Perl 5

Web development can be fun again

A Mojolicious Book

Brian Medley

Contents

License

Rationale

1.1 Mojolicious

Thus begins our quest for Mojo - a tutorial approach to learning web technologies. We will be learning Mojolicious [http://mojolicious.org]; at the time of this writing it is a "next generation web framework for the Perl Programming language". The next generation feature set is:

- `Full stack HTTP and WebSocket client/server`
- `IPv6, TLS, SNI, IDNA, Comet`
- `Non-blocking I/O and embeddable web server`
- `JSON and HTML/XML parse with CSS selectors`
- `Perlish Templates`
- `Sessions`
- `Cookie management`
- `HTTP / WebSocket`
- `Routes`
- `CGI / PSGI auto-detection`
- `Static files`
- `Testing framework`
- `Plugins`

Chapter 2 will focus on introducing Mojolicious while chapter 3 has several apps for perusal. A photo app, blogging app, and a real-time chat app. Chapter 4 has several practical examples.

Preparation

Perl

Mojolicious requires Perl. These days we separate a system perl with a user installed Perl. The user installed is necessary so that we have free reign over what is installed, configured, and can allow for easier debugging.

We will be using Perl on OS X with 5.20.1. Your system Perl will be used to bootstrap our install with Perl-Build and App::cpanminus. Perl-Build will download, configure, and install Perl, while App::cpanminus allows for easier installation of modules.

```
$ curl -L -n -O https://raw.githubusercontent.com/tokuhirom/Perl-Build/master/perl-build
$ perl perl-build 5.20.1 /opt/perl-5.20.1
```

```
Fetching 5.20.1 ...
Downloaded http://.../SHAY/perl-5.20.1.tar.bz2 to .../perl-5.20.1.tar.bz2
Configuring perl '5.20.1'
...
```

```
$ curl -L http://cpanmin.us | /opt/perl-5.20.1/perl - App::cpanminus
```

```
--> Working on App::cpanminus
Fetching http://www.cpan.org/authors/id/M/MI/MIYAGAWA/App-cpanminus-1.7014.tar.gz ... OK
Configuring App-cpanminus-1.7014 ... OK
...
Building and testing App-cpanminus-1.7014 ... OK
Successfully installed App-cpanminus-1.7014
3 distributions installed
```

After doing these things it will be easier to install Mojolicious and any required modules. Also, there is now a fully functional Perl install that can be tinkered with to your heart's content. To verify the install run:

```
$ /opt/perl-5.20.1/bin/perl -v
```

```
This is perl 5, version 20, subversion 1 (v5.20.1) built for darwin-2level

Copyright 1987-2014, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl". If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
```

For convenience, a symlink can be added so that /opt/perl points to /opt/perl/5.20.1/bin/perl.

```
$ ln -s /opt/perl-5.20.1/bin/perl /opt/perl
```

Installation

Adding the Mojolicious CPAN module to the system installs everything required for developing and running a web app.

```
$ /opt/perl-5.20.1/bin/cpanm Mojolicious
```

```
...
Mojolicious will be installed by the cpanm utility.
```

Hello World In Text

Now that we have Mojolicious installed, we can start writing code for our website. Our first example is a full program and you are not expected to understand everything. It is geared to get your feet wet. After this, we will have a tutorial introduction in HTTP, HTML, Javascript, CSS, and other Web technologies. The first example is a Mojolicious::Lite application. It is a real-time micro web framework. This micro web framework can have the entire web structure in a single file.

Several key features that enable the real-time aspects of the framework are websockets, RESTful routes, non-blocking I/O web server, and long polling.

The framework is predominately written by Sebastian Riedel.

Now, lets turn our attention onto some code. In the code below, line 1 turns your Perl script into a full featured web application that uses strict, warnings, and utf8. It's fun.

Listing 1.1: Hello World - ex1_1.pl

```
1 use Mojolicious::Lite;
2
3 get '/' => {text => 'Mojolicious is awesome!'};
4
5 app->start;
```

Run the program like so:

```
$ morbo -v hello_world_text.pl
```

```
["Mon Sep 3 19:08:38 2012] [info] Listening at "http://*:3000".
Server available at http://127.0.0.1:3000."
```

We now have a full featured web server running on port 3000. You may view in any browser. In your URL bar put: http://127.0.0.1:3000 and view the page.

