Asatru PHP - Documentation

Codename: dnyAsatruPHP
Author: Daniel Brendel <dbrendel1988@gmail.com>

Welcome to the documentation of Asatru PHP – the lightweight PHP framework. This documentation covers all required information in order to create a new app with the framework and covers information of both the framework and the app skeleton. If you have any questions then feel free to contact me.

Table of contents

Installation

The installation is very easy. We suggest to use composer in order to create a new Asatru PHP app:

- Create a new folder where you want to have your project in
- Open your desired command terminal and switch to the directory
- Run the command: composer create-project danielbrendel/asatru-php
- Wait until composer has finished its job.
- That‘s it. You can now start developing apps with the framework.
- If everything worked fine you should see a welcome message under <url>/<project-dir>/index

Controller

New controllers are required to be in the folder app/controllers. First you may want to set your route. Open the file app/config/routes.php and add your definition to the array. The first token is the route. There are normal routes and special routes. Special routes are prefixed with the $-sign. For example there is a special route $404 which is used to define the handler if there occurs an error 404. You can also define any other server response code as special route which can be used by using the helper abort() that triggers a call to the associated handler. Normal routes specify the URL it handles. For instance if you want to handle the URL index then you just use as first token ‚/index'. You can have long routes, too, for instance /my/long/route. You can specify parameters by using {name} where „name" is the parameter identifier which can later be used in the controller to fetch the provided value.
The second token is the method and the handler. The method specifies the request type of which this route shall be handled, e.g. GET, POST, PATCH, etc. Specify ANY if the handler shall handle this route on any request type. The handler syntax is <object>@<method> where object both specifies the PHP script file to load and also the name of the controller class (in ucfirst-style, appending the „Controller" string) and method is the method of the controller class.

Let's assume we have the tokens: array(‚/index', ‚GET', ‚index@index')

So, the next step is to create a controller file. Controller files are located in app/controller. Create there a new file called index.php. Now inside create the class IndexController and also create a method „index()". This method will then be called when the URL is hit using GET method.

Additionally you can also specify a unique name of the route. This way you can access the route URL later via the route() helper. This is especially helpful if you use a route URL in multiple locations, so incase the route would change, you don't need to adjust all hardcoded route URL strings:

array(‚/index', ‚GET', ‚index@index', ‚index.get')

Now we can process the logic inside the controller. Regarding output your controller method can return class object instances of type ViewInterface. There are different ones provided already, but you can also create your own ones. Currently these exists:

- Asatru\View\ViewHandler: Lets you return a view. See section ‚Views' for details
- Asatru\View\PlainHandler: Lets you return plain text. No HTML is rendered
- Asatru\View\JsonHandler: Lets you return Json content
- Asatru\View\XmlHandler: Lets you return XML content
- Asatru\View\CsvHandler: Lets you return CSV content
- Asatru\View\RedirectHandler: Lets you redirect to an URL
- Asatru\View\DownloadHandler: Lets the client download a file
- Asatru\View\CustomHandler: Lets you customize the type of output

Each controller handler method accepts a $request variable. It references a request object which can be used to query the dynamic URL parameters or the request data:

- Call $request→arg(<name>, <fallback>) in order to fetch an URL parameter
- Call $request→params()→query(<name>, <fallback>) fo fetch a HTTP request data.

If your controller handles a form then you also want to verify the form data. Therefore you can use the Asatru\Controller\PostValidator class. In the constructor you specify your verify tokens. After that you can call the isValid() method of the class to check if the data is valid. It returns true on success. If returned false it means that your form data is invalid. To get to know all invalid items you can get an array of error messages via errorMsgs(). Every form will be checked for the CSRF token, so don‘t forget to @csrf in your form markup. The constructor requires an array of form items to be checked. The first token is the name of the form element and the second token is the validation string. Each token can have multiple validators. The following validators exist

- required: This item must be provided
- email: The item must contain a valid E-Mail address
- min:<num> The item must contain at least <num> characters
- max:<num> The item must contain not more than <num> characters
- datetime The item needs to be in a valid datetime format
- number The item needs to be a valid number
- regex:pattern The item must match the regex pattern

Separate a validator with the | character, e.g. „required|email“.

