Web Applications With Maypole

Simon Cozens

Web Applications With Maypole

by Simon Cozens

Table of Contents

List of Figures

List of Tables

Chapter 1. Basic CRUD Sites

The Maypole web application framework can be used on two levels: first, as a simple way to add an interface to a database (and not much else besides), and second, as a toolkit for building more sophisticated web applications. In principle, there is a continuum of possible usages between these two levels, but it seems best for the purposes of teaching to entirely separate them.

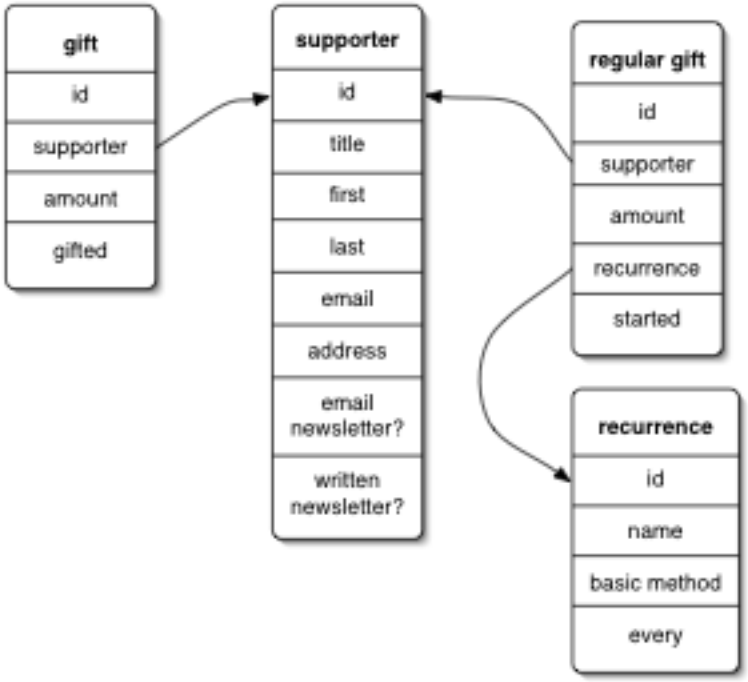
In the first part of this tutorial, we're going to quickly dispatch the first level of operation, putting a front-end onto a database. Not only is this useful in its own right in many cases, it both gives a demonstration of the power and flexibility of Maypole, and provides a useful basis for showing how Maypole applications can be expanded.

The Scenario

As many of you know, I'm not really a Perl programmer any more; instead, I'm a missionary. It's a bit of a strange job, since I don't actually get paid, but I get funded by various supporters. I have a very bad memory, and need to be reminded who my supporters are, how much they pay me, and how I need to get in touch with them to let them know what I'm doing and how their money is being used.

This sounds like the ideal job for a little relational database, and so I drew up the following schema:

Figure 1.1. Supporters database




```
CREATE TABLE supporter (
    id integer auto_increment primary key,
    title varchar(5),
    first varchar(30),
    last varchar(50),
```

```
    email varchar(255),
    address text,    email_newsletter integer(1),
    written_newsletter integer(1)
);
CREATE TABLE gift (
    id integer auto_increment primary key,
    supporter integer,
    amount decimal(6,2),
    gifted date
);

CREATE TABLE regular (
    id integer auto_increment primary key,
    supporter integer,
    amount decimal(6,2),
    recurrence integer,
    started date
);

CREATE table recurrence (
    id integer auto_increment primary key,
    name varchar(255),
    basic_method varchar(255),
    every integer
);
```

Now, to add and update supporters, what do I do? I don't really want to type in SQL all the time, so I wanted a web front-end to this database. This is where Maypole comes in.

Writing the "driver"

We're going to start with a very, very basic Maypole application, and then build it up to be a little more sophisticated. Here's the bare minimum we need:

```
package Supporters;
use Maypole::Application;
Supporters->setup("dbi:mysql:supporters");
Supporters->config->{uri_base} = "http://localhost/supporters/";
Supporters->config->{template_root} =
 "/home/simon/maypole-sites/supporters/templates/";

Supporters->config->{loader}->relationship($_) for (
    "a supporter has many gifts",
    "a supporter has many regulars",
    "a regular has a recurrence"
);
1
```

This should be reasonably self-evident: we say that we're a Maypole application, set up the database DSN we want to use, tell the application where we're going to live on the web, and where our templates are going to be, and then define some relationships: a supporter gives one-off gifts and regular gifts, and a regular gift has a recurrence relation to tell us how regular it is.

The actual version I use has a few more niceties to make it work more smoothly, but we'll come to those in time.

Configuring Apache

Now we have the Perl module which is going to drive our Maypole application, we next need to tell Apache about it. First if we haven't installed this driver module and it's not sitting in Perl's `@INC`, then we need to tell Perl where to find it:

```
<Perl>
use lib qw(
          /home/simon/maypole-sites/supporters/
        );
</Perl>
```

And next we have to associate the driver with a location:

```
<Location /supporters>
    PerlHandler Supporters
    SetHandler perl-script
</Location>
```

Almost finally, we need to put Maypole's factory templates where we said we would:

```
% cp -r ~/maypole/templates/factory /home/simon/maypole-sites/supporters/templates
```

And finally, we restart Apache. Now on going to the relevant URL, we should have an application up and running...

What we get

Those few lines of code have actually got us a long way. Here we can add and edit rows in the database, view relationships between various rows, and delete entries. Each table has a set of pages: `view`, `list`, `edit`, `delete` and so on.

For the supporters database project, I didn't actually need anything else - I just needed the ability to put rows into the database and see what was already there. For quick applications like this, Maypole is ideal. But Maypole comes into its own when you start to add "actions" to the application - for instance, if I wanted to have a page which totalled all donations from different sources, then I'd have to start writing some code. We'll see how that works in the next few chapters.

Behind the scenes

Warning

This is an advanced section which we're not going to go into during the tutorial, but which explains, for your benefit when rereading these notes afterwards, how all of this works. This should help you to learn about the continuum of Maypole applications, and how to go from simple application like this to the more complex application we see in the second part of the tutorial - that means this isn't going mean very much on the first reading. I'd encourage you to read all of the tutorial notes through before coming back to this; it'll make much more sense that way.

