NAME

Apache::HTML::ClassParser − Apache mod_perl extension for generating dynamic HTML pages based on CLASS attributes

SYNOPSIS

```
# In Apache's httpd.conf file:
AddType text/html      .chtml
<Files *.chtml>
  SetHandler           perl-script
  PerlHandler          +Apache::HTML::ClassParser
</Files>
```

DESCRIPTION

Apache::HTML::ClassParser is yet another mod_perl Apache module for dynamically generating HTML. Its distinctive feature, unlike existing techniques, is that it uses *pure*, standard HTML files: no print-statement-laden CGI scripts, no embedded statements from some programming langauge, and no pseudo-HTML elements. Code is cleanly separated into a separate file. What links the two together are CLASS attributes for HTML elements.

CONFIGURATION

In order to have ClassParser be a handler for HTML files, configuration directives, such as those shown in the SYNOPSIS, must be added to Apache's httpd.conf file. The HTML files to be handled by ClassParser, as with all Apache handler modules, can be specified either by location, filename extension, or both. For more detail on Apache configuration directives, see [Apache].

Cooperation with Apache::Filter

ClassParser is Apache::Filter−aware. However, if used in a filter chain, it **must** be the first filter. (This is because it uses the HTML::Tree module that uses *mmap*(2) to read an HTML file.)

For example, to configure Apache to have HTML files run through ClassParser then Apache::SSI (server-side includes), do:

```
PerlModule           Apache::Filter
PerlModule           Apache::HTML::ClassParser
PerlModule           Apache::SSI

AddType text/html    .chtml
<Files *.chtml>
  SetHandler         perl-script
  PerlSetVar         Filter On
  PerlHandler        Apache::HTML::ClassParser Apache::SSI
</Files>
```

TERMINOLOGY AND CONVENTIONS

Element vs. Tag

It is often the case that the term HTML ''tag'' is used when the correct term of ''element'' should be. From the HTML 4.0 specification, section 3.2.1, ''Elements'':

> *Elements are not tags. Some people refer to elements as tags (e.g., "the* P *tag"). Remember that the element is one thing, and the tag (be it start or end tag) is another. For instance, the* HEAD *element is always present, even though both start and end* HEAD *tags may be missing in the markup.*

In this documentation, the distinction between ''element'' and ''tag'' is necessary.

Class vs. `CLASS`

In this documentation, there are unfortunately two meanings of the word ‘‘class’’:

1. A class attribute of an HTML element, e.g.:

```
<H1 CLASS="heading_1">Introduction</H1>
```

   where `CLASS`es are typically used to convey style information.

2. A Perl class from which objects are created, e.g.:

```
package MyClass;
sub new {
    my $class = shift;
    my $this = {};
    return bless $this, $class;
}
$object = MyClass->new();
```

   (See [Wall], pp. 290−292.)

Therefore, throughout this document, ‘‘class’’ written as "`CLASS`‘‘ shall mean the class attribute of an HTML element (case 1) and ’’class‘‘ written simply as ’’class" shall mean a Perl class (case 2).

The HTML File

The file for a web page is in pure HTML. (It can also contain JavaScript code, but that’s irrelevant for the purpose of this discussion.) At every location in the HTML file where something is to happen dynamically, an HTML element must contain a `CLASS` attribute (and perhaps some ‘‘dummy’’ content). (The dummy content allows the web page designer to create a mock-up page.)

For example, suppose the options in a menu are to be retrieved from a relational database, say the flavors available on an ice cream shop’s web site. The HTML would look like this:

```
<!-- ice_cream.chtml -->
<SELECT NAME="Flavors" CLASS="query_flavors">
  <OPTION CLASS="next_flavor" VALUE="0">Tooty Fruity
</SELECT>
```

The `CLASS`es `query_flavors` and `next_flavor` will be used to generate HTML dynamically. The values of the `CLASS` attributes can be anything as long as they agree with those in the code file (specified later). The text ‘‘Tooty Fruity’’ is dummy content.

The `query_flavors CLASS` will be used to perform the query from the database; `next_flavor` will be used to fetch every tuple returned from the query and to substitute the name and ID number of the flavor.

The value of a `CLASS` attribute may contain multiple classes separated by whitespace. (More on this later.)

