Routing Engine

Overview

Our first example (getting-started) wasn't too hard to swallow, was it? If you like a little more flavor in your Fat-Free soup, insert another route before the `$f3->run()` command:

```
$f3->route('GET /about',
    function() {
        echo 'Donations go to a local charity... us!';
    }
);
```

You don't want to clutter the global namespace with function names? Fat-Free recognizes different ways of mapping route handlers to OOP classes and methods:

```
class WebPage {
    function display() {
        echo 'I cannot object to an object';
    }
}

$f3->route('GET /about','WebPage->display');
```

HTTP requests can also be routed to static class methods:

```
$f3->route('GET /login','Controller\Auth::login');
```

> Stuck with a **404 Not Found** error? Check your server configuration (routing-engine#sample-apache-configuration).

You can also watch a video (https://youtu.be/XGldwy1pmU0) that goes over many of the points in this routing user guide.

Routes and Tokens

As a demonstration of Fat-Free's powerful domain-specific language (DSL), you can specify a single route to handle different possibilities:

```
$f3->route('GET /brew/@count',
    function($f3) {
        echo $f3->get('PARAMS.count').' bottles of beer on the wall.';
    }
);
```

This example shows how we can specify a token `@count` to represent part of a URL. The framework will serve any request URL that matches the `/brew/` prefix, like `/brew/99`, `/brew/98`, etc. This will display `'99 bottles of beer on the wall'` and `'98 bottles of beer on the wall'`, respectively. Fat-Free will also accept a page request for `/brew/unbreakable`. (Expect this to display `'unbreakable bottles of beer on the wall'`.) When such a dynamic route is specified, Fat-Free automagically populates the global `PARAMS` array variable with the value of the captured strings in the URL. The `$f3->get()` call inside the callback function retrieves the value of a framework variable. You can certainly apply this method in your code as part of the presentation or business logic. But we'll discuss that in greater detail later.

Notice that Fat-Free understands array dot-notation. You can use `PARAMS['count']` regular notation instead in code, which is prone to typo errors and unbalanced braces. In views and templates, the framework permits `@PARAMS.count` notation which is somewhat similar to Javascript. (We'll cover views and templates later.)

Here's another way to access tokens in a request pattern:

```
$f3->route('GET /brew/@count',
    function($f3,$params) {
        echo $params['count'].' bottles of beer on the wall.';
    }
);
```

You can use the asterisk (`*`) to accept any URL after the `/brew` route - if you don't really care about the rest of the path:

```
$f3->route('GET /brew/*',
    function() {
        echo 'Enough beer! We always end up here.';
    }
);
```

An important point to consider: You will get Fat-Free (and yourself) confused if you have both `GET /brew/@count` and `GET /brew/*` together in the same application. Use one or the other. Another thing: Fat-Free sees `GET /brew` as separate and distinct from the route `GET /brew/@count` . Each can have different route handlers.

IMPORTANT: All route handlers are automatically passed the framework instance and the route tokens. See here (base#route-handler).

Named Routes

When you define a route, you can assign it a name. Use the route name in your code and templates instead of a typed url. Then if you need to change your urls to please the marketing overlords, you only need to make the change where the route was defined. The route names must follow php variable naming rules (no dots, dashes nor hyphens).

Let's name a route:

```
$f3->route('GET @beer_list: /beer', 'Beer->list');
```

The name is inserted after the route VERB (`GET` in this example) preceeded by an `@` symbol, and separated from the URL portion by a colon `:` symbol. You can insert a space after the colon if that makes it easier to read your code (as shown here).

To redirect the visitor to a new URL, call the named route inside the `reroute()` method like:

```
// a named route is a string value
$f3->reroute('@beer_list'); // note the single quotes
```

If you use tokens in your route, F3 will replace those tokens with their current value. If you want to change the token's value before calling reroute, pass it as the 2nd argument:

```
$f3->route('GET @beer_list: /beer/@country', 'Beer->bycountry');

// a set of key-value pairs is passed as argument to named route
$f3->reroute('@beer_list(@country=Germany)');

// if more than one token in your route is needed
$f3->route('GET @beer_village_list: /beer/@country/@village', 'Beer->byvillage');
$f3->reroute('@beer_village_list(@country=Germany,@village=Rhine)');
```

Remember to `urlencode()` your arguments if you have characters that do not comply with RFC 1738 guidelines for well-formed URLs.

