

Website Architecture

Technical Report

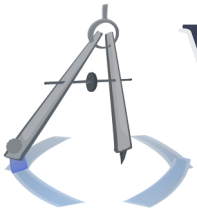
The HTTPPS

Documentation for the library which solves the problem of writing Web pages.

Michael Serritella

Summer 2010





Website Architecture

Technical Report | The HTTPPS

I Intro to the HTTPPS

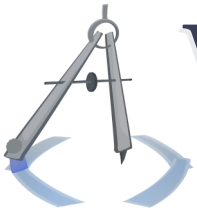
The Hot Text Pretty Printer Suite is a PHP library which stands as a layer of abstraction between the application logic and the composition of markup-language code. Using the HTTPPS, application writers make semantically-driven calls on a pretty-printer object instead of calling the built-in `print()` function. The object manages the necessary tags, including opening, closing, and writing them properly and with proper escape sequences, among other things. This layer of abstraction affords the opportunity to change the markup language, including its version, without changing the application logic, in addition to other features. Last but not least, the code of the final document is perfectly clean and perfectly formatted.

Basic features:

- Makes you want to listen to "P.Y.T."
- Produces perfectly valid, perfectly formatted and indented markup.
- Insulates applications against changes in both the grammar and elements of markup languages.
- Insulates applications against problems due to content negotiation; serves different pages to those who want HTML, XHTML, XHTML Basic, etc.
- Extensible architecture allows for application-specific customization of how tags are written and easy integration of library updates.
- Encourages object-oriented site design.
- Provides a simple interface for Web widgets to output their representations to Web pages or XML files.
- By producing perfect markup, makes the browser's rendering task as easy as possible, which should save significant CPU and power resources.

The HTTPPS was written by Mike Serritella and is published by MASiv Productions, LLC; an alpha was created in January 2010, and version 1.0 was finalized in July 2010.





Website Architecture

Technical Report | The HTTPPS

2 Code examples

Here are a few of the most demonstrative examples for the HTTPPS. In each of them, the `$page` variable has already been declared as a Web-page pretty printer. This is not always the case - the pretty printer could be for XML - and there are some opportunities for customization. But we will see those later.

In the coming sections, we will see six examples:

News headlines How to write a news headline with a byline; this illustrates some of the basic functions, including those of tag containment.

Opening a page How to open a page using a function which automates the most common case.

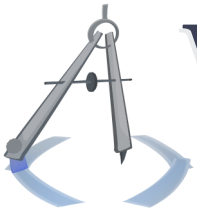
Opening a page with advanced links and metadata Demonstrates some of the options you have when writing a document `<head>` manually, including easy linking to RSS feeds, including CSS files with extra options, and meta tags.

Writing lists How to write ordered/unordered lists and definition lists, including one-stop functions for each.

Writing a table The HTTPPS has a comprehensive set of table-writing functions, including functions for features of tables that almost no one uses (e.g. writing a table footer). More importantly, the functions offset some of the annoyance of writing tables by closing any necessary tags before starting a new tag. For instance, when opening a table cell, you can set a parameter to ensure that it's opened in a proper context - within a `<tr>` - and the HTTPPS will close tags until the proper context exists. Similarly, a `<tr>` may be guaranteed to appear within a `<table>` or `<tbody>`.

Writing a form Like its table-writing functions, the HTTPPS has a set of form-writing functions which offer great convenience. Some of the most elaborate functions are those which write groups of form controls, including selecting any of them which you want to be initially selected.