Hello World In HTML

Next, modify the contents of hello_world.pl to look like this:

Listing 1.2: HTML Hello - ex1_2.pl

```
1  use Mojolicious::Lite;
2
3  get '/' => sub {
4      my $self = shift;
5
6      $self->render("index");
7  };
8
9  app->start;
10
11 __DATA__
12
13 @@ index.html.ep
14
15 <!DOCTYPE html>
16 <html>
17 <head>
18     <title>Hello World</title>
19 </head>
20 <body>
21     Hello world<br> %# Edit here
22 </body>
23 </html>
```

Our morbo server should automatically restart and load the new file contents. The code at line 6 displays the inlined page below.

Next, add "Carpe diem" after line 21. Save the file and reload the page in the browser. The morbo server will restart your server once a change is detected.

The other way to build an app is with a well-structured web application. This is where the business logic and application set up are put in several files, as opposed to just one. A full app is created with the "generate app" command, and we will discuss this further with the Photo app.

We are going to be using lite apps going forward until the photo album is encountered later on.

Going Forward

2.1 Preamble

Here we get down with Mojo. A brief introduction of HTTP 1.1 is given and then we look at some specific ways that Mojolicious creates a framework around web technologies to make things easier for development. Specifically, we will look at what follows:

- `Routes`
- `Logging`
- `Placeholders`
- `Query Parameters`
- `Templates and Stash`
- `Sessions`
- `Forms`

After that our attention will be turned to apps and examples.

2.2 HTTP 1.1

Under the Hood

As you know, web pages are loaded from a server. When you go to domain.com, then the default HTML is loaded using HTTP. As you will see, the transfer, or request, from the web browser client (aka user agent) to the server can be considered a file transfer. For example, when http://127.0.0.1:3000 was visited something like the following was sent from your browser to the server (the request):

```
GET / HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_5) AppleWebKit/537.36 (KHT
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
```

Then, the server responded with (the reply):

```
HTTP/1.1 200 OK
Content-Length: 2
Server: Mojolicious (Perl)
Connection: keep-alive
Date: Wed, 30 Oct 2013 23:44:04 GMT
Content-Type: text/html;charset=UTF-8
```

If we were to go to http://127.0.0.1:3000/hello, then the GET request would start with:

```
GET /hello HTTP 1.1
```

Of note, is that when we access http://127.0.0.1:3000 there is an implied "/" and when http://127.0.0.1:3000/hello there is a specific "/hello". These are the pages that the server is being requested to serve.

Also, the structure of the request and response should be noted. The lines that are Word: or Word-Dash-Word: are called headers. These are metadata that describe your request and response.

2.3 Routes

Finding our way

Routes enable Mojolicious to easily glue together an incoming request with code.

For example, if we pointed our browser at http://127.0.0.1:3000, then the following request would be sent to the server:

```
GET / HTTP 1.1
```

And the route below would be used.

Listing 2.1: Index Route

```
1 get '/' => sub {
2     my $self = shift;
3
4     $self->render(text => "Hello world");
5 };
```

The **get '/'** will redirect the **HTTP GET /** request to the anonymous subroutine shown. It should be noted that a GET and POST can be redirected to different subroutines; and we will examine POST later in the chapter. This is a very powerful construct that allows us to execute arbitrary business logic (system commands, SQL, control flow logic, etc) for a given request.

Also, $self is a Mojolicious::Controller object. This contains both the HTTP request and HTTP response.