Named routes in templates

To access the named route in a template, you can pass the route name to the alias filter:

```
<a href="{{ 'beer_list' | alias }}">view beer list</a>
```

If you want to build a link to a named route that contains tokens, you can pass additional parameters to the alias filter:

```
<a href="{{ 'beer_village_list', 'country=Germany,village=Rhine' | alias }}">view bee
r list from Rhine, Germany</a>
```

This also works with variables, which looks like `{{ @name, 'a=5,b='.@id | alias }}`. You only need to set or overwrite tokens in named routes that you need to change. All other tokens are resolved automatically, based on the current route.

If you need to replace wildcard tokens in your route (i.e. `GET @complex:/resize/@format/*/sep/*`), use a numeric index to specify it's new value, for instance: `{{'complex','format=20x20,2=foo/bar,3=baz.gif'|alias}}`.

To generate URLs in your controller code, see the alias (base#alias) and build (base#build) methods.

Dynamic Web Sites

Wait a second - in all the previous examples, we never really created any directory in our hard drive to store these routes. The short answer: we don't have to. All F3 routes are virtual. They don't mirror our hard disk folder structure. If you have programs or static files (images, CSS, etc.) that do not use the framework - as long as the paths to these files do not conflict with any route defined in your application - your Web server software will deliver them to the user's browser, provided the server is configured properly.

PHP 5.4's Built-In Web Server

PHP's latest stable version has its own built-in Web server. Start it up using the following configuration:

```
php -S localhost:80 -t /var/www/
```

The above command will start routing all requests to the Web root `/var/www`. If an incoming HTTP request for a file or folder is received, PHP will look for it inside the Web root and send it over to the browser if found. Otherwise, PHP will load the default `index.php` (containing your F3-enabled code).

Sample Apache Configuration

If you're using Apache, make sure you activate the URL rewriting module (mod_rewrite) in your apache.conf (or httpd.conf) file. You should also create a .htaccess file containing the following:

```
RewriteEngine On

RewriteRule ^(app|dict|ns|tmp)\/|\.ini$ - [R=404]

RewriteCond %{REQUEST_FILENAME} !-l
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule .* index.php [L,QSA]
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization},L]
```

The script tells Apache that whenever an HTTP request arrives and if no physical file (`!-f`) or path (`!-d`) or symbolic link (`!-l`) can be found, it should transfer control to `index.php`, which contains our main/front controller, and which in turn, invokes the framework.

The `.htaccess file` containing the Apache directives stated above should always be in the same folder as `index.php`.

You also need to set up Apache so it knows the physical location of `index.php` in your hard drive. A typical configuration is:

```
DocumentRoot "/var/www/html"
<Directory "/var/www/html">
    Options -Indexes +FollowSymLinks +Includes
    AllowOverride All
    Require all granted
</Directory>
```

In case you just put your fat-free project into a sub-folder of an existing document root, some Apache configurations possibly need a defined `RewriteBase` as well in your `.htaccess` file. If the app is not working, or the default route `/` works, but `/test` maybe fails, try to add the base path:

```
RewriteEngine On
RewriteBase /fatfree-project/
```

