Embperl - How to Build Large Scale Websites/Webapplications With Perl

ApacheCon 2002

Gerald Richter

ecos gmbh

http://www.ecos.de

Introduction

In the early days of the of the web, (server-side) dynamic web pages meant CGI scripts. However, CGI scripts were lousy to read because of a lot of `print` statements with a lot of escaping, over long lines and nearly no chance to guess the final output at one glance.

These problems were mostly solved by the so called templating systems. Popular templating systems are PHP, JSP and ASP. In Perl there are multiple templating systems, each which it's own advantages and disadvantages. One of the most powerfull and widely used systems is *Embperl*.

These templating systems embed the program code into the markup and not vice versa as CGI scripts did. This usually made the code more readable and the final output easier to guess. The possibility to include other files made complex web pages manageable, because you can have common portions of the site in just one file and include them in many others. So you could change the layout of the headers, footers or menus each in just one file and the design changed on all pages.

But imagine, you once want to the menu component - which always was on the left side - on the right side. You would have to change all files and move the statement which generates or includes the menu from in front of the content to behind the

content. If your site is large, this can be a lot of (dumb) work.

Another disadvantage of this method is, that if you want one branch of your site to have a slightly different layout - let's say it differs in one of the included files - you'll have to change this information in each of the affected files at top level. In object-oriented programming, you would derive a new class from an existing one, overriding just one or a few methods and all other methods will be inherited from it's parent class (es).

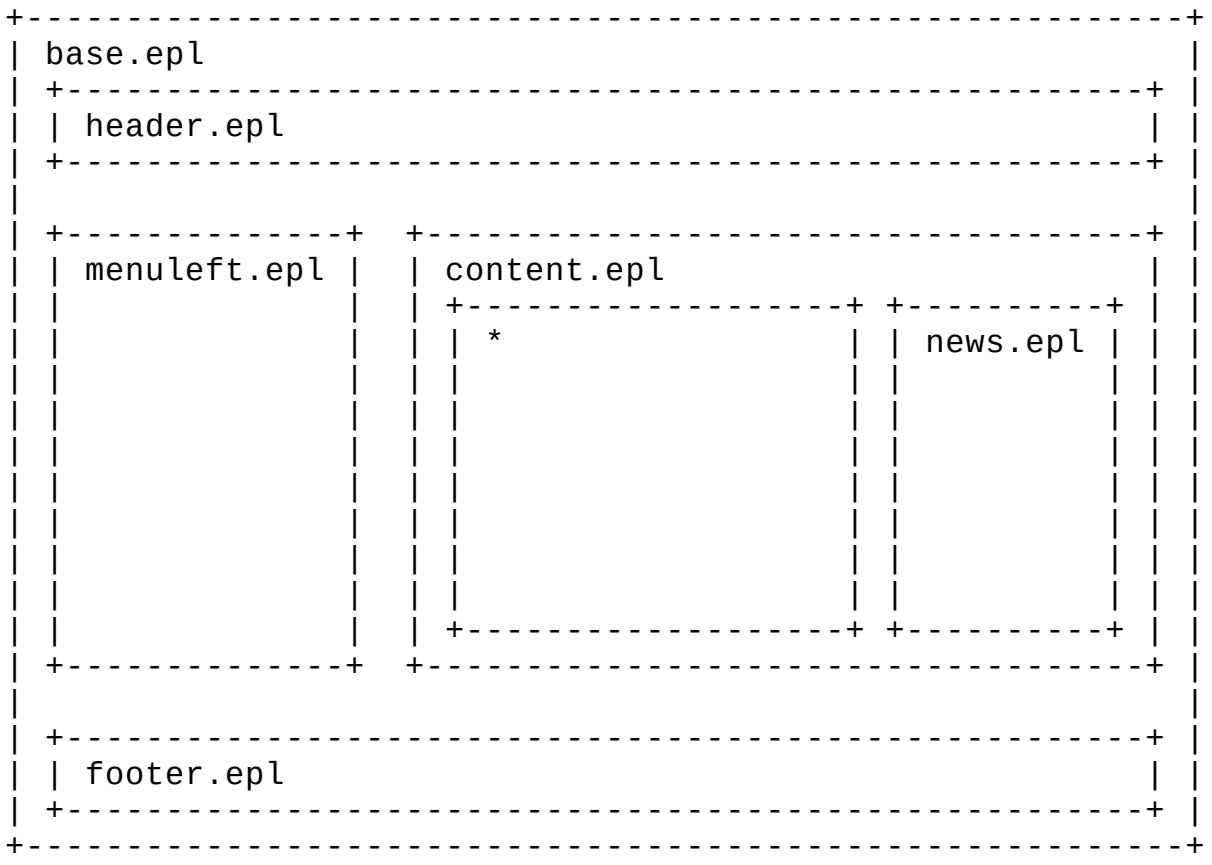
So why not carryover the concept of object-oriented programming to creating dynamic websites? Embperl does that by providing *Embperl::Object* and thereby facilating the conception and implementation of big dynamic component-based websites.

Using Embperl's own website (http://perl.apache.org/embperl/) as example, the following sections will show how *Embperl::Object* works. The full source code of the Embperl website is included in the distribution of version 2.0 of Embperl in the directory `/eg/web`

The Layout

Overview

The Embperl website basically is structured as follows:



If a web page is requested (e.g. */index.epl*, *Embperl::Object* first searches for the base template. In case of the Embperl website this is *base.epl* (name can be configured in

the web server configuration), which looks like this:

```
<html>
    <head>
        <title>Embperl</title>
    </head>
    <body bgcolor="#ffffff">
        [- Execute ('header.epl') -]
        <table width="100%" border="0">
            <tr>
                <td>[- Execute ('menuleft.epl') -]</td>
                <td>[- Execute ('content.epl')  -]</td>
            </tr>
        </table>
        [- Execute ('footer.htm') -]
    </body>
</html>
```

base.epl contains several calls to Execute. In our example `Execute` just includes the named pages, but it is very powerfully and has a long list of possible arguments.

So *base.epl* will include *header.epl*, *menuleft.epl*, *content.epl* and finally include *footer.htm*.

Let take a look at *content.epl*. It looks like this:

```
<table width="100%" border="0">
    <tr>
        <td>[- Execute('*') -]</td>
        <td>[- Execute('news.epl') -]</td>
    </tr>
</table>
```

It contains again calls to `Execute` of which one will call *news.epl* and the other one is special: `Execute('*')` includes the file initially requested from the web server, in our case */index.epl*.

So we have separated the layout from the content in a way, which doesn't need any inclusion of headers, footers or menus in the files providing the content. If we want to change the content, we just have to modify one of the above mentioned files and the whole sites may have changed it's layout without great effort.

There is another advantage: On pages other then the home page, we don't want to show the news column and this can be simply done by replacing *content.epl* in a subdirectory. For example under the directory */pod* all the documentation is located. Now we put there the file */pod/content.epl*, which only contains:

```
[- Execute('*') -]
```

What's happeing now is, that when you request a file under the */pod* directory Embperl::Object uses this content.epl file and because of that, the news comlumn will

not be included.

So let make an example. When you request the file */pod/doc/index.epl*, which contains a list of all the documentation available, Embperl::Object first searches the base template (base.epl). It does this by walking up the directory tree, starting in the directory where the requested file is located, until it either found it or reached the document root (or the directory configured with `EMBPERL_OBJECT_STOPDIR`). When base.epl is found, the same search is taking place for all files that are called via `Execute`. This is the reason why it picks up the /pod/content.epl in this case and header.epl etc. are still taken form the same directory as before.

So what we done here is, that we have overridden *content.epl* in the sub directory *pod*.

Separation of Application Logic and Content

After having separated the content from the layout, we usually still have content mixed with application logic. To isolate the application logic from the content, `Embperl::Object` provides with `EMBPERL_OBJECT_APP` the possibility to define a file, which contains all application code. For the Embperl website, the application code resides *epwebapp.pl*. For loading it, `Embperl::Object` searches the same path as for all other included elements and the base template.

For each application file loaded this way, Embperl create on the fly a package and a hash reference. It then *bless*es the hash reference into the package. So it provides easy object-oriented access to the application. (Because Embperl already does this, you should **not** include a `package` statement in that file.) The application code file also will be automatically inherited from `Embperl::App` via `@ISA`. This enables easy access to all methods of superior objects as e.g session handling. Also note that the application file only contains Perl code and no markup, since we are defining the application logic.

After loading the application code and preparing all request related informations (like e.g. submitted form data, session data), `Embperl::Object` calls the method `init`, which - as usual for Perl methods - get's a reference to the application object as first parameter. The second parameter is Embperl's request object.

The following `init` method is used at the emperl website e.g. to generate the menus.

```
sub init
    {
    my ($self, $r) = @_;

    my $config = Execute({ object => 'config.pl', syntax =>
    $config->new($r) ;

    $r->{config} = $config ;
    $r->{menu}   = $config->get_menu($r);

    fill_menu($config, $r->{menu}, $r->{baseuri}, $r->{root}

    my $filename = map_file($r);
```

```
$r->param->filename($filename);

return 0;
}
```

First the file *config.pl* is loaded and used to generate an object (as happend with the application code file itself), which is returned by `Execute()`. Then it initializes the new object by calling its `new` method and generates a menu by calling the method `get_menu`.