2.4 Logging

From whence we came

At times it is appropriate to log data while processing code in the backend. In development mode a file in **log/**development.log is used and in production mode a file **log/**production.log is utilized. The modes can be switched around via *MOJO_MODE* environment variable. Merely creating a log directory in the app's home directory will enable the output into these files.

```
$ mkdir log
```

Now we can send logging data to the proper file with code such as:

Listing 2.2: ex2_2.pl

```
1  use Mojolicious::Lite;
2
3  get '/:name' => {name => 'Default'} => sub {
4      my $self = shift;
5
6      my $name = $self->param("name");
7
8      $self->render(text => "Hello world: $name");
9  };
10
11 app->start;
```

In addition, usage statistics for each request are logged to the development.log file. For example, the below is from the route snippet above.

```
   [Mon Sep 3 22:07:35 2012] [info] Listening at "http://*:3000".
   [Mon Sep 3 22:07:38 2012] [debug] Your secret passphrase needs to be changed!!!
   [Mon Sep 3 22:07:38 2012] [debug] GET /carpe (Mozilla/5.0 (X11; Linux x86\_64)
      AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.89 Safari/537.1).
   [Mon Sep 3 22:07:38 2012] [debug] Routing to a callback.
   [Mon Sep 3 22:07:38 2012] [debug] {
     'now' => 'Mon Sep 3 22:07:38 2012'
   }
```

2.5 Placeholders

What's in a name...

Input and output are intrinsic to any computer program, perhaps doubly so to a web application. A significant source of user input comes from the URL that the user typed in. These are placeholders and GET/POST parameters.

They are embedded in the the URLs, which makes things easy for copying and pasting with friends. We will be discussing placeholders in this section.

A placeholder is embedded in the URL and parsed by Mojolicious. They are available via the ->param method.

As a specific example, the following code can be used to demonstrate a GET request with an optional placeholder. The parameter is a name.

Listing 2.3: ex2_2.pl

```
1  use Mojolicious::Lite;
2
3  get '/:name' => {name => 'Default'} => sub {
4      my $self = shift;
5
6      my $name = $self->param("name");
7
8      $self->render(text => "Hello world: $name");
9  };
10
11 app->start;
```

The application usage is straightforward: visit 'http://127.0.0.1:3000/' and you get a default name; visit 'http://127.0.0.1:3000/Ben' and you get the name 'Ben'.

The above paragraph is detailed below via curl.

```
$ curl -v http://127.0.0.1:3000/
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 20
< Server: Mojolicious (Perl)
< Date: Fri, 29 Apr 2016 21:21:12 GMT
< Content-Type: text/html;charset=UTF-8
<
* Connection #0 to host 127.0.0.1 left intact
Hello world: Default
$ curl -v http://127.0.0.1:3000/Ben
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET /Ben HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 16
< Server: Mojolicious (Perl)
< Date: Fri, 29 Apr 2016 21:21:16 GMT
< Content-Type: text/html;charset=UTF-8
<
* Connection #0 to host 127.0.0.1 left intact
Hello world: Ben
```

Of note is that the default GET request looks exactly similar with the one in the prior section. However, when we add /Ben onto the URL we get "GET /Ben".

2.6 Parameters

Variety is the spice...

This section deals with GET and POST parameters. These parameters present themselves through the request in the query string. This will be made clearer in the curl output below; however, something like "GET /?age=25" is presented to the server when the user types (or clicks on a link that has) http://127.0.0.1:3000/?age=25

As a specific example, the following code can be used to demonstrate a GET request with a placeholder and GET parameter. Both are optional.

Listing 2.4: ex2_3.pl

```
1  use Mojolicious::Lite;
2
3  get '/:name' => {name => 'Default'} => sub {
4      my $self = shift;
5
6      my $name = $self->param("name");
7      my $age = $self->param("age") // 20;
8
9      $self->render(text => "Hello world: $name and $age years old.");
10 };
11
12 app->start;
```

The application usage is straightforward: visit 'http://127.0.0.1:3000/' and you get a default name; visit 'http://127.0.0.1:3000/Ben?age=30' and you get the name 'Hello world: Ben and 30 years old.'.

The above paragraph is detailed below via curl.

```
$ curl -v http://127.0.0.1:3000/
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/html;charset=UTF-8
< Server: Mojolicious (Perl)
< Content-Length: 38
< Date: Fri, 29 Apr 2016 21:55:57 GMT
<
* Connection #0 to host 127.0.0.1 left intact
Hello world: Default and 20 years old.
$ curl -v http://127.0.0.1:3000/Ben?age=30
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET /Ben?age=30 HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/html;charset=UTF-8
< Date: Fri, 29 Apr 2016 21:56:01 GMT
< Content-Length: 34
< Server: Mojolicious (Perl)
<
* Connection #0 to host 127.0.0.1 left intact
Hello world: Ben and 30 years old.
```

Again, the default GET request looks exactly similar with the one in the prior section. However, when we add /Ben?age=30 onto the URL we get "GET /Ben?age=30".