Website Architecture

Technical Report | The HTTPPS

2.1 News headlines

PHP

```
<?php

// Open the outermost div.
$page->OpenDiv(true, array("class"=>"NewsEntry"));
    // (#0) End with a newline and indent further.
    // (#1) Give an associative array of attributes.

// Write a div that contains the article's title.
$page->WriteDiv($articleTitle, array("class"=>"Title"));

// Write a div that contains the article's date.
$page->WriteDiv($articleDate, array("class"=>"Date"));

// Open a div for the byline.
$page->OpenDiv(true, array("class"=>"Byline"));

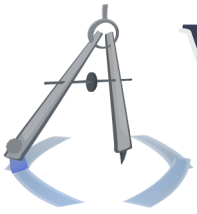
// Write the "By " text.
$page->Write("By ");

// Write the author's name, with a link to his/her page.
$page->WriteLink($authorPageURI, $authorName);

// Close the byline div and outermost div.
$page->CloseTags(2);

?>
```





Website Architecture

Technical Report | The HTTPPS

2.2 Opening a page

PHP

```
<?php

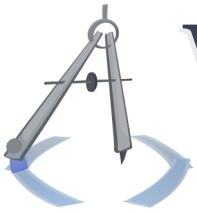
$page->WriteHeadAndOpenBody("A Page Title",
    "images/myfavicon.ico",
    array("mainStyles.css", "extraStyles.css"),
    array("scripts.js"),

    // Flush the transmission after sending
    // the <head>, which is a recommended
    // browser optimization.
    true,

    // Attributes for the <body>
    array("class"=>"ContactUs"));

?>
```





Website Architecture

Technical Report | The HTTPPS

2.3 Opening a page with advanced links and metadata

PHP

```
<?php

$page->OpenHead();

$page->WriteMeta_ContentType("text/html", "UTF-8");
$page->WriteMeta_Keywords("shoes socks");
$page->WriteMeta_Robots("noindex, nofollow");

$page->WriteTitle("Shoe Store");

$page->IncludeCSS("defaultStyles.css");

$page->IncludeCSS("alternateStyles.css",
                 null, // No special media.
                 "Pastel Styles",
                 true); // Is an alternate.

$page->IncludeCSS("printStyles.css", "print");

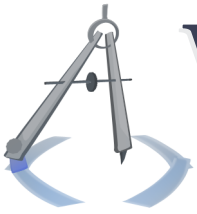
$page->LinkToRSSFeed("feeds/newShoes.rss", "New Shoes!");

$page->CloseTag();
$page->FlushTransmission();
$page->OpenBody();

// Declare that we are in <body>;
// unique in its specificity;
// used by more obscure parts of the HTTPPS.
$page->BeginDocumentContentTags();

?>
```





Website Architecture

Technical Report | The HTTPPS

2.4 Writing lists

PHP

```
<?php

$page->OpenUnorderedList();

$page->WriteListItem("Item One");

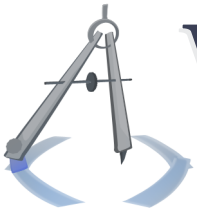
$page->OpenListItem();
$page->WriteSpan("Item Two:");
$page->Write(" ");
$page->WriteSpan("Subtitle");
$page->CloseTag();

$page->CloseTag();

$page->WriteDefinitionList(null, // These three are a common triple
                           null, // and you can almost always omit
                           null, // them in other functions. They
                               // describe the attributes of the
                               // list element.
                           array("Stapler"=>"Noun; fastens documents",
                                "Speaker"=>"Noun; Emits sound"));

?>
```





Website Architecture

Technical Report | The HTTPPS

2.5 Writing a table

PHP

```
<?php

// Open a table.
$page->OpenTable(null, null, null, // Same as with the definition list.
                "Table Caption Text");

// Open a row.
$page->OpenTableRow();

// Open a cell.
$page->OpenTableCell();
$page->Write("Hello, ");

// Open a cell with a column span; no need to close the prior cell.
$page->OpenTableCell(2);
$page->Write("World!");

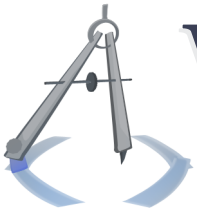
// Open a row; no need to close the prior row.
$page->OpenTableRow();

// Open a cell with a column span.
$page->OpenTableCell(3, // Column span.
                    null, // No row span.
                    true, // Ensure context.
                    array("class"=>"UnifiedRow")); // Attributes
$page->Write("~ ~ ~ ~ ~ ~ ~ ~ ~ ~");

// CloseTag() may be used here, but this ensures all tags
// through <table> are closed. Optionally, you could call
// CloseTag() with parameters which do the same thing.
$page->CloseTable();

?>
```





