WebDyne Installation and Usage Guide

Table of Contents

Table of Contents

WebDyne Installation and Usage Guide

Andrew Speer

<andrew.speer@webdyne.org>

This article describes how to install and use the WebDyne mod_perl/FastCGI dynamic content engine.

Table of Contents

1. Introduction

WebDyne is a Perl based dynamic HTML engine. It works with web servers (or from the command line) to render HTML documents with embedded Perl code.

Once WebDyne is installed and initialised to work with a web server any file with a `.psp` extension is treated as a WebDyne source file. It is parsed for WebDyne or CGI.pm pseudo-tags (such as <perl> and <block> for WebDyne, or <start_html>, <popup_menu> for CGI.pm) which are interpreted and executed on the server. The resulting output is then sent to the browser.

Pages are parsed once, then optionally stored in a partially compiled format - speeding up subsequent processing by avoiding the need to re-parse a page each time it is loaded. WebDyne works with common web server persistant/resident Perl modules such as mod_perl and FastCGI to provide fast dynamic content.

---

2. Legal Information - Licensing and Copyright

---

3. Credits

WebDyne relies heavily on modules and code developed and open-sourced by other authors. Without Perl, and Perl modules such as mod_perl/FCGI, CGI.pm, HTML::TreeBuilder and Storable, WebDyne would not be possible. To the authors of those modules - and all the other modules used to a lesser extent by WebDyne - I convey my thanks.

---

4. Installation

WebDyne can be installed via the following methods:

Perl CPAN
: Install from the Perl CPAN library. Installs dependencies if required (also via CPAN). Use this method if you are familiar with CPAN. Destination of the installated files is dependent on the local CPAN configuration, however in most cases it wil be to the Perl site library location. WebDyne supports installation to an alternate location using the PREFIX option in CPAN. Binaries are usually installed to `/usr/bin` or `/usr/local/bin` by CPAN, but may vary by distribution/local configuration.

  Assuming your CPAN environment is setup correctly you can run the command:

  perl -MCPAN -e "install WebDyne"

  To install the base WebDyne module, which includes the Apache installer.

  To get the installer for Lighttpd (after the base package above is installed) run

  perl -MCPAN -e "install WebDyne::Install::Lighttpd"

  To get the complete suite of WebDyne modules, including all installers and externsion modules (Session manager etc.) run:

  perl -MCPAN -e "install Bundle::WebDyne"

WebDyne must be initialised after installation. To get started quickly run **wdapacheinit** after CPAN installation to setup WebDyne for an Apache/mod_perl environment, or **wdlighttpdinit** for a Lighttpd/FastCGI environment (make sure the Lighttpd FastCGI module is installed - it is often packaged as a separate component in many Linux distributions). See the initialisation section for more information

4.1. Prerequisites

WebDyne requires mod_perl available when running with Apache, or FastCGI support if running with Lighttpd. Installation on Windows requires ActiveState Perl be installed, and Apache/mod_perl be available. Installation with the Strawberry Perl distribition should work, but has not been tested (feedback welcome).

In pathological cases WebDyne can run in CGI mode (does not need mod_perl or FastCGI) however such a configuration is not supported by the installation scripts, and would be extremely inefficient and CPU intensive under any non-trivial load.

4.2. Compatibility

WebDyne should install on any modern Linux distribution. It will run with mod_perl 1.x, 1.99_x, 2.0.x and 2.2, and has been tested on a variety of distributions.

Installation from CPAN or the source code should be possible with little or no effort on most *nix/compatible systems such a *BSD, Solaris etc. so long as the correct development tools and libraries are available.

WebDyne has been installed onto a Windows 2003 SP1 server running Apache 2.0/mod_perl, Apache 2.2/mod_perl and IIS. The **wdapacheinit** installer may work in other Windows environments but has not been tested.

5. Initialisation

After installation Web Server configuration files must be updated so that the WebDyne software will be used to generate output when a .psp file is invoked. WebDyne comes bundled with an installer for Apache (**wdapacheinit**), and installers for selected other web servers (e.g. `WebDyne::Install::Lighttpd`) are available separately from CPAN.

Initialisation can be done via one of two methods:

Manual Initialisation
: Web server configuration files can be hand-edited, and cache directories manualy created.

Script Initialisation
: Scripts to automate the initialisation process for Apache and Lighttpd have been written - they will attempt to locate and update the Web Server config files (and create neccessary directories, set permissions etc.) as required. The scripts will work in common cases, but may have trouble on unusual distributions, or if a custom version of Apache (or other Web Server) is being used

Scripted installation is easiest if it works for your distribution - it will take care of all configuration file changes, directory permissions and ownership etc.

5.1. Running wdapacheinit to initialise the software for use with Apache/mod_perl

Once the WebDyne software in installed it must be initialized. The **wdapacheinit** command must be run to update the Apache configuration files so that WebDyne pages will be correctly interpreted:

```
[root@localhost ~]#  /opt/webdyne/bin/wdapacheinit

[install] - Installation source directory '/opt/webdyne'.
[install] - Using existing cache directory '/opt/webdyne/cache'.
[install] - Updating perl5lib config.
[install] - Writing Apache config file '/etc/httpd/conf.d/webdyne.conf'.
[install] - Granting Apache write access to cache directory.
[install] - Install completed.
```

By default WebDyne will create a cache directory in `/var/cache/webdyne` on Linux systems when a default CPAN install is done (no PREFIX specified). If a PREFIX is specified the cache directory will be created as `PREFIX/cache`. Use the `--cache` command-line option to specify an alternate location.

Once **wdapacheinit** has been run the Apache server should be reloaded or restarted. Use a method appropriate for your Linux distribution.

```
[root@localhost ~]#  service httpd restart
Stopping httpd:                                            [  OK  ]
```

```
Starting httpd:                                            [  OK  ]
```

WebDyne should be now ready for use.

If the Apache service does not restart, examine the error log (usually `/var/log/httpd/error.log`) for details.

The script will look for Apache components (binary, configuration directories etc.) using common defaults. In the event that the script gives an error indicating that it cannot find a binary, directory or library you may need to specify the location manually. Run the script with the `--help` option to determine the appropriate syntax.

5.2. Manual configuration of Apache

If the **wdapacheinit** command does not work as expected on your system the Apache config files can be modified manually.

Include the following section in the Apache httpd.conf file (or create a webdyne.conf file if you distribution supports conf.d style configuration files):

```
#  Put this in if WebDyne was installed using the PREFIX option to its own directory - replace
#  /opt/webdyne with the correct path to the perl5lib.pl library.
#
#  This will adjust the @INC path so all WebDyne modules (and modules WebDyne relies on) can be
#  located and loaded.
#
#  Not needed if installed to default Perl library location, but does not hurt to leave in.
#
PerlRequire   "/opt/webdyne/bin/perl5lib.pl"


#  Preload the WebDyne and WebDyne::Compile module
#
PerlModule    WebDyne WebDyne::Compile


#  Associate psp files with WebDyne
#
AddHandler    perl-script    .psp
PerlHandler   WebDyne


#  Set a directory for storage of cache files. Make sure this exists already is writable by the
#  Apache daemon process.
#
PerlSetVar    WEBDYNE_CACHE_DN    '/opt/webdyne/cache'


#  Allow Apache to access the cache directory if it needs to serve pre-compiled pages from there.
#
<Directory "/opt/webdyne/cache">
Order allow,deny
Allow from all
Deny from none
</Directory>
```

Substitute directory paths in the above example for the relevant/correct/appropriate ones on your system.

Create the cache directory and assign ownership and permission appropriate for your distribution (group name will vary by distribution - locate the correct one for your distribution)

```
[root@localhost ~]#  mkdir /opt/webdyne/cache
[root@localhost ~]# chgrp apache /opt/webdyne/cache
[root@localhost ~]# chmod 770 /opt/webdyne/cache
```

Restart Apache and check for any errors.

5.3. Running wdlighttpdinit to initialise the software for use with Lighttpd/FastCGI

If using WebDyne with Lighttpd download and install the `WebDyne::Install::Lighttpd` module. Then run the **wdlighttpdinit** command to update the Lighttpd configuration files so that WebDyne pages will be correctly interpreted:

WebDyne depends on the Lighttpd mod_fastcgi module (on RPM systems the package is sometimes called lighttpd-fastcgi). Please ensure the it is installed before running the initialisation script.

```
[root@localhost ~]#  /opt/webdyne/bin/wdlighttpdinit

[install] - Installation source directory '/opt/webdyne'.
[install] - Using existing cache directory '/opt/webdyne/cache'.
[install] - Updating perl5lib config.
[install] - Writing Lighttpd config file '/etc/lighttpd/webdyne.conf'.
[install] - Lighttpd config file 'lighttpd_conf_fn'
[install] - Lighttpd config file '/etc/lighttpd/lighttpd.conf' updated.
[install] - Granting Lighttpd write access to cache directory.
[install] - Install completed.
```

Once **wdlighttpinit** has been run the Lighttpd server should be reloaded or restarted. Use a method appropriate for your Linux distribution.

```
[root@localhost ~]#  service lighttpd restart
Stopping lighttpd:                                         [  OK  ]
Starting lighttpd:                                         [  OK  ]
```

WebDyne should be now ready for use.

If the Lighttpd service does not restart, examine the error log (usually `/var/log/lighttpd/error.log`) for details.

5.4. Manual configuration of lighttpd

If the **wdlighttpdinit** command does not work on your system the Lighttpd config files can be modified manually..

Include the following section into the lighttpd configuration file:

```
#  Include the Lighttpd FastCGI module - make sure it is present on the system, install if not
#
#
server.modules += (
    "mod_fastcgi",
),
```

```
#  Register the psp extension with the FastCGI module
#
fastcgi.server = (

    ".psp" => (
        "localhost" => (

            #  Change paths as appropriate for your system, socket dir must be writable by
            #  Lighttpd daemon process owner.
            #
            "socket"           => "/opt/webdyne/cache/wdfastcgi-webdyne.sock",
            "bin-path"         => "/opt/webdyne/bin/wdfastcgi",

            #  Optional, must be writable by Lighttpd daemon process owner
            #
            "bin-environment"  => (
                "WEBDYNE_CACHE_DN"=> "/opt/webdyne/cache"
            )
        )
    )
)
```

6. Getting Started/Basic Usage

Assuming the installation has completed with no errors you are now ready to start creating WebDyne pages and applications.