Multiple paramters can be added with a &. The first requires a ? and then the rest are separated by &.

For example http://127.0.0.1:3000/Ben?age=30&developer=1.

2.7 Templates and Stash

How did that get there?

Templates are the bread and butter of dynamic content generation. They take input from the controller and load that into a templating system to create content. The content is usually a web page with HTML, CSS, and Javascript. However, there are other possibilities, as well, such as text, XML, and pdf.

The input from the controller is placed in a "stash" data structure. There are a few ways to set up this data structure. One approach is to use the "stash" controller method.

Listing 2.5: ex2_4.pl

```
1  use Mojolicious::Lite;
2
3  get '/:name' => {name => ''} => sub {
4      my $c = shift;
5
6      my $name = $c->param("name");
7
8      $c->stash(name => $name);
9
10     $c->render("slash");
11 };
12
13 app->start;
14
15 __DATA__
16
17 @@ slash.html.ep
18
19 % if (stash('name')) {
20     You are <%= stash('name') %>
21 % } else {
22     Please pass in a name to the url like so '<%= url_for('/Ben')->to_abs %>'.
23 % }
```

For example, in line 8 of ex2_4.pl we set the "name" stash parameter which will be available in the templates. Specifically, at line 19 the "stash" helper is used to dynamically adjust the template output.

2.8 Sessions

Lets stick around

A session is mechanism that uses cookies for saving state between different requests and responses. The cookie is a special header in the browser that is saved by the browser between requests and responses in a cookie jar. Technically, the session is defined as:

```
Persistent data storage for the next few requests, all session data gets
serialized with Mojo::JSON and stored Base64 encoded in HMAC-SHA1 signed
cookies, to prevent tampering. Note that cookies usually have a 4096 byte (4KB)
limit, depending on browser.
```

In the code below, when we visit http://127.0.0.1:3000 and reload several times, then the browser will remember the counter at line 6. Then, the user will be shown how many times they have visted.

Listing 2.6: ex2_5.pl

```
1  use Mojolicious::Lite;
2
3  get '/' => sub {
4      my $self = shift;
5
6      ++$self->session->{count};
7
8      $self->render("slash");
9  };
10
11 app->start;
12
13 __DATA__
14
15 @@ slash.html.ep
16
17 % if (1 == session("count")) {
18     You have visted once.
19 % } else {
20     You have visted <%= session("count") %> times.
21 % }
```

Lets examine what happens in the actual request and response. What we're going to do is use curl and show the headers that are interchanged via the user agent and the server.

Our initial request is here. The last line shows that this is the first time this page has been visted. Please note that there the Set-Cookie header in the response is parsed by curl and saved into the file "jar".

```
$ curl -v --cookie jar --cookie-jar jar http://127.0.0.1:3000/
> GET / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 29 Apr 2016 01:59:07 GMT
* Added cookie mojolicious="eyJleHBpcmVzIjoxNDYxODk4NzQ3LCJjb3VudCI6MX0---72e64d66da00d8ca1f
    80ee4f8384f0957612188d" for domain 127.0.0.1, path /, expire 1461898747
< Set-Cookie: mojolicious=eyJleHBpcmVzIjoxNDYxODk4NzQ3LCJjb3VudCI6MX0---72e64d66da00d8ca1f80
    ee4f8384f0957612188d; expires=Fri, 29 Apr 2016 02:59:07 GMT; path=/; HttpOnly
< Server: Mojolicious (Perl)
< Content-Type: text/html;charset=UTF-8
< Content-Length: 26
<
    You have visted once.
```