Website Architecture

Technical Report | The HTTPPS

2.6 Writing a form

PHP

```
<?php

// Open a form.
$page->OpenHTMLForm("processingPage.php", "get", array("id"=>"MyForm"));

// Write a check box and an accompanying label.
$page->WriteHTMLForm_CheckBox("IsCitizen",
                              "1",          // Value if checked.
                              false,
                              array("id"=>"ACheckBox_IsCitizen"));
$page->WriteHTMLForm_Label("ACheckBox_IsCitizen", "U.S. Citizen?");

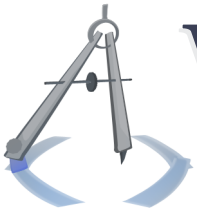
// Write a group of radio buttons.
$page->WriteHTMLForm_RadioButtons("MaritalStatus",
                                  array("Single"=>"Are you single?",
                                        "Married"=>"Are you married?",
                                        "Divorced"=>"Are you divorced?"),
                                  "Single",
                                  true,
                                  true);

// (#0) Form-data name.
// (#1) Array of form-data values and their corresponding labels.
// (#2) The initially-checked value; null if none.
// (#3) Whether to escape the label texts according to the markup
//       language.
// (#4) Whether to separate the controls with a line break in the
//       rendering.

// Close the form; analogous to CloseTable().
$page->CloseHTMLForm();

?>
```





Website Architecture

Technical Report | The HTTPPS

3 Features

The HTTPPS offers a strong set of features that are a dream for both the client and the software engineer. To the best of the author's knowledge, none of its strongest features exist in any other product.

3.1 Production of perfectly clean markup

Using the HTTPPS, it is quite easy to produce markup that is perfectly formed.

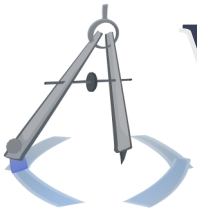
For the user, this means that the browser will have the easiest job possible in rendering the page. This impacts the CPU resources, RAM, and electric power necessary to render the page. At the time of this writing, the most popular sites on the Internet are riddled with errors. It is completely expected that there are hundreds of errors and hundreds of warnings per each page of these most popular sites. This puts a strain especially on mobile devices, and it is unbelievably wasteful to shift this cost to the millions of consumers who load these pages millions of times per day; the pages should be written correctly (*once*).

Most sites on the Internet have vastly incorrect markup and are thus not compliant with the recommended standards of Web technologies. This makes it more difficult for browsers to accommodate them, and this includes the growing market of alternative-media browsers, including browsers for mobile devices, browsers for the vision-impaired, and any other device that processes a Web page in order to render it differently (e.g. a search engine, which would likely produce more correct results). Standards compliance is important so that the whole world knows how to read a Web page, not just the testers in the author's office.

For the site author, this perfect markup usually means a reduction in bloat - unnecessary increase in the size of a page's code. There are likely no objective studies of page bloat on popular sites, but popular sites probably have an average of 10-75% bloat. Page bloat must account for a similar burden for all users of popular sites, and for the site author, this means that an overly large page must be transmitted to the user, which directly corresponds to operating costs.

Aside from a (very) likely decrease in page bloat and increase in standards compliance, the user's improved experience due to efficiency will translate to happier customers, and the perfect markup will seem extremely professional (if not magic) to anyone who is interested enough to look. Writing perfect markup is a responsible choice for both standards compliance and energy efficiency on a global scale.





Website Architecture

Technical Report | The HTTPPS

3.2 Abstraction of markup language choice

Multiple classes in the HTTPPS implement the same interface, which allows interoperability between the application and many different pretty printers. Some of the pretty printers are for HTML, and some are for XHTML. The application does not need to know which it is using, though it may find out. One class of pretty printers may even switch markup languages after it has been instantiated, allowing for late binding after some business logic has made an educated decision.