6.1. Basics - integrating Perl into HTML

Some code fragments to give a very high-level overview of how WebDyne can be implemented. First the most basic usage example:

Example 1.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Note the perl tags -->

Hello World <perl> localtime() </perl>

</body>
</html>
```

Run

So far not too exciting - after all we are still mixing code and content. Lets try again:

Example 2.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Empty perl tag this time, but with method name as attribute -->

Hello World <perl method="hello"/>

</body>
</html>

__PERL__

sub hello { return localtime }
```

Run

Better - at least code and content are distinctly separated. Note that whatever the Perl code returns at the end of the routine is what is displayed. Although WebDyne will happily display returned strings or scalars, it is more efficient to return a scalar reference, e.g.:

```
#  Works
#
sub greeting { print "Hello World" }


#  Is the same as
#
sub greeting { return "Hello World }
sub greeting { my $var="Hello World"; return $var }


# But best is
#
sub greeting { my $var="Hello World"; return \$var }


# This will cause an error
#
sub greeting { return undef }


# If you don't want to display anything return \undef,
#
sub greeting { return \undef }


# This will fail also
#
sub greeting { return 0 }


#  If you want "0" to be displayed ..
#
sub greeting { return \0 }
```

Perl code in WebDyne pages must always return a non-undef/non-0/non-empty string value (i.e. it must return something that evals as "true"). If the code returns a non-true value (e.g. 0, undef, '') then WebDyne assumes an error has occurred in the routine. If you actually want to run some Perl code, but not display anything, you should return a reference to undef, (**\undef**), e.g.:

```
```

```
sub log { &dosomething; return \undef }
```

Up until now all the Perl code has been contained within the WebDyne file. The following example shows an instance where the code is contained in a separate Perl module, which should be available somewhere in the `@INC` path.

Example 3.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Perl tag with call to external module method -->

Hello World <perl method="MyModule::hello"/>

</body>
</html>
```

If not already resident the module (in this case "MyModule") will be loaded by WebDyne, so it must be available somewhere in the `@INC` path. The example above cannot be run because there is no "MyModule" package on this system.

6.2. Use of the <perl> tag for in-line code.

The above examples show several variations of the <perl> tag in use. Perl code that is enclosed by <perl>..</perl> tags is called *in-line* code:

Example 4. Simple in-line perl code

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
<pre>

<!-- Perl tag containing perl code which generates output -->

<perl>

for (0..3) {
        print "Hello World\n"
}

#  Must return a positive value, but don't want anything
#  else displayed, so use \undef
#
\undef;

</perl>


</pre>
```

```
</body>
</html>
```

Run

This is the most straight-forward use of Perl within a HTML document, but does not really make for easy reading - the Perl code and HTML are intermingled. It may be OK for quick scripts etc, but a page will quickly become hard to read if there is a lot of in-line Perl code interspersed between the HTML.

in-line Perl can be useful if you want a "quick" computation, e.g. insertion of the current year:

Example 5. Simple in-line perl code

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Very quick and dirty block of perl code -->

Copyright (c) <perl>(localtime())[5]+1900</perl> Foobar Gherkin corp.

</body>
</html>
```

Run

Which can be pretty handy, but looks a bit cumbersome - the tags interfere with the flow of the text, making it harder to read. For this reason in-line perl can also be flagged in a WebDyne page using the shortcuts !{!!}, or by the use of processing instructions (<*?* .. *?*>) e.g.:

Example 6. in-line code using alternative denotation

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- Same code with alternative denotation -->

The time is: !{! localtime() !}

<p>

The time is:  <? localtime() ?>


</body>
</html>
```

Run

The !{!!} denotation can also be used in tag attributes (processing instructions, and <perl> tags cannot):

Example 7. in-line code in tag attributes

```

```

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Perl code can be used in tag attributes also -->

<font color="!{! (qw(red blue green))[rand 3] !}">

Hello World

</font>
</body>
</html>
```

Run

6.3. Use of the <perl> tag for non-inline code.

Any code that is not co-mingled with the HTML of a document is *non-inline* code. It can be segmented from the content HTML using the __PERL__ delimiter, or by being kept in a completely different package and referenced as an external Perl subroutine call. An example of non-inline code:

Example 8.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Empty perl tag this time, but with method name as attribute -->

Hello World <perl method="hello"/>

</body>
</html>

__PERL__

sub hello { return localtime }
```

Run

Note that the <perl> tag in this example is explicitly closed and does not contain any content. However non-inline code can enclose HTML or text within the tags:

Example 9.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- The perl method will be called, but "Hello World" will not be displayed ! -->

<perl method="hello">
```

```
Hello World
</perl>

</body>
</html>

__PERL__

sub hello { return localtime() }
```

Run

But this is not very interesting so far - the "Hello World" text is not displayed when the example is run !

In order for text or HTML within a non-inline perl block to be displayed, it must be "rendered" into the output stream by the WebDyne engine. This is done by calling the `render()` method. Let's try that again:

Example 10.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- The perl method will be called, and this time the "Hello World" will be displayed-->

<perl method="hello">
Hello World
</perl>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();
        $self->render();

}
```

Run

And again, this time showing how to render the text block multiple times. Note that an array reference is returned by the Perl routine - this is fine, and is interpreted as an array of HTML text, which is concatenated and send to the browser.

Example 11.

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- The "Hello World" text will be rendered multiple times -->

<perl method="hello">
```

```
<p>
Hello World
</perl>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();
        my @html;
        for (0..3) { push @html, $self->render() };
        return \@html;
}
```

Run

6.4. Passing parameters to subroutines

The behaviour of a called __PERL__ subroutine can be modified by passing parameters which it can act on:

Example 12.

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- The "Hello World" text will be rendered with the param name -->

<perl method="hello" param="Alice"/>
<p>
<perl method="hello" param="Bob"/>
<p>

<!-- We can pass an array or hashref also - see variables section for more info on this syntax --

<perl method="hello_again" param="%{ firstname=>'Alice', lastname=>'Smith' }"/>

</body>
</html>

__PERL__

sub hello {

        my ($self, $param)=@_;
        return \"Hello world $param"
}

sub hello_again {

        my ($self, $param_hr)=@_;
        my $firstname=$param_hr->{'firstname'};
        my $lastname =$param_hr->{'lastname'};
        return \"Hello world $firstname $lastname";
```

```
}
```

Run

6.5. Notes about __PERL__ sections

Code in __PERL__ sections has some particular properties. __PERL__ code is only executed once. Subroutines defined in a __PERL__ section can be called as many times as you want, but the code outside of subroutines is only executed the first time a page is loaded. No matter how many times it is run, in the following code `$i` will always be 1:

Example 13.

```
<html>
<head><title>Hello World</title></head>
<body>

<perl method="hello"/>
<p>
<perl method="hello"/>

</body>
</html>

__PERL__

my $i=0;
$i++;

my $x=0;

sub hello {

        #  Note x may not increment as you expect because you will probably
        #  get a different Apache process each time you load this page
        #
        return sprintf("value of i: $i, value of x in PID $$: %s", $x++)
}
```

Run

Lexical variables are not accessible outside of the __PERL__ section due to the way perl's eval() function works. The following example will fail:

Example 14.

```
<html>
<head><title>Hello World</title></head>
<body>

The value of $i is !{! \$i !}

</body>
</html>
```

```
__PERL__

my $i=5;
```

Run

Package defined vars declared in a __PERL__ section do work, with caveats:

Example 15.

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- Does not work -->
The value of $i is !{! \$::i !}
<p>

<!-- Ugly hack, does work though -->
The value of $i is !{! \${__PACKAGE__.::i} !}
<p>

<!-- Probably best to just do this though -->
The value of $i is !{! &get_i() !}
<p>

<!-- Or this - see variable substitution section  -->
<perl method="render_i">
The value of $i is ${i}
</perl>

</body>
</html>

__PERL__

our $i=5;

sub get_i { \$i }

sub render_i { shift()->render(i=>$i) }
```

Run

See the Variables/Substitution section for clean ways to insert variable contents into the page.

6.6. Variables / Substitution

WebDyne starts to get more useful when variables are used to modify the content of a rendered text block. A simple example:

Example 16.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- The var ${time} will be substituted for the correspondingly named render parameter -->

<perl method="hello">
Hello World ${time}
</perl>

</body>
</html>

__PERL__

sub hello {
        my $self=shift();
        my $time=localtime();
        $self->render( time=>$time );
}
```

Run

Note the passing of the `time` value as a parameter to be substituted when the text is rendered.

Combine this with multiple call to the `render()` routine to display dynamic data:

Example 17.

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- Multiple variables can be supplied at once as render parameters -->

<perl method="hello0">
<p>
Hello World ${time}, loop iteration ${i}.
</perl>

<br>
<br>

<perl method="hello1">
<p>
Hello World ${time}, loop iteration ${i}.
</perl>

</body>
</html>

__PERL__

sub hello0 {

        my $self=shift();
        my @html;
        my $time=localtime();
        for (my $i=0; $i<10; $i++) {
                $self->render( time=>$time, i=>$i)
```

```
        };
        return \@html;
}

sub hello1 {

        #  Alternate syntax using print
        #
        my $self=shift();
        my $time=localtime();
        for (my $i=0; $i<3; $i++) {
                $self->render( itime=>$time, i=>$i)
        };
        return \undef
}
```

Run

Variables can also be used to modify tag attributes:

Example 18.