On the other hand, if you feel you understand everything so far and you're bored during the tutorial and want something to read, feel free to dig into this more challenging material.

So far we've just told Maypole about a database and its relationships, and it's come up with a reasonably useful web application all on its own. How?

There are three factors here: first, database introspection. Maypole's view class provides a lot of metadata about the selected model class to the templates. The `classmetadata` template variable is a hash, which contains the following elements:

Table 1.1. The `classmetadata` hash

Name	Description
`name`	The Perl class name of the model class
`table`	The name of the database table
`columns`	A list of the columns we're allowed to display on this table
`colnames`	A mapping between the column names and the same names formatted better for display
`related_accessors`	Method names of the accessors of any has-a relationships
`moniker` / `plural_moniker`	A "display name" for this model class. (e.g. "supporter" and "supporters")
`cgi`	A hash of HTML elements to be used in constructing forms for this table

From this set of metadata, you can construct a set of templates which do the right thing for pretty much any table. For instance, this template snippet:

```
[% FOR col = classmetadata.columns %]
<p> [% classmetadata.colnames.$col %] : [% object.$col %] </p>
[% END %]
```

will list all the columns of a database row along with the value for this particular object.

This brings us to the second phase in the CRUD strategy: Maypole comes with a sufficiently generic set of templates (called the "factory templates") for the CRUD operations we've been using so far; it uses the class metadata to produce pages which look right for whatever shape of database you throw at it.

To finish off the process, there are default actions in each Model base class (for instance, `Maypole::Model::CDBI` for the default, `Class::DBI`-based applications) which use the CRUD methods of the underlying data representation to perform the appropriate edit, delete, or whatever actions. So actually, there are a lot of actions going on - both templates and code - but they're squirreled away inside the default model class.

Exercises

Exercise 1. So far we've got a web application which can handle relationships between tables, and add, display, update, search for and list records. Come up with three examples of a web application that sounds like it would be a complicated problem, but can actually be done as a simple Maypole application like the one we've just created, only with (a) changes to the way the data is displayed and (b) some sort of access control. One free example is a product catalogue for an online business.

Exercise 2. Come up with three examples of a web application which can't be done with these simple ac-

tions, and analyse what additional functionality would be required to make it work. Remember these, because we're going to use them later in the tutorial.

Chapter 2. Maypole basics

Now we've seen what Maypole is capable of, let's now take a bit more detailed look at how it actually works. Here we're going to discuss the theory of operating a Model-View-Controller process like Maypole, but we'll also try to tie it into some practical examples as well.

Important concepts

There are at least two really important concepts that are central to Maypole's operation, and, annoyingly, they're both common words used in a jargony way. So it's important to know what we're talking about when we refer to a *Maypole request* and a *Maypole action*.

The view of Maypole from a thousand miles looks like this:

Figure 2.1. Maypole workflow (from space)

That is, we start with something from the user - a request for a page, typically, with maybe some form parameters. Basically, nothing much. This hits our Maypole application as a request, and we spend a little bit of time fleshing it out until it becomes a big fat object which has enough information to respond to the request, at which point it becomes something small again (a page).

"Fleshing it out" typically involves working out what we need to do to produce that page, pulling the data out of the database, munging it appropriately, finding the appropriate template to display the data, collecting together the variables which have to go into that template, and so on. Let's take a closer look at this workflow.

Maypole request

A Maypole request is analogous to an Apache mod_perl request; it contains all the information that Maypole gathers from the environment and the user in order to do something and spit out a page. The funky thing about Maypole requests, unlike mod_perl requests, is that your application is a specialisation of the `Maypole` class; the Maypole request will actually be an object in the `Supporters` class or whatever your application is called. This seems strange, but it turns out to be really useful later.

We'll see soon what data the request object actually gathers, and what happens when the request is pro-

cessed, but for now you need to know that it's essentially an encapsulation of your application and the user's request for a web page.

Maypole action

Generally when you're requesting a page from a Maypole application, you want something to happen. At the very least, you're going to want some text to spit back as a web page; (this is the "View" of the MVC web application paradigm) additionally, you're going to want to do something with your data - either to load up some data to view it, to edit a record in the database, and to delete one, or some other activity. (This is the "Model" part. Loosely.)

An example

We access the "supporter/list" page of our supporters application. "supporter" is the database table, and "list" is the action. The job of the model part is to grab a bunch of database rows from the database. This happens like so: Maypole turns the *table* name "supporter" into the *class* name `Supporters::Supporter`, since this is the model class which governs the "supporter" table. (We'll see in the CDBI chapter why this is.) `Supporters::Supporter` inherits from one of the default Maypole model classes, which provides a "list" method. So Maypole calls `Supporters::Supporter->list` to grab the database rows, which are modelled as `Supporters::Supporter` objects.

These objects are associated with the Maypole request object. The view part then puts the objects it has gathered into a template variable, and processes that template to produce a page containing the details of the database rows.

Therefore the response to a Maypole request breaks down as the execution of a method call on a model class, and the selection and processing of a template in a view class. Together, these two stages are known as an *action*. A useful thing to know is that the view part is generally good enough for almost everything you need to do with Maypole - typically you're just processing templates, and customization of this part of the action is done by writing better templates. You don't need to write Perl code for this bit, just templates.

Exception to general rule...

We've just said that the second part of an action requires writing templates. Of course, there are times when this isn't true, and it doesn't need to be true. So when, for instance, you're pulling a picture out of the database, you don't want to put the JPEG data through a template, you want to just spit it out to the browser. Don't worry, you can do that - the template step is optional - and we'll see how to do it later.

On the other hand, you will need to write methods, because I can't tell in advance everything that your Maypole application needs to do! If you want a page which processes a credit card request, you'll need to write a method which does the credit card processing. That's the bad news. The good news is that Maypole does everything else for you.

So "writing an action" in Maypole-speak typically means writing a method in a model class, and writing a template. Since writing a template is to be expected, sometimes an "action" just refers to the method.

