Web Development with Perl

Paul Fenwick
Jacinta Richardson
Kirrily Robert

Web Development with Perl
by Paul Fenwick, Jacinta Richardson, and Kirrily Robert

Table of Contents

List of Figures

Chapter 1. Introduction

Introduction

Welcome to Perl Training Australia's *Web Development with Perl* training course. This is a two-day course in which you will learn how to write dynamic, interactive web applications using the Perl programming language.

Course outline

- Day 1 -- Classical CGI programming
- Day 2 -- Introduction to HTML::Mason

Assumed knowledge

This course assumes that you already know HTML. You don't need to be an HTML genius, but you need to know what HTML tags look like, and how they work.

This course also assumes a comfortable understanding of Perl; variable types, operators and functions, conditional constructs, subroutines, regular expressions, objects and references.

Web standards

While this is not a course on web standards, it is highly recommended that you follow them whenever possible. The examples in this book will use or assume the use of Cascading Style Sheets (CSS) for presentation. CSS allows the HTML to represent the *logical* construction of a document, and can make things significantly simpler in teaching, development, and production.

What we don't cover

This course does not cover any client-side programming technologies, such as JavaScript. Some parts of the course may make reference to JavaScript, or use it demonstrate particular concepts, but these are for illustrative purposes rather than being part of the core course material.

Platform and version details

This module is taught using a Unix or Unix-like operating system. Most of what is covered will work equally well on other operating systems. Your instructor will inform you throughout the course of any areas which differ.

All of Perl Training Australia's training courses use Perl 5, the most recent major release of the Perl language. At the time of writing the most recent stable release of Perl is 5.8.8.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographic conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appear *`like this`*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Chapter 2. What is CGI?

In this chapter...

CGI is the *Common Gateway Interface*, a standard for programs to interface with HTTP (web) servers. CGI allows the HTTP server to run an executable program or script in response to a user request, and generate output on the fly. This allows web developers to create dynamic and interactive web pages.

CGI programs can be written in any language. Perl is a very common language for CGI programming as it is largely platform independent and the language's features make it very easy to write powerful applications. However, CGI programs are also written in C, Java, Python, PHP and shell.

It is important to remember that CGI is not a language in itself. CGI is merely a type of program which can be written in any language.

Clients and servers

A web server is a computer that manages and shares web based applications accessible anytime from any computer connected to the Internet. For example the training server we'll be using today is a web server for our purposes.

A client is a computer that's utilising a web server. For example, the web browsers on your machines, which you will be using to access content from our training server, are clients.

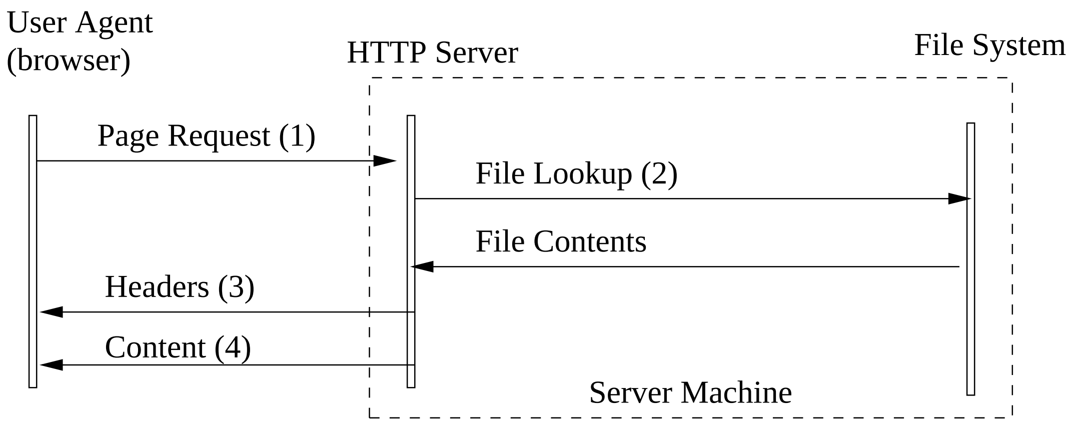
This course focuses in using Perl for server-side applications. We can use Perl to generate dynamic content, process forms, and other useful tasks.

HTTP basics

HTTP stands for HyperText Transfer Protocol, and is the protocol used for transferring hypertext documents such as HTML pages on the World Wide Web.

To understand how CGI works, you need some understanding of how HTTP works.

Figure 2-1. A typical HTTP connection



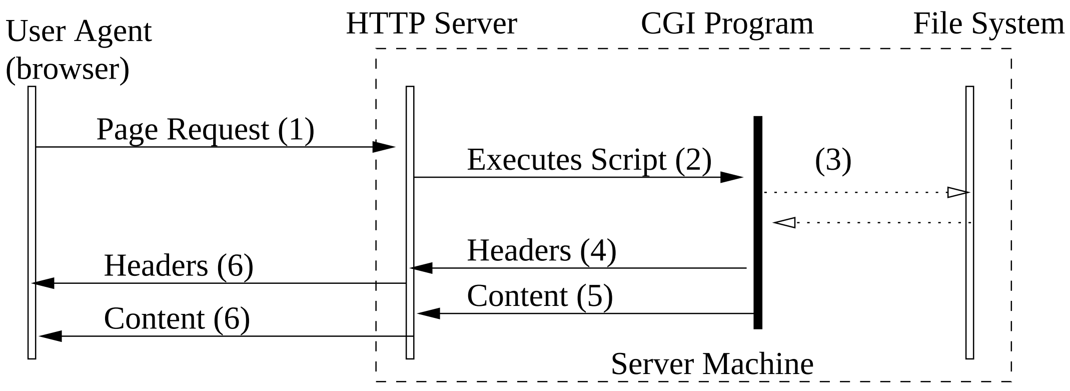
A simple HTTP transaction, such as a request for a static HTML page, works as follows:

1. The user agent (a web browser) connects to the port upon which the HTTP server is running (usually port 80). The user agent sends a request such as `GET /index.html`. The user agent may also send other headers.
2. The HTTP server receives the request and finds the requested file in its filesystem.
3. The HTTP server looks at the file content and generates HTTP headers to tell the client what kind of file it is sending, for example, text or image.
4. The HTTP server then sends the file contents to the client and closes the connection.

The CGI request

CGI requests different from HTTP requests as CGI scripts can return any kind of output; text, images, music; even from the same program! Thus for CGI programs, the server cannot guess the headers from looking at the file contents. Instead everything must be done by the CGI program.

Figure 2-2. A typical CGI connection



1. The user agent connects to the port upon which the HTTP server is running. It sends a page request and any other headers.
2. The HTTP server receives the request and executes the CGI program.
3. The CGI program runs, fetching from or writing to the file system as required. It may also contact other machines or services.
4. The program produces the appropriate HTTP headers and sends them to the HTTP server.
5. The program produces the content and sends that to the HTTP server.
6. The HTTP server forwards the headers and content to the client and closes the connection.

HTTP Methods

There are a number of ways of requesting data from a server. The most common of these are GET, POST and HEAD, which we'll briefly describe below. We'll cover GET and POST more during the rest of the course.

The indented text in each of the below sections is quoted from RFC 2616 Fielding, et al. (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html).

GET

> The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

Most web page requests are GET requests, such as:

```
http://www.example.com/index.html
```

GET requests can also submit data to CGI programs as follows:

```
http://www.example.com/script.cgi?name=Ben&age=29
```

HEAD

> The HEAD method is identical to GET except that the server *must not* return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request *should* be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself.

HEAD requests are rarely sent by users, but are often sent by proxies to check whether the content has changed before fetching the full page.

POST

> The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:
>
> - Annotation of an existing resource
> - Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
> - Providing data, such as the result of submitting a form, to a data-handling process
> - Extending a database through an append operation.

Most form submissions on webpages use the POST method.

HTTP Responses

From the user point of view, if a request has been successful, they receive their data, anything else means the request wasn't successful. On the protocol level there are a number of possible responses. We mention a few of the common ones below:

- 200 -- OK
- 301 -- Moved Permanently
- 302 -- (Found) Moved Temporarily
- 404 -- Not Found
- 500 -- Internal Server Error

Clients want to receive a response of 200, although 301 and 302 are okay (particularly if they include a redirect to the new location). You'll probably see a lot of 500 responses during the day, and we'll play a little with the other responses tomorrow.

Chapter summary

- CGI stands for Common Gateway Interface.
- HTTP stands for Hypertext Transfer Protocol. This is the protocol used for transferring documents and other files via the World Wide Web.
- HTTP clients (web browsers) send requests to HTTP (web) servers, which are answered with HTTP responses.
- All HTTP responses consist of headers and content.

Chapter 3. Classical CGI programming

In this chapter...

For a long time Perl was the de-facto language for developing web applications. The *Common Gateway Interface*, or more commonly just *CGI*, is a specification on how input can be accepted and decoded from browsers.

Perl has a `CGI` module that is part of the standard distribution. It provides a the ability to parse CGI arguments and data, as well as features for generating HTML. The module is sometimes referred to as `CGI.pm` to provide a distinction between the module and the specification.

This chapter covers how to write CGI programs in Perl.

CGI setup

The training server has been set up so that each user has their own web space underneath their home directory. All files which will be accessible via the web should be placed in the directory named `www`.

The directory `username/www/` on the training server maps to the URL `http://hostname/username/` on the web.

On many servers CGI programs need to be placed in a special directory (traditionally named `cgi-bin`), have a particular extension (commonly `.cgi`), have particular permission bits set, or a combination of any of these. These restrictions exist to ensure that non-CGI programs do not get executed by accident, and to ensure that CGI programs are executed, rather than just displaying their source-code.

Your CGI directory appears under your `www` directory The directory `username/www/cgi-bin/` on the training server maps to the URL `http://hostname/username/cgi-bin/` on the web.

If you were setting this up for yourself, you would need to ensure that:

- Your html and cgi-bin directories are world executable.
- All of your .html files are world readable.
- **Your CGI scripts are world readable and executable**.

Anatomy of a CGI program

CGI programs do not interact with the client directly, instead they receive information from the web-server and pass back appropriate responses.

CGI programs are expected to produce output to their standard output (STDOUT). This output includes headers (such as content type and cookies), and the actual document itself. The headers and content are divided by a single blank line.

The following is an example of a simple CGI program that displays the current time:

```
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\n\n";
print scalar localtime;
```

Hello World

The following HTML provides a simple "Hello World" message.

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

As this document is entirely HTML, this page will remain static. No matter how often we visit it, it will say the same thing. The only way of changing its contents is to change the file directly.

CGI programs, on the other hand, are able to generate data which depends on the time of day, a random number and what you’ve put in your *shopping cart*.

Exercises

1. Write a `text/plain` CGI program which prints out "Hello World!". Use the `localtime` example above to get you started.
2. Change your Hello World program to instead generate HTML, as shown above. Your header should now say `Content-type: text/html\n\n`.
3. Change your Hello World message to also print "`You are visitor number X`" where X is a random number.

   You can get a random integer between 0 and 10,000 with: `int(rand(10_000))`.

The CGI.pm module

Perl has a standard module called `CGI.pm` that simplifies web development. While many new technologies and techniques have been created since `CGI.pm` was developed, it can still be found in a very large amount of deployed code, and can be used when installing more modern tools is not an option.

There are still a huge number of traditional CGI programs that exist in production environments. `CGI.pm` programs will work on practically any system with a Perl installation, and this portability is partially responsible for their popularity.

An alternative

`CGI.pm` was originally written by Lincoln Stein in 1995 and has been actively maintained since. It is a monolith of code which does all of the following and more:

- Dealing with the CGI protocol, including parameter parsing
- Creating and managing cookies
- Generating HTML
- HTML and URI character escaping

With all of this functionality, much of which is not used by a typical program, `CGI.pm` could be very slow. Instead, it makes use of a number of very clever tricks to make it fast. Still, some alternative modules have been written to make it even faster.

One of the best known of these alternatives is `CGI::Simple`. It provides an object oriented interface like `CGI.pm`'s and is designed as a drop in replacement for `CGI.pm`. `CGI::Simple` only handles the CGI aspect of the `CGI.pm` module and does not include the HTML generation. However it is written to be more maintainable (the code is `strict` and `warnings` compliant) and faster.

Functional versus object-oriented

CGI provides both a functional and object-oriented interface. Throughout these notes we'll be using the *object-oriented* style, as it reduces the chance of conflicts between CGI subroutines, our own subroutines, and those built-in to Perl. It is also the style most commonly seen in examples in the `CGI.pm` documentation.

The following two examples demonstrate the difference between the styles. While the object-oriented interface may seem a little more typing at first, it can save a significant amount of time in debugging later on.

```
# Functional interface

use CGI qw(:standard);

print header(),
      start_html(),
      p("Hello World"),
      end_html();



# Object oriented interface (recommended)

use CGI;
my $cgi = CGI->new;

print $cgi->header(),
      $cgi->start_html(),
      $cgi->p("Hello World"),
      $cgi->end_html();
```

In these cases we're also showing off CGI's HTML generation abilities.

header()

The first piece of data we must send to the client is the header. This contains information regarding what kind of data is to follow (image, text, html, encrypted) and any browser directives such as cookie information, language, expiration date and caching suggestions.

To print a standard header we can just write:

```
print $cgi->header();
```

This will generate something like:

```
Content-Type: text/html; charset=ISO-8859-1
```

which is usually all that is required for CGI programs. The `header` function can take a number of arguments including non-standard ones which we might want to throw in:

```
print $query->header(
        -type=>'text/html',
        -expires=>'+3d',
        -cookie=>$cookie,
);
```

The expires parameter tells the client to cache the page result and not to re-invoke the program the next time the user requests that data. This is not a guarantee that the data will not be requested as both the client and the user are able to ignore this instruction.

start_html() and end_html()

Almost every invocation of your CGI program will result in a need to print out the start HTML tag, a header block, and your body tags. Fortunately, `CGI.pm`'s `start_html` method creates the top of your page, and can be used to reliably create much of the optional information which controls your page's appearance and behaviour.

A simple header should include the page's title, for example:

```
print $cgi->start_html(
        -title   => "Hello World",
)
```

which produces:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head><title>Hello World</title>
</head><body>
```

We can pass in other parameters to define the page author, document base, frame targets, meta information, style-sheets, background colour and links. Read **perldoc CGI** for more information.

On the other hand, the `end_html` method finishes the page. This involves closing the body tag, and the html tag:

```
print $cgi->end_html();

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

If you don't want to generate XHTML, you can use `CGI`'s `-no_xhtml` pragma, like this:

```
use CGI qw(-no_xhtml);
```

You can also pass a `-dtd` and many other parameters to `start_html`. For more information, read `perldoc CGI.pm` under the section *CREATING THE HTML DOCUMENT HEADER*.

Exercise

1. Alter your previous hello world program to use CGI's `header`, `start_html` and `end_html` methods.

Debugging CGI programs

When writing CGI programs, there are many problems which may affect their execution. Since these will not always be easily understood by examining the web browser output, there are other ways to determine what's going wrong.

If there seems to be a problem first try the following steps:

1. Check that your program compiles by using `perl -c`
2. Check the permissions on your cgi program. If it is not world executable then it won't work.
3. Run your program on the command line. If your program waits for input, that's your opportunity to pass in parameters. For the moment, press **CTRL-d**.
4. If your program runs fine on the command line but still does not output to the browser, make sure that you have not forgotten to print the header before any other output.
5. Check the HTML source that you're printing. Make sure that you've closed any tables you've opened and not made any obvious HTML errors.
6. Check the web server's log files. The location of these vary from system to system. On our system they're in `/var/log/apache/`.

Failing gracefully with CGI::Carp

You can read all about `CGI::Carp` by reading **perldoc CGI::Carp**. It's also covered briefly on page 878 of the Camel book (but not in 2nd Ed.).

CGI scripts often leave warning messages in the error logs without time stamps or script name. This can make it much harder to identify which program caused the error, and how long ago it occurred. Fortunately we can use Perl's `CGI::Carp` module to add both of these pieces of information:

```
use CGI::Carp;
```

We can also make our errors go to a separate log, by using the `carpout` subroutine. This needs to be done inside a `BEGIN` block in order to catch compiler errors as well as ones which occur at the interpretation stage.

```
BEGIN {
        use CGI::Carp qw(carpout);
        open(my $log, ">>", "cgi-logs/mycgi-log")
                or die("Unable to open mycgi-log: $!\n");
        carpout($log);
}
```

You will need to ensure that the user id that your program is running under has the permissions to access the directory and file that you provide.

Fatal errors

One of the most common uses of `CGI::Carp` is to cause any fatal errors to have their error messages and diagnostic information output directly to the browser:

```
use CGI::Carp qw(fatalsToBrowser);

die 'Some disaster!';   # This will be printed to the browser
```

`CGI::Carp` is an excellent tool to use during debugging. However, it is not a good idea to leave it enabled with `fatalsToBrowser` in production code. There are two reasons for this. The first is that errors should be handled properly by a default error page or something equivalent. The second, and more important, is that `fatalsToBrowser` provides a lot of information about your script's internals. This information is not going to help your average user to know what they did wrong, but it may help malicious users discover ways they can exploit your code.

Warnings

Just as `fatalsToBrowser` allows us to see Perl's fatal errors, we can also use `CGI::Carp` to show us Perl's warnings. These are printed in our HTML source as HTML comments so that they don't interfere with our normal output.

```
use CGI::Carp qw(warningsToBrowser);
warningsToBrowser(1);

warn 'Some warning';    # This will be printed in HTML comments
```

As these are only useful after we have printed out our CGI headers, `CGI::Carp` will buffer our warnings until after we have called `warningsToBrowser(1)`, which tells `CGI::Carp` that it is safe to now print warnings in HTML comments. We can also turn buffering back on if we are generating HTML structures which don't allow comments internally: `warningsToBrowser(0)`

We can combine both `fatalsToBrowser` and `warningsToBrowser` to turn them both on:

```
use CGI::Carp qw( fatalsToBrowser warningsToBrowser );
```

Exercises

1. Edit your `hello.cgi` program so that it dies with an error before the CGI headers are printed. What does the browser display?
2. Add `fatalsToBrowser` to your program and make sure that you now receive that error now printed to the browser.
3. Use `warn` to print a warning during your program's execution. Turn on `warningsToBrowser` and ensure that it appears in your program's comments.

Environment values

The CGI specification makes available a large amount of additional information including what type of browser is accessing our server, and its IP address, as well as our server's name and which virtual host is being accessed. This information is placed into environment variables which we can access through `CGI.pm`'s methods.

user_agent()

> Returns `HTTP_USER_AGENT`. If provided with an argument, it will use that for pattern matching allowing you to write `user_agent("netscape")` to determine whether the user agent string includes the word "`netscape`".

remote_host()

> The remote host name or IP address.

server_name()

> The name of the server the program is running on. Very useful when running a testing site and a production site, but keeping identical copies of code.

virtual_host()

> Name of host to which the browser attempted to connect, if virtual hosts are in use.

Exercises

1. Edit your Hello World program to print out the user agent and ip address of the visitor.

   Try calling your program with different browsers.

Chapter summary

- CGI programs produce output to their STDOUT, this must include both the headers and the content.
- The `CGI.pm` module can be used to produce both the headers and the content.

- The `CGI.pm` module can be used in both a functional and object oriented fashion.

Chapter 4. HTML Forms

Introduction

Forms are a common and often integral part of any web application. In simple terms, a form is an interface whereby users can enter or modify data, and then submit that data to a server for processing.

HTML generation with CGI

In general, it is better to separate HTML and code into separate files. This allows HTML designers to alter the page structure without having all this ugly code get in the way! We'll talk about how to use templating systems to achieve this in a later chapter.

The examples in the previous chapter work equally well for `CGI.pm` and `CGI::Simple` and the two modules are interchangeable. However, `CGI.pm` also supports basic HTML generation.

`CGI.pm` has methods for all the standard HTML tags. So, for example if we wanted to print out an anchor:

```
<a href="http://www.perltraining.com.au/">
Further information about Perl Training Australia.</a>
```

we'd type:

```
use CGI;
my $cgi = CGI->new();

print $cgi->a(
        {-href => "http://www.perltraining.com.au/"},
        "Further information about Perl Training Australia."
);
```

We can also create start tags, and end tags directly, in order to generate data to go between them:

```
use CGI;
my $cgi = CGI->new();

print $cgi->start_ul;
foreach my $name (qw/Jacob Jeremy Jacinta Jenni Jack/)
{
        print $cgi->li($name);
}
print $cgi->end_ul;
```

This HTML generation is most useful when we're creating form elements, particularly when a lot of our data is coming from a file or database. This is because it allows us to use our existing data structures to easily generate the HTML we need.

`CGI.pm` methods typically take two arguments; a hash reference of options and the data. In the cases where you are not passing any data in (such as `start_html`), you can pass in key/value pairs instead. So the following are equivalent:

```
print $cgi->start_html( {
        -title => "Hello World"
})

print $cgi->start_html(
        -title => "Hello World"
)
```

In the cases where you are happy to use the default options, you can leave off the first argument and just provide your data:

```
print $cgi->li($name);
```

It is a mistake to provide both options and data, without including the options in a hash reference:

```
# Won't work as desired:
print $cgi->li( -class => $class, $name );

# Should be:
print $cgi->li( { -class => $class }, $name );
```

Some methods also allow you to use short cuts, so the below example is also equivalent to the previous `start_html` examples:

```
print $cgi->start_html("Hello World");
```

The form element

The `form` element is a block level element, which means that the browser will present it on a new line, as it does with headings and paragraphs. It is also a container, which means that other elements can appear inside the form.

It's attributes include:

method

How the form should be submitted to the web server. These include GET and POST, which we'll cover in a moment.

action

A relative or absolute URL for the CGI program which the form information should be submitted to.

enctype

The form's encoding type. This should be `application/x-www-form-urlencode` for standard forms and `multipart/form-data` for forms with file-uploads.

Creating a form using CGI.pm is easy.

```
print $cgi->start_form({
        -action  => "myscript.cgi",
        -method  => "POST",
});

# print form internals.

print $cgi->end_form();
```

If we don’t specify our method, then CGI.pm will assume we intended POST. Likewise if we don’t specify an action, it will assume we intend for the submission to come back to the current CGI script. CGI provides an encoding type of (`application/x-www-form-urlencode`) by default, so we can leave that off too. Assuming that our program is called `myscript.cgi`, we can reduce the above to:

```
print $cgi->start_form();

# print form internals.

print $cgi->end_form();
```

Multi-part forms (which allow us to upload files) use a different method.

```
print $cgi->start_multipart_form({
        -action  => "myscript.cgi",
        -method  => "POST",
        -enctype => "multipart/form-data",
});

# print form internals.

print $cgi->end_form();
```

GET vs POST

There are two commonly used methods for form submission. These methods are called GET and POST, representing the underlying action taken in the HTTP protocol. It is worth briefly examining the difference between these two methods for form submissions, and the advantages and disadvantages of each.