The next time we run curl we'll send the Cookie header in the request so that the server can retrieve the value(s) that were saved earlier. This cookie is then made available in the session object for the route.

```
$ curl -v --cookie jar --cookie-jar jar http://127.0.0.1:3000/
> GET / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Cookie: mojolicious=eyJleHBpcmVzIjoxNDYxODk4NzQ3LCJjb3VudCI6MX0---72e64d66da00d8ca1f80ee4f
    8384f0957612188d
>
< HTTP/1.1 200 OK
< Server: Mojolicious (Perl)
< Content-Length: 29
< Content-Type: text/html;charset=UTF-8
* Replaced cookie mojolicious="eyJleHBpcmVzIjoxNDYxODk4NzQ5LCJjb3VudCI6Mn0---5a8ec66a8db5e54
    a4842f1159e1f116631e62a27" for domain 127.0.0.1, path /, expire 1461898749
< Set-Cookie: mojolicious=eyJleHBpcmVzIjoxNDYxODk4NzQ5LCJjb3VudCI6Mn0---5a8ec66a8db5e54a4842
    f1159e1f116631e62a27; expires=Fri, 29 Apr 2016 02:59:09 GMT; path=/; HttpOnly
< Date: Fri, 29 Apr 2016 01:59:09 GMT
<
    You have visted 2 times.
```

Do you see the message that we visited the index page twice? Our cookie jar saved the state and the server correctly interpreted Cookie: header in the request.

2.9 Forms

User Input

Without user input the web would be boring. We need the user's data so that we can do meaningful work. A large percentage of user data originates from forms. For this section, lets look at the example in different parts. First, we'll have our application logic, and next we'll look at the templates.

The application logic displays the form at line 2; processes our form at line 8; and either displays an error, or success at lines 16 and 19.

Listing 2.7: Application logic

```
1  # Present form
2  get '/' => "slash";
3
4  post '/' => sub {
5      my $self = shift;
6
7      # Process
8      if ("Bender" eq $self->param("name")) {
9          $self->redirect_to("/bender");
10
11         return;
12     }
13
14     # Error
15     $self->flash(error => "Not bender");
16     $self->redirect_to("/");
17 };
18
19 get '/bender';
```

We use the flash for presenting any error message during form processing. The flash is set at line 15 and its value is used in the template. The flash is a methodology for passing a value from one request to another and only lasts for that one request.

This is perfect for passing success, informational, and error messages between requests - such as what is done at line 3 below.

One more thing to note is that in our <form> we have an "action" attribute. With this we set the destination of our POST request. Very important so that data shows up at the proper spot.

Listing 2.8: Templates

```
@@ slash.html.ep

% if (flash("error")) {  #
    <%= flash("error") %><br>
% }

<form method=post action="/"> %#
Name: <input type=text name=name>
</form>

@@ bender.html.ep

Awesome!
```

Now, lets examine exactly what happens when we use our form.

Using curl, we're going to request our form, and then submit the form. The initial request is shown below. This is exactly as if typing "http://127.0.0.1:3000" in the URL bar and then hitting enter.

This will send off a GET request to the server which responds with our form.

```
$ curl -v http://127.0.0.1:3000/
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 73
< Server: Mojolicious (Perl)
< Date: Fri, 29 Apr 2016 03:02:40 GMT
< Content-Type: text/html;charset=UTF-8
<

<form method=post action="/">
Name: <input type=text name=name>
</form>
```

The next command will simulate sending filling out the form with "Bender" and pressing enter. Note how the curl request is now a POST. This queues our application logic to process the 'post => "/"' route. The route verifies the user input (that name does, in fact, equal "Bender") and redirects us onto success route.

Note the use of 302 Found redirection logic.

This is a separate request that the user agent must follow in order to complete the form.

```
$ curl -v -L -d 'name=Bender' http://127.0.0.1:3000/
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> POST / HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 11
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 302 Found
< Location: /bender
< Date: Fri, 29 Apr 2016 03:06:38 GMT
< Content-Length: 0
< Server: Mojolicious (Perl)
<
* Connection #0 to host 127.0.0.1 left intact
* Issue another request to this URL: 'http://127.0.0.1:3000/bender'
* Switch from POST to GET
* Re-using existing connection! (#0) with host 127.0.0.1
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET /bender HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 29 Apr 2016 03:06:38 GMT
< Content-Length: 9
< Server: Mojolicious (Perl)
< Content-Type: text/html;charset=UTF-8
<
Awesome!
```

We have achieved Awesome via our form processing.