```
<html>
<head><title>Hello World</title></head>
<body>

<!-- Render paramaters also work in tag attributes -->

<perl method="hello">
<p>
<font color="${color}">
Hello World
</font>
</perl>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();
        my @html;
        for (0..3) {
                my $color=(qw(red green yellow blue orange))[rand 5];
                push @html, $self->render( color=>$color );
        }
        \@html;

}
```

Run

Other variable types are available also, including:

- `@{var,var,..}` for arrays, e.g. `@{'foo', 'bar'}`
- `%{key=>value, key=>value, ..}` for hashes e.g. `%{ a=>1, b=>2 }`
- `+{varname}` for CGI form parameters, e.g. `+{firstname}`
- `*{varname}` for environment variables, e.g. `*{HTTP_USER_AGENT}`

- `^{requestmethod}` for Apache request (`$r=Apache->request`) object methods, e.g. `^{protocol}`. Only available for in Apache/mod_perl, and only useful for request methods that return a scalar value.

The following template uses techniques and tags discussed later, but should provide an example of potential variable usage:

Example 19.

```
<html>
<head>
<title>Variables</title>
</head>
<body>

<!-- Environment variables -->

<p>
<!-- Short Way -->
Mod Perl Version: *{MOD_PERL}
<br>
<!-- Same as Perl code -->
Mod Perl Version:
<perl> \$ENV{'MOD_PERL'} </perl>


<!-- Apache request record methods. Only methods that return a scalar result are usable -->

<p>
<!-- Short Way -->
Request Protocol: ^{protocol}
<br>
<!-- Same as Perl code -->
Request Protocol:
<perl> my $self=shift(); my $r=$self->r(); \$r->protocol() </perl>


<!-- CGI params -->

<form>
Your Name: <textfield name="name" default="Test" size="12">
<submit name="Submit">
</form>
<p>
<!-- Short Way -->
You Entered: +{name}
<br>
<!-- Same as Perl code -->
You Entered:
<perl> my $self=shift(); my $cgi_or=$self->CGI(); \$cgi_or->param('name') </perl>
<br>
<!-- CGI vars are also loaded into the %_ global var, so the above is the same as -->
You Entered:
<perl> \$_{'name'} </perl>


<!-- Arrays -->
```

```
<form>
<p>
Favourite colour 1:
<popup_menu name="popup_menu" values="@{qw(red green blue)}">


<!-- Hashes -->

<p>
Favourite colour 2:
<popup_menu name="popup_menu"
values="%{red=>Red, green=>Green, blue=>Blue}">

</form>

</body>
</html>
```

Run

6.7. Integration with Lincoln Stein's CGI.pm module

WebDyne makes extensive use of Lincoln Stein's CGI.pm module. Almost any CGI.pm function that renders HTML tags can be called from within a WebDyne template. The manual page for CGI.pm contains the following synopsis example:

Example 20.

```
use CGI qw/:standard/;
   print header,
         start_html('A Simple Example'),
         h1('A Simple Example'),
         start_form,
         "What's your name? ",textfield('name'),p,
         "What's the combination?", p,
         checkbox_group(-name=>'words',
                        -values=>['eenie','meenie','minie','moe'],
                        -defaults=>['eenie','minie']), p,
         "What's your favorite color? ",
         popup_menu(-name=>'color',
                    -values=>['red','green','blue','chartreuse']),p,
         submit,
         end_form,
         hr;

    if (param()) {
        print "Your name is",em(param('name')),p,
              "The keywords are: ",em(join(", ",param('words'))),p,
```

If the example was ported to a WebDyne compatible page it might look something like this:

Example 21.

```
<!-- The same form from the CGI example -->
```

```
<start_html title="A simple example">
<h1>A Simple Example</h1>
<start_form>
<p>
What's your name ? <textfield name="name">
<p>
What's the combination ? <checkbox_group
name="words" values="@{qw(eenie meenie minie moe)}" defaults="@{qw(eenie minie)}">
<p>
What's your favourite color ? <popup_menu
name="color" values="@{qw(red green blue chartreuse)}">
<submit>
<end_form>
<hr>


<!-- This section only rendered when form submitted -->

<perl method="answers">
<p>
Your name is: <em>+{name}</em>
<p>
The keywords are: <em>${words}</em>
<p>
Your favorite color is: <em>+{color}</em>
</perl>

__PERL__

sub answers {

        my $self=shift();
        my $cgi_or=$self->CGI();
        if ($cgi_or->param()) {
                my $words=join(",", $cgi_or->param('words'));
                return $self->render( words=>$words )
        }
        else {
                return \undef;
        }

}
```

Run

6.8. More on CGI.pm generated tags

We can use CGI.pm tags such as <popup_menu>, instead of <select><option>...</select>. The following example:

```
<popup_menu value="%{red=>Red, green=>Green, blue=>Blue}"/>
```

is arguably easier to read than:

```
<select name="values" tabindex="1">
<option value="green">Green</option>
<option value="blue">Blue</option>
<option value="red">Red</option>
</select>
```

So there is some readability benefit, however the real advantage shows when we consider the next example:

Example 22.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Generate all country names for picklist -->

<form>

Your Country ?
<perl method="countries">
<popup_menu values="${countries_ar}">
</perl>

</form>
</body>
</html>

__PERL__

use Locale::Country;

sub countries {

        my $self=shift();
        my @countries = sort { $a cmp $b } all_country_names();
        $self->render( countries_ar=>\@countries );

}
```

Run

That saved a lot of typing !

6.9. Access to CGI query, form and keyword parameters

As mentioned above WebDyne makes extensive use of the CGI.pm Perl module. You can access a CGI object instance in any WebDyne template by calling the CGI() method:

Example 23.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Note use of CGI.pm derived textfield tag -->

<form>
Enter your name: <textfield name="name">
 
```

```
<submit>
</form>


<!-- And print out name if we have it -->

<perl method="hello">
Hello ${name}, pleased to meet you.
</perl>

</body>
</html>

__PERL__

sub hello {
        my $self=shift();

        #  Get CGI instance
        #
        my $cgi_or=$self->CGI();

        #  Use CGI.pm param() method. Could also use other
        #  methods like keywords(), Vars() etc.
        #
        my $name=$cgi_or->param('name');

        $self->render( name=>$name);
}
```

Run

From there you can all any method supported by the CGI.pm module - see the CGI.pm manual page (**man CGI**)

Since one of the most common code tasks is to access query parameters, WebDyne stores them in the `%_` global variable before any user defined Perl methods are called. For example:

Example 24.

```


<html>
<head><title>Hello World</title></head>
<body>
<p>
<form>
Enter your name: <textfield name="name">
<submit>
</form>

<!-- Quick and dirty, no perl code at all -->

<p>
Hello +{name}, pleased to meet you.


<!-- Traditional, using the CGI.pm param() call -->

<p>
<perl method="hello1">
```

```
Hello ${name}, pleased to meet you.
</perl>


<!-- Quicker method using %_ global var -->

<p>
<perl method="hello2">
Hello ${name}, pleased to meet you.
</perl>


<!-- Quick and dirty using inline Perl -->

<p>
Hello !{! \$_{name} !}, pleased to meet you.

</body>
</html>


__PERL__

sub hello1 {
        my $self=shift();
        my $cgi_or=$self->CGI();
        my $name=$cgi_or->param('name');
        $self->render( name=>$name);
}

sub hello2 {

        my $self=shift();

        #  Quicker method of getting name param
        #
        my $name=$_{'name'};
        $self->render( name=>$name);
}
```

Run

6.10. Quick Pages using CGI.pm's <start_html><end_html> tags

For rapid development you can take advantage of CGI.pm's <start_html> and <end_html> tags. The following page generates compliant HTML (view the page source after loading it to see for yourself):

Example 25.

```
<start_html title="Quick Page">
The time is: !{! localtime() !}
<end_html>
```

Run

The <start_html> tag generates all the <html>, <head>, <title> tags etc needed for a valid HTML page plus an opening body tag. Just enter the body content, then finish with <end_html> to generate the closing <body>

and <html> tags. See the CGI.pm manual page for more information.

7. Advanced Usage

A lot of tasks can be achieved just using the basic features detailed above. However there are more advanced features that can make life even easier

7.1. Blocks

Blocks are a powerful dynamic content generation tool. WebDyne can render arbitrary blocks of text or HTML within a page, which makes generation of dynamic content generally more readable than similar output generated within Perl code. An example:

Example 26.

```
<html>
<head>
<title>Blocks</title>
</head>
<body>
<p>

<form>
2 + 2 = <textfield name="sum">
<submit>
</form>

<p>
<perl method="check">


<!-- Each block below is only rendered if specifically requested by the Perl code -->

<block name="pass">
Yes, +{sum} is the correct answer ! Brilliant ..
</block>

<block name="fail">
I am sorry .. +{sum} is not correct .. Please try again !
</block>

<block name="silly">
Danger, does not compute ! .. "+{sum}" is not a number !
</block>

<p>
Thanks for playing !

</perl>

</body>
</html>

__PERL__
```

```
sub check {

        my $self=shift();

        if ((my $ans=$_{'sum'}) == 4) {
                $self->render_block('pass')
        }
        elsif ($ans=~/^[0-9.]+$/) {
                $self->render_block('fail')
        }
        elsif ($ans) {
                $self->render_block('silly')
        }

        #  Blocks aren't displayed until whole section rendered
        #
        return $self->render();

}
```

Run

There can be more than one block with the same name - any block with the target name will be rendered:

Example 27.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
<form>
Enter your name: <textfield name="name">
 
<submit>
</form>

<perl method="hello">


<!-- The following block is only rendered if we get a name - see the perl
        code -->

<block name="greeting">
Hello +{name}, pleased to meet you !
<p>
</block>


<!-- This text is always rendered - it is not part of a block -->

The time here is !{! localtime() !}


<!-- This block has the same name as the first one, so will be rendered
        whenever that one is -->

<block name="greeting">
<p>
It has been a pleasure to serve you, +{name} !
```

```
</block>


</perl>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();

        #  Only render greeting blocks if name given. Both blocks
        #  will be rendered, as the both have the name "greeting"
        #
        if ($_{'name'}) {
                $self->render_block('greeting');
        }

        $self->render();
}
```

Run

Like any other text or HTML between <perl> tags, blocks can take parameters to substitute into the text:

Example 28.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
<form>
Enter your name: <textfield name="name">
 
<submit>
</form>

<perl method="hello">


<!-- This block will be rendered multiple times, the output changing depending
        on the variables values supplied as parameters -->

<block name="greeting">
${i} .. Hello +{name}, pleased to meet you !
<p>
</block>

The time here is <? localtime() ?>

</perl>

</body>
</html>

__PERL__
```

```
sub hello {

        my $self=shift();

        #  Only render greeting blocks if name given. Both blocks
        #  will be rendered, as the both have the name "greeting"
        #
        if ($_{'name'}) {
               render_block('greeting', i=>$i );
                }
        }

        $self->render();
}
```

Run

Blocks have a non-intuitive feature - they still display even if they are outside of the <perl> tags that made the call to render them. e.g. the following is OK:

Example 29.