The Web is divided into two major markup languages: HTML and XHTML. XML is also a common language, but it is not commonly used for Web pages. The HTTPPS can write in all three languages. Each of these languages serves different communities of users, and each language continues to evolve, sometimes quite independently of the others. A site author may want to write his/her pages in the best markup language for each client, but variations between these languages make that an arduous if not intractable task. Most popular sites have given up completely or tried to adopt cumbersome compromises.

With the HTTPPS, the site author may determine the best language for the client and then swap out the pretty printer, leaving all business logic unchanged. Even better, the HTTPPS can do this automatically, as we will soon see.

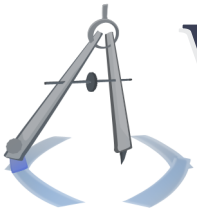
3.3 Abstraction of markup language version

In the same vein as the abstraction of markup language choice, the versions of these markup languages may be chosen dynamically. Pretty printers may exist for each version of each language, and they would all implement the same interface for printing Web pages.

The markup languages we use are ever changing, and aside from the problem of choosing a markup language, simply keeping up with the changes of one markup language can be painful for a site author.

For instance, XHTML Basic is a flavor of XHTML that is better suited for mobile devices, as it is a subset of XHTML. The HTTPPS contains an XHTML Basic pretty printer, which translates XHTML into XHTML Basic in a sensible and customizable way. Again, this may be done with no changes to the business logic.





Website Architecture

Technical Report | The HTTPPS

3.4 Automatic content negotiation

When users request a Web page, they advertise what markup languages they prefer. The process of analyzing these preferences and then serving back an appropriate format is called content negotiation. A component of the HTTPPS may be used to analyze these preferences and give back the optimal pretty printer object. Application users can substitute this into their programs without changing business logic.

More importantly, another component of the HTTPPS, called a variable-markup pretty printer, acts as a proxy for any given pretty printer. It can be used in the above scenario to automatically determine the best pretty printer and then act as its proxy, all while exposing a single, consistent class to the application. The application developer may also derive from this class to provide custom functionality, avoiding the problem of deriving from every possible pretty printer.

3.5 Facilitation of object-oriented site design

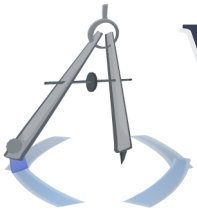
Currently, sites tend to be written using blunt software designs, with an overwhelming lack of software engineering principles. To the skilled observer, this is even evident in the markup code, though the blunt style is pervasive in nearly every PHP snippet available. Pages tend to be written as if one giant function. For one reason or another, the Web is sorely lacking in software engineering, which would have all of its usual benefits: greater assurances of correctness, easier scalability, et. al.

Using the HTTPPS, classes may be created for each business object or Web-page feature, and they each may expose a `ToMarkup()` function, which takes a pretty printer and perhaps other parameters for customization. In this function, they call the given pretty printer's methods and write their representation. The pretty printer may be of a variable language, of course, and it may even be further customized by the application developer to override default behavior. Even still, the business class does not need to change.

This type of versatile decoupling was not possible before, as the printing of the business object may have needed to be aware of its context within the page, either globally (e.g. markup language) or locally (e.g. in the main content area or in or the footer). Even the indentation is kept perfectly when using the HTTPPS, whether the object is written in one part of the page or another.

Using the HTTPPS, sites may finally be designed as clean and robust pieces of software.





Website Architecture

Technical Report | The HTTPPS

3.6 Easy site maintenance through consolidation and extensibility

The HTTPPS encourages software designs that are as easy as possible to maintain, both with respect to maintaining business objects and entire page layouts.

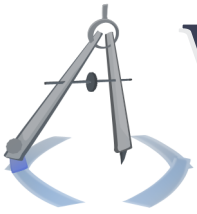
Since object-oriented design is encouraged, the output of each business class is likely to be written exactly once, so it may be updated much more easily than in the traditional `print()` approach. With the old style, how would you be sure to catch every instance of a class being printed? Text search for "`<div`" or "`<div class='Something'`"? The string may be built in any conceivable way before being passed to `print()`, so that approach is intractable. The HTTPPS offers all the usual benefits of object-oriented design in this case, since it allows for such strong decoupling.