You can also implement own validators. These are placed into the \app\validators folder. The script file name must also be the class name (ucfirst) with „Validator“ appended. It extends the Asatru\Controller\BaseValidator class. You need to provide the following methods:

- public function getIdent(): Return the name of your validator ident. This is used in the validation string identifying your validator.
- Public function verify($value, $args = null): This method is used to validate the data. It recieves the value that needs to be checked and optionally arguments that to adjust the validation
- public function getError(): Here you need to return an error string describing the failure if the validation has failed

You can create a validator comfortably using the asatru CLI command.

Your controller class can be derived from Asatru\Controller\Controller. It defines two additional methods: preDispatch() and postDispatch(). The first one is called before the route handler. Here you can do initialization stuff and throw exceptions if something bad happens. The latter one is called after the the route handler has finished.

Note: You can also derive your controllers from the app skeleton base controller (which in turn can derive from the framework base controller). The app skeleton base controller must be placed in the controller folder and named _base.php and its class name must be BaseController.

Views

When your controller method is called you will most of the times want to render a view. There you can utilize the Asatru\View\ViewHandler class. You can specify three different items:

- ViewHandler::setVars(<array>): Lets you pass variables to the view as key-value pairs
- ViewHandler::setLayout(<string>): Lets you specify the layout file to be used
- ViewHandler::setYield(<string>, <string>): Lets you specify various yields defined in your layout file.

Alternatively you can call the static method create() with first argument the layout file, as second argument a key-value pair of yields and as last argument the key-value paired variables array. There is also a helper function view() available.

A layout file is used as your basis view layout. There you can for instance specify the header of the HTML document and a footer. A yield is defined via {%name%} where „name“ is the name of the yield which is used via setYield(). Basically every yield token will then be replaced with the content of the yield file.

Inside your view files you can also use the template engine. This is especially useful for writing code faster and more convenient. The following template commands exist

- @if: Renders an if statement, for instance: @if ($myvar === true)
- @elseif: Continue with an alternative if statement. For instance: @elseif ($myvar === false)
- @else: Specify an else statement
- @endif: Ends the if condition
- @for: Renders a for loop. For instance: @for ($i = 0; $i < 10; $i++)
- @endfor: Ends the for loop
- @foreach: Renders a foreach loop. For instance: @foreach ($tokens as $token)
- @endforeach: Ends the foreach statement
- @while: Renders a while statement. For instance: @while ($a < 10)
- @endwhile: Ends a while statement
- @break: Renders a break statement
- @continue: Renders a continue statement
- @switch: Renders a switch statement. For instance @switch ($var)
- @case: Renders a case condition. For instance @case 1
- @default: Renders a default condition.
- @endswitch: Ends the switch statement
- @comment: Renders a comment. For instance @comment This is rendered as a comment
- @include(‚file‘): Replaces the line with the rendered content of the specified file. For instance @include(„my-file.php“)
- @isset: Checks whether an object is set
- @isnotset: Checks whether an object is not set
- @endset: Ends the block if an isset/isnotset statement
- @empty: Checks if an expression is empty
- @endempty: Ends the block of @empty
- @notempty: Checks if an expression is not empty
- @endnotempty: Ends the block of @notempty
- @debug: Handles the block if in debug mode

- @enddebug: Ends the block of @debug
- @env(‚var‘, ‚opt:value‘): Either checks if an env var exists or if a non null value is provided it checks if it equals the value
- @endenv: Ends the block of @env
- @php: Start a PHP code block
- @endphp: End a PHP code block
- @end: Generic block ending
- @csrf: Inserts a hidden input field with the current CSRF token used for forms
- @method(‚name‘): Inserts a hidden input field with a custom request method, useful to use request methods such as PATCH, PUT or DELETE in your forms.
- {{ <output> }}: This will output your code into the buffer. This can be used to output variables. Internally it uses htmlspecialchars in order to prevent XSS.
- @{{ expression }}: Will leave the expression wrapped in brackets.
- {!! <output> !!}: This will output your code into the buffer, but without using htmlspecialchars. Useful if you want to render HTML into the document

You can also create custom template commands. Add them with the helper template_command().