```
<html>
<head><title>Hello World</title></head>

<body>

<!-- Perl block with no content -->
<perl method="hello">
</perl>

<p>

<!-- This block is not enclosed within the <perl> tags, but will still render -->
<block name="hello">
Hello World
</block>

<p>

<!-- So will this one -->
<block name="hello">
Again
</block>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();
        $self->render_block('hello');

}
```

Run

You can mix the two styles:

Example 30.

```
<html>
<head><title>Hello World</title></head>

<body>
<perl method="hello">

<!-- This block is rendered -->
<block name="hello">
Hello World
</block>

</perl>

<p>
<!-- This one is not -->
<block name="hello">
Again
</block>

</body>
</html>

__PERL__

sub hello {

        my $self=shift();
        $self->render_block('hello');
        $self->render();

}
```

Run

7.2. File inclusion

You can include other file fragments at compile time using the include tag:

Example 31.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
The services file on this machine:
<pre>
<include file="/etc/services">
</pre>
</body>
</html>
```

Run

If the file name is not an absolute path name is will be loaded relative to the directory of the parent file. For example if file "bar.psp" incorporates the tag<include file="foo.psp"/> it will be expected that "foo.psp" is in

the same directory as "bar.psp".

The include tag pulls in the target file at compile time. Changes to the included file after the WebDyne page is run the first time (resulting in compilation) are not reflected in subsequent output. Thus the include tag should not be seen as a shortcut to a pseudo Content Management System. For example <include file="latest_news.txt"/> will probably not behave in the way you expect. The first time you run it the latest news is displayed. However updating the "latest_news.txt" file will not result in changes to the output (it will be stale).

There are betters ways to build a CMS with WebDyne - use the include tag sparingly !

7.3. Static Sections

Sometimes you want to generate dynamic output in a page once only (e.g. a last modified date, a sidebar menu etc.) Using WebDyne this can be done with Perl or CGI code flagged with the "static" attribute. Any dynamic tag so flagged will be rendered at compile time, and the resulting output will become part of the compiled page - it will not change on subsequent page views, or have to be re-run each time the page is loaded. An example:

Example 32.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
Hello World
<hr>


<!-- Note the static attribute -->

<perl method="mtime" static="1">
<em>Last Modified: </em>${mtime}
</perl>

</body>
</html>

__PERL__

sub mtime {

        my $self=shift();
        my $r=$self->request();

        my $srce_pn=$r->filename();
        my $srce_mtime=(stat($srce_pn))[9];
        my $srce_localmtime=localtime $srce_mtime;

        return $self->render( mtime=>$srce_localmtime )

}
```

Run

In fact the above page will render very quickly because it has no dynamic content at all once the <perl> content is flagged as static. The WebDyne engine will recognise this and store the page as a static HTML file in its cache. Whenever it is called WebDyne will use the Apache lookup_file() function to return the page as if it was just serving up static content.

You can check this by looking at the content of the WebDyne cache directory (usually /var/webdyne/cache). Any file with a ".html" extension represents the static version of a page.

Of course you can still mix static and dynamic Perl sections:

Example 33.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>
Hello World
<p>

<!-- A normal dynamic section - code is run each time page is loaded -->

<perl method="localtime">
Current time: ${time}
</perl>
<hr>

<!-- Note the static attribute - code is run only once at compile time -->

<perl method="mtime" static="1">
<em>Last Modified: </em>${mtime}
</perl>


</body>
</html>

__PERL__


sub localtime {

        shift()->render(time=>scalar localtime);

}


sub mtime {

        my $self=shift();
        my $r=$self->request();

        my $srce_pn=$r->filename();
        my $srce_mtime=(stat($srce_pn))[9];
        my $srce_localmtime=localtime $srce_mtime;

        return $self->render( mtime=>$srce_localmtime )
```

```
}
```

Run

If you want the whole pages to be static, then flagging everything with the "static" attribute can be cumbersome. There is a special meta tag which flags the entire page as static:

Example 34.

```
<html>
<head>

<!-- Special meta tag -->
<meta name="WebDyne" content="static=1">

<title>Hello World</title>
</head>
<body>
<p>
Hello World
<hr>


<!-- A normal dynamic section, but because of the meta tag it will be frozen
        at compile time -->

<perl method="localtime">
Current time: ${time}
</perl>

 

<!-- Note the static attribute. It is redundant now the whole page is flagged
        as static - it could be removed safely. -->

<perl method="mtime" static="1">
<em>Last Modified: </em>${mtime}
</perl>


</body>
</html>

__PERL__


sub localtime {

        shift()->render(time=>scalar localtime);

}


sub mtime {

        my $self=shift();
        my $r=$self->request();

        my $srce_pn=$r->filename();
        my $srce_mtime=(stat($srce_pn))[9];
```

```
        my $srce_localmtime=localtime $srce_mtime;

        return $self->render( mtime=>$srce_localmtime )

}
```

Run

If you don't like the idea of setting the static flag in meta data, then "using" the special package "WebDyne::Static" will have exactly the same effect:

Example 35.

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>
Hello World
<hr>

<perl method="localtime">
Current time: ${time}
</perl>

 

<perl method="mtime">
<em>Last Modified: </em>${mtime}
</perl>

</body>
</html>

__PERL__


#  Makes the whole page static
#
use WebDyne::Static;


sub localtime {

        shift()->render(time=>scalar localtime);

}


sub mtime {

        my $self=shift();
        my $r=$self->request();

        my $srce_pn=$r->filename();
        my $srce_mtime=(stat($srce_pn))[9];
        my $srce_localmtime=localtime $srce_mtime;

        return $self->render( mtime=>$srce_localmtime )
```

```
}
```

Run

If the static tag seems trivial consider the example that displayed country codes:

Example 36.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Generate all country names for picklist -->

<form>

Your Country ?
<perl method="countries">
<popup_menu values="${countries_ar}">
</perl>

</form>
</body>
</html>

__PERL__

use Locale::Country;

sub countries {

        my $self=shift();
        my @countries = sort { $a cmp $b } all_country_names();
        $self->render( countries_ar=>\@countries );

}
```

Run

Every time the above example is viewed the Country Name list is generated dynamically via CGI.pm and the Locale::Country module (on a sample machine Apache Bench measured the output at around 55 pages/sec). This is a waste of resources because the list changes very infrequently. We can keep the code neat but gain a lot of speed by adding the static tag attribute:

Example 37.

```
<html>
<head><title>Hello World</title></head>
<body>
<p>

<!-- Generate all country names for picklist -->

<form>

Your Country ?
```

```
<perl method="countries" static="1">

<!-- Note the addition of the static attribute -->

<popup_menu values="${countries_ar}">
</perl>

</form>
</body>
</html>

__PERL__

use Locale::Country;

sub countries {

        my $self=shift();
        my @countries = sort {$a cmp $b} all_country_names();
        $self->render( countries_ar=>\@countries );

}
```

Run

By simply adding the "static" attribute output on a sample machine increased from 55 Pages/sec to 280 Pages/sec ! Judicious use of the static tag in places with slow changing data can markedly increase efficiency of the WebDyne engine.

7.4. Caching

WebDyne has the ability to cache the compiled version of a dynamic page according to specs you set via the API. When coupled with pages/blocks that are flagged as static this presents some powerful possibilities.

Caching will only work if `$WEBDYNE_CACHE_DN` is defined and set to a directory that the web server has write access to. If caching does not work check that $`WEBDYNE_CACHE_DN` is defined and permissions set correctly for your web server.

There are many potential examples, but consider this one: you have a page that generates output by making a complex query to a database, which takes a lot of CPU and disk IO resources to generate. You need to update the page reasonably frequently (e.g. a weather forecast, near real time sales stats), but can't afford to have the query run every time someone view the page.

WebDyne allows you to configure the page to cache the output for a period of time (say 5 minutes) before re-running the query. In this way users sees near real-time data without imposing a high load on the database/Web server.

WebDyne knows to enable the caching code by looking for a meta tag, or by loading the `WebDyne::Cache` module in a __PERL__ block.

The cache code can command WebDyne to recompile a page based on any arbitrary criteria it desires. As an example the following code will recompile the page every 10 seconds. If viewed in between refresh intervals WebDyne will serve up the cached HTML result using Apache r$->lookup_file() or the FCGI equivalent, which is very fast.

Try it by running the following example and clicking refresh a few times over a 20 second interval

Example 38.

```
<html>
<head>
<title>Caching</title>
<!-- Set static and cache meta parameters -->
<meta name="WebDyne" content="cache=&cache;static=1">
</head>

<body>
<p>

This page will update once every 10 seconds.

<p>

Hello World !{! localtime() !}

</body>
</html>

__PERL__


#  The following would work in place of the meta tags
#
#use WebDyne::Static;
#use WebDyne::Cache (\&cache);


sub cache {

        my $self=shift();

        #  Get cache file mtime (modified time)
        #
        my $mtime=${ $self->cache_mtime() };


        #  If older than 10 seconds force recompile
        #
        if ((time()-$mtime) > 10) {
                $self->cache_compile(1)
        };

        #  Done
        #
        return \undef;

}
```

Run

WebDyne uses the return value of the nominated cache routine to determine what UID (unique ID) to assign to the page. In the above example we returned \undef, which signifies that the UID will remain unchanged.