Note

Key point: in particular, an action is a method in a model class which is marked by the `:Exported` attribute. The purpose of this attribute is to prevent a user in front of a web browser from being able to call any method at all on your model classes! Actions are therefore declared like so:

```
sub something :Exported {
   my ($class, $r) = @_;
   $r->objects([ $class->get_the_appropriate_objects ]);
}
```

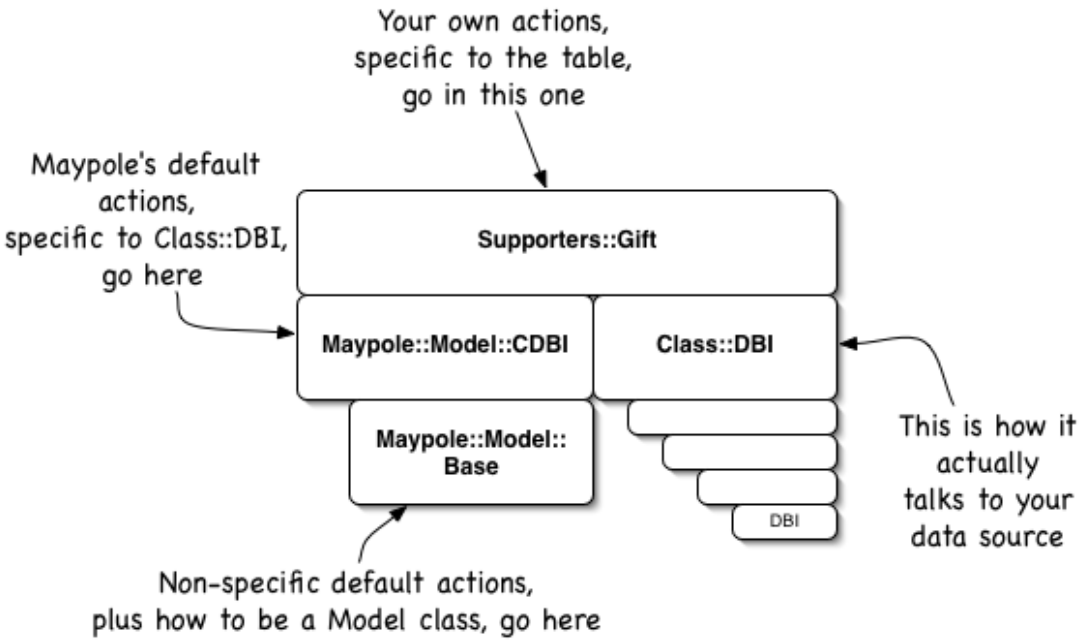
where `$r` is the Maypole request object.

The scary stuff with the model class

This is probably the messiest and hardest to understand part of Maypole's operation. Once you get it, everything becomes clear.

The concept of a "model class" in Maypole is a multi-layered one. This is because Maypole wants to provide you with somewhere you can place your own actions specific to each table, but of course it wants to do a lot of the work for you. First, it wants to provide default actions for things like listing, editing and viewing rows; second, it wants a basis for talking to the database table in the first place - something like `Class::DBI`. So your model class actually looks something like this:

Figure 2.2. Model class inheritance



Warning

Maypole causes the per-table model classes to inherit from `Maypole::Model::CDBI` at runtime; this can occasionally cause a gotcha if you're writing your own actions. For instance:

```
package Supporters;
```

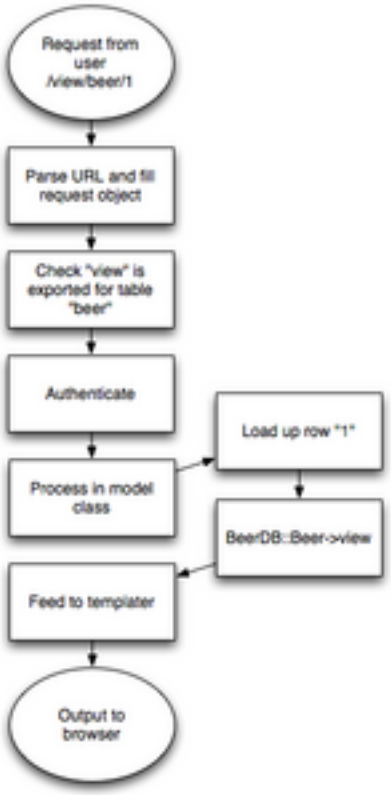
```
use Maypole::Application;
Supporters->setup("...");
# ...
package Supporters::Supporter;

sub total :Exported {
    # Calculate the total given by this person,
    # and display it
}
```

This won't work. This is because `Supporters::Supporter` doesn't know anything about the `:Exported` attribute until `Supporters->setup` has run, and so when Perl tries to compile the attribute, it dies horribly. Putting `BEGIN { }` around the call to `setup` is one way to take care of this.

The Maypole workflow

Now we're in a position to have a closer look at the Maypole workflow. Here's a diagram which adds more details to the "view from space" we saw at the beginning of the chapter:

Figure 2.3. Maypole workflow



Another view of this process can be found in the `Maypole::Workflow` manual page which comes with Maypole.

Note

The first stage, receiving the request and parsing the URL parameters, is actually carried out by a front-end module. If you're using `Maypole::Application` (as you probably should) it will arrange for your application to inherit from either `Apache::MVC` (for mod_perl scenarios) or from `Maypole::CGI` (for CGI use) rather than from `Maypole` directly.

