Asatru PHP - Documentation

Codename: dnyAsatruPHP
Author: Daniel Brendel <dbrendel1988@yahoo.com>
Version: 0.1

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. 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:dev-master
- 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. 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. POST or GET. 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 regardless if it's POST or 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: Let's you return a view. See section ‚Views' for details
- Asatru\View\PlainHandler: Let's you return plain text. No HTML is rendered
- Asatru\View\JsonHandler: Let's you return Json code
- Asatru\View\XmlHandler: Let's you return XML code
- Asatru\View\RedirectHandler: Let's you redirect to an URL
- Asatru\View\DownloadHandler: Let the client download a file
- Asatru\View\CustomHandler: Let's 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 POST or GET data:

- Call $request→arg(<name>) in order to fetch an URL parameter
- Call $request→request()→query(<name>) fo fetch a POST or GET parameter.

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

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.

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>): Let‘s you pass variables to the view
- ViewHandler::setLayout(<string>): Let‘s you specify the layout file to be used
- ViewHandler::setYield(<string>, <string>): Let‘s you specify various yields defined in your layout file.

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. It saves you a bit of time. 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: 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
- @csrf: Inserts a hidden input field with the current CSRF token used for 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.

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

Resources such as CSS or JavaScript files should be placed inside the app/resources folder.

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. example_migration.php. Now add a class with the name Example_migration_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.

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, MyModel.php. Then open the script file and create a class MyModel. It must have the same name as the script file to be resolved. Also it extends the Asatru\Database\Model class. You have to implement the static method tableName() which returns a string that identifiers the migration associated with this model. After that 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
- Model::get(): Perform the query and get the items
- Model::all(): Get the entire table
- 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

The result of the operation depends of the its kind:

- For fetching data it returns an instance of Asatru\Database\Collection
  - 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 each(<callback>) in order to iterate through all collected items
- 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(): Drops all tables and recreates them
- migrate_list(): Runs only newly created migration scripts
- migrate_drop(): Drops all migrations

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

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.

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 your app is in debug mode 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:

- LOG_INFO: An information message
- LOG_DEBUG: A debug message
- LOG_WARNING: A warning message
- 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().

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 his on .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_SESSION: Toggle here whether you want to use sessions in your app
- 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 supported are MySQL (mysql), MS SQL Server (mssql), Oracle (oracle) and SQLite (sqlite)

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_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(): Returns the full path to your projects root directory
- app_path(): Returns the full path to your app directory
- resource_path(): Returns the full path to your resources directory
- base_url(): Returns the full URL to your projects folder
- app_url(): Returns the full URL to your app directory
- resource_url(): Returns the full URL to your resources directory
- csrf_token(): Returns the current CSRF token of your session

Exceptions

You may want to throw exceptions from time to time. 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 show on as output in your browser. You can edit the layout if you want 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 are also appended to the log buffer.

CLI interface

Asatru PHP comes with a handy CLI interface 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> <table>: Creates a model and migration file where name is the name of the model and table is the associated table of the database
- 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
- 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 interface 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']);