Now, aside from business objects, what if you want to change the way that all `<div>`s are printed? For a toy example: perhaps there is a newfound security problem in websites, and you need to change the way that every instance of a certain element is printed. You may derive from a pretty-printer class and then override any functions you want to change. Pass an instance of this customized pretty printer to your business objects, and they will never know the difference.

3.7 Semantically-clear functions for nearly all markup-language features

The Web-page pretty printers in the HTTPPS come with functions for nearly every element in the (X)HTML standard, even including ones that almost no one uses. Functions like `WriteHTMLForm_RadioButton()` take arrays of form-control values as parameters, and they hide the unnatural markup-language syntax which is normally required to do the job. Once the application programmer has a feel for the naming conventions of the API, writing perfect Web pages should be a comfortable process.





Website Architecture

Technical Report | The HTTPPS

4 Suggested use

The HTTPPS has been designed in a forward-thinking manner, with facets for several kinds of site architectures. To get the most out of the HTTPPS, you should probably use it according to the following guidelines.

4.1 Derive for application-specific behavior

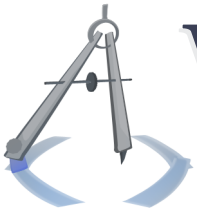
To make broad changes to the way your site is printed, including linguistic changes, you should derive from a pretty-printer class. If you want to change the way that a particular language is written, such as XHTML, then you can derive from an XHTML pretty printer. If you want a more generalized point of entry, you can derive from the variable-markup pretty printer. You could, of course, customize both language-specific and language-variable pretty printers.

For example, some issues in writing Web pages are unresolved, such as the best way to embed Flash movies or Java Applets, especially with respect to strict markup languages and flexibility for older browsers. The HTTPPS may offer some clean and sensible solutions in this regard, which work for modern-era browsers, but you may want to write a more flexible solution (even if more verbose). You can do this by deriving from a pretty printer and overriding the functions which write this code.

The HTTPPS is great for customizing business classes, but aside from business classes in their most familiar form, you will want to write reusable functions which help to write the features of your site layout, such as `WriteFooter()`. You can do this by deriving from a pretty printer and then defining these new functions.

To make each page, you may want to simply call one function from your PHP script, which instantiates a pretty printer and then makes all the proper method calls. Or you may want to build these methods within your customized pretty printer and then instantiate the pretty printers within each script file, calling the methods from the script file. The choice is a matter of style, but you should probably do the former.





Website Architecture

Technical Report | The HTTPPS

4.2 Standardized widget interfaces

The HTTPPS presents a fantastic opportunity for widget designers. If you want to make a widget and distribute it to the world, you can now provide a generic but locally customizable way of writing your widget to the document. Expose a `ToMarkup()` method (or perhaps choose another name), and accept a Web-page-pretty-printer interface as one of your parameters. You may accept additional parameters for further customization, such as for additional CSS classes or for a level of detail. You may also want to provide a `ToXML()` function. You could provide multiple methods, like `ToHTML()` and `ToXHTML()`, but you may want to just handle that internally with your `ToMarkup()` function, in which you can inspect the pretty printer and delegate to your own private methods.

What is perhaps best about this approach is that it is not intrusive into the design of applications. Applications may pass a customized pretty printer to widgets, and the pretty printer has final say on the implementation.

In printing your own widget representation and/or that of other widgets, you may want to be careful that the proper number of tags are closed after each object is written, keeping a sort of equilibrium. This might at least be useful during debugging. To do this, the methods `TagDepth()` and `CloseTagsToDepth()` are provided. At the top of your `ToMarkup()` function, you can sample the depth, and at the end, you can ensure that the tags are closed to the original depth. If you call another object's `ToMarkup()` method from within yours, and if you are cynical, you may want to wrap each foreign `ToMarkup()` call in this sequence.

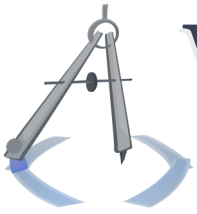
4.3 Prepare markup in advance

The HTTPPS is well implemented, but there is still some overhead in writing such pretty code. For most sites, this is negligible, but for large-scale sites, you may want to circumvent this. All pretty printers come with methods that allow such design patterns.

The methods `WriteBlock()` and `WriteBlockIndented()` may be used here. The former writes a block of text to output without any special processing or indentation. The second writes a block to output but indents the entire block; it accepts an array of text lines in order to achieve this.

But how do you produce such an isolated block? The output of a pretty printer may be redirected to a file, an array, a string, or any other place by using the scribe system, which is discussed in the Implementation section. In case you want to grind around in your own filth, you can even choose a scribe which omits indentations and newlines, printing only a minimal amount of whitespace.





Website Architecture

Technical Report | The HTTPPS

5 Implementation

The implementation of the HTTPPS is characterized by the most regal of design choices.

5.1 Software architecture

The software architecture is broken up into four systems: scribes, tag managers, document schemata, and pretty printers. Scribes are composed within tag managers, which are composed within pretty printers; document schemata are ephemeral and are passed to pretty printers upon initialization.

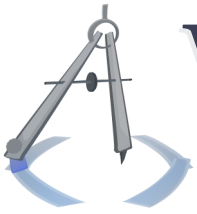
Scribes A scribe is ultimately responsible for writing the text to output. It has a simple interface; it accepts strings for writing principal text, strings that represent indentation, and requests for newlines. And it may direct them all to a file, to an array, to a string, or to the output stream, to name some typical examples. Custom scribes may be created and given to a pretty printer/tag manager. By default, no scribe is defined, and the tag manager writes directly to the output stream; this is an optimization that may be circumvented by explicitly using the "default scribe". Several scribes are available by default, including the compact scribe, which collapses newlines and indentations.

Tag Managers A tag manager is a manager of the basic grammar of a markup language. It opens and closes tags, including keeping track of indentation and keeping track of which tags are open. It exposes public interfaces for writing to output, including the possibility of escaping text with respect to the markup language before output. In opening tags, it accepts an associative array of attributes and properly escapes the attribute values for quotes and for the markup language. In short, it manages the grammar of the markup language without caring for the specific elements. Only XML and HTML tag managers exist, since XHTML uses XML's basic grammar. For an illustrative example: an HTML tag manager may write "`
`" while an XML tag manager writes "`
`".

Document Schemata A document schemata object is a collection of schema objects, each of which describes a schema for the document. The schemata can each be of different formats, such as DTD, XSD, or RELAX NG. When the document is opened via the pretty printer, the document schemata object is passed to the pretty printer, and the pretty printer uses its information to make the proper schema declarations.

Pretty Printers A pretty printer is concerned with the elements of a markup language, not with its basic grammar. It exposes the same methods as its tag manager, but it adds methods that assist in writing the elements of the markup language, like `WriteBR()` and `WriteLink()`. Most of the functions of a pretty printer are nearly-trivial wrappers for its tag manager, provided for the user's convenience. But some are fairly nice abstractions. There are XML, XHTML, and HTML pretty printers, and there is a variabe-markup pretty printer which can act as a late-bound proxy for any HTML or XHTML pretty printer.





Website Architecture

Technical Report | The HTTPPS

5.1.1 Supplementary systems: request analysis and pretty-printer catalogues

In addition to those four systems, there are two supplementary systems. They serve different purposes but have very similar architectures, as we will see.

Request Analysis In writing Web pages, you almost always want to know about the request - the preferred content format, any GET or POST variables, perhaps some specific headers, etc. A request-analysis system provides access to this information, and it offers at least some advantages over the standard way of doing things. A request-metadata object provides information gathered from the HTTP headers, including the client's preferred content type. A request-data object provides information from the core of a request, including its method (GET/POST/etc.), its URI, and its GET/POST/COOKIE/SESSION data. A more semantically clear interface is provided for this data than the default interface, and it is encapsulated, which allows for a few new possibilities.

These objects can be created on the fly and injected with artificial data, so that they can be sent to pretty printers and widgets and effectively misinform them; this would be a form of dependency injection. The data may be changed without changing the "superglobal" data that PHP holds. This may be useful for testing purposes or for generating content in advance for a variety of platforms.

A derived and customized pretty printer should probably hold a reference to these objects.

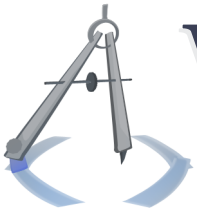
Pretty-Printer Catalogues In the process of content negotiation, the best pretty printer for the request should be selected. In order for this to happen, something needs to have knowledge of all pretty printers. This is a pretty-printer catalogue. The pretty-printer catalogue is a static class that exposes methods which return instances of pretty printers and schemata, either as a best-fit choice or from another specific request by the caller (e.g. "latest XML"). Though the class is static, it is cleanly extensible.

The best-fit algorithm may follow the advertised preferences for some browsers and be less trustful of other browsers, choosing to intervene with an executive decision.

Each of these systems exposes a global - i.e. static - factory method. The request-analysis system can produce objects based on the current request, and the pretty-printer catalogue can produce objects based on the set of possible pretty printers and data from the request-analysis system. Each of these works by a pattern that may be called the Extensible Singleton.

In the Extensible Singleton, a mostly-static class provides a public interface via static methods. For its implementation, however, it defers to an instance of itself, which has a set of virtual, protected methods. The class may have a protected static property called `$Instance`, which is initially `null`. The class also has a protected, static method called `Initialize()`, which may set `$Instance` to an instance of itself. If any of the principal interface methods are called and `$Instance` is `null`, `Initialize()` should be called so that there is at least an object there which may be used.





Website Architecture

Technical Report | The HTTPPS

To extend the Extensible Singleton, a derived class defines overrides for the protected, virtual functions which provide the implementation. A new, public, static method is defined, which may be called `Initialize()` or anything else. This sets the original `$Instance`, of which there is only one, to an instance of this new class. The programmer must call the new `Initialize()` before any of the interface methods are called, which may be called on the original class.

Taking it a step further, we may want to call a number of different implementations, imposing a certain order or priority on them, such that if one function does not produce an answer (e.g. returns `null`), the corresponding function from the next implementation is tried. Thus, `$Instance` can be an array - likely a stack - of such instances.

Using this design, application programmers or even authors of library extensions can expand on the built-in behavior without modifying the original code or any code that uses it. This could easily allow for new pretty printers to be recognized as the latest versions of those for their markup languages, and it could allow for customized or updated handling of HTTP requests.

The variable-markup pretty printer can auto-set its pretty printer according to the catalogue, and thus its choosing algorithm is customizable by the application programmer (in addition to the pretty printer being explicitly malleable).

5.2 Lower-level designs

This section explains some of the lower-level design choices in the implementation.

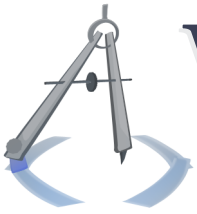
5.2.1 HTML as default behavior

In a few of the abstract base classes, like for tag managers and Web-page pretty printers, some function implementations are given whose behavior should vary between different markup languages. The definitions are given to be valid for HTML, and XHTML-specialized derived classes override those methods. This is for the sake of conceptual simplicity and for a slight reduction in code bloat.

5.2.2 Selective omission of correctness checks

In some cases, the HTTPPS will not check its inputs for correctness with respect to the markup language or even common sense, such as when the inputs are expected to come from the application developer directly, without chance of user input. For instance, the names of tag attributes are not specially processed nor escaped, and neither are tag names. This is for the sake of efficiency and for hands-off management of instructions that are presumed to come directly from the application





Website Architecture

Technical Report | The HTTPPS

developer. This behavior may be overridden, of course, should you have a low opinion of application developers.