You can start to get more advanced in your handling of cached pages by returning a different UID based on some arbitrary criteria. To extend our example above: say we have a page that generated sales figures for a given month. The SQL code to do this takes a long time, and we do not want to hit the database every time someone loads up the page. However we cannot just cache the output, as it will vary depending on the month

the user chooses. We can tell the cache code to generate a different UID based on the month selected, then cache the resulting output.

The following example simulates such a scenario:

Example 39.

```
<!-- Start to cheat by using start/end_html tags to save space -->

<start_html>
<form method="GET">
Get sales results for: <popup_menu name="month" values="@{qw(January February March)}">
<submit>
</form>

<perl method="results">
Sales results for +{month}: $${results}
</perl>

<hr>
This page generated: !{! localtime() !}
<end_html>

__PERL__

use WebDyne::Static;
use WebDyne::Cache (\&cache);

my %results=(

        January         => 20,
        February        => 30,
        March           => 40
);

sub cache {

        #  Return UID based on month
        #
        my $uid=undef;
        if (my $month=$_{'month'}) {

                #  Make sure month is valid
                #
                $uid=$month if defined $results{$month}

        }
        return \$uid;

}


sub results {

        my $self=shift();
        if (my $month=$_{'month'}) {

                #  Could be a really long complex SQL query ...
```

```
                #
                my $results=$results{$month};


                #  And display
                #
                return $self->render(results => $results);
        }
        else {
                return \undef;
        }

}
```

Run

Take care when using user-supplied input to generate the page UID. There is no inbuilt code in WebDyne to limit the number of UID's associated with a page. Unless we check it, a malicious user could potentially DOS the server by supplying endless random "months" to the above page with a script, causing WebDyne to create a new file for each UID - perhaps eventually filling the disk partition that holds the cache directory. That is why we check the month is valid in the code above.

8. Error Handling

8.1. Error Messages

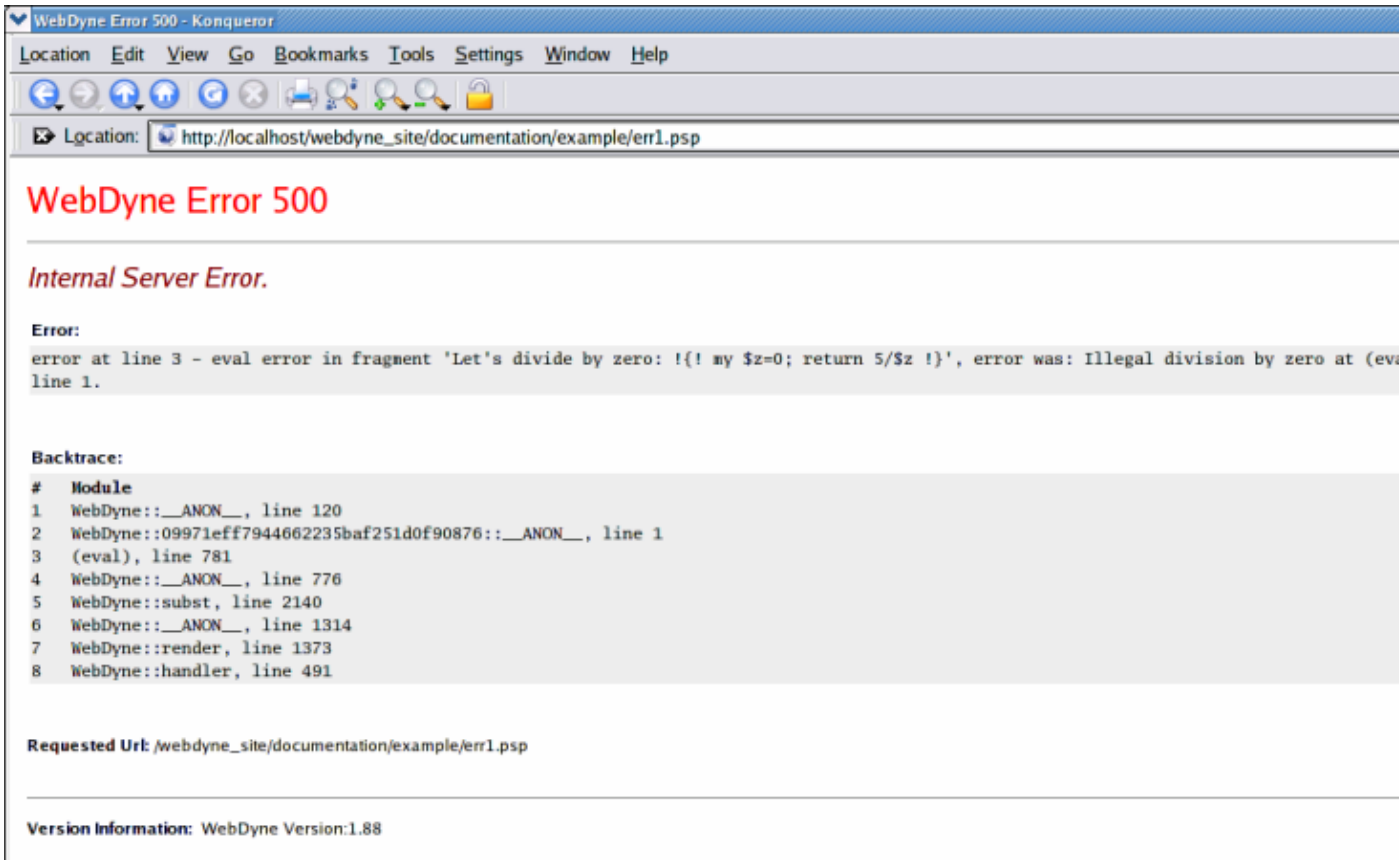
Sooner or later something is going to go wrong in your code. If this happens WebDyne will generate an error showing what the error was and attempting to give information on where it came from: Take the following example:

Example 40.

```
<start_html title="Error">
Let's divide by zero: !{! my $z=0; return 5/$z !}
<end_html>
```

Run

If you run the above example an error message will be displayed:.



In this example the backtrace is not particularly useful because the error occurred within in-line code, so all references in the backtrace are to internal WebDyne modules. However the code fragment clearly shows the line with the error, and the page line number where the error occurred (line 3) is given at the start of the message. The reference to "(eval 268) line 1" is a red herring - it is the 268th eval performed by this perl process, and the error occurred in line 1 of the text that the eval was passed - standard perl error text, but not really helpful here.

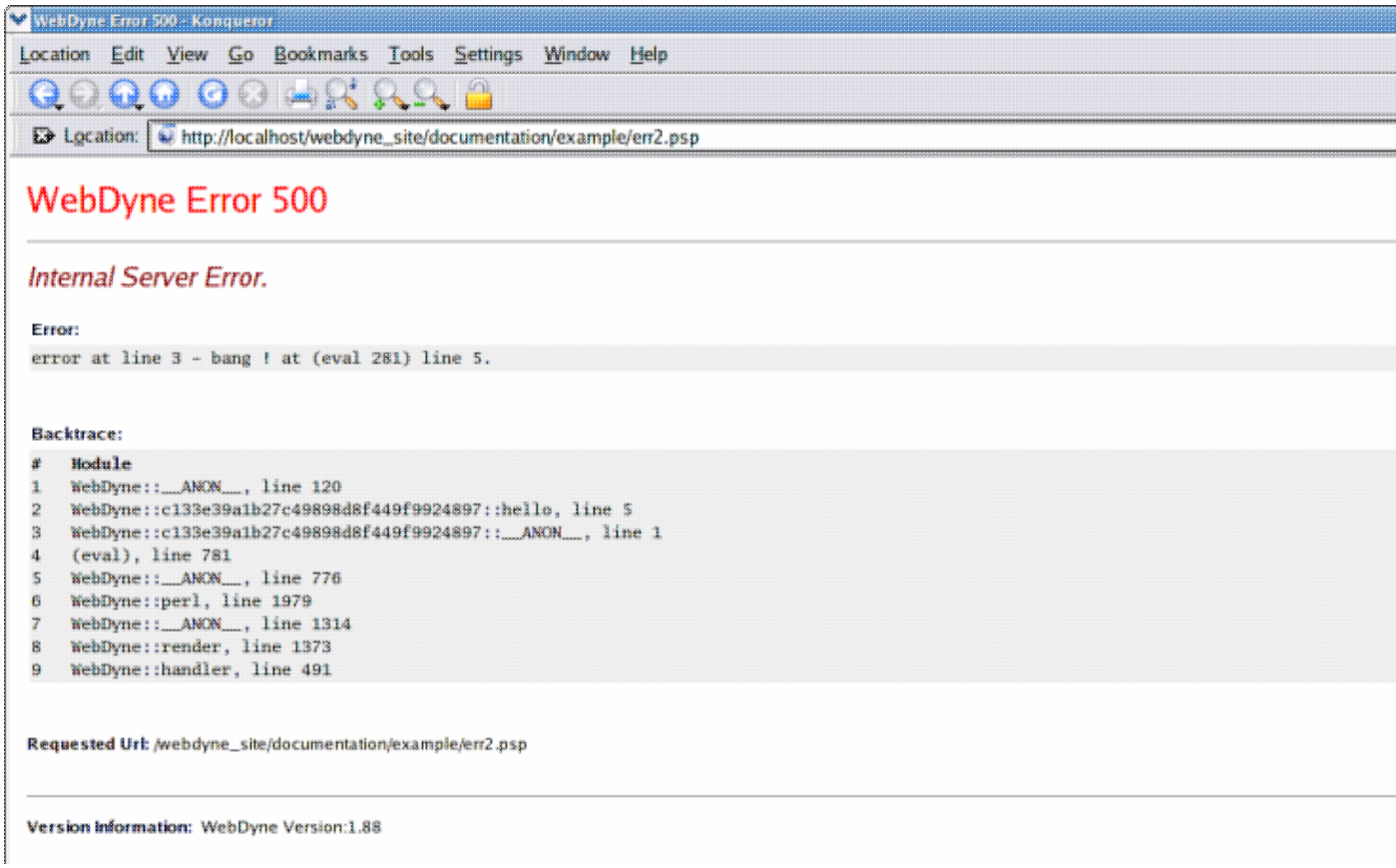
If we have a look at another example:

Example 41.

```
<start_html title="Error">
<perl method="hello"/>
<end_html>

__PERL__

sub hello {

        die('bang !');

}
```

Run

And the corresponding screen shot:



The backtrace is somewhat more helpful. Looking through the backtrace we can see that the error occurred in the "hello" subroutine (invoked at line 3 of the page) on line 5 - In this case "line 5" means the 5th line down from the __PERL__ delimiter. The 32 digit hexadecimal number is the page unique ID - it is different for each page. WebDyne runs the code for each page in a package name space that includes the page's UID - in this way pages with identical subroutine names (e.g. two pages with a "hello" subroutine) can be accommodated with no collision.