Defining the navigation structure

How the menu structure is defined in *config.epl* doesn't matter. In case of the Embperl website this is done within a Perl hash, but it could also have been a XML file, the only point is that the method `get_menu` returns it in a well defined way.

The config object and the menu struture is placed into the request object. Just like the application object, the request object is a blessed hash reference. You can use theses hashs to store your own object data. Embperl itself doesn't store anything inside of these hashs. The difference between request and application object is their life time. While the request object and all data it contains, is destroyed at the end of the request, the application object is only destroyed when the server ends.

The method `fill_menu` now takes this menu structure and the parameters of the request and prepares it for displaying. So when finaly `menuleft.epl` is invoked to display the menu, it only has to take the prepared data and surround it with a nice layout. It doesn't contain any logic anymore, so we have seprated the logic into the application object and the layout into the template.

Another imported feature of the application object is, that it is invoked before any output is generated, so you are able to modify most of the request parameters. This is done in the next few lines of the `init` method, by callining `map_file`. `map_file` tries to locate the requested uri in the configuration provided by config.epl and, if found, returns the actual filename for it. It also takes into account other parameters like the prefered language to map to the correct file. The init method now modifies the request to serve this file, instead of using the one that come out of the mapping done by Apache.

As we have seen before the application object is search in the same way as other pages. We can use this to define a derived application object to extent functionality. For the Embperl website this is done in the `/db` directory. The website provides several informations which are stored in a database, like news, links, examples, etc.

All necessary pages for the database access are beneath the `/db` directory and it also contains a file `epwebapp.pl`. So when any page underneath /db is requested Embperl::Object will find this application object instead of the one in the base directory. This application object provides all necessary logic for the database access, but we still need the functions from application object we have discussed above. So what we do is tell Embperl that is application object inherits from the first one. This is done by calling

Execute with the isa parameter:

```
BEGIN { Execute ({isa => '../epwebapp.pl', syntax => 'Perl'}
```

This call load and compiles the base object and adjusts the `@ISA` array of the calling object accordingly to get a proper inherence. This object also has an `init` method, which looks like this:

```
sub init
    {
    my $self = shift ;
    my $r = shift ;

    $self->SUPER::init($r) ;
    $self->initdb($r) ;

    if ($fdat{-add_category})
        {
        $self -> add_category ($r) ;
        $self -> get_category($r) ;
        }
    elsif ($fdat{-add_item})
        {
        $self -> add_item ($r) ;
        $self -> get_category($r) ;
        $self -> get_item_lang($r) ;
        }
    elsif ($fdat{-show_item})
        {
        $self -> get_category($r) ;
        $self -> get_item_lang($r) ;
        }
    else
        {
        $self -> get_category($r) ;
        $self -> get_item($r) ;
        }
    return 0 ;
    }
```

First it calls `SUPER::init` to give the base class a chance to do its initialization. Then it calls `initdb`, which sets up database connections etc. As next step it checks the hash `%fdat`, which contains all the form data that is send by GET or POST to the page. Depending on what the user requested when he/she submit the form, different methods are called, which do the database access, like retrieving data and inserting new items etc. The result of the database access is again placed into the request object so it's available to the be displayed.

Converting different formats: Providers and Recipes

Not only on the Embperl website the content has different source formats. For example the documentation is written in POD (Plain Old Documentation) while the home page

is HTML and other pages are HTML with some Perl code in it. To manage these different formats you can give the `syntax` parameter to the `Execute` function and tell Embperl how the source should be interpreted. Embperl comes with different predefined syntaxes (among others SSSI, ASP, Text, Perl, RTF, POD), but you can also define your own syntax.

In the above example we can see that when reading the configuration file, syntax => 'Perl' is used to tell Embperl that the configuration file contains only Perl code. Similar you can use syntax => 'Text' to pass the file through without doing any interpretation of the content.

Thinks get more compilcated when we try to process POD, because Embperl not only has to understand the syntax, but also need to generate the markup (HTML in this case).

For this purpose Embperl provides *recipes*. A recipe defines which steps are taken to process a source file. Each of these steps are done by a provider. If no recipe is selected, the default is used which defines the steps parse, compile, execute and output. Additionaly there are recipes for processing XML and doing XSLT as part of the Embperl distribution. If they don't fit your needs, you can define your own recipes. For displaying POD on the Embperl website, we use the `EmbperlXSLT` recipe. Addtionaly we set the `syntax` parameter to POD. This tells Embperl to convert the POD source into XML data, so the XSLT provider defined by the recipe can transform this into the destination format (e.g. HTML). To make this happen an addional provider cares about reading the XSL stylesheet and providers transforms the text version of the XML and XSL into some internal format suitable for the XSLT processor. Since Embperl is able to cache any of these intermediate results, this can speed up pocessing considerably, when doing a lot of pages.

Since we don't want to configure for any individual page which recipe to use, it seems to be a good idea to use file extentions for selecting a recipe.

This can be implemented by overiding the method `get_recipe` in the application object. Embperl is calling this method before every file is processed. So in our epwebapp.pl we define the following method:

```
sub get_recipe

    {
    my ($class, $r, $recipe) = @_ ;

    my $self ;
    my $param  = $r -> component -> param  ;
    my $config = $r -> component -> config  ;
    my ($src)  = $param -> inputfile =~ /^.*\.(.*?)$/ ;
    my ($dest) = $r -> param -> uri =~ /^.*\.(.*?)$/ ;

    if ($src eq 'pl')
        {
        $config -> syntax('Perl') ;
```

```
        return Embperl::Recipe::Embperl -> get_recipe ($r, $
        }

    if ($src eq 'pod' || $src eq 'pm')
        {
        $config -> escmode(0) ;
        if ($dest eq 'pod')
            {
            $config -> syntax('Text') ;
            return Embperl::Recipe::Embperl -> get_recipe ($
            }

        $config -> syntax('POD') ;
        if ($dest eq 'xml')
            {
            return Embperl::Recipe::Embperl -> get_recipe ($
            }

        $config -> xsltstylesheet('pod.xsl') ;
        $r -> param -> uri =~ /^.*\/(.*)\.(.*?)$/ ;
        $param -> xsltparam({
                page      => $fdat{page} || 0,
                basename  => "'$1'",
                extension => "'$2'",
                imageuri  => "'$r->{imageuri}'",
                baseuri   => "'$r->{baseuri}'",
                }) ;
        return Embperl::Recipe::EmbperlXSLT -> get_recipe ($
        }

    if ($src eq 'epd')
        {
        $config -> escmode(0) ;
        $config -> options($config -> options | &Embperl::Co

        if ($dest eq 'pod')
            {
            $config -> syntax('EmbperlBlocks') ;
            return Embperl::Recipe::Embperl -> get_recipe ($
            }

        $config -> xsltstylesheet('pod.xsl') ;
        $r -> param -> uri =~ /^.*\/(.*)\.(.*?)$/ ;
        $param -> xsltparam({
                page      => $fdat{page} || 0,
                basename  => "'$1'",
                extension => "'$2'",
                imageuri  => "'$r->{imageuri}'",
                baseuri   => "'$r->{baseuri}'",
                }) ;
        return Embperl::Recipe::EmbperlPODXSLT -> get_recipe
        }

    if ($src eq 'epl' || $src eq 'htm')
```

```
            {
            $config -> syntax('Embperl') ;
            return Embperl::Recipe::Embperl -> get_recipe ($r, $
            }

        $config -> syntax('Text') ;
        return Embperl::Recipe::Embperl -> get_recipe ($r, $reci
        }
```

First `get_recipe` determinates the extentions of the source and destination file (`$src` and `$dest`). Depending on the combination of these two it selects the correct recipe. Because of that you can produce different output formats (e.g. POD, XML, HTML) from the same source. Additional get_recipe set some parameters like syntax, output escaping and parameters passed to the XSLT stylesheet, so they fit to the desired source and destionation format.

Include external components

When running Embperl with Apache 2.0 there are some extented possibilities. While Apache 1.x has send all it's ouput directly to the browser, Apache 2.0 introduces a concept of filters, which allows to process the output of any Apache handler through a chain of filters. Embperl can use this to embed any output that Apache can generate as a Embperl::Object component, just like it is any native Embperl page. This can be done by using the `subreq` parameter:

```
    [- Execute ({subreq=>'/cgi-bin/script.cgi'}) -]
```

The above code includes the output of a cgi script into a page.

This is especially usefull for application that are not newly written from ground up, but has grown over years, because you can include existing solution into your Embperl::Object driven website. Because of the flexibilty of the recipe/provider concept, you can not only include the output of thoses other components, but also postprocess it. For example you can include the output of a cgi script, for which you don't have the source code and can adapt the output to your current layout.

In the same way you can combine applications written in differnet languages like PHP and Java under a common layout. When you have included the Apache proxy module, the source must not reside localy on your machine, but you are able to request it from any webserver. You may for example query XML data from another server, for example news in the RSS format and run an XSL-transformation to make it look nicely into your layout.

Finaly

This text has only touched some of the most important features of Embperl, but should have give you an impression of what is possible.

If you interested in more you find addtional information on the Embperl website

http://perl.apache.org/embperl

or

http://www.ecos.de/embperl

Authors

Gerald Richter (richter at ecos dot de) Axel Beckert (abe at ecos dot de)