Mojolicious Applications

3.1 A Photo app

The previous portion of this book focused on Mojolicious::Lite applications; however, in order to realize our Photo application it will be easier to use the full blown application features of Mojolicious. An application of this type can be started with:

```
$ mojo generate app Photo
```

```
  [mkdir] /opt/photo/script
  [write] /opt/photo/script/photo
  [chmod] /opt/photo/script/photo 744
  [mkdir] /opt/photo/lib
  [write] /opt/photo/lib/Photo.pm
  [mkdir] /opt/photo/lib/Photo/Controller
  [write] /opt/photo/lib/Photo/Controller/Example.pm
  [mkdir] /opt/photo/t
  [write] /opt/photo/t/basic.t
  [mkdir] /opt/photo/public
  [write] /opt/photo/public/index.html
  [mkdir] /opt/photo/templates/layouts
  [write] /opt/photo/templates/layouts/default.html.ep
  [mkdir] /opt/photo/templates/example
  [write] /opt/photo/templates/example/welcome.html.ep
```

As you can see, it has an invocation script, startup package (lib/Example.pm), an example Controller, some tests, and example content (the index.html and .ep template files).

However, starting from scratch is not necessary; we are going to use the git checkout of this book which includes all the code for our Photo app.

```
$ cd /opt
$ git clone git@github.com:brianmed/mojo_book.git
$ cd mojo_book
```

The app is runnable via:

```
$ cd photo
$ morbo -v script/photo
```

When we point our browser to http://127.0.0.1:3000 we'll get our app. As you can see, our version is in /opt/mojo_book/photo and expects its config file to be in /opt/photo.

After the config file has been set up, we can run our app via "script/photo"; this is a Perl script that bootstraps our app. Most of the time you won't need to modify this; however, if module directories are needed, then "use lib" statements can be put here.

What happens next?

3.1.1 Startup

Mojolicious initializes our app and then calls Photo::startup.

The startup method adds and configures plugins; sets up some logging; adds in helpers; and sets up routes. For example, as you can see below in the line from lib/Photo.pm our route for the index page is set up.

Listing 3.1: Index Route

```
    $r->get('/')->to(controller => 'Index', action => 'slash');
```

When a user agent does a GET /, then Photo::Controller::Index::slash will be called and the slash.html.ep file will be served.

Given that, lets go back and dissect Photo::startup one "section" at a time.

First, we turn on development logging for when we are running in production mode. This is a convenience for early stage production or when debugging something. Simply comment out for Production. Next, we initialize the config data structure with a file. These are lines 1 and 3 below.

Listing 3.2: Setup

```
1   $self->log->level("debug");
2
3   my $site_config = $self->plugin("Config" => {file => $self->home->rel_file('../
        photo.config')});
```

This file is just a Perl data structure and can be hand edited or programmatically defined. Another option is to use JSONConfig and a JSON config file. Below is an example config file.

Listing 3.3: Configuration

```
1 {
2     site_secret => "MOAR COREZ foR all the things!",
3 };
```

The next set up phase is to initialize the secret passphrase. Multiple passphrases are supported that allows for phasing out an old passphrase. These passphrases are used for things like signed cookies (which are used in the sessions).

Listing 3.4: Secrets setup

```
    $self->secrets([$$site_config{site_secret}]);
```

After our passphrase we set up our routes; an "under" nested route that verifies the session; and route set up itself. At line 15 the "entry" point to our app from the user agent is defined - namely the initial page. It should be noted that we can define "get" and "post" for these routes.

Listing 3.5: Under route

```
1     my $r = $self->routes;
2
3     my $have_album = $r->under (sub {
4         my $self = shift;
5
6         if (!$self->session("album")) {
7             my $url = $self->url_for('/');
8             $self->redirect_to($url);
9             return undef;
10        }
11
12        return 1;
13    });
14
15    $r->get('/')->to(controller => 'Index', action => 'slash');
```

Arbitrary Perl (DB lookups, JSON parsing, file processing) can be put in the nested route. If "1" is returned the child route will be allowed to run and if "undef" is returned it won't. Very handy for doing basic authentication and authorization checks. If they don't pass either one, then a redirect can happen, or some flag can be set.