GET

A ’GET’ form submission operates by taking the form data and encoding it into the URL. As an example, let’s pretend that our form asks for the user’s favourite colour and food:

```
<form method="get" action="http://example.com/favourites.html">
<p>
        What is your favourite colour?
        <input type="text" name="colour" /><br/>
        What is your favourite food?
        <input type="text" name="food" /><br/>
        <input type="submit" />
</p>
</form>
```

If the user enters the colour ’red’ and the food ’apples’, this will generate a request to `http://example.com/favourites.html?colour=red&food=apples`.

The primary advantage of GET and URL encoding is that it results in URLs that can be easily bookmarked. It's also easy for humans to modify the URL, which can make ad-hoc testing simpler. GET encoding should never be used for forms involving passwords, as the resulting URL including the password may appear in the browser history, cache logs, and server logs.

GET requests are poorly suited for large forms, and cannot be used for file uploads.

POST

A 'POST' form submission operates by composing the form data into the body of the request submitted to the server. The data can be of any length, and file uploads and other binary objects can be handled cleanly. POST operations do not alter the URL in any way.

POST operations do not result in pages that are easy to bookmark, but at the same time they do not have problems with passwords ending up in logs and history files.

Form elements

There are a large number of possible form elements we can use. We'll cover some of these here briefly.

Submit

The submit element creates a button which, when pressed, submits the form to the server.

```
<input type="submit" name="personal_data" value="Finished!" />
```

We can create these with CGI.pm with:

```
print $cgi->submit({
        -name  => "personal_data",
        -value => "Finished!",
});
```

Exercises

With this and the later exercises you will develop a form which takes a number of different kinds of inputs.

1. Write a program which creates a form which will POST the content back to itself.
2. CGI has a method called `Dump` which prints out any values passed into a script through a GET or POST submission. We can call this as follows:

   ```
   print $cgi->Dump();
   ```

   Use `Dump` to print the values passed into your form.
3. Test that your program works. We'll be adding form elements to make use of `Dump` as we work through this chapter.

4. Add a submit button to your form. Traditionally these appear at the end of the form.

   Run your program and see how it goes.

Text

The text input field.

```
<input type="text" name="email_address" value="bob@example.com" />
```

We can create this with CGI.pm by either of the following:

```
print $cgi->textfield({
        -name  => 'email_address',
        -value => 'bob@example.com',
});

print $cgi->input({
        -type  => 'text',
        -name  => 'email_address',
        -value => 'bob@example.com',
});
```

Exercises

Add a text field to your form. Type something into the box and submit your form.

Hidden

Hidden fields allow us to pass data around without having to display it to the user. Using hidden doesn't mean that the user *cannot* see the data -- as it's there in the source -- but it means that the user doesn't have to worry about it.

Hidden fields should always have a value defined.

```
<input type="hidden" name="stage" value="3" />
```

We can create this with CGI.pm by either of the following:

```
print $cgi->hidden({
        -name  => "stage",
        -value => 3,
});

print $cgi->input({
        -type  => "hidden",
        -name  => "stage",
        -value => 3,
});
```

Exercise

Add a hidden field to your form. Give it a value and submit your form. What happens?

Password

Password fields allow the user to enter a password without fear of on-lookers learning it. Values entered into a password field are obscured with asterisk characters (`*`).

```
<input type="password" name="user_password" />
```

We can create this with CGI.pm by either of the following:

```
print $cgi->password_field({
        -name  => "user_password"
});

print $cgi->input({
        -type => "password",
        -name => "user_password",
});
```

Just because the information is obscured on the screen doesn’t mean that it is secure. Under a regular HTTP request, all fields (including password fields) are passed to the server in plain text. This allows anyone with a packet-sniffer to read what data was entered.

Likewise, setting a password value when generating HTML will be visible to anyone who looks at the HTML source. If keeping passwords secure is important, make sure you are using secure-HTTP (HTTPS).

Exercise

Add a password field to your form. Enter something into the box and submit the form.

Checkbox

Checkboxes allow users to set a value to on or off. Used singly these might be used to opt-in to a mailing list, or request that an email copy of an invoice be sent. If the checkbox should appear as "on", we set the "checked" parameter:

```
<input type="checkbox" name="send_email" value="yes" checked="checked" />
Email copy of itinerary?
```

We can create this with CGI.pm with:

```
print $cgi->checkbox({
        -name    => "send_email",
        -value   => "yes",
        -checked => 1,
        -label   => ’Email copy of itinerary?’
});
```

If the user selects the checkbox, it will appear in the form parameters with the specified name and value. If the checkbox is not selected, then there will be no parameter of that name, rather than the name and a false value.

Checkbox groups

Another common use of checkboxes is to create a group of them. These receive the same input name, but have different values. We can tell which (if any) values the user selected, by looking at the list of values given to us under that input name.

```
<input type="checkbox" name="interests" value="bowl"  />Bowling <br />
<input type="checkbox" name="interests" value="fish"  />Fishing <br />
<input type="checkbox" name="interests" value="climb" />Climbing<br />
<input type="checkbox" name="interests" value="ski"   />Skiing  <br />
<input type="checkbox" name="interests" value="dive"  />Diving  <br />
```

Rather than create these individually we can use CGI.pm to do it all at once:

```
print $cgi->checkbox_group({
        -name      => "interests",
        -values    => [ qw( bowl fish climb ski dive ) ],
        -labels    => {
                bowl  => "Bowling",
                fish  => "Fishing",
                climb => "Climbing",
                ski   => "Skiing",
                dive  => "Diving",
        },
        -default   => [],
        -linebreak => 1,
});
```

Let’s look at that a little. Specifying a hash of labels let’s us tell CGI.pm what text to put beside each checkbox, specifying an array of values, tells CGI.pm the order in which the checkboxes should be placed. We can leave the labels hash empty if the values and labels are the same.

Note that our values, defaults and labels are all passed as anonymous references. If we have access to arrays and hashes with this information in it (perhaps because we’ve pulled it from a file or database), we can pass in references to those instead:

```
my %labels = (
        bowl  => "Bowling",
        fish  => "Fishing",
        climb => "Climbing",
        ski   => "Skiing",
        dive  => "Diving",
);
my @values   = qw( bowl fish climb ski dive );
my @defaults = qw();

print $cgi->checkbox_group({
        -name      => "interests",
        -values    => \@values,
        -labels    => \%labels,
        -default   => \@defaults,
        -linebreak => 1,
});
```

The `linebreak` option, if true, puts a `<br />` tag after each checkbox.

Exercises

1. Add a checkbox field to your form. Try submitting your form with it checked and unchecked, what is the result?
2. Add a group of checkboxes to your form. Try submitting your form with none, some and all of the boxes checked. What is the result?

Radio button groups

Radio buttons allow users to set one value in a group to on or off. Although it is possible in HTML to have a single radio button, the correct equivalent is a stand-alone checkbox.

```
<input type="radio" name="age_group" value="10" />Under 10<br />
<input type="radio" name="age_group" value="20" />11 to 20<br />
<input type="radio" name="age_group" value="30" />21 to 30<br />
<input type="radio" name="age_group" value="40" />31 to 40<br />
<input type="radio" name="age_group" value="50" />40 and over<br />
```

Logically, a group of radio buttons is equivalent to a single value select list (such as that generated in a pop-up list). It is often a good idea to use radio buttons for very small sets of options (such as binary decisions) and when seeing all of the options is important. With a larger set of options, using a pop-up list will improve the usability of your website.

Creating a radio group is almost identical to creating a checkbox group:

```
print $cgi->radio_group(
        -name       => "age_group",
        -values     => [ 10, 20, 30, 40, 50 ],
        -default    => 0,
        -linebreak  => 1,
        -labels     => {
                10 => "Under 10",
                20 => "11 to 20",
                30 => "21 to 30",
                40 => "31 to 40",
                50 => "40 and over",
        },
);
```

Setting the default value to a non-existent value ensures that no value will be originally selected.

Select

There are two types of select lists in HTML. One is often referred to as a "pop-up list" as their implementation typically has the list options pop-up over the browser window when you wish to scroll through them. These allow you to select one value, and that is the value which remains visible.

The second is often referred to as a "scrolling list" or a "multi-list". These may show a number of entries at once, and you can can select multiple values if you desire.

The type of select list is defined by two factors. The first is a `size` parameter (how many values to show at once) and the `multiple` attribute.

```
<!-- Simple pop-up list -->
<select name="computers">
<option                     value="1">1</option>
<option selected="selected" value="2">2</option>
<option                     value="3">3</option>
</select>

<!-- Scrolling, select multiple list -->
<select name="operating_systems" size="3" multiple="multiple">
<option value="win32">Microsoft Windows (XP, 2000, 98)  </option>
<option value="linux">Linux (RedHat, Debian, Ubuntu etc)</option>
<option value="mac">  Apple (OS9, OSX)                  </option>
</select>
```

We can create a pop-up menu with CGI.pm with:

```
print $cgi->popup_menu({
        -name    => "computers",
        -values  => [1, 2, 3],
        -default => 2,
        -labels  => {},
});
```

We can create a scrolling list with:

```
print $cgi->scrolling_list({
        -name     => "operating_systems",
        -values   => [ qw(win32 linux unix mac) ],
        -default  => [],
        -labels   => \%labels,
        -multiple => 1,
        -size     => 1,                # Viewport length
});
```

Exercises

1. Create a popup menu for your form.
2. Create a scrolling menu for your form.
3. Experiment with passing both anonymous references, and references to existing variables.

File upload

The file upload field allows us to upload files from the user. These may be files of any type (text, image, mp3...). In order for these to be useful, we must use CGI's `start_multipart_form` or otherwise specify that we're using a different encoding type.

The file upload box automatically includes a `Browse` button on most web browsers. We can generate a file upload box with:

```
print $cgi->start_multipart_form();

print $cgi->filefield({
        -name      => "file_upload",
        -default   => "Please enter a file name",
        -size      => 50,
});
```

The size specifies the boxes width, and the maxlength the maximum number of characters allowed in the filename and path. The default value may appear in the file field box, but is ignored by most browsers. It is safe to leave off all values excepting the field name.

POST_MAX and DISABLE_UPLOADS

CGI has two package variables which control the maximum size of POSTings and whether or not uploads can be used. To set a maximum size for your posts set `$CGI::POST_MAX`, this should be set to a reasonable value such as 1 megabyte.

To disable file uploads completely, set `$CGI::DISABLE_UPLOADS` to a true value.

```
use CGI;

$CGI::POST_MAX        = 1024 * 1024;   # 1 MB posts
$CGI::DISABLE_UPLOADS = 1;             # No uploads

my $cgi = CGI->new();
```

Exercise

Add a file upload box to your program.

Pretty HTML

CGI.pm generates very dense HTML by default. That is, it doesn’t add any extra newlines or spaces between grouped elements (for example for checkbox and radio groups). This can make reading the source code very difficult, and it also may mess up the neat formatting of your templates. For example, the following is how a checkbox group is formatted:

```
<input type="checkbox" name="check" value="a" />a<br /> <input
type="checkbox" name="check" value="b" />b<br /> <input type="checkbox"
name="check" value="c" />c<br /> <input type="checkbox" name="check"
value="d" />d<br />
```

We can make this a lot nicer, by adding our own newlines. Calling a CGI.pm method in list context, returns the generated elements in a list. We can then join this list with newlines and any other spacing desired:

```
my @boxes = (’a’ .. ’d’);

my $check = join("\n", $cgi->checkbox_group(
        -name      => "check",
        -values    => \@boxes,
        -linebreak => 1,
));
```

This will produce:

```
<input type="checkbox" name="check" value="a" />a<br />
<input type="checkbox" name="check" value="b" />b<br />
<input type="checkbox" name="check" value="c" />c<br />
<input type="checkbox" name="check" value="d" />d<br />
```

The above does not work for select lists, which are always returned as a single scalar. Fortunately they already include newlines.

Chapter summary

- `CGI.pm` can be used to generate all sorts of HTML tags. This is most useful for form elements.
- GET form submissions encode their data in the request string. POST submissions encode their data in the request body.

Chapter 5. Accepting and processing form input

In this chapter...

CGI programs are often used to accept and process data from HTML forms. In this section, we show how we can use the `CGI.pm` module to parse form data.

CGI Parameters

One of the biggest advantages of using the `CGI.pm` module is the easy access it provides to all of the CGI session information. The most useful of these are the values passed in from the user, but other information such as the URL, hostname, path details and referrer can also prove helpful.

The parameters from an HTML form are usually encoded a "url-encoded" format:

```
name=Paul&company=Perl%20Training%20Australia
```

In this format input is encoded as `key=value` pairs, with each `key=value` combination separated with an ampersand. Spaces, most punctuation, and non-printable characters are replaced by a percentage followed by their ASCII value in hexadecimal.

For a GET operation, this encoded string appears as part of the URL. The web-server extracts this portion of the URL and places it into the `QUERY_STRING` environment variable.

```
http://example.com/cgi-bin/test.cgi?name=Paul&company=Perl%20Training%20Australia
```

For a POST operation these are provided as part of the message body, and are fed to the program via `STDIN`.

As you can imagine, decoding this by hand is hard work. It’s even harder than you may think, since there are alternative encoding schemes that may be used, and alternative character sets to consider.

Fortunately, we should never need to decode a form submission ourselves, as we can use `CGI.pm`’s `param()` method to fetch them:

```
#!/usr/bin/perl -w
# Prints a "Hello" to the name given or to "Stranger"
use strict;
use CGI;

my $cgi = CGI->new();

my $name = $cgi->param(’name’) || "Stranger";

print $cgi->header(),
      $cgi->start_html(’Hello!’),
      $cgi->p("Hello, $name!"),
      $cgi->end_html();
```

Using CGI’s `param()` method (or an equally well-respected module) is *always* a better idea than parsing the parameter string ourselves.

All parameter names

We can get a list of all the parameters passed in by calling `param` without any arguments:

```
my @all_parameters = $cgi->param();
```

Calling `param()` in context

Certain types of form input fields define multiple values of the same name. For example a check box group may have more than one check box checked. A scrolling list might have more than one element selected. To access these we ask `param` for an array.

```
# put all the check box values that were checked into @checked.
my @checked = $cgi->param('group_name');
```

Of course if we only expect one value we can say:

```
my $checked = $cgi->param('send_email');
```

When we call `param` in a scalar context, we will always get a scalar result. If that parameter was actually given a number of values, we'll just get the first one of them; nothing will tell us that there was more than one.

Context issues

When called in a list content, `param` will return a list of values. If there was no value set for that parameter, it will return an empty list. This can be a problem if you do the following:

```
check_input($cgi->param("name"), $cgi->param("phone"));

# later

sub check_input {
        my ($name, $phone) = @_;

        ...
}
```

This will work *most of the time*, however if there were multiple values for `name` then `$phone` will be set to the wrong value. Perhaps worse, if there is no value for the `name` parameter, then `$name` will be set to the value for the `phone` parameter!

The correct solution is to explicitly specify the context we want from `param()`:

```
my $name  = $cgi->param("name");
my $phone = $cgi->param("phone");
check_input($name, $phone);

# or
check_input(scalar($cgi->param("name")), scalar($cgi->param("phone")));
```

Exercise

1. Use `param` to print out the values of your text and hidden fields in your form program.

2. Using `param` in list context print out the values of your select lists.

Setting our own parameters

`param` can also be used to set or override parameter values for the invocation of your program. This can be useful to provide missing information (perhaps from a database) before using the `CGI` object to populate values in a template.

```
# Add an age to the CGI object:
$cgi->param( -name  => "age",
             -value => 15,
);

# Add colours to the CGI object:
$cgi->param( -name   => "colours",
             -values => ['orange', 'black', 'purple' ],
);
```

We can also append values to a parameter for the life of the program:

```
# Add these numbers to any already selected
$cgi->append( -name   => "number",
        -values => [ 2 .. 5, 7 .. 10],
);
```

Deleting parameters

Sometimes we want to use a parameter value and then delete it from the parameter list. For example if we are printing out submitted data to a file for later reference, we may wish to avoid including any passwords. Rather than putting in checks for each excluded field, we can instead just delete the values.

```
# Delete the passwords now that we no longer need it
$cgi->delete( "password", "repeat_password" );
```

To delete all of the parameters (perhaps as part of a form reset) we can use `delete_all`.

```
# Delete all the parameters
$cgi->delete_all();
```

Printing out parameters

When debugging, it often helps to see all the values passed in from the previous script. Unfortunately `Data::Dumper` does not provide a friendly HTML format, and sometimes access to the error log may not be available. Fortunately we can use `Dump` to print these values for us in HTML.

```
# Print all the user supplied values
$cgi->Dump;
```

This creates HTML similar to the following:

```
<ul>
        <li>name1</li>
        <ul>
                <li>value1</li>
                <li>value2</li>
        </ul>
        <li>name2</li>
        <ul>
                <li>value1</li>
        </ul>
</ul>
```

The same behaviour can be achieved by interpolating the `$cgi` object in a string:

```
print "These are my values: $cgi";
```

Exercises

1. Change your earlier form program to also print "BINGO" if four or more fields have been given values and submitted to your program. Try to distinguish between empty but present fields (such as the text field with no data) and fields with actual data.
2. Print a funny message if any field has been submitted with multiple values (for example your scrolling list or checkbox group).

Debugging with the `CGI.pm` module's offline mode

CGI.pm allows us to run our CGI scripts in debug mode. This allows us to specify parameters on the command line, rather than via a browser. To do this, we specify debug mode in our use line:

```
use CGI qw(-debug);

my $cgi = CGI->new();
```

Once debug mode is turned on, we will be prompted for input each time we run the program on the command line:

```
% ./hello_name.pl
(offline mode: enter name=value pairs on standard input; press ^D or ^Z
when done)
```

This allows you to enter parameters in the form `name=value` for testing and debugging purposes. **CTRL-D** on Unix or **CTRL-Z** on Windows (the end-of-file character) to indicate that you are finished:

```
(offline mode: enter name=value pairs on standard input; press ^D or ^Z
when done)
name=fred
age=40
^D
```

CGI.pm assumes that the `value` pairs that you pass it are url-encoded. We're just about to cover how you can url-encode a variable.

CGI.pm and input fields

A nice but sometimes surprising behaviour of the CGI.pm class is to assign parameters from `param` to your input fields. This means that if your script submits to itself and some of the validation fails you can reprint the passed in data with no further effort. On the other hand you may not get the value you expected to come out in your field.

To solve this problem, if you want the value you supply to *always* be the initial value in that input then use the override option:

```
$cgi->hidden({
        -name     => "student_id",
        -value    => 36887,
        -override => 1,
});
```

Exercises

These exercises build on the form you created during the previous chapter. Add the first two answers into your program after you `Dump` out the submission results. Call `Dump` again, after these actions, to verify the changes.

1. Using `param` change one of submitted values.
2. Append a value to the submitted checkbox values.
3. Add a default value to your pop up list. Submit the form with a different value and look at the form elements. Does the pop up list get filled in with your submitted result or the default value? Use `override` to force the default to show.

Building a GET string

Very occasionally we don't actually want to have the user input data through a form, rather we'd just like to give them a pre-made link to follow that passes our script any parameters we need. In this case we have to build the GET string ourselves. One thing that we need to make sure of is that the parameters we pass are in a form that our browser will support. So, we have to replace spaces with `%20` or `+` and escape other punctuation with the hexadecimal representation of their ASCII values.

Fortunately the `CGI.pm` module is very helpful here, with a function called `escape`. This function has an opposite `unescape` such that:

```
unescape(escape($string)) eq $string
```

is true, but you shouldn't need to use `unescape` all that often.

To build a GET string just do something like the following:

```
my $get_string = "section=" . $cgi->escape( "Underwater photography" );
my $url        = a({-href=>"my_script.cgi?$get_string"}, "Current section");
```

In most cases GET strings are formed for us by the browser.

File upload

`CGI.pm` can also be used to allow users to upload files. To do this we need to specify the correct encoding type in the form element, if we use CGI.pm's `start_multipart_form()` function, then it'll do the right thing. Alternately we can specify it ourselves manually.

You'll find the below code it in `www/upload.html`.

```
<html>
<head>
<title>Upload a file</title>
</head>
<body>
<h1>Upload a file</h1>
<p>
Please choose a file to upload:
</p>
<form action="cgi-bin/upload.cgi" method="POST"
         enctype="multipart/form-data">
<input type="file" name="filename">
<input type="submit" value="OK">
</form>
</body>
</html>
```

To handle file uploads we use `upload()` instead of `param()`. The value returned is special -- in a scalar context, it gives you the filename of the first file uploaded with that input name. In a list context it gives you all of the filenames uploaded with that input name. These filenames can also use be used as filehandles.

```
my $filename = $cgi->upload('filename');

while( <$filename> ) {

        # do something with file contents

}
```

To save the contents of the uploaded file, we can use `File::Copy`. We also use `File::Temp` to ensure that we have unique filenames. `Fatal` saves us from having to check the supplied functions for failure, by replacing them with a version which throws an exception instead.

```
use File::Copy qw(copy);
use File::Temp qw(tempfile);
use Fatal      qw(copy chmod);

my $file_in = $cgi->upload('file');

# If we have uploaded a file
if( $file_in ) {
        my ($fh_save, $new_filename) =
                tempfile("student_XXXXX", DIR => "/tmp/");

        # Prevent newline translations by Perl
        binmode($file_in);
        binmode($fh_save);

        copy($file_in, $fh_save);
        # Change the permissions so that you will be able to read it.
        # In most cases this isn’t necessary as usually it will only
        # need to be read and edited by the web server
        chmod 0644, $new_filename;

        # Tell the user what the file has been saved to:
        print "File copied to $new_filename";
}
```

The above code can be found in `www/cgi-bin/upload.cgi`.

Be mindful of the user id that runs your CGI programs on the server. In our case, all CGI programs are run by `www-data`. This means that your CGI programs can see, read and over-write file uploads from the other members of your class. However, without changing the permissions, you will not be able to read those files yourself. In other set-ups your programs may run with the same permissions as yourself.

Differences of permissions and environments between your user id and that of your running CGI programs can cause subtle errors. For example you may find that your program runs perfectly from the command line, but not from a browser. This is often caused by having configuration files, libraries or data directories with insufficient permissions to allow web server use.

To assist with text file processing, Perl attempts to translate newline characters from the filesystem format to its own internal format. In the case of binary files this can be a problem. To ensure that newline character codes remain untouched, we can use the `binmode` method, as shown above.

Exercises

1. Edit the `www/cgi-bin/upload.cgi` file to change the `tempfile` template (`student_XXXXX`) to include your student number. For example `student1_XXXXX`.
2. Upload a file and ensure that it appears in the `/tmp` directory.
3. Edit your form script to handle file uploads. Upload a text file and print its content out to the browser.
4. Now print out only every second line. (Hint: You can use `$line % 2 == 0` to determine if `$line` is an even number.)
5. (Advanced) Edit the `www/cgi-bin/upload.cgi` file to include a popup menu, listing out the filenames for files you have previously uploaded. Allow the user to select one of these files (or upload another) for display.