If you're developing several applications simultaneously, a virtual host configuration is easier to manage:

```
NameVirtualHost *
<VirtualHost *>
    ServerName site1.com
    DocumentRoot "/var/www/site1"
    <Directory "/var/www/site1">
        Options -Indexes +FollowSymLinks +Includes
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
<VirtualHost *>
    ServerName site2.com
    DocumentRoot "/var/www/site2"
    <Directory "/var/www/site2">
        Options -Indexes +FollowSymLinks +Includes
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

Each `ServerName` (`site1.com` and `site2.com` in our example) must be listed in your `/etc/hosts` file. On Windows, you should edit `C:/WINDOWS/system32/drivers/etc/hosts` . A reboot might be necessary to effect the changes. You can then point your Web browser to the address `http://site1.com` or `http://site2.com` . Virtual hosts make your applications a lot easier to deploy.

Alternate Apache configuration

If `mod_rewrite` is not available on your server or if you want to improve performance a bit, you can take benefit of the FallbackResource (https://httpd.apache.org/docs/2.4/mod/mod_dir.html#fallbackresource) directive, available in Apache 2.4 and higher.

In that case, just add the following line in your `.htaccess` file or in your VirtualHost configuration:

```
FallbackResource /index.php
```

In case your web app is running in a subfolder of the server root, don't forget to modify the directive accordingly:

```
FallbackResource /fatfree-project/index.php
```

Notice: There's currently a bug (https://bz.apache.org/bugzilla/show_bug.cgi?id=52403) that skips the directive when the requested URI ends with `.php` and is located at the application root:

- `http://localhost/fatfree-project/foo.php` => error 404
- `http://localhost/fatfree-project/foo/bar.php` => OK

Sample Nginx Configuration

For Nginx servers, here's the recommended configuration (replace ip_address:port with your environment's FastCGI PHP settings):

```
server {
    root /var/www/html;
    location / {
        index index.php index.html index.htm;
        try_files $uri /index.php?$query_string;
    }
    location ~ \.php$ {
        fastcgi_pass ip_address:port;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

Sample Lighttpd Configuration

Lighttpd servers are configured in a similar manner:

```
$HTTP["host"] =~ "www\.example\.com$" {
    url.rewrite-once = ( "^/(.*?)(\?.+)?$"=>"/index.php/$1?$2" )
    server.error-handler-404 = "/index.php"
}
```

Sample IIS Configuration

Install the URL rewrite module (https://www.iis.net/downloads/microsoft/url-rewrite) and the appropriate .NET framework corresponding to your Windows version. Then create a file named `web.config` in your application root with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <rule name="Application" stopProcessing="true">
          <match url=".*" ignoreCase="false" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" matchType="IsFile" ignoreCase="false" neg
ate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" ignoreCase="fals
e" negate="true" />
          </conditions>
          <action type="Rewrite" url="index.php" appendQueryString="true" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
</configuration>
```

Rerouting

So let's get back to coding. You can declare a page obsolete and redirect your visitors to another site:

```
$f3->redirect('GET|HEAD /obsoletepage', '/newpage');
```

which is the same as

```
$f3->route('GET|HEAD /obsoletepage',
    function($f3) {
        $f3->reroute('/newpage');
    }
);
```

If someone tries to access the URL `http://www.example.com/obsoletepage` using either HTTP GET or HEAD request, the framework redirects the user to the URL: `http://www.example.com/newpage` as shown in the above example. You can also redirect the user to another site, like `$f3->reroute('http://www.anotherexample.org/');` .

Rerouting can be particularly useful when you need to do some maintenance work on your site. You can have a route handler that informs your visitors that your site is offline for a short period.

HTTP redirects are indispensable but they can also be expensive. As much as possible, refrain from using `$f3->reroute()` to send a user to another page on the same Web site if you can direct the flow of your application by invoking the function or method that handles the target route. However, this approach will not change the URL on the address bar of the user's Web browser. If this is not the behavior you want and you really need to send a user to another page, in instances like successful submission of a form or after a user has been authenticated, Fat-Free sends an `HTTP 302 Found` header. For all other attempts to reroute to another page or site, the framework sends an `HTTP 301 Moved Permanently` header.

Triggering a 404

At runtime, Fat-Free automatically generates an HTTP 404 error whenever it sees that an incoming HTTP request does not match any of the routes defined in your application. However, there are instances when you need to trigger it yourself.

Take for instance a route defined as `GET /dogs/@breed`. Your application logic may involve searching a database and attempting to retrieve the record corresponding to the value of `@breed` in the incoming HTTP request. Since Fat-Free will accept any value after the `/dogs/` prefix because of the presence of the `@breed` token, displaying an `HTTP 404 Not Found` message programmatically becomes necessary when the program doesn't find any match in our database. To do that, use the following command:

```
$f3->error(404);
```

ReST: Representational State Transfer

Fat-Free's architecture is based on the concept that HTTP URIs represent abstract Web resources (not limited to HTML) and each resource can move from one application state to another. For this reason, F3 does not have any restrictions on the way you structure your application. If you prefer to use the Model-View-Controller (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) pattern, F3 can help you compartmentalize your application components to stick to this paradigm. On the other hand, the framework also supports the Resource-Method-Representation (http://www.peej.co.uk/articles/rmr-architecture.html) pattern, and implementing it is more straightforward.

Here's an example of a ReST interface:

```
class Item {
    function get() {}
    function post() {}
    function put() {}
    function delete() {}
}

$f3=require('lib/base.php');
$f3->map('/cart/@item','Item');
$f3->run();
```

Fat-Free's `$f3->map()` method provides a ReST interface by mapping HTTP methods in routes to the equivalent methods of an object or a PHP class. If your application receives an incoming HTTP request like `GET /cart/123`, Fat-Free will automatically transfer control to the object's or class' `get()` method. Similarly, a `POST /cart/123` request will be routed to the `Item` class' `post()` method.

Mapped methods can be prefixed using the PREMAP (quick-reference#PREMAP) variable.

Note: Browsers do not implement the HTTP `PUT` and `DELETE` methods in regular HTML forms. These and other ReST methods (`HEAD`, and `CONNECT`) are accessible only via AJAX calls to the server. They can however be tunneled through a `POST` request by setting the `_method` parameter to the desired HTTP verb.

If the framework receives an HTTP method that's not implemented by a class, it generates an `HTTP 405 Method Not Allowed` error. F3 automatically responds with the appropriate headers to HTTP `OPTIONS` method requests. The framework will not map this request to a class.

> **Notice:** In case you plan to build a whole ReST API, you probably need to get around the possibility of fetching single items, whole collections of items and saving and updating records, which does not always fit into a single class or the given HTTP methods. It's always best to plan your API before you start coding. Here is a very good and **free** eBook about Web API Design (https://pages.apigee.com/ebook-web-api-design-registration.html?utm_source=hpc&utm_medium=website&utm_campaign=ebook) from apigee that could be useful.

The F3 Autoloader

Fat-Free has a way of loading classes only at the time you need them, so they don't gobble up more memory than a particular segment of your application needs. And you don't have to write a long list of `include` or `require` statements just to load PHP classes saved in different files and different locations. The framework can do this automatically for you. Just save your class files (one class per file) in a folder (for example "myclassfiles") and set the autoload variable to point to that folder:

```
$f3->set('AUTOLOAD','myclassfiles/');
```

Important: The class name and file name must be identical so the framework can autoload your class. If your class is named `BarBaz` , your file must be named `BarBaz.php` . Lowercase `barbaz.php` will also work. (see below (routing-engine#autoloader-case-handling) for details).

When you call your class or method using `$obj=new Barbaz;` , Fat-Free Framework will search for the file barbaz.php in the path(s) specified in the autoloader variable. Once it finds the file, it will include in using PHP's `require` command. This is how autoloading works.

The `AUTOLOAD` path is searched from the location of your index.php file. You can set your `AUTOLOAD` variable using an absolute path i.e. `/var/www/mywebsite.com/myclassfiles/` or using a relative path as viewed from the location of your index.php file `../myclassfiles/` , when index.php is located in `/var/www/mywebsite.com/` .

You can also have multiple autoload paths. If you have your classes divided into different folders, you can instruct the framework to autoload the appropriate class when a static method is called or when an object is instantiated. Modify the `AUTOLOAD` variable to point to multiple folders:

```
$f3->set('AUTOLOAD','admin/autoload/; user/autoload/; default/');
```

Working with Namespaces

`AUTOLOAD` allows class hierarchies to reside in similarly-named subfolders, so if you want the framework to autoload a namespaced class that's invoked in the following manner:

```
$f3->set('AUTOLOAD','autoload/');
$obj=new Gadgets\iPad;
```

You can create a folder hierarchy that follows the same structure. Assuming `/var/www/html/` is your Web root, then F3 will look for the class in `/var/www/html/autoload/gadgets/ipad.php`. The file `ipad.php` should have the following minimum code:

```
namespace Gadgets;
class iPad {}
```

Remember: All directory names in Fat-Free must end with a slash. You can assign a search path for the autoloader as follows:

```
$f3->set('AUTOLOAD','main/;aux/');
```

> **NB:** Namespaces are here to help you organize your code. If you use them, you can decide to do it with F3 autoloader or not. If you use the autoloader, you have to create a folder for each namespace. If you don't use the autoloader, you can store your files however you like, keeping in mind that you will need to include each of them manually.

Routing to a Namespaced Class

F3, being a namespace-aware framework, allows you to use a method in namespaced class as a route handler, and there are several ways of doing it. To call a static method:

```
$f3->set('AUTOLOAD','classes/');
$f3->route('GET|POST /','Main\Home::show');
```

The above code will invoke the static `show()` method of the class `Home` within the `Main` namespace. The `Home` class must be saved in the folder `classes/main/home.php` for it to be loaded automatically.

If you prefer to work with objects:

```
$f3->route('GET|POST /','Main\Home->show');
```

will instantiate the `Home` class at runtime and call the `show()` method thereafter.

Autoloader case handling

On case-sensitive systems like UNIX the class gets auto-loaded only if the class file and path have the same case as the namespaced class **or** if they are lowercase. E.g:

If the autoloaded class is `Main\Home`, possible filenames are `Main/Home.php` or `main/home.php`.

⇒ `main\Home.php`, `Main\hoME.php` or `MAIN\HOME.php` will not load!

If you need to define a custom case handling, you can set the `AUTOLOAD` variable as an array of a path and a custom function. Let's say that all your filenames are uppercase. Then instead of defining `$f3->set('AUTOLOAD','classes/'`, you should define:

```
$f3->set('AUTOLOAD',array('classes/',function($class){
  return strtoupper($class);
}));
```

Event Handlers

F3 has a couple of routing event listeners that might help you improve the flow and structure of controller classes. Say you have a route defined as follows:

```
$f3->route('GET /','Main->home');
```

If the application receives an HTTP request matching the above route, F3 first instantiates `Main`, but *before* executing the `home()` method, the framework looks for a method in this class named `beforeRoute()`. If present, F3 runs the code contained in the `beforeRoute()` event handler before transferring control to the method specified in the route, in our example the `home()` method. Once the method is terminated, the framework then looks for an `afterRoute()` event handler that is called if present. The `beforeroute()` and `afterroute()` event handlers are common to a given class. It means if you have defined different routes using different methods of the same class, e.g. `'GET /login','User->login'` and `'GET /logout','User->logout'`, both routes will share the same `beforeroute()` and `afterroute()` event handlers.

Create `beforeRoute()` and `afterRoute()` functions in your base controller class, and have them apply to every request. Of course, you can override them in the child controllers by defining custom `beforeRoute()` and `afterRoute()` handlers in the child classes. One can also create `beforeRoute()` and `afterRoute()` in a child class, and inherit the front controller handlers by calling `parent::beforeRoute();` or `parent::afterRoute();` in the children.

Dynamic Route Handlers

Here's another F3 goodie:

```
$f3->route('GET /products/@action','Products->@action');
```

If your application receives a request for, say, `/products/itemize`, F3 will extract the `'itemize'` string from the URL and pass it on to the `@action` token in the route handler. F3 will then look for a class named `Products` and execute its `itemize()` method.

Dynamic route handlers may have various forms:

```
// static method
$f3->route('GET /public/@genre','Main::@genre');
// object mode
$f3->route('GET /public/@controller/@action','@controller->@action');
```

F3 triggers an `HTTP 404 Not Found` error at runtime if it cannot transfer control to the class or method associated with the current route, i.e. there is no defined class and/or method to match the requested route.

AJAX and Synchronous Requests

Routing patterns may contain *modifiers* that instruct the framework to base its routing decision on the type of the HTTP request:

```
$f3->route('GET /example [ajax]','Page->getFragment');
$f3->route('GET /example [sync]','Page->getFull');

; routes.ini style
POST /formsubmit [ajax] = Form->process_post_via_ajax
```

The first statement will route the HTTP request to the `Page->getFragment()` callback only if an `X-Requested-With: XMLHttpRequest` header (AJAX object) is received by the server. If an ordinary (synchronous) request is detected, F3 will simply drop down to the next matching pattern, and in this case it executes the `Page->getFull()` callback.

If no modifiers are defined in a routing pattern, then both AJAX and synchronous request types are routed to the specified handler.

Route pattern modifiers are also recognized by `$f3->map()` (base#map).

Routing in CLI mode

Web syntax

If you want to run a specific route from a command line tool, a shell script, the console or as cron job, you can emulate a HTTP GET request by the following command:

```
cd /path/to/test/suite
php index.php /my-awesome-route
```

NB: query strings are also supported:

```
php index.php /my-awesome-route?foo=bar
```

Shell syntax

Instead of requesting a URI, you can pass shell arguments & options. They will be automatically converted to an emulated HTTP GET request, which makes it easy to build a shell tool in the same way you would build a web application.

The following mapping rules apply for conversion:

- space-separated arguments map to path components
- short-form (aka flags) and long-form options map to query string arguments
- short-form options can be combined
- options can be passed in any order (before, between or after arguments)

Here are some examples:

- `php index.php test` maps to `GET /test`

- `php index.php log show --limit=50 --full` maps to `GET /log/show?limit=50&full=`
- `php index.php cache clear -f -v -i -n=23` maps to `GET /cache/clear?f=&v=&i=&n=23`
- the following are all equivalent to the previous request:
  - `php index.php cache clear -fvi -n=23`
  - `php index.php cache clear -fvin=23`
  - `php index.php cache -fvin=23 clear`
  - `php index.php -fvin=23 cache clear`
  - `php index.php -fvi cache clear -n=23`

CLI options are accessed via the `$_GET` global variable.

Here are two slightly different approaches for processing options:

```
// syntax #1 (validating data, mixing short and long forms)
$opts=[
    'force' => $f3->exists('GET.f'), // -f
    'limit' => abs((int)$f3->get('GET.limit')) ?: 20, // --limit=N (default: 20)
    'verbose' => $f3->exists('GET.v') || $f3->exists('GET.verbose'), // -v OR --verbo
se
];

// syntax #2 (not validating data, long-form only)
$opts=$_GET+[
     'force'=>0, // --force
     'limit'=>20, // --limit=N (default: 20)
     'verbose'=>0, // --verbose
];
```

Defining routes

CLI routes are defined in the same way as web routes, the only difference being that they can be restricted to CLI mode using the `[cli]` modifier. E.g:

```
[routes]
GET /log/show       [cli] = CLI\Log->show
GET /log/clear      [cli] = CLI\Log->clear
GET /cache/clear    [cli] = CLI\Cache->clear
GET /help           [cli] = CLI\Help->index
GET /help/@command  [cli] = CLI\Help->@command
```

NB: the CLI (quick-reference#CLI) variable tells if the request comes from the CLI or not.

Mocking

CLI routes can be mocked just like web routes. Beware that the shell-syntax is not allowed here:

```
$f3->mock('GET /log/show?limit=50 [cli]');
```

← 1. Getting Started (getting-started) 3. Framework Variables → (framework-variables)