8.2. Exceptions

Errors (exceptions) can be generated within a WebDyne page in two ways:

- By calling die() as shown in example above.
- By returning an error message via the err() method, exported by default.

Examples

```
__PERL__


#  Good
#
sub hello {

    return err('no foobar') if !$foobar;

}
```

```
# Also OK
#
sub hello {

    return die('no foobar') if !$foobar;

}
```

8.3. Error Checking

So far all the code examples have just assumed that any call to a WebDyne API method has been successful - no error checking is done. WebDyne always returns "undef" if an API method call fails - which should be checked for after every call in a best practice scenario.

Example 42.

```
<start_html title="Error">
<perl method="hello">

Hello World ${foo}

</perl>
<end_html>

__PERL__

sub hello {

        #  Check for error after calling render function
        #
        shift()->render( bar=> 'Again') || return err();

}
```

Run

You can use the err() function to check for errors in WebDyne Perl code associated with a page, e.g.:

Example 43.

```
<start_html title="Error">
<form>
<submit name="Error" value="Click here for error !">
</form>
<perl method="foo"/><end_html>

__PERL__

sub foo {

&bar() || return err();
        \undef;

}
```

```
sub bar {

        return err('bang !') if $_{'Error'};
        \undef;
}
```

Run

Note that the backtrace in this example shows clearly where the error was triggered from.

9. WebDyne API

9.1. WebDyne tags

Reference of WebDyne tags and supported attributes

<perl>
: Run Perl code either in-line (between the <perl>..</perl>) tags, or non-inline via the method attribute

  method=[Module::Name]::method
  : *Optional*. Call an external perl subroutine in a pre-loaded module, or a subroutine in a __PERL__ block at then of the HTML: file

  param=scalar|array|hash
  : *Optional*. Parameters to be supplied to perl routine. Supply array and hash using @{1,2} and %{a=>1, b=>2} conventions respectively.

  static=1
  : *Optional*. This Perl code to be run once only and the output cached for all subsequent requests.

<block>
: Block of HTML code to be optionally rendered if desired by call to `render_block` Webdyne method:

  name=identifier
  : *Mandatory*. The name for this block of HTML.

  display=1
  : *Optional.* Force display of this block even if not invoked by render_block WebDyne method. Useful for prototyping.

  static=1
  : *Optional*. This block rendered once only and the output cached for all subsequent requests

<include>
: Include HTML or text from an external file

  file=filename
  : *Mandatory*. Name of file we want to include. Can be relative to current directory or absolute path.

  head=1
  : *Optional*. File is an HTML file and we want to include just the <head> section

  body=1
  : *Optional*. File is an HTML file and we want to include just the <body> section.

  block=blockname
  : *Optional*. File is a .psp file and we want to include a <block> section from that file.

<dump>

Display CGI paramters in dump format via CGI->Dump call. Useful for debugging. Only rendered if `$WEBDYNE_DUMP_FLAG` global set to 1 in WebDyne constants (see below)

display=1

Optional. Force display even if `$WEBDYNE_DUMP_FLAG` global not set

9.2. WebDyne methods

When running Perl code within a WebDyne page the very first parameter passed to any routine (in-line or in a __PERL__ block) is an instance of the WebDyne page object (referred to as `$self` in most of the examples). All methods return undef on failure, and raise an error using the `err()` function. The following methods are available to any instance of the WebDyne object:

`CGI()`

Returns an instance of the CGI.pm object for the current request.

`r(), request()`

Returns an instance of the Apache request object.

`render( key=>value, key=>value, .. )`

Called to render the text or HTML between <perl>..</perl> tags. Optional key and value pairs will be substituted into the output as per the variable section. Returns a scalar ref of the resulting HTML.

`render_block( blockname, key=>value, key=>value, ..).`

Called to render a block of text or HTML between <block>..</block> tags. Optional key and value pairs will be substituted into the output as per the variable section. Returns scalar ref of resulting HTML if called with from <perl>..</perl> section containing the block to be rendered, or true (\undef) if the block is not within the <perl>..</perl> section (e.g. further into the document, see the block section for an example).

`redirect({ uri=>uri | file=>filename | html=>\html_text })`

Will redirect to URI or file nominated, or display only nominated text. Any rendering done to prior to this method is abandoned.

`cache_inode( seed )`

Returns the page unique ID (UID). Called inode for legacy reasons, as that is what the UID used to be based on. If a seed value is supplied a new UID will be generated based on an MD5 of the seed. Seed only needs to be supplied if using advanced cache handlers.

`cache_mtime( uid )`

Returns the mtime (modification time) of the cache file associated with the optionally supplied UID. If no UID supplied the current one will be used. Can be used to make cache compile decisions by WebDyne::Cache code (e.g if page > x minutes old, recompile).

`cache_compile( )`

Force recompilation of cache file. Can be used in cache code to force recompilation of a page, even if it is flagged static. Returns current value if no parameters supplied, or sets if parameter supplied.

`no_cache()`

Send headers indicating that the page is not be cached by the browser or intermediate proxies. By default WebDyne pages automatically set the no-cache headers, although this behaviour can be modified by clearing the `$WEBDYNE_NO_CACHE` variable and using this function

`meta()`

Return a hash ref containing the meta data for this page. Alterations to meta data are persistent for this process, and carry across Apache requests (although not across different Apache processes)

9.3. WebDyne Constants

Constants defined in the WebDyne::Constant package control various aspects of how WebDyne behaves. Constants can be modified globally by altering a system file (`/etc/webdyne.pm` under Linux distros), or by altering configuration parameters within the Apache or lighttpd/FastCGI web servers.

9.3.1. Global constants file

WebDyne will look for a system constants file under `/etc/webdyne.pm` and set package variables according to values found in that file. The file is in Perl Data::Dumper format, and takes the format:

```
# sample /etc/webdyne.pm file
#
$VAR1={
        WebDyne::Constant => {

                WEBDYNE_CACHE_DN       => '/data1/webdyne/cache',
                WEBDYNE_STORE_COMMENTS => 1,
                #  ... more variables for WebDyne package

       },

       WebDyne::Session::Constant => {

                WEBDYNE_SESSION_ID_COOKIE_NAME => 'session_cookie',
                #  ... more variables for WebDyne::Session package

       },

};
```

The file is not present by default and should be created if you wish to change any of the WebDyne constants from their default values.

Always check the syntax of the `/etc/webdyne.pm` file after editing by running **perl -c -w /etc/webdyne.pm** to check that the file is readable by Perl.

9.3.2. Setting WebDyne constants in Apache

WebDyne constants can be set in an Apache httpd.conf file using the PerlSetVar directive:

```
PerlHandler     WebDyne
PerlSetVar      WEBDYNE_CACHE_DN                '/data1/webdyne/cache'
PerlSetVar      WEBDYNE_STORE_COMMENTS          1

#  From WebDyne::Session package
#
PerlSetVar      WEBDYNE_SESSION_ID_COOKIE_NAME  'session_cookie'
```

WebDyne constants cannot be set on a per-location or per-directory basis - they are read from the top level of the config file and set globally.

Some 1.x versions of mod_perl do not read PerlSetVar variables correctly. If you encounter this problem use a <Perl>..</Perl> section in the httpd.conf file, e.g.:

```
# Mod_perl 1.x

PerlHandler     WebDyne
<Perl>
$WebDyne::Constant::WEBDYNE_CACHE_DN='/data1/webdyne/cache';
$WebDyne::Constant::WEBDYNE_STORE_COMMENTS=1;
$WebDyne::Session::Constant::WEBDYNE_SESSION_ID_COOKIE_NAME='session_cookie';
</Perl>
```

9.3.3. Setting WebDyne constants in lighttpd/FastCGI

WebDyne constants can be set in lighttpd/FastCGI using the bin-environment directive. Here is a sample lighttpd.conf file showing WebDyne constants:

```
fastcgi.server = ( ".psp" =>
                   ( "localhost" =>
                      (
                        "socket" => "/tmp/psp-fastcgi.socket",
                        "bin-path" => "/opt/webdyne/bin/wdfastcgi",
                        "bin-environment"     => (
                          "WEBDYNE_CACHE_DN   => "/data1/webdyne/cache"
                        )
                      )
                   )
                 )
```

9.3.4. Constants Reference

The following constants can be altered to change the behaviour of the WebDyne package. All these constants reside in the `WebDyne::Constant` package namespace.

`$WEBDYNE_CACHE_DN`
: The name of the directory that will hold partially compiled WebDyne cache files. Must exist and be writable by the Apache process

`$WEBDYNE_STARTUP_CACHE_FLUSH`
: Remove all existing disk cache files at Apache startup. 1=yes (default), 0=no. By default all disk cache files are removed at startup, and thus pages must be recompiled again the first time they are viewed. If you set this to 0 (no) then disk cache files will be saved between startups and pages will not need to be re-compiled if Apache is restarted.

`$WEBDYNE_CACHE_CHECK_FREQ`
: Check the memory cache after this many request (per-process counter). default=256. After this many requests a housekeeping function will check compiled pages that are stored in memory and remove old ones according to the criteria below.