   You may want to look at `glob` to help select the files matching your template.

Chapter summary

- The `CGI.pm` module can be used to access parameters passed to the CGI program using the `param()` function.
- Using the `param()` function in a list context will return all of the values passed to the program with that key.
- Care should be taken if `param()` is ever going to be passed to subroutines.
- Calling `param()` in a list context without a key will return all of the names of the name=value pairs.
- CGI.pm will fill in all of your form input fields with values from `param()` if possible. To prevent this you have to use the `override=>1` option in your input field.
- File uploads must use multipart forms.
- To access the file from an upload call the `upload()` function rather than the `param()` function. The return value can be used as both the filename and a filehandle.

Chapter 6. Security issues

In this chapter...

In this section we briefly examine some security issues arising from the use of CGI scripts including the risks of handling tainted data and how to avoid problems.

This is not a complete guide to CGI security, but rather a simple discussion of a few important points. Following all the recommendations in this chapter will not guarantee that your script is free of security flaws, but it will certainly help.

For a more complete guide to Perl security, Perl Training Australia's *Perl Security* course-notes can be found on-line at http://perltraining.com.au/courses/perlsec.html .

The need for security

Always trust your users. Never trust their input.

It is easy to believe that, as a web programmer, you don't need to worry about security. *Nothing could be further from the truth.* Web programmers have the greatest need to understand security issues as web programs are the source of a huge number of machine and data compromises.

Web programs run on your server, with access to your data, on behalf of strangers who have unknown motives. Many of these strangers will be neutral or benign. Some will be malicious. All are security risks, because the problems they cause (even by accident) can alter your systems, and corrupt your data.

If you neglect the security issues inherent in writing code that gives strangers access to your servers, then you run the risk of giving those strangers more access than you intended. This is true whether your CGI programs are written in Perl, Python, C, Java, PHP or anything else.

Potential security pitfalls

Most of us wouldn't give shell access on a secure machine to any random person who asked. Neither would we install code from an unknown party just on their request. Yet it's surprising how often security is overlooked when writing code. Any time that a program accepts input from an unknown party and does not verify that input before using it to affect your system, it is inviting a security violation.

Cleaning up after security violations can be a tremendous job. It makes sense, therefore, to try to avoid them. Being aware of the issues is the first step; knowing how to avoid most of them is the second.

The biggest security pitfall in most programs (regardless of language) is best summed up as *unintended consequences*. Consider the following Perl code:

```
#!/usr/bin/perl -w
# DON’T USE THIS CODE
use strict;
use CGI;

my $filename = CGI->param(’file’);

open(FILE, "/home/test/$filename")
      or die "Failed to open /home/test/$filename for reading: $!";

# print out contents of requested file
print <FILE>;
```

In this code we have used the two-argument version of `open`. Further, we haven’t specified a mode for opening the file. Under normal circumstances, Perl will assume we meant to open this file for reading. To many beginners, this code looks innocent. Yet imagine that we pass in the value:

```
../../etc/passwd
```

Oops. We just printed out the contents of `/etc/passwd`! Now imagine that we pass in the value:

```
../../bin/rm -rf /home/test/ |
```

This tells Perl to execute the command on the left and pipe the output to the given filehandle. Printing out the contents of `/etc/passwd` is bad, but executing arbitrary commands is a disaster.

This isn’t rocket science. An average attacker can exploit this mistake to see the contents of files they shouldn’t, overwrite existing files and run system commands. Writing code like the above is like giving shell access to anyone who asks. And yet it’s such a common mistake.

Coding for security

Perl’s `open` function isn’t the only place where you can go wrong. Any function or operator that passes input via the shell requires careful attention, as it may contain *shell meta-characters*. Assuming you can’t just avoid all such functions and operators, the only way to ensure your code is safe is to *never trust input from the user*.

Fortunately this isn’t too hard, and can be done without too much effort. If we know what characters a field is allowed to have, we can use a regular expression to make sure that only these characters are used:

```
#!/usr/bin/perl -w
use strict;
use CGI;

my $filename = CGI->param(’file’);

unless ($filename =~ /^([\w.-]+)$/) {
        die "Filename is not valid!\n";
}

# Filename is okay (only contains A-Z, a-z, 0-9,  _, . and -)

open(FILE, "<", "/home/test/$filename")
      or die "Failed to open /home/test/$filename for reading: $!";

# print out contents of requested file
print <FILE>;
```

It is always better to specify what is allowed, rather than what is not allowed. This is because it's much easier to modify your expression to allow a few extra characters if necessary, whereas it is almost impossible to be sure that you've listed *all* the potentially bad characters.

However, even if we're careful, we can still make mistakes. Wouldn't it be nice if Perl could provide some extra level of security to ensure that we don't use untrusted input by accident? It can, by using *taint mode*.

Taint checking

It's always important that we validate our input, and this is particularly true if we're working in a security sensitive context. Unfortunately it's easy to forget our validation steps, even if you are programming defensively.

To help prevent this; Perl has a *Taint mode*. Taint mode enforces the following rule:

> You may not use data derived from outside your program to affect something else outside your program -- at least, not by accident.

Taint mode achieves its aim by marking all data that comes from external sources as *tainted*. This data will then be considered unsuitable for certain operations:

- Executing system commands
- Modifying files
- Modifying directories
- Modifying processes
- Invoking any shell
- Performing a match in a regular expression using the `(?{ ... })` construct
- Executing code using string eval

Attempting to use tainted data for any of these operations results in an exception:

> Insecure dependency in open while running with -T switch at insecure.pl line 7.

Tainted data is communicable. Thus the result of any expression containing tainted data is also considered tainted.

Turning on taint

Taint mode automatically enabled when Perl detects that it's running with differing real and effective user or group ids -- which most commonly occurs when the program is running setid.

Taint mode can also be explicitly turned on by using the `-T` switch on the shebang line or command line.

```
#!/usr/bin/perl -wT       # Taint mode is enabled
```

It's highly recommended that taint mode be enabled for any program that's running on behalf of someone else, such as a CGI script or a daemon that accepts connections from the outside world. Once taint checks are enabled, they cannot be turned off.

Using taint checks is often a good idea even when we're not in a security-sensitive context. This is because it strongly encourages the good programming (and security) practice of checking incoming data before using it.

Untainting your data

The only way to clear the taint flag on your data is to use a capturing regular expression on it.

```
($clean_filename) = ($filename =~ /^([\w.-]+)$/);

if (not defined $clean_filename) {
        die "Filename is not valid!\n";
}

# Filename is okay (only contains A-Z, a-z, _, . and -)
```

The contents of the special variables `$1`, `$2`, (and so on) are also considered clean, but it's *strongly* recommended that you use the list-capturing syntax shown above. `$1`, `$2` can be set to indeterminate-yet-clean values if your regular expression fails, whereas a list-capturing syntax guarantees `$clean_filename` will be undefined on failure.

Passing your data through a regular expression does not mean that it's safe to use. However it should force you to think about it first. There's nothing to stop you from bulk-untainting data with an expression like `/(.*)/s`, but doing so is extremely trusting of your data, and certainly not recommended.

Environment variables $ENV{PATH}

In addition to data our program receives while running, we also have to be aware of environment variables that can be set. In particular, if we are intending to make any system calls, we need to be aware of `$ENV{PATH}`.

The `PATH` environment variable tells Perl where to look for system commands we might invoke. However, since this value comes from outside our program, it contains tainted data. The best solution is to ensure that we set `$ENV{PATH}` to a known, good value:

```
#!/usr/bin/perl -wT
use strict;

$ENV{PATH} = q{/bin:/usr/bin};
```

Names can have odd characters

When constructing your taint checks keep in mind that people have names which may contain all sorts of letters. For example some names are hyphenated: *Anne-Maree*, others include spaces *Wellington Smith*. Some names even include punctuation: *O'Hara*, *Smith Jr.*, *Lt.-Col Ivan*.

Company names may include even more punctuation options: *Young&Jacksons*, *Yahoo!* etc.

When allowing characters for names, make sure you try to be as reasonable as possible. That doesn't mean you should allow any character in, but it does mean that if you're adding this information into a database, then you probably want to consider any consequences there as well!

Exercises

1. The HTML file `www/finger.html` asks the user for a username and passes that to the `www/cgi-bin/finger.cgi` program. Enter your username and see that it works.
2. Why is the data from the user tainted?
3. Turn on taint for `www/cgi-bin/finger.cgi`. Try re-submitting the form, it should fail.
4. Change `www/cgi-bin/finger.cgi` so that it untaints the data. Make sure that your script is only allowing alpha-numeric characters.
5. Try submitting the form with various usernames to test it. Make sure it rejects ones that are invalid. Below are some possible usernames to try:

   ```
   pjf
   1234
   %foo
   fred; echo $PATH
   fred;echo$PATH
   ```

Cross-site scripting

Cross-site scripting is an exploit where the attacker inserts malicious coding into otherwise trusted data. The malicious coding might be javascript designed to read cookies and submit that information to a third-party site, or to take advantage of a known browser bug. Or it might just be used to by-pass profanity filters, in order to upset your site's audience.

Using taint checking can do a lot to help avoid cross-site scripting attacks. So can using the `CGI.pm` module. For example, imagine that you have the following code:

```
use CGI;

my $cgi  = CGI->new();
my $name = $cgi->param('name');

print $cgi->header, "<p>Hello $name</p>";
```

what happens if the user submits the following name?

```
Fred
<script><!--
alert("Give me your money");
--></script>
```

This will generate:

```
<p>Hello Fred
<script><!--
alert("Give me your money");
--></script></p>
```

What we've done is allow an otherwise unknown user to execute javascript of their choosing on our website. This may read cookies, intercept mouse movements, or even rewrite our webpage in subtle or not-so-subtle ways. If our submitted information is used to populate pages visible to user users (as may happen in a content management system, wiki, online forum, or other site) then this sort of *cross site scripting* attack could be used to fool innocent users into revealing their their login details or other information.

We can avoid cross-site scripting attacks in a few ways. If we don't intend our user to be submitting HTML, then we can *escape* it before sending it to the browser. It converts HTML characters such as less-than (<) into HTML entities such as `&lt;`. CGI automatically escapes these characters when they're used as arguments for *form generating functions*. However it *does not* escape them when passed to any other functions, such as `h1()` or `p()`.

To make sure that we escape our text, we have to explicitly call CGI's `escapeHTML` function:

```
my $cgi  = CGI->new();
my $name = $cgi->param('name');

print $cgi->header, "<p>",$cgi->escapeHTML("Hello $name"),"</p>";

# Alternatively:

print $cgi->header, $cgi->p( $cgi->escapeHTML("Hello $name") );

# Another alternative:

my $safe_name = $cgi->escapeHTML($name):
print $cgi->header, "<p>Hello $safe_name</p>";
```

If you need to accept HTML for display then you may wish to examine the `HTML::Scrubber` and `HTML::Sanitizer` modules available from the CPAN.

Other forms might be submitted

One of the biggest mistakes people have made in the CGI programs is to believe that only *their* form will be submitted back to the server. Thus, if the HTML specifies that only 30 characters can be added to a field, this naive programmer may believe that the data returned will only ever have 30 characters for that field.

This is not true. Anyone can submit any form they like, from any server, to your CGI program. This means that they can edit the hidden price field on your form, to give them a better price than you were offering. It means they can submit hundreds of characters when your database is only expecting 10. It means that they can add fields, delete fields and generally do what they like to your form information. And your CGI program will have to handle it.

You cannot rely on client-side code to validate the data that a user might send. Because the client may not be using your form, or they might have client-side code disabled. So not only must you check that your data contains safe characters, but you must ensure that the data is the correct length if you have length restrictions and that other restraints are handled.

We'll cover more on data validation soon.

Privacy

Keep in mind when you code that under standard HTTP *everything* is submitted in the clear. Even though passwords are hashed out when users enter them, this is only to prevent casual over-the-shoulder disclosure. This information will still be submitted in clear text and may be stored during the journey.

If data security is important, if your data is at all sensitive, use secure HTTP. The HTTPS protocol opens a secure connection between the web client and server. All data on this connection is encrypted. This is essential for all transactions involving private information (such as medical details, bank information, credit card numbers, etc) more secure.

CGI scripts run on a secure server exactly as they do on any other server.

In this chapter...

- Security should be a major concern for all web developers.
- Web programs are run by unknown parties with unknown motives.
- We should never trust users' data.
- Taint mode helps identify unvalidated data from the user before we pass it to an external program.
- We can untaint our data by capturing it from a regular expression.
- `CGI.pm` can protect us from problems caused by cross-site scripting attacks.
- We cannot assume that data coming to our program was submitted from our associated form.

Chapter 7. Splitting HTML and code with HTML::Template

In this chapter...

Embedding HTML inside your Perl script can make it difficult to maintain both your HTML and your code. This is especially the case if somebody else is writing the HTML and may wish to change it at a later date.

To avoid the issue of mixing code and HTML, Perl has a number of useful templating modules which can be used to keep things separate. These have a great many advantages -- it's easy to change the interface, or have multiple interfaces available. Web-designers and programmers are less likely to step on each others toes, and people can use the most appropriate tools for each part. This chapter will explain the use of the `HTML::Template` module, although you should be aware that other templating modules are available.

The HTML::Template module does not come standard with Perl, but can be easily downloaded from CPAN (the Comprehensive Perl Archive Network) (http://www.cpan.org/). You can get documentation for it by reading **perldoc HTML::Template**.

If you need a more powerful templating system than `HTML::Template`, then you may wish to use Template Toolkit (http://www.template-toolkit.org/).

What is HTML::Template

Like the `CGI` module, `HTML::Template` is a module to help make your life easier when writing CGI scripts. Instead of embedding HTML into your code, `HTML::Template` allows you to load a custom template or blueprint and fill in special fields. If used properly, `HTML::Template` can eliminate the need to have any HTML in your script at all.

Here's a simple template that prints a library-book reminder.

```
<html>
<head><title>Library reminder</title></head>
<body>
<p>
Dear <!-- TMPL_VAR name="name" -->,
</p>
<p>
Don't forget that your book titled <!-- TMPL_VAR name="title" -->
by <!--TMPL_VAR name="author" --> is due back
<!-- TMPL_VAR name="duedate" -->.
</p>
<p>
If your book is returned late, a fine of $<!-- TMPL_VAR name="fine" -->
will apply for each <!-- TMPL_VAR name="timeperiod" --> the book
is late.
</p>
```

```
<p>
Yours sincerely,
<br />
<i>The management</i>.
</p>
</body>
</html>
```

The `TMPL_VAR` comments are used by `HTML::Template`, and get replaced with text supplied by the program at execution time. Here's a script that uses the template we've just seen to print a library reminder.

```
#!/usr/bin/perl -w
use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "library.html");

$template->param(
       name       => "Paul Fenwick",
       title      => "Programming Perl, 3rd Ed",
       author     => "Larry Wall, Tom Christiansen and Jon Orwant",
       date       => "next Wednesday",
       fine       => 2.20,
       timeperiod => "week"
);

print "Content-Type: text/html\n\n",$template->output;
```

Yes, it really is that simple. Since `HTML::Template` let's us split the HTML from the programming interface, we'll talk about them separately.

The template explained

`HTML::Template` provides a very powerful templating mechanism with many features more than just simple variable substitution. In this section we'll talk about these features, starting from the simple ones and proceeding onto more advanced topics.

Conventions

`HTML::Template` accepts two kinds of tags. In the example above, we used the HTML comments style. These allow us to create valid HTML which we can edit with standard HTML editors.

We can also use tags similar to standard HTML tags. These are more compact, but may upset various HTML editors, and are likely to cause problems with validation services. We can use either comment-style or tag-style templating methods, and we can mix both styles in the same document if we desire. Here's the example above using the HTML-style templating.

```
<html>
<head><title>Library reminder</title></head>
<body>
<p>
Dear <TMPL_VAR name="name">,
</p>
<p>
Don't forget that your book titled <TMPL_VAR name="title">
by <TMPL_VAR name="author"> is due back
```

```
<TMPL_VAR name="duedate">.
</p>
<p>
If your book is returned late, a fine of $<TMPL_VAR name="fine">
will apply for each <TMPL_VAR name="timeperiod"> the book
is late.
</p>
<p>
Yours sincerely,
<br />
&nbsp;&nbsp;&nbsp;<i>The management</i>.
</body>
</html>
```

In these notes and exercises we'll use the comments-style tags. These show up more clearly with syntax highlighting, allow us to validate our code, and are generally recommended.

There's another note of convention that we need to mention before we progress any further, and that's of filenames. `HTML::Template` doesn't care what the name of a template file is [1] so we can use anything we like. Programmers traditionally prefer to use files ending in `.tmpl` as it makes it obvious that they're templates. Web-designers, on the other hand, tend to prefer `.html` because it means their favourite HTML-editor is more likely to play nicely with the file. In these notes, we'll use `.html` -- after all, a plain HTML file is just a `HTML::Template` file without any special tags.

Simple template fields

With the code already presented we've shown how to use `TMPL_VAR` fields as place holders for data that we'll plug into the document at run-time. How they work should be fairly self-explanatory -- the tag is removed and the data we supply is inserted into its place.

`TMPL_VAR` and other templating tags don't need to obey the regular rules of HTML. For example, it's perfectly valid to have a template tag inside an HTML tag. The following code lets us set the alt tag on an image at run-time.

```
<IMG SRC="/images/picture.jpg" ALT="<!-- TMPL_VAR NAME=foo -->">
```

Exercises

1. View `www/cgi-bin/petpage.cgi` in your browser.
2. Take the HTML in `templates/petpage.html` and modify it to insert templating fields for `name`, `age` and `pet`.
3. View `www/cgi-bin/petpage.cgi` in your browser and observe the effects.

Non-web accessible templates

It should be noted that our templates are stored outside of our web-accessible document root, and there's a very good reason for this. Templates are not intended to be seen by the end-user - they need to be processed by one of our programs first. Serving a raw template is likely to be confusing to a user at best. At worst, it may disclose information that we intended to keep secret.

By storing the templates in a separate, non-web accessible directory, we avoid any risk of them accidentally being served to the world. We also have the advantage of keeping all of our templates in one place, making them easier to maintain the future.

The same applies to configuration files, modules, and libraries. We certainly don't want these being served by accident, as they may contain passwords or other sensitive information that could place our systems at risk.

Escaping in template fields

Sometimes we want to don't want our data to appear verbatim inside the HTML that we're producing. This is particularly the case if we're inserting data that might contain less-than or greater-than signs, or other characters that have special meaning. In this case we want to do *HTML encoding* Sometimes we want to encode information into a URL, in which case we want to do *URL encoding*. Sometimes we might even want to display data in its encoded and unencoded forms in the same page.

Rather than having to do this tedious escaping ourselves in our Perl code, we can get `HTML::Template` to do the hard work for us. This is also best illustrated by example.

```
This is how I escape for HTML:
    <!-- TMPL_VAR name="data" escape="html" --><br />
This is how I escape for a URL:
    <!-- TMPL_VAR name="data" escape="url"  --><br />
Here is my data with no escaping:
    <!-- TMPL_VAR name="data"               --><br />
```

Most importantly, by ensuring that we correctly escape our data we reduce the opportunity for *cross-site scripting* attacks.

If you're generating the contents of your template tags with the `CGI` module, then you won't need to use these escapes. As described previously in these notes, the `CGI` module will use the appropriate type of escaping needed for the task at hand.

Conditionals

Sometimes you'll want to display different content depending upon the execution of your program. In some cases we might select a template to use at runtime, depending upon if, for example, our user was borrowing or returning a book. In other cases, we might want to display fundamentally the same page, but choose to add or remove some sections depending upon circumstances. With our library example, we might want to display a reminder to the user if they have a book that's overdue, or alert them that a book they've placed on hold is available for borrowing.

We could use a `TMPL_VAR` tag which we can then bind to either the empty string or the HTML which contains our reminder message and associated formatting. That will work, but it potentially means having ugly chunks of HTML in our code, especially if the reminder comes wrapped in a table with images and special fonts. We started using `HTML::Template` to avoid this very situation, so isn't there a better way?

The solution is to use `HTML::Template`'s conditional tags. Here's our example above with an optional section that only gets displayed if a special message exists.

```
<html>
<head><title>Library reminder</title></head>
<body>
<!-- TMPL_IF name="message">
        <div id="message">
        <b>PLEASE NOTE</b>
        <!-- TMPL_VAR name="message" -->
        </div>
<!-- /TMPL_IF -->
<p>
Dear <!-- TMPL_VAR name="name" -->,
</p>
<p>
Don't forget that your book titled <!-- TMPL_VAR name="title" --> by
<!-- TMPL_VAR name="author" --> is due back <!-- TMPL_VAR name="duedate" -->.
</p>
<p>
If your book is returned late, a fine of $<!-- TMPL_VAR name="fine" -->
will apply for each <!-- TMPL_VAR name="timeperiod" --> the book
is late.
</p>
<p>
Yours sincerely,
</p>
&nbsp;&nbsp;&nbsp;<I>The management</I>.
</body>
</html>
```

`HTML::Template` uses the same rules for truth as does regular Perl. As you might expect, there's also `TMPL_ELSE` and `TMPL_UNLESS` tags too. Unfortunately there is no such thing as a `TMPL_ELSIF` tag.

There is no such thing as a `/TMPL_ELSE` (close TMPL_ELSE) tag. Instead you should close with the same tag that you used to open the conditional. That means that every `TMPL_IF` needs to have a matching `/TMPL_IF`, and every `TMPL_UNLESS` needs a matching `/TMPL_UNLESS`, regardless of whether you use `TMPL_ELSE` tags or not.

Exercises

1. Take the template in `templates/tmpl-cond.html` and modify it so that it displays some extra text and the contents of the `error` parameter only if it exists.
2. Use the `www/cgi-bin/tmpl-cond.cgi` script to test your changes.

Looping constructs

The examples that we've seen so far are great if we're dealing with singular pieces of data, but what if we want to display a list of books a user has currently borrowed? This list could contain any number of books. How would we write a template to deal with that?

A naive approach would be to use a `TMPL_VAR` tag where we wish to insert the list, then build that list and bind it into place. That's great, except now our program has the work of doing the HTML mark up for the list, and that's something we're trying to avoid. To solve this, we need to be able to deal with loops in our templates.

Loops in `HTML::Template` are almost identical in concept to Perl's `foreach` loops -- that is, we step through a list of values, examining the next one in our list every time we go through the loop. Let's see an example.

```
<ul>
        <!-- TMPL_LOOP name="books_borrowed" -->
        <li><i><!-- TMPL_VAR name="author" --></i>
             <b><!-- TMPL_VAR name="title"  --></b>
             (<!-- TMPL_VAR name="publisher" -->)
        </li>
        </TMPL_LOOP>
</ul>
```

Here we've printed an unordered list of books, with some special formatting for author, title and publisher. If we wanted to, we could have the title of the book come first, or we could print the books out in a table, or do any other formatting change, all without having to touch our Perl code at all.