The Code File

For every HTML file that is to be used with this technique, there **must** be an associated code file in Perl. It **must** have the same name as the HTML file except that the extension is `.pm` rather than `.chtml`. (There is an exception; see "Using a Different Code File via `pm_uri`" below.) That code file **must** define its own package (a.k.a. module), e.g.:

```
# ice_cream.pm
package IceCream;
```

to implement a class; that class **must** have a constructor.

The Constructor and `class_map`

A package requires a constructor method that **must** be named `new()`. A minimal such constructor (for which most of the code is taken from [Wall], p. 295) is:

```
sub new {
    my $that = shift;
    my $class = ref( $that ) || $that;
    my $this = {
        class_map => {
            query_flavors => \&query_flavors,
            next_flavor   => \&next_flavor,
        },
        # other stuff you want here ...
    };
    return bless $this, $class;
}
```

A second requirement is that the object's hash **must** contain a `class_map` key whose value is a reference to a hash containing a mapping from `CLASS` attribute values from the HTML file to functions (methods of the class) in the Perl file.

This is the key concept: it is the `class_map` that links the HTML file to the Perl code.

In the above constructor, the `CLASS` names `query_flavors` and `next_flavor` both map to methods having the same name. In practice, this probably will (and should) be the case; however, there is no requirement that it be so. This allows more than one `CLASS` name to map to the same method. (If there were such a requirement, there would be no need for the `class_map`.)

Class Methods

Class methods are passed the following arguments:

`$this` A reference to an object of a class.

`$node` A reference to an HTML element node, e.g., `SELECT`.

`$class` The value of the `CLASS` attribute of the HTML element the method is being called for. This is useful if more than one class maps to the same method.

`$is_end_tag` True only when the method is being called for the end tag of an HTML element.

Class methods **must** return a Boolean value (zero or non-zero for false or true, respectively). There are two meanings for the return value; they are the same as for *visitor* functions used by **HTML::Tree**. Repeated here for convenience, they are:

1. If the `$is_end_tag` argument is false, returning false means: do not visit any of the current node's child nodes, i.e., skip them and proceed directly to the current node's next sibling and also do not call the *visitor* again for the end tag; returning true means: do visit all child nodes and call the *visitor* again for the end tag.
2. If the `$is_end_tag` argument is true, returning false means: proceed normally to the next sibling; returning true means: loop back and repeat the visit cycle from the beginning by revisiting the start tag of the current element node (case 1 above).

The implementation of the `query_flavors()` and `next_flavor()` methods shall be presented in stages.

The `query_flavors()` method begins by getting its arguments as described above:

```
sub query_flavors {
    my( $this, $node, $class, $is_end_tag ) = @_;
```

The query must be performed upon encountering the start tag of the SELECT element; therefore, the method returns false immediately if $is_end_tag is true. This tells ClassParser not to proceed with parsing the SELECT element's child elements again (in this case, the single OPTION element) and to proceed to its next sibling element (i.e., the element after the /SELECT end tag):

```
        return 0 if $is_end_tag;
```

The bulk of the code is standard DBI/SQL. A copy of the database and statement handles is stored in the object's hash so the next_flavor() method can access them later:

```
        $this->{ dbh } = DBI->connect( 'DBI:mysql:ice_cream:localhost' );
        $this->{ sth } = $this->{ dbh }->prepare( '
            SELECT   flavor_id, flavor_name
            FROM     flavors
            ORDER BY flavor_name
        ' );
        $this->{ sth }->execute();
```

(If the Apache::DBI module was specified ahead of ClassParser in the Apache httpd.conf file, database connections will be transparently persistent. See [Stein], pp. 236–237.)

Finally, the method returns true to tell ClassParser to proceed with parsing the SELECT element's child elements:

```
        return 1;
    }
```

The next_flavor() method begins identically to query_flavors():

```
    sub next_flavor {
        my( $this, $node, $class, $is_end_tag ) = @_;
```

The fetch of the next tuple from the query must be performed upon encountering the start tag of the OPTION element; therefore, the method returns true immediately if $is_end_tag is true. This tells ClassParser to loop back to the beginning of the element, in this case to do another fetch:

```
        return 1 if $is_end_tag;
```

The next portion of code fetches a tuple from the database. If there are no more tuples, the method returns false. This tells ClassParser not to emit the HTML for the OPTION element and also tells it to stop looping:

```
        my( $flavor_id, $flavor_name ) = $this->{ sth }->fetchrow();
        unless ( $flavor_id ) {
            $this->{ sth }->finish();
            $this->{ dbh }->disconnect();
            return 0;
        }
```

The code also disconnects from the database. (However, if Apache::DBI was specified, the disconnect() becomes a no-op and the connection remains persistent.)

