file contains all of the constants needed for Marmalade to run. The notes next to every constant should give a good idea what they are used for.

When defining URL or folder location constants like ROOT_URL or ASSET_URL, leave off the trailing slash. For example, the ROOT_URL should be written as an empty string most of the time. This way, whenever you use a location constant, you can prepend a "/" which will help keep locations consistent even if you move your files around in the future. In Marmalade, this mainly shows up when using ASSET_URL in views. If you want to load a CSS file located under a "css" folder in the assets folder, you would use *ASSET_URL."/css/my.css"*

The database constants are not necessary to run Marmalade. However, if you want to use nonces or API authentication you will need to set up the database. By default the Marmalade\Database class is set up for MySQL. This class can be extended or simply ignored if you use a different database.

If you enable *XHR_DATABASE_NONCES* or *API_DATABASE_NONCES* you should install the cron tab example provided to clear out nonces older than their expiration on a regular basis. Otherwise, your database may become full of old data. This is set to run on the *one_day()* cron job in Marmalade.

### Routes.php

This file contains all of the routes your app will utilize. This consists of both static and dynamic routes. This topic will be explored more deeply in the next section.

### Hooks.php

The hooks file contains methods that Marmalade will call to allow the user to handle certain situations. Marmalade will work perfectly fine without you editing any of these, but if you would like to handle certain things like 404 errors yourself, this is the place to do it.

## How It Works:

Marmalade uses the front controller pattern, simply meaning that all requests are routed through the same *index.php* file. Marmalade then takes the requested URI and finds a matching route that has been set up (i.e. "/blog"). Once Marmalade has matched a route, it instantiates the "Controller" defined in the route and then executes a function of the controller which is also defined in the route. This function can instantiate a "Model" which manages the data for that particular route or resource. Depending on your app, you may not need a model, so this step is optional. This function also loads view(s) (the representation of the data) into the Response class to be rendered for the user. Once the route's controller method has been executed, the response is output for the user to see with all appropriate headers.

### Controller

The controller is the heart of how your route/page will work. The controller instantiates and manipulates models which it can then pass into the appropriate view(s) for the user. Controllers should be thin and not be responsible for any business or application logic. This should be left to the models.

### Model

Models manage the data backing your application. For a typical website, this can be as simple as the meta-data for a page (i.e. page title, meta description, .etc). For larger applications, this should include database queries and application logic for managing your application's data.

While you could theoretically do all of this work inside the controller, it is not recommended due to a general desire to separate your concerns. Several routes might use different controllers but still have the potential to manipulate the same set of data in your application. With a single model, this is possible. Without using a model, you code will most likely become difficult to understand and maintain.

### View

Views handle outputting the data from the model to the user. For a typical website, these views could include reusable parts like a header and a footer. For more complex applications, views handle representing your data in the way you would like to send to your users.

Marmalade can handle views in a couple ways. Views can either be a standard PHP file (i.e. header.php and footer.php) or they can be class objects that extend *Marmalade\Views\View*. If you are using a PHP file view, it will have access to a *$model* variable (this is sent from the controller). When using a View object, all of your output will be handled in the *build_ouput()* function. This function takes a parameter of *$model* and will be accessible to you as well. The *build_output()* method is also called from the controller.

## Database:

Marmalade comes with an optional MySQL database install. The database only contains tables for nonces and API users. If you don't need true nonces (nonces that expire after one use) and are not using Marmalade as an API with authentication, you do not need to install the database. If you are planning on using API authentication or want once-time-use nonces you can install the database by following the installation instructions above.

## Namespaces in Marmalade:

Marmalade fully utilizes namespaces. If you are unfamiliar with namespaces this page might be a good place to start.

Marmalade's autoloader does not work without namespaces. This is intentional, because having to traverse the whole app directory looking for a class file is inefficient. With namespaces, the autoloader knows exactly where to look for a class and prevents you from having to make sure all of your *requires()* and *includes()* are in order. In addition, when using namespaces, it's very easy to know where to find a class to modify it. That being said, **it is very important that your namespaces reflect your folder structure; otherwise, Marmalade will not work correctly. The only exception to this is if you have added a custom namespace prefix.**

Marmalade also supports namespace prefixes. Namespace prefixes allow you to define a shorter prefix for a directory or nested directory. For example, you could create a "Controllers" namespace prefix and register its base directory as "app/Controllers/". That way, instead of writing *App\Controllers\BlogController* you could just use *Controllers\BlogController*. There are a few default namespace prefixes in Marmalade:

- **Marmalade**: Used for all Marmalade core classes. Its base directory is the *marmalade* directory.
- **MarmaladeApp**: Used for Marmalade configuration files. Its base directory is the *app* directory.
- **App**: Default user application namespace. Its base directory is the *app* directory.

It is also possible to define your own namespace prefixes using *Hooks::register_custom_namespace_prefixes()*. You may also assign multiple base directories to the same namespace prefix and Marmalade's autoloader will search them all for the appropriate class.

For more information on how Marmalade's autoloader works with namespaces, see the PSR-4 specification.

## Routing:

Routing in Marmalade is handled solely through the *Routes.php* file located in */app/Config*. This file has a function called load_routes() where you will add all the routes for your application.

In order for routes to show up accurately in a nav (when not using Marmalade as an API), all child paths should directly follow their parent. Also, if you are using namespaces, make sure to prefix your controller (i.e. My\Namespace\Controller).

To add a route, use the passed in $routes array and add created *Routes* to the array. Here are some examples:

```
$routes[] = (new Route(""))->get("App\Controllers\MyPageController:home");
$routes[] = (new Route("products"))
   ->get("App\Controllers\MyPageController:products")
   ->post("App\Controllers\MyPageController:products_post");
$routes[] = (new Route("products/product-1"))->get("App\Controllers\MyPageCor
$routes[] = (new Route("products/product-2"))->get("App\Controllers\MyPageCor
$routes[] = (new Route("blog"))->get("App\Controllers\MyBlogController:all_po
$routes[] = (new Route("blog/{post}"))->get("App\Controllers\MyBlogControlle
```

```
class Route(string $path, [, array $options])
```

Properties

**Path:**
The path of route. Take note that there is no leading or trailing slash. There is no need to provide leading or trailing slashes, this will only slow the router down. The root of your application will always have an empty string for its path (as shown in the first example). The path is only the section of the URL after the domain.

The path can also include a named variable as shown in the last example. There can be any number of these named variables and they will be passed into the controller method as a parameter. The syntax for adding a path variable is {your_variable_name}.

**Options:**
An optional array for any global route options. Any option placed in this array, will be accessible to all actions regardless of the HTTP verb used. The list of options can be found in the *Route* class

located in */Marmalade/Route.php*. This is also where you may define any custom options you would like to store in your route.

You may access these options through the active route stored *Marmalade::get_instance()->route*. You may then use the *$route->option("your option")* to retrieve an option from your route.

## Methods

Then *Route* class in Marmalade provides helper functions for the main types of requests you will make (GET, POST, PUT, DELETE). The helper methods corresponding to these HTTP methods return the *Route* instance in order to allow method chaining. This allows you to quickly add support for additional HTTP methods to your Route.

Marmalade automatically handles the OPTIONS and HEAD HTTP methods, so you do not need to worry about defining routes for those methods.

**$action**
All of the HTTP method helper methods take an "action" parameter, this is for the controller class and the method that will handle the route for the given HTTP method. This parameter is the fully qualified name of the controller class (including namespaces) concatenated with a name of a method from the controller. This controller and method are executed when someone navigates to the matching *Route*. The controller name and method name must be concatenated with a colon character ":". The controller class should extend Marmalade\Controller or another class that extends Marmalade\Controller. The controller is responsible for handling the route, manipulating/creating a model if needed, and loading the views necessary for output.

If you have the USE_API_VERSIONING constant set to true, you must specify your action with an array with the key representing the version number and the value representing the action. You can learn about this in further detail in the API versioning section.

If you want, you may have separate controller for every route; this does not affect Marmalade negatively. However, if you do not need to do a lot of work in a controller, it might make sense to have fewer controllers with more methods to render your routes.

**$options**
In addition, the helper methods take an optional $options parameter. This parameter will store options specific to the *Route* and HTTP method. You may access these options through the active route stored *Marmalade::get_instance()->route*. You may then use the *$route->option("your option")* to retrieve an option from your route. Any options stored for a specific HTTP method will override the *Route's* global options.

```
public Route get(string action [, array options])
```
Add support for a GET request.

public Route post(string action [, array options])
Add support for a POST request.

public Route put(string action [, array options])
Add support for a PUT request.

public Route delete(string action [, array options])
Add support for a DELETE request.

public Route custom(string $http_method, string action [, array options])
Add support for a custom HTTP method request. This method allows you to add support for HTTP methods not natively supported by Marmalade (i.e. PATCH).

## Request Class

The Request class is a simple abstraction of the request to your application. It contains all of the basic data of the request and provides a simple way to access your information while also parsing request bodies with known Content-Type headers.

```
abstract class Request
```

Properties

static string $http_verb
The HTTP verb used in the request (GET, POST, PUT, DELETE, .etc).

static array $headers
All of the HTTP headers from the request.

static int $api_version
The API version requested. The default value is "0".

static string $host
The requested host.

static string $uri
The requested URI path.

static array $query
The parsed query string.

static string $raw_query
The raw query string

static boolean $is_tls
TRUE if the request is secured with HTTPS

static string $raw_body
The request body. Raw body will be empty if the content type is multipart/form-data.

static mixed $body
The parsed (if possible) request body.

static string $content
The base Content-Type of the request (no extra parameters).

static boolean $is_xhr
TRUE if the request was sent as an XHR/AJAX request. This is only set through *\Marmalade\Controllers\XHRController*.

## Response Class

The Response class is an abstraction of your response based on the request sent to your application. It handles returning the status code, headers, and body of your response.

All of the response body is loaded into the *php://temp* stream as your views are loaded in. Once all the views are loaded and the controller has finished executing its action, the Response class will send the body to the client in chunks with the chunk size specified by *Constants::RESPONSE_CHUNK_SIZE*.

```
abstract class Response
```

Methods

**public static void set_header(string $header, string $value)**
Set an HTTP header to be returned in the response. Make sure to use the correct case when dealing with headers. "Content-Type" is not the same as "Content-type."

**public static void remove_header(string $header)**
Remove a header from the response.

**public static array get_headers()**
Retrieve all of the headers to be sent in the response.

**public static void set_output(string $content)**
Set the response body to the given string of content.

**public static void append_output(string $content)**
Append the given string of content to the response body.

**public static void set_status_code(int $code)**
The HTTP code to be returned to the client.

## Creating a Page; from start to finish:

If all you require is a static website, you can be up and running with Marmalade in just a minute or two. Once you have set up your configuration options in *Constants.php*, open up *Routes.php* located in */app/Config/Routes.php* and add the following line of code to the *load_routes()* method:

```
/app/Config/Routes.php

$router->add_route("GET", "", "App\Controllers\PageController:hello_world");
```

After you have added your route, go ahead and create a file called "hello_world.php" in your Views folder located in */app/Views*. Inside of that file paste the following code:

```
/app/Views/hello_world.php

<h1>Hello World!</h1>
```

Once you have done this, create a new file called *PageController.php* in */app/Controllers* and place the following code inside:

```
/app/Controllers/PageController.php

namespace App\Controllers;
use \Marmalade\Controllers\Controller;

class PageController extends \Marmalade\Controllers\Controller {
   function hello_world() {
      $this->load_view(APP_DIR."/Views/hello_world.php");
   }
}
```

Now, if you navigate to the root of your project in your browser, you should see "Hello World!"

*load_view()* can be called many times in the same controller function to load many different views. For example, you may load a "header.php," "blog.php," and "footer.php" to show a standard web page. This allows you to separate your views into smaller pieces to help your code stay DRY (don't repeat yourself). This is optional, but recommended.

## `void Controller::load_view(mixed $view [, mixed $model])`

### Parameters

**View**
This parameter can either be:

- A string representing the full location of a standard PHP file
- An view object extending *\Marmalade\Views\View*. All output must be generated in the *View::build_output()* function.

**Model**
This optional parameter is a model that will be passed into either the *build_output()* function or into the template PHP file.

### Return Value

None

**Using parameters in a Route**

It is also possible to use variables/parameters in routes. These variables can be easily defined by using brackets in your *add_route()* call. For example, let's add another route to our Routes.php:

```
$router->add_route("GET", "/blog/{id}", "App\Controllers\PageController:blog_
```

Now let's create a new controller method in our PageController to handle this route:

```
/app/Controllers/PageController.php

function blog_single($id) {
   $this->load_view(APP_DIR."/Views/blog_single.php", $id);
}
```

You will notice a difference in this code if you look closely. In the method declaration there is now a parameter being passed in. This parameter is being passed from the *Router*. The router will take the variables passed in from the requested URL for routes that have variables defined and pass them into the controller method as parameters. If there are two variables, two parameters will be passed to the controller, .etc.

Another difference in this code is that we are passing the $id variable as the model for the blog view. You can send this however you like (array, object, .etc), but for simplicity, we will just pass it as the model.

Let's see this in practice. Go ahead and create another view file called "blog_single.php" and place the following code:

```
/app/Views/blog_single.php

<h1>The requested blog post's id is: <?php echo $model; ?></h1>
```

An important thing to note here is that from the perspective of all Views, all data passed in is contained in the *$model* variable. So even though in the controller the blog id variable was contained in the *$id* variable, to the view, it is *$model*.

Now, if you navigate to "/blog/1" you should see the requested blog post set to "1". If you navigate to "/blog/2000" you should see the requested blog post set to "2000", .etc. This also works with strings and any other URL valid characters.

Great job! You've successfully implemented two whole routes including one with a variable URL!

## XHR/AJAX Requests:

Marmalade will work out of the box with XHR/AJAX requests. The XHR route is set up by default in the *Routes.php* file. The default route for XHR requests is */xhr/{action}*. You will need to extend *\Marmalade\Controllers\XHRController* and add methods for all XHR actions. For example, if you wanted to add an XHR action to retrieve all blog posts you could use the URI */xhr/posts*. You would then need to create a *posts()* method in your XHR controller.

Marmalade's default XHR setup includes the use of nonces and user-defined XHR actions. Nonces are a security measure that help prevent malicious use of your application. If you have installed Marmalade's database, nonces will be true nonces and only be able to be used once. Without Marmalade's database, nonces will only expire after the user defined XHR_NONCE_EXPIRATION_MINUTES. Nonce expiration will work if the database is installed or not.

Nonces will be verified for every XHR action by default. To disable this, you will need to override the *execute* method. You can copy the *XHRController->execute()* method for a starting place.

You may define the nonce in one of three ways:

- A "nonce" query parameter in the URL
- A "nonce" field in the JSON request body
- The Authorization header

To create a XHR POST request, you can use the following code:

```
var xhr = new XMLHttpRequest();
var data = {nonce: "<?php echo \Marmalade\Security::create_nonce("posts"); ?>
xhr.open("POST", "<?php echo ROOT_URL; ?>/xhr/posts");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function() {
   if (this.readyState === 4 && this.status === 200) {
   }
}
xhr.send(JSON.stringify(data));
```

You may also create a simple XHR GET request with the following code:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "<?php echo ROOT_URL; ?>/xhr/posts?nonce=<?php echo \Marmalad
xhr.onreadystatechange = function() {
   if (this.readyState === 4 && this.status === 200) {
   }
}
xhr.send();
```

## Error Handling

There are certain instances where an error may occur in your script due to a bad request, lack of a database connection, an invalid page URL, .etc. Marmalade handles many of these types of errors for you. Marmalade will, by default, clear the existing response, set the status code to the appropriate status code, and output a basic JSON representation of the error.

*To be clear, PHP syntax errors, warnings, and notices are NOT handled by Marmalade.*

### Error Handling with your own ErrorController

You have the option to provide your own ErrorController if you would like to handle errors with your own view. For example, if you would like to handle a "404 Page Not Found" error, you can register your own error controller in *Hooks::get_error_controller()*. This controller must extend *\Marmalade\Controllers\ErrorController* in order to work properly.

If you have supplied your own error controller, it will now receive every error that is thrown in Marmalade. If you only want to handle particular errors, you may specify in your controller to load the default view for all the errors you do not want to handle.

### Creating Your Own Errors

Custom errors are also supported in Marmalade. In order to show a custom error, instantiate an instance of *\Marmalade\Models\ErrorModel* with your desired HTTP status code, error type, error message, and any extra data you wish to pass to your error view. Once you have your error model, you may call *Marmalade::error($model)*.

## class ErrorModel(int $http_code, string $type, string $message [, array $extras])

Parameters

**$http_code**
The status code to return to the browser.

**$type**
A short description of the error for identification purposes.

**$message**
A message describing the error.

**$extras**
Any additional information to pass back to the view.

## void Marmalade::error(extends \Marmalade\Models\ErrorModel $error_model)

Parameters

**$error_model**
The error model to pass to the error controller and then to the corresponding error view.

## class ErrorController

Methods

public void **error($error_model) {}**
The action that *Marmalade::error()* will call when an error occurs.

## Using Marmalade as an API:

Marmalade has native API support and was built with such applications in mind. Running Marmalade as an API is almost identical to running Marmalade as a normal web app, with few exceptions.

- Marmalade's default XHR implementation is turned off for APIs. This is handled in *MarmaladeApp\Routes*.
- Route authentication is handled through *Hooks::verify_api_signature()*.
- One header is added by default: "Content-Type: application/json" is set as the default response Content-Type in Marmalade. This header can be changed or removed using *Response::set_header()* or *Response::remove_header()* respectively.

In order to run Marmalade as an API, all you need to do is change the constant *IS_API* in the Constants class to "true." Once this constant has been set to true, your app is ready to run as an API.

If you have the constant *REQUIRE_AUTHENTICATION* enabled, you will not be able to test your endpoints without making a proper Marmalade API request. An example of how this can be done can be found int the *Marmalade\APIClient* class. Marmalade's API authentication is based on a public and private API key hashing algorithm. **In order to use Marmalade's default API authentication, you must use Marmalade's database install.** To learn more about making a request, read the making an API request section.

You may also use your own custom API authentication or disable authentication altogether. If you wish to write your own authentication, use *Hooks::verify_api_signature()* to do all the work you need. You will also need to handle all errors and kill the script yourself, if necessary.

## API Versioning:

Marmalade has API versioning support out of the box. It was designed in hopes of being flexible and simple. To begin using API versioning, you must first change the values of the IS_API and USE_API_VERSIONING constants in the *App\Config\Constants* class to TRUE.

Once the constants values have been switched, Marmalade will now support passing an array of actions into the *Router::add_route()* method. One route will contain the actions for every version that Route has available to it. If a user requests a version that is not defined in a route, they will receive a 404 error. This allows you to version some parts of your API without affecting others.

When using versioning, make sure that both your routes and class names use the correct namespaces. Also remember that namespaces must follow the folder structure of your app.

Here are some examples of versioning:

```
$routes[] = (new Route(blog/"))->post(array(
   1 => "App\Controllers\V1\BlogController:new_post",
   2 => "App\Controllers\V2\BlogController:new_post"));
$routes[] = (new Route("blog/{post_id}"))->put(array(
   1 => "App\Controllers\V1\BlogController:update_post"));
```

In this example, there are two routes. One route has two available versions while the other route only has one available version. This provides a way to having a rolling-release API where new version of endpoints can be made public when they're ready. This is of course optional. You may version everything at the same time if you like.

It is also important to note that the folder hierarchy can be different than this example. In this example the folder structure would be:

```
app
  Controllers
    V1
      BlogController.php
    V2
      BlogController.php
assets
marmalade
```

However, you could also structure your app the following way as long as your folder structure is accurately reflected in your namespaces.

```
app
  V1
    Controllers
      BlogController.php
  V2
    Controllers
      BlogController.php
assets
marmalade
```

#### Requesting a Version as a Client

When using API versioning, clients of the API must request a version in a particular way. Marmalade handles versioning differently than most other frameworks with the intention of being more RESTful. Instead of using an endpoint like "/v2/posts," clients of Marmalade will define the version they are requesting in the *Accept* header. There are a few formats you may use to request a version in your accept header:

```
application/vnd.mydomain.myapp.resource.v1+json
application/vnd.mydomain.myapp.resource.v1
application/vnd.mydomain.myapp.v1+json
application/vnd.mydomain.myapp.v1
```

The "resource" and the additional "+json" content-type request are optional, but may be helpful in versioning particular resources. The "vnd.mydomain.myapp" section must match the *VENDOR_ID* constant in the Constants class. Marmalade does nothing with these extra pieces of information. The goal was to allow flexibility with the accept header for development.

It should be noted that if you plan on using Marmalade's default API versioning implementation, passing the version in the accept header is required. However, if you want to roll your own versioning, you may. In order to do this, use the Hook *Hooks::get_custom_api_version()*.

## Making an API Request:

Making an API request in Marmalade is very simple without using authentication. If you are using authentication, making an API request is a little more specific. With these differences in mind Marmalade provides an *\Marmalade\APIClient* class to help abstract the specifics of authenticating requests in Marmalade.

#### Unsigned API Requests

Making an unsigned (unauthenticated) request in Marmalade is just the same as making a normal network request. There are no extra steps that need to be taken unless you are trying to access a particular version of the API. You may use the *APIClient* class, but you can also completely ignore it.

#### Authenticated API Requests

When you are accessing an endpoint that requires authentication, it is easier (but not required) to use the provided *APIClient* class. *APIClient* will handle signing the request and passing the correct Authorization header.

Marmalade's default signature generation is based from a public and private API key that are stored in the "api_users" table of the database. This is why **you must install the database in order for Marmalade's default API authentication to work**.

To generate an API user with their keys, use the *\Marmalade\Models\APIUserModel::create()* method. Once you have a valid API user, you can use their public and private keys to make authenticated requests.

## class APIUserModel

### Properties

**public int $id**
The unique id for an *APIUserModel* instance.

**public string $public_key**
The public key for the user. This is passed in all API requests.

**public string $private_key**
The private key for the user. This is used to sign all authenticated requests. The private key should never be accessible to the public. Only the user and the server should know about the private key.

### Methods

**static APIUserModel|boolean load_by_id(int $id)**
Retrieve a user by an id and return an *APIUserModel* instance. If a user is not found, FALSE is returned.

**static APIUserModel|boolean load_by_public_key( string $public_key)**
Retrieve a user with their public and return an *APIUserModel* instance. If a user is not found, FALSE is returned.

**static APIUserModel|boolean create()**
Creates and inserts a user into the `api_users` table and returns the created *APIUserModel* instance. FALSE is returned if there was an error.

**static int|boolean delete(int $id)**
Delete a user from the database. If a user was deleted, 1 will be returned. If a user was not deleted 0 will be returned (this will occur if the user was already deleted or could not be found). FALSE will be returned if there was an error.

With the user's keys, you may now create an instance of *APIClient* and make an authenticated request to a resource.

## class APIClient

Methods

public APIClient **__construct($host, $app_name [, $public_key [, $private_key [, $verify_ssl = true]]])**

string **$host**
The host domain the client will be calling (i.e. https://my-api.com).

string **$app_name**
The application name. This must match *APP_NAME* found in *Constants.php*.

string|boolean **$public_key**
Your public key. FALSE if the API does not require authentication.

string|boolean**$private_key**
Your private key. FALSE if the API does not require authentication.

boolean **$verify_ssl**
Sets if SSL certificates should be verified or ignored. Default is TRUE.

public string|boolean **call($http_verb, $uri [, $headers [, $request_body [, $sign_request]]])**

string **$http_verb**
The type of request being made (GET, POST, PUT, DELETE, .etc).

string **$uri**
The endpoint URI of the resource to call. For example: "/products".

array **$headers**
The headers to send in the request.
Example: array("Content-Type" => "application/json");

string|array **$request_body**
The request body to send to the API (array if sending a file or multipart/form-data, string for everything else).

When sending a file, you must use the Content-Type multipart/form-data and send your $request_body as an array. This array must use an instance of *\CurlFile*. Files do not work with PUT requests at this moment.

**boolean $sign_request**
TRUE if the URI being called requires authentication. FALSE if the URI does not require authentication. The default value is TRUE.

**public mixed get_info()**
Returns the curl response information.

**public int get_response_code()**
Returns the response code from the request.

Here are some examples of using the *APIClient*:

```
// Create a client instance
$client = new \Marmalade\APIClient("http://example.com", "my_app"
   "xsd9fa79s6df897a6s9d87f698as6d99asd987fa98s7df987as98d7f87as9d7a",
   "gf789d7s98df7g987sscv7b8s89d79b86sdb56s4d5b3s3d53sdb47s64s8s5b85");

// Send a file
$response = $client->call("POST", "/file-upload",
   array("Content-Type" => "multipart/form-data"),
   array("file" => new \CURLFile("/my/full/file/path/file.txt")));

// Send a JSON body
$response = $client->call("POST", "/blog",
   array("Content-Type" => "application/json"),
   '{"title":"My Title","body":"This is a test body"}');

// Send multipart/form-data
$response = $client->call("POST", "/blog",
   array("Content-Type" => "multipart/form-data"),
   array("title" => "My Title", "body" => "This is a test body"));

// Send a x-www-form-urlencoded body
$response = $client->call("POST", "/blog",
   array("Content-Type" => "application/x-www-form-urlencoded"),
   "title=My+Title&body=This+is+a+test+body");
```

### Manually Signing Requests

It is also possible to manually sign requests if you want. To do this, the *APIClient* is the best tool of reference. The signature generation algorithm is as follows:

```
// Create the request signature
$timestamp = floor(microtime(true) * 1000);
$canonical_request = "{$http_verb}\n".
   "{$uri_path}\n".      // The path. Leading and trailing slashes will make
   "{$query_string}\n".  // The query string without the "?"
   hash("SHA256", $request_body);
$string_to_sign = hash("SHA256",
   "{$app_name}\n".
   "{$timestamp}\n".
   hash(Security::HASH_ALGO, $canonical_request));
$signing_key = hash_hmac("SHA256", $timestamp, $private_key);
$signature = hash_hmac("SHA256", $string_to_sign, $signing_key);

// Headers
Authorization: {$app_name}:{$public_key}:{$signature}
{$app_name}-timestamp: {$timestamp}
```

It is important to note that if you are sending data with the Content-Type of "Multipart-form-data", you must use an empty string "" for the *{$request_body}* variable. This is because Multipart-form-data requests are not available in the php://input stream and this is what Marmalade uses to retrieve the request body to test signatures against.

## Cron Jobs:

Marmalade has native support for cron jobs. Cron jobs are a way of scheduling certain tasks on a server. In Marmalade's case, cron jobs are a way of executing certain tasks in your web application.

In order to run cron jobs in Marmalade, you must install a cron tab on the server marmalade is running on. There is an example of a cron tab in *app/Config/examples/* that you may base your cron tab off of. A valid marmalade cron tab would be:

```
*/5 * * * *     php /path/to/marmalade/cron.php five_minutes
*/30 * * * *    php /path/to/marmalade/cron.php thirty_minutes
0 * * * *       php /path/to/marmalade/cron.php one_hour
0 */6 * * *     php /path/to/marmalade/cron.php six_hours
0 */12 * * *    php /path/to/marmalade/cron.php twelve_hours
0 0 * * *       php /path/to/marmalade/cron.php one_day
```

Your cron tab will execute the file *cron.php* located in the *marmalade* folder. The argument that is passed in is the function/method to call in your cron controller. You may create a cron controller and base it off of *Marmalade\Controllers\CronController*. In order to specify which controller to use, edit the *Hooks::get_cron_controller()* method and return a string of the full name of your controller. For example, if your cron controller is called "CronController" and located in a "Controllers" directory, you would return "App\Controllers\CronController".

## Marmalade Nav:

Marmalade comes with a navigation builder built in. Marmalade's navigation builder works based on the routes provided to the router. Every *Router* object has a method *Router::get_nav($options, $nav_object)* that is used to return a navigation builder object. This builder can build a navigation menu based on the routes provided. For convenience, there is a helper function in the *Util* class called *build_nav():*

```
string Util::build_nav(array $options [, mixed $nav_object])
```

Parameters

**Options:**
An array of options to pass to the Nav class

**Nav Object:**
A custom nav object to send into the Nav class instead of the default $nav_object created from the Router's routes. This allows for creating a custom navigation menu outside of the routes in your application. This object may differ from Marmalade's default depending on if you are using a custom Router or not.

Return Value

An HTML string of a navigation menu.

Several different options can be passed into the default Nav provided by Marmalade. However, if you are using a custom router, make sure that the router you are using supports the following options:

**class** (string)
Place a custom HTML class on the *nav* element.

**start_depth** (int)
The starting depth of the nav. If you only want second level items visible, set the *start_depth* to 1. This is a 0 index based integer. 0 refers to the first level of routes.
*Default: 0*

**end_depth** (int)
The depth to stop the menu. If you only want to show first level items in a nav, set *end_depth* to 0;
*Default: 99*

**branch_mode** (boolean)
Branch mode designates a different view of the nav based on the current page. If branch mode is turned on, in relation to the current page, only children, siblings, ancestors, and the ancestor's siblings will be shown. This mode is particularly helpful when only trying to show a sub-nav when a page has child items. Branch mode is exclusive and no other mode can be enabled with it.
*Default: false*

**ancestor_mode** (boolean)
Ancestor mode is also based on the current page and will only show the current page and its ancestors in the navigation menu. Ancestor mode is helpful for showing breadcrumb menus when used with *flatten*. Ancestor mode may also be used in conjunction with sibling mode.
*Default: false*

**sibling_mode** (boolean)
Sibling mode will only show siblings of the current page as well as the current page. This mode can be used in conjunction with ancestor mode.
*Default: false*

**hide_if_empty** (boolean)
If this option is set to true, the whole nav (including its container) will not be output to the screen.
*Default: false*

**flatten** (boolean)
If flatten is set to true, the entire navigation menu will be flattened into one *<ul>* element. All children will be in the same list. This mode is helpful when showing breadcrumbs and when used in conjunction with

ancestor mode.
*Default: false*

## Object Caching In Marmalade:

Marmalade is set up to work with XCache and APCu natively. This is handled through the *Marmalade\Cache* class. This class can be used by your application as well as any extension. By default Marmalade only caches your application's routes once they are built (this requires the *ENABLE_CACHE* constant to be set). This enables Marmalade to store the routes array in memory instead of having to rebuild it on every request.

If caching is disabled in Marmalade it will automatically use a native PHP array to store items. So even if caching is disabled, all of your code will continue to work the way you intended it to. In this mode, the *Cache* class can be a global storage container for data you want to pass around your application. Using the *Cache* class this way with caching enabled would not be recommended unless you truly want your data to persist and be identical for all users.

### `class Marmalade\Cache`

Methods

**static boolean Cache::has(string $key)**
TRUE if the cache has the requested key.

**static mixed Cache::get(string $key)**
Returns the cached object with the matching key.

**static boolean Cache::set(string $key, mixed $var [, int $ttl])**
Store an object in the cache with the given key.

**static boolean Cache::delete(string $key)**
Remove an object from the cache.

## Tips and Tricks:

There are several things that have been included in Marmalade to make your life easier. This section is meant to take note of some of those things and make your aware of them.

### Constants

Make sure to use the provided constants whenever you can. There are several helpful constants available in Marmalade:

- APP_DIR: The location of the "app" directory. This is defined by Marmalade and should not ever change.
- ROOT_DIR: The location of the root directory of your applciation.
- ROOT_URL: The root URL of your application. This is defined in the Constants class.
- ASSET_URL: The URL of your assets directory. This is defined in the Constants class.

### PageModel

A way to help your static pages get rendered with the correct meta data is to make a PageModel. This can be done very simply and then passed into your views. A simple page model example could be as follows:

```
/app/Models/PageModel.php
namespace App\Models;

// Page model
class PageModel extends \Marmalade\Models\Model {
   public $title;
   public $meta;
   public $styles = array();
   public $scripts = array();

   // Constructor
   public function __construct($title = "", $meta = array()) {
      $this->title = $title." - Ethos Softworks";
      $this->meta = $meta;
   }
}
```

Once you have a page model, you can pass it in to your views and use the data. A simple page view could use this model in the following manner:

/app/Controllers/PageController.php

```
namespace App\Controllers;
use \Marmalade\Controllers\Controller;

class PageController extends \Marmalade\Controllers\Controller {
   function home() {
      $model = new PageModel("Hello World");
      $model->styles[] = ASSET_URL."/css/main.css";
      $this->load_view(APP_DIR."/Views/hello_world.php", $model);
   }
}
```

/app/Views/hello_world.php

```
<!DOCTYPE html>
<html>
   <head>
      <?php foreach ($model->meta as $key => $value) { ?>
         <meta name="<?php echo $key; ?>" content="<?php echo $value; ?>">
      <?php } ?>
      <title><?php echo $model->title; ?></title>
      <?php foreach ($model->styles as $style) { ?>
         <link rel="stylesheet" href="<?php echo $style; ?>">
      <?php } ?>
   </head>
   <body>
      <h1>Hello World!</h1>
      <?php foreach ($model->scripts as $script) { ?>
         <script src="<?php echo $script; ?>"></script>
      <?php } ?>
   </body>
</html>
```