These looping constructs can be very powerful. Your template can be set up to perform different actions for the first and last lines of your loop (for example, opening and closing table tags), and can distinguish between odd and even rows (for example, in case you want alternating rows to have different backgrounds). It's even possible to have conditional constructs based upon whether or not a given loop is empty or not. While the coverage of these concepts is beyond the scope of this course, all the information can be found using **perldoc HTML::Template**.

Including files

Often you'll be working on a website that has elements that are common to every page, like headers or footers, or huge animated advertisements with musical scores. If people are sensible, these common elements are usually placed into separate files and then inserted into the HTML using some mechanism depending upon your web-server or operating environment.

Now, it wouldn't it be nice if we could include these files when using our templates as well? Well, there's a better way than loading the contents into a `TMPL_VAR` tag, and that's using a `TMPL_INCLUDE` tag. Let's see some in operation.

```
<!-- TMPL_INCLUDE name="header.html" -->
Thank-you for flying with <!-- TMPL_INCLUDE name="airline_logo" -->
<!-- TMPL_INCLUDE name="footer.html" -->
```

`TMPL_INCLUDE` includes the file contents as if it were cut'n'pasted directly into the parent file at that point. This means that your include files can include templating information (including further include directives), just like your parent file. Since included files can include other files, there's potential to get into trouble with files endlessly including each other. `HTML::Template` provides some protection to this by only allowing includes 10 levels deep, although you can change or disable that if you like.

Using Template Objects

Now, you've all had some experience with writing templates, and as you can see it's possible to do this without any understanding of what the code that processes these templates looks like. That's an important thing to remember, someone doesn't need to know Perl (or any programming language) to create or edit a template. That's what makes them so useful.

In this section, we'll cover what the programmer needs to know in order to have templates work the way they expect.

Binding simple parameters

We've already seen a script that binds values to parameters in a template. We use the `param` method to set values. While this has the same name and a similar function to that of the `CGI` module, don't be fooled -- the way it processes arguments is subtly different.

To bind a value to a parameter, we pass in that parameter's name and its value, like this:

```
#!/usr/bin/perl -w
use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "library.html");

$template->param(title => "Programming Perl, 3rd Ed");
$template->param(author => "Larry Wall, Tom Christiansen and Jon Orwant");
```

As you can see from the example above, there's no need to set all the parameters in the same call, you can figure out parameters and set them as you go. If you do want to set a number of values at once, you can; just pass in as many name-value pairs as you need:

```
$template->param(
       name       => "Paul Fenwick",
       title      => "Programming Perl, 3rd Ed",
       author     => "Larry Wall, Tom Christiansen and Jon Orwant",
       date       => "next Wednesday",
       fine       => 2.20,
       timeperiod => "week"
);
```

You might have realised that these name-value pairs look awfully familiar to things we put into (and take out of) hashes. If you already have a hash of all the data you need, you can plug that directly into the `param()` method and things will work how you'd expect:

```
$template->param(%info);
```

If you try to bind a parameter that doesn't exist in the template you're using, an exception will be thrown (usually resulting in your script dying with an appropriate error). Often this is what you want, as it makes typos immediately obvious.

Sometimes you specifically *don't* want this behaviour. For example, you might write a subroutine which fills in information about the current user. The subroutine would like to provide that information without caring that the template will use all of it, and having your script die just because you didn't want to show the user's age can be a major headache. In these cases, you can request that `HTML::Template` just ignore parameters that don't exist. This is requested at the time you create the template, like this:

```
my $template = HTML::Template->new(
        filename          => "invite.html",
        die_on_bad_params => 0,
);
```

Binding complex parameters

We've seen how to deal with simple parameters, which are great for dealing with singular pieces of data, but they don't answer what we need to do for loops. That needs something a little bit more complex. We still use `param` to bind values, but instead of binding each parameter to a single value, we instead bind it to a list reference.

Now, we can't just use any old list reference. You see, `HTML::Template` lets us set a whole swag of different variables each time we go around one of its loops, and a simple list like `[3,4,6,7]` only contains a single value in each position. What we instead want to use is a list of hash references, because a hash *can* contain multiple name-value pairs.

Relax, it sounds difficult, but it's really quite simple, especially when you have an example to work by.

```
$template->param(library_books =>
        [
                {
                        title => "Programming Perl",
                        author => "Larry Wall, et al",
                },
                {
                        title => "Object Oriented Perl",
                        author => "Damian Conway",
                },
        ]
);
```

In the example above, our loop would have two iterations. The first time around we'd be dealing with the "Programming Perl" book, and the second time the "Object Oriented Perl" book. If you feel comfortable with references, you can build up this structure in other ways.

Exercises

This should bring all of these concepts together.

1. In `templates/total.html` you'll find a HTML template. This prints out the headers and footers, and an empty table. Add a looping construct to the table body so that we can fill in the table.
2. In `www/cgi-bin/total.cgi` you'll find scaffolding for this exercise. This scaffolding includes a number of hash references. Use these to populate the table from the above exercise.
3. You'll notice some more template variables in `templates/total.html`, pick some values and set these from your cgi program.

Associating other objects

Wow, this `HTML::Template` module is great stuff. I've got a CGI script which takes input from a user, and then displays some or all of it back on a confirmation page along with some other details. Being a good programmer, I'm much too lazy to pull everything out from my `CGI` object and push them back into my `HTML::Template` object. Is there any way I can do this automatically?

It so happens that `HTML::Template` allows you to *associate* a template with another object which has a parameter list, such as a `CGI` object. This means that if you don't provide a value for a given parameter, the value on the *associated object* will be used instead.

Say that a user has filled in their name, address, phone number, and number of plush penguin toys they own. Rather than having code like this:

```
use strict;
use CGI;
use HTML::Template;

my $cgi = CGI->new;
my $template = HTML::Template->new(filename => "penguinrego.html");

$template->param(
        "name",    $cgi->param("name"    ),
        "address", $cgi->param("address" ),
        "phone",   $cgi->param("phone"   ),
        "penguins",$cgi->param("penguins")
);
```

We can instead have code that looks like this:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use HTML::Template;

my $cgi = CGI->new;
my $template = HTML::Template->new(
        filename  => "penguinrego.html",
        associate => $cgi
);
```

That's it. Fields we don't fill in explicitly just get copied out of our CGI object without any extra work on our behalf. Fields that we do fill in ourselves will have those values. Nifty, isn't it?

Using CGI.pm with HTML::Template

We've already seen how `HTML::Template` and the `CGI` module can work together in sharing data. This section now explores using both modules together for maximum program maintainability, as well as a few common tricks that you might like to employ.

HTML::Template can only fill in the parts of our template for which we have tags. This is fantastic for single values. However consider the problem of a select list in which we're generating values based upon data available to our CGI application. Also consider that we wish to dynamically select a value, potentially based upon previous user input:

```
<select name="title">
<option value="Mr">Mr</option>
<option value="Mrs">Mrs</option>
<option value="Miss">Miss</option>
<option selected="selected" value="Ms" >Ms</option>
</select>
```

In a select list, we mark the selected value by adding the string: `selected="selected"`. Values which aren' t selected don't get this string. We can generate code for select lists with a TMPL_LOOP, however it isn't very tidy:

```
<select name="title">
<!-- TMPL_LOOP name="title_loop" -->
        <option value="<!-- TMPL_VAR name=value -->"
                <!-- TMPL_IF name="selected" -->
                selected = "selected"
                <!-- /TMPL_IF -->
        ><!-- TMPL_VAR name="value" --></option>
<!-- /TMPL_LOOP -->
```

Further, we need to ensure that we set a `selected` value for each item in our select list in our CGI script:

```
my @title_loop;
foreach my $title (qw( Mr Mrs Miss Ms ) ) {

        my $selected = ($cgi->param("title") eq $title);

        push @title_loop, {
                value    => $title,
                selected => $selected,
        };
}

$template->param(title_loop => @title_loop);
```

What a nightmare! This should be an easy matter, and so it is. All we need to do is get CGI to generate the select list and then plug that into our template! Now the above template becomes:

```
<!-- TMPL_VAR name="title_select" -->
```

and the CGI code becomes:

```
my $title_select = $cgi->popup_menu(
        -name   => "title",
        -values => [ qw( Mr Mrs Miss Ms ) ],
);

$template->param(title_select => $title_select);
```

That’s easier to read, maintain and much less likely to contain bugs. Unfortunately it reduces the ability for the template maintainer to control the HTML which is generated. There’s an alternative way of handling situations like this using `HTML::FillInForm` which we will examine in a few pages time.

Exercise

1. Write a basic form using `HTML::Template` and `CGI`. Make sure your form includes a text box, and either a checkbox group or select list. Ensure that you have template tags for the values of each form field.
2. Write a CGI script which prints out that form.
3. Submit your form and see that your text box value is not automatically filled in.
4. Associate your `CGI` object with your template, and submit some values. Make sure that your whole form is now automatically filled in.

Less templating with HTML::FillInForm

`HTML::Template` and `CGI.pm` make a great pair. However, sometimes it would be even easier if we didn't have to use them. Editing each HTML form to add templating tags for all the values is a chore. Generating each form element in CGI affects the opportunities for the web developers to have things appear exactly the way they want. It also means that the resulting template isn't complete, it's missing a number of form elements that are essential for its use.

Wouldn't it be nice, if we could just say: there's a HTML form over there, fill it in?

This is what `HTML::FillInForm` lets us do! `HTML::FillInForm` automatically inserts data into HTML input fields, text areas, radio buttons, checkboxes and select tags. It can be used to insert data from a database, a pre-existing form submission or purpose generated object or hash.

```
use CGI;
use HTML::FillInForm;

my $cgi = CGI->new();
my $fif = HTML::FillInForm->new();

# Fill in the HTML form with data from $cgi
my $output = $fif->fill(
        file    => "form.html",
        fobject => $cgi,
);

print $cgi->headers();
print $output;
```

The above example will open the file in `form.html` and fill it in with the data previously submitted (from `$cgi`). If nothing has been submitted, the form will use any default arguments; just as if you had navigated to it directly.

`HTML::FillInForm` isn't a replacement for `HTML::Template`. There are many situations where you may wish to use both of them together. For example, you may wish to prompt the user to correct some details on a registration page, using the same form that they filled in. This message can be added using `TMPL_IF` and `TMPL_VAR` tags.

To use `HTML::FillInForm` with `HTML::Template` we need to tweak the above code a little to allow us to give `HTML::FillInForm` the processed template:

```
use CGI;
use HTML::FillInForm;
use HTML::Template;

my $cgi      = CGI->new();
my $fif      = HTML::FillInForm->new();
my $template = HTML::Template->new(
        filename  => "form.html",
        associate => $cgi
);

# Add any extra stuff to the template, and then get its output
my $html = $template->output;

# Fill in the HTML form with data from $cgi
my $output = $fif->fill(
        scalarref => \$html,
        fobject   => $cgi,
);
print $cgi->header();
print $output;
```

That’s it! Now we can write our forms out in full HTML and have them "just work". We can use templates when we want to uniformly include content, or generate conditional material. CGI.pm processes and handles our parameters but is not necessary for most HTML generation.

Exercise

1. Change your template from the previous exercise to include the select list or checkbox group in the HTML.
2. Change your CGI script to use HTML::FillInForm.
3. Submit your form and verify that your select list or checkbox group is filled in.
4. Convert the rest of your template to just be standard HTML. Verify that `HTML::FillInForm` continues to fill in the form values.

Chapter Summary

- `HTML::Template` allows you to split your Perl code from your HTML code.
- You can use either standard tags or *comment tags* for writing templating fields. You can mix both in the same document.
- Template fields can be used to escape the text which is bound to them for both inclusion in HTML and in URLs.
- `HTML::Template` supports conditionals and loops, which are particularly useful for generating tables.
- `HTML::Template` can be used to include files into your HTML. These files are also evaluated for templating tags.
- To bind values to loops, we need to pass `HTML::Template` a list reference containing many hash references.
- It’s possible to *associate* a `CGI` object with a `HTML::Template` object, to have parameters submitted by the user automatically filled in.
- `CGI.pm` can be used to generate multi-part form elements for `HTML::Template`.
- `HTML::FillInForm` can be used to fill in a HTML form with data from `CGI.pm`.

Notes

1. In fact, you don’t even have to use filenames at all. You can pass `HTML::Template` other things to use as templates. See **perldoc HTML::Template** for more details.

Chapter 8. Data validation

In this chapter...

Data validation is essential for any program, regardless of its role, however it is particularly important when writing web applications. Users may be lazy, misguided, have poor typing skills, or may even be looking for a way to exploit your system. As such, it's important that we check and verify any data that is provided by the browser.

Client-side checking

One elegant way of performing validation is to use javascript and perform *client-side checking*. This allows our application to give immediate feedback if the user has missed a required field or provided invalid data. However client-side checking should *never* be relied upon. Users may have javascript disabled or restricted, and a clever attacker will simply submit data directly to your server. So while client-side checking will improve your user-experience, it does little to improve the overall security of your application.

Instead, it is essential that all data is checked on the server before being used. Perl's taint mode can assist as a safety harness in this regard, however it doesn't solve all possible problems.

Although we don't cover client-side checking in this course, there are a number of excellent Javascript resources to help you. In particular, the JSAN (JavaScript Archive Network) library's `Data.FormValidator` can be built from the same profile as `Data::FormValidator` (covered below) to provide consistent checking both client and server-side.

More information about `Data.FormValidator` can be found at http://www.openjsan.org/doc/u/un/unrtst/Data/FormValidator/.

Simple server-side checking

Below is an example of basic data validation. A form to submit to this file can be found in `www/validate.html` and the below code in `www/cgi-bin/validate.cgi`.

```
#!/usr/bin/perl -w
use strict;
use CGI;
use CGI::Carp qw(fatalsToBrowser);

my $cgi = CGI->new();

print $cgi->header;
print $cgi->start_html({-title=> "Validation Script"});

my @errors;
push (@errors, "Year must be numeric")        if $cgi->param('year') !~ /^\d+$/;
push (@errors, "You must fill in your name")  if $cgi->param('name') eq "";
push (@errors, "URL must begin with http://") if $cgi->param('url')  !~ m{^http://};
```

```
if (@errors) {
        print $cgi->h2("Errors"), "\n",
              $cgi->start_ul,    "\n";

        foreach (@errors) {
                print $cgi->li($_), "\n";
        }

        print $cgi->end_ul,      "\n";
} else {
        print $cgi->p("Congratulations, no errors!"), "\n";
}
print $cgi->end_html;
```

Exercise

Open the form for the validation program in your browser. Try submitting the form with various inputs.

Group exercise

1. What, if any, issues can you spot with the current validation checks that we have?
2. How can we improve these checks?
3. How could we handle validating multiple fields with similar constraints: multiple graduation dates, first name fields vs surnames, etc.

Data::FormValidator

The version of `Data::FormValidator` we show off in this chapter is 3.63. Later versions are backwards compatible, but may have different recommendations. To learn more about `Data::FormValidator` visit its page on the CPAN at http://search.cpan.org/dist/Data-FormValidator/.

As identified in the above exercises, form validation can get both very repetitive and rather complicated. Since it’s such a common problem, there’s already a good solution. `Data::FormValidator` allows us to specify which fields are required, and also to specify validation functions for each of these fields. There are some defaults we can use, but we can also create our own. The collection of these rules is called a profile.

Required and optional fields

The most logical part of our profile to start with is that which specifies what is, and isn’t required. We can do this by creating the keys in our profile hash:

```
my %profile = (
        required => [ qw (
                title firstname lastname email postcode
                phone check radio )
        ],
        # These fields are optional
        optional => [ qw ( initial title_other ) ],
);
```

The list of required fields plus the list of optional fields should cover all the fields on our form.

Once we have these fields we can perform some very basic validation on our form:

```
my $results = Data::FormValidator->check( $cgi, \%profile);

if( $results->has_missing() ) {
        $template->param( message => "The following fields are ".
                           "missing: " . join(" ", $results->missing());
}
else {
        $template->param( message => "Everything accounted for!");
}
```

Exercise

1. You can find a simple validation program in `www/cgi-bin/data_validation.cgi` with a corresponding template in `templates/data_validation.html`. Edit the profile to make the `name` and `year` fields required. Make the `favourite_bird` and `favourite_animal` fields optional.

Dependencies

Sometimes we want to require one field, only if another has been filled out, or has a certain value. For example, if we provide a limited range of titles (Mr, Mrs, Ms, Miss, Dr) we may also add an "Other, please specify" option. In this case, we'll want the "Other" box filled in.

Another example is for things like credit cards. If the credit card number is filled in, then we'll want to ensure that the card type, expiry date and card holder name are also included, even though filling in the credit card details itself may be optional.

We can do this by using the dependencies keyword in our profile hash:

```
my %profile = (
        dependencies => {
                # If our title is "other", we need the "other"
                # box filled in.
                title => {
                        other => [ "title_other" ],
                },
                ccard_no => [ qw( ccard_type ccard_exp ccard_name ) ],
        },
);
```

Exercise

1. Edit your profile to make the `favourite_bird` field required if the `favourite_animal` field is `bird`.

Constraints

The guts of the profile is in the constraints. These tell `Data::FormValidator` how to *validate* our data. The constraints hash can contain fields that are not mentioned in your `required` or `optional` hashes. This is handy as it means you can use the same set of constraints (thus keeping them all in one location) for all of your applications and pass different `required` and `optional` hashes for each page.

For the most part, our constraints are regular expressions:

```
my %constraints = (
        firstname =>   qr{^[a-zA-Z,.& -]+$},
        postcode  =>   qr{^\d{4}$},
);
```

We can also name our constraints, which comes in handy later:

```
my %constraints = (
        firstname =>   {
                name       => "name"
                constraint => qr{^[a-zA-Z,.& -]+$},
        }
        postcode  =>   qr{^\d{4}$},
);
```

`Data::FormValidator` has a number of predefined constraints, which you can read about using `Data::FormValidator::Constraints`. We can refer to these predefined constraints in our profile by name:

```
my %constraints = (
        firstname =>   {
                name       => "name"
                constraint => qr{^[a-zA-Z,.& -]+$},
        }
        postcode  =>   qr{^\d{4}$},
        email     =>   'email',
);
```

Finally, we can specify our own constraint rules by passing in a subroutine reference (more on this later in this chapter):

```
my %constraints = (
        firstname => {
                name       => "name"
                constraint => qr{^[a-zA-Z,.& -]+$},
        }
        postcode  => qr{^\d{4}$},
        email     => 'email',
        phone     => \&validate_phone,
);

# and later:
sub validate_phone { ... }
```

Anything not covered by a constraint rule is assumed to be valid regardless of its value.

Adding constraints to our profile

To add our constraints to a profile, we just pass in a reference:

```
my %profile = (
        required     => [ ... ],
        optional     => [ ... ],
        dependencies => { ... },
        constraints  => \%constraints,
);
```

Checking for validity

Our above example of using `check` only checked whether data was missing. We can also check to see if anything was invalid:

```
my $results  = Data::FormValidator->check( $cgi, \%profile );

if( $results->has_invalid() or $results->has_missing() ) {
        $template->param( message => "There were some errors." );
}
else {
        $template->param( message => "All valid!  Thankyou." );
}
```

`Data::FormValidator` provides methods to see which fields were missing or invalid:

```
foreach my $field ( $results->missing() ) {
        print "$field was required but is missing.";
}

foreach my $field ( $results->invalid() ) {
        print "$field has been given an invalid value.";
}
```

Exercises

1. Write constraints for each field in your form.
2. Test your form by submitting valid and invalid data. Check that you get the results you expect.
3. Add a constraint name to your `year` field and a corresponding message.

More complex validation

Regular expressions are a great for simple validation, but what if we want to perform other transformations on that data? For example, Australian telephone numbers may start with the country code (61 -- usually preceded by a plus) and a single digit area code. Alternately they may start with a two digit area code, possibly in parentheses. They may have no punctuation at all, or they may contain spaces or dashes separating the digits.

Writing a regular expression to match all of these options isn't easy. Wouldn't it be easier to get access to the data, strip out all the extra formatting that people like to add, and then use a regular expression to verify the numbers themselves? That's the kind of thing we might want to put into a subroutine.

`Data::FormValidator` allows us to specify subroutines instead of regular expressions as our constraints. These subroutines are called in scalar context and should return the data if valid, or nothing if invalid. Zero values are treated correctly.

We can provide subroutines for our data validation in two ways. The first is to provide an anonymous subroutine:

```
my %constraints = (
        ...
        phone     => sub {
                my ($val) = @_;
                my ($match) = ($val =~ /^([\d +()-]+)$/);
                return $match;
        },
        ...
);
```

The above example does not fulfil all of our requirements as it does not count the number of digits, nor does it strip out additional punctuation. In order to do that we need a longer subroutine, and rather than clutter our constraints hash we can use a reference to subroutine defined elsewhere in our file:

```
my %constraints = (
        ...
        phone     => \&validate_phone,
        ...
);

# and later:

# This only does Australian-style phone validation

sub validate_phone {
        my ($phone) = @_;

        # Remove spaces, parentheses and dashes
        $phone =~ tr{ ()-}{}d;

        # +61 should have nine digits after it.
        if ($phone =~ m{^ \+61 ( \d{9} ) $}x ) {
                # Return 0-prefixed form, without country code.
                return "0$1";

        } elsif ($phone =~ m{^ (0 \d{9} ) $}x ) {
                return $1;
        }

        # If we reach here then our phone number wasn't
        # valid.
        return;
}
```

When we provide a reference to an existing subroutine, we must not include parentheses. Writing the following:

```
\&validate_phone()      # oops!
```

calls the subroutine and takes a reference to the return value.

Exercises

1. Change the constraint on your `year` field such that it must specify a leap year. Use a subroutine constraint to test this.

   (Hint: A leap year is a year which is divisible by 4, but not by 100, unless it is also divisible by 400.)
2. Test your form by submitting valid and invalid data. Check that you get the results you expect.

Error messages

Sometimes it would be useful if we could provide more information than just the fact that the field is invalid. For example, we might want to tell the user what the field is allowed to contain so that they can spot what's wrong.

We can do this by mapping our constraint names (as mentioned above) to messages.

```
my %msgs = (
        constraints => {
                name     => q{ Names may only contain letters
                               in the alphabet, commas, dots,
                               ampersands, spaces and hyphens.
                },
                postcode => q{ Please enter an Australian
                               postcode of exactly four digits.
                }
        }
);

my %profile = (
        required     => [ ... ],
        optional     => [ ... ],
        dependencies => { ... },
        constraints  => \%constraints,
        msgs         => \%msgs;
);
```

Like the constraints, we can store messages in the messages hash that don't refer to values in the `required` or `optional` lists. This allows us to store consistent messages for all of our constraints across all of our applications, in one place.

We can also specify other information with our messages, such as custom error prefixes, the default missing and invalid messages and formatting string. Read the documentation (http://search.cpan.org/dist/Data-FormValidator/lib/Data/FormValidator.pm) for more information.

Using our error messages

We can access the messages that we've entered by calling the `msgs()` method on our results object. It returns a hash of field name, message pairs such as:

```
firstname => '* Missing'
```

These messages are designed to appear next to the form elements they refer to.

So how can we get those messages next to their form elements? Well, they're not form data, so `HTML::FillInForm` can't work its magic here for us. We'll have to add template tags. We could go forth and add them in ourselves, but this would add a lot of clutter to our HTML and distract from our form layout. Alternatively, we could use a Perl module to do it for us:

```
use Data::FormValidator;
use Data::FormValidator::Util::HTML qw(add_error_tokens);

# Add the error tags for HTML::Template
my $templ = add_error_tokens(
        html   => "example.html",
        prefix => "err_",
);

# Use the above template to create our template object
my $template = HTML::Template->new( scalarref => \$templ );

# Do validation and then check results
my $results  = Data::FormValidator->check( $cgi, \%profile );

if( $results->has_invalid() or $results->has_missing() ) {
        $template->param(message => "There were some errors");
        $template->param( $results->msgs() );
}
else {
        $template->param(message => "All valid!  Thankyou.");
}
```

`Data::FormValidator::Util::HTML` takes the HTML provided to it and adds `HTML::Template` tags to each of the form fields. These are named according to the prefix specified in the `add_error_tokens` function. Thus in our case they'll be called `err_title`, `err_firstname` and so on.

We can specify a prefix for each of our error messages in `Data::FormValidator` by adding the `prefix` key to the `msgs` hash:

```
my %msgs = (
        prefix      => "err_",
        constraints => {
                name => q{ Names may only contain letters in the
                           alphabet, commas, dots, ampersands,
                           spaces and hyphens.
                },
        }
);
```

The above approach of using `Data::FormValidator::Util::HTML` only works if you are not generating any form elements with `CGI.pm` and plugging them in via `HTML::Template` or another templating tool. If you are generating form elements this way (as described in the previous chapter) then `Data::FormValidator::Util::HTML` will not see them and thus will not be able to generate the message tags for you. Of course you can always generate them yourself.

Exercises

1. Add messages for your year and favourite bird constraints. You will need to add names to your constraints in your constraints hash.
2. Test your form by submitting valid and invalid data. Check that you get the results you expect.
3. Add a constraint name to your `year` field with a corresponding message.

Validation and tainting

Most validation with `Data::FormValidator` is done using regular expressions. If we write these regular expressions carefully then we can be certain that the resulting data is not only valid, but that it can also be safely untainted. To make this easy, `Data::FormValidator` will untaint valid values for us if requested.

We can mark particular fields to be untainted by using the `untaint_constraint_fields` parameter:

```
my %profile = (
        ...,
        untaint_constraint_fields => [ 'year', 'bird' ],
);
```

After this, values returned by the `valid` method are untainted for the specified fields. All other data will remain tainted.

Exercise

1. Enable untainting for your profile:

   ```
   my %profile = (
           ...,
           untaint_constraint_fields => [ 'year', 'bird' ],
   );
   ```

   Use the following code to test your valid values for taintedness:

   ```
   use Scalar::Util qw(tainted);

   foreach my $field ( $results->valid() ) {
           if( tainted( $results->valid( $field ) ) ) {
                   print "<p>$field is still tainted</p>";
           } else {
                   print "<p>$field is not tainted</p>";
           }
   }
   ```

   Be careful! `$results->valid( $field )` will return an array reference, if there was more than one value for that field name.

Validation code and modules

Many of your programs will have identical constraints on the fields of the same names. It's often a good idea to place your constraints, messages, and dependencies into a module where they can be easily accessed. This means that each program that needs to perform validation need only provide its list of required and optional fields to work.

Chapter Summary

- Client side data validation should never be relied on.
- Ad hoc data validation is difficult, time consuming and error prone.
- `Data::FormValidator` provides us with a very straight forward method of validating our form fields.
- We can use `Data::FormValidator::Util::HTML` to generate `HTML::Template` fields in our HTML to contain error messages.
- `Data::FormValidator` allows us to use our validation expressions to untaint our data.

Chapter 9. Cookies and sessions

In this chapter...

Unlike traditional applications such as word processors, calculators and flight simulators, CGI programs are not inherently stateful. This means that each request from a client is treated as an entirely new conversation with effectively no reference to any previous conversation. For static HTML this stateless situation causes no problem. The server does not care whether the browser requesting page 3 has previously seen pages 1 and 2.

Dynamic websites, on the other hand, do care about state. It would be terrible if there was only a single "shopping cart" used by all clients currently talking to the server. It would be impossible to order just the things you wanted, yet alone correctly handle payment and shipping for just your share!

We could try and identify each machine by its IP address, but due to the presence of proxies, network address translation, and dynamic address assignments this is rarely a workable solution.

Fortunately, there's a fairly simple solution. HTTP cookies.

What are cookies?

A cookie is a message given to a web browser by a web server. The browser stores this information and passes it back to that server each time it requests a page.

The main purpose of cookies is to associate information with a specific browser. The server can then use this information when making choices such as how to generate and serve content. For example, a cookie may record a user's preference in how they want particular elements laid out on the screen.

Uses for cookies

Cookies can be used for all sorts of things. They can be used to help gauge how many unique visitors a site has, and how often those visitors are returning. They can be used to track a user's movements throughout your website, or to save a user's page preferences semi-anonymously. Cookies are used to store session information for shopping carts and other site-related data.

Cookies might be used to ensure download agreements are seen, allow visitors to *bookmark* the page of the article they are reading, and to generate bread-crumbs to show the user where else on the site they've been. They can also be used to highlight new content since the user's last visit.

The use of cookies for some of these things is quite controversial. Most people are happy for their use for shopping carts, but unhappy about search engines using cookies to collate their search terms, or for marketing firms to use them for targeted advertising.

Naming cookies

Each cookie is a simple name/value pair, and *all* the information in a cookie is visible to any user who wishes to check it. Some browsers can be configured to ask the user if they wish to accept a particular cookie, and the name of the cookie may influence their decision.

It is suggested that you give cookies straightforward and easy to understand names. If your cookie records a user's preferred style, then call it `style` or `preferred_style`. This is both friendlier to your users, and can make your application easier to debug.

Cookie Security

Like all information sent from the web-browser, the user ultimately has *absolute control* over the cookies stored on their machine and transmitted to your application. All cookies should be regarded as user-input, and should be carefully checked and validated before use.

Before using a cookie for any purpose, you should ask yourself whether your application would break or could be subverted if the cookie's value was changed. It would be a *very* bad idea to have a cookie that recorded if the current user had administrative privileges; a clever user could simply set this cookie and bypass your access controls.

Later in this chapter we will discuss how you can use cookies to *reference* information in a secure fashion, without revealing that information to the user themselves. This is useful for tracking logins, privileges, and other information that the user should not be able to see or edit directly.

It should also be noted that cookies are sent to the server with *every request*, including for images, javascript files, cascading style sheets, and other content. You should avoid storing large amounts of information in cookies as they can provide a significant load on both your server and your clients. Browsers are not required to accept cookies of more than 4096 bytes, or more than 20 cookies per domain.

You should also remember that some browsers may not support cookies, or may have them disabled. You should always test that your web application provides a sensible response if cookies are not enabled, and does not get stuck in an endless redirect or display an unexpected error message.

Generating cookies with CGI::Cookie

Perl comes with a standard module called `CGI::Cookie` that allows cookies to be both accessed and generated. Perl's `CGI.pm` module uses `CGI::Cookie` internally for cookie management. In this course we will be discussing how to use `CGI::Cookie` directly, as it is useful in many circumstances when `CGI.pm` is not used.

`CGI::Cookie` allows you to control all aspects a cookie. A complete example is shown below, although it should be noted that most arguments are optional, as `CGI::Cookie` provides sensible defaults. We'll discuss each option as we progress through this chapter.

```
use CGI::Cookie;

my $cookie = CGI::Cookie->new(
        -name    => 'favourite_colour',
        -value   => 'blue',
        -expires => '+1d',
        -domain  => '.perltraining.com.au',
        -path    => '/',
        -secure  => 0,
);
```

Once we have a cookie, we can send it to the browser. Cookies appear as part of the HTTP header. When using `CGI.pm` we can provide an argument to the `header` function:

```
print CGI->header(-cookie => $cookie);
```

Multiple cookies can be sent by passing an array reference:

```
print CGI->header(-cookie => [$cookie1, $cookie2, $cookie3] );
```

The arguments that can be passed to `CGI::Cookie` are as follows:

name

Each cookie sent to a client should be given a name which relates to its purpose. For example `session_id`, `preferences` or `breadcrumbs`. Sending a cookie with the same name as an existing cookie (for the same site) will *overwrite* that cookie. In this way cookies can be updated or deleted.

value

Cookies can be thought of as key/value pairs. The value contains any information we want to store in the cookie, which may be preferences, a session hash, the contents of a shopping cart, or other useful information.

`CGI::Cookie` supports the setting of *simple* scalars, array references, and hash references using the `value` parameter:

```
my %prefs = (
        size    => 'big',
        colour  => 'red',
        texture => 'shiny',
);

my $cookie = CGI::Cookie->new(
        -name   => 'preferences',
        -value  => \%prefs,
);
```

For more complex data structures it is necessary to *serialise* the structure into a string first. We can use Perl's built-in `Storable` module for this purpose:

```
use Storable qw(freeze);

my $cookie = CGI::Cookie->new(
        -name   => 'preferences',
        -value  => freeze(\%prefs),
);
```

expires

Without an expiration time, `CGI::Cookie` creates a cookie that lasts until the end of the current session. In almost all circumstances, the session ends when the user closes their browser, although users can always expire cookies early if they wish.

Expiration dates are usually set relative to the current time. For example, a cookie may expire "one hour from now", or "one day from now". The following format types may be used:

```
+10s            10 seconds from now
+10m            10 minutes from now
+10h            10 hours from now
+10d            10 days from now
+10M            10 months from now
+10y            10 years from now
now             right now
-1y             one year ago (already expired)
```

Setting an expiry date in the past is the standard way to force a cookie to be removed from the browser.

It’s polite to not use overly long expiry times on your cookies without a good reason. If your cookie is to help record registrations for an event that is happening in a month’s time, then there’s little reason to keep your cookie for longer than that. If your cookie’s purpose is a long-term "remember me" function, then it may be sensible to set it to last for years.

A useful trick with cookies is to set a relatively short expiry time (eg, "+10m"), but to send the cookie on each request. This means that the cookie will expire after 10 minutes of inactivity from the client, which can be useful if you want a simple inactivity logout for your application. Unfortunately such cookies will persist (during the allocated timeframe) even if the browser is closed and re-opened, which may not be desirable for your application. Our chapter on session management covers this concept in more depth.

domain

The partial or complete domain for which your cookie is valid. The client will pass the cookie to any host which matches this domain. For example specifying a domain of `.perltraining.com.au` will ensure that the cookie is passed to servers on `www.perltraining.com.au`, `example.perltraining.com.au` and `testing.perltraining.com.au`.

If we only wished to pass cookies to `testing.perltraining.com.au` we could provide that instead as our domain. By default, the domain is set to the same server the cookie originated from. You can’t set cookies for a domain of which you are not a member.

path

Like the domain, this allows us to restrict the file system path our cookies are passed to. Thus a path of `/cgi-bin/user/` would ensure the cookie was passed to `/cgi-bin/user/order.pl` but not to `/cgi-bin/admin/delete_order.pl`. By default the path is set to `/` which will cause the cookie to be sent with all requests to your server.

"secure" flag

If the secure flag is set, the cookie will only be sent over secure connections using the *https* protocol. You should keep in mind that the *user* can still read the cookie from their browser, and the cookie may still be stored on their disk in an unencrypted format.

Fetching cookies

To get the cookie’s sent by the browser, we use the `fetch` method provided by `CGI::Cookie`, which can be called in either a list context (returning a hash), or a scalar content (returning a reference to a hash):

```
my %cookies        = CGI::Cookie->fetch;
my $cookie_hashref = CGI::Cookie->fetch;
```

Each cookie in our hash is a fully-formed `CGI::Cookie` object, however *only* the `name` and `value` attributes are meaningful; all other attributes are set to their default values.

Using `fetch` always returns *all* the cookies available. Once you have the cookie hash, you can retrieve an individual cookie from it:

```
my $pref_cookie = $cookies{preferences};

my $preferences;
if ($pref_cookie) {
        $preferences = $pref_cookie->value;
}
```

What we would really like is just the cookie. We can use `CGI.pm`'s `cookie` method to automatically fetch a cookie's value (if it exists).

```
my $preferences = $cgi->cookie('preferences');
```

If our value had previously been frozen with the `Storable` module, we'll also need to thaw it:

```
use Storable (thaw);

my $thawed_preferences = thaw($preferences);
```

We can also get a list of all our cookie names, which can be used to loop through all our cookies in turn:

```
my @cookie_list = $cgi->cookie();

foreach my $cookie_name (@cookie_list) {
        # Do something with $cookie_name
}
```

Exercises

1. The `www/cgi-bin/set_cookie.cgi` program will emit a simple cookie to your browser. Navigate there now, as having a cookie already set will help with the later exercises.
2. The `www/cgi-bin/cookie.cgi` program contains some skeleton code. Update it to retrieve a list of all cookies and display their names.
3. Update the code to also display the cookies' values.
4. Update the code to allow the user to set cookies based upon the fields provided. Test that this works.

Sessions

Cookies are a useful feature, allowing information to be kept in the browser itself, rather than being cumbersomely passed from page to page. However it's not without its flaws. Cookies can be modified by the user, making data integrity difficult, and they are inefficient when storing anything besides from small strings of data.

One way of keeping much of the usefulness of cookies but without the disadvantages is the concept of using *sessions*. Put simply, a *session key* or *session hash* is generated and given to the client as a cookie. On the server-side, a record is kept of that key and the data associated with it. This keeps the cookies small, our data secret, and makes it easy to verify whether or not our cookie is valid.

A good session cookie has two important properties. It should be unique for every session, and it should be randomly generated. The requirement for uniqueness is to make it possible to identify each individual session, but why the requirement that session cookies should be random?

The reason is simple. If session cookies contained predictable information, then it may be possible for an attacker to impersonate another user's session. By selecting random values from a large address space (eg, a 128 or 512 bit digest) the odds of an attacker guessing a session key is vanishingly small. Further security can be added by binding a session key to a particular IP address or other browser characteristics.

It's possible to use sessions without cookies, but doing so involves passing session information with every link and form submission. This can be a challenging task, especially for a website that mixes both static and dynamic content and is beyond the scope of this course.

CGI::Session

The `CGI::Session` module is available from the Comprehensive Perl Archive Network (CPAN), and provides an automated way to manage sessions. It provides a highly configurable way of storing session information, and can be used in conjunction with `CGI.pm` or with other technologies.

Getting started with `CGI::Session` is easy:

```
use CGI;
use CGI::Session;

# Create our new session
my $session = CGI::Session->new;

# Send our headers (including session cookie)
print $session->header();
```

There's a lot happening behind the scenes with our code above, so let's take some time to examine what happens in detail.

When we create a `CGI::Session` object it automatically checks the cookies that were sent by the browser. If it finds one named `CGISESSID` then it uses that as our session key. If it doesn't find any such cookie, then it generates a session key.

Calling `$session->header()` prints our standard HTTP headers, but also includes the session cookie. We can pass any arguments to `$session->header()` that we could pass to `CGI.pm`'s own `header` function.

Once we have our session established, we can use `CGI::Session`'s `param` method. This works exactly like the method of the same name from `CGI.pm`, except that instead of getting and setting parameters passed from the client, it gets and sets parameters in our local session storage area (on our server).

As an example, let's say that our user had successfully logged in, and we wanted to store that information:

```
$session->param('username',   'alice');
$session->param('fav_colour', 'blue' );
```

We can retrieve these details later, provided it's during the same session. It could be in a different script, or a different instance of the same script:

```
my $name   = $session->param('username');
my $colour = $session->param('fav_colour');
```

The above code assumes that we’re using a version 4 of `CGI::Session`, earlier versions required three arguments be passed into `new` to specify the data source name, session id and options.

```
my $session = CGI::Session->new(undef, undef, {Directory=>’/tmp’});
```

See the documentation for what your version requires.

Saving submissions

A very common use of sessions is to store information submitted previously by the user. For example, we may wish to remember the user’s favourite colour for later use. Unfortunately doing this results in a rather lengthy and somewhat repetitive line of code:

```
$session->param( ’fav_colour’, $cgi->param(’fav_colour’) );
```

Fortunately, `CGI::Session` provides a way to automate this process:

```
$session->save_param( $cgi, [ ’fav_colour’ ] );
```

We can even store many parameters at once in this way:

```
$session->save_param( $cgi, [ qw(fav_colour timezone username) ] );
```

Without a second argument *all* parameters are stored:

```
$session->save_param( $cgi );
```

If we need to ensure that the parameters are saved to disk immediately, we can call `flush`. Most of the time this isn’t necessary as the session object should call `flush` itself when the variable goes out of scope.

```
$session->flush();
```

While it’s certainly convenient to copy all form parameters to our session, it may not be very wise. Remember that the user can submit *any* parameters they like, and set them to any value they want. Blindly copying parameters into our session may overwrite or set parameters that we hadn’t intended. In almost all cases it’s *much* better to explicitly list the parameters we want saved.

Clearing session data

Clearing entries in our session is very straightforward:

```
$session->clear( ’fav_colour’ );
$session->clear( [qw(fav_colour fav_movie)] );

$session->clear();      # Careful!  Clears entire session
```

Deleting sessions

If you know a session isn't going to be used anymore (usually because the user has 'logged out'), you can delete it:

```
$session->delete();
```

This permanently removes the session from the store, this does not delete the cookie.

Exercises

1. Write a script which creates a session and run it. Verify that you receive a cookie from it named `CGISESSID`.
2. Create a HTML form which asks for name, age, street name and favourite colour. Submit this form to your script and store the submitted values in your session.
3. Print out the current values of the session before changing them to those from the new submission.

Session expiry

`CGI::Session` supports the concept of session expiry. If a session goes for a certain period of time without being used, then it's considered *stale* and will be cleaned up the next time we try to access it. This is useful for applications where we want the user to be automatically logged-out after a certain amount of inactivity. It's also a way to avoid accumulating sessions which never get cleaned up.

To set the expiry on our session, we can call the `expire` method:

```
$session->expire('+1h');        # Expires in 1 idle hour
```

Our code above does *not* say that the session expires in an hour. It expires after an *hour of inactivity*. The session itself could last for years provided that the user never spends longer than one hour between accesses.

It's also possible to cancel a session expiration, meaning that it will never expire.

```
$session->expire(0);
```

One of the nicest features of `CGI::Session` is that it's possible to *selectively* expire certain pieces of information. For example, we may want a parameter `payment_auth` to expire after only five minutes of inactivity:

```
$session->expire('payment_auth','+5m');
```

All parameters that aren't explicitly tagged will expire with the rest of the session.

When a session expires, *all* the parameters it holds are cleared. This means that setting a per-parameter expiry time longer than the session-expiry time effectively does nothing; the parameter will expire with the session, along with everything else.

It is not an error to set a parameter-specific expiry that's longer than the session expiry. An example of where this may be useful is when we want to limit a parameter's lifespan, but don't want to care if our sessions are short-lived, or longer-lived "remember me" sessions.

To expire an session immediately, use `$session->delete`. To expire a parameter immediately, use `$session->clear('name')`.

Session storage

Unless configured otherwise, `CGI::Session` stores its sessions in files on the disk, inside your operating system's designated temporary directory.

Storing sessions as files works everywhere, but it may not be the best solution for your application. If you're dealing with a large number of sessions, then storing them as individual files can get rather inefficient. In addition, there are housekeeping issues of having to clean them up.

To learn how to set `CGI::Session` to work with databases, read the documentation from **perldoc CGI::Session** or at http://search.cpan.org/dist/CGI-Session/lib/CGI/Session.pm.

Housekeeping

`CGI::Session` will automatically clean up expired sessions when they are used. However when a session expires, there's a good chance it will *never* be used again; the user has closed their browser, or gone on to do other things. This can result in an accumulation of expired sessions in storage, which take up space but which will never be touched again.

`CGI::Session` could try to do housekeeping every time it's called, but that would involve searching through all of our stored sessions and cleaning them up. That's not very efficient if your program is being run 100 times a second, and every single process is spending considerable time repeating the same work.

To avoid both of these problems, `CGI::Session` has separate housekeeping functionality. Now we can do our housekeeping at scheduled intervals (for example, every hour), and with a minimum of impact to client queries.

The housekeeping provided by `CGI::Session` is invoked by using its `find` class method. This walks through all the sessions, and executes code for each. At the most basic level, we can clean all our expired sessions using:

```
CGI::Session->find( sub {} );
```

That code walks through all our sessions and does *nothing* when it encounters each one, so how does it help? Well, before our subroutine is called, `CGI::Session` loads the session, and if it's expired, it removes it. So just by getting `CGI::Session` to examine everything, we get what we need.

We can also use this functionality to gain information about active sessions, or to even change them! The subroutine passed to `find` receives a fully-formed `CGI::Session` object each time it is called. So if we wanted to flag all our currently logged-in users for a prize-draw, we could do something similar to the following:

```
CGI::Session->find( \&prize_draw );

sub prize_draw {
        my ($session) = @_;

        # Skip expired sessions
        next if $session->is_empty;

        # Set prize-draw flag
        $session->param('prize_draw', 1);

        # Make sure our changes are flushed to storage.
        $session->flush;

}
```

If you’re using a store other than the default disk store you’ll also need to tell `find` where to locate it. `CGI::Session`’s documentation provides further information on how to do this.

As of `CGI::Session` 4.14, the `find` functionality is considered experimental. It is mentioned in these notes because it is extremely useful for housekeeping purposes, but you are strongly recommended to read the documentation for your installed `CGI::Session` module in case the syntax or semantics have changed.

Sessions and HTML::Template

`CGI::Session` works very nicely with `HTML::Template`. Because our session objects have a `param` method, we can associate them with a template and have fields automatically filled based upon our session information. The following code fills a template with information stored in our session, and then sends the filled template to the user:

```
my $session  = CGI::Session->new;

my $template = HTML::Template->new(
        filename  => "example.tmpl",
        associate => $session,
);

print $session->header;
print $template->output;
```

Exercise

1. Create a `HTML::Template` with tags for the name, age, street name and favourite colour from your previous script. Use the session object to fill these values in for each submission.

Chapter summary

- Cookies are small pieces of information given to the client by the server. These cookies are returned to the server with each request, allowing some state information to be kept.

- Users can edit their cookies, thus it is important to ensure that important data is verified.
- We can create cookies using `CGI::Cookie`.
- Sessions are created by giving clients unique ids and then storing relevant information for that id on the server.
- `CGI::Session` allows us to create, work with and expire sessions.
- If desired, we can associate `HTML::Template` objects with `CGI::Session` objects like we can with `CGI.pm` objects.

Chapter 10. Introduction to HTML::Mason

In this chapter...

In this chapter we'll introduce you to a powerful and integrated approach to web development. `HTML::Mason` provides a logical, high-level interface to web development making dynamic and semi-dynamic websites quick and easy to build.

Problems with classical CGI

There is a lot that can be done with classical CGI. Particularly once you start using some of the excellent helper modules such as `HTML::Template`. `HTML::FillInForm` and `Data::FormValidator`. However CGI still has a number of significant drawbacks.

CGI scripts typically map one HTML page per CGI program. Now each of those pages may have many logical parts: headers, footers, bread-crumbs, a search box, navigation, and actual content. This means that every CGI program ends up doing a lot of extra work (generating these other page parts) before it can do it's real task. Even if we put this generation into subroutines, we still need to make sure that each subroutine is called.

This lack of conceptual division of page elements, means that every CGI program is generating each page part, and must therefore handle the data for each template in the hierarchy. This leads to duplication across both code and templates. It also can lead to massive code rework when another page element is created.

This penalty for small changes often leads to CGI scripts falling behind the main website when look-and-feel changes occur. Even when they don't, due to the difficulty for non-programmers to understand CGI scripts, the programmers are constantly asked to make minor adjustments to ensure the pages look "just right".

Finally CGI scripts are often hosted on a separate (CGI-enabled) part of the web server. This means that making a website more dynamic may mean moving all of it into cgi-land, even when the only desire is to integrate something simple, such as a stock-report.

Due to these reasons and more, there have been a number of improvements and CGI replacements. In this second half of this course, we'll look at one replacement called HTML::Mason.

What is Mason?

Mason is a free, open-source, cross-platform web development environment, written in and supported by the Perl programming language. Mason's basic features can be used without any Perl knowledge as well, but to use Mason at full-strength an understanding of Perl is essential.

Mason's preferred environment is running under mod_perl inside the Apache web-server, where it makes use of numerous optimisations to enhance performance. Mason can also be configured to run under other environments (including IIS), operate as a standard Common Gateway Interface (CGI) program, or work in a non-web environment entirely.

For information on how to setup and administer Mason read Appendix A.

Mason vs traditional CGI

Traditional CGI programs are code with embedded HTML. In good programs the HTML is abstracted into templates and external files, but program execution still fundamentally follows the *code*, which then decides what HTML to generate.

Mason takes this model and flips it over. Mason sites consist of HTML that may contain embedded code. This means that a Mason file layout often exactly mirrors the site layout, which can make management significantly easier. Indeed, converting a static website to a Mason website is often a very straightforward process.

When Mason is used, it is normally enabled for *all* directories, or all files with a `.html` extension. This is the reverse of traditional CGI programs that are usually confined to a single directory.

A sample page

Unless otherwise designated, Mason pages are plain HTML:

```
<html>
<head>
<title>My First Mason Page</title>
</head>
<body>
<p>This is my first Mason page.</p>

<!-- This following line displays the result of a calculation. -->
<p>2 + 2 = <% 2 + 2 %> </p>

</body>
</html>
```

Of particular interest to us is the snippet containing the tags `<% 2 + 2 %>`. The special `<% %>` tags indicate that a Perl expression should be evaluated, and the result placed into the page at this point. This is commonly used to substitute variables, for example `Hello <% $name %>`.

Component Basics

Mason's basic unit is a component. This is a piece of HTML, possibly including Perl code, which generates a page or part of a page. In many cases you will have one component per page, but in other cases you may choose to call multiple components which each generate a small part of the final page.

An example component could be as simple as:

```
<p>Hello World!</p>
```

without any code at all. Of course, our components can also contain Perl code. We've already the uses of `<% expr %>` to evaluate and display the results of an expression, but we can also embed regular Perl code as well:

```
% my @friends       = qw(Paul Jacinta Damian Kirrily);
% my $random_friend = $friends[rand @friends];

<p><% $random_friend %> says "Hello World!"</p>
```

In the code above, a line beginning with a percent character (`%`) tells Mason to interpret that as a full line of code. The `%` *must* appear as the first character on the line, in all other locations it's considered just another character.

If we have a lot of Perl that we want to execute, we can use a special `<%perl>` block:

```
<%perl>
my @friends       = qw(Paul Jacinta Damian Kirrily);
my $random_friend = $friends[rand @friends];
</%perl>

<p><% $random_friend %> says "Hello World!"</p>
```

We can freely intermix Perl and HTML, which is especially useful in looping constructs and other areas of HTML generation:

```
<%perl>
my @friends       = qw(Paul Jacinta Damian Kirrily);
my $random_friend = $friends[rand @friends];
</%perl>

<p>My friends include:</p>

<ul>
% foreach my $friend (@friends) {
        <li><% $friend %></li>
% }
</ul>

<p><% $random_friend %> says "Hello World!"</p>
```

Exercises

1. You can find the above code in `www/friends.html`. Update the list with your own friends, and verify the generated HTML code changes accordingly.
2. Modify your code to also display the number of friends you have. (Hint: You can find the size an array with `my $num_friends = @friends`).

Calling components

One of Mason's great strengths is that it's possible to call components from other components. In this way we can create reusable snippets of code and HTML that can be embedded into multiple locations. We can consider Mason components to be similar to Perl's subroutines.

An example of a commonly used component is that of site navigation. We can write a component called `navigation.mhtml`, which may just contain static HTML:

```
<ul id="navigation">
<li><a href="/">Home</a></li>
<li><a href="/specials.html">Specials</a></li>
<li><a href="/games.html">Fun and Games</a></li>
<li><a href="/contact.html">Contact Us</a></li>
</ul>
```

We can now include this component to provide navigation on our pages:

```
<html>
<head><title>My Example Page</title></head>
<body>
<!-- Here’s our navigation -->
<& navigation.mhtml &>
<!-- Here’s our content -->
<p>Welcome to my example page!</p>
</body>
</html>
```

The `<& navigation.mhtml &>` tag calls the specified component and includes its content. We can call our components anything we want; our component could just as easily be named `nav.mhtml` if we wanted a shorter filename.

Components in other directories can be called by prepending the path:

```
<& ../shared/todays_weather.mhtml &>
```

The component root

All Mason components must live inside a directory hierarchy. The top level of this hierarchy is known as *the component root*. It is impossible to access components that live outside the component root. This provides an extra level of security (arbitrary files cannot be executed), and also allows Mason to perform a number of optimisations.

When Mason receives a request, it maps the request onto the filesystem by adding the component root to the start of the path. For example, a request for `/chickens/dorkington.html` and a component root of `/var/www/my-mason/` would result in Mason looking for a file with an absolute path of `/var/www/my-mason/chickens/dorkington.html`.

Filename conventions

When working with Mason, it is common to use a number of filename conventions. These have the advantage of making it easier for developers to quickly determine the use of a component, and also allowing rules to be specify to the web-server regarding the serving of content.

Throughout these notes we will use the following filename conventions, which we also suggest for your development and production sites:

.html

Either plain HTML, or a `HTML::Mason` file designed to be rendered to a browser. These may be referred to as *top-level components*, as they often contain an aggregation of smaller components to do their work.

.mhtml

A Mason component that produces HTML output, but which is designed to be called from another component, rather than displaying directly to a browser. Examples include navigation bars, weather displays, shopping cart summaries, and other components that may be included in a larger page.

.mpl

A mason component that produces no HTML output, but instead performs calculations and/or returns information to the calling component. In general we prefer the use of Perl modules to perform these tasks, but it may be appropriate under some circumstances to have Mason components that fulfil a similar role.

.cgi

A traditional CGI script.

Exercises

1. We've already written a `navigation.mhtml` file for you. Edit your existing `friends.html` to call the navigation component.

Chapter Summary

- Mason is a web development environment written in Perl.
- Mason's basic unit is the component.
- Components represent a logical part of a website.
- Components can call other components to build up a full page.

Chapter 11. Component Arguments

In this chapter...

We've examined some of Mason's basic features for generating dynamic content, and the basic use of components to build more sophisticated pages. However none of our pages accept form submissions, and none of our components take any arguments. In this chapter we'll cover Mason's argument handling features.

Form processing

Mason has a very straightforward way of processing arguments to forms. Any component can have a `<%args>` block, which specifies what arguments may be passed to the form. Let's see a simple example:

```
<%args>
$name => ""
</%args>

% if ($name) {
        <p><b>Hello <% $name %></b></p>
% }

<form method="post">
Your name:
<input type="text" name="name" />
<input type="submit" />
</form>
```

In our component above, we accept a single scalar argument `$name`, which has a default value of the empty string. When our form is submitted (by default back to the same page), then if `$name` contains a true value we display a bold message greeting that person.

Variables declared inside an `<%args>` block are lexically scoped (using `my`) for the entire component, and are automatically populated using form-submission variables of the same name. The `<%args>` block can appear anywhere in the component. It's not uncommon for `<%args>` to appear after the HTML in a component, which can make it easier when the HTML is written by non-programmers.

We can specify many arguments if required, and they need not all have defaults. It's also possible embed blank lines and comments into `<%args>` blocks. The following block provides a default `$name`, but requires that both `$age` and `$address` be passed:

```
<%args>

# While our users may not give their name, we
# require their name and address.

$name => "anonymous"    # optional field
$age                    # required field
$address                # required field

</%args>
```

Mason considers it a fatal error if an argument without a default is not supplied. It is *highly* recommended that any component wishing to use submitted form inputs supplies defaults to *all* of its arguments. You can and *should* always provide your own argument validation code.

Multiple form values

It’s very common for forms to have multiple elements with the same name. For example, the following provides a list of favourite colours that the user can select. The resulting form input may have no colours selected, all of them selected, or anything in between.

```
<p><b>Favourite colour</b> (tick all that apply):</p>

% foreach my $colour (qw/red blue green cyan black/) {
    <input type="checkbox" name="fav_colour" value="<% $colour %>" />
    <% $colour %><br />
% }
```

The simplest way to accept multiple form inputs with the same name is to use an array in our `<%args>` block:

```
<%args>
@fav_colour => ()
</%args>
```

We now have access to our submitted data (if any) using the `@fav_colour` array.

It should also be noted that we could have used a scalar variable:

```
<%args>
$fav_colour => []
</%args>
```

However this is *not* recommended. Mason will populate `$fav_colour` with an array reference if it receives two or more values, but will use a simple *string* if only a single value is received. Trying to determine if `$fav_colour` contains a reference or a string will just add needless complexity to our code.

This demonstrates the importance of checking our arguments. Since our form submissions can receive *any* number of arguments, we should never rely upon a scalar argument containing only a simple string.

We can force an argument to be a string or number with the following code:

```
ref($name) and die "name must not be a reference";
```

Exercises

1. The file `www/form.html` contains a simple form. Modify it to take arguments using an `<%args>` block and display the results in the designated area.

The %ARGS hash

Mason provides another way of processing arguments, and that's via the `%ARGS` hash. Inside a component, `%ARGS` contains a complete list of arguments that were submitted. Using `%ARGS` is particularly valuable when you expect a large or indeterminate number of arguments to your component, or you have form submissions that contain field names that cannot be valid Perl variable names. This may occur if you're using an image-input element that submits `name.x` and `name.y` fields to your component.

It's perfectly acceptable to use both an `<%args>` block and the `%ARGS` hash at the same time. For example, the following snippet of code would populate extra variables from an image-input submission:

```
<%args>
$name    => ""
$address => ""
</%args>

<%init>

# These inputs would be generated from a HTML tag such as:
# <input type="image" name="button" src="button.png" />

my $button_x = $ARGS{'button.x'};
my $button_y = $ARGS{'button.y'};

</%init>

<p>
Hello <% $name %> of <% $address %>, you clicked the
image at <% "($button_x,$button_y)" %>.
</p>
```

Another good example of using `%ARGS` is when we wish to validate our data using `Data::FormValidator`. We can simply pass through the entire `%ARGS` hash:

```
<%init>
use Data::FormValidator;
my $results = Data::FormValidator->check( \%ARGS, \%profile);
</%init>

% if ( $results->has_invalid or $results->has_missing ) {
%       # Re-display our form, giving it our results
%       # to allow for better feedback.
        <& form.mhtml, results => $results &>
% } else {
        <p>Thank-you for your submission!</p>
%       # Do something with our form submission.
% }
```

Exercises

1. Modify your `www/form.html` file to perform validation on the submitted fields. If validation fails, then display the `form_error.mhtml` component instead of your submission results.

Calling components with arguments

Argument handling is a straightforward and easy way to manage form submissions in top-level components, but its usefulness extends far beyond managing user input. We can also call Mason components and pass them arguments. For example, let’s pretend that we have a component that retrieves and displays information about a book, given the ISBN, and also takes an optional discount:

```
% # display_book.mhtml
<%args>
$isbn
$discount => 0
</%args>

<b><% $book->title %></b>, $<% $book->price %>

% if ($discount) {
        <br />
        <i>Just $<% $book-price * (1 - $discount/100) %>
        including your <% $discount %>% discount!</i>
% }

<%init>
use Local::Book::Catalogue;
my $book = Local::Book::Catalogue->new($isbn);
</%init>
```

We can now call our component with arguments like so:

```
<& display_book.mhtml, isbn => ’0596001738’, discount => 20 &>
```

and we could expect this to display:

```
<b>Perl Best Practices</b>, $75 <br />
<i>Just $60 including your 20% discount!</i>
```

This also demonstrates why we may wish to have arguments blocks that do not always include defaults. It would be an error for us to ever call this component without specifying an ISBN, and as it is not a top-level component there should be no risk of it being called directly by a user.

Component calls with arguments also introduces the concept of being able to pass *hashes* as arguments, as well as scalars and arrays:

```
% # user_details.mhtml
<%args>
$name
$address
@hobbies
%telephone
</%args>
```

We could then call this component like so:

```
<& user_details.mhtml,
        name      => "Bruce Wayne",
        address   => "123 Bat Ave",
        telephone => {
                home   => "555 1234",
                work   => "555 4567",
                mobile => "04 BAT PHONE",
        },
        hobbies => [ "chess", "electronics", "fighting crime" ],
&>
```

It should be noted that we always pass *references* to arrays or hashes to our components, just like we would if we were calling regular Perl subroutines. I

Although the content of an `<%args>` block looks like Perl code, it isn't. Each variable definition must be on a single line and lines do not end with semi-colons (`;`). Perl expressions are allowed, including those that refer to previous arguments.

Exercises

1. The `joke.mhtml` component takes two arguments, `joke` and `punchline`, and uses some simple javascript to display them. Call it now from one of your components, passing in the details of one of your favourite jokes.
2. Write your own component that takes a number of arguments (your choice) and formats them. Call this component from another component.
3. If you have time, modify your component so that one of its arguments is a list of values, and display the contents as an itemised list.

Chapter summary

- We can specify component arguments in the `<%args%>` block.
- Scalar values in our `<%args%>` block will contain array references if there were multiple values in the form by that name.
- The %ARGS hash gives us direct access to all of our arguments.

Chapter 12. Autohanders

In this chapter

In this chapter we will explore one of Mason's most powerful features, the *autohandler*. Autohandlers allow for broad modifications to be applied to content and components, such as the addition of navigation and style information. They may also be used for more advanced operations such as access controls.

Before we discuss autohandlers, we'll examine some of the reasons why a broad way of modifying or controlling content is essential to creating a maintainable website.

Consistency

The cornerstone of any good website is a consistent look and feel. Every page should have the same layout, navigation, and style; or it should differ from the norm in well-defined ways. Users are almost always looking for your content, and don't want to have the extra problems of dealing with changing navigation and layout.

Cascading Style Sheets (CSS) provide an excellent way to define how a site should look and behave. While an in-depth discussion of CSS is beyond the scope of this course, the correct use of CSS can significantly increase the maintainability of any website.

However even with the use of style-sheets there are still HTML elements that need to appear on every page. These can be logos, navigation menus, copyright information, and everyday structure to ensure that content appears where it should. When we want to update one of these elements, we need to do so for every page in our site. That's a real drag!

Luckily, Mason provides a system to elegantly solve the consistency problem. Put simply, whenever a page is called, a special component known as the `autohandler` has an opportunity to act first. Most autohandlers simply take the output of a page and wrap it in a set of standard headers and footers. This means that individual pages need to only contain *content*.

Let's see an example autohandler now.

```
<html>
<head>
<title>Ascidian Central</title>
<link rel="stylesheet" type="text/css" href="/style.css" />
</head>
<body>
<h1>Ascidian Central</h1>

% $m->call_next;

</body>
</html>
```

In the above code, we output some headers, and a title and finish up with some footers. In the middle we have a strange line: `$m->call_next`. This tells Mason to call the next item along in our chain, which will either be another autohandler or the final requested component.

By storing our header and footer in a component which is called automatically, we reduce the likelihood that a wrong header or footer component is called or that an important component is forgotten.

We also limit the scope of changes to our invariant sections to one file, regardless of whether we're adding a new navigation bar, further copyright sections or removing something.

The execution chain

When a Mason component is requested the following steps take place:

1. Mason checks to ensure the requested component exists. If it does not then Mason uses the `dhandler` (default handler), which is covered later in this course.
2. Mason opens the requested component and checks to see if there are any special inheritance directives.
3. If there are no special inheritance directives, Mason looks in the same directory as the component for an `autohandler` file. If there is a special inheritance directive, Mason follows that instead. We'll examine this in more detail later.
4. Mason then walks up the directory structure looking for further autohandlers.
5. Each autohandler is then executed in turn from top most to bottom most, followed by the component and any components it includes.

call_next

In the above example we use `$m->call_next` to call the next item in our chain, which is often the page component itself, but could be another autohandler in our chain.

Let's assume we have the following three files:

```
autohandler
===========

<html>
<head>
<title>Ascidian Central</title>
<link rel="stylesheet" type="text/css" href="/style.css" />
</head>
<body>
<h1>Ascidian Central</h1>

% $m->call_next;

</body>
</html>

cold_water/autohandler
======================

<!-- Begin section -->
<h2>Cold water varieties</h2>

% $m->call_next;
<!-- End section -->
```

```
cold_water/sea_tulips.html
==========================

<!-- Begin content -->
<p>
Sea Tulips are sessile filter feeders found in coastal waters at
depths up to 80m. Their name derives from their appearance  a
knobbly 'bulb' attached to a long 'stalk'. The colouration of Sea
Tulips depends upon their association with a symbiotic sponge that
covers their surface.
</p><p>
Despite their name Sea Tulips are animals, not plants.
</p>
<!-- End content -->
```

We'd access our sea tulips' content by navigating to `http://our.url/cold_water/sea_tulips.html` which would build the following html:

```
<html>
<head>
<title>Ascidian Central</title>
<link rel="stylesheet" type="text/css" href="/style.css" />
</head>
<body>
<h1>Ascidian Central</h1>

<!-- Begin section -->
<h2>Cold water varieties</h2>

<!-- Begin content -->
<p>
Sea Tulips are sessile filter feeders found in coastal waters at
depths up to 80m. Their name derives from their appearance  a
knobbly 'bulb' attached to a long 'stalk'. The colouration of Sea
Tulips depends upon their association with a symbiotic sponge that
covers their surface.
</p><p>
Despite their name Sea Tulips are animals, not plants.
</p>
<!-- End content -->
<!-- End section -->

</body>
</html>
```

Exercises

1. There is already a `www/autohandler`. Modify it to display a tagline, copyright notice, or other text at the bottom of each rendered page.
2. Modify the `www/autohandler` to include your navigation component on each page. You may also make any other site-wide modifications if you wish.
3. Create an autohandler inside your `www/products` directory that displays a simple title, or title and footer. Verify that pages inside the `products` directory contain this extra information, and those on the rest of your site do not.

Methods

Our new website is great! It’s easy to add new content, and it’s easy to change our layout and design. However, there is still one problem that can be seen. The title on all of our pages remains static.

Ideally, we’d like to have a way to query our content for information such as titles, keywords, access controls, and other per-page specifics. In Mason this is easy to do by using *methods*.

Methods are attached to a component, but can be called from anywhere. They allow components to provide extra services. It’s also possible to attach methods to autohandlers, providing a "default" method when no specific method exists on the component itself.

Calling methods is very similar to calling components:

```
<& /path/to/component:method &>
```

We could call the `title` method on our `index.html` component as follows:

```
<& index.html:title &>
```

Of course, in our autohandler our requested component will be different for each request. Luckily, Mason provides us with a shortcut. The special component `REQUEST` always refers to our current request:

```
<& REQUEST:title &>
```

We’re more likely to see this embedded inside `<title>` or `<h1>` tags, like this:

```
<title><& REQUEST:title &></title>
```

Now that we know how to call a method, we need to know how to write them. Creating a method is done using a special `<%method>` block:

```
<%method title>
Example title
</%method>
```

Our method can contain any valid Mason code, and can also take arguments, and do anything else that a normal component can do.

Exercises

1. Modify your `www/form.html` page to include a `title` method that returns a static title.
2. Modify your `www/autohandler` to call the `title` method on the requested component and display at the top of the page in `<h1>` tags.
3. Modify your autohandler also insert the title in between the `<title>` tags in the `<head>` block.
4. What happens if you try to browse to a different page that has no `title` method? We’ll learn about this more in the next section.

Default methods

Methods are fantastic when they exist, but attempting to call a method that does not exist results in a fatal error from Mason. It’s a lot of work to add methods to every component in our site, especially if our site is very large. Worse still, we may end up duplicating method code if we want many of our methods to work the same way.

Fortunately, we don't need to go to quite so much effort, as methods can be *inherited* from autohandlers. If we can't find a method on a component, then we look at its autohandler instead. If it's missing the method, we look at the autohandler's autohandler. This continues up the chain until a method is found, or we run out of autohandlers and an error is generated.

The great advantage of this inheritance is that we can provide a single method in our autohandler, and simply override it as needed for each individual component.

Exercise

1. Modify your `www/autohandler` to provide a `title` method that contains a default title.
2. Verify that your title default title is used on pages missing their own `title` method.
3. Now add a `title` method to the `www/products/autohandler` file. What happens to the title of content inside that directory?

Attributes

Closely related in concept to methods are that of component *attributes*. Just like methods, attributes can be set on components, and just like methods they can be inherited from autohandlers. However attributes are much simpler than methods, as they're just static data.

Attributes in Mason are set using a `<%attr>` block:

```
<%attr>
name            => "Bonsai Car BBQ"
price           => 99.95
show_navigation => 1
categories      => [ 'outdoor', 'floral', 'automotive' ]
</%attr>
```

Note that attributes in Mason are just simple key/value pairs, and can contain any type of scalar data, including complex data structures. Attributes are declared one per line, and there are no additional commas as you would find if you were declaring a hash.

In order to access attributes, we need to query our components. To do this we first need access to our *component object*. One of the most commonly used components is that of our request itself:

```
<p>The price is $<% $price %></p>

<%init>
my $comp  = $m->request_comp;
my $price = $comp->attr('price');
</%init>
```

We can also skip loading our component into a separate variable:

```
my $price = $m->request_comp->attr('price');
```

We can also fetch any component on our system:

```
my $comp  = $m->fetch_comp('/path/to/comp.mhtml');
my $price = $comp->attr('price');
```

Trying to fetch an attribute that does not exist will result in an exception, which unless caught will halt your component and signals an error to the Mason interpreter. To tell Mason it's okay if an attribute does not exist, we can use the `attr_if_exists` method:

```
my $categories = $comp->attr_if_exists('categories');
```

`attr_if_exists` returns the undefined value if the attribute does not exist.

Keep in mind that attributes can only ever contain *static* content. If you need to generate something dynamically, either from calculation, or looking it up in a database, or through other means, you'll *have* to use a method.

It's easy to think that attributes are perfect for things such as page titles, and for simple sites they are. However on complex sites the titles may be dynamically generated, and if you're using attributes you'll find yourself having to rewrite some of your site.

You can have the best of both worlds by providing a default method that simply performs an attribute lookup, which allows individual components to override that and supply their own dynamically generated methods when needed:

```
<%method title>
<% $m->request_comp->attr_if_exists('title') || "My Homepage" %>
</%method>
```

Changing autohandler inheritance

In most circumstances we want our components to inherit from their default autohandler, but there will sometimes be circumstances where we do not want this to be the case. One such example is generating non-HTML pages in Mason. We don't want our plain text file or our cascading style sheet to inherit from an HTML-centric autohandler, although we may wish to use Mason to generate those pages.

Another common example where we want to use a higher-level autohandler is in `index.html` files in subdirectories. If we have a directory filled with books, then the `index.html` is likely to be the only page that is *not* a book, and therefore requires an exception to our special book formatting.

Luckily, changing our autohandler is easy. Mason has a special `<%flags>` block, which allows us to modify component behaviour. The only key that is presently defined for `<%flags>` is `inherit`, which allows us to specify a new location for the autohandler:

```
<%flags>
inherit => /some/other/autohandler
</%flags>
```

If we set `inherit` to the special value of `undef`, then our component will execute without an autohandler at all.

```
<%flags>
inherit => undef
</%flags>
```

Exercises

1. Modify the `www/products/index.html` file to inherit directly from the top-level autohandler.
2. Check that your new `www/products/index.html` page contains only the top-level autohandler details. Ensure that other pages in that directory continue to use the normal inheritance chain.

Autohandlers for access-control

Autohandlers aren't just good for layout, they're useful for access control as well. Let's pretend that we have a 'members only' section of our website, and require that members login before granting access. Rather than having to worry about access control on each of our pages, we instead place our members only pages into their own directory, with an autohandler at the top:

```
% # File: /members/autohandler
%
% if (member_is_logged_in() ) {
%       $m->call_next;
% } else {
        <& login.mhtml &>
% }
```

Our autohandler simply does whatever work is needed to determine if the user is logged in. If they are, then we display the content as normal, and if not we display the `login.mhtml` component, which should display an appropriate message or login page.

Note that our autohandler contains no other formatting or special mark-up. Because our members autohandler is automatically invoked by the top-level autohandler, we know that layout and navigation is already managed for us.

Chapter summary

- Autohandlers allow us to provide consistency to our websites.
- All autohandlers from parent directories are also used.
- We can use methods to specify information like page titles.
- Default methods (in the autohandler) allow us to avoid errors from missing component methods.
- Attributes are static data which a component may have.
- We can change our autohandler inheritance, although this is rarely useful.

Chapter 13. Components in depth

In this chapter...

We've seen how Mason sites are built up from components, and we have already used a number of their special features. In this chapter we'll be taking an in-depth look at components and their capabilities.

Special Globals

$m

Mason's special variable `$m` contains the `HTML::Mason::Request` object. The allows us to to retrieve information on the current request, call other components and affect the flow of execution. `$m` always exists when using Mason.

You can read more about `$m` by using `perldoc HTML::Mason::Request`.

$r

When Mason is running under `mod_perl`, `$r` contains the Apache request object. This variable provides access to the full Apache API, thus allowing us to set HTTP headers, send messages to the Apache logs and access configuration information. Read `perldoc Apache::Request` to find out more.

It should be noted that `$r` *only* exists when running under `mod_perl`, or something that emulates a `mod_perl` environment. It's considered good practice to try and separate code that uses `$r` from the rest of your code, to make components easier to re-use and test.

%init and %cleanup blocks

One of the most commonly seen special-purpose blocks is the `<%init>` block. Conceptually, the `<%init>` block is identical to a `<%perl>` block at the start of your component. So why do they exist, and why are they so useful?

It's very common for a component to require some sort of initialisation. It needs to load modules, calculate values, perform queries, validate input, or otherwise perform setup. We could just put this at the top of our component, but doing so would make it harder for a casual reader to identify the component's primary purpose.

An `<%init>` block means we can move our initialisation code to the *bottom* of our component. This means that when we open our file we immediately see what makes it interesting and unique. In fact, our component could look and feel just like a regular HTML file, except for the `<%init>` block down the bottom. This makes our components *much* more friendly to non-programmers who may need to edit or change the HTML.

Even though our `<%init>` block may be at the end of our component, it still acts as if it were at the top. Any variables declared in the `<%init>` block can be seen throughout the whole component. Here’s an example:

```
<p>
Hello <% $name %>!  Today is <% $weekday %>, the
<% $day %> of <% $month %>.
</p>

<%args>
$name => "anonymous"
</%args>

<%init>
use POSIX qw(strftime);
my @today   = localtime();
my $weekday = strftime("%A",@today); # Monday/Tuesday/...
my $day     = strftime("%e",@today); # 1..31
my $month   = strftime("%B",@today); # January/February/...
</%init>
```

Mason also has a `<%cleanup>` block, which is just like having a `<%perl>` block at the very *end* of your component. `<%cleanup>` blocks are much more rarely seen, since they don’t have the same stylistic advantages of moving code away from the top of your component.

It should also be noted that `<%cleanup>` blocks are *not* guaranteed to run before your component exits. If your component explicitly uses `return`, `die`, issues a redirect, or otherwise halts execution, then your `<%cleanup>` block may never get run.

%doc blocks

Documentation is important in any program, but it’s particularly important in large projects. It’s always possible to add documentation to a Mason program by using standard Perl comments:

```
<%perl>
# This is a comment!
my $price = 10.95;      # This is also a comment
</%perl>

<p>The price is $<% $price %>.</p>

% # This line is also a comment.
```

We could also use HTML comments:

```
<!-- This is an HTML comment -->
```

however those comments will be transmitted to the browser. While they can be useful to mark sections of code to ease in debugging, or mark authorship, they are *not* recommended for regular program comments. Your users may be able to read how your website works, and you’ll needlessly increase the amount of data that you need to transmit to clients.

Perl’s standard comments are great for short notes, but they rapidly become unwieldy when working with large amounts of documentation. To solve this, we can use a `<%doc>` block. Anything inside a `<%doc>` section is considered a comment, and is ignored by Mason:

```
<%doc>

Ice-cream flavour component.  More than 50 flavours!

Written by Paul Fenwick <pjf@perltraining.com.au>, August 2006

Usage: <& ice_cream.mhtml, flavour => "chocolate", scoops => 2 &>

</%doc>
```

If you're familiar with POD (see `perldoc perlpod`) then you can embed this into your `<%doc>` sections to have `perldoc` browsable documentation:

```
<%doc>

=head1 NAME

ice_cream.mhtml - Ice cream with more than 50 flavours!

=head1 SYNOPSIS

    <& ice_cream.mhtml, flavour => "chocolate", scoops => 2 &>

=head1 AUTHOR

Paul Fenwick <pjf@perltraining.com.au>, August 2006

=cut

</%doc>
```

Avoiding work with %once

Mason components are similar to subroutines. They run, do their work, and clean up afterwards. However this can be inefficient if we're calculating the same information every time, particularly if that information is expensive to compute.

A `<%once>` block evaluates when the component is first loaded, *before* any requests. The most common use of `<%once>` blocks is to create persistent lexical variables. Variables declared in a `<%once>` block are visible in the main component, as well as its subcomponents and methods and don't get cleaned up when the component has finished.

Code inside a `<%once>` block isn't run as part of a request. This means that you cannot use `$r` or `$m`. It also means that you should not do anything that cannot survive a `fork`; including connecting to a database or other application.

However we *can* use a `<%once>` block to prepare for a persistent database or other connection later on:

```
<%once>
# This creates our $dbh variable, but doesn't initialise it.
# However because it's declared in the %once block, it will
# continue to live between requests.

my $dbh;
</%once>
```

```
<%init>
use DBI;

# We may already have a database connection from a previous
# component call.  If we don’t have a connection, or it’s
# not responding, then make a new connection.

unless ($dbh and $dbh->ping) {
        $dbh = DBI->connect(...);
}
</%init>
```

Component internals: other named blocks

There are a number of other special Mason blocks. These are mentioned below.

%def

Used to create sub-components. These are like methods, but they can only be called from the containing component.

%filter

Used to filter the output of a component. We’ll cover this more in a later chapter.

%text

Used to provide output which is *not* parsed by Mason. This is typically used to provide Mason examples.

%shared

Variables created in the %shared block exist both in the component body and also in any subcomponents and methods. The contents of the %shared block run before the main component, its methods and subcomponents and may do initialisation. Unlike the %once block it runs once for each request.

Escaping content

As a programmer, generating HTML and URLs has a particularly annoying requirement. You have to make sure that everything is properly encoded in order for things to work. For example, in HTML a less than character (<) should be replaced with: `&lt;` and in a URL each space character is replaced with a plus (+).

Having to remember the encodings can be a real headache, and forgetting to encode can result in broken links, mis-formed HTML, or even the possibility of cross-site scripting attacks (see the chapter on security for more information). Fortunately, Mason takes much of the pain out of this by allowing you to escape your content as appropriate on it’s way out. Thus we can write:

```
<p>Hello <% $name |h %></p>

<p>Your personalised page can be found
<a href="http://example.com/user/<% $name |u %>">here</a></p>
```

The use of `|h` indicates that the result of our expression should be HTML escaped. The use of `|u` indicates it should be URL escaped.

It's possible to configure Mason to *always* apply (usually HTML) escaping to `<% %>` expressions. This provides a safety net when data *could* contain characters that need escaping. In these environments you can use `|n` to disable escapes. Specifying `|n` when no default escapes are used has no effect.

It's a good habit to always specify the escaping scheme that should be used for expressions. This ensures your output is *always* formatted the way you want, regardless of your operating environment. It also means you need to think about how each piece of data is used before you emit it, which will help avoid bugs and potential security flaws later on.

Exercises

1. Create variables containing: an email address, a url, the name of a company which includes some punctuation characters.
2. Interpolate these into your component output using appropriate escapes.

Escaping by default

In some cases you may find that you need to escape everything you generate before it sent out as output. You can achieve this by setting the configuration variable `default_escape_flags` to `h` or `u` as appropriate.

If the `default_escape_flags` variable is set and you wish to avoid escaping some given output, you can use the `|n` encoding flag, which applies no escaping. The `n` encoding can be combined with the other filter options so that `|nh` turns of any default encoding and turns on HTML encoding, while `|nu` does the same for URL encoding.

Creating your own escapes

Mason allows you to create your own escape filters. This is an advanced concept, but a very powerful one.

Let's say that your site regularly displays amounts of money, and you'd like to be able to round these with two decimal places ($2.00 instead of just $2, or $1.50 instead of $1.4999). We can use Perl's `sprintf` function to do this:

```
sprintf("$%.2f", $price);
```

however this is cumbersome to write frequently, and annoying if we want to change our currency formatting in the future, perhaps by adding commas ($1,234.00 vs $1234.00) or even performing conversion into the user's local currency.

This is an excellent example of when to use a custom escape. In our top-level autohandler we can set:

```
$m->interp->set_escape => (
        money => sub { sprintf("$%.2f",$_[0]) }
);
```

This defines a new `money` escape. Given the following snippet of code:

```
Price (incl GST): <% $price * 1.1 |money %>
```

this would produce:

```
Price (incl GST): $55.00                (when $price = 50)
Price (incl GST): $1.35                 (when $price = 1.23)
```

Whenever calling escapes other than the defaults, it’s necessary to separate them using commas. The following would apply both the `money` and `h` escapes, in that order:

```
<% $price |money,h %>
```

Modules vs components

Components are great for generating website specific things like navigation menus, headers, footers, search boxes, and weather displays. In fact anything that generates HTML is a perfect use of components.

However, there are a lot other things dynamic websites might do, that don’t immediately result in the generation of HTML. For example connecting to and interacting with a database, generating thumbnails for a photo gallery, sending email updates.

It can be tempting to put these tasks in components, to keep all of your code in one place, but is it wise? An alternative is to put all non-website specific code into separate modules. These are a little faster than Mason components, much easier to test, and can be reused outside of the web environment.

For example, consider the situation where your website provides various status reports from information stored in a database. The code for connecting to the database, and collating the information for the reports is not specifically website related. Having this code in a module will make it much easier when your manager asks whether its possible for you to also send out a weekly summary by email. If instead you had embedded the database interaction in your Mason components, you’d have to duplicate parts of it (which means you’ll have twice as much code to maintain), or move the code out to a module anyway.

A general rule of thumb is as follows:

> If the code generates HTML, then use it as a component.
>
> If the code does not generate HTML, but generates something that is only useful in a HTML context (for example a list of available style-sheets, or something which loads extra javascript to handle specific browser CSS bugs), then use it as a component.
>
> For everything else, put it in a module.

Returning a value from a component

The vast majority of Mason components generate HTML (or other output), as such it is fairly rare for a component to return a value. However, if you do want your component to return a value, or list of values, you can add an explicit `return` statement.

```
return $result;
```

In these cases, the `return` statement is usually added to the `<%init>` block.

Using cookies with Mason

Just as in our CGI code, there are reasons we might want to give users cookies from our Mason code. To do this, we use `CGI::Cookie` just as we have before:

```
<%init>
my %cookies = CGI::Cookie->fetch();

unless ( $cookie{preferences} )
        my $cookie = CGI::Cookie->new(
                -name  => 'preferences',
                -value => \%prefs,

        $r->headers_out->add('Set-cookie' => $cookie);
}
</%init>
```

Exercise

1. Add a cookie to one of your Mason components.
2. Use your CGI program from yesterday to check that you received the cookie.

Chapter summary

- Mason has two special global variables: `$r` and `$m`.
- `%init` blocks can be used to put initialisation code elsewhere than the top of a Mason component.
- `%doc` blocks can be used to store documentation.
- `%once` blocks allow you to perform an action once, when the component is first loaded, and then cache the result for future accesses.
- Mason provides a number of useful escaping mechanisms.
- Creating modules is often a better alternative to components for many non-HTML specific actions.

Chapter 14. dhandler - The default handler

In this chapter...

We’ve already discovered *autohandlers*, a powerful mechanism for manipulating content on our site. Mason comes with another mechanism for manipulating content that *doesn’t* exist on our site. This is the *default handler*, or `dhandler` as they are conventionally called in Mason.

The `dhandler` is invoked when Mason goes looking for content, but fails to find it. `dhandlers` are important because they have a chance to produce content, generate a redirect, or simply display a more useful *404 - Not found* page.

Finding dhandlers

In the same way that Mason will recursively search for `autohandlers` in the current directory and its parents, the same applies when trying to find a `dhandler`. A `dhandler` in the current directory takes precedence over a `dhandler` in the parent directory, and so on. If no `dhandler` can be found, then a standard `404 - Not found` response will be generated.

Arguments

Inside a `dhandler`, the original page requested can be found in `$m->dhandler_arg`. This is the full path, but excludes any leading slash. For example, a request for `/products/foo.html` would have `$m->dhandler_arg` return `products/foo.html`.

We can use our original page to generate a redirect, query a database, conditionally include components, perform a search, alert an operator, or do anything else that may be appropriate for the data concerned.

Not Found

There’s a good chance that sooner or later you’ll want a `dhandler` that really does generate a *Not Found* page. To do this, we need to explicitly set the status using `$r`, the apache request object.

```
% use Apache::Constants qw(NOT_FOUND);
% $r->content_type('text/html');
% $r->status(NOT_FOUND);
```

It’s critically important that the status be set when generating a *Not Found* page. This informs search engines and user agents that the content was not found. If the status is not set, then Mason will use a status of *200 Success*, and search engines and other services would try indexing content that simply doesn’t exist!

Any content in the `dhandler` is interpreted as text that should be sent to the browser. `dhandlers` will inherit from `autohandlers` in the same way as other components, so any *Not Found* page will have any navigation and other features provided to the rest of your site. This makes dhandlers a very useful way of handling missing content, as they can provide a useful page that remains up-to-date with the rest of the site.

Generating redirects

One potential use for `dhandlers` is generating redirects. This may be done because pages have moved, allow us to correct common mis-spellings, or handle users that haven't logged in.

The Mason interpreter has a method called `redirect` that allows current processing to be stopped, and for the browser to be redirected to a new page. While this can be used in any component, it is of particular use in `dhandlers`, where redirecting to moved content is a common task.

```
% $m->redirect($new_url);
```

By default, `redirect` will generate a "moved temporarily" (302 Found) status. This indicates that the user-agent should continue to use the original URL in the future. The redirect method can also be called with a second argument to generate a "moved permanently" (301) status if desired:

```
% use Apache::Constants qw(MOVED);
% $m->redirect($new_url, MOVED);
```

An example dhandler

The following is a simplified version of the dhandler used on the *Perl Training Australia* website. It checks for pages that are known to have moved and generates a redirect. If the page is not one known to have moved, then we instead generate a *Not Found* status instead.

```
<%init>
use Apache::Constants qw(NOT_FOUND MOVED);
my $request = $m->dhandler_arg;

# Some pages have moved permanently, and these are
# itemised in the table below.

my %new_location_of = (
        q{bookings.html}     => q{/bookings/},
        q{bookings/All.html} => q{/bookings/},
        q{courses.html}      => q{/courses/},
        q{books.html}        => q{/books/},
);

# If our request is for a known-moved page, then
# redirect immediately with a "moved permanently"
# status.  Calling redirect finishes our request.

if ($new_location_of{$request}) {
        $m->redirect($new_location_of{$request}, MOVED);
}

# Otherwise, our page is "Not Found".  We explicitly
# set our content type and status.  Our message will
# be automatically wrapped by our autohandler.

$r->content_type('text/html');
$r->status(NOT_FOUND);

<%init>
<p>
Sorry!  The page <tt>/<% $request %></tt> could not be found
on this server.  If you were expecting it to exist, then please
<a href="/contact.html">contact us</a> and let us know.
</p>
</%init>
```

Exercises

1. Try visiting a non-existent page in your Mason area. Observe what message you receive.
2. Create a `www/dhandler` file that contains a simple "page not found" message. Now what happens when you try to navigate to a page that does not exist?

Virtual pages

Another common use of dhandlers is to generate content on the fly. For example, suppose part of your website provides access to a mailing list archive.

You could write a program that records emails sent to the list and writes files appropriately, but that would require a lot of work, and potentially generate a large number of files that require management. If we want our files to contain links to follow-ups or other messages, then we may be re-writing files many times.

A standard CGI approach would be to store the emails in some standard format (perhaps a database) and then to provide the relevant information upon request. Requests may look like:

```
http://example.com/mailing-lists/archive.pl?year=2005&month=11&day=28
```

An alternative would be to use Mason's dhandlers. Your dhandler may work a lot like your CGI approach, but it comes with the added power of the Mason component's hierarchy as well as allowing you to use more meaningful URLs:

```
http://example.com/mailing-lists/archive/2005/Nov/28
```

Virtual quotations

A good use of a `dhandler` is displaying information from databases, stored files, or other non-HTML data. As a simple example, we're going to see how `dhandlers` can be used to access quotations.

The `fortune` Unix command displays a random and often amusing quotation. The fortune file format consists of a series of quotations, each separated by a percentage sign on a line by itself:

```
Although the Perl Slogan is There’s More Than One Way to Do It, I
hesitate to make 10 ways to do something.
        -- Larry Wall, 1990
%
And don’t tell me there isn’t one bit of difference between null and
space, because that’s exactly how much difference there is.
        -- Larry Wall, 1990
%
It should be illegal to yell ’Y2K’ in a crowded economy.
        -- Larry Wall, 1998
```

We'd like an application that allows us to visit a URL and receive a quotation:

```
/quotes/Larry_Wall/3
```

In this case we want the third quote by Larry Wall.

Exercises

1. The `www/quotes/dhandler` file contains some basic code to find and parse these quotation files. Modify it so that it extracts and displays the expected quote.
2. Create a new file in your `quotes` directory and add a few quotations from your favourite film, book, or even yourself. Verify that these quotes are now added to your site.

Caching pages

Sometimes creating our page can represent quite a bit of effort. Even with our simple quote example we had to search through the file for the quote that we're after. Unless our data needs to be regenerated every time, we can cache it.

When caching the result of a `dhandler`, we almost always want to key the cache to the requested page. If we didn't do this, then we'd serve the same (cached) page to any `dhandler` request.

Caching based upon the request URL is easy:

```
<%init>
return if $m->cache_self(
        key        => $m->dhandler_arg,
        expires_in => '10 mins'
);
</%init>
```

We talk more about caching in the next chapter.

`cache_self` only caches *content*, not the HTTP status. If your `dhandler` generates a redirect, file-not-found, or any status other than a simple *200 OK* then you should handle that before invoking any caching code.

It's a good idea not to even try to cache redirects (since they contain no content), and *404 Not Found* pages (as there is an infinite number of those), and only cache `dhandler` requests that result in real content.

Exercise

1. Update your `www/quotes/dhandler` to cache quotes for 30 seconds. Verify (by looking at the timestamp generated by the component) that quotes are being cached successfully.

Declining the request

Sometimes your `dhandler` may be called when it really would be more suitable for its parent or grandparent handler to handle the request. Perhaps your image-album `dhandler` has been called for an image that does not exist. We'd like to decline the request, allowing a parent `dhandler` to generate a *Not Found*, redirect, or other content as appropriate.

Declining a request is easy:

```
$m->decline();
```

A call to decline passes control up the component tree, until a dhandler is found which does not decline the request. If no dhandler will accept the request, an error is generated.

Any output generated by a dhandler while processing the request is discarded upon the call to decline. This simplifies the code for the dhandler as processing can start as normal until sufficient conditions occur to generate the decline.

Chapter summary

- dhandlers are called when Mason looks for content but fails to find it.
- dhandlers can be used to generate Not Found pages, generate redirects and also create content (virtual pages).
- Pages created by dhandlers can be cached for later accesses.
- dhandlers can decline requests, when this happens the next higher dhandler is called.

Chapter 15. Caching

In this chapter...

Writing a successful website is not just about having something that looks good and is functional. It needs to be scalable as well. Sometimes you'll find some of your components have expensive operations, such as downloading news items, or performing database queries, and having these run for every request is slow and inefficient. Luckily, Mason comes with a rich set of caching utilities out of the box.

General cache

Mason comes with a built-in cache mechanism. This allows us to store, retrieve, and expire values with a minimum of fuss. Caching is done on a *per-component* basis, so you need not worry about your keys conflicting with other keys on the system.

We can gain access to our cache by calling `$m->cache`. Let's say that we have a component that's part of an administrator login process, and as part of that process we report the last time the administrator logged in. Our code may look like this:

```
# Get our last login, or 'never' if we've never seen a
# login before.

my $last_login = $m->cache->get('last_login') || "never";

# Update our last login to now.
$m->cache->set('last_login', scalar localtime)
```

It's also possible to cache more complex data structures. This is done by passing a reference to the data structure as the second argument to `set`. For example, consider that we have a list of stock prices. Generating these takes a while so it's not something we want to do for every request.

First we attempt to get our prices out of the cache; if we succeed, we'll use those. If we don't, we'll generate new ones and use those instead.

```
<%perl>
my $stock_ref = $m->cache->get('stock');

if (not $stock_ref) {

        # This assumes that get_stock_prices has been
        # written elsewhere, or imported from a module.

        $stock_ref = get_stock_prices();
        $m->cache->set('stock', $stock_ref);

}
</%perl>

<p>Current stock prices:</p>
<ul>
% foreach my $symbol (keys %$stock_ref) {
        <li><% "$symbol -- $stock_ref->{$symbol}" |h %> </li>
% }
</ul>
```

Cache expiry

By default our cache keeps keys forever, however we may not always want that for all data. Some data, such as stock prices, are only valid for a certain length of time. We can set keys to automatically expire after a certain period of time by providing a *third* argument to `set`. For example:

```
# Stock prices are really only good for about 20 minutes
$m->cache->set('stock', $stock_ref, '20 minutes');
```

We can also expire values explicitly by using the `remove` call:

```
$m->cache->remove('stock');     # No stock values left
```

Exercises

1. The file `www/weather.html` predicts the weather with a low degree of accuracy. Modify it so that it caches the weather report for 30 seconds.
2. Check that your page now displays the cached weather when available, or generates and stores the weather when it's not available.

Caching pages

When creating a site, you may find that you want many of your pages to be rendered with Mason, but they only change rarely once they have been rendered. These pages are ideal candidates for Mason's simple page-caching strategy.

It's easy to mark that a page should be cached indefinitely:

```
<%init>
return if $m->cache_self();
</%init>
```

The `cache_self` call has one of two effects. If the page has not already been cached, then `cache_self` signals to Mason that a copy of the page should be saved after it's been generated. In this circumstance it returns a false value.

If `cache_self` already has a cached copy of the component, then it will return a true value, and send the cached copy to the browser or calling component. By checking for this we are able to return immediately without evaluating the rest of the component.

With no arguments, `cache_self` will cache our page indefinitely, until we explicitly clear it (more on this later). However it's also possible to cache our page for a specific period of time:

```
<%init>
return if $m->cache_self(expire_in => '10 mins');
</%init>
```

Of course, simple caches are great for simple things, but some things are more complex. Let's take an example of our component that displays the weather in a city. We don't want to fetch weather information for every request, and we need to maintain a separate cache for each city.

```
<%args>
$city
</%args>

<%init>
return if $m->cache_self(expire_in => ’2 hours’, key => $city);
</%init>
```

By using the `key` argument we are now able to tie our cache to a particular city, as well as ensuring that we only refresh our cache every two hours.

Busy locks

Let’s pretend that it takes ten seconds to update our weather report for a city, and that our previously cached copy has expired. What happens if a new request happens during this time? Since the cache is expired then it’s *also* going to try and update the weather report. That’s going to happen to *every request* until one (probably the first) finishes generating the new weather page and updates the cache. If we’ve been receiving five hits per second, and it takes ten seconds to generate the report, then that that’s *fifty* processes all trying to update the cache, when really we only needed a single one doing the work.

Clearly this situation is unacceptable, and Mason provides a solution. A *busy lock* allows us to specify that the first process to start updating the cache should be allowed to do so, but subsequent requests should continue to use the old value while the new one is being calculated:

```
<%init>
return if $m->cache_self(
        expire_in => ’2 hours’,
        key       => $city,
        busy_lock => ’30 secs’,
);
</%init>
```

Here we have specified a busy lock of 30 seconds. When our data expires, the *first* time we try to access it `cache_self` will return false, and at the same time extend the expiry time of our existing data by 30 seconds. This means that one component will work on recomputing the value, while the rest continue to use that old data. Once the new value has been computed, it’s used immediately.

The busy lock allows us to avoid needless recomputation, but also provides a safety net that if a particular process fails then another process will take over. If 30 seconds pass without an update, then the cycle repeats.

The time on a busy lock should be *at least* the longest time expected to recompute a given page. If too short a time is used then we may end up duplicating work, although too long a time can result in delay updates when recoverable errors occur.

Chapter summary

- Mason comes with a rich caching system out of the box.
- Each component has its own cache for storing data.
- Whole pages can also be cached.

- Busy locks can be placed on pages to allow only one process to update the page cache at once.

Chapter 16. Filters

In this chapter...

Mason makes it easy for a component's output to be inspected or changed before being sent to the browser. There are a great many uses for such a facility, including removal or addition of HTML comments, XHTML validation, HTML tidy-up, automatically filling in forms, censoring profanity, and highlighting search results.

%filter blocks

To filter a component we can use a `<%filter>` block. Inside the filter block the output of our component is stored inside `$_`. Our filter can inspect and change this content before it is passed to the next higher component in our chain (or to the browser, if we're in a top-level autohandler).

Component calls with content

Sometimes we don't want to filter the result of an entire component. Instead, we'd simply like to take a block of text and transform it in some way. The following example, adapted from the `HTML::Mason::Devel` documentation, demonstrates how we can do this for creating multi-lingual websites:

```
<&| /filters/i18n.mhtml &>
        <en>Good-day</en>
        <en-au>G'day</en>
        <fr>Bonjour</fr>
</&>
```

Our `i18n.mhtml` component could look like this:

```
% my $lang = $m->session->{language} || 'en';
% my ($text) = ( $m->content =~ m{<$lang>(.*?)</$lang>} );

<% $text %>
```

This checks our session to see if a preferred language has been selected, and if not it uses a default of English. It then simply finds the appropriate language block and displays it.

It should be noted that this component is not complete, as it doesn't deal with the situation where a particular language is not available. In that situation we may wish to choose a default language (such as English), and log an error that we don't have a translation for the text concerned.

Exercise

1. Write a filter block which changes all text to upper-case. Put this into `www/filters/uc.mhtml`.

   (Hint: This should only need to call `uc` on `$m->content` and then interpolate the result).

2. Use this filter block in one of your components.

Pre-filling forms in Mason

Mason has a special type of block called a `<%filter>` block. A filter allows us to make alterations to a component's results, and can be used for a variety of purposes. By using filters in combination with `HTML::FillInForm`, we can easily pre-fill our forms based on user submissions.

```
<%filter>
use HTML::FillInForm;
$_ = HTML::FillInForm->new->fill(
        scalarref => \$_,
        fdat      => \%ARGS,
);
</%filter>
```

The `HTML::FillInForm` module provides a number of useful options. Some of the more useful ones are detailed below:

target

The `target` option specifies that only the target form should be filled in. This is useful if your page contains multiple HTML forms, of which our data is intended for only a single form.

```
$_ = HTML::FillInForm->new->fill(
        scalarref => \$_,
        fdat      => \%ARGS,
        target    => "registration",
);
```

will only fill in fields inside a form named "registration":

```
<form name="registration"> ... </form>
```

fill_password

By default, password fields are also filled-in. To disable this, set the `fill_password` option to false:

```
$_ = HTML::FillInForm->new->fill(
        scalarref     => \$_,
        fdat          => \%ARGS,
        fill_password => 0,
);
```

Preventing password fields from being auto-filled removes the risk of the client's browser or proxy caching passwords and other sensitive information on disk.

ignore_fields

Rather than removing fields from the `%ARGS` hash to avoid auto-population, we can also specify them using the `ignore_fields` to `HTML::FillInForm`:

```
$_ = HTML::FillInForm->new->fill(
        scalarref     => \$_,
        fdat          => \%ARGS,
        ignore_fields => ['username','password'],
);
```

Exercises

1. The `www/login.html` page was used in the Sessions chapter. Upon submission it stores the given information into a session.

Change this code so that it uses `HTML::FillInForm` to present to the user any information already found in the session. Upon submission it should change the session information, and redisplay the content in the form.

Chapter summary

- Filter blocks can be used to change our output before it is sent to the browser.
- Filters can be used to select various language options for internationalisation.
- We can use `HTML::FillInForm` to pre-fill forms in Mason, just as we have done for `HTML::Template` forms.

Chapter 17. Session management

In this chapter...

Just as sessions are fundamental to the smooth working of CGI programs, they are essential to the smooth working of Mason programs too. In this chapter we take another look at sessions and consider how to use them with our Mason content.

Sessions in Mason

A default Mason installation does not come with any session capabilities, however they're easy to add in a relatively transparent way. Mason has a number of extensions available that live in the `MasonX::` namespace, and one of these is `MasonX::Request::WithApacheSession`.

The `MasonX::Request::WithApacheSession` extension uses the popular `Apache::Session` module to provide session management that is effectively transparent to the end developer.

The extension adds two new methods are added to the Mason request object:

```
$m->delete_session;     # Deletes the session from storage,
                        # also removes any cookies from browser.

$m->session;            # Provides access to the underlying
                        # Apache::Session hash.
```

The `$m->session` call makes available the special session hash, which can contain any information you like. This hash is automatically preserved by Mason, so anything placed in there by one request will be available in the next.

You should keep in mind that the *sessions* themselves are temporary, they last only for the length of the visitors browser session, or potentially even less. Anything placed in the session hash will eventually be cleared. When a user creates an account at your website it's perfectly reasonable to create a session that remembers their login, but their registration details should be placed into a more permanent store such as a database.

An example

Let's pretend that our website has a number of different stylesheets available to it, allowing the user a selection of look-and-feels. We'd like the user to be able to pick a style, and we'll use a session to remember this choice.

Firstly, let's see a component that allows style selections.

```
% my @styles = qw(default bigtext colourblind);

<form method="post" action="style_select.html">
<p>Select a style:</p>

<select name="style">
% foreach my $style (@styles) {
        <option value="<% $style %>"><% $style %></option>
% }
</select>
```

```
<input type="submit" value="Go" />
</p>
</form>
```

This simply creates a drop-down list which allows our user to select from one of three styles. Using this list submits data to `style_select.html`, which may look like the following:

```
<%once>
my %allowed_styles = map { $_ => $_ } qw(default bigtext colourblind);
</%once>

<%args>
$style    => "default"
</%args>

<%init>
my $session = $m->session;      # Obtain our session hash.

# Pick a style if allowed, or otherwise pick 'default'

$session->{style} = $allowed_style{$style} || 'default';
</%init>
```

Now that we have a component that allows us to set our style, we need our web-pages to make use of it. Luckily, Mason's autohandler means that implementing site-wide styles is a breeze:

```
<%init>
my $session = $m->session;
my $style   = $session->{style} || "default";
</%init>

<html>
<head>
<link rel="stylesheet" type="text/css"
      href="/styles/<% $style %>.css" />
</head>
<body>
% $m->call_next;
</body>
</html>
```

Now all of our pages will load the user-selected style-sheet. If no style-sheet has been selected, or if the user is not accepting cookies, then the `default` stylesheet will be used.

In our example we have listed our allowed styles in two different components. This may cause us headaches when we're expanding our website in the future, as we may update the style list in one location, but not the other.

A better solution would be to move the list of allowed styles into its own module, and then to `use` that module. This means that the allowed style lists can be kept in a single location. It also allows for the possibility of a dynamically generated style list.

Exercises

Your workspace is already configured to use `MasonX::WithApacheSession`.

1. The `www/login.html` page allows the user to enter some basic information about themselves (name and favourite colour). Write code that processes this information and stores it into the user's session.
2. The `www/profile.html` page is intended to display the user's profile. Update this code to read the information from the user's session and display it.
3. Our user may visit the `www/profile.html` page without having an active profile in their session. Determine a sensible course of action in this instance and implement it.

Session cleanup

Unlike `CGI::Session` which allows maximum expiry times, sessions made with `Apache::Session` will last forever. We can always clear our sessions explicitly:

```
$m->delete_session;
```

This will delete both the session from the store, and the cookie from the browser.

It's common practice to keep a timestamp on sessions, allowing us to determine the last time a session was accessed, and also ensuring that sessions are continually refreshed in the store. This could be placed in a top-level autohandler:

```
$m->session->{timestamp} = time;
```

It's commonplace for a periodically scheduled application to handle session clean-up. The most common implementations involve testing session files for their last modified date, or checking the last-modified timestamp on database rows.

Chapter summary

- Sessions are essential to keeping state between client requests.
- These are usually achieved by giving the user a session cookie.
- Mason has integration hooks with `Apache::Session`.
- `Apache::Session` does not allow us to expire sessions as easily as `CGI::Session` did, however we can achieve a similar effect by recording timestamps.

Chapter 18. Further Resources

Online Resources

- Embedding Perl in HTML with Mason - http://www.masonbook.com/
- Mason Headquarters - http://www.masonhq.com/
- Ovid's CGI course - http://users.easystreet.com/ovid/cgi_course/
- PerlNet - The Australian Perl Portal - http://perl.net.au/
- The Perl Directory - http://perl.org/
- Comprehensive Perl Archive Network - http://search.cpan.org/
- Perl Mongers user groups - http://pm.org/
- PerlMonks - http://perlmonks.org/
- O'Reilly's Perl.com - http://perl.com/

Books

Embedding Perl in HTML with Mason, Dave Rolsky Ken Williams, O'Reilly and Associates

Perl Best Practices, Damian Conway, O'Reilly and Associates

Programming Perl, Larry Wall et al, O'Reilly and Associates

Perl for System Administration, David N. Blank-Edelman, O'Reilly and Associates

The Perl Cookbook, Tom Christiansen and Nathan Torkington, O'Reilly and Associates

See Also

- Catalyst http://www.catalystframework.org/, a powerful and elegant Perl MVC framework (like Ruby on Rails, but for Perl!)
- Jifty http://jifty.org/view/HomePage, a web application framework with tools to support common operations.
- Template Toolkit http://www.template-toolkit.org/, a fast, powerful and extensible template processing system; fills many of the same needs as `HTML::Mason`.

Appendix A. Mason Setup and Administration

Introduction

In this chapter we will examine the configuration and setup of HTML::Mason under Apache 1.3 using mod_perl. This covers the most commonly encountered Mason installation environment.

You can learn more about Mason configuration by using `perldoc HTML::Mason::Admin`, or by reading the administrators manual online at http://www.masonhq.com/docs/manual/Admin.html. This topic is also covered on pages 102 - 112 in the Mason book.

Quick Setup

Setting up Mason under Apache is a simple process. The simplest configuration involves adding the following stanza to a mod_perl enabled Apache configuration file:

```
PerlModule HTML::Mason::ApacheHandler

<Location />
        SetHandler      perl-script
        PerlHandler     HTML::Mason::ApacheHandler
</Location>
```

This instructs Apache that all requests should be managed by the `HTML::Mason::ApacheHandler` module. This setup will work fine if everything in your directory is to be Mason-enabled, but that's rarely the case. The following configuration enables Mason for `.html`, and `.txt` files, and explicitly denies access to internal components.

```
PerlModule HTML::Mason::ApacheHandler
PerlModule Apache::Constants

<LocationMatch "\.(html|txt)$">
        SetHandler      perl-script
        PerlHandler     HTML::Mason::ApacheHandler
</Location>

<Locationmatch "\.(m(html|pl|txt)|dhandler|autohandler)$">
        SetHandler perl-script
        PerlInitHandler Apache::Constants::NOT_FOUND
</LocationMatch>
```

Mason with the lid off

On a more complex website, or on a server that serves many independent Mason-enabled sites, we need to do a little more work to have our system running smoothly. In order to understand the advanced configuration, we need to examine a few concepts first.

The Mason interpreter

At Mason’s heart is the *Mason interpreter*. At a very basic level the interpreter takes a component and executes it, however it also handles caching of compiled components on disk, the maintenance of in-memory caches, and making sure that all output and errors go to the right places.

Most site-specific configuration for Mason is set or passed to the Mason interpreter.

You can learn more about the Mason interpreter by reading `perldoc HTML::Mason::Interp` and pages 98 - 100 of the Mason book.

The component root

As described in earlier chapters, all Mason components must live inside a directory tree. The top of this directory tree is known as *the component root*.

It is possible to specify multiple component roots to Mason, which will be searched in the order specified. This allows for a directory of default or common components to be specified, and for new components to be written or existing ones to be overridden on a per-file basis.

To learn more about Mason component roots read pages 84-86 and 203-204 in the Mason book.

The data directory

Mason can make use of a directory in which to cache code and components, known as the *data directory*, or the `data_dir` in many configuration files. Mason doesn’t require a data directory, but it can speed up execution dramatically.

The data directory stores compiled Mason components, meaning that rather than having to interpret components every time they are used, this only needs to be done once per component. By default, Mason will check to see if the original component has been altered; if so, it will be recompiled and stored into the data directory.

Mason wrappers

The greatest level of control over how requests are handled is by writing our own wrapper around the Mason interpreter. When handling a request, Apache can be instructed to look inside a given namespace for a `handler` method. By writing our own package and placing a `handler` inside it, we can choose the code which is used to execute each request.

Here’s a simple wrapper which configures a Mason handler and passes requests directly to it:

```
package MyCompany::Mason;

use strict;
use HTML::Mason::ApacheHandler;

# First we create an ApacheHandler that will manage
# all requests.  This happens when Apache first starts,
# and is shared between all child processes.

my $ah = HTML::Mason::ApacheHandler->new(
        comp_root => '/path/to/comp_root',
        data_dir  => '/path/to/data_dir',
);

# Whenever we need to handle a request, we just pass it
# to the handler created above.

sub handler {
        my ($request) = @_;

        return $ah->handle_request($request);
}
```

The wrapper above does not provide us with any extra functionality, but it does provide us with a very useful starting point for more advanced handlers.

To use a handler as part of Apache, we load it using a `PerlRequire` statement:

```
PerlRequire /etc/apache/handler.pl
```

Once loaded, we can reference the package declared inside the handler as the target for our Mason requests. Note the use of `MyCompany::Mason` in the configuration below.

```
<LocationMatch "\.(html|css|txt)$">
        SetHandler      perl-script
        PerlHandler     MyCompany::Mason
</Location>

<Locationmatch "\.(m(html|pl|txt|css)|dhandler|autohandler)$">
        SetHandler perl-script
        PerlInitHandler Apache::Constants::NOT_FOUND
</LocationMatch>
```

One common configuration requirement is to have a number of virtual sites on a single server, all of which use Mason. A wrapper script is perfect for this, as we can check to which site is associated with a given request, and handle it appropriately.

```
package MyCompany::Mason;

use strict;
use HTML::Mason::ApacheHandler;
use Apache::Constants qw(DECLINED);

my %handler_for;

# Walk through each site, and establish an ApacheHandler for each.
# We use our site name to find the component root and establish
# our data directories.
```

```
foreach my $site (qw( www.perltraining.com.au www.example.com ) ) {
        $handler_for{$site} = HTML::Mason::ApacheHandler->new(
                comp_root => "/var/www/$site",
                data_dir  => "/var/cache/mason_data/$site"
        );
}

# Each request is passed to our handler, which finds the
# associated site and fires off the appropriate ApacheHandler
# created above.

sub handler {
        my ($request) = @_;

        # Query our request object for the server hostname.
        my $site = $request->server->server_hostname;

        my $handler = $handler_for{$site};

        # Decline sites that we don’t know how to handle.
        return DECLINED unless $handler;

        return $handler->handle_request($request);

}
```

Using Mason through CGI scripts

Mason provides the best performance when running in conjunction with Apache and mod_perl, however you may encounter situations where you can’t or don’t want to run Mason in a mod_perl environment.

Mason has a special module called `HTML::Mason::CGIHandler` that allows a regular Mason environment to be provided by a CGI script. A basic CGIHandler program looks like this:

```
#!/usr/bin/perl -w
# For this example, this script is ’mason_handler.cgi’

use strict;

# This simply creates a new CGIHandler object and passes
# the request to it.  It’s a lot slower than running under
# mod_perl, as we have to re-create the handler each time.

my $handler = HTML::Mason::CGIHandler->new(
        comp_root => ’/path/to/comp_root’,
        data_dir  => ’/path/to/data_dir’,
);

$handler->handle_request;
```

For this program to work, it still requires a bit of help from Apache; however it does not require mod_perl. The following configuration simply instructs Apache to pass all requests for `.html` files to our `mason_handler.cgi` program above.

```
<LocationMatch "\.html$">
        Action html-mason /cgi-bin/mason_handler.cgi
        AddHandler html-mason .html
</LocationMatch>
```

Using Mason in stand-alone scripts

Mason can be used independently of a CGI environment, or as part of a larger CGI application. To do this, we need to create our own *Mason Interpreter* to execute our components. This is particularly useful when building a testing framework for our components, as we can load and execute them independently of a web-server, and can tightly control their input and examine their output.

```
#!/usr/bin/perl -w
use strict;
use HTML::Mason;

my $output_buffer;

my $interp = HTML::Mason::Interp->new(
        comp_root  => '/path/to/comp_root',
        data_dir   => '/path/to/data_dir',
        out_method => \$output_buffer,
);

$interp->exec("my/component.html", @args_to_pass);
```

Our interpreter takes a number of arguments, but all of them are optional. Each argument is described below:

comp_root

This specifies our component root, where our Mason components can be found. If not specified, it defaults to the current working directory.

data_dir

This specifies Mason's data directory, where compiled components are cached. If not specified then caching is disabled, and components will be recompiled every time.

out_method

This specifies where component output should be sent. If not specified then output will go to `STDOUT`. If a reference to a scalar is provided then all output will be written to that scalar. This is perfect for testing, as the scalar can be examined to ensure correct output is being generated.

Conclusion

The most common Mason environment is under Apache. Setup is a simple process with most site-specific configuration being set in or passed to the Mason interpretor. We can control how requests are handled by writing a wrapper around the interpreter.