In addition "any" is supported - which is useful for supporting GET and POST on the same route. Below are a couple example routes. One for GET and another for POST.

Listing 3.6: Initial routes

```
    $r->get('/album/switch/:name')->to(
        controller => 'Album',
        action => 'switch',
        name => undef
    );
    $r->post('/album/save')->to(
        controller => 'Album',
        action => 'save'
    );
```

One thing that should be mentioned about our routes is that they have two modes of operation. One is when we have no session and another is when there is a session. Our session stores the selected album and the $have_album nested route verifies if the session is still valid. If not, the user agent will be redirected back to the landing page.

The above app logic is shown in the code below.

Listing 3.7: Session route

```
    $have_album->get('/album/show')->to(
        controller => 'Album',
        action => 'show'
    );
```

3.1.2 Controller

A controller encapsulates the business logic for our website. Fine grained control can be exerted over the app by using packages and subs. One package can control the "main" page with login and logout and other packages can focus on other sub systems of our app.

For our app, we have the Index and Album controllers. One is the landing page for our app and the other holds all the business logic for the photo album.

We can view an album, create one, switch to a new album, upload a photo, and view a photo.

3.1.3 Landing Page

The main page does several things - allow for viewing; selecting, or creating an album. Below is the method responsible for all this.

Listing 3.8: Landing page logic

```
1  sub slash {
2      my $c = shift;
3
4      my $all = SiteCode::Albums->new(path => $c->app->home->rel_dir("albums"))->all;
5
6      if (0 == @{ $all }) { # Create an album if none found
7          my $url = $c->url_for('/album/create');
8          return($c->redirect_to($url));
9      }
10
11     if ($c->session("album")) { # Show the album if we have a session
12         my $url = $c->url_for('/album/show');
13         return($c->redirect_to($url));
14     }
15
16     my $url = $c->url_for('/album/switch'); # Select an album if nothing selected
17     return($c->redirect_to($url));
```

Code starting at line 4 checks to see if any albums have been created; if not, the user is redirected to /album/create. In addition, we're using the handy url_for and redirect_to methods. These methods allow for more easily working with URLs and generating a 302 response.

One last thing, is that this switch-a-roo logic is made possible by the marvelous session handling in Mojolicious. There are several ways to use the session, two are detailed below. The first line reads a variable from the session and the second line sets the variable.

```
my $foo = $c->session('foo');
$c = $c->session(foo => 'bar');
```

That's all you got to do, and you get:

```
Persistent data storage for the next few requests, all session data gets
serialized with Mojo::JSON and stored Base64 encoded in HMAC-SHA1 signed
cookies, to prevent tampering. Note that cookies usually have a 4096 byte (4KB)
limit, depending on browser.
```

Photo

Initially, no photo albums exist in the model.

Given that, the landing page redirects the user to an action that creates an album. This action is not in Photo::Controller::Album, it uses automatic rendering (explained below).

```
The renderer can be manually started by calling the method "render" in
Mojolicious::Controller, but that's usually not necessary, because it will get
automatically called if nothing has been rendered after the router finished its
work. This also means you can have routes pointing only to templates without
actual actions.
```

This automated rendering takes our template file and displays it. The template name is generated by taking the controller, action, format, and handler and combining them like so: controller/action.format.handler. Therefore, our template is album/create.html.ep.

The main things to note in this template are the error and form handling. The error section is here:

Listing 3.9: Error handling

```
% if (flash('error')) {
    <div class="row">
    <div class="span12">

    <div class="alert alert-error">
      <button type="button" class="close" data-dismiss="alert">&times;</button>
      <%= flash('error') %>
    </div>

    </div>
    </div>
% }
```

The main points are the % to embed Perl code in the template sandbox and the flash usage. The flash is persistent for one-request only and is stored in the session. The error would come from the POST form action (shown below).

Listing 3.10: Form start

```
    <form action="/album/save" method="post">
```

The POST action Album::Controller::save does several things, most notably it validates the input, creates the album, and stores the album name in the session. That is, it "switches" to that album. Below is the error handling. Of note is that the flash error value is stored with an API similar to the stash. After setