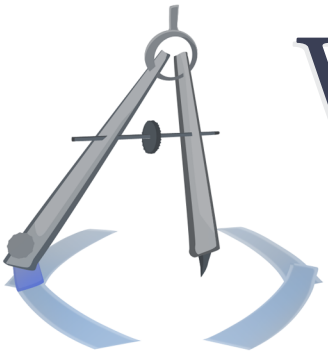


Website Architecture

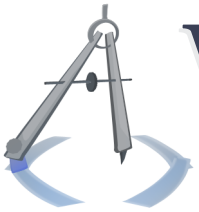
Technical Report

MiniArc

A minimal intro to the PHP website-building library.

Michael Serritella

Summer 2012



Website Architecture

Technical Report | MiniArc

Intro to MiniArc

This is an introductory introduction into MiniArc – a prelude to the intro. But really, it's probably all you need. MiniArc is a PHP library that handles the ins & outs of writing websites (mostly reading & analyzing input; writing output in fancy ways). This document only goes as far as students in Website Architecture class would care about on 6/22/2012.

You can peruse the online documentation at **here** (for now).

Download the library here (for now): **zip**, **tar.gz**, or **tar.bz2**.

Overall structure

MASivArc > MiniArc > (HTPPS, Sanitization, Validation, ...)
MASivArc is a collection of libraries; only MiniArc is ~done and relevant to you right now.

MiniArc has the following modules you would use now:

Validation Validates user input; checks data against patterns & formats

Sanitization Sanitizes user input; conforms data to patterns & formats

Request Analyzes the HTTP request; provides derived metrics

Response Helps to write the HTTP resonse, including headers, compression, translations

Email Helps to compose and send email in a civilized manner

Debug Helps to log state and configure the working environment for debugging

HTPPS Prints markup, so pretty

Where your code goes

MiniArc does not restrict where your application code goes, such as the index.php and hello.php files, nor your own libraries; you can put them anywhere. Your code can have any kind of architecture/relationship.

MiniArc mostly provides base classes with static methods. You can derive these for application-specific use and, since almost all members are static, you can use them from anywhere and there are no real dependencies you need to satisfy. Additionally, no module of MiniArc is dependent upon any other module, so you may choose to include only some of its modules in your project.

Base classes, and you

For almost all of the modules (Sanitization, Validation, Request, Response, Debug), there is a base-class-and-subclass structure, with the subclass already stubbed out for your customization. For example:

```php
<?php
class SanitizeBase {
        // my stuff
        public static function OnlyNumeric(&$Variable) { ... }
}

class Sanitize extends SanitizeBase {
        // your stuff
}
?>
```

Due to PHP's static/inheritance feature, you can do this from your application: **`Sanitize::OnlyNumeric($userInput)`**;

However, you should probably not do this, just because there are better ways to organize your application code.

The members of base classes typically have generic and obnoxiously good names (see

`OnlyAlphanumericPunctuationWithWhitespace()`, et. al.). People would often complain that they didn't like typing these in their applications. This is because they didn't know what the deal. You should only be typing these names a finite number of times – perhaps only once.

You should define functions with application-relevant semantics in these derived classes, which call functions from the base class. For example:

PHP

```php
<?php
// in Validate
public static function GPA($Value) {
        return self::FloatIn($Value, 0, 4);
}
?>
```

So you can do this from your application:

PHP

```php
<?php
if (Validate::GPA($userGivenGPA)) { ... }?>
```

Instead of this:

```php
<?php
if (Validate:: FloatIn($userGivenGPA, 0, 4)) { ... }
```

You may see that this is much more flexible, readable, and professional. Cleaner code leads to cleaner thinking; don't litter your application with phrases with no meaning and constant values everywhere (like 0 & 4). You would look like a juvenile hack. This can also save you a ton of time if the definition of GPA ever changes (4.5?); you change it once instead of N times.

Validate & Sanitize, powers & abilities

All functions in `Validate` (Base) operate in the same way. All functions in `Sanitize` (Base) operate the in same way. Let's see it real quick.

Functions in `Validate` take an unvalidated value as their first parameter, then maybe some additional parameters, and return true or false; true if it passes the test and is valid, false otherwise.

Functions in `Sanitize` take an unsanitized value as their first parameter, then maybe some additional parameters, and then they modify the value in-place and also return it. The value is passed by reference, which means you can't pass it the result of an expression, like `Sanitize::Float(5 + 7)`. In addition, if it makes sense, the functions have parameters called `$TraverseArrays` and `$TraverseObjects`. If the given variable is a structure, these will apply the sanitization operation to the structure's members recursively.

Request & Response, powers & abilities

Using **Request** (Base), you can gather information about the requested URI, the request method (GET/POST/etc.), any parameters sent via GET/POST/cookies, and read and analyze HTTP headers. There are very many methods, grouped into clusters, with diverse purposes. You can see some example methods from each of several important clusters and then explore the rest yourself:

Method & URI `MethodIsPOST()`, `ResourceFilesystemPath()`, `IsInternallyRedirected()`

G/P/C fields `GET()`, `GETFieldExists()`, `GETFieldsWithPrefix()`, `FieldsWithPrefix()`

File uploading `POSTFile()`, `POSTFiles()`, `PUTFile()`

Headers `HTTPHeader()`, `HTTPHeaders()`

User/client & their preferences `UAIsFirefox()`, `UAIsMobile()`, `VisitorPrefersMediaType_XHTML()`, `VisitorIPAddress()`

Server metrics `ServerIPAddress()`, `ServerDocumentRoot()`, `ServerUnixTimestamp()`

Using **Response** (Base), you can write custom HTTP headers and transform the response body, including with gzip compression. There are quite a few methods and clusters here, as well. Some examples:

Headers/generic `WriteHTTPResponseCode()`, `WriteHTTPHeader()`, `BuildHTTPHeader()`

Headers/specific `HTTP_ContentType_JPEG()`, `HTTP_ContentDisposition()`, `DeterCaching()`

Serving binary `ServeFile()`, `ServeBinaryString()`, `ServeStream()`

Output manipulation `CompressOutput()`, `ThrottleBandwidth()`, `SaveOutput()`, `SaveOutput_Stop()`, `TranslateOutputCharset()`

Email

Using the **EmailSender** (Base), you can compose emails using nice and civilized methods instead of encoding all of your custom options as MIME headers and cramming the string into the last parameter of PHP's single email function, **mail()**. **EmailSender** is object-oriented, and each object corresponds to one email, which may be sent and re-sent to multiple people. Static methods are used for instantiation and also for queueing large sending operations. The class is natural enough; explore it in the documentation.

The HTPPS

The HTPPS is a kinda-revolutionary system which helps in composing markup in a manner that is robust, flexible, and painstakingly pretty. Basically, you have this object, called a pretty printer, which goes like this:

PHP

```php
<?php
$page = HTPPS_Catalogue::GetPP('HTML 4.01 Strict');
// or just 'HTML' or other strings
$page->OpenDocument();
// ...
$page->OpenDiv(true, array('class'=>'MyDivClass'));
$page->WriteSpan('Hello');
$page->CloseTag();
?>
```

Sure, it looks crazy. But the HTPPS allows you to use new types of software architectures in websites and serve a more correct and efficient markup document, including varying the document type (such as by the user's expressed preferences) without modifying application code. It is software engineering brought to Web. We'll see it in a larger document.

You can always check out the documentation, play around for

five minutes, and get a feel for what it does. Its intent is to help you write correct code, not enforce correctness in all places (which would be a performance nightmare and practically useless in production code). So you can put a table-row tag in the middle of nowhere, for example; it won't stop you.

Read the documentation for `HTPPS_iWebPagePP`, which is perhaps the "keystone" class of the hierarchy; it outlines the methods available for writing tags within Web pages.