`$WEBDYNE_CACHE_HIGH_WATER`
: Remove compiled from pages from memory when we have more than this many. default=64

`$WEBDYNE_CACHE_LOW_WATER`
: After reaching HIGH_WATER delete until we get down to this amount. default=32

`$WEBDYNE_CACHE_CLEAN_METHOD`
: Clean algorithm. default=1, means least used cleaned first, 0 means oldest last view cleaned first

`$WEBDYNE_EVAL_SAFE`
: default=0 (no), If set to 1 means eval in a Safe.pm container.

`$WEBDYNE_EVAL_SAFE_OPCODE_AR`

The opcode set to use in Safe.pm evals (see the Safe man page). Defaults to "[':default']". Use [&Opcode::full_opset()] for the full opset. CAUTION Use of WebDyne with Safe.pm not comprehensively tested.

`$WEBDYNE_EVAL_USE_STRICT`

The string to use before each eval. Defaults to "use strict qw(vars);". Set to undef if you do not want strict.pm. In Safe mode this becomes a flag only - set undef for "no strict", and non-undef for "use strict" equivalence in a Safe mode (checked under Perl 5.8.6 only, results in earlier versions of Perl may vary).

`$WEBDYNE_STRICT_VARS`

Check if a var is declared in a render block (e.g $ {foo}) but not supplied as a render parameter. If so will throw an error. Set to 0 to ignore. default=1

`$WEBDYNE_DUMP_FLAG`

If 1, any instance of the special <dump> tag will print out results from CGI->dump(). Use when debugging forms. default=0

`$WEBDYNE_DTD`

The DTD to place at the top of a rendered page. Defaults to: <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

`$WEBDYNE_HTML_PARAM`

attributes for the <html> tag, e.g. { lang =>'en-US' }. undef by default

`$WEBDYNE_COMPILE_IGNORE_WHITESPACE`

Ignore source file whitespace as per HTML::TreeBuilder ignore_ignorable_whitespace function. Defaults to 1

`$WEBDYNE_COMPILE_NO_SPACE_COMPACTING`

Do not compact source file whitespace as per HTML::TreeBuilder no_space_compacting function. Defaults to 0

`$WEBDYNE_STORE_COMMENTS`

By default comments are not rendered. Set to 1 to store and display comments from source files. Defaults to 0

`$WEBDYNE_NO_CACHE`.

WebDyne should send no-cache HTTP headers. Set to 0 to not send such headers. Defaults to 1

`$WEBDYNE_DELAYED_BLOCK_RENDER`

By default WebDyne will render blocks targeted by a render_block() call, even those that are outside the originating <perl>..</perl> section that made the call. Set to 0 to not render such blocks. Defaults to 1

`$WEBDYNE_WARNINGS_FATAL`

If a programs issues a warning via warn() this constant determines if it will be treated as a fatal error. Default is 0 (warnings not fatal). Set to 1 if you want any warn() to behave as if die() had been called..

`$WEBDYNE_CGI_DISABLE_UPLOADS`

Disable CGI.pm file uploads. Defaults to 1 (true - do not allow uploads).

`$WEBDYNE_CGI_POST_MAX`

Maximum size of a POST request. Defaults to 512Kb

`$WEBDYNE_ERROR_TEXT`

Display simplified errors in plain text rather than using HTML. Useful in interal WebDyne development only. By default this is 0 => the HTML error handler will be used.

`$WEBDYNE_ERROR_SHOW`

Display the error message. Only applicable in the HTML error handler

`$WEBDYNE_ERROR_SOURCE_CONTEXT_SHOW`

Display a fragment of the .psp source file around where the error occurred to give some context of where the error happened. Set to 0 to not display context.

`$WEBDYNE_ERROR_SOURCE_CONTEXT_LINES_PRE`

Number of lines of the source file before the error occurred to display. Defaults to 4

`$WEBDYNE_ERROR_SOURCE_CONTEXT_LINES_POST`
: Number of lines of the source file after the error occurred to display. Defaults to 4

`$WEBDYNE_ERROR_SOURCE_CONTEXT_LINE_FRAGMENT_MAX`
: Max line length to show. Defaults to 80 characters.

`$WEBDYNE_ERROR_BACKTRACE_SHOW`
: Show a backtrace of modules through which the error propagated. On by default, set to 0 to disable,

`$WEBDYNE_ERROR_BACKTRACE_SHORT`
: Remove WebDyne internal modules from backtrace. Off by default, set to 1 to enable.

`$WEBDYNE_AUTOLOAD_POLLUTE`
: When a method is called from a <perl> routine the WebDyne AUTOLOAD method must search multiple modules for the method owner. Setting this flag to 1 will pollute the WebDyne name space with the method name so that AUTOLOAD is not called if that method is used again (for the duration of the Perl process, not just that call to the page). This is dangerous and can cause confusion if different modules use the same name. In very strictly controlled environments - and ebev then only in some cases - it can result is faster throughput. Off by default, set to 1 to enable.

Extension modules (e.g., WebDyne::Session) have their own constants - see each package for details.

9.4. WebDyne Directives

A limited number of directives are are available which change the way WebDyne processes pages. Directives are set in either the Apache or lighttpd .conf files and can be set differently per location. At this stage only one directive applies to the core WebDyne module:

`WebDyneHandler`
: The name of the handler that WebDyne should invoke instead of handling the page internally. The only other handler available today is WebDyne::Chain.

This directive exists primarily to allow lighttpd/FastCGI to invoke WebDyne::Chain as the primary handler. An example from the lighttpd.conf file (see the WebDyne::Chain documentation for information on the **WebDyneChain** directive):

```
fastcgi.server = ( ".psp" =>
                   ( "localhost" =>
                      (
                        "socket" => "/tmp/psp-fastcgi.socket",
                        "bin-path" => "/opt/webdyne/bin/wdfastcgi",
                        "bin-environment"     => (
                          #  Handle WebDyne requests via WebDyne::Chain, which in turn will
                          #  pass all requests through WebDyne::Session so that a unique session
                          #  cookie is assigned to each user.
                          "WebDyneHandler"    => "WebDyne::Chain",
                          "WebDyneChain"      => "WebDyne::Session",
                        )
                      )
                   )
                 )
```

It can be used in Apache httpd.conf files, but is not very efficient:

```
#  This will work, but is not very efficient
#
<location /shop/>
PerlHandler     WebDyne
PerlSetVar      WebDyneHandler              'WebDyne::Chain'
```

```
PerlSetVar      WebDyneChain                'WebDyne::Session'
</location>


#  This is the same, and is more efficient
#
<location /shop/>
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain                'WebDyne::Session'
</location>
```

As with Apache you can do per-location/directory configuration of WebDyne, however the configuration is a little more complex. The example below shows how to set different directives on a per location basis, using WebDyne and WebDyne::Chain directives as examples:

```
$HTTP["url"] =~"^/proxcube/" {

        server.document-root = "/opt/proxcube/lib/perl5/site_perl/5.8.6/ProxCube/HTML/html/",

        fastcgi.server = (

                ".psp" => (
                        "localhost" => (
                                "socket"      => "/tmp/psp-fastcgi-proxube.socket",
                                "bin-path"    => "/opt/webdyne/bin/wdfastcgi",
                                "bin-environment" => (
                                        "WebDyneHandler"  => "WebDyne::Chain",
                                        "WebDyneChain"    => "ProxCube::HTML ProxCube::License"
                                        "WebDyneTemplate" => "/opt/proxcube/lib/perl5/site_perl/5
                                        "WebDyneFilter"   => "WebDyne::Template WebDyne::CGI",
                                        "MenuData"        => "/opt/proxcube/lib/perl5/site_perl/5
                                        "WebDyneLocation" => "/proxcube/"
                                )
                        )
                )
        )

},


$HTTP["url"] =~"^/example/" {

        server.document-root = "/var/www/html/",

        fastcgi.server = (

                ".psp" => (
                        "localhost" => (
                                "socket"      => "/tmp/psp-fastcgi-example.socket",
                                "bin-path"    => "/opt/webdyne/bin/wdfastcgi",
                                "bin-environment" => (
                                        "WebDyneLocation" => "/example/"
                                )
                        )
                )
        )
}
```


As noted in the configuration file the socket file names must be unique for each location that you want different WebDyne directives/constants for.

10. Miscellaneous

10.1. Command Line Utilities

Command line utilities are fairly basic at this stage. By default they are located in `/opt/webdyne/bin`.

wdapacheinit
: Runs the WebDyne initialization routines, which create needed directories, modify and create Apache .conf files etc.

wdcompile
: Usage: **wdcompile filename.psp**. Will compile a .psp file and use Data::Dumper to display the WebDyne internal representation of the page tree structure. Useful as a troubleshooting tool to see how HTML::TreeBuilder has parsed your source file, and to show up any misplaced tags etc.

wdrender
: Usage: **wdrender filename.psp**. Will attempt to render the source file to screen using WebDyne. Can only do basic tasks - any advanced use (such as calls to the Apache request object) will fail.

wddump
: Usage: **wddump filename**. Where filename is a compiled WebDyne source file (usually in /var/webdyne/cache). Will dump out the saved data structure of the compiled file.

wdfastcgi
: Used to run WebDyne under FastCGI - not usually invoked interactively

10.2. Other files referenced by WebDyne

`/etc/webdyne.pm`
: Used for storage of local constants that override WebDyne defaults. See the WebDyne::Constant section for details

`/etc/perl5lib.pm`
: Contains a list of directories that WebDyne will look in for libraries. Effectively extends Perl's `@INC` variable. If you install CPAN or other Perl modules to a particular directory using **perl Makefile.PL PREFIX=/opt/mylibs**, then add '`/opt/mylibs`' to the `perl5lib.pm` file, WebDyne will find them.

11. Extending WebDyne

WebDyne can be extended by the installation and use of supplementary Perl packages. There are several standard packages that come with the Webdyne distribution, or you can build your own using one of the standard packages as a template.

