

# Gyro essentials: Routing, controllers, and views

This tutorial shows how to link URLs to your content.

## Defining a route

In Gyro, every possible URL must be defined, using so-called routes. Routes map URLs invoked by the user to functions in your PHP code. These functions are called "actions".

Routes are defined on Controller-classes. Controllers are created directly in the directory `controllers` right beneath your application directory. The file name is of form `[name].controller.php` and contains a class named `[Name]Controller`. This class is called a controller class. Note the lower- and uppercase variants of `[name]`, this is important. The controller class must inherit from class `ControllerBase`.

Each route is constructed with

- an URL to map
- a controller instance responsible for handling the call
- the name of the action to invoke
- an optional array of parameters to specialize things like caching, or access control checking for this special route

Note you don't pass the name of a function to be called as third parameter, but the name of an action. The function itself must be called `action_[name of action]`.

Unless some other popular PHP frameworks, the mapping of URLs to PHP code to call is not done automatically. This is: Just dropping functions on a controller will do nothing. You always must declare a route.

## Example

Let's jump right into the code to get things clear. As example we create a Playground-Controller which should show a little message when URL `/playground/` is invoked. Create a file `/app/controller/playground.controller.php` and type in the following code :

```
1 <?php
2 class PlaygroundController extends ControllerBase {
3     /**
4      * Return array of Route instances which are
5      * handled by this controller
6      *
7      * @return array Array of Routes
8      */
9     public function get_routes() {
10         return array(
11             new ExactMatchRoute('playground/', $this, 'playground_index')
12         );
13     }
14
15     /**
16      * Display the playground
17      */
18     public function action_playground_index(PageData $page_data) {
19         $view = ViewFactory::create_view(
20             ViewFactory::CONTENT,
21             'playground/index',
22             $page_data
23         );
24     }
25 }
```

```

23     );
        $view->render();
25     }
    }
}

```

Second, create a file `/app/view/templates/default/playground/index.tpl.php` containing the following code:

```

<?php
2 $title = 'Welcome to the playground';
  // Set title and meta-description in html head tag
4 $page_data->head->title = $title;
  $page_data->head->description =
6   'The playground is open for everyone younger than 12 years';
  // Create a breadcrumb
8 $page_data->breadcrumb = WidetBreadcrumb::output('Playground');
  ?>
10 <h1><?=$title?></h1>
    <p>The playground is open!</p>

```

Let's step through the code, to see what's going on there.

First, the class `ExactMatchRoute` defines a route that requires the URL invoked to be exactly like the one defined as the first parameter - `playground/` in this case. Some more complex routes will be discussed later. Second, the route will delegate to the action named `playground_index` on the controller `PlaygroundController`, since we passed `$this` as instance.

Due to the Gyro naming conventions, the action maps to the function `action_playground_index()`. Each action gets an instance of `PageData` passed as first parameter. `PageData` is used to allow easy communication between different layers of an application, without coupling them too tight. The action `action_playground_index` in our example doesn't do much except of creating a view and delegating rendering to it.

## Views and templates

Views are created using the `ViewFactory`. Since we are dealing with the content of a page here, we tell the factory to create a content view by passing `IViewFactory::CONTENT` as type. The second parameter defines the template and the third are parameters, which depend on the view type. A content view requires a `PageData` instance to be passed.

The template name passed is mapped to a template file on the file system. Template files are located in `view/templates/[language]/` and must end with `.tpl.php`. Gyro will look up templates in the following order:

- Application's template directory of current locale: `/app/view/templates/[language]/`
- Application's default template directory: `/app/view/templates/default/`
- Module's template directory of current locale: `/[module name]/view/templates/[language]/`
- Module's default template directory: `/[module name]/view/templates/default/`
- Core's template directory of current locale: `/core/view/templates/[language]/`
- Core's default template directory: `/core/view/templates/default/`

The modules are reverse iterated in the order they were loaded, so the module loaded last is checked first, and the module loaded first is checked last. This allows overloading templates of one module by another. For each module localized templates and default templates are checked before stepping to the next.

Within a content view template, the `PageData` instance is available as `$page_data`, and it can be used to set the title for the document we create, along with the meta description. As you can see in the example, we can set the breadcrumb, too.

View templates can be seen as html files containing PHP code. That's in the example above we can simply write some html tags like `<h1>` or `<p>`. If you need to execute php code, you must wrap in `<?php ?>`.

## The `<?= ?>` short tags

Gyro also allows you to use the short form `<?= ?>` to just output something using html escaping. This was introduced for convenience and to make templates easier to overlook.

It was inspired by the idea, that most people avoid typing too much, which makes them use

```
1 <?php print $title; ?>
```

rather than

```
1 <?php print htmlentities($title, ENT_QUOTES, {Encoding}); ?>
```

Additionally, template people may not be so much into PHP they even know how to correctly use `htmlentities()` at all. Using Gyro they now have an even shorter but safe way of saying "print":

```
1 <?=$title?>
```

Note that if you explicitly want to output html code, you need to use `<?php print ...; ?>`, since `<?= ?>` translates brackets to their html entities, so `<` will become `&lt;`, and `>` will become `&gt;`.

## Parameterized Routes: Flexible and typesafe

It was mentioned above that there are more advanced routes than `ExactMatchRoute`. The one probably used the most is `ParameterizedRoute`, which can be configured to contain arbitrary, yet type checked parameters.

Let's have a look into the code right away, to see how `ParameterizedRoute` is used.

```
1 <?php
  class PlaygroundController extends ControllerBase {
3     /**
     * Return array of Route instances which are
5     * handled by this controller
     *
7     * @return array Array of Routes
     */
9     public function get_routes() {
        return array(
11         new ExactMatchRoute('playground/', $this, 'playground_index'),
        new ParameterizedRoute(
13         'playground/enter/{name:s}/{age:ui}',
            $this, 'playground_enter')
15         );
    }
17
    /**
19     * Display the playground
     */
21     public function action_playground_index(PageData $page_data) {
        $view = ViewFactory::create_view(
23         ViewFactory::CONTENT,
            'playground/index',
25         $page_data
        );
27         $view->render();
    }
29
    /**
31     * Let someone enter the playground
```

```

33  */
34  public function action_playground_enter(PageData $page_data, $name, $age) {
35      if ($age > 12) {
36          $page_data->status = new Status(tr('No entrance above 12 years'));
37          return self::ACCESS_DENIED;
38      }
39
40      $view = ViewFactory::create_view(
41          IViewFactory::CONTENT,
42          'playground/enter',
43          $page_data
44      );
45      $view->assign('name', $name);
46      $view->assign('age', $age);
47      $view->render();
48  }
49  }

```

Parameters on a `ParameterizedRoute` are declared using curly brackets like this:

```
{name[: type[: parameters]]}
```

The following types are supported:

- `i`: integer
- `ui`: unsigned integer
- `ui>`: integer greater than 0
- `s`: string (default). You can specify a length like so: `s:2` - String of two ANSI characters
- `e`: enum. Usage is `e:value1,value2,value3,...,valueN`

`ParameterizedRoute` also supports optional element, see the class documentation for details.

In the above example a route was defined that takes a name, and an age, where the age must be a positive integer. The `ParameterizedRoute` type checks the URL, and would return a 404 Not Found if type check fails.

The following URLs would be successful:

- `/playground/enter/john/11`
- `/playground/enter/mary/121`
- `/playground/enter/peter+paul+mary/1`

The following URLs would cause errors:

- `/playground/enter/john/mary`
- `/playground/enter/mary/`
- `/playground/enter/mary/0`
- `/playground/enter/`

Since `ParameterizedRoute` does the type checking, you can rely on the data passed. `$age` of the function `action_playground_enter()` is ensured to be an unsigned integer greater than zero.

## Passing parameters to views

As you may have noticed in above example, we can pass variables to the view using the method `assign()`.

```
1 $view->assign('name', $name);
```

The variables are accessible in the view using the name you passed as first parameter. To demonstrate this, we will complete our example with the according view for `playground_enter`. Place the following code at `/app/view/templates/default/playground/enter.tpl.php`:

```

1 <?php
  $title = "A new kid's on the playground";
3 // Set title and meta-description in html head tag
  $page_data->head->title = $title;
5 // Create a breadcrumb
  $page_data->breadcrumb = WidgetBreadcrumb::output(
7   array(
      WidgetActionLink::output('Playground', 'playground_index'),
9     'Enter'
    )
11 );
  ?>
13 <h1><?=$title?></h1>
  <p><?=$age?> year old <?=$name?> has entered the playground!</p>

```

Note how `$name` and `$age` are used on the view.

## Secured sites using https

You may declare a route to use `https` by prefixing it with `https://` like so:

```
... new ExactMatchRoute('https://playground/secure/', ...) ...
```

This of course works with every route, not only for `ExactMatchRoute`.

Gyro will force the route to use a secured connection. If a user navigates to `http://playground/secure/`, she will get redirected to `https://playground/secure/`.

You can disable secure routes completely by setting the constant `APP_ENABLE_HTTPS` to `false`. All routes declared with `https://` then will be treated as normal, non-secure routes.

## Referencing existing routes: action mapping

Mapping a URL to an action is one thing. Gyro also allows you to map an action to an URL. This can be done using the so-called `ActionMapper`. The `ActionMapper` class has two important static function: `get_path` and `get_url`. `get_url` returns an absolute URL, whilst `get_path` will return a relative URL - unless the action URL points to another host or another scheme (e.g. `https`), in which case it will return an absolute URL, too.

Mapping actions to URL allows to change application's URLs later on without having to change code - except the route, of course.

Both function take two parameters:

- action: The name of an action.
- params: Additional paramters

The simplest case is to retrieve the URL for an `ExactMatchRoute`. We only pass the action name:

```
1 print ActionMapper::get_path('playground_index'); // outputs /playground/
```

For parameterized routes we can pass an associative array with paramter names as keys:

```

1 print ActionMapper::get_path(
  'playground_enter',
3   array('name' => 'John', 'age' => 10)
); // outputs /playground/enter/John/10

```

Instead of an array, you may also pass an object, in this case `ActionMapper` reads the object's properties. If you pass an `IDataObject`, you may also use a short form for the action. Gyro will try to prefix the action with the model name:

```

$model = DB::get_item('users', 'id', 1);
2 print ActionMapper::get_path(
    'view',
4  $model
); // Will look for action "users_view"!

```

On top of the `ActionMapper` is `WidgetActionLink`. It will return a html anchor tag for an action's URL. `WidgetActionLink` always uses the `get_path` function on `ActionMapper`.

We already saw an example for that:

```

1 print WidgetActionLink::output('To the Playground', 'playground_index');

```

The first parameter is the text for the anchor, the second the action, and the optional third the parameters for the route. The above code will return

```

1 <a href="/playground/">To the Playground</a>

```

If you pass an instance implementing `ISelfDescribing` as first parameter, the widget will use `get_title()` to retrieve the anchor's text. This makes it possible to invoke the widget like this:

```

1 $model = DB::get_item('users', 'id', 1);
  print WidgetActionLink::output($model, 'view', $model);

```

The above is equivalent to

```

$model = DB::get_item('users', 'id', 1);
2 print WidgetActionLink::output($model->get_title(), 'view', $model);

```