You can use flash messages. You can check if a flash message exists with FlashMessage::hasMsg(<name>) and query a flash message with FlashMessage::getMsg(<name>) both passing the name of the flash message. You can set a flash message in your controller with FlashMessage::setMsg(<name>, <text>).

Resources such as CSS or JavaScript files should be placed inside the app/resources folder when using as input (for instance for webpack) and in public/ when referencing in your code.

Database access

In order to perform database access you may want to create models and migrations. The models are the interface between your app and the database. It uses prepared statements in order to prevent SQL injection. The migrations are used to create a fresh database with fresh tables.

In order to create a migration go to the app/migrations folder and create a new file, e.g. ExampleModel.php. Now add a class with the name ExampleModel_Migration. The „_Migration“ will be appended to the script file name (ucfirst, too) in order to resolve the class name. Inside the class you specify two methods: up() and down(). The down method is used to remove the table and the up method is used to create the table. Your class must provide a constructor where it recieves the PDO connection handler instance. Save it to a private member variable. Then in the up() method you instantiate a Asatru\Database\Migration object passing the name of the table as first argument and the connection handler reference as second argument. After that you can call the methods of the created object to define your table. At first you may want to drop the old table via the drop() method. Then you will want to define the columns. Therefore you use the add() method. You pass an SQL query string in order to create a column, e.g. „text varchar(260) not null“. When you have added all your desired columns then you call the create() method in order to create the table. In order to alter a table you just use the method append() in order to insert a new column. Note that you may not mix creation of new tables with altering tables.

Additionally to the add() method you can also use the following helper to create columns. Using these helpers you must first call the column() method to init a new column creation, following some other helpers, and then finish it via calling the commit() method.

- column(name, data_type, opt:size): Init new column creating with name and datatype. Optionally you can also specify the size
- collation(ident): Specify a collation for the column
- charset(ident): Specify a character set for the column
- nullable(opt:flag): Specify if the column can be null or not
- default(value): Specify a default value for the column
- unsigned(opt:flag): Specify if the column can be unsigned
- comment(text): Specify a comment for the column
- auto_increment(): Enable auto increment for the column
- primary_key(): Set this column as primary key
- after(column): Place this column after the specified column
- commit(): Finish column creation

In your down() method you call the drop() method of a database object.

Next step is to create a model for that migration. These are created in the app/models folder. Create a file called, for instance, ExampleModel.php. Then open the script file and create a class ExampleModel. It must have the same name as the script file to be resolved. Also it extends the Asatru\Database\Model class.

Important note: The model file and class name are associated with the belonging migration. A model class named ExampleModel expects a database table with the name ExampleModel. In order to comfortable create a model and migration please use the CLI tool. There you just have to specify your model name and then everything will be created automatically, so you don’t have to manually create models and migrations.

Now you can implement your static getters and setters. You can perform select, update, insert, delete and raw queries:

- Model::where(name, comparision, value): Use a conditional and-query. Call the method for each condition
- Model::whereOr(name, comparision, value): Use a conditional or-query. Call the method for each condition
- Model::limit(count): Limit the query result
- Model::groupBy(ident): Group items by ident
- Model::orderBy(ident, type): Order items by ident. Type is either asc or desc.
- Model::first(): Get first item. Returns a collection with accessors to the items
- Model::get(): Perform the query and get the items. Returns a collection with collections for each retrieved item
- Model::all(): Get the entire table. Returns a collection with collections for all items.
- Model::find(id, key): Find an entry by id. Use key parameter if you want to specify the name of the key so look for
- Model::count(): Get the amount of found items
- Model::aggregate(type, column, opt:name): Find an aggregate of the column (avg, min, max, sum, etc.)
- Model::whereBetween(column, value1, value2): Use a conditional between and-query. Call the method for each condition
- Model::whereBetweenOr(column, value1, value2): Use a conditional between or-query. Call the method for each condition
- Model::update(ident, value): Add this item to the updated item list
- Model::insert(ident, value): Add this item to the inserted item list
- Model::go(): Perform either an update or insert operation
- Model::delete(): Perform a delete operation
- Model::raw(qry, args): Perform a raw database operation (Identify current table with the @THIS special variable)
- Model::toSql($withParams): Return the prepared SQL statement instead of performing the actual query. Set optional parameter to true if the actual params shall be integrated.

The result of the operation depends of the its kind:

- For fetching data it returns an instance of Asatru\Database\Collection. The Collection class implements Iterator and Countable, so you can use the count() function and also use a class instance with the foreach iteration loop.
  - Call the count() method to get the amount of collected entries
  - Call the get(<ident>) method to get the related item. This can be an instance of the collection class, too.
  - Call the first() method to get the first item in list
  - Call the last() method to get the last item in list
  - Call each(<callback>) in order to iterate through all collected items
  - Call the asArray() method if you want to return the data as array
- For inserting, updating and deleting it returns a boolean indicating whether the operation could be executed
- For getting the count it returns the amount of found entries

In order to manage migrations you can use the following functions:

- migrate_fresh($echo = false): Drops all tables and recreates them
- migrate_list($echo = false): Runs only newly created migration scripts
- migrate_drop($echo = false): Drops all migrations

If you set $echo to true then it will print out the current handled migration.

The database connection is adjusted via the .env file. See the related section for details.

Modules

Where database models are solely dedicated to be a representation of a database table you will also want to create business logic modules in order to keep your controllers clean. As a general good coding practice you should not put everything into your controllers. Business logic can be placed in modules. Modules are basically classes that are autoloaded during application bootstrap process. Then you just need use the class as how you wish. Modules are placed inside the modules directory. The file name must equal the class name.

In order to create a new module you can use the console.

Autoloading

The framework supports composer autoloading. But you can also use the frameworks own autoloader. Therefore go to the app/config/autoload.php file and add your file to the array. There you specify a path to your file relative to the app directory.

Config management

The framework offers the possibility to query configuration data from config files. Config scripts should preferably return arrays.

A config file can look as follows:

```
return [
  ‚var‘ => ‚value‘,
  ‚another‘ => [
     //More data
  ]
];
```

Config files are placed in the app/config directory. You can query config data using the config() function:

```
$data = config(‚test‘); //Loads data from app/config/test.php
//Or
$data = config(‚folder/test‘); //Loads data from app/config/folder/test.php
```

If the config script returns an array then it is returned as object by default. In order to turn off this behaviour and return the array instead, simply pass ‚false‘ as second argument.

```
$data = config(‚test‘, false);
```

Localization

The framework supports localization. You can create own language files within the app/lang directory. For each language you create a new folder with the name of the localization identifier, e.g. „en“ for english. Then you can create the language files. It does not matter what name it is, but you use the name of the file later to query a phrase inside that file. For instance create a file called app.php and insert your phrases there. The script file shall return an array with the name of the phrase and the belonging text sentence. You can then query a phrase with the __(phrase, opt:keyvalues) function. For instance if you want to query the phrase „myphrase“ inside app/lang/app.php you just call the function like __(„app.myphrase“). If the phrase is found within the language file then it will be returned, otherwise the phrase identifier is returned. You can also pass an optional key-value pair array providing variables that can be set. In the language phrases variables can be defined with {name}.

In order to change the current language you can call setLanguage(<localeidentifier>), e.g. „en“. To get the current language you can call getLanguage().

Logging

The framework supports logging. When logging is enabled a logfile will automatically be created inside the app/logs directory.

Use the addLog() function in order to log to the buffer. You can use different log types:

- ASATRU_LOG_INFO: An information message
- ASATRU_LOG_DEBUG: A debug message
- ASATRU_LOG_WARNING: A warning message
- ASATRU_LOG_ERROR: An error message

In order to clean the current log buffer use clearLog(). In order to force storing the current log buffer use storeLog().

Logging can be toggled via the LOG_ENABLE environment variable.

Environment configuration

The app configuration in done via a .env file located in your projects root directory. It covers installation related environment configurations. For example on your local machine you can have a different .env file than on your webserver. Of if you have created an app which ships to different users, each user has an own .env file. The .env file is automatically parsed. There you can specify different variables. Currently these exist:

- APP_NAME: The name of your app
- APP_VERSON: The version of your app
- APP_AUTHOR: The name of the author of the app
- APP_CONTACT: Contact information of the app
- APP_DEBUG: Toggle debug mode
- APP_BASEDIR: Use this to specify the base dir on the webserver where your project is located in
- APP_TIMEZONE: Set a custom timezone by specifying a valid timezone identifier or just null or an empty string if you want to use the system default timezone
- SESSION_ENABLE: Toggle here whether you want to use sessions in your app
- SESSION_DURATION: Time in seconds when the session shall expire or null/0 if it shall expire when closing browser/tab
- SESSION_NAME: Specify an alternative session cookie name
- DB_ENABLE: Set to true if you want to connect to a database
- DB_HOST: The host address for your database connection
- DB_USER: The database user name
- DB_PASSWORD: The database password
- DB_PORT: The port which shall be used for connection
- DB_DATABASE: The name of the actual database
- DB_DRIVER: The driver to be used. Currently only MySQL is supported.
- DB_CHARSET: Charset for when connecting to the database.
- SMTP_FROMNAME: Specify the name of the sender, e.g. your application name
- SMTP_FROMADDRESS: Specify the e-mail address which shall be used as the sender address
- SMTP_HOST: Hostname or address of the SMTP server
- SMTP_PORT: Port to connect through, e.g. 587 for TLS encrypted connection
- SMTP_USERNAME: Login name for the server
- SMTP_PASSWORD: Login password for the server
- SMTP_ENCRYPTION: Either ‚tls‘ or ‚smtps‘.
- LOG_ENABLE: Boolean to toggle logging

There is a special environment variable for testing (.env.testing) required depending on your environment:

- APP_URL: Base URL to be used for testing controller actions. For instance: http://localhost:8000

To query an environment variable you can just use the $_ENV superglobal or use the env_get() function. These helper function exists:

- env_parse(<file>): Parse additional .env files
- env_get(<ident>): Query an environment variable value
- env_exists(<ident>): Checks if an environment variable exists
- env_clear(): Clear all variables. This is normally not needed
- env_hash_error(): Check if there has been an error when parsing the env file
- env_errorStr(): Query the error string of the last parsing error

Feel free to add more environment variables to your .env file. Supported are strings, integers, floats, booleans and null.

Helpers

The framework provides some basic helper functions in order to ease your workflow. The following ones exist:

- base_path($path): Returns the full path to your projects root directory where additional $path can be appended
- app_path($path): Returns the full path to your app directory where additional $path can be appended
- public_path($path): Returns the full path to your public directory where additional $path can be appended
- base_url($port = false): Returns the full URL to your projects folder. If port is not false then the port will be included in the URL.
- url($to): Return the full URL to the given destination or the URL root if none specified
- asset($asset, $mt = false): Return the full URL to the specified asset. When in debug mode and the asset does not exist then the framework throws an exception. Set $mt to true in order to include last modification time. This is useful when you want the browser to update its cache when the asset has been updated.
- csrf_token(): Returns the current CSRF token of your session
- template_command($ident, $callback): Adds a new template command. First param is the name of the command, second param is a callback of type: function (string $code, array $args):string. $code contains the code line with the template. $args contains the arguments of the template command if any (must be wrapped in brackets like valid PHP code). Return value is a string with the code that the template command line shall be replaced with.
- view($layout, array $yields, $vars = array()): Short way of spawning a ViewHandler instance. As opposed to using the ViewHandler directly you can also pass only an array with yield and render file instead of array containing an array. This is useful if you only want to render one yield.
- json(array $content): Short way of returning JSON content
- xml(array $content, $root = ‚data‘): Short way of returning XML content
- csv(array $content, array $header = null): Short way of returning CSV content
- text($content): Short way of returning plain text content
- custom($type, $content): Short way of returning custom content
- redirect($to): Short way of redirecting to an URL. Also used for form submissions if everything is valid. Old POST data is not stored.
- back(): Go back to last URL. Use case is for submitting forms. Useful for dealing with errors during submitting forms. Old POST data is temporarly saved. Internally the system determines the proper request URL by saving the URL from the last GET request that was not an XHR request.
- download($resource): Short way of letting the client download a resource
- env($item, $fallback = null): Shortname function to call env_get().
- envck($item, opt:$value): If $value is provided and not null the values are checked, otherwise it just indicates if the env var exists
- abort($code, $ctrl = null): Used to handle server error response codes. Triggers a call to an associated special route handler that handles the error.
- old($key): Return old POST data. Useful if form data shall not be lost if the form data is invalid and the user has to re-submit it.
- slug($content, $delimiter = ‚-‘): Create a slug from a source content string using the given delimiter

- route($name, $values = []): Get a named route and optionally set all variables via the key-value paired array

If you want to use localized Carbon then you can use the Carbon helper to automatically use the current locale. Therefore just create an instance from Carbon (global namespace) instead of Carbon\Carbon.

Exceptions

Exceptions may occur during development of your app or during production. The framework has an exception handler installed. You can control its behaviour via the APP_DEBUG variable (located in your .env file).

If this variable is set to true then every output of an exception will be shown as output in your browser with detailed information. You can edit the layout in the file app/views/error/exception_debug.php.

If it is set to false (or does not exist) then there will just be an error shown (server error 500) in your browser. This is useful to hide those debug messages from your clients. You can edit its layout in the file app/views/error/exception_prod.php.

Exceptions can also be stored to the log file.

CLI tool

Asatru PHP comes with a handy CLI tool which allows you to perform some operations.

Just open your prefered terminal and cd to the directory of your project. Then run the command „php asatru“ to get a list of commands.

The following commands exist:

- help: Displays the help text
- make:model <name>: Creates a model and migration file according to the name
- make:module <name> <opt:args[]>: Creates a new module that is dedicated to your business logic. Arguments can be –base to create a base class, --extends <opt:name> to extend a base class and –final to create a final class.
- make:controller <name>: Creates a new controller
- make:language <ident>: Creates a new language folder structure with an app.php
- make:validator <name> <ident>: Creates a new validator with the given name and the associated validator ident
- make:event <name> <initial_method>: Creates a new event handler with name as class name and an initial handler method called (initial_method)
- make:command <name>: Creates a new command class
- make:auth: Creates a model and migration used for authentication
- make:cache: Creates a model and migration used for caching
- make:test <name>: Creates a new test case with the given name
- migrate:fresh: Creates a fresh migration of your database. Warning: This will erase all previously inserted data, so please be careful.
- migrate:list: This will only run the newly created migrations
- migrate:drop: This drops all migrations
- serve <port>: Starts a development server. If port is not provided it uses the port 8000.

Note that the CLI tool is only available in debug mode.

Events

In some situations you may want to raise an event where specific tasks are processed. You can do that by the event management of Asatru PHP. First of all you have to register an event in the \app\config\events.php configuration file. The key of an item is the name of the event which will later be used to raise that event. The value is a pair of event handler and method. You have to create a PHP script in the \app\events directory with the name of the first token (lowercase). Then create a class with the name of the first token. Then implement a method with the name of the second token with a param ‚data' defaulted to null.

For example if you have the line ‚my_event' => ‚MyEventHandler@myMethod' you will create a file \app\events\myeventhandler.php and create a class named MyEventHandler and implement there the public method myMethod($data = null).

In order to raise an event you call the event() function passing the name of the event and optionally an argument with data. For instance event(‚my_event', [‚someData' => ‚my value']);

Commands

You can also create custom commands that can be used via the asatru CLI tool. Command handler classes are stored in the app/commands directory and they have to be registered via the app/config/commands.php configuration script file.

For each command handler you can add an array item to the config array where the first item specifies the name of the command, the second specifies a help description text and the third item specifies the class and script name of the command handler.

```
[
  array(‚command:name‘, ‚This is a description‘, ‚TestCommand‘),
],
```

Then there needs to be a TestCommand.php file in the app/commands directory defining a class called TestCommand that needs to implement the Asatru\Commands\Command interface.

A command class has to at least implement the non-static handle($args) method. The argument parameter recieves an object of Asatru\Commands\Arguments that is iterable and countable. Each item is an Asatru\Commands\Argument class instance that you can use to access the actual argument. The following methods are available:

- getValue(): Gets the current value
- getType(): Gets the value type
- isNull(): Indicates if the value is null
- isEmpty(): Indicates if the value is empty

You can use the CLI tool to comfortably generate a custom command.

Authentication

Via the Asatru CLI you can create basic authentication components. Be sure to run the migration command (preferably list) after you have issued the command.

The authentication components consist of migrations and models for authentication and session. It features multisession logins.

The authentication model provides you with the following methods:

- register(username, email, password): To register a new user
- confirm(token): Used to confirm validity via a confirmation token
- login(email, password): To log a user in
- logout(): To log the current user out
- getAuthUser(): Get current authenticated user if any
- getByEmail(email): Get a user data object by email
- getById(userId): Get a user data object by user ID

The session model provides you with the following methods:

- loginSession(userId, session): Logs the user in with current session
- logoutSession(session): Performs a logout from the current session
- findSession(session): Returns a data object from the given session
- hasSession(userId, session): Indicates if there is an active user session
- clearForUser(userId): Clears all sessions of that given user

Caching

Via the Asatru CLI you can create a caching model and migration. Be sure to run the migration command (preferably list) after you have issued the command.

The cache model provides you with the following methods:

- remember(ident, timeInSeconds, closure): To fetch a value either from cache or directly depending on the item existence / expiring status
- query(ident, default): Return the value of the given element if exists or the default value if not. Additionally you can return other column values specified via the dot-syntax. For instance: my_cache_item.updated_at returns the updated_at value of the item with ident „my_cache_item“ if it exists, otherwise the specified default value
- has(ident): To check if an item exists in the cache
- elapsed(ident, timeInSeconds): Returns true if the elements cache time is elapsed, otherwise false
- pull(ident): To obtain and remove an item from the cache
- put(ident, value): Write the given value to the cache with the given ident
- forget(ident): To remove an item from the cache

Testing

Asatru PHP utilizes PHPUnit in order to perform tests. Tests shall be located in the app/tests directory. A test file has to have the postfix ‚Test'. For example if you have the test „Example" then the file must be named „ExampleTest.php" and the class of the test case must be named „ExampleTest".

A test case class should be derived from Asatru\Testing\Test class. This way your test is well suited for testing your application so you can also test your routes.

For testing the .env is not parsed, but instead a .env.testing which you have to create. The APP_DEBUG is ignored because for testing it is forced to true.

If you want to test a route you can simply call Asatru\Testing\Test::request(method, url, data). Method is the method of request, e.g. GET, POST, PATCH, etc. The following request constants exist:

Asatru\Testing\Test::REQUEST_GET
Asatru\Testing\Test::REQUEST_POST
Asatru\Testing\Test::REQUEST_PUT
Asatru\Testing\Test::REQUEST_PATCH
Asatru\Testing\Test::REQUEST_DELETE

The URL is the same as someone would type it in your browser. Data is an array where you can set GET and POST data. For instance:

array(„GET" => array(„somevar" => „somevalue), „POST" => array(„somevar" => „somevalue"))

You can also put query params into the URL which will then be added to the $_GET array.

Note that you can also directly test entities such as Models or Modules because your tests are executed inside the context of the application. However testing specific routes helps you validate the workflow of an execution circle as it could happen in reality.

The following methods are available to deal with controller route tests:

- request($method, $route, $data = array()): Simulates a request to a controller route
- getResponse(): Returns the response of the previous controller route request
- getDatatype(): Returns the data type of the previous controller route request response. If it is a class then the actuall full qualified class name will be returned
- getDebugInfo(): Returns the debug info for the previous controller route request response. Internally uses print_r to generate the debug info
- getTimeDiff(): Returns a float that holds the execution time for the previous controller route test workflow.

You can use the Asatru CLI to create a new test case.

mail() wrapper

The framework comes with a convenience wrapper for the PHP inbuilt mail() function. It can be used to use views with templating as mail content.

The related Asatru\Mailwrapper\Mail class provides the following methods:

- setRecipient(value): The E-Mail address of the recipient
- setSubject(value): The subject
- setMessage(content): The message content
- setView(layout, array yields, opt:data): Used layout and yields with optionally variables
- setAdditionalHeaders(headers): Set additional header information as string here
- setAdditionalParameters(params): Set additional parameters as string here
- send(): Actually call mail() to send the E-Mail

SMTP Mailer

The framework provides an interface to the PHPMailer package. Like the mail() wrapper you can send HTML content by specifying views. Except it uses SMTP protocol to send mails. This requires an SMTP service running on a server. The SMTP settings can be adjusted via the .env file (see related section in this documentation).

The related Asatru\SMTPMailer\SMTPMailer class provides the following methods:

- setRecipient(value): The E-Mail address of the recipient
- setSubject(value): The subject
- setMessage(content): The message content
- setView(layout, array yields, opt:data): Used layout and yields with optionally variables
- setProperties(array): Used to specify further properties for the current PHPMailer instance. Does overwrite properties previously set by other methods.
- send(): Send the mail using the SMTP server

Html helper

The Html helper class can be used to programmatically render HTML elements into the document. It provides the following methods:

- renderTag($name, array $attr): Returns a string with the Html tag where you have to provide the tag name and an optional array of attributes. Specify null for attribute value to have the attribute be rendered without a value.
- renderCloseTag($name): Returns a string containing the closed Html tag

Form helper

The Form helper class extends the Html helper class and provides a convenient way to render forms into the document. It provides the following methods:

- begin(array $attr): Begins the form. You can optionally specify attributes via the array param.
- putElement($name, array $attr): Put a form element. You can optionally specify element attributes via the array param
- closeElement($name): Close the current form element
- end(): Finish form rendering

Note that each method returns a string with the related content.

npm/webpack

The framework supports npm and webpack to build your JavaScript and CSS files. Define your JavaScript implementations in ‚app/resources/js/app.js‘ (you can also create other files which can be imported into the app.js) and define your SASS rules in ‚app/resources/sass/app.scss‘ (you can also create other files which can be imported into the file).

Whenever you have finished something and want to build your work then run the command ‚npm run build‘. This will bundle your SASS and JavaScript code into the app.js in the ‚public/js/app.js‘ file.

You can also run ‚npm run watch‘ if you want to let your file changes be monitored and re-built whenever you make a change.

Additional JavaScript and CSS files can be put to public/js and public/css directories. Images can be placed into public/img directory.