The Maypole request object

Once the Maypole process of dealing with a request is kicked off (generally by Apache calling `YourApp->handler`, which Maypole provides for you) a request object is instantiated. As mentioned previously, this is actually an object of `YourApp`'s class, and it is generally referred to, in Apache mod_perl style, as `$r`. The object contains the following slots, which get filled in by the handler:

Table 2.1. `$r`

`config`	The configuration object for this application
`view_object`	An object used to view the data; typically a `Template` object
`ar`	The `Apache::Request` object for this request, if applicable
`path`	The full path requested, as returned from Apache or from the CGI environment; let us assume this is `/supporter/view/4`. It would then be further split up into:
`table`	The database table to act on, in this case `supporter`.
`action`	The requested action: `view`
`args`	Any further arguments in the path - in this case, `4`, which will later be turned into an object representing the Supporter with ID 4 in the database.
`params`	Any CGI form parameters
`model_class`	The table-specific model class in use; ie, `Supporters::Supporter`
`template`	The name of the template file to be processed. In the example above, this would be `view` unless the `view` method decided to change it.
`objects`	Populated with the objects to be acted on; for instance, the `4` argument would be converted to a `Supporters::Supporter` object here. The list action would populate this with multiple objects.
`template_args`	Any user-supplied arguments to be handed on to the template; this is populated either in the action or in the `additional_data` method. (Or both, of course)
`content_type`	Used to mark the content type to be returned to the browser.
`output`	Eventually filled in by the view class, unless the action wants to intervene, this is the data that gets sent back to the browser.

Template Selection

Maypole searches for templates in three different places: first, it looks for a template specific to a class; then it looks for a custom template for the whole application; finally, it looks in the `factory` directory to use the totally generic, do-the-right-thing template.

The basic application we saw in the first chapter used factory templates to achieve everything; serious Maypole applications will generally want to use their own templates for everything.

Therefore underneath whatever you specify to be the template root for your Maypole application, we expect at least three subdirectories; if you're using factory templates, you should copy all the templates which ship with Maypole into a subdirectory called `factory`, as we did in the first chapter. Second, if you're customizing some of those templates, or want a common area to place, for instance, macro files, there should be a `custom` subdirectory. Finally, for each table there should be a subdirectory containing the templates you want to use. For instance, if you want to customize the way `supporter/view` looks, you place your own `view` template in the `supporter` subdirectory. It's as simple as that.

Here's the code in Maypole which makes it all happen:

```
sub paths {
    my ( $self, $r ) = @_;
    my $root = $r->config->template_root || $r->get_template_root;
    return (
      $root,
      (
        $r->model_class
         && File::Spec->catdir( $root, $r->model_class->moniker )
      ),
      File::Spec->catdir( $root, "custom" ),
      File::Spec->catdir( $root, "factory" )
    );
}
```

Configuration

Maypole gives each application a hash in which it can configure itself; in our application, that's accessed through `Supporters->config`. Here are the various slots in the configuration hash:

Table 2.2. Maypole configuration hash

dsn	The DBI DSN provided to `setup`.
template_root	The path underneath which the template files can be found.
uri_base	The application's root on the web site; that is, the base URL of this application
model	The model class for this application - typically either `Maypole::Model::CDBI` or `Maypole::Model::CDBI::Plain`.
tables / classes	The tables in the database we can play with, and the associated model classes; these are typically set up by the call to `setup` - the tables and classes are determined automatically in the `Maypole::Model::CDBI` case, and manually supplied to `setup` in the case of

	`::CDBI::Plain`
loader	In the case of `Maypole::Model::CDBI`, the `Class::DBI::Loader` (see next chapter) object is placed here
rows_per_page	When `Class::DBI::Pager` (again, see next chapter) is used, the number of data rows to be displayed on a single page
Anything else	The config hash is to be used as a generic place for the application or any plugins it uses to specify their configuration details, so all kinds of things may appear in other slots.

Review of workflow

Let's once again trace the path of a request through Maypole.

Once a request is made for, say, `/supporter/edit/20`, Maypole's handler first calls out to a front end class to get the environment and parse the request; it knows that the table should be `supporter`, the action and the template `edit` and the first argument `20`.

Next the table is looked up and converted to a class name, `Supporters::Supporter`. We check to see if the `edit` method is available; if not, we just process the template.

If it is, we call the `authenticate` method to ensure we can do this; if you haven't overriden this method it lets every request through.

What happens next depends on the Maypole model class. For the default classes, the first argument (20) is turned into a `Supporters::Supporter` object representing that row in the table, and placed in the `objects` array in the request object.

Then the `Supporters::Supporter->edit` method is called. This may do anything, including changing the template, rewriting the template arguments, supplying its own output, but in the general case, it will just perform some action and prepare the objects in the Maypole request object.

The generic hook `additional_data` is called to allow the user a final chance to fiddle with each request. Typically this is where additional template arguments are added.

Finally, the request object is fed to the view class, which handles filling out the template, and the results are sent back to the front-end for delivery to the user's browser.

This seemingly very specific process is actually the general backbone behind most web applications; since Maypole does it all for you, it frees you up to only write the code that is specific to your particular application.

Chapter 3. `Class::DBI` primer

Now we know the theory, we can start to look at how Maypole application communicate with the data source. Typical Maypole applications use the `Class::DBI` library to do this, although theoretically Maypole is designed to allow other object relational mapping libraries to serve as model classes; in fact, an Alzabo base class has been written. However, just because Maypole has the flexibility, this doesn't mean it has to be used, and effectively, Maypole is coupled to `Class::DBI` and the Template Toolkit.

Basic operation

`Class::DBI` is an object relational mapping class, and we will assume you understand the basics of what that means: it means we're going from database rows to Perl objects and vice versa.