5.2.3 Magic methods and properties

Since the variable-markup pretty printer acts as a relatively-transparent proxy, it should replicate its internal pretty printer's members exactly. PHP provides a nice method of doing this, but it is too inefficient for pervasive use, so it has been employed as a backup technique. All of the functions currently in the Web-page-pretty-printer interface have been (re)written explicitly in the variable-markup pretty printer, all of which simply relay the call to the internal object. However, as a failsafe, the variable-markup pretty printer also defines magic methods, accessors, and mutators.

5.3 Code comments and automatically-generated documentation

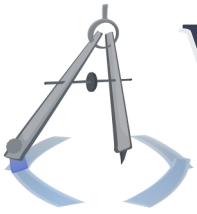
The HTTPPS is painfully well-commented, including coarse-grain comments for class members and fine-grain comments for implementations. The coarse-grain comments are written in a JavaDoc (or "PHPDoc") style which is tailored to the program Doxygen. Using Doxygen, a small website of documentation for the HTTPPS has been generated, which contains explanations for all classes, interfaces, functions, and function parameters. These comments should also be readable by a PHP IDE, so that it may give tooltips and perform code completion.

5.4 Use of interfaces

Interfaces are used pervasively, with each variant of a scribe, tag manager, and pretty printer having its own interface. Interfaces exist for markup languages in general and for each of their versions, with the latter deriving from the former.

Additionally, some of the interfaces may be used by applications and widgets to enforce version requirements on the markup language, such as requiring a pretty printer for XHTML version 1.1 or newer. Nominal interfaces exist for this purpose which do not have any extra functionality; e.g. the interface `HTTPPS_iXHTMLPP_v10Plus` (for XHTML 1.0+). They derive from each other when appropriate, and all instantiable classes implement them when appropriate.





Website Architecture

Technical Report | The HTTPPS

6 Future work

The HTTPPS was written to be customizable and upgradeable, and it was also written to be as expressive as possible without requiring an upgrade.

6.1 Extensibility of architecture

As we have seen in discussing the software architecture, classes in the HTTPPS may be derived for custom behavior and even for compatibility with new markup languages. Interfaces exist which allow both broad and specific compatibility with applications and widgets.

Both the elements and the grammar of a markup language may be updated within the HTTPPS, although the tag managers, which handle grammar, should require very little modification. The method `OpenTag()` can take any tag name as a parameter, and attribute names are not specially processed. Thus, as long as new markup languages are anything like their modern-day counterparts, the HTTPPS should be an appropriate solution.

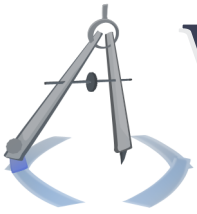
6.2 Recommended update methodology

As indicated above, an update to the library may be done by deriving a new pretty printer and then deriving from the extensible singleton of the pretty-printer catalogue. Now, we look at the file structure of an update. The version 1.0 distribution has a wrapper file, `HTTPPS.inc`, which simply includes an implementation file, `HTTPPS_1.0.inc`.

Extensions to the library should exist in their own implementation files, peer-level with `HTTPPS_1.0.inc`, and they should be included from `HTTPPS.inc`, after the original include. Update files should not include any files of their own; they should assume that they have been included into the proper context, with their dependencies already defined. At the bottom, they should probably include a call to their extensible singleton's initialization function, which should add an instance of the new catalogue to the stack of existing implementations.

The application programmer or administrator can decide the ordering and configuration of files within `HTTPPS.inc`.





Website Architecture

Technical Report | The HTTPPS

7 Conclusions

The HTTPPS is a mighty solution to a host of problems which affect every Internet user, including such global issues as waste of bandwidth and waste of power. It brings the fruits of software engineering to the Web platform, where they have been so curiously absent. The HTTPPS nearly trivializes the problem of conforming to Web standards, all while achieving maximal cross-browser compatibility and flexibility for the future.

