Cache Engine

Keeping Javascript and CSS on a Healthy Diet

Client-Side Caching

PHP Code Acceleration

Bandwidth Throttling

Optimization

Cache Engine

Caching static Web pages - so the code in some route handlers can be skipped and templates don't have to be reprocessed - is one way of reducing your Web server's work load so it can focus on other tasks. You can activate the framework's cache engine by providing a third argument to the `$f3->route()` method. Just specify the number of seconds before a cached Web page expires:

```
$f3->route('GET /my_page','App->method',60);
```

Here's how it works. In this example, when F3 detects that the URL `/my_page` is accessed for the first time, it executes the route handler represented by the second argument and saves all browser output to the framework's built-in cache (server-side). A similar instruction is automatically sent to the user's Web browser (client-side), so that instead of sending an identical request to the server within the 60-second period, the browser can just retrieve the page locally. The framework uses the cache for an entirely different purpose - serving framework-cached data to other users asking for the same Web page within the 60-second time frame. It skips execution of the route handler and serves the previously-saved page directly from disk. When someone tries to access the same URL after the 60-second timer has lapsed, F3 will refresh the cache with a new copy.

Web pages with static data are the most likely candidates for caching. Fat-Free will not cache a Web page at a specified URL if the third argument in the `$f3->route()` method is zero or unspecified. F3 conforms to the HTTP specifications: only GET and HEAD requests can be cached.

Here's an important point to consider when designing your application. Don't cache Web pages unless you understand the possible unwanted side-effects of the cache at the client-side. Make sure that you activate caching on Web pages that have nothing to do with the user's session state.

For example, you designed your site in such a way that all your Web pages have the menu options: `"Home"`, `"About Us"`, and `"Login"`, displayed when a user is not logged into your application. You also want the menu options to change to: `"Home"`, `"About Us"`, and `"Logout"`, once the user has logged in. If you instructed Fat-Free to cache the contents of `"About Us"` page (which includes the menu options), it does so and also sends the same instruction to the HTTP client. Regardless of the user's session state, i.e. logged in or logged out, the user's browser will take a snapshot of the page at the session state it was in. Future requests by the user for the `"About Us"` page before the cache timeout expires will display the same menu options available at that time the page was initially saved. Now, a user may have already logged in, but the menu options are still the same as if no such event occurred. That's not the kind of behavior we want from our application.

Furthermore, when using included files in cached files, e.g. `require_once('../inc/account_header.php')` , the relative paths used won't work anymore, as the cached files are not stored in the original files directory. You'll either need to provide the full path to your included files; or tell F3 to store the cached files in a directory php can find; or add the path where the cached files are stored to the php include path (this can be done in the php.ini file).

Some pointers:

- Don't cache dynamic pages. It's quite obvious you don't want to cache data that changes frequently. You can, however, activate caching on pages that contain data updated on an hourly, daily or even yearly basis.For security reasons, the framework restricts cache engine usage to HTTP `GET` routes only. It will not cache submitted forms!Don't activate the cache on Web pages that at first glance look static. In our example, the "About Us" content may be static, but the menu isn't.
- Activate caching on pages that are available only in ONE session state. If you want to cache the `"About Us"` page, make sure it's available only when a user is not logged in.
- If you have a RAMdisk or fast solid-state drive, configure the `CACHE` global variable so it points to that drive. This will make your application run like a Formula 1 race car.

Note: Don't set the timeout value to a very long period until you're ready to roll out your application, i.e. the release or production state. Changes you make to any of your PHP scripts may not have the expected effect on the displayed output if the page exists in the framework cache and the expiration period has not lapsed. If you do alter a program that generates a page affected by the cache timer and you want these changes to take effect immediately, you should clear the cache by erasing the files in the cache/ directory (or whatever path the `CACHE` global variable points to). F3 will automatically refresh the cache if necessary. At the client-side, there's little you can do but instruct the user to clear the browser's cache or wait for the cache period to expire.

PHP needs to be set up correctly for the F3 cache engine to work properly. Your operating system timezone should be synchronized with the date.timezone setting in the `php.ini` file.

Similar to routes, Fat-Free also allows you to cache database queries. Speed gains can be quite significant, specially when used on complex SQL statements that involve look-up of static data or database content that rarely changes. Activating the database query cache so the framework doesn't have to re-execute the SQL statements every time is as simple as adding a 3rd argument to the F3::sql command - the cache timeout. For example:

```
$db->exec('SELECT * from sizes;',NULL,86400);
```

If we expect the result of this database query to always be `Small` , `Medium` , and `Large` within a 24-hour period, we specify `86400` seconds as the 2nd argument so Fat-Free doesn't have to execute the query more than once a day. Instead, the framework will store the result in the cache, retrieve it from the cache every time a request comes in during the specified 24-hour time frame, and re-execute the query when the timer lapses.

The SQL data mapper also uses the cache engine to optimize synchronization of table structures with the objects that represent them. The default is `60` seconds. If you make any changes to a table's structure in your database engine, you'll have to wait for the cache timer to expire before seeing the effect in your application. You can change this behavior by specifying a third argument to the data mapper constructor. Set it to a high value if you don't expect to make any further changes to your table structure.

```
$user=new DB\SQL\Mapper($db,'users',NULL,86400);
```

By default, Fat-Free's cache engine is disabled. You can enable it and allow it to auto-detect APC, WinCache or XCache. If it cannot find an appropriate backend, F3 will use the filesystem, i.e. the `tmp/cache/` folder:

```
$f3->set('CACHE',TRUE);
```

Disabling the cache is as simple as:

```
$f3->set('CACHE',FALSE);
```

If you wish to override the auto-detection feature, you can do so - as in the case of a Memcached back-end which F3 also supports:

```
$f3->set('CACHE','memcache=localhost:11211');
```

You can also use the cache engine to store your own variables. These variables will persist between HTTP requests and remain in cache until the engine receives instructions to delete them. To save a value in the cache:

```
$f3->set('var','I want this value saved',90);
```

`$f3->set()` method's third argument instructs the framework to save the variable in the cache for a 90-second duration. If your application issues a `$f3->get('var')` within this period, F3 will automatically retrieve the value from cache. In like manner, `$f3->clear('var')` will purge the value from both cache and RAM. If you want to determine if a variable exists in cache, `$f3->exists('var')); returns one of two possible values: FALSE if the framework variable passed does not exist in cache, or an integer representing the time the variable was saved (Un*x time in seconds, with microsecond precision).

Keeping Javascript and CSS on a Healthy Diet

Fat-Free also has a Javascript and CSS compressor available in the Web plug-in (web#minify). It can combine all your CSS files into one stylesheet and all your Javascript files into one single script in order to drastically reduce the number of HTTP requests needed by a Web page. Reducing the number of HTTP requests to your Web server results in faster page loading and a better UX.

To put this easy improvement in place, you first need to prepare your HTML templates to take advantage of this feature.

For the CSS, you could start like this:

```
<link rel="stylesheet" type="text/css" href="/minify/css?files=typo.css,grid.css" />
```

And do the same with your Javascript files:

```
<script type="text/javascript" src="/minify/js?files=dialog.js,main.js"></script>
```

Of course we need to set up a route to handle the necessary call to the Fat-Free CSS/Javascript compressor:

```
$f3->route('GET /minify/@type',
    function($f3, $args) {
        $path = $f3->get('UI').$args['type'].'/';
        $files = preg_replace('/(\.+\/)/','',$_GET['files']); // close potential hack
ing attempts
        echo Web::instance()->minify($files, null, true, $path);
    },
    3600*24
);

// @type will make `PARAMS.type` variable base point to the correct path
// make sure you organize your files to be minified into sub-folders, per type, i.e.
/ui/css/ /ui/js
// minify will grab each file specified in the querystring var named 'files' and comb
ine into 1 output
// Save the minified file in F3 cache for 24 hours. future requests for this route wi
ll use cached version
```

Warning! You have to make sure that `$_GET['files']` is sanitized and does not contain `../` chars, which could potentially open a security issue in your application. This was fixed in v3.5.2, but you need handle this yourself in any previous version.

And that's all there is to it! `minify()` reads each file (`typo.css` and `grid.css` in our CSS example, `dialog.js` and `main.js` in our Javascript example), strips off all unnecessary whitespaces and comments, combines all of the related items as a single Web page component, and attaches a future expiry date so the user's Web browser will cache the data and not hit the server for every url request. It's important that the `PARAMS.type` variable base points to the correct path. Otherwise, the URL rewriting mechanism inside the compressor won't find the CSS/Javascript files.

Client-Side Caching

In our examples, the framework sends a future expiry date to the client's Web browser so any request for the same CSS or Javascript file will use the cached version from the user's local computer file storage. When a file request is made to the webserver, F3 will check if the route (in this example, CSS or Javascript files) has already been cached. In the minify example above, the route we specified has a cache refresh period of `3600` seconds (1 hour) times 24, equaling 24 hours. Additionally, if the Web browser sends an `If-Modified-Since` request header and the framework sees the cache hasn't changed, F3 just sends an `HTTP 304 Not Modified` response so no bandwidth is wasted and the content loads faster. Without the `If-Modified-Since` header, Fat-Free sends the cached file if available. Otherwise, the route's code is executed with each request.

Tip: If you're not modifying your Javascript/CSS files frequently (as it would be if you're using a Javascript library like jQuery, MooTools, Dojo, etc.), consider adding a cache timer to the route leading to your Javascript/CSS minify handler (3rd argument of F3::route()) so Fat-Free doesn't have to compress and combine these files each time the request is received.

PHP Code Acceleration

Want to make your site run even faster? Fat-Free works best with either Alternative PHP Cache (APC), XCache, or WinCache. These PHP extensions boost performance of your application by optimizing your PHP scripts (including the framework code).

Bandwidth Throttling

A fast application that processes all HTTP requests and responds to them at the shortest time possible is not always a good idea - specially if your bandwidth is limited or traffic on your Web site is particularly heavy. Serving pages ASAP also makes your application vulnerable to Denial-of-Service (DOS) attacks. F3 has a bandwidth throttling feature that allows you to control how fast your Web pages are served. You can specify the bandwidth to use when a request is served with route definitions like this:

```
$f3->route('GET /login','\Backoffice\Login->handler', 0, 64);
```

In this example, the framework will serve the GET requests to /login at a maximum rate of 64 KiB/s (`64 * 1024` bytes per second).

Bandwidth throttling at the application level can be particularly useful for login pages. Slow responses to dictionary attacks is a good way of mitigating this kind of security risk.

← 6. Plug-Ins (plug-ins)

8. Unit Testing → (unit-testing)