The following gives an overview of the standard packages included in the distribution

11.1. WebDyne::Chain

WebDyne::Chain is a module that will cascade a WebDyne request through one or more modules before delivery to the WebDyne engine. Most modules that extend WebDyne rely on WebDyne::Chain to get themselves inserted into the request lifecycle.

Whilst WebDyne::Chain does not modify content itself, it allows any of the modules below to intercept the request as if they had been loaded by the target page directly (i.e., loaded in the __PERL__ section of a page via the "use" or "require" functions).

Using WebDyne::Chain you can modify the behaviour of WebDyne pages based on their location. The WebDyne::Template module can be used in such scenario to wrap all pages in a location with a particular template. Another would be to make all pages in a particular location static without loading the WebDyne::Static module in each page:

```
<Location /static>

#  All pages in this location will be generated once only.
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Static'

</Location>
```

Multiple modules can be chained at once:

```
<Location />

#  We want templating and session cookies for all pages on our site.
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Session WebDyne::Template'
PerlSetVar      WebDyneTemplate '/path/to/template.psp'

</Location>
```

The above example would place all pages within the named template, and make session information to all pages via $self->session_id(). A good start to a rudimentary CMS.

`WebDyneChain`
: Directive. Supply a space separated string of WebDyne modules that the request should be passed through.

11.2. WebDyne::Static

Loading WebDyne::Static into a __PERL__ block flags to WebDyne that the entire page should be rendered once at compile time, then the static HTML resulting from that compile will be handed out on subsequent requests. Any active element or code in the page will only be run once. There are no API methods associated with this module

See the Static Sections reference for more information on how to use this module within an individual page.

WebDyne::Static can also be used in conjunction with the WebDyne::Chain module to flag all files in a directory or location as static. An example httpd.conf snippet:

```
<Location /static/>
```

```
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Static'

</Location>
```

11.3. WebDyne::Cache

Loading WebDyne::Cache into a __PERL__ block flags to WebDyne that the page wants the engine to call a designated routine every time it is run. The called routine can generate a new UID (Unique ID) for the page, or force it to be recompiled. There are no API methods associated with this module.

See the Caching section above for more information on how to use this module with an individual page.

WebDyne::Cache can also be used in conjunction with the WebDyne::Chain module to flag all files in a particular location are subject to a cache handling routine. An example httpd.conf snippet:

```
<Location /cache/>

#  Run all requests through the MyModule::cache function to see if a page should
#  be recompiled before sending it out
#
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Cache'
PerlSetVar      WebDyneCacheHandler '&MyModule::cache'

</Location>
```

Note that any package used as the WebDyneCacheHandler target should be already loaded via "PerlRequire" or similar mechanism.

As an example of why this could be useful consider the caching examples above. Instead of flagging that an individual file should only be re-compiled every x seconds, that policy could be applied to a whole directory with no alteration to the individual pages.

11.4. WebDyne::Session

WebDyne::Session generates a unique session ID for each browser connection and stores it in a cookie. It has the following API:

`session_id()`
: Function. Returns the unique session id assigned to the browser. Call via $self->session_id() from perl code.

`$WEBDYNE_SESSION_ID_COOKIE_NAME`
: Constant. Holds the name of the cookie that will be used to assign the session id in the users browser. Defaults to "session". Set as per WebDyne::Constants section. Resides in the `WebDyne::Session::Constant` package namespace.

Example:

Example 44.

```
<start_html>

Session ID: !{! shift()->session_id() !}

<end_html>

__PERL__

use WebDyne::Session
```

Run

WebDyne::Session can also be used in conjunction with the WebDyne::Chain module to make session information available to all pages within a location. An example httpd.conf snippet:

```
<Location />

# We want session cookies for our whole site
#
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Session'

#  Change cookie name from "session" to "gingernut" for something different
#
PerlSetVar      WEBDYNE_SESSION_ID_COOKIE_NAME    'gingernut'

</Location>
```

11.5. WebDyne::State::BerkeleyDB

WebDyne::State::BerkeleyDB works in conjunction with WebDyne::Session to maintain simple state information for a session. It inherits from WebDyne::State, and it could be used to build other state storage modules (e.g. WebDyne::State::MySQL)

`login()`
: Function. Logs a user in, creating a state entry for them. Returns true if successful, undef if fails.

`user()`
: Function. Returns scalar ref containing the name of the logged user for this session, undef if fails

`logout()`
: Function. Logout the current user, deleting all state info.

`state_store( key=>value, key=>value | hashref )`
: Function. Store a key and associated value into the state database. Returns true for success, undef for failure. You can optionally pass a hash ref to state_store, in which case it will replace the existing state hash.

`state_fetch( key )`
: Function.Fetch a previously stored key, Returns scalar ref to key value if successful, undef for failure. If no key name is supplied a hash ref of the current state hash will be returned.

`state_delete()`
: Function.Delete the state database for this session. Returns true for success, undef for failure. Actual deletion does not take place until cleanup() phase of Apache lifecycle.

`filename( filename )`
: Function.Fetch or set the name of the file where the state information will be held (defaults to `$WEBDYNE_CACHE_DIR/state.db`). Must be set before any state operations take place.

`$WEBDYNE_BERKELEYDB_STATE_FN`

Constant. Name of the file that will hold the state database. Can be just a file name or an absolute path name. Set as per WebDyne::Constants section. Defined in the `WebDyne::State::BerkeleyDB::Constant` namespace. Defaults to `state.db`

`$WEBDYNE_BERKELEYDB_STATE_DN`

Constant.Name of the directory where the state file will be located. If an absolute filename (i.e. one that includes a directory name) is given above then this variable is ignored. Set as per WebDyne::Constants section. Defined in the `WebDyne::State::BerkeleyDB::Constant` namespace. Defaults to `$WEBDYNE_CACHE_DN`

Example:

Example 45.

```
<start_html title="State">

<perl method="state">

<block name="login">
You are logged in as user "${user}"
<p>
The following data is associated with user ${user}:
<pre>
${data}
</pre>
</block>

<block name="logout">
You are not logged in.
</block>
<hr>

<form>
<submit name="login" value="Login">
<submit name="logout" value="Logout">
<submit name="refresh" value="Refresh">
</form>
</perl>

<end_html>

__PERL__

use WebDyne::Session;
use WebDyne::State::BerkeleyDB;
use Data::Dumper;

sub state {

        my $self=shift();

        if ($_{'login'}) {
                $self->login('alice') || return err();
                my %data=( fullname=>'Alice Smith', favourite_colour=>'Blue');
                $self->state_store({ data=>\%data }) || return err();
        }
        elsif ($_{'logout'}) {
                $self->logout
```

```
        }

        if (my $user=${ $self->user() || return err() }) {
                my $data_hr=$self->state_fetch();
                $self->render_block('login', user=>$user, data=>Dumper($data_hr));
        }
        else {
                $self->render_block('logout');
        }

        $self->render();

}
```

Run

State information is stored against the browser session ID, not against a user ID. The same user on two different machines will have two different state entries.

WebDyne::State is meant for simplistic storage of state information - it is not meant for long term storage of user preferences or other data, and should not be used as a persistent database.

WebDyne::State::BerkelyDB can also be used in conjunction with the WebDyne::Chain module to make state information available to all pages within a location. An example httpd.conf snippet:

```
<Location />

#  We want state information accessible across the whole site. WebDyne::State only works
#  in conjunction with WebDyne::Session, so it must be in the chain also.
#
PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Session WebDyne::State::BerkeleyDB'

</Location>
```

11.6. WebDyne::Template

One of the more powerful WebDyne extensions. WebDyne::Template can be used to build CMS (Content Management Systems). It will extract the <head> and <body> sections from an existing HTML or WebDyne page and insert them into the corresponding head and body blocks of a template file.

The merging is done once at compile time - there are no repeated search and replace operations each time the file is loaded, or server side includes, so the resulting pages are quite fast.

Both the template and content files should be complete - there is no need to write the content without a <head> section, or leave out <html> tags. As a result both the content and template files can be viewed as standalone documents.

The API:

`template ( filename )`
: Function. Set the file name of the template to be used. If no path is specified file name will be relative to the current request directory

WebDyneTemplate
: Directive. Can be used to supply the template file name in a Apache or lighttpd/FastCGI configuration file.

Example:

The template:

Example 46.

```
<html>

<head>
<block name="head" display="1">
<title>Template</title>
</block>
</head>

<body>

<table width="100%">

<tr>
<td colspan=2 bgcolor="green">
<span style="color:white;font-size:20px">Site Name</span>
</td>
</tr>

<tr>
<td bgcolor="green" width="100px">
<p>
Left
<p>
Menu
<p>
Here
</td>

<td bgcolor="white">

<!-- Content goes here -->
<block name="body" display="1">
This is where the content will go
</block>

</td>
</tr>

<tr>
<td colspan=2 bgcolor="green">
<span style="color:white">
<perl method="copyright">
Copyright (c) ${year} Foobar corp.
</perl>
</span>
</td>
</tr>


</table>

</body>
```

```
</html>

__PERL__

sub copyright {

        shift()->render(year=>((localtime)[5]+1900));

}
```

Run

The content, run to view resulting merge:

Example 47.

```
<html>
<head><title>Content 1</title></head>

<body>
This is my super content !
</body>

</html>

__PERL__

use WebDyne::Template qw(template1.psp);
```

Run

In real life it is not desirable to put the template name into every content file (as was done in the above example), nor would we want to have to "use WebDyne::Template" in every content file.

To overcome this WebDyne::Template can read the template file name using the Apache dir_config function, and assign a template on a per location basis using the WebDyneTemplate directive. Here is a sample httpd.conf file:

```
<Location />

PerlHandler     WebDyne::Chain
PerlSetVar      WebDyneChain    'WebDyne::Template'
PerlSetVar      WebDyneTemplate '/path/to/template.psp'

</Location>
```

A. GNU General Public License

A.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

A.2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to

any such program or work, and a "work based on the Program " means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification ".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

A.2.2. Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

A.2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

   Exception:

   If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

A.2.4. Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

A.2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the

Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

A.2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

A.2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

A.2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

A.2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

A.2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

A.2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

A.2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS