Table of Contents

Mjölnir

`--install-bigger-hammer`

DISCLAIMER: *The following documentation is work in progress and has yet to be properly formatted, proof read, or completed. It is provided as-is even in it's current state so that it may be of use, as well as for internal development purposes.*

Mjölnir, pronounced "mee-uhl-neer", is an all purpose PHP module-based library (can also be considered a framework) primarily aimed for web development but adept at any task otherwise possible though PHP. Based on a (PSR-0 compatible) cascading modular class and file system, PHP traits and convention though interfaces, the library is designed to mold itself to your use case. The main design goals are, in order:

- maintainable code
- flexible infrastructure
- reusability
- security
- ease of use & simplicity
- easy integration with other tools

In Mjölnir all classes, methods, variables, and values are replaceable, overwritable, customizable, extendable, and discardable. If it exists, it exists to be given a purpose, not as a requirement. Files, user interaction, execution, request patterns, project structure, are all designed to allow for interpretation in the context of the problem at hand.

The library is based on PHP, because PHP facilitates the libraries technical requirements via class autoloading, among other features.

The modules are designed around up to date PHP. At this time PHP 5.4.4 and above is required. Using the latest version is highly recommended. New notable and useful features in the the language will be adopted as soon as possible.

All documentation is created to be human readable, is part of the codebase, and integral part of the release process. As per the release philosophy a version can not be stable with out complete documentation. There are no API docs, since its usefulness is debatable; to conserve time it is ignored. All code has been written to be readable by itself, various docblock patterns have still been used to the extend that is useful for editor autocompletion and other tooling. Even though doc-style comments are almost intentionally omitted the code is still commented extensively; in place of machine language, paragraphs and examples are written in plain english and detail.

For understanding how the library works, and how to use it effectively, it is recommended to start with the cascading file system module; and continue to the base module. Other modules are all contextual in nature, so after understanding how the module system works feel free to skip to any point of interest to your own projects. Extensive functional examples are provided as often as possible.

1.1 Cascading File System

The Cascading File System (cfs for short) module allows the implementation of projects based on a modular pattern where points of interest in an application are split into modules (separate directories with a namespace) and stacked in order of priority, with top modules taking precedence over lower modules. The modules are built on a PSR-0 compliant structure and fully support namespaces.

If properly applied, all classes, files, and configurations on the application become easily overwritable and customizable.

The system is compatible with dependency injection but for most cases dependency injection overlaps with the module system in purpose. Modules will solve roughly the same problems with less hassle, less code, and more intuitive patterns.

This module is part of Mjölnir, but may be used on it's own for creating projects, frameworks, etc. Its only foreign dependency is to the `mjolnir\types` namespace (a pure interface module), where it retrieves type information for caching and database binding methods (both functionally optional features). `mjolnir\testing` is also used by this module, but only for behavior tests; we recommend running tests in a typical mjolnir setup so this should not be a concern for anyone who wishes to use only this module.

For creating an application based on this module, but not on mjolnir as a whole the mjolnir-template-app can still be used as a guide. Note that all the structure is merely a recommendation.

If you wish to create your own version based on this module but keep most of it you can include this module as a dependency to your project and create another class which extends it via composer. You will have to extent `\mjolnir\cfs\CFS` in your new class.

For versioning information and methodology see https://github.com/ibidem/ibidem/blob/master/versioning.md

1.1.1 Modules

In a cascading file system, modules are the foundation blocks for everything within the system. Without modules the systems can not function.

A module can contain the following,

1. Classes
2. Configuration Files
3. Files, such as Views, Themes, Vendor/3rd-party code, etc
4. Anything else ("if it fits, it's okey", eg. documentation, drafts, etc)

Classes and configuration files are the first class citizens in a module. The entire module structure is designed around classes, and configuration files are merged together, which is different to other files (including classes).

Assuming default structure is used, a module works as follows:

1. all files are located in directories (on the module root) prefixed with a "+"
2. all configuration files and (files known by the loading process) are stored in the main application files directory (by default "+App"), any other directories (eg. "+Docs") are not available in the file system.

Under normal conventions

1. Configuration files are stored in `+App/config`
2. View files are stored in `+App/views`
3. Theme files are stored in `+App/themes`
4. Drafts are stored in `+App/drafts`
5. Vendor files are stored in `+App/vendor`
6. Internationalization/grammar files are stored in `+App/lang`
7. Functions are stored in `+App/functions`
8. Special classes are stored in `+App/includes`
9. Behavior tests are stored in `+App/features`
10. Unit tests are stored in `+App/tests`
11. Special temporary files are stored in `+App/tmp`

Note: The`+App/honeypot.php` files are designed to be read by your IDE to facilitate autocompletion, refactoring, etc; they serve no other purpose and the only time you should be opening them is when your IDE is failing to scan them.

To get started with a base structure go to: https://github.com/ibidem/mjolnir-template-app and follow the instructions outlined in the `README.md` file (github should offer you a parsed version at the given link).

1.1.2 Loading Classes

For a class (from a registered module) to be loadable the following conditions must be met.

1. The module in which the class is present must be known by the autoloader; meaning, when using the recommended structure you must specify it in your `environment.php` file. If you are relying on a non-default structure this condition resumes to: it must be included by `CFS::modules`, `CFS::frontmodules`, `CFS::backmodules`, or for namespace only access `CFS::namespacepaths`.
2. If underscores within the class name are replaced with directory separators specific to the system, the class should result in a valid path segment and in combination with the path to the module itself and the current extention (defined by the current value of EXT) should produce a valid path to the class file. Confused? Let's say we have the example class `Controller_AcmeOrganization` the correct path to it if `MODULE` is the path to the module, and `EXT` is `.php` is `MODULE/Controller/AcmeOrganization.php`. If the class is placed in any other file it will not be recognized.
3. The full namespace of the class should correspond (exactly) to the namespace defined for the module. So as an example, the `\mjolnir\access\ReCaptcha` class resides in the `access` module, which has the namespace `mjolnir\access`.
4. Another file with the same path segment pattern (ie. same class name) is not available in a higher module (this DOES NOT apply to namespace invocation; discussed bellow)

If all conditions are met the class will be loaded. Otherwise it will be passed on to any other autoloader on the system (eg. bridges to other module systems, composer's autoloader, etc).

Let's take an example,

```
<?php namespace acme\security\access;
# MODULE/Controller/AcmeOrganization.php
class Controller_AcmeOrgnaization
{
    // ...
} # class
```

We can call this class in a number of ways. First we can call it by namespace:

```
\acme\security\access\Controller_AcmeOrgnaization
```

If all else fails this method will always work assuming you have composer setup correctly, since all modules are PSR-0 compliant.

If we say don't care what `Controller_AcmeOrganization` it is in `acme\security` we can simply call it by:

```
\acme\security\Controller_AcmeOrganization
```

Similarly, if we don't care for the security segment, we can call:

```
\acme\Controller_AcmeOrganization
```

There are however three conditions to this shorthand namespace resolution:

1. the full namespace must be a namespace known to the cascading file system; namespaces only known via composer will not resolve.
2. you may only omit entire segments at a time; so `\acme\sec\Controller_AcmeOrgnaization` (note: "sec" instead of "security") will not resolve to our example class.
3. the namespace you are using as a shorthand must not be registered in the cascading file system. This is purely by design to prevent false positives. If the namespace is registered and the class is not within it then the class will NOT resolve. This behaviour is also intended to avoid confusion.

When extending any class in mjolnir it is recommended (and expected) you use the shorthand `mjolnir` namespace; so if we had a class `mjolnir\example\Hello` we expect you to use:

```
class Hello extends \mjolnir\Hello
```

Instead of this form:

```
class Hello extends \mjolnir\example\Hello
```

This allows us to (if needed) move the `Hello` class to `mjolnir\legacy` with out breaking your code. Remember this type of loading only works on registered namespaces and not namespaces available via composer.

The last (and most common) way of resolving the class is via the special `app` namespace, ie.

```
\app\Controller_AcmeOrgnaization
```

When we resolve a class via the `app` namespace we are always asking for the most advanced implementation of said class; which simply boils down to which namespace holding such a class is at the top of the stack in your module declarations (or as a result of your your module declarations; depending on your setup). In Mjölnir every use of every class is via the `app` namespace so by creating a top level class in your application you can replace and/or customize any class in the system.

The only direct dependencies to the library files are the interfaces which have been used with explicit namespaces to discourage bad patterns, and encourage consistency (more on this in the types section).

Namespaces must be unique

Each module may have one namespace, and that namespace you choose must be unique.

The namespace must be unique both in the project, and the world. The namespace must not appear anywhere else, on anything other then this module, even if the place it appears on is a project that does not rely on the class loading system described here. If it is PHP code, or can interchange calls with PHP code, it is an invalid namespace, because it fails to be unique.

To understand why, you have to first understand what problems namespaces solve, and how they solve them. The are three main problems:

1. name conflicts with other peoples code
2. name conflicts with your old code
3. name conflicts with your yet to be written code

Let's consider the earlier example `\acme\security\access\Controller_AcmeOrganization` as a benchmark. The first part of the namespace (ie. `acme`) solves the first problem: it is unique and can act as a "family" name for the rest of the code. One can thus safely write any function or class within it with out fear of it conflicting to one in another unknown library, framework, plugin, etc.

Eventually as the code family grows out we start having problems of managing name conflicts within it. We can avoid confusion by creating a smaller namespace within it. Since the `acme` namespace is a blank slate we can choose this time from common words, so we get the added benefit of organizing our code better at the same time, which solves the second problem "name conflicts with your old stuff".

When we grow past this point we can continue to add segments as a means of separating concerns, so when multiple modules are being created simulataniously with potentially conflicting class names the code stays safe from potential reuse of names (ie. there could be a `\acme\security\protocols\Controller_AcmeOrganization`) by working in the `acme\security\access` namespace we don't have to care, thus achieving point three in our initial problems list, future proofing.

Following the above, here are some patterns to avoid.

Namespaces as extentions of the class name, ie. `\acme\Controller\Organization`. This is very impractical, and mostly abused for purely pointless sugarcoding purposes. If `Controller` there establishes a sub space and `Organization` is a controller, then what is a controller in a namespace other then `Controller` in the same `acme` namespace, other then confusing? In addition, if all controllers are meant to go into this `Controller` namespace how can you have another `Organization` controller? The answer is "you can't", neither can you for practical applications but also mistakenly creating a class with the name `Organization` is errornous and means you have to be aware of problems 2 & 3 outlined above by yourself, rather then the namespace resolving it for you (as it should). If that was not enough one has to also consider how the classes are completely incorrect with this pattern: an `Organization` class might act the function but it is

not very intuitive and nobody will understand it as a `Controller_Organization` outside the namespace context.

Namespaces should act as a "name space" first, anything else *third*, so we recommend avoiding these "beautification" patterns.

Incidentally, the`app` namespace is actually a valid namespace. Even though it doesn't follow the exact recommendation above, it does meet the requirements due to how it functions: all classes in it are unique at runtime.

1.1.3 Loading Files

There are a few ways to load files known by the system. The first way, which is the most convenient for single files, is to use `CFS::file($file, $ext = EXT)` this will search all modules from top to bottom and stop when it finds a file; so it will give you the top file in the module stack.

Another way to load files is to load a file via its directory; this is mostly done with vendor/3rd-party code since we want some gurantee we're getting the right "config.php" and we don't really care for the file itself per se as much as we do about getting it from the correct directory. The method for this is `CFS::dir($directory)` and a simple use case example would look like this:

```
require_once \app\CFS::dir('vendor/awesomesomething').'mainclass.inc';
```

If we need all the files of the given name we use `CFS::file_list($file, $ext = EXT)` which functions almost the same way as `CFS::file` only instead of the top file we'll get back and array of all the matching files.

If we need to have a more sophisticated search, we can use `CFS::find_files($pattern, array $contexts = null, array & $matches = [])`. This function can be used even outside the context of the cascading file system by simply providing different contexts in it (in the absence of any contexts, it will default to searching all registered file paths).

A different way to get to files is by retrieving the path and doing your own handling. You can get the path to the module's root via `CFS::modulepath($namespace)` (or the practically equivalent method for standard modules `CFS::classpath($namespace)`), and the file path via `CFS::filepath($namespace)`.

1.1.4 Loading Configuration Files

To load a configuration "file" the function `CFS::config($key, $ext = EXT)` is used. In certain extreme cases you may want to explicitly make sure the configuration you are loading is coming from a physical file (and not something else, such as a database) in which case you would use `CFS::configfile($key, $ext = EXT)`.

By default configuration files are mere PHP files which return an array. If required a configuration file may be externally loaded via an include as follows:

```
$config = include 'path/to/configuration.php';
```

This however will rarely be equivalent to the result of `CFS:config('path/to/configuration')` due to how configurations are managed.

In the cascading file system the values for a configuration file is obtained as follows.

1. the system will search for all configuration files in all modules; more specifically the search will match the given pattern to `+App/config` of all modules, and just `config` for any explicit paths (such as private files).
2. the resulting arrays will be recursively merged starting from the values obtained from the bottom modules and going up. So values you place in top modules will always overwrite values in lower modules.
3. if no configuration files were present an empty array is returned

So the value of a single configuration file is not necessarily representative of the complete result.

Typically you will place defaults in the module which implements the configuration and overwrite as needed in the modules that use the configured implementation.

Because configuration files are plain old PHP code, you can have any amount of complexity in it. Here are just a few examples:

You can generate a configuration dynamically; for example if www path is not defined you may attempt to resolve the configuration to some other more useful value; remember that the configuration files are still plain old PHP files so there is very little limitation on what processing they can do.

You can split the configuration into a series of arrays and simply return the merged output; for example in the case of a script configuration, you can form small manageable arrays with points of interest (form helpers, modals, etc) then merge them and remove duplicates. You thus avoid having monolithic declarations, have an easy mechanism to dealing with script duplication, and best of all: it's far more maintainable.

You can use variables for cleaner syntax; for example in routing, with the exception of certain abstract patterns, you often have to define various repeating patterns for said routes, you can use variables to avoid this, which is

extremely useful when dealing with 40+ routes (as is the case a lot of the time). Example:

```
<?php
// segments
$id = ['id' => '[0-9]+'];
$slug = ['slug' => '[a-z0-9-]+'];
// mixins
$resource = '<id>/<slug>(/<action>)';
// access
$control = ['GET, 'POST'];
return array
    (
        "/example/{$resource}"
            => [ 'example', $id + $slug + ['action' => '(insert)'], $control ],
    );
```

You can place closures within configuration files allowing you to create a dynamic collection of them for easy management. For example url generators, such as a thumbnail or action urls for forms, a closure for generating the correct path for a given filename saves space and is very flexible.

You can translate the configuration from an external 3rd party source directly in the configuration file and output it; this means that if the source configuration is updated your configuration is updates as well; which is useful for capturing changes to defaults or extra options that become available; this may be a json, yaml, another php file, etc, or if necessary the application might even resort to going to the web to get updates (eg. list of countries, cities, etc), regardless of format, or changes in the format, when you need the configuration you merely perform a standard call.

Configuration files are resolved once. Any subsequent calls to `CFS::config` *with the same parameters merely results in the previous (cached) result. This means you can abuse calls, but it also means you should treat values from configuration files as static. A "timer" value will not update for example; but you can always use a closure within the configuration for those cases.*

If you wish to cache the resolved configuration file between requests you can add a `@cfs` key to the configuration file which will be read by the system post merge and settings parsed. By passing cachable the system will be instructed to persist the configuration values between requests. The values must be serializable for the values to be persisted so if your configuration has functions defined avoid making the configuration cachable.

Example configuration using `@cfs`,

```
<?php return array
    (
        '@cfs' => array
            (
                'cachable' => true,
            ),
        'example' => 12,
    ); #config
```

Accessing the configuration will return `[ 'example' => 12 ]` in this case.

If you need the `@cfs` key, simply write it as `\@cfs`.

Please **DO NOT** store security keys, passwords and other sensitive information in configuration files located in your source repositories. Not only is it a security liability, but it is also a pain for any development outside your production server (unless all your test servers, along with every site you ever built somehow has the same keys; which would be nonsensical).

The correct way of dealing with sensitive configuration entries is to place them in a separate file path that sits at the top of the cascading file system and outside your `DOCROOT`. The mjolnir-template-app shows an example of this: you specify the path to the private files via a `private.files` entry in `WWWPATH/config.php` and a `DOCROOT/.key.path` file for CLI access. The `DOCROOT/.key.path` is ignored via your `.gitignore` and merely contains a path.

1.1.5 Composer Integration

First of all you may load and use any composer compatible projects. It is not recommended to configure them as modules; they should be used via the namespace resolution only (so that they are handled by composer only).

Modules, as previously described, are PSR-0 *compatible*, so as long as there are no major dependencies to the cascading file system, they may be loaded directly via composer and used like a regular composer package.

If a composer based class would serve better as a class within the cascading file system, the recommended way of integrating it, assuming it was not designed to be used in this context to begin with, is to construct a wrapper and extend it. *This is also the case for any embeded code within the module.*

Ideally modules will define any dependencies via their `composer.json` file, which assuming the module is itself loaded via composer results in said dependencies being transparently pulled in and updated as is the case.

1.1.6 Overwriting Classes

To overwrite a class it's very simple: you just create a class in a higher level module.

For example, let's say your `etc/environment.php` defines the following modules:

```
$modpath.'module1' => 'demo\module1',
$modpath.'module2' => 'demo\module2',
$modpath.'module3' => 'demo\module3',
```

In this configuration `module1` has the highest priority and `module3` has the lowest priority. Or if we go by namespace we can say `demo\module1` is configured to be of higher priority then `demo\module2` which is of higher priority then `demo\module3`.

Lets say we have a class `demo\module3\Example` already defined. When we access `\app\Example` the system will resolve the class to the highest priority module and since it can not find the class in module1 and can't find the class in module2, the module3 version of the class will get loaded.

If we define a class `demo\module1\Example` however, since it's in a module of higher priority when we call `\app\Example` we'll actually get the module1 version now instead of the module3 version.

That's all there is to it.

Remember that the system may cache paths for fast resolution so if you're testing and getting the wrong version just run a `order cleanup` to flush out the caches.

1.1.7 Overwriting Behaviour in Classes

We've talked about how to replace a class, but often times what you really want is to replace functionality in a class rather then re-write the entire class.

To start with you'll first need to overwrite the class. For the sake of our example we'll assume our class is `Example` as with the previous section, and we're overwriting the class `demo\module3\Example` with `demo\module1\Example`.

If you replace the entire class your `demo\module1\Example` class will likely look something like this:

```
<?php namespace demo\module1;
class Example
{
    // empty
```

```
} # class
```

Functional (sort of) but we've just thrown away all the functionality of the previous class. So lets say we don't want to do that.

The first way we can pull functionality from the previous class back in is by extending the other class directly. While we generally refer to classes though the magic `app` namespace, we can also write the full namespace, so writing the following will pull in the previous class into our class:

```
<?php namespace demo\module1;
class Example extends \demo\module3\Example
{
    // empty
} # class
```

This works but is a little inflexible. If say we were making a module and did this then we have just said "our module needs to be the highest priority module that extends the Example class" and "only our module can extend the Example class," more or less (a module with knowledge of our module can circumvent this limitation at the expense of it being unusable outside use with our module so fat chance of that ever happening). So now lets do better:

```
<?php namespace demo\module1;
class Example extends next\Example
{
    // empty
} # class
```

Pay close attention to the syntax, it's NOT `\next\Example` it's `next\Example`, ie. no slash before the special `next` keyword. Now we don't have any of the previous problems. What we've done is tell the system we want to extend the Example class that's next in line in module priority. So let's say we had three modules (module1, module2, module3, in that order) with three copies of the `Example` class (with obviously namespace on changed accordingly, and no `extends` directive in the module3 version), when we access `\app\Example` we would get the class `\demo\module1\Example` (since it's of the highest priority) then have it extend `\demo\module2\Example` since due to `next\Example` resolving to the next in line, then have that itself extend `\demo\module3\Example` for the same reason. If we swaped our module2 with module1 in the modules section of our `etc/environment.php` file we would then get `\demo\module2\Example` extending `\demo\module1\Example` extending `\demo\module3\Example` with out making any file changes.

The last way to extend the class is though partial namespace resolution, this is useful sometimes but generally you'll want to use `next\Class` unless you have a really good reason to be specific. The way partial namespace resolution works is since when extending a class you're generally only interested in the class and not the namespace segments of the class (other then the main one) you can just extend a class with the main segment. So take the class `\mjolnir\access\User` you can write:

```
<?php namespace demo\module1;
class Example extends \mjolnir\access\User
{
    // empty
```

```
} # class
```

Or, you can omit the module namespace segments and just write:

```
<?php namespace demo\module1;
class Example extends \mjolnir\User
{
    // empty
} # class
```

This makes your class a little bit more robust, if User is moved to a different namespace your code will still work, but unless you *really need to be specific* you're better off with the `next\Class` method.

1.1.8 Overwriting Configuration Entries

Unlike classes configuration files don't overwrite each other, but instead merge into each other. Associative arrays will get keys replaced by keys in modules with higher priority, non-associative key arrays will get values combined.

Here is a basic example. Given the following,

```
<?php return array # in module1
    (
        'color' => 'red',
        'people' => [ 'John' => 'Plummer' ],
        'letters' => [ 'a', 'b', 'c' ],
    ); # config
```

And the following:

```
<?php return array # in module2
    (
        'date' => 'today',
        'color' => 'blue',
        'people' => [ 'John' => 'Carpenter', 'Anna' => 'Witch' ],
        'letters' => [ 'd', 'e', 'f' ],
    ); # config
```

When we read the configuration in question we'll get:

```
<?php return array
    (
        'date' => 'today',
        'color' => 'red',
        'people' => [ 'John' => 'Plummer', 'Anna' => 'Witch' ],
        'letters' => [ 'a', 'b', 'c', 'd', 'e', 'f' ],
```

```
    ); # config
```

Higher priority overwrites lower priority.

1.1.9 Overwriting Files

To overwrite files simple place another file with the same name in a higher priority module.

When dealing with vendor files it's usually a good idea to place them in a directory vendor inside the `+App` folder (ie. general files folder), in their own folder then use `\app\CFS::dir` to pull them in.

```
require_once \app\CFS::dir('vendor/the_vendor').'main_class.php';
```

The reason for doing this is so you can overwrite the folder instead of the file though in the case of most vendors dependencies that load everything manually you'll get semi-equivalent results if searching for the file or searching for the folder then append the file like above.

1.2 Introductory Tutorial

In the following section we'll cover creating a basic application. This is the fastest way to get up and running, but if you wish you may skip the section and go into specific sections. If you wish to have a good technical understanding on types used you may skip all the way to the types section which explains all types in the system.

In the following tutorial some of the paths may be changed for easier development; for correctness server optimal structure is illustrated.

For clarity we are going to assume `~/www` points to your server's public directory. We're also going to assume we are creating our project in `~/demo` and the project is called "Demo" and our root namespace for the project is "demo." Replace paths with your own and feel free to replace names with your own as well. We'll use ~ for home directory paths but in cases where the path needs to be absolute we'll assume ~ to be `/home/site_user/`. We are also going to assume development is done on localhost, so the domain in question for our demo is `127.0.0.1` (note: there are complications with using the `localhost` variant in some browsers; the choice here is not just personal preference).

In the tutorial we will cover all commands and details on what's happening. Keep in mind that the time to complete the tutorial (ie. read, copy commands, etc) is not representative of the time it will take you to repeat it on a real project from scratch. We will also illustrate how to perform some basic troubleshooting and cover errors you might encounter which will add several "dead steps" to the process; we find it important you be aware how to not get bogged down, but these also add significant time to the process.

For the sake of brevity we will assume you are familiar with `PHP` and `git` and will only cover what we consider potentially non-intuitive details.

For the purpose of this tutorial you should have the following installed on your development machine: git, PHP (with console access), Ruby (1.9.x generally), Sass (`gem install sass`), Ruby Zip (`gem install rubyzip`), java (used for compiling javascript with google's closure compiler), a server

On windows we recommend using *git bash* for the tutorials, it will give you access to a unix style command line and tools. Recommended servers on windows are Uniform Server, EasyPHP. nginx based stacks are available but we'll be assuming apache servers for simplicity sake.

1.2.1 Basic Structure

```
git clone https://github.com/ibidem/mjolnir-template-app.git ~/demo/0.1.x
cd ~/demo/0.1.x
git checkout mj/2.x/blank
git remote rename origin mjolnir
git remote add origin YOUR_PROJECT_URL.git
git checkout -b development
```

We generally recommend the following branch structure:

- `production` - self explanatory, whatever is in production is "always ready to be pulled in a live version," so avoid direct work on it outside of merges
- `development` - integration branch for unstable features
- `fixes` - very minor changes that don't require special "feature branches" or too much testing; for example: style fixes, typos, single-line fixes, formatting, very minor bugs, etc
- misc feature branches for anything significant; when branches are merged into development remove them

In `fixes` you should only pull changes from production. All feature branches should merge into development for integration (never into production directly). Everything can pull from `fixes` and `production`. Development is merged into production whenever it's current state has been tested.

In the above when cloning `YOUR_PROJECT_URL.git` we recommend using the ssh version of url.

The reason we are keeping the template is to be able to pull tweaks and changes to it in time, eg. changes to project `drafts/` (will be discussed later).

1.2.2 Installing Dependencies

```
bin/vendor/install
```

You can also use `bin/vendor/development`, here's the difference:

- development uses `~/demo/0.1.x/etc/composer.json` and git clones the repositories
- development usually contains things like testing dependencies
- install uses `~/demo/0.1.x/composer.json` and tries to use prepackaged archives
- install is super fast compared to development
- install skips non-production dependencies (note that dependencies have tons of dependencies of their own so its a LOT of stuff)

In production you almost always want `bin/vendor/install`.
You can check the dependencies installed with `bin/vendor/status`.

You can also edit the composer file(s) and run it again to install more.

1.2.3 Private and Public Files

```
cd ~/demo/0.1.x/
cp -R drafts/keys.draft/ ../private/
```

You should now fill in configuration information. Since we're just starting it's only library specific configuration we have to deal with so go into `~/demo/private/config/mjolnir` and review the configuration files there, they should be fairly self explanatory.

Here are some keys to help you fill them faster, and also to give you an idea of how they should look;
only use these in development.

- recapcha testing public key: 6Lfy4d4SAAAAADCqgQpTXxyHVEOFc-ViJP334ZqY
- recapcha testing private key: 6Lfy4d4SAAAAAJXHfni6PmLpOAVFB80B-0eHlGJf
- example cookie key: U0YgC213 ...200 characters... b0XLuyqvxN
- example api key: JKUCI0 ...200 characters... Bn4YsaAO2

That covers the private files.

We now need to copy the public files. For this example we're going to assume we're installing into a folder on our domain, called "demo."

```
cd ~/demo/0.1.x/
cp -R drafts/www/ ~/www/demo/
```

We now need to also copy any public server specific files, in our case since we're using apache we'll need the contents of `www.apache`; unfortunately there is no easy command for this, you'll just need to do it mostly manually.

We'll also need to configure the files in question, in our case of using apache `.htaccess` files we just need to set the `RewriteBase` in the `~/www/demo/.htaccess` file to `/demo/` since we're in a folder (had we been on the root of the site, we wouldn't have had any configuration to do).

All that's left now is to configure the main site settings, located in the `~/www/demo/config.php` in our case.

Note that the file in question is split into (from the top) "Important Settings," "Performance Settings" and "Optional Settings." As you might guess you only need to fill in the "Important Settings" to get up and running. Here is an extract of said settings:

```
# Important Settings
# ----------------------------------------------------------------------
// where are your passwords and secret keys located?
'key.path' => null, # absolute path
// where are the project files located?
'sys.path' => null, # absolute path
// are you in a development environment?
'development' => false,
// what is the domain of your site? eg. www.example.com, example.com
'domain' => 'your.domain.tld',
// is your site in a directory on the server?
'path' => '/', # must end and start with a /
```

Simply follow the comments. Here's how it would look like in our case:

```
# Important Settings
# ----------------------------------------------------------------------
// where are your passwords and secret keys located?
'key.path' => '/home/site_user/demo/private/', # absolute path
// where are the project files located?
'sys.path' => '/home/site_user/demo/0.1.x/', # absolute path
// are you in a development environment?
'development' => false,
// what is the domain of your site? eg. www.example.com, example.com
'domain' => '127.0.0.1',
// is your site in a directory on the server?
'path' => '/demo/', # must end and start with a /
```

We will also need to tell our project of the private files.

```
cd ~/demo/0.1.x/
echo '/home/site_user/www/demo/' > .www.path
```

You can also add a `.key.path` file but the system will read the path from the `/home/site_user/www/config.php` if you have a `.www.path`.

At this point you have a very good base. However the `mj/2.x/blank` is a very minimal branch designed to allow you to easily pull in changes with out manually having to change the files yourself (ie. updated drafts and so on). This means it has no controllers, no themes, etc.

1.2.4 Accessing the Backend

On unix systems by default you have to prefix executable files with `./` due to `$PATH` ordering (local directory `.` is last, instead of first). In development you can change this for convenience or just remember you have to prefix with `./` (ie. `./order`) all the commands bellow.

```
cd ~/demo/0.1.x/
order help
```

This is the list of all commands (aka. tasks), please see on screen help for more information. You can skip the help argument; useful for quick reference. What commands you see depends on your modules (you can create your own tasks).

```
order compile
```

Try opening the site. You should see a 500 error. If you enable development in `~/www/config.php` the system won't hide the error. We are going to continue on assuming development is disabled.

```
order log:short
```

This log maintains a 1 line entry for all errors. While the command is open new errors will pop on screen. You should see the error "Theme Corruption: No themes present in environment file." You can exit with `Ctrl+C`.

```
cp -R themes/empty-theme themes/classic
```

Do **not** use the `mv` command to do this or `git mv`, `empty-theme` needs to remain as-is for pulling updates. You don't have to call it "classic," just remember that's how we're referring to it as.

Open `~/demo/0.1.x/etc/environment.php` and update the `themes` section with `'classic' => $syspath.'themes/classic/',`. The section should look something like this:

```
'themes' => array
    (
        'classic' => $syspath.'themes/classic/',
    ),
```

If you open the site now, you should... still see an error.

If you're observant you'll notice you've been redirected to `127.0.0.1/demo/access/signin`, this is because you have no routes, controllers, etc. What you are viewing is the internal default access page which you were redirected to due to not having access to view anything else. The reason it's not working is because we didn't set up the database, yet. If take a moment to run `order log:short` you should see confirmation of this.

So let's set it up.
We're going to assume you know how to create your database in phpmyadmin, the mysql console or whatever your favorite tool is for doing so. We do not provide automated database creation since that generally involves potentially insecure database configurations. We recommend creating a user specifically for your application and having said user have access only to your applications database.

If you're creating the database and/or user at this time, please review the database settings in the configuration `~/demo/private/config/mjolnir/database.php`.

```
order pdx:reset
```

This will create the database tables and update the schema to the latest version. To see what it did you can view the migration history with `order pdx:history --detailed`.

If you open the site now you should see a form; however by default no administrator accounts are created.

```
order make:user -u admin --role admin -p adminadmin --email admin@example.tld
```

This will create a `admin` user called `admin` with the password `adminadmin`. The `-u` flag stands for `--username` and the `-p` flag stands for `--password`.

You can now go back to the form and log in with the new user `admin`. After doing so please proceed to the "Backend" section (see link on screen).

The backend is the main administration panel; it is generally meant for with server knowhow so applications may have their own custom "admin panels" which their non-technical "administrators" may frequent.

The panels in the main administration panel are customizable; you can easily add more stuff and modules can easily add panels.

At the moment you should be viewing the "System Information" panel which shows the current state of the system by running the `mjolnir/require` configuration in every module. Your current state should be yellow and marked as "Usable." In production you want it to be all green. At the moment the only errors you should be seeing should be "non-dev email driver" and "system email," so we'll skip over fixing them.

Please proceed to "File Permissions." It is mandatory at this point the section be "Stable." On windows you should have an easy time, but on unix system you may have trouble. The point of the section is to check that all file permissions are in order. Every time you pull new changes or update vendors or do something else (eg. compile files) check back with this section, it will ensure no server specific bugs emerged; it's also good idea to check the "System Information" just in case some faulty configuration was pulled in.

Once file permissions are in the green you're good to go. You can check out the "Users" section there but you should see only an admin user at this time.

1.2.5 Routing, Controllers, etc

We'll start by opening `~/etc/config/routes.php`. You should see:

```
<?php return array
    (
        '/'
            => [ 'home.public' ],
    );
```

The "public" in "home.public" specifies what layer stack it uses (it's still part of the name mind you). The "public" stack is one of the default stacks and is essentially: HTTP, Access, HTML, Theme, MVC, in that order. We'll talk about creating your own stack when talking about creating an api. For reference, other default stacks are: log, html, raw, jsend, json, csv, resource (depending on your modules you may have more).

The file you are viewing is part of the "routing system," there is also a "relay system" (the routing system has priority in processing). Both achieve the same function but the relay system is a more advanced (and verbose) version that you should be using when you want to specify routes in modules. All routes in the routing system count as relays. The routing system is just a very space efficient way to write them; since the main application tends to have a lot.

The `'/'` is the pattern that's matched (ie. "the route"), in this case the route is the root of the site, ie. `127.0.0.1/demo/`. Here's some more pattern examples: `'/'`, `'/home'`, `'/people/person/<id>'`, `'/<organization>/employees(/<action>)'`. Words in angled brackets are route parameters, they are processed when the route matches and are available in the controllers. Parentheses specify optional components; a url with out the part in parentheses will still match the route.

The array pointed to by the url pattern is the route's configuration, this consists of in order:

- **the route name**; this is also get resolved to a controller class. If you wish to have the name merely be an alias you can write the route with the following syntax: `'/' => [ [ 'alias-route.public' => 'actual.public' ] ]`, note how the name is an array now instead of a string. The name of the route is mandatory.
- **the route parameters**; can be omitted if you don't have parameters. It's generally in the form of `'/<id>' => [ 'example.public', [ 'id' => '[0-9]+' ] ]`
- **the route methods**; if omitted will be interpreted as `['GET', 'POST']`

For our purposes we are going to change `home.public` to `landing.public`. You should now have this

file:

```
<?php return array
    (
        '/'
            => [ 'landing.public' ],
    );
```

We now need to give access rights to this route. Open `~/demo/0.1.x/etc/config/mjolnir/access.php`. This is the main access control file.

The access system does not work based on ACLs, and will run perfectly fine on it's own with no database access; assuming you don't need users. Due to it not requiring database access you can perform a lot of very complex "can" operations. All that said, you may create ACLs and any other system you desire though the use of Protocol classes; we are mostly going to keep it simple and not have database dependencies so we'll use vanilla protocols (ie. built-in default helpers that come with the library).

In the `whitelist` part of the file define the following rule:

```
Auth::Guest => array
    (
        Allow::relays
            (
                'landing.public'
            )
            ->unrestricted(),
    ),
```

`Auth::Guest` here gives us the guest account (ie. anonymous visitors, or anyone not logged in). `Allow::relays` is a variable parameter function that returns a `Protocol` object allowing entries from a specific relay (we could simply instantiate the object in question, this is just more readable). As mentioned our route `landing.public` also counts as a relay.

The method `unrestricted()` configures the object to ignore context. Normally, `landing.public` with the context of `['action' => 'index']` is not equivalent to `landing.public` with no context, and so we would have to specify every possible value for the action (for clarity we're not going to do this and just allow any parameter since it's perfectly fine in this case).

It's important to know that the access system will DENY until specified otherwise, not the other way round. So, as long as you're specific it's impossible to ALLOW by mistake. Similarly unless you create a protocol in the `Auth::Guest` section (highly unlikely) it's very hard to accidentally give access to anonymous visitors.

You can enable and view `order log:access` to see access resolution as it's happening in case you're dealing with access errors. We won't cover that here though.

You can create your own custom classes. For example, `Protocol_VaultAccess` might only allow access if it's a specific time of day, day of week (work day) and not a holiday. Or a `Protocol_Members_ACL` might

call on the database for information to determine the users access (ACLs are only "better" when you need to have options for customizing access inside the application; every other case you're better off with no dependency on the database). Since it's programmatic you can also do things like grant access to an user based on the user's relationship to someone else. So if X is in a Division with Y then X should have access to the project Y is currently working on; this is simpler and more intuitive then granting and removing privileges to X for the project which would require checks and operations when project is created, assigned, Y is assigned/unassigned to a project, X is assigned/unassigned to Y and all sorts of other relationship concerns.

Before continuing please open another console instance and run the following command:

```
clear ; order log:short --erase
```

We'll refer to this from now on as the "error console." It's a good idea to have it open at all times in development, any errors that happen in the background will appear here even if while navigating the application you may not notice them; note that some errors can be hidden by html markup such as an error that occurs inside a tag attribute value, so even the page you see may have errors in it.

The `--erase` (short form: `-e`) parameter tells the `log:short` task to throw away the previous log (this avoids confusion).

Now open `127.0.0.1/demo/` again.

In the error log window you should see an entry with `Class '\app\Controller_Landing' not found`. The error is telling us our route works and our access rule works but we don't have a controller yet.

If you see the backend, you're still logged in with the admin account (user roles can have custom "dashboard" pages to which users are redirected to; the admin panel is the default for the admin role), please sign out and open the specified link again.

We'll first need to create a module for our controller.

```
order make:module --name core --namespace 'demo\core'
```

In the file `~/demo/0.1.x/etc/environment.php` add the new module you've just created to the modules section at the very top. The line you need to write will be specified by the command above upon successful creation, and will look like this:

```
$modpath.'core' => 'demo\core',
```

We can now create the class.

```
order make:class -c 'demo\core\Controller_Landing'
```

You can very well just create the class by hand but this tends to be better since it checks for the namespace, fills in comments, updates honeypot files, may fill in the class with placeholders, adds a `@todo` comment, etc.

We're going to create a very standard and easy looking controller, you do have the option to make the controller

anything you need.

Open the new file `~/demo/0.1.x/modules/core/Controller/Landing.php`. If you're confused on the path please review the Cascading File System section of the documentation for the very basics of the file system's inner workings.

You should now change the extended class to `\app\Controller_Base`. `Controller_Base` is a shorthand; it's essentially extending `Puppet`, implementing the `Controller` interface and using the `Controller` trait, which more or less in plain english means it's a special generic controller that has all the traits of a controller but also happens to have a name and allow for operations based on it's name (basically you can call functions such as `codename` or `codegroup` and others, to do meta programming inside it).

You class should should look like this:

```
class Controller_Landing extends \app\Controller_Base
```

We will now create the main trait,

```
order make:trait -t 'demo\core\Trait_Controller_DemoCommon'
```

In the lifetime of a application we'll be creating more then one controller, it's useful to have at least one common trait so we can share functionality between them; we don't use a base class since we want to have the option of extending different types of controllers as well. Complex class hierarchies are also harder to maintain then trait hierarchies.

Inside the body of the `Controller_Landing` class add the following declaration:

```
use \app\Trait_Controller_DemoCommon;
```

Now any methods we add in the trait will be injected in the class, so long as we don't create a method with the same name in the class, in which case the trait method will be overwritten by the class method.

Now open the new trait file:
`~/demo/0.1.x/modules/core/Trait/Controller/DemoCommon.php`

A common requirement of many controllers is the index action, it's typically the same functional code or if it's more complex typically it delegates to some other classes to resolve so we can create a default one in our trait.

We explained earlier how in `'landing.public'` the public is the name of the stack we execute. The convention with routes (relays can do whatever) is that the name of the stack used is also the prefix of the action (implied underscore). Another convention is that unless specified as a parameter in the url (ie. if the route is `/something(/<action>)` then `/something/test` has the action `test`) the default action (be it if there is a `<action>` segment or not) is always "index." So given we don't specify even an `<action>` segment the action is always `public_index`. If we weren't using the "public" stack and instead using say a custom "api" stack the action would be instead `api_index`.

With regard to controllers and data flow, the controller is expected to return a value that can be interpreted by

the layers in it's stack, in the case of the "public" stack that value must be either a `string` or a object implementing the `\mjolnir\types\Renderable` interface (see types section). An api stack on the other hand might require you always return a PHP `array`.

Please add the following `public_index` method:

```
/**
 * @return \mjolnir\types\Renderable|string
 */
function public_index()
{
    return 'hello, world';
}
```

If you now open `127.0.0.1/demo/` you should see "hello, world." If you don't please check your error console, you may have a typo or some other error.

Above is the basic example, let's do something more complex; again remember you can always have your own way of doing things, there are only some basic interface requirements (which if you wish you can get rid of by not using the "public" stack). Replace the above `public_index` method with the following:

```
/**
 * @return \mjolnir\types\Renderable|string
 */
function public_index()
{
    $this->channel()->set('title', 'Demo');
    return \app\ThemeView::fortarget(static::dashsingular(), \app\Theme::instance())
        ->pass('control', $this)
        ->pass('context', $this);
}
```

Back in the `Controller_Landing` class add `static $grammar = [ 'landing' ];`, this will allow for the little bit of meta programming that's happening with the `static::dashsingular()` method above. Your `Controller_Landing` class should look something like this:

```
class Controller_Landing extends \app\Controller_Base
{
    use \app\Trait_Controller_DemoCommon;
    static $grammar = [ 'landing' ];
} # class
```

If you now open the site you should see in your error log a message containing the following: `Theme Corruption: undefined target`. This error is telling you that the "theme target" you tried to access (ie. "landing") doesn't exist. In our case the target in the theme corresponds to the controller name because we chose to have it this way, but in general you can have the theme targets be whatever. This means that using a basic mockup controller you can mockup an entire site and work on the style and layout independent of the site's functional code.
We'll only cover basics to working with themes.

First, go to the theme configuration file: `~/demo/0.1.x/themes/classic/+theme.php`

Now add the following in the "mapping" section of the file:

```
    'landing' => [ 'landing' ],
```

This tells the theme system that you want to resolve the target "landing" using the given array of files; in our case just the one "landing" file. The file paths are resolved from the root of the theme and if we had provided multiple files they would have been placed one in another.

Now create the file `~/demo/0.1.x/themes/classic/landing.php`

Based on how we've written our `public_index` method we now have access to two variables `$control` and `$context`. `$context` is in general the visible user recognizable data on the page, while `$control` is the meta-data on the page (and almost always heavily tied into the controller, hence the name). Things like the action of a form, the state of the page (editable, non-editable) or some other details (what kind of page it is, theme options, etc) generally fall in the category of page meta-data and will be accessed via `$control`. In our case both $context and $control point to the same object, a `Controller_Landing` instance; sometimes it's useful to split contexts outside of the scope of the controller so you can have multiple contexts compose into a single context that you feed to the page.

We are not going to bother with `$control` and `$context` for now, please just write "hello, theme" inside the file.

If you now re-open the site you should see "hello, theme." If you don't, as before, please check your error console for typos and other errors.

One thing you probably have not noticed is that the entire output has been wrapped in the correct html (even when you just returned "hello, world" from the controller earlier), this is due to the html layer. If you do not have very good understanding of the theme system you are advised to avoid using a stack with out the html layer or a stack with out a compatible drop-in-replacement of the html layer, since it does a lot more then just wrap your content in correct html meta.

We will now continue with this basic static site example by adding a method to display the "hello world" message.

Add the following method in your `Controller_Landing` class:

```
/**
 * @return string
 */
function say_hello()
{
    return 'hi!';
}
```

Replace the contents of your `~/demo/0.1.x/themes/classic/landing.php` file with `<?= $context->say_hello() ?>`.

If you open the site you should now see "hi!"

It's good to use the correct variable when accessing method even though they are on the same class since they might not always be on the same class; `say_hello` is not metadata so we use the `$context` variable.

Continuing on with the discussion on themes, themes may have a variety of different modules: style module, scripts module, etc. You can define your own if you want; if you don't like how the style module supports multiple styles and requires sass you can just make your own custom style module that just works with vanilla css.

Which modules are enabled for the theme is determined by the loaders section in the theme `+theme.php` configuration file, but we won't cover that here.

All modules in general will work with the same target you provide to the theme. Where the theme resolves it to a page composition, the module resolves it to it's own composition. In the case of scripts for example the target specifies which scripts appear on the page (assuming we don't specify we want all scripts on all pages).

We will add basic script to the page to illustrate.

We start by executing a monitoring script that will compile the javascript to single file. Please run: `~/demo/0.1.x/themes/classic/+scripts/+start.rb` this will open a console; while the console is open files in the scripts directory will be monitored for changes and compilation done automatically.

Now open the main configuration file `+scripts.php` in the same folder as the monitoring script. As with the theme configuration we specify the rules for the target in the `mapping` section; or to be exact `targetted-mapping` since we want per page customization. Currently there should be a rule there `frontend` already defined, since we're not using it, please rename `frontend` to `landing` and add `hello` to the existing list. Your `mapping` section should look like this when done:

```
'targeted-mapping' => array
    (
        'landing' => array
            (
                'base',
                'hello'
```

```
            ),
    ),
```

Now we need to create the file `hello`. If you look at the configuration you'll notice that the `sources` is set to `src` so our `hello.js` file would be located at `~/demo/0.1.x/themes/classic/+scripts/src/hello.js`.

In the file just add an `alert('hello, world');`. If you now open the site you should see an alert with "hello, world." You may have to Ctrl+R, alternatively you can simply update the script version in the configuration file with the mapping. If your browser has support for source maps you should be able to find `src/hello.js` in your inspect menus.

1.2.6 Creating an API

This section follows immediately after the code you obtain from the previous sections and focuses on creating a REST API.

For this section it's highly recommended you install the Postman REST client extention. You may install another so long as you can follow along. The extention is not mandatory for completing this section but using it (or a similar technique) is highly recommended.

The reason you need the extention is for testing; it's quite pointless to test your api while running the application. Worry about your api working with an extention like Postman that lets you run POST, GET, DELETE, PUT, etc requests with json payloads against your server and once you know it's all working worry about the frontend application you create for your users consuming it; who you create first, the backbone collection or the api is up to your personal preference, just remember that you should not work on both at the same time since, while by no means impossible, it's harder and more time consuming even for small issues.

There are other reason too, such as redirects; if you test your api in the browser the internals will try to redirect you to an appropriate error page in case of an error; while helpful in a live application it's very disruptive in development (you can also enable development mode to get rid of this functionality, go to your `~/www/demo/config.php`).

At this point you may choose to replace your `landing.php` with the following to help you build a Backbone application.

```
<?
    namespace app;
    /* @var $theme ThemeView */
    $templates = array
        (
        // pages
//          'Dashboard' => 'pages/Dashboard',
        // modules
```

```
            // no module templates
        // extentions
            // no 3rd party extentions
        );
?>
<div id="sheep-context">
    <div class="container">
        <h1>Loading...</h1>
    </div>
</div>
<? foreach ($templates as $template => $path): ?>
    <script type="text/x-underscore-template" id="<?= $template ?>-template">
        <?= $theme->partial("templates/$path")->render() ?>
    </script>
<? endforeach; ?>
```

The new `landing.php` will solve most of your issues with templates. You just put all your templates in a template folder and your application will dump them (based on the template configuration you specify above) into the `landing.php` page which in turn is your main application entry point. This is why we changed the name to "landing" in the early steps; home and frontend are generally more appropriate for static pages.

We're not going to make use of it in this section since we'll be focusing on creating the api, that should be roughly all you need to be able to start using almost any backbone beginner tutorial.

Getting back to the API itself, please replace your `~/demo/0.1.x/etc/config/routes.php` with the following:

```
<?php
    $id_regex = '[0-9]+';
    $id = ['id' => $id_regex];
    $apimethods = ['GET', 'POST', 'PUT', 'PATCH', 'DELETE'];
return array
    (
    // ---- API ---------------------------------------------------------------
        // clients
        '/api/v1/client(/<id>)'
            => [ 'v1-client.api', $id, $apimethods],
        '/api/v1/clients'
            => [ 'v1-clients.api', [], $apimethods],
    // ---- Pages -------------------------------------------------------------
        '/'
            => [ 'landing.public' ],
    );
```

There's not too much of a difference, we just added 2 api routes. The 2nd and 3rd parameters are optional and when specified must be arrays, the first one specifies what regex the segments need to match to be valid (by default it's `[]` as in no parameters which defaults the parameters to matching everything) and the 3rd one specifies what methods are allowed (by default `['GET', 'POST']` if not specified).
We use `$id` as an array since PHP allows `+` on arrays and it acts as a merge. So if we had `$name = ['name' => '[a-z]+']` then `$id + $name` would be equivalent to `['id' => '[0-9]+'] + ['name' => '[a-z]+']` and produce `['id' => '[0-9]+', 'name' => '[a-z]+']`. This is merely used for clarity, `$id + $name` is very short and to the point.

Our two new routes use the `api` stack. By default the framework does not come with an `api` stack to avoid modules polluting logic by making assumptions on how the `api` stack works. For that reason the `api` stack is considered application reserved and you have to define it yourself. To do so create the file `~/demo/0.1.x/modules/core/+App/config/layer-stacks.php` with the following code:

```
<?php return array
    (
        'api' => function ($relay, $target)
            {
                $json = \app\CFS::config('mjolnir/layer-stacks')['json'];
                return $json($relay, $target);
            },
    ); # config
```

You now have defined the stack. We cheated and just have it call the `json` stack, but that's perfectly fine implementation for our use case.

In Postman access `127.0.0.1/demo/api/v1/clients`. At this point you should see an error `{ "error": "URL called is not a recognized API." }` with the 404 status. This is because we haven't given access rights to our api.

To give access rights, in `~/demo/0.1.x/etc/config/mjolnir/access.php` replace your previous `Auth::Guest` rule with the following version:

```
    Auth::Guest => array
        (
            Allow::relays
                (
                    'landing.public'
                )
                ->unrestricted(),
        // API, v1
            Allow::relays
                (
                    'v1-client.api',
                    'v1-clients.api'
                )
                ->unrestricted(),
        ),
```

We'll also need to create the appropriate classes to handle the request.

```
order make:module --name api.v1 -n 'demo\api\v1'
```

Don't forget to enable it in your `~/demo/0.1.x/etc/environment.php`.

```
order make:class -c '\demo\api\v1\Controller_V1Clients'
```

Open the new class, located in `~/demo/0.1.x/modules/api.v1/Controller/V1Clients.php` and change the extended class to `\app\Controller_Base_V1Api`.

In Postman now access `127.0.0.1/demo/api/v1/clients`, you should see the error "Not Implemented" with a 501 status. The error is generated by the placeholder GET handler provided by `Controller_Base_V1Api`.

Now that we got the groundwork done for `127.0.0.1/demo/api/v1/clients` it's time to get the internals sorted. There are several ways to do this, one way is to use `Marionette` models which are models specifically designed to help in creating APIs. More precisely they follow Backbone's flavor of APIs and to help managing them they also mirror backbone's conventions, so you have a class for the collection and a class for the model and all methods you call on are equivalent to the http methods you would call. So calling `$model->put($conf)` will update the entry and return an updated entry, calling `$model->patch($conf)` will partially update the entry and return the updated entry, calling `$collection->get($conf)` will return all the entries, and so on.

The other type available by default (you're free to define your own system that works for you) is a static library model, where we simply define a class with static methods and use traits to inherit functionality; this is very flexible but less automated—in practice this means we can get complicated jobs done easier and more intuitively when using the static library model method (because you have a lot of control) but can get a lot of simple jobs done quicker when using the `Marionette` method (because you have a lot of automation). We'll talk about the static library models later, for clients we'll show how to define a `Marionette` model system. We are going to continue with Marionette models. We're going to assume a client is defined by merely a "family_name" and "given_name".

```
order make:class -c 'demo\core\ClientCollection'
order make:class -c 'demo\core\ClientModel'
```

In `~/demo/0.1.x/models/core/ClientCollection.php` change the extended class to `\app\MarionetteCollection`. You can remove the `@todo`, the class will work as-is based on it's name only.

In `~/demo/0.1.x/models/core/ClientModel.php` change the extended class to `\app\MarionetteModel`.

With the main classes created we now need to create the configuration file that goes with them. Create the file `~/demo/0.1.x/modules/core/+App/config/client.php` with the following content:

```
<?php return array
    (
        'name' => 'client',
        'key' => 'id',
        'fields' => array
            (
                'id' => 'number',
                'given_name' => 'string',
                'family_name' => 'string',
            ),
    ); # config
```

Finally we need to a database table. For this we'll create a paradox migration. Paradox migrations work roughly like this:

1. you have channels, usually each module has it's own channel but sometimes a set of modules may have their own; for our application we'll create a "demo" channel
2. each channel has it's own version history, so 1.0.0 for the demo channel is different then 1.0.0 of the mjolnir-access channel
3.

the system keeps the history in the database; you can view it on the command line by using pdx:history

4. the system also works by default in lock mode so it won't allow destructive operations such as uninstalling the database

5. each migration is a entry in the mjolnir/paradox configuration file, which in our case would be located in `~/demo/0.1.x/modules/demo/core/+App/config/mjolnir/paradox.php`

6. when you run pdx:upgrade the system will look for changes and run any previously not executed migrations

7. migrations only go forward so when testing and or moving between branches with different database structures in development you'll need to generally turn of database locking so you can perform database resets (ie. uninstall -> reset to latest, or specific version + upgrade if testing migrations)

8. migration operations NEVER call on anything but basic low level database operations with the only exception being the "table" static method in modules which provides the table name. The reason for this is that anything above low level apis is dependent on the state of the database and hence dependent on both certain things existing and certain things existing in a particular state, both of which are not guaranteed when the migrations are running. Even a basic count method can potentially reference a certain security field when performing the count which might only be available from a certain migration onward.

So lets start by creating the paradox file, for clarity we'll be relocating the actual migration part to a separate configuration file (this is recommended since migrations are many and can get quite long).

Create the file `~/demo/0.1.x/modules/demo/core/+App/config/mjolnir/paradox.php` with the following contents:

```
<?php return array
    (
        'demo' => array
            (
                'database' => 'default',
                // versions
                '1.0.0' => \app\Pdx::gate('demo/1.0.0'),
            ),
    ); # config
```

`\app\Pdx` is the main "Paradox" library class, provides access to helpers (and the operations themselves if you need to call them in code). The call to `\app\Pdx::gate('demo/1.0.0')` will basically add `timeline/` to the key and load it as a configuration file, returning it as an array.
Create the file `~/demo/0.1.x/modules/core/+App/config/mjolnir/timeline/demo/1.0.0.php` with the following contents:

```
<?php return array
    (
        'description'
            => 'Install for Clients.',
        'configure' => array
            (
                'tables' => array
                    (
                        \app\ClientModel::table(),
                    ),
            ),
        'tables' => array
            (
                \app\ClientModel::table() =>
                    '
                        id          :key_primary,
                        given_name  :name,
                        family_name :name,
                        PRIMARY KEY(id)
                    ',
            ),
    ); # config
```

The description will be dumped into the history when the migration runs. The `configure` key is for providing meta information to the migration system, in this case we're telling it what tables it should be aware of with this migration (this informations is used for things like uninstalling). The other `tables` key is telling it which tables we want to create; we're using placeholders for easy customization of the installation (they're also easier to read).

With a lot of configuration options you can just throw in a function and do whatever you want, but this requires some knowledge on the internals so we'll leave it as-is.

In this example we also just used one channel, but you can define as many channels as you need at the application level. You can also set dependencies between versions. Version `1.1.0` of `demo` might depend on version `1.0.1` of `mjolnir-access`. This is done by defining a `require` key with an `array` of dependencies. Generally it's recommended you place the array dependencies in the paradox file which when using `Pdx::gate` you would do by passing the array as the second argument.

With our migration in place we now only need to upgrade our database:

```
order cleanup
order pdx:upgrade --dry-run
```

We use `cleanup` in case the system cached the previous configuration state. The `pdx:upgrade --dry-run` will show you the steps it will do but not actually do them. You should see 1 line that reads "1.0.0 demo" as in "run the demo 1.0.0 migration."

```
order pdx:upgrade
```

You should get "Upgrade complete."

```
order pdx:history
```

You should see your migration as the last one executed. You can use `--detailed` to get the description too.

Now that we have the database and model we can return to our api.

Add the following method to `~/demo/0.1.x/modules/api.v1/Controller/V1Clients.php`

```
/**
 * @return array
 */
function get($req)
{
    $collection = \app\ClientCollection::instance();
    $conf = [];
    ! isset($req['limit']) or $conf['limit'] = $req['limit'];
    ! isset($req['offset']) or $conf['offset'] = $req['offset'];
    return $collection->get($conf);
}
```

In Postman access `127.0.0.1/demo/api/v1/clients`, you should see "200 OK" on the status and `[]` on the returned value.

With the Collection part done, we'll now create the Model part to get some items in. We've already got the database and model class setup from before so we just need to create the api.

```
order make:class -c '\demo\api\v1\Controller_V1Client'
```

The file created is located in `~/demo/0.1.x/modules/api.v1/Controller/V1Client.php`. Please change the extended class to `\app\Controller_Base_V1Api` and add the following methods:

```
/**
 * @return array
 */
function get($req)
{
    $id = $this->channel()->get('relaynode')->get('id');
    $model = \app\ClientModel::instance();
```

```
        $entry = $model->get($id);
        if ($entry == null)
        {
            $this->channel()->set('http:status', '404 Not Found');
            return [ 'error' => 'Client with id ['.$id.'] does not exist.' ];
        }
        return $entry;
    }
    /**
     * @return array
     */
    function post($req)
    {
        $collection = \app\ClientCollection::instance();
        $entry = $collection->post($req);
        return $entry;
    }
    /**
     * ...
     */
    function delete($req)
    {
        $id = $this->channel()->get('relaynode')->get('id');
        $model = \app\ClientModel::instance();
        $model->delete($id);
        return null;
    }
    /**
     * @return array
     */
    function patch($req)
    {
        $id = $this->channel()->get('relaynode')->get('id');
        $model = \app\ClientModel::instance();
        return $model->patch($id, $req);
    }
```

A few things to explain before moving on. The `$this->channel()` call refers to the channel for communication used by the current request. The channel object is shared between layers, the controller and whatever else participates in the request, and its purpose is to allow for isolation of metadata specific to the request. In this case we're using it to get the `relaynode` (ie. the route object, since routes are relays) and from the relay node we retrieve the id parameter in our route. In the get method we're also communicating with the channel how the `http:status` should change to `404 Not Found` for the case where the entry does not exist.

You might be confused by why we're using `ClientCollection` to perform the post operation. This is a slight deviation to the way Backbone works for correctness; the correct way to create a new entry in a model is to post to the collection, but backbone posts to the root of the model, hence why we're calling the collection

there (the correct way to do it) in the client api (the api backbone expects to be able to post new entries to). Just to be clear, the very obscure functionality related to doing a POST against a model is not supported.

If you wish you may also change the routes like so:

```
'/api/v1/clients/<id>'
    => [ 'v1-client.api', $id, $apimethods],
'/api/v1/clients'
    => [ 'v1-clients.api', [], $apimethods],
```

ie. add an "s" to the v1-client.api route and make the id mandatory.

Now you can have the post method into the `Controller_V1Clients` class instead of `Controller_V1Client`.

It is indeed more correct; we've chosen to explain it as we did in case you wanted to have separate urls, and because it's less confusing on how the url work with respect to what the models in backbone are calling. With the above code what will happen is backbone will try to call the root of the client url and the request won't match but will match the collection url since it's equivalent to the root.

In Postman, set the method to POST, select payload as raw (JSON) and run the following:

```
{
    "family_name": "Joe",
    "given_name": "Average"
}
```

You should get back a json with the fields and an id. Run it 4 more times; feel free to change the name if you wish, but since we didn't add in validation you can run it as-is.

Here are some basic operations:

1. in Postman, do a GET request on `127.0.0.1/demo/api/v1/client/2`, you should get the entry with id 2.
2. in Postman, change the GET to a DELETE and run the request
3. in Postman, change back to GET and run the request, you should get a 404 status this time with an error message
4. in Postman, do a GET on `127.0.0.1/demo/api/v1/clients`, if you been following along you should get entries with IDs 1, 3, 4, 5 since we DELETEd entry with ID 2 earlier.
5.

in Postman, enable URL params and add "limit" with the value "2", you should now only see entries 1 and 3

6. in Postman, add the url parameter "offset" with value "1", you should now see entries 3 and 4

We won't cover how to actually write the backbone code since at this point there's no different between writing it in your application or a plain html file (we showed how to setup javascript earlier; which is the main component you need).

Finally we're also going to show how to use a static model library. Generally when you have an application that only uses static model libraries to function you would go with the naming convention `Model_Client` which will place your class inside a `Model` directory. When working with `Marionette` classes the naming convention is `ClientLib` since that places it next to the `Model` and `Collection` class, which is a lot easier to manage. Obviously we're going to go with the `ClientLib` variant.

```
order make:class -c '\demo\core\ClientLib'
```

Remove the extends declaration and add the following traits to the class body:

```
use \app\Trait_Model_Factory;
use \app\Trait_Model_Utilities;
use \app\Trait_Model_Collection;
```

We'll also need to resolve the table name. Normally we would add a static field `$table` with the name but since we are using marionettes and have a configuration file setup we'll overwrite the table() method to retrieve the correct value so everything is in one place.

Now add the following method so the the model knows which table to use:

```
/**
 * @return string table name
 */
static function table()
{
    return \app\ClientModel::table();
}
```

To show that it all works we'll modify the get method for the collection, since other methods work differently then what backbone expects.

Replace the method "get" in `Controller_V1Clients` with the following version:

```
/**
 * @return array
 */
function get($req)
{
    $limit = isset($req['limit']) ? $req['limit'] : null;
```

```
        $offset = isset($req['offset']) ? $req['offset'] : 0;
        return \app\ClientLib::entries(1, $limit, $offset);
    }
```

In Postman, do a GET on `127.0.0.1/demo/api/v1/clients` with limit 2 and offset 1. You should get 3 and 4 like before.

We won't go into exact examples on why you would use one or the other but to give you an idea, lets say you needed to have a very special relationship and a very specific data type. In the marionette model you have to (a) find a way to interpret it though GET, POST, PUT etc, (b) write a driver to handle all the operations in a dynamic way (which isn't as easy as it sounds), and (c) use the driver in your configuration. On the other hand in the static library method you just boil down the problem to raw SQL and place it in whatever method you want; you can just create your own method, since the model is designed to act as a library, not follow an interface, and all methods are independent (a given since they are static). Writing raw SQL solves problems really really fast. So, as mentioned earlier, the question boilds down to: do you want control, or do you want automation. Mind you both have mechanisms for dealing with repetition, drivers for the marionette system and native traits for the static library model system.

Also, if you ever need a special static method to perform an operation, the correct way is to create the `ClientLib` class equivalent, since the `Model` and `Collection` classes are specifically designed to just consume drivers and should not be forced to do anything more. There are other reasons too, the Lib class has access to static helpers, the Lib class works with static methods, whereas the others require instantiation and so you may require instantiation of the class you're into to call methods you need, the Lib class is also a clearer place to have the methods then having them split over two classes, etc.

But again, as mentioned earlier, you can simply create your own model system to suit your own needs, these are just the defaults provided.

1.2.7 Upgrading

As the previous sections this is more server oriented for clarity.

It's assumed we're upgrading the instance created in the previous part.

```
cd ~/demo/
cp -R 0.1.x/ 0.2.x/
```

We `cp` (ie. "copy") instead of `git clone` to preserve any file permissions, special files like `.www.path`, logs, etc. It's also much faster in some cases since we don't have to connect to the internet to check for updates and such.

Update `~/www/config.php` paths, namely "sys.path", enable maintenance mode. The good thing about

keeping our private keys and such in a seperate "private" folder is that we now don't have to worry about it. Since we `cp`'ed the directory, if we don't do any database upgrade we can change back to the old source tree by reverting `sys.path` to the `0.1.x` version.

```
git pull origin production
bin/vendor/install
order compile
```

If you are using packaged mode you can skip the `compile` step.

```
order pdx:upgrade --dry-run
```

Check that everything is as expected. Then run it:

```
order pdx:upgrade
```

At this point check your admin panel that everything is in the green.

When everything is in order disable maintenance mode in `~/www/config.php`, and you're done.

Keep in mind projects may have extra dependencies that require extra configuration to be performed.

1.3 Types

To allow for easier use of classes a lot of common functionality has been centralized into a set of interfaces.

This is a multifaceted feature. However, to avoid confusion the interfaces are designed for work within the library, and while they may be used outside the library as you please, they are not designed with that in mind. Essentially if it exists, it's because it has a purpose within the library, not because it's some sort of standard.

Some functionality interfaces establish within the library,

1. simplified interaction with classes sharing common themes, such as file manipulation, document-like content, etc
2. easier to understand and use patterns; employing interfaces creates repeatable easy to pick up patterns
3. easy integration, we try to avoid classes accepting implementation extending some base class; instead of just implementing the interface is fine
4. facilitate adapter patterns

(This is not an exhaustive list)

Fine details will be treated in individual sections, however one common point is getters, commands, and setters.

In the mjolnir interfaces a getter is always a function with the name of the property:

```
$content = $document->body(); # "content equals document body"
$writer->eolstring();
```

If useful you can return sub types derived from the main type of the property. You should do this by appending a keyword after the property name, but avoiding and underscore.

```
$view->file();        # relative path
$view->filepath();    # absolute path
$view->filehandler(); # file handler
$view->fileobject();  # file as an object
```

It can be as crazy as you like, for example:

```
$view->fileurl();
```

A setter on the other hand is a function that is never just the name of the property. In mjolnir all setters are composed of the name of the property (always the first part) followed by either the generic term, such as "is", a type or some other descriptive word, concatenated together with an underscore. So lets say we have a class representing a html tag, setters for the class property might look as follows.

```
$tag->class_is('btn'); # "tag class is btn"
$tag->class_string('btn btn-primary');
$tag->class_array(['btn', 'btn-primary']);
$tag->class_from($other_tag);
```

This patterns keeps the getters and setters close togheter and allows for a lot of setters with very intuitive syntax.

The third category is commands. So for example, continuing from the example above the following are comamnds:

```
$tag->appendclass('btn-primary');
$tag->removeclass('btn');
$tag->standard('twitter');
```

The variant `$tag->class_append('btn-primary')` is discouraged because it blurs the line between what's a variant on a setter and what's a manipulation function. Functions, such appending a class, aren't looked any differently then any other non-setter or non-getter function. When it comes to naming, active wording is preferred, but not required.

To clarify, as a rule of thumb if you have a function that only partially sets a property, it's probably a manipulation function and not a setter. A setter should (typically) set the whole value.

A setter doesn't imply a getter, and neither does a getter imply a setter. Both can exist with out the other, and of course you can just have internal properties or states that are simply manipulated but never explicitly gotten or set on their own.

Sometimes the rules may be broken in favor of using very established terminology.

1.3.1 Type Traits

All types have a corresponding trait using the naming convention of prefixing the interface name with `Trait_`.

Within the library one reason why traits are used is to keep the codebase dry by moving a lot of boilerplate code (such as getters, aliases, magic method) to the corresponding trait. So for example the trait for the `HTMLForm` defines all the field shorthand methods (ie. `text`, `password`, etc) which are all just fancy aliases for the `field` method.

Another reason is to manage trait bloat. Essentially one problem that happens when you use a fair amount of traits is that you start to have long hard to follow trait hierarchies, so for example a `Task` is a `Executable`, `Meta`, `Writable`. This means that every class that implements `Task` needs to use the corresponding traits for the `Executable`, `Meta` and `Writable` interface. If the class is just implementing `Task` the trait declarations might not be too unintuitive but as you add more interfaces to the class they become unwieldy. To combat this, all traits of a interface are responsible for managing the traits of the extended interfaces. This makes for easy to follow trait declarations. **There is one exception.** If a interface is intended to be used by a class which is a child class of another class, the trait will not borrow the implementation from the super interface since the class by extending the class will automatically get the implementation anyway. For example, the trait for the `HTMLFormField_Boolean` interface (used by radio and checkbox fields) does not touch the trait for the `HTMLFormField` interface. Or, the trait for the `HTMLFormField` interface does not touch the trait for the `HTMLTag` interface.

Finally the main reason traits are extensively (and to some extend why interfaces are so extensively used) is to allow for high level of code injection though out the library. For example the `Meta` interface manages most metadata related tasks; if you ever want to do some specific operation or have some specific shorthand you need only extend the trait in your application modules, or some specific plugin, add the functionality and it will be inherited by all the classes within the library that use the `Meta` interface (which in this case would be a lot!).

Here is an example of how you would go about extending the trait:

```
trait Trait_Meta
{
    use next\Trait_Meta;
    // your extra features
} # trait
```

In the example `next\Trait_Meta` will be resolved to the closest trait in the module hirarchy so this code will transparently create a chain of traits extending other traits for functionality (finishing up with `\mjolnit\types\Trait_Meta`). For more information on the special namespace segment `next` see the Cascading File System section.

1.3.2 Generic Types

The following types are core types used though out the type system as base types for other types:

- Meta
- Renderable
- Executable
- Standardized
- Filebased
- Processed
- Channeled
- Paged

Some of the more specialized generic interfaces

- Writable
- Savable
- Recoverable
- Matcher
- Linkable
- Contextual
- Eloquent
- Exportable

`Meta` interface

The `Meta` interface is one of the most extensively used interfaces within the library since they simplify property management. Using a class implementing the interface is quite easy:

```
// you set a property via the set method
$object->set('my_property', 'my_value');
// you retrieve the property via the get method
$object->get('my_property');
// you can also provide a default when getting a value
// if not specified the default value is null
$object->get('my_property', 'default_value');
// when a property is an array you use add instead of set, though you can
// still use set to replace the entire contents or empty the array
$object->add('my_property', 'my_value');
// sometimes you want to just get the entire metadata
$object1->metadata();
```

```
// ...typically you want an objects metadata to pass it to another
$object2->metadata_is($object1->metadata());
```

Generally you would want to use `Meta` when you have a class that would otherwise be bloated with a ton of properties. You don't need to sacrifice usability when using meta since if you think a user might find a property hard to remember you can simply create a magic method that just calls the corresponding meta method.

Typically you'll also get cleaner class code by working with one `metadata` attribute, compared to working with 10+ attributes.

`Renderable` interface

A renderable object is one that can be translated to string. Typically the whole point of the object is to eventually get translated to a string.

The interface has one main method `render` which performs the translation to string (it's use is self explanatory). The interface also defines several utility methods: `addmetarenderer`, `metarenderer` and `injectmetarenderers` which are used for objects which need help in rendering subparts of themselves. For most renderable objects though, these will do nothing since they don't have meta renderable parts to them.

For correctness while in development mode the `__toString` method of a `Renderable` object is defined to throw an error (it will simply attempt to call `render` in production). This is the default because a lot of the time rendering involves injecting outside data and is not merely self contained to the object, which may easily lead to errors and unfortunately PHP's magic `__toString` method does not allow for errors to happen. In fact the best result you can get when errors occur in the `__toString` method is to just return `null`, which will also constitutes an error, but is at least semi-recoverable.

Occasionally implementations may have good reason to use `__toString` in which case they would overwrite the behaviour defined by `Trait_Renderable` and allow `__toString` to call `render` in both production and development, but still return `null` on errors. The reasoning here is that these are systems where an error is very unlikely, rare, or easily recoverable within the context of the class.

`Executable` interface

An executable is anything that can `run` (ie. execute).

There's nothing more to say about this interface, it's very much self explanatory.

`Standardized` interface

A standard is defined as a specific configuration on an object bearing a certian name. So for example if we consider a form, the way twitter bootstrap defines we should create the markup for said form is a essentially a standard. We can create a `twitter` standard (in this case in the `mjolnir/htmlform` configuration file) and we would use it on a `HTMLForm` like so:

```
$form->apply('twitter');
```

Now presumably the form would output input fields with twitter bootstrap markup; assuming we've defined it correctly.

Standards are useful because they provide an easy way to deal with boilerplate configuration. Simply define the configuration in one place as a standard and use it anywhere. They also help make a lot of configuration DRY with out requiring the class itself from housing a self-configuration method for each; instead you would intuitively place them in a configuration file.

`Filebased` interface

There are many classes dealing with files. This interface standardizes the way you communicate to such classes about the file they are working with.

```
// specify a file based on a relative path determined by the rules and
// conventions of the object in question
$object->file_is('my_file');
// specify via an absolute path
$object->file_path('/path/to/my_file');
// get file path
$object->filepath();
```

`Processed` interface

A `Processed` object is an object that has either processing before or after (or both) it's execution or some other important event in it's life cycle.

The interface provides a means to add dynamic processors via `add_preprocessor` and `add_postprocessor`, and also the main means of executing said processors or otherwise specialized code though `preprocess` and `postprocess`.
An example of a class making use of this is your average Controller. You will typically want to execute some code before and after the requested action.

`Channeled` interface

A channeled object is an object that communicate using a channel or needs a channel to work. A `Channel` is just a `Processed`, `Meta` object. So essentially the whole idea of channels is you have this shared `Meta`.

This interface merely specifies the main getter and setter for such an object when dealing with channels, ie. `channel_is` and `channel`.

`Paged` interface

A lot of the time you deal with pagination. Paged does not deal with creating the pagination but telling an object what page you want. It is used as follows:

```
// limit result to a certain page
$statement->page(2, 20, 3); # 2nd page, showing 30 (skipped 3)
// the offset (3rd parameter) is optional
$statement->page(1, 15); # 1st page, showing 15
// if you want all simply provide null, if infinity is not an option this
// method will merely retrieve the maximum number of entries possible
$statement->page(null);
```

`Writable` interface

A writable enabled object is one that accepts a writer. The interface merely provides a generic way to add and retrieve the objects writer.

Implementing the interface doesn't mean the object itself is writable on, just that it works with a writer

An example interface that uses this type is the `Task`.

`Savable` interface

A savable object is one that requires it's present state to be saved when performing operations.

`Recoverable` interface

A recoverable object is one that provides a `recover` operation. The operation should reset the state of the object to something that can be processed given it's present unstable state.

Typically this involves the object being executable or renderable and once recovered the object is merely re-executed or re-rendered.

`Matcher` interface

A matcher object is one that provides a `check` method returning a boolean value of `true` or `false`. As it's name implies it's designed to match something, however the interface does not force any input conventions to allow for any type of match. Typically the matcher object will simply store the information for what to match against in it's state.

`Linkable` interface

A linkable object is one that can be converted to a URL, ie. universal resource locator.

The interface is specifically designed for converting to URLs and not URIs.

`Contextual` interface

An object implementing the contextual interface means the object has context, typically security relevant context (but not enforced as such). The interface provides a single universal `context` method that causes the object to either return null or an array representation of it's context.

This should not be thought of as serialization where the object in question can be deserialized. A contextual object is merely one that can provide context for inspection purposes.

`Eloquent` interface

An eloquent object is one that allows language prefixes. The way it works is that you provide a language prefix and any keys used for translation in the objects internals get prefixed with said language prefix, giving you control over all the objects translations but allowing you to have multiple instances of the same object with different translations.

`Exportable` interface

An exportable object is one that can produce an array representation of it's state. This is similar to serializing an object only the data representation is an array instead of a string.

Exporting an object is generally meant for exporting to another medium, hence the easier to work with array type, rather then exporting and importing back into a later session (ie. what serialization is meant for).

1.3.3 Caching Types

Caching consists of a main type `Cache` and two subtypes `Stash` and `TaggedStash`. The `Cache` type is merely a composite of the two.

The `Stash` consists of basic access methods: `get` and `set`, which function the same as the `Meta` type with the only exception that `set` also accepts an `expires` parameter.

In addition there is also a `delete` method for explicitly removing a key, and a `flush` method for deleting all keys.

A basic example,

```
// set a key with a value and keep the value for at most 60 minutes
$cache->set('my_key', 'my_value', 3600);
// you can omit expires to just use the default
$cache->set('my_key', 'my_value');
// retrieve the value
$some_var = $cache->get('my_key');
```

```
// by default if not set you will get null, you can configure this though
// the 3rd parameter
$some_var = $cache->get('some_key', 'some_default_value');
// explicitly remove a key
$cache->delete('my_key');
// purge the cache of all keys
$cache->flush();
```

The traits for `TaggedStash` will emulate the behaviour though the `Stash` interface if not explicitly implemented.

1.3.4 HTML Types

The main HTML type is `HTMLTag` which is mainly a `Meta`, `Renderable` composite with the added attributes `tagname` and `tagbody`. All the meta attributes directly translate to attributes on the tag.

So for example,

```
// create a tag
$tag = HTMLTag::instance()
    ->tagname_is('p')
    ->tagbody_is('hello, world');
// add a class to the tag
$tag->add('class', 'an');
// add another class
$tag->add('class', 'example');
```

Implementations will typically provide a meta renderer for `class` by default so rendering the `$tag` object above would yield `<p class="an example">hello world</p>`.

`HTMLForm` interface

The `HTMLForm` type is used to facilitate form management. The interface involves primarily a series of methods for creating fields, mainly `field` and several aliases; but implementation wise they all might be specialized. Fundamentally it is a `Standardized`, `HTMLTag`.

With a few exceptions, all fields follow the same pattern of `$label, $fieldname`. The `$fieldname` parameter is optional to allow for creating creating form elements that are meant to be script manipulated and not submitted, or submitted directly; for example fields in an equation only serves the purpose of providing input.
Exception to the field rule above are the following: `hidden` and `composite`. A `hidden` field is hidden so it does not have a label, only a field name. On the other hand a composite field is a amalgam of other fields so it has a label but does not have a field name.

To set up autocomplete you would use the `autocomplete` method (which accepts an array of values) and to retrieve a value you would use the autovalue method. Implementations will typically autopopulate a form if the request had a form parameter with the form's name.

Errors are specified via `errors_are` method. You can retrieve errors for a given field via the `errors` method.

Formatting wise you have access to proxy methods that will setup how `HTMLFormField` will be configured. These methods are `addfieldtemplate`, `addhintrenderer`, `adderrorrenderer`. You also have access to getters on the specific configuration via: `fieldtemplate`, `hintrenderer` and `errorrenderer`.

The interface also issues several signing methods. These are used to specify how the a specific tag in html belongs to the form. Typically these are used by `HTMLFormField` and on buttons in practice; since it's easier to write a button and sign it then to do it via the `HTMLTag` interface (very little benefit to doing it via `HTMLTag` as well).

The signature methods are `signature`, `sign` and `mark`. `signature` retrieves a raw signature for a given id, or return the form's base signature if no id is provided. `sign` will generate a basic sinature, typically just specifying the `form` attribute, while `mark` will issue a signature and any additional relevant meta; for example a `tabindex`.

The `basicuploader` and `nonuploader` are shorthand methods for configuring the form to handle file uploads; or not handle file uploads.

The form will typically include it's signature when it's created, to allow for autocomplete when a submission fails with errors, but sometimes this is not desirable such as when we have a GET based search form. To prevent the form from including additional fields on it's own, the `disable_metainfo` method can be used. `enable_metainfo` can be used to switch it back.

`HTMLFormField` interface

Like `HTMLForm`, a `HTMLFormField` is also a `Standardized`, `HTMLTag`. A `HTMLFormField` consists

of several parts. First are it's form related methods: `form` (getter and setter), and the operations `noerrors` and `showerrors`. `form` deals with which form we're handling and the two other methods toggle on or off the display of errors passed down by the form to the field.

After it's form related parts the field has it's basic attribute handlers, related to it's `value` and `fieldlabel`. `value` typically won't have a getter just a series of setters, with at least a default setter `value_is`; this is due to the potentially volatile nature of a field's value.

Finally a field has a template handling methods dealing with the following segments that compose a field:

- `hints`, which represent tips to the user one might attach to the field; such as what the minimum length for a user's password is, or that their password can be any length, etc
- `errors`, which represent a list of error messages. Some fields might consider only the top most error relevant but the interface always expects a list to be what is processed
- `fieldrender` which represents the core part of a field with everything else stripped away (for most fields this would be the humble `input` tag).

The default methods defined by the interface for the above are: `hint`, `hints`, `adderror` (to insert an error message), adderrors (to insert many messages), `errors` and `fieldrender`.

To defined the composition you would specify it via a `fieldtemplate`. The template will have the following symbols replaced with the corresponding element of a form:

- `:id` will be replaced with the field's id
- `:label` will be replaced with the field's label
- `:field` will be replaced with the field's `fieldrender`
- `:hints` will be replaced with the rendered version of the field's hints
- `:errors` will be replaced with the rendered version of the field's hints

By default the template will simply be `:field` for most implementations.

Also by default the `hintrenderer` and `errorrenderer` used above to replace `:hints` and `:errors` will output an empty string.

1.3.5 Database Types

The following types are database related:

- `SQLDatabase` deals with operations on a SQL based database
- `SQLStatement` are a byproduct of access to a SQL database
- `Schematic` is the standard migration interface
- `Marionette` is a general purpose object based modeling class
- `MarionetteDriver` is a driver support for marionette
- `MarionetteModel` is a base interface for single entity operations
- `MarionetteCollection` is a base interface for collection operations

Note: the `Marionette` system is a object system designed specifically for APIs, other systems are supported.

`SQLDatabase` interface

The `SQLDatabase` interface is designed to very easily interface with PDO. To this extent you'll have a `prepare`, `quote` and `last_inserted_id` methods. The methods should be self explanatory, `prepare` to create a new prepared statement, `quote` to make a string safe for concatenation and `last_inserted_id` is used for retrieving the id of the last entry.

In addition, the `SQLDatabase` interface requires the underlying database system to support transaction (denoted by `being`, `commit`, `rollback`). Transactions need to be *usable* when nested, so multiple begins, commits and rollbacks should function as expected and not interfere with each other.

Note that this is a *SQL database* interface and not meant to be used as a generic *database* interface. A generic database interface is not provided because there are no *generic* features to place in such a interface.

`SQLStatement` interface

The `SQLStatement` interface is a `Paged`, `Executable`, designed around compatibility with PDO.

For clarity the method syntax is very short and the interface forces a lot of shorthands. While daunting most are implemented into the trait so the explicitly required methods are actually very few unless the underlying system that implements the interface can offer a very specialized and efficient interface of it's own that matches said shorthands.

We'll avoid covering each and every method since there are many shorthands, but as a general readers guide:

- `num` stands for numeric and should be used with float, or integer values
- `str` stands for string and should be used with string values
- `date` stands for date & time and should be used with date and time values

- `bool` stands for boolean and should be used with boolean values or values that can be translated to a boolean
- the setters are the above with no other prefix or suffix
- to mass assign you take the initial types and add a **`s`** (ie. make them plural)
- if you wish to bind you precede the method with `bind` (ie. `bindnum` or `bindstrs`)
- stored procedure arguments are specified only though `arg` and `args`

To retrieve results you use the `fetch_*` methods. So `fetch_object` for retrieving as an object, `fetch_entry` for retrieving the first entry in a result, and `fetch_all` for retrieving all the entries. If your query was a calculation such as `SELECT COUNT(*) FROM something` you can use `fetch_calc` to get the value (you can also pass a default).

Both `fetch_entry` and `fetch_all` accept a fieldformat parameter which is essentially a list of mutation functions that are applied to the entry or entries after they are retrieved. For example you can specify that a datatime field is `'datetime'` and the result will have said field as a `\DateTime` object. Excessive use of this functionality is not recommended, since you will find you often do not need said field and hence just wasted processing time.

`Schematic` interface

The schematic interface is composed of actions that are performed to push the database schema up. There is no support for dropping the database down, beyond uninstalling everything. The reason for this is because returning to a previous state should be implemented as simply a more advanced state that is identical to the desired earlier state. This ensures no data is lost because you are forced to convert the data back.

The operations specified by a schematic are:

- `down`, drop a database (removing columns or renaming tables should be in `move`)
- `up`, this action is designed for creating new tables
- `move`, this action is designed for any changes to tables
- `bind`, any change related to constraints
- `build`, any operation that involves populating the database

`Marionette*` interfaces

The `Marionate` interface is designed as a base interface for `MarionetteModel` and `MarionetteCollection`, among others.

The `MarionetteCollection` interface is designed for use in APIs. To facilitate this it provides methods that correspond to a REST structure (ie. `get`, `put`, `post`, `delete`).
The `MarionetteModel` interface is designed for use in APIs. As the collection equivalent, it provides an API based on REST.

Following the marionette design, the implementation of `Marionette*` classes should not contain any non-REST operations.

Since the `Marionette` interfaces are designed to emulate REST, injecting custom methods for performing tasks is not compatible, so any functionality should be implemented though drivers, via the `MarionetteDriver` interface. The interface provides the following:

- `compile`, performed on POST, you should resolve input dependencies; operation will happen before validation
- `latecompile`, similar to `compile` only it is preformed after the the entry has been created; this operation is designed for tasks that require the entry to have an id such as associating tags to an entry
- `compilefields`, manipulates field list before database insertion happens
- `inject`, is performed on GET and works by alterning the query execution plan before it's executed (ie. joins, fields, postprocessors, etc)

Several misc setters are also provided.

1.3.6 Application Types

Application types are used in application composition. These are `Application`, `Layer`, `Channel` and to some extent `Controller`. Almost any application will be composed of a `Application` object managing `Layer` objects that communicate though a `Channel`. Optionally you have Controllers in the terminating layer doing work.

The layered design is meant to act as a modular execution plan. As an example, you can have execution plans that have security features, or have http features or have html features built in, and you can also have execution plans that don't even know what http is. The separation avoids complicated state logic and monolithic "kernel" objects. This is highly beneficial since the layers may be juggled around and reused for different specialized

goals.

As an aside, it is recommended you only use this approach if you can form a stack out of your execution. If the execution is only composed of one terminal layer that is very unlikely to accept any other layer then you're better of implementing the plan as a simple class that is self contained rather then use Application, Layer and so on. The task runner Overlord is a good example of what not to implement as a Application stack.

1.3.7 View Types

The view types are `RawView`, `View` and `ViewStash`.

`RawView` interface

The `RawView` interface is the base view and designed for generic implementation such as views that are not based on files.

To pass variables into the view you would use either `bind` for passing by reference or `pass` for passing the value. Note: PHP is "copy-on-write" so this is actually faster then passing by reference if you are not doing some complex manipulation.

To get a list of all the variables you use the `viewvariables` accessor. If you want to pass the variables of one view to another, you can call `inherit` and pass the desired parent view.

To generate inline views you use the `frame` and `endframe` methods. Calling `endframe` will return a string.

`View` interface

The `View` interface is merely a `FileBased RawView`.

`ViewStash` interface

A `ViewStash` can be used when processing relatively static content that requires a lot of processing to generate.

1.3.8 Theme Types

The theming system is defined by `Theme` objects that are used by `ThemeLoader`s for integrating themes in the request, `ThemeDriver`s responsible for handling theme specific requests and `ThemeViews` responsible for rendering themed view-based content.

`Theme` interface

A `Theme` object is a `Channeled Meta` object with various specialized accessors and setters, namely:

- `themename_*`/`themename` for retrieving and setting the theme name
- `themepath_*`/`themepath` for retrieving and setting the theme path
- `themeview` for creating `ThemeView` objects based on the theme

`ThemeDriver` interface

A `ThemeDriver` object is responsible for non-view theme requests (eg. css files, resource files, etc). A theme drivers is merely a `Channeled Renderable Resetable Recoverable` composite.

The point of theme drivers is to allow for support of both different languages (mjolnir for example has support for dart), but also alternative handling and build patterns, ie. supporting a specific compiled language, or resource files, etc.

`ThemeLoader` interface

A `ThemeLoader` object is responsible for the most part integrating `ThemeDriver`s and other dependencies into the request, as specified by the theme configuration.

A theme loader is merely a `Meta Channeled Executable` composite.

`ThemeView` interface

A `ThemeView` object is a `Channeled View` (note: `View` is `FileBased`), with a few theme specific

additions.

- `themepath_*`/`themepath` for retrieving the theme path
- `partial` for retrieving a sub view in the same theme
- `resource` for retrieving a file URL, a resource driver/loader is assumed to be provided by the implementation (albeit not internally)

Implementations may provide additional utalitarian methods. As an example, `mjolnir\theme\ThemeView` provides a `headinclude` and `footerinclude` for injecting code in the head of the page and right in the last part of the body respectably for cases where a HTML layer is used and hence the theme doesn't have direct control over those parts with out said methods.

1.3.9 Miscellaneous Types

Miscellaneous types are specialized types that unlike generic types are not designed to be reused by other types. They can have types based on them, but they are not created with the intention of having other types based on them.

`Instantiatable` interface

`Instantiatable` is the default interface for classes that have state. The mjolnir conventions require avoiding using `new` and always producing objects though factories. This interface merely enforces the convention on an object.

In general you should extend the class `Instantiatable` which implements this interface and enforces the convention. This interface is specifically designed for adapter patterns.

The interface merely provides an `instance` method, which as per the conventions needs to allow for the production of "default" or "neutral state" objects (ie. all parameters must be optional).

`Writer` interface

A `Writer` is as it's named implies an interface designed for objects that write to a specific medium (eg. console writer). The interface enforces conventions that allow for writing as well formatting.

`Lang` interface

The `Lang` interface is designed to facilitate internationalization. This interface is more of a convention because the object in question in not designed to be passed around as a parameter to another object (in general).

The interface is amalgam of static and object based handling and various other tools that are required.

The following are the tools described by the interface (all static), others may be included in an implementation.

- `targetlang_*`/`targetlang`, for specifying the language to translate to
- `idlang`, underscore version of language tag

In modules, static (ie. global) handlers should be used:

- `term` is used when it's desirable to fallback on the key text if the translation is missing
- `key` is used for retrieving a key based translation
- `load` is a helper for loading structured directories in the language files
- `file` is a helper for loading a file as a translation (eg. legal documents or just documents in general, that are better translated as a whole, rather then in pieces)

In the application space the library/index system (object based) is recommended:

- `addlibrary` loads a library into the translation index
- `idx` loads a term from the specific library

The indexed system is much cleaner and easier to use then the global system, but is ill suited for use in modules which are meant to be reusable.

The language configuration is left to the implementation, but in general passing parameters to the translation needs to be supported to ensure grammar can be taken into account when translating, since otherwise grammar rules need to be included in the code requesting the translation which either forces bad translations, convoluted grammar rules, or prevents certain languages from getting translations due to their grammar rules not being compatible with the rules provided.

`PDFWriter` interface

The `PDFWriter` interface provides a standard for converting html to PDF and distributing it, including stream.

`Pager` interface

The `Pager` interface like the `Lang` interface is more of a convention.

The interface provides some of the minimal functionality required for a pager. Implementations may provide additional functionality.

`Protocol` interface

The `Protocol` interface like the `Lang` interface is more of a convention.

As with the `Pager` the interface merely ensures minimum functionality.

`Puppet` interface

The Puppet interface provides a convention for named objects. The point of having named objects is to allow for dynamic resolution between classes, removing redundant declarations.

Implementations typically should extend the `Puppet` class provided by the library. This interface is provided for adapter patterns.

The interface ensures a class has the following methods:

- `singular`, singular name
- `plural`, plural name
- `dashsingular`, dashed version of singular
- `dashplural`, dashed version of plural
- `codename`, underscore version of signular
- `codegroup`, underscore version of plural
- `camelsingular`, camelcase version of singular
- `camelplural`, camelcase version of plural

`RelayNode` interface

A `RelayNode` is designed for universal routing.

Modules should always use `RelayNode`s.

`Task` interface

Task objects are a `Executable Meta Writable` composite.

Tasks are designed primarily for console use.

`TaskRunner` interface

A task runner object is a `Executable Writable` composite.

`URLRoute` interface

`URLRoute` objects are used in specifying as the name implies URL based routes, since routes may match other patterns that are not URL related.

`Validator` interface

A validator interface is a `Matcher` with methods for specifying fields and rules and the notion of errors.

`VideoConverter` interface

The video converter interface is used as a convention for converting videos from one format to another with a `convert` method.

1.4 Foundation Classes

1.5 Base Classes

1.6 Profiling

1.7 HTML Utilities

1.8 Access System

1.9 Cache Classes

1.10 Backend System

1.11 Themes

1.12 Documentation

1.13 Database Classes