The next portion of code substitutes content in the HTML that will be emitted. The first line sets the value of the OPTION element's VALUE attribute to be the flavor_id from the tuple:

```
        $node->att( 'value', $flavor_id );
```

and the next line substitutes the text of the first child node (in this case, the text ''Tooty Fruity'') for the flavor_name from the tuple:

```
    ($node->children())[0]->text( $flavor_name );
```

Finally, the method returns true to tell `ClassParser` to emit the HTML for the `OPTION` element now containing the dynamically generated content:

```
    return 1;
}
```

Other Stuff

Built-in CLASSes

There are several HTML manipulations that are performed routinely; therefore, CLASSes to perform these manipulations are built-in without needing to be explicitly listed in a `class_map`.

All of the built-in CLASSes always return false when called for an end tag; all but `if` and `unless` always return true for a start tag.

`append::`*attribute*`::`*key*

Append to the value of an attribute of the current element. The *attribute* is the name of the HTML attribute whose value is to be appended to and *key* is the key into `$this` that contains the value to be appended. This example appends the value of `$this->{ flavor_id }` to the `HREF` attribute:

```
<A HREF="flavor_detail.chtml?flavor_id="
 CLASS="append::href::flavor_id">Flavor details</A>
```

Note that `append::`*attribute*`::`*key* must not be used inside loops since it always appends.

`check::`*key*

Set the `CHECKED` attribute of the current `INPUT` element for checkboxes and radio buttons only if a value is true. The *key* is the key into `$this` that contains the value to test. This example selects the checkbox only if `$this->{ sprinkles }` is true.

```
<INPUT TYPE=checkbox NAME="Sprinkles"
 CLASS="check::sprinkles">Sprinkles
```

`href::`*key*

This is a shorthand for `sub::href::`*key* since it occurs frequently.

`href_id::`*key*

This is a shorthand for `sub_id::href::`*key* since it occurs frequently.

`if::`*key*

Emit the HTML up to and including the end tag for the current element only if a value is true. The *key* is the key into `$this` that contains the value to be tested. This example emits a table only if `$this->{ results }` is true:

```
<TABLE CLASS="if::results">
  ...
</TABLE>
```

`select::`*key*

Set the `SELECTED` attribute of the current `OPTION` element depending upon the value of the `VALUE` attribute. The *key* is the key into `$this` that contains the value to compare against. This example selects the option only where the `VALUE` is equal to `$this->{ flavor_id }`:

```
<SELECT NAME="Flavor">
  <OPTION CLASS="select::flavor_id" VALUE="0">Chocolate
  <OPTION CLASS="select::flavor_id" VALUE="1">Strawberry
  <OPTION CLASS="select::flavor_id" VALUE="2">Vanilla
</SELECT>
```

`sub::`*attribute*`::`*key*

Substitute the value of an attribute of the current element. The *attribute* is the name of the HTML attribute whose old value is to be substituted and *key* is the key into `$this` that contains the new value to be substituted. This example substitutes the value of the `TYPE` attribute with the value of `$this->{ form_type }`:

```
<INPUT TYPE="radio" NAME="flavor_id" VALUE="1"
 CLASS="sub::type::form_type">
```

`sub_id::`*attribute*`::`*key*

Substitute the ''ID part'' of the value of an attribute of the current element. The *attribute* is the name of the HTML attribute that contains the value whose ''ID part'' is to be substituted and *key* is the key into `$this` that contains the new ID to be substituted. The ''ID part'' matches the pattern `\d+` within the value, i.e., a numeric ID. This example substitutes the ''1'' in the value of the `HREF` attribute with the value of `$this->{ flavor_id }`:

```
<A HREF="/cgi-bin/get?id=1"
 CLASS="sub_id::href::flavor_id">Go</A>
```

`text::`*key*

Substitute the text of the first child node (which **must** be a text node). The *key* is the key into `$this` that contains the text to be substituted. This example substitutes the text ''Tooty Fruity'' with the value of `$this->{ flavor_name }`:

```
<SPAN CLASS="text::flavor_name">Tooty Fruity</SPAN>
```

`unless::`*key*

Emit the HTML up to and including the end tag for the current elemment only if a value is false. (This is the opposite of `if::`*key*.) The *key* is the key into `$this` that contains the value to be tested. (See the example for `if::`*key*.)

`value::`*key*

This is a shorthand for `sub::value::`*key* since it occurs frequently.

`value_id::`*key*

This is a shorthand for `sub_id::value::`*key* since it occurs frequently.

Multiple CLASS Values

The value of a `CLASS` attribute may contain multiple classes separated by whitespace. When that is the case, the function that each maps to is called in turn and the return result is the logical-and of the functions. Just as with Perl's `&&` operator, the first to return false ''short-circuits'' the evaluation.

However, that is true only when the functions are being called for the start tag; when being called for the end tag, only the first function is called regardless of its return value.

In light of this, the first example can be rewritten as:

```
<SELECT NAME="Flavors" CLASS="query_flavors">
  <OPTION CLASS="next_flavor value::flavor_id text::flavor_name"
   VALUE="0">Tooty Fruity
</SELECT>
```

This would call `next_flavor()` as before, but, if it returns true, it would also call the remaining functions in turn.

The `next_flavor()` function would also have to change to:

```
sub next_flavor {
    my( $this, $node, $class, $is_end_tag ) = @_;
    return 1 if $is_end_tag;
    ( $this->{ flavor_id }, $this->{ flavor_name } ) =
            $this->{ sth }->fetchrow();
    unless ( $this->{ flavor_name } ) {
        # ... same as before ...
    }
    return 1;
}
```

The switch to using `$this->{ flavor_id }` and `$this->{ flavor_name }` is necessary since all the built-in CLASSes use `$this->{` *key* `}` for the values they use.

Because the built-in CLASSes are called to do the substitutions, the lines:

```
    $node->att( 'value', $flavor_id );
    ($node->children())[0]->text( $flavor_name );
```

that are present in the original version of `next_flavor()` have been removed.

Improved Performance via `bind_values()`

Although the following has more to do with DBI than `ClassParser`, it's presented here since it will improve the overall performance when using the two together.

Generally, the DBI function `bind_values()` should be used when retrieving multiple tuples from a database query. The `query_flavors()` function can be rewritten to use it as follows:

```
sub query_flavors {
    # ... same as before ...
    $this->{ sth }->bind_values(
        \$this->{ flavor_id }, \$this->{ flavor_name }
    );
    return 1;
}
```

(See the DBI manual page for details.) In this case, the result attributes **must** to be bound to `$this->{` *key* `}` so the values can be accessed by the `next_flavor()` function that can be rewritten as follows:

```
sub next_flavor {
    my( $this, $node, $class, $is_end_tag ) = @_;
    return 1 if $is_end_tag;
    unless ( $this->{ sth }->fetch() ) {
        # ... same as before ...
    }
    return 1;
}
```

Notice that `fetchrow()` has been replaced by `fetch()` and that there are no assignments.

Handling GET Requests

Dynamically generated web pages often need parameters to determine the content. Such paramaters can be passed via the query string as in:

```
http://www.icecream.com/flavor/detail.chtml?flavor_id=4
```

It's desirable to perform parameter validation before displaying any part of a web page because, if you start to display the page and discover an invalid parameter, it's too late. For example, the `flavor_id` should be checked to ensure that it refers to an existing flavor in the database: if it doesn't, a redirection can be done to an error page.

For this to work, the code file **must** define a method named `get()`. The implementation for this example shall be presented in stages. The `get()` method begins simply as:

```
sub get {
    my $this = shift;
```

The code then prepares an SQL query via DBI:

```
    my $dbh = DBI->connect( 'DBI:mysql:ice_cream:localhost' );
    my $sth = $dbh->prepare( '
        SELECT 1
        FROM   flavors
        WHERE  flavor_id = ?
    ' );
```

`ClassParser` stores a copy of the reference to the current `Apache::Request` object into `$this->{ r }` from which query parameters can be extracted:

```
    my $r = $this->{ r };
```

(The variable name `$r` is used by convention.) A check is made to ensure `flavor_id` was given as a parameter and, if so, an attempt is made to execute the query and fetch the result row. If anything fails, the display of the web page is aborted by returning the standard Apache `NOT_FOUND` error:

```
    return NOT_FOUND unless
        $r->param( 'flavor_id' ) &&
        $sth->execute( $r->param( 'flavor_id' ) ) &&
        $sth->fetchrow();
```

If all succeeds, then the function simply returns `OK`:

```
    return OK;
}
```

The return value **must** be one of the `Apache::Constants`. If the value is `OK`, then the display of

the web page proceeds normally; if the value is anything other than `OK`, then the web page is not displayed. In either case, that return value is passed back to Apache.

Handling POST Requests

Ordinarily, the `ACTION` attribute in an HTML `FORM` element contains a URI pointing at a CGI script, e.g.:

```
<FORM ACTION="/cgi-bin/process_my_data.cgi" METHOD="post">
```

After processing form data, some web page needs to be displayed to the user. The CGI could either emit the HTML for that page itself or do a redirect to an existing web page. A `ClassParser`-parsed HTML file, however, can handle `POST` requests itself, e.g.:

```
<FORM ACTION="/my.chtml" METHOD="post">
```

For this to work, it **must** define a method named `post()` such as:

```
sub post {
    my $this = shift;
    # do something yummy with $this->{ r }->param( 'Flavors' ) ...
    return OK;
}
```

Similarly to the GET request case, `ClassParser` stores a copy of the reference to the current `Apache::Request` object into `$this->{ r }` from which form parameters can be extracted. In the example, `Flavors` is the value of the `NAME` attribute of the `SELECT` element, so `$this->{ r }->param('Flavors')` is the value of the `VALUE` attribute of the selected `OPTION` element, or the flavor ID.

Identically to the GET request case, the return value **must** be one of the `Apache::Constants`.

Using a Different Code File via `pm_uri`

As previously described, every HTML file **must** have an associated code file having the same name as the HTML file except with an extension of `.pm` rather than `.chtml`. However, if a FORM parameter `pm_uri` has a value either via a GET or POST (but most often via a POST), then **that** code file will be used instead to handle either the GET or POST request. This allows you to consolidate form-processing code into a single code file. The easiest way to specify a value for `pm_uri` is using a hidden `INPUT` element:

```
<INPUT TYPE=hidden NAME="pm_uri" VALUE="other.pm">
```

Persistence

A code file is loaded, compiled once, and stored in memory. Additionally, a single object is created via the constructor and stored in memory also. Both the code and the object are used for all subsequent HTTP requests. The code is reloaded, recompiled, and stored, and a new object is created only if the modification time of the code file is later than the last time it was loaded.

This code/object persistence is done for efficiency/performance reasons only. In particular, although an object is persistent across multiple HTTP requests, data can not be stored in the object to implement anything like ''sessions'' for users. The reason is because Apache forks multiple child processes and code and objects are persistent only within a **single** child process and not among them. Hence, it is most likely the case that a given user's HTTP requests will be handled by multiple Apache child processes. (There are several techniques for doing ''sessions'': see [Stein], Chapter 5.)

Web Page Caching

`ClassParser` sets the `no_cache` flag for the Apache request. Since web pages generated with `ClassParser` are dynamic, they should always reflect the latest information.

SEE ALSO

perl(1), *Apache::DBI*(3), *Apache::Filter*(3), *DBI*(3), HTML:\fIs0:Tree(3).

Apache Group. *Apache HTTPD Server Project*, `http://www.apache.org/`

Apache/Perl Integration Project, *mod_perl*, `http://perl.apache.org/`

Dave Raggett, et al. ''On SGML and HTML: SGML constructs used in HTML: Elements,'' *HTML 4.0 Specification, section 3.2.1*, World Wide Web Consortium, April 1998. `http://www.w3.org/TR/PR-html40/intro/sgmltut.html#h-3.2.1`

--. "The global structure of an HTML document: The document body: Element identifiers: the `id` and `class` attributes," *HTML 4.0 Specification, section 7.5.2*, World Wide Web Consortium, April 1998. `http://www.w3.org/TR/PR-html40/struct/global.html#h-7.5.2`

Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C*, O'Reilly & Associates, Inc., Sebastopol, CA, 1999.

Larry Wall, et al. *Programming Perl, 2nd ed.*, O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

AUTHOR

Paul J. Lucas <*pjl@best.com*>

CREDIT

As far as I know, the technique for using pure HTML to generate dynamic web pages using only `CLASS` attributes was invented by Erik Ostrom.