To set it up, we subclass `Class::DBI` to create a "driver" class specific to our database:

```
package CD::DBI;
use base 'Class::DBI';
__PACKAGE__->connection("dbi:mysql:musicdb");
```

Next we set up classes for each of the tables we're interested in, subclassing from our driver and setting up the table name and column list:

```
package CD::Artist;
use base 'CD::DBI';
__PACKAGE__->table("artist");
__PACKAGE__->columns(All => qw/artistid name popularity/);
```

CRUD

`Class::DBI` adds a few more methods to the `CD::Artist` class to help us search for and retrieve database rows:

```
my $waits = CD::Artist->search(name => "Tom Waits")->first;
print $waits->artistid; # 859
print $waits->popularity; # 634

my $previous = CD::Artist->retrieve(858);
print $previous->name; # Tom Petty and the Heartbreakers

# So how many Toms are there?
my $toms = CD::Artist->search_like(name => "Tom %")->count;
print $toms; # 6

for my $artist ( CD::Artist->retrieve_all ) {
    print $artist->name, ": ", $artist->popularity, "\n";
}
```

We can also create a new artist by passing in a hash reference of attributes:

```
$buff = CD::Artist->create({
   name => "Buffalo Springfield",
```

```
   popularity => 10
});
```

Class::DBI automatically creates data accessors for each of the columns of the table; we can also update columns in the database by passing arguments to the accessors:

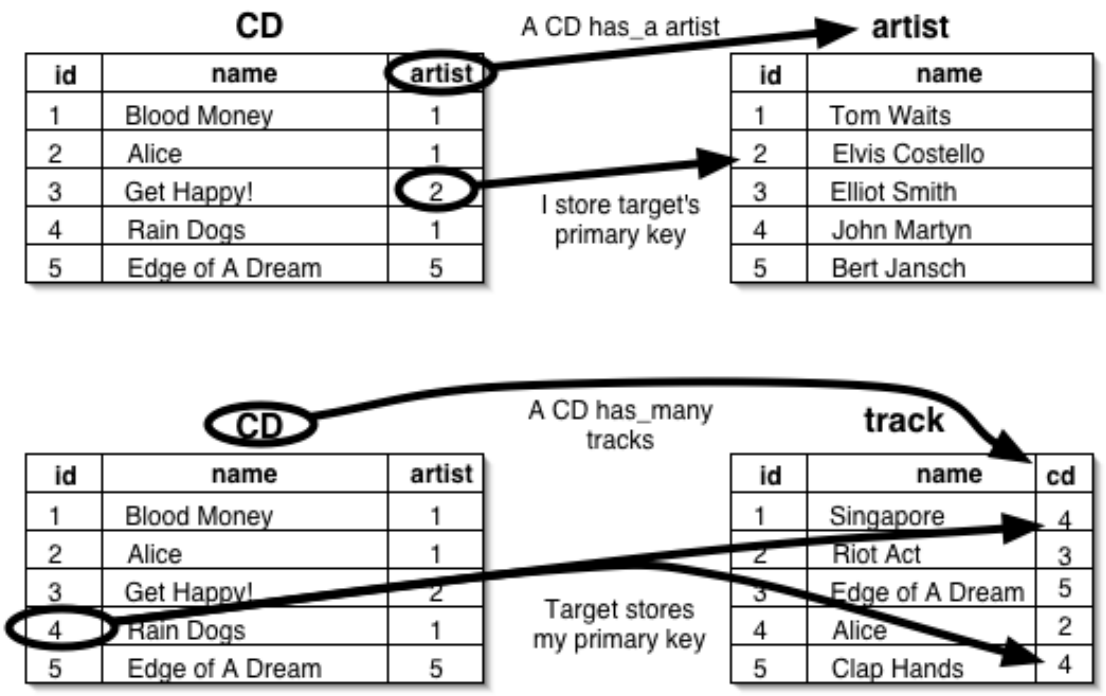
```
print $buff->name;
$buff->popularity(20);
```

Relationships

Class::DBI supports several types of database relationship. The two most common are has_a and has_many. It also allows you to use or write plug-in modules to declare other relationship types.

The following diagram illustrates the difference between has_a and has_many:

Figure 3.1. has_a versus has_many



We've already seen the use of a has_a relationship, between CDs and artists - each CD has_a artist. We've also already written some code to implement a nice Perlish interface to it: when we ask a CD object for its artist, it takes the artist primary key, finds the row in the artist table with that ID, and returns the appropriate object. However, in Class::DBI, instead of writing our own accessor, we just declare the relationship:

```
CD->has_a(artist => "CD::Artist");
CD::Track->has_a(song => "CD::Song");
# ...
```

The nice thing about this is that we can also declare relationships to classes which are not Class::DBI based, but which follow the same general pattern: find the column in the database, do something to it, and turn it into an object. For instance, the publishdate column needs to be turned into a Time::Piece object:

```
CD->has_a(publishdate => 'Time::Piece',
              inflate => sub { Time::Piece->strptime(shift, "%Y-%m-%d") },
              deflate => 'ymd',
          );
```

As before, we relate a column to a class, but we also specify a subroutine which goes from the data in the database to an object, and a method to go the other way, to serialize the object back into the database.

has_many relationships are also easy to set up; instead of writing the tracks accessor as we did before, we ask Class::DBI to do it for us:

```
CD->has_many(tracks => "CD::Track");
```

Now, for instance, to dump all the tracks in the database, we can say:

```
for my $cd (CD->retrieve_all) {
    print "CD: ".$cd->title."\n";
    print "Artist: ".$cd->artist->name."\n";
    for my $track ($cd->tracks) {
        print "\t".$track->song->name."\n";
    }
    print "\n\n";
}
```

Maypole's default list action uses retrieve_all to stuff the objects slot full of all the records in a particular table.

Question

Can you see a problem with that? Can you guess how we might fix it?

Plugins

One of the nice things about Class::DBI is that it's flexible and extensible - flexible because it's very easy to play with its internals (at least once you're comfortable with the tangled web of modules that makes it all work) and extensible because you can pull in plug-in modules to extend its functionality. One of those modules is going to solve the problem that we presented at the end of the last section.

CDBI::mysql

Class::DBI often wants me to set up things by hand that the computer should be able to do for me. For instance, I feel I shouldn't have to specify the columns in the table. Thankfully there are numerous database specific extensions for Class::DBI on CPAN which know how to interrograte the database for this information:

```
package CD::DBI;
use base 'Class::DBI::mysql';
__PACKAGE__->connection("dbi:mysql:musicdb");
__PACKAGE__->autoupdate(1);

package CD::Artist;
use base 'CD::DBI';
__PACKAGE__->set_up_table("artist");
```

This uses the `mysql` extension to query the database for the columns in the table.

