Phlyty - Documentation

Release 0.1.0dev

Matthew Weier O'Phinney

August 26, 2014

Contents

1 Overview **3**
- 1.1 Installation . . . 3
- 1.2 Basic Usage . . . 3

2 Routes **7**
- 2.1 Constraints and Defaults . . . 7

3 Controllers **9**
- 3.1 Anonymous Function . . . 9
- 3.2 Closures . . . 9
- 3.3 Lambdas . . . 10
- 3.4 Named Functions . . . 10
- 3.5 Static Class Methods . . . 10
- 3.6 Instance Methods . . . 11
- 3.7 Functors . . . 11

4 Helpers **13**
- 4.1 Workflow Helpers . . . 13
- 4.2 HTTP-Related Helpers . . . 15
- 4.3 Route-Related Helpers . . . 16
- 4.4 View-Related Helpers . . . 16

5 Events **19**
- 5.1 Defined Events . . . 20
- 5.2 Use Cases . . . 20

6 Views **23**
- 6.1 The ViewInterface . . . 23
- 6.2 Mustache Integration . . . 23

7 Api docs **25**

8 Getting help **27**

9 Indices and tables **29**

Phlyty is a PHP microframework written on top of Zend Framework 2 components.

Contents:

CHAPTER 1

Overview

`Phlyty` is a PHP microframework written using Zend Framework 2 components. It's goals are:

- Route based on HTTP method and path, but allow the full spectrum of ZF2 routes.
- Allow any callable as a "controller".
- Provide basic features and helpers surrounding flash messages, URL generation, and request/response handling.
- Provide view rendering out-of-the-box, but allow the user to plugin whatever view rendering solution they desire.

The features and API are roughly analagous to Slim Framework.

1.1 Installation

I recommend using *Composer <https://getcomposer.org/>*. Once you have composer, create the following `composer.json` file in your project:

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "http://packages.zendframework.com/"
        }
    ],
    "minimum-stability": "dev",
    "require": {
        "phly/phlyty": "dev-master"
    }
}
```

Then run `php composer.phar install` to install the library. This will ensure you retrieve `Phlyty` and all its dependencies.

1.2 Basic Usage

The most basic "Hello World!" example looks something like this:

```
use Phlyty\App;
include 'vendor/autoload.php';
```

```
$app = new App();
$app->get('/', function ($app) {
    echo "Hello, world!";
});

$app->run();
```

Assuming the above is in `index.php`, you can fire up the PHP 5.4 development web server to test it out:

```
php -S 127.0.0.1:8080
```

If you then visit `http://localhost:8080/`, you'll see your "Hello, world!" text.

1.2.1 Routing

The main `Phlyty\App` class contains methods for each of the main HTTP request methods, and these all have the same API: `method($route, $controller)`. They include:

- `get()`
- `post()`
- `put()`
- `delete()`
- `options()`
- `patch()`

All of them return a `Phlyty\Route` object, allowing you to further manipulate the instance – for example, to name the route, indicate what additional HTTP methods to respond to, or to access the controller or the composed ZF2 route object. (You can actually instantiate a ZF2 route object and pass that instead of a string for the route, which gives you more power and flexibility!)

```
$app->map('/', function ($app) {
    echo "Hello, world!";
})->name('home'); // name the route
```

Alternately, you can use teh `map()` method. This simply creates the route, but does not assign it to a specific HTTP method. You would then use the `via()` method of the route object to assign it to one or more HTTP methods:

```
$app->map('/', function ($app) {
    echo "Hello, world!";
})->via('get', 'post')->name('home'); // name the route, and have it respond
                                      // to both GET and POST requests
```

By default, if you pass a string as the `$route` argument, `Phlyty\App` will create a ZF2 `Segment` route; you can read up on those in the ZF2 manual. In such routes, a string preceded by a colon will indicate a named variable to capture: `/resource/:id` would capture an "id" value. You can have many named segments, and even optional segments.

1.2.2 Controllers and Helpers

Your controllers can be any PHP callable. In the examples, I use closures, but any callable is accepted. The callable will receive exactly one argument, the `Phlyty\App` instance.

From the App instance, you have the following helper methods available:

- `params()` returns a `Zend\Mvc\Router\RouteMatch` instance, from which you can then pull values. In the example in the previous paragraph, you can pull the "id" using `$app->params()->getParam('id', false)`.
- `request()` returns a `Zend\Http\PhpEnvironment\Request` instance. This gives you access to headers, query, post, cookie, files, env, and system parameters. In most cases, you use `getType($name, $default);` e.g. `$app->request()->getQuery('name', 'Matthew')` would retrieve the "name" query string value, using "Matthew" as the default.
- `response()` returns a `Zend\Http\PhpEnvironment\Response` instance. This allows you to manipulate response headers, and to set the response body.
- `flash($name, $message)` lets you both set and receive flash messages.
- `urlFor($route = null, array $params = [], array $options = [])` allows you to generate a URI based on the routes you've created. If you pass no arguments, it assumes it should use the current route. Otherwise, you must pass a route name; as such, it's good practice to name your routes. (Any `$params` you provide will be used to replace named segments in the route.)
- `pass()` tells the application to move on to the next matching route, if any.
- `redirect($uri, $status = 302)` will redirect. Hint: use `urlFor()` to generate the `$uri` value!
- `halt($status, $message = '')` halts execution immediately, and sends the provided message.
- `stop()` halts execution, sending the current response.
- `events()` accesses the composed event manager, allowing you to register listeners and trigger events.
- `event()` returns a `Phlyty\AppEvent` instance with the current route composed.
- `trigger` triggers an event.
- `view()` returns the view renderer, which should implement `Phlyty\View\ViewInterface`. You can call `setView()` to change the view implementation. Additionally, you can always instantiate and use your own view implementation.
- `viewModel()` returns a *ZendViewModelModelInterface*' implementation; by default, it's of type `Phlyty\View\MustacheViewModel`. This allows you to inject variables, set the template, etc. If you want to use an alternate view model, either directly instantiate it, or provide a prototype instance to `setViewModelPrototype()`.
- `render($template, $viewModel = [])` will render a template and/or a view model, and place the rendered content into the Response body.

CHAPTER 2

Routes

Routing in `Phlyty` is the act of matching *both* an HTTP request method *and* path information to the controller which will handle it.

Withing `Phlyty`, Zend Framework 2 routes are used. By default, ZF2's "Segment" route is used. Visit the Zend Framework 2 documentation for full documentation of the segment route.

At its most basic, the segment route takes literal paths interspersed with named captures of the form `:name`, called *segments*. The segment name must consist of alphanumeric characters *only*. Additionally, you can indicate *optional* captures using brackets ("[" and "]"). These two simple rule allow using segments in creative ways:

- `/calendar/event/:year-:month-:day.:format` would match "/calendar/event/2012-08-19.json", and capture year as "2012", month as "08", day as "19", and format as "json".
- `/news/:post[/:page]` would match both "/news/foo-bar" as well as "/news/foo-bar/3".

All that said, you may desire more flexibility at times.

2.1 Constraints and Defaults

For example, what if you want to add constraints to your named segments? As an example, what if "page", or "year", or "month", or "day" should only ever consist of digits?

What if you want to supply defaults for some values?

To do these things, create the ZF2 route manually, and then pass it to the appropriate HTTP-specific method of `Phlyty\App`. As an example, let's work with the "calendar" route we established above. We'll provide both constraints and defaults for the route.

```
use Phlyty\\App;
use Zend\Mvc\Router\Http\Segment as SegmentRoute;

$route = SegmentRoute::factory(array(
    'route' => '/calendar/event/:year-:month-:day[.:format]',
    'constraints' => array(
        'year'   => '20\d{2}',
        'month'  => '(0|1)\d',
        'day'    => '(0|1|2|3)\d',
        'format' => '(html|json|xml)',
    ),
    'defaults' => array(
        'format' => 'html',
    ),
));
```

```
$app = new App();

$app->get($route, function ($app) {
    // handle route here
})->name('calendar');
```

Note how we pass the `SegmentRoute` instance as the argument to `$app->get()`. This allows us to create a fully-configured, robust route instance with constraints and defaults, while still honoring the interface that `Phlyty\\App` offers.

You could extend this to provide tree routes, literal routes, and more; basically, any route type Zend Framework 2 provides may be used.

For more information on ZF2 routes, please visit the ZF2 routes documentation.

CHAPTER 3

Controllers

Controllers are simply any PHP callable. Controllers will receive exactly one argument, the `Phlyty\App` instance that invokes the controller.

Controllers can thus be:

- anonymous functions, closures, or lambdas
- named functions
- static class methods
- instance methods
- functors (classes defining `__invoke()`)

This also means you can define and configure your controllers where you want.

3.1 Anonymous Function

Anonymous functions are functions not assigned to a variable, and defined in-place. Using an anonymous function is perhaps the easiest way to define a controller.

```
$app->get('/', function ($app) {
    // do work here
});
```

3.2 Closures

Closures are anonymous functions that import variables from the current scope into the scope of the function. This is done using the `use` directive when declaring the function.

```
$config = include 'config.php';
$app->get('/', function ($app) use ($config) {
    // You can access $config now.
    // Do work here.
});
```

3.3 Lambdas

Lambdas are anonymous functions or closures that are assigned to a variable; this allows using them in multiple contexts, as well as passing them around by variable.

```
// As a normal lambda
$lambda = function ($app) {
    // Do work here.
};
$app->get('/', $lambda);

// As a closure
$config = include 'config.php';
$lambda = function ($app) use ($config) {
    // You can access $config now.
    // Do work here.
};
$app->get('/', $lambda);
```

3.4 Named Functions

You can also declare functions either in the global namespace or within a user-defined namespace, and pass the string function name.

```
namespace My
{
    function home($app)
    {
        // do work here
    }
}

$app->get('/', 'My\\home');
```

3.5 Static Class Methods

Static class methods may also be used. You may pass these either in the form of `[$className, $method]` or `ClassName::method`.

```
namespace My
{
    class Hello
    {
        public static function world($app)
        {
            // do work here...
        }
    }
}

// Using array callback notation
$app->get('/hello/:name', ['My\Hello', 'world']);
```

```
// Using string callback notation
$app->get('/hello/:name', 'My\Hello::world');
```

3.6 Instance Methods

A typical PHP instance method callback can be used. This is great for situations where you have configurable stateful behavior.

```
namespace My
{
    class Hello
    {
        protected $config;

        public function __construct($config)
        {
            $this->config = $config;
        }

        public static function world($app)
        {
            // do work here...
        }
    }
}

$config = include 'config.php';
$hello  = new My\Hello($config);

// Using array callback notation
$app->get('/hello/:name', [$hello, 'world']);
```

3.7 Functors

"Functors" are objects that define the magic method `__invoke`, and can thus be called as if they are a function. (Interesting trivia: this is basically how the PHP internal class `Closure` works.) In such an object, you'd simply have a single method that could act as a controller, the `__invoke()` method. You must instantiate a functor for it to work as such, however.

```
namespace My
{
    class Hello
    {
        protected $config;

        public function __construct($config)
        {
            $this->config = $config;
        }

        public static function __invoke($app)
        {
            // do work here...
```

```
        }
    }
}

$config = include 'config.php';
$hello  = new My\Hello($config);

// As a functor
$app->get('/hello/:name', $hello);
```

CHAPTER 4

Helpers

`Phlyty` ships with a number of built-in helper methods in the `Phlyty\App` class. These fall under roughly four categories:

- Workflow-related helpers (`halt`, `stop`, `pass`, `redirect`, `events`, `event`, `trigger`, `getLog`)
- HTTP-related helpers (`request`, `response`)
- Route-related helpers (`params`, `urlFor`)
- View-related helpers (`view`, `viewModel`, `render`, `flash`)

4.1 Workflow Helpers

Workflow helpers shape the flow of the application. They allow you to return from execution early, either because the response is ready, or because we know an error condition has occurred; redirect to another URI; pass on execution to another route and controller; or work with events.

`halt($status, $message='')` Halts execution immediately, setting the response status to `$status`, and, if `$message` is provided, setting the response body to that message. No further code will be executed in the controller following this call.

```
$name = $app->params('name', false);
if (!$name) {
    $app->halt(500, 'Missing name; cannot continue execution');
}
// do something with $name now...
```

`stop()` Halts execution, sending the response as it currently exists. You might call this if you wanted to return a file download, for instance.

```
$image = $app->params('image', false);
if ($image && file_exists($image)) {
    $stream = fopen($image, 'r');
    $out    = fopen('php://output', 'w');
    stream_copy_to_stream($stream, $out);
    $app->response()->setBody($out);
    $app->stop();
}
// show some error message here...
```

`pass()` Tells the application that no more processing of this controller should be done, but that it should continue iterating through routes to look for another one that matches the current URI.

```
$app->get('/:locale', function ($app) {
    $locale = $app->params()->getParam('locale', 'en_US');
    Locale::setDefault($locale);
    $app->pass();
});

$app->get('/[:locale]', function ($app) {
    // This matches the previous route, which means when pass() is
    // called by the previous controller, this route will be matched
    // and this controller invoked.
    //
    // Display home page
});
```

redirect($url, $status = 302) Sets the response status code to `$status` and the `Location` header to `$url`, and immediately halts execution and sends the response. Any code following the call in the controller will not be executed.

```
$app->get('/user/:username', function ($app) {
    $username = $app->params()->getParam('username', false);
    if (!$username) {
        $this->redirect('/login');
    }
    // Code below here will only execute if we did not redirect
});
```

events() Returns a `Zend\EventManager\EventManager` instance. This allows you to attach event listeners as well as trigger events. *See the section on events for more information*.

```
$app->events()->attach('route', function ($e) use ($app) {
    $route = $e->getRoute();
    if (!in_array($route->getName(), ['profile', 'comment', 'post']) {
        return;
    }

    // check if we have an authenticated user, and throw an exception
    // otherwise
    // ...
}, -10); // registering to execute after routing finishes
```

event() Returns a new `Phlyty\AppEvent` instance with the target set to the `Phlyty\App` instance, and the route populated with the currently matched route.

trigger($name, array $params = []) Trigger the named event, optionally passing parameters to compose in the `Phlyty\\AppEvent` instance.

```
$app->get('/', function ($app) {
    $app->trigger('homepage', $app->params()->getParams());
});
```

getLog() Gets the currently registered `Zend\Log\Logger` instance, lazy-loading one if none is present. You will need to attach writers to the log instance, and then invoke one or more logging methods.

```
$logger = $app->getLog()
$logger->addWriter('stream', [
    'stream'        => 'php://stderr',
    'log_separator' => "\n",
]);
$logger->info('This is an informational message');
```

4.2 HTTP-Related Helpers

A web application is really about receiving an HTTP request, deciding what to do with it, and returning an HTTP response back to the client. In `Phlyty\App`, the request and response objects help you with this.

`request()` Returns the request object. See the ZF2 Zend\Http\PhpEnvironment\Request documentation for more details.

```
// Getting query string (aka GET) parameters
$query  = $app->request()->getQuery();
$single = $app->request()->getQuery($name, $default);

// Getting POST parameters
$post   = $app->request()->getPost();
$single = $app->request()->getPost($name, $default);

// Getting headers
$headers = $app->request()->getHeaders();
$header  = $app->request()->getHeader($name, $default);
$value   = $header->getFieldValue();

// Getting ENV values
$values = $app->request()->getEnv();
$value  = $app->request()->getEnv($name, $default);

// Getting $_SERVER values
$values = $app->request()->getServer();
$value  = $app->request()->getServer($name, $default);

// Get the URI
$uri = $app->request()->getUri(); // Zend\Uri\Uri object
$uri = $app->request()->getUriString(); // string

// Get the Cookie header
$cookies = $app->request()->getCookie();
$cookie  = $cookie[$cookieName];

// Testing request type
$app->request()->isXmlHttpRequest();
$app->request()->isGet();
$app->request()->isPost();
$app->request()->isPut();
$app->request()->isDelete();
$app->request()->isOptions();
$app->request()->isPatch();

// The base url should be auto-detected, but you can also set it explicitly
$app->request()->setBaseUrl('/~matthew/sites/foo');
```

`response()` Returns the response object. See the ZF2 Zend\Http\PhpEnvironment\Response documentation for more details.

```
// Setting a header
$app->response()->getHeader()->addHeaderLine($name, $value);

// Setting the status code
$app->response()->setStatusCode(201);
```

```
// Setting the response body
$app->response()->setContent($content);
```

4.3 Route-Related Helpers

The main purpose of a microframework is to map URL paths to their handlers. Once you have, there are two principal route-related activities you will be performing in most requests: you will need to access parameters matched in the URL, and you will need to generate URLs based on the routes you've defined.

`params()` Returns the Zend\Mvc\Router\RouteMatch instance returned by the route that matched the URL. The API is roughly as follows:

```
$params = $app->params();
$single = $params->getParam('single', 'default value');
$array  = $params->getParams();
```

`urlFor($route = null, array $params = [], array $options = [])` Generates a URL based on the named `$route`, using `$params` to fill in named segments in the URL, and any route-specific generation `$options` provided. If `$route` is not present, it will assume the current matched route; if `$params` is not present, any defaults used when creating the route will be used.

If a base URL is present in the request, it will be prepended to the generated URL.

```
$app->get('/blog[/:year[/:month[/:day]]]', function ($app) {
    // ...
})->name('blog-by-date');

$url = $app->urlFor('blog-by-date, [
    'year'  => 2012,
    'month' => '08',
    'day'   => 21,
]); // "/blog/2012/08/21"
```

4.4 View-Related Helpers

The goal of a controller is to produce a response to return to the client. In most cases, that response will contain some content. In web applications, this is typically referred to as a "View" (from the design pattern "Model-View-Controller", or "MVC"). Typically, the "view" is functionality that renders a template.

`Phlyty` provides helpers for setting and retrieving the view object that will be used to render templates, as well as a method for actually rendering a named template using the current view object. Other helpers allow you to set a "view model" – an object that encapsulates the data you wish to represent in the view – as well as retrieve instances of that view model. Finally, `Phlyty` provides functionality for setting and retrieving "flash" messages – messages you wish to present in the view layer – but most likely on a subsequent page (typically following a redirect – for instance, to indicate that a record was updated).

- View-related helpers (`view`, `viewModel`, `render`, `flash`)

`setView(Phlyty\View\ViewInterface $view)` Sets the view object. The `ViewInterface` defines simply a method `render($template, $viewModel = []).`

`view()` Retrieves the current view object, which should implement the `ViewInterface`. By default, this is `Phlyty\View\MustacheView`, which is an implementation that utilizes phly_mustache, a Mustache implementation.

`setViewModelPrototype($model)` Allows specifying a prototype object to use for view models. The object provided will be *cloned* when retrieved later.

```
$model = new stdClass();
$app->setViewModelPrototype($model);
```

`viewModel()` Retrieves a *clone* of the currently registered view model object. By default, if none has been registered, an instance of `Phlyty\View\MustacheView` is provided.

```
$model = $app->viewModel();
$model->foo = $bar;
$model->bindHelper('bar', function () {
    return $this->__escaper()->escapeHtml($this->foo) . '!';
});
```

`render($template, $viewModel = [])` Renders a named `$template` using the currently registered view object, and passing the specified `$viewModel`, if any. It is up to the view object to resolve the template name to a resource it may use, and to determine how to utilize the `$viewModel` provided.

Once the content is rendered, it's injected as the content of the response object.

```
$app->render('pages/foo', $model);
```

`flash($name, $message = null)` Create or retrieve a flash message. Flash messages expire after a single "hop"; in other words, after more than one page visit, the flash message will disappear. If you pass just a `$name` to `flash()`, it will attempt to retrieve the message; passing a `$message` to it will set it.

By default, the `MustacheViewModel` composes the `$app` instance, allowing you to retrieve flash messages. As an example, you could do the following to create a view variable for retrieving a formatted message:

```
$model = $app->viewModel();
$model->bindHelper('messages', function () {
    $message = $this->__app()->flash('foo');
    if (empty($message)) {
        return '';
    }
    return sprint(
        '<div class="flash">%s</div>',
        $this->__escaper()->escapeHtml($message)
    );
});
```

For more about views, *see the section on Views*.

CHAPTER 5

Events

`Phlyty\App` composes a Zend\EventManager\EventManager instance. This allows you to trigger events, and attach listeners to events. It also allows the application to trigger events – and for you as a developer to write listeners for those events.

To attach to an event, you simply call `attach()`; the first argument is the event name, the second is a valid PHP callable, usually a closure.

```
$events = $app->events();

$events->attach('do', function ($e) {
    echo "I've been triggered!";
});

$events->trigger('do'); // "I've been triggered!"
```

The `EventManager` allows you to specify the *priority* at which a listener is triggered. This allows you to order listeners – but, more importantly, it allows you to decide when a listener is triggered in relation to the *default* listeners. This is important when you consider that the application triggers a number of events; many of these have listeners registered by default in the application, *at the default priority*. This means:

- If you register with a *higher* (positive) priority, the listener will be triggered *earlier*.
- If you register with a *lower* (negative) priority, the listener will be triggered *later*.

The priority argument comes after the listener argument.

```
$events = $app->events();

$events->attach('do', function ($e) {
    echo "Default priority\n";
});
$events->attach('do', function ($e) {
    echo "Low priority\n";
}, -100);
$events->attach('do', function ($e) {
    echo "High priority\n";
}, 100);

$events->trigger('do');

/* output:
High priority
Default priority
```

```
Low priority
\*/
```

5.1 Defined Events

As noted previously, the application triggers several events, some of which have default handlers defined.

begin Triggered at the very beginning of `run()`.

route Triggered during routing. A default route listener is defined and registered with default priority.

halt Triggered when `halt()` is invoked.

404 Triggered if no route matches the current URL.

501 Triggered if a controller bound to a route cannot be invoked (usually because it's not a valid callable).

500 Triggered when an exception is raised anywhere during `run()`.

finish Triggered immediately prior to sending the response.

5.2 Use Cases

You may attach to any of these events in order to alter the application work flow.

5.2.1 Error Pages

As an example, if you wish to display a 404 page for your application, you might register a listener as follows:

```
$app->events()->attach('404', function ($e) {
    $app = $e->getTarget();
    $app->render('404');
});
```

You could do similarly for 500 and 501 errors.

5.2.2 Caching

You could implement a quick-and-dirty caching layer using the "begin" and "finish" events.

```
// Assume we've instantiated $cache prior to this
$app->events()->attach('begin', function ($e) use ($cache) {
    $app  = $e->getTarget();
    $req  = $app->request();
    if (!$req->isGet()) {
        return;
    }

    $url  = $req->getUriString();
    $data = $cache->get($url);
    if (!$data) {
        return;
    }
```

```
    $app->response()->setContent($data);
    $app->response()->send();
    exit();
}, 1000); // register at high priority

$app->events()->attach('finish', function ($e) use ($cache) {
    $app  = $e->getTarget();
    if (!$app->request()->isGet()) {
        return;
    }
    if (!$app->response()->isOk()) {
        return;
    }

    $url  = $app->request()->getUriString();
    $data = $app->response()->getContent();
    $cache->save($url, $data);
}, -1000); // register at low priority
```

The above would look for a cache entry matching the current URI, but only if we have a GET request. If a cache entry is found, we set the response content with the data, send it, and exit immediately.

Otherwise, when the request is finished, we check if we had a successful GET request, and, if so, save the response body into the cache using the current request URI.

CHAPTER 6

Views

Views are the presentation layer of the application. Typically, you will use a templating engine to create the presentation, though `Phlyty` makes no assumptions about what or how that engine works. It only requires that you provide a class implementing `Phlyty\View\ViewInterface` that provides a `render` method; it is then up to you to pass the two arguments to that method on to your templating engine in order to obtain a representation.

If the above does not suit your needs, you can, of course, always instantiate your own view objects and use them as you see fit in the application.

6.1 The ViewInterface

The `ViewInterface` is a trivial definition:

```
namespace Phlyty\View;

interface ViewInterface
{
    /**
    * Render a template, optionally passing a view model/variables
    *
    * @param  string $template
    * @param  mixed $viewModel
    * @return string
    \*/
    public function render($template, $viewModel = []);
}
```

6.2 Mustache Integration

`Phlyty` uses *phly_mustache <http://weierophinney.github.com/phly_mustache* by default, and provides some convenience classes and functionality around this templating engine.

First, it provides `Phlyty\View\MustacheView`. This is a simple extension of `Phly\Mustache\Mustache` that alters the `render()` method to make it suit the `ViewInterface`.

Second, it provides `Phlyty\View\MustacheViewModel`. This class can simplify creation of your view models by providing several convenience features. First, it composes the application instance, as well as an instance of `Zend\Escaper\Escaper`. These allow you to access any application helpers you might want when providing your view representation, as well as context-specific escaping mechanisms (for instance, to escape CSS, JavaScript, HTML attributes, etc.). Additionally, it provides a convenience method, `bindHelper()`, which allows you to create

closures as model properties, and have them bound to the model instance; this allows the closures to have access to the model via `$this`, and thus access the application and escaper instances, as well as all properties.

The application instance is available via the pseudo-magic method `__app()`, and the escaper via `__escaper()`.

```
$model = $app->viewModel();
$model->route = 'bar';
$model->bindHelper('link', function () {
    return $this->__app()->urlFor($this->route);
});
```

The template might look like this:

```
You should <a href="{{link}}">visit</a>
```

CHAPTER 7

Api docs

- API docs are available.

CHAPTER 8

Getting help

- Issue Tracker

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*