CDBI::Loader

Just as in the same way that `Class::DBI::mysql` asked the database for its rows, you can set up all your classes at once by asking the database for its tables as well. The `Class::DBI::Loader` module does just this:

```
my $loader = Class::DBI::Loader->new(
    dsn => "dbd:mysql:music",
    namespace => "MusicDB"
);
```

With our database, this will set up classes called `MusicDB::CD`, `MusicDB::Artist`, and so on. All we need to do is set up the relationships between the classes.

CDBI::Loader::Relationship

For very simple relationships, `Class::DBI::Loader::Relationship` can help set these up as well:

```
$loader->relationship("a cd has an artist");
$loader->relationship("a cd has tracks");
# ...
```

Since this is exactly the sort of thing that helps setting up quick Maypole applications, Maypole avails itself of both `Class::DBI::Loader` and `Class::DBI::Loader::Relationship`, putting a `loader` slot in the config object:

```
Supporters->config->{loader}->relationship($_) for (
    "a supporter has many gifts", "a supporter has many regulars",
    "a regular has a recurrence"
);
```

CDBI::Pager

Finally, we want to solve the problem of the `list` method returning everything in our database. `Class::DBI::Pager` is another plugin which enables us to restrict the number of items returned on each retrieval::

```
use CD;
```

```
package CD;
use Class::DBI::Pager;
use constant ITEMS_PER_PAGE => 20;
use CGI;
my $page = param("page") || 1;
my $pager = CD->page(ITEMS_PER_PAGE, $page);
my @cds = $pager->retrieve_all;
```

Class::DBI::Pager is a mix-in for Class::DBI-based classes which allows you to ask for a particular page of data, given the number of items of data on a page and the page number you want. Calling page returns a Data::Page object which knows the first page, the last page, which items are on this page, and so on, and can be used in our template for navigation:

```
[% IF pager.previous_page %]
<A HREF="?page=[%pager.previous_page%]"> Previous page </A> |
[% END %]
Page [% pager.current_page %]
[% IF pager.next_page %]
| <A HREF="?page=[%pager.next_page%]"> Next page </A>
[% END%]
```

This is exactly how the default list templates work, and you are encouraged to copy this style for your own lists. Maypole sets aside the config variable rows_per_page to pass into Class::DBI::Pager.

Chapter 4. Template Toolkit Primer

Once we have the data we want, the next stage is to display it, and this is the job of the view class. Again, Maypole is not necessarily tied down to one particular view class, and classses have been written to allow the use of systems like `HTML::Mason` for templating Maypole applications. You can certainly go that way if you want to, but as with `Class::DBI`, Maypole has been most successfully used with the Template Toolkit. In this section, therefore, we'll present an introduction to how the Template Toolkit is used in general, and also specifically within Maypole.

Basic templating

The most basic uses of Template Toolkit use it just like other templating systems: to fill scalars into a form. Variable names are enclosed in `[% ... %]` pairs, like so:

```
[% today %]

[% title %] [% forename %] [% surname %]
[% address %]

Dear [% title %] [% surname %],
   Thank you for your letter dated [% their_date %]. This is to
confirm that we have received it and will respond with a more
detailed response as soon as possible. In the mean time, we
enclose more details of ...
```

Notice, however, that our variables inside the Toolkit `[%` and `%]` delimiters aren't Perl variables with the usual type sign in front of them; instead, they're now Template Toolkit variables. Template Toolkit variables can be more than just simple scalars, though; complex data structures and even Perl objects are available to Template Toolkit through a simple, consistent syntax. For instance, we could say:

```
[% today %]

[% name.title %] [% name.forename %] [% name.surname %]
```

The dot operator is equivalent to Perl's `->` - it dereferences array and hash reference elements, and can also be used to call methods on objects, so here `name` could be a hash, or it could be an object, but so long as it has those accessors which give appropriate answers, we don't need to care. If we know they are objects, then of course we can give parameters to the method calls:

```
[% name.salutation("Dear %s,") %]
```

If our variables are arrays underneath, we can use `FOREACH` to loop over their elements. Here's an example from a newsletter I produce with TT:

```
    <h2>In brief</h2>
    <ul>
    [% FOREACH point = brief %]
    <li>[% point %]</li>
    [% END %]
    </ul>
```

As you can see, the syntax is inspired by Perl - we can `foreach` over a list and use a local variable `point` to represent each element of the iterator.

Similarly, there's also the `IF`/`ELSIF`/`ELSE` block:

```
[% IF delinquent %]
   Our records indicate that this is the second issuing of this
invoice. Please pay IMMEDIATELY.
[% ELSE %]
   Payment terms: <30 days.
[% END %]
```

Includes, Macros, Plugins, Filters

If that were all that Template Toolkit would do, it would be on a par with other templating systems. But there's far more.

Includes

Includes allow us to split out the template into more managable components, placed in individual files; for instance, Maypole's factory templates use static header and footer files. Here's the header:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
        <title>[% config.application_name || "A poorly configured
Maypole application" %]</title>
    <meta http-equiv="Content-Language" content="en" />
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"
/>
    <link title="Maypole Default" href="/maypole.css" type="text/css"
rel="stylesheet" />
    <script type="text/javascript">
</script>
</head>
<body>
    <div class="content">
```

This is placed inside a file called `header`, and so the factory templates begin with:

```
[% INCLUDE header %]
```

So if you want to supply your own look and feel to the default CRUD site, you could do so by creating `custom/header`, which will override the default factory header. As you can see, included files receive the template variables as well, such as `config` in our example.

Macros

Additionally, you can define macros to simplify repeated pieces of coding. For instance, we've seen that Maypole URLs are dissected into table, action, and arguments; the factory macros include some code to construct a link by putting these back together again:

```
[%
MACRO link(table, command, additional, label) BLOCK;
    '<A HREF="' _ base _ "/" _ table _ "/" _ command _ "/" _
          additional _ '">';
    label;
    "</A>";
END;
%]
```

Notice here the use of Perl 6 style concatenation operators.

Plugins

Of slightly less common use, but astonishing versatility, is the Template Toolkit plugin system. This provides the ability for templates to call out to Perl code. For instance, the amazingly helpful (but somewhat dangerous) `Template::Plugin::Class` allows you to call class methods from within TT:

```
[% USE tagcloud_c = Class("HTML::TagCloud");
SET tagcloud = tagcloud_c.new;
FOREACH tag = tags;
   tagcloud.add(...);
%]
```

Filters

Template Toolkit filters are a little like Unix filters - they're little routines which take an input, transform it, and spit it back out again. And just like Unix filters, they're connected to our template output with a pipe symbol. (`|`)

For instance, the oddly named `format` filter performs `printf`-like formatting on its input:

```
[% job.description | format("%60s") %] : [% job.cost %]
```

We can also filter whole blocks of template content. For example, if we wanted to format the output as HTML, we could apply the `html_entity` filter to replace entities with their HTML encoding:

```
[% FILTER html_entity %]
Payment terms: < 30 days.
[% END %]
```

This turns into:

```
Payment terms: &lt; 30 days.
```

Other interesting filters include the `upper`, `lower`, `ucfirst` and `lcfirst` filters to change the casing of the text; `uri` to URI-escape any special characters; `eval` to treat the text to another level of Template processing, and `perl_eval` to treat the output as Perl, eval *that*, and then add the output to

the template. For a fuller list of filters with examples, see the `Template::Manual::Filters` documentation.

TT within Maypole

As with any view class, TT receives certain template variables from Maypole:

Table 4.1. Template variables

request	The Maypole request object
objects	The objects to be displayed
base	The base URL
config	The `Maypole::Config` object
classmetadata	The class metadata as described in Table 1.1, “The `classmetadata` hash”
Anything else	The contents of `$r->template_args` is added to the set of template variables at this point.

Additionally, Maypole makes an alias for the objects it passes in; for instance, in `/supporters/list/` the objects can be referred to as `supporters` as well as `objects`. If there's only one item in the `objects` array, it's given a singular name such as `supporter`. This makes the templates much easier to read.

The factory templates are written to use these template variables, and particularly the `classmetadata`, to provide a generic CRUD interface. When you are writing your own templates, you can use as many or as few elements of the factory templates as you like, but you will find that writing your own templates is actually considerably simpler.

Chapter 5. Developing web applications with Maypole

This chapter is designed to draw the main points out of the live coding example that goes on on the stage. Because it's live coding, I don't know exactly how it's going to go, but I do know some of the points I want to make to explain what goes through my head when I'm writing a Maypole application. Therefore this chapter is going to be a mixed bag - both a collection of random notes, hints and tips, and Maypole programming best practices, and also a printout for your reference of useful code snippets used in the demonstration application.

Maypole best practices

When I started writing Maypole I had no idea how to write good Maypole applications. When I had written about ten or fifteen applications, I noticed a series of patterns of development, and wrote them up in the Maypole manual. Now I've written another load of applications, the patterns have changed again. Essentially, I'm learning the best practices as I go along. That doesn't mean that Maypole is unstable in any way; it just means that the best methodology for creating complex applications in a simple and maintainable way can only be learnt and developed through experience, and I want to share the current state of my experience with you.

Start from scratch

This is the one tip that I wish I'd discovered initially. Failing to understand this has been the cause of big problems in many people's understanding of how Maypole operates. If you don't get this, you won't understand how powerful Maypole is, and you'll conclude that it's only for "toy" applications which do little more than the CRUD example.

The factory templates, the relationship model, even the `Class::DBI::Loader`, are attractive ways to start writing an application, because they do most of the work for you. However, as with all these things, there is a tradeoff between convenience and flexibility, and if you're writing a production application, rather than a quick development spike, you want to veer much more toward the flexibility end of the spectrum. You will be developing your own actions, using custom templates for each action, and doing a lot more than the standard actions do.

So to psychologically cut yourself free from the CRUD application we developed in the first chapter, be prepared to start writing your application from scratch. Of course it won't be *entirely* from scratch, because you'll still be using Maypole and all the tools it gives you to make writing a web application easier, but you will benefit from the following advice:

- *Don't use the factory templates*. The factory templates are useful if you want to prototype an application, and particularly to prototype your schema, and then make minor tweaks to the CRUD design. However, when you're writing your own full-sized application, you will find it easier to write templates from scratch.

  One reason for this is that, as you are passed in objects by a friendlier name, and you know all the accessors of a particular object, individualised templates are much simpler than generic ones. For instance, the factory "view" template is 90 lines long; to do the same job for "supporters" would look like this:

```
[% PROCESS macros %]
[% INCLUDE header %]
<h2>[% supporter.name %]</h2>
```

```
<table class="view">
    <tr><td class="field">First Name</td><td>[% supporter.first %]</td></tr>
    <tr><td class="field">Last Name</td><td>[% supporter.last %]</td></tr>
    <tr><td class="field">Title</td><td>[% supporter.title %]</td></tr>
    <tr><td class="field">Email</td><td>[% supporter.email %]</td></tr>
    <tr><td class="field">Address</td><td>[% supporter.address %]</td></tr>
    <tr>
        <td class="field">Email newsletter?</td>
        <td>[% supporter.email_newletter %]</td>
    </tr>
    <tr>
        <td class="field">Written newsletter?</td>
        <td>[% supporter.written_newletter %]</td>
    </tr>
</table>

<h2> Gifts </h2>
<ul>
[% FOR gift = supporters.gift %]
   <li> [% gift.amount %]: [% gift.gifted.ymd %] </li>
[% END %]
</ul>
```

Guess which is easier to customize and maintain?

- *Use plain `Class::DBI`*. By using ordinary `Class::DBI` instead of `Class::DBI::Loader`, you can put each class, its relationships and its actions into a separate module file; again, it just makes the code easier to organise.
- *Be prepared to override built-in actions*. For the same reason as overriding built-in templates, actions have to be sufficiently generic to do the right thing in all circumstances, but then they can't really be customized beyond that, and they're generally more complicated than one written from scratch would be.

Exception to general rule...

Of course, saying that, I've just looked at a recent production application and found it uses one driver file using `Class::DBI::Loader` with a bunch of actions in it, and only overrides one of the built-in actions. In many cases, the default actions are good enough - that's why they're the default - but on the whole, it's best to override.

Authentication

Most applications need authentication, and the standard answer to the problem in Maypole is `Maypole::Plugin::Authentication::UserSessionCookie`. This issues cookies to the user, and on presentation of a username and password credentials which gets looked up in a particular class, a `user` slot is added to the request object. If you have a table called `user` which has `user` and `password` columns, then the authentication process is simple:

```
use Maypole::Application qw(Authentication::UserSessionCookie);
use Maypole::Constants;
sub authenticate {
    my ($self, $r) = @_;
    $r->get_user;
    if (!$r->user and $r->action =~ /^(delete|edit|do_edit|....)$/) {
```

```
        $r->template("login"); # Setting template stops action happening
    }
    return OK;
}
```

If you don't have such columns, then it's a bit more complicated, but only a little; see the documentation to `M::P::A::USC`.

Other random tips

Here are some other collected best practices which will come out during the demonstration.

Static pages

We don't want static pages, like the CSS, to be processed through the whole Maypole system, so we can use the authentication process to get rid of them early. Put them all in a directory called `static`, and then:

```
sub authenticate {
   my ($r) = @_;
   return DECLINED if $self->path =~ /static/;
   $r->get_user;
   # Do authentication stuff here.
   return OK;
}
```

Uploading

Uploading files used to be difficult, but it got easier with the release of `Maypole::Plugin::Upload`. This adds the `upload` method to the Maypole request object, which returns a hash with filename, content, MIME type and so on:

```
sub do_upload :Export {
    my ($self, $r) = @_;
    my $photo = $self->create({
        uploader => $r->user, # We know who the user is
        uploaded => Time::Piece->new(),
        title => $r->params->{title}
    });

    die "Can't write ".$photo->path("file")." because $!"
        unless open OUT, ">". $photo->path("file"))
    print OUT $upload{content};
    close OUT;
    # ...
}
```

Additional data

We've already mentioned that the `additional_data` method is a good thing to override to insert additional template variables or do any other post-processing:

```
use Time::HiRes qw(gettimeofday tv_interval);
sub authenticate {
```

```
    my $r = shift;
    $r->{template_args}{started} = [gettimeofday];
}

sub additional_data {
    my $r = shift;
    $r->{template_args}{time_processing_model} =
        tv_interval($r->{template_args}{started}, [gettimeofday];
}
```

Methods on request object

Sometimes there are things you want to do in your template that require calls out to Perl, and you're not sure how to do this. For instance, in our application we will build up a couple of lists on the right hand side - a tag cloud, and a list of recently uploaded photographs. Both are implemented by putting methods into the `Memories` class which return the appropriate data:

```
package Memories;
sub tagcloud {
    my $cloud = HTML::TagCloud->new();
    # fill $cloud
    $cloud;
}
```

Because we pass in the request object `$r` to the templates as `request`, this method can be accessed from the template as `[% request.tagcloud %]`

Listings

For your reference, here are various code listings which accompany the demonstration application.

SQL Schema

The photo sharing application has the following database schema.

```
CREATE TABLE photo (
    id integer not null auto_increment primary key,
    title varchar(255),
    uploader integer,
    uploaded datetime
);

CREATE TABLE comment (
    id integer not null auto_increment primary key,
    name varchar(255),
    content text,
    photo integer
);

CREATE TABLE user (
    id integer not null auto_increment primary key,
    name varchar(255),
    password varchar(255)
);

CREATE TABLE tag (
```

```
    id integer not null auto_increment primary key,
    name varchar(255)
);

CREATE TABLE tagging (
    id integer not null auto_increment primary key,
    tag integer,
    photo integer
);
```

CSS

And here is the CSS style sheet for the application:

```
BODY { background: #994; border: 0; margin: 0; padding: 0 }
a { text-decoration: none }
a:hover { text-decoration: underline }
a img { border: 0 }
#nav a { color: #222 }
#nav { background: #fff; padding: 2px; font-weight: bold; font-family:
sans-serif; font-size: 80%; text-align: center}
#rhs { float:right; width: 200px; background: #ddd; height: 100%;
padding: 0 10px 0 10px; border-left: 2px solid black }

#login { margin-top: 10px;
font-weight: bold; font-family: sans-serif; background: #eee}
.loginfield { font-weight: bold; font-size: 80%; }

#main { border: 1px solid black; background: #eee; padding: 5px; margin: 0 0 0 10p

h1 { font-family: sans-serif; }
p { font-family: sans-serif; }

#recentuploads { background: #aaa; border: 1px solid black; width: 400px }
.thumb {
    width: 100%;
    text-align:center; font-size: 80%; font-family:sans-serif;
    border-bottom: 1px solid black; }

.photoview {
    text-align:center;
    }
.userlist {
    padding: 5px;
    text-align:center; font-weight:bold; font-family:sans-serif;
    }
.userlist td { background: #bbb; padding: 10px; }
.comment {
    background: #ccc; padding: 10px;
    font-family:sans-serif;
    margin: 5px 50px 5px 50px
}
.info { font-weight: normal; font-size: 70% }

#tabmenu { border-bottom:2px solid black; margin: 12px 0 0 0;
    padding: 0; z-index:1; padding-left:10px;
}

#tabmenu li { display:inline; oveflow:hidden; list-style-type: none; }

#tabmenu a, a.active {
```

```
    background: #ffa;
    color: #000;
    font-size: 10pt;
    font-weight:bold; font-family:sans-serif;
    margin: 0; padding: 2px 5px 0px 5px;
    border-right: 1px solid #cc7;
    border-top: 1px solid #cc7;
    text-decoration:none;
}

#tabmenu a.active {
    background: #dd8; border-bottom:3px solid #dd8;
    border-left: 1px solid black;
}

#tabmenu a:hover { color: #fff; background: #ac9; text-decoration:none;}
#tabmenu a.active:hover { color: #000; background: #dd8; }

#content { background: #dd8; border:2px solid black; border-top: none;
    z-index: 2;margin:0; padding:20px }

.exiftag { font-size: 8pt; background: #ffa; }
.exifvalue { font-size: 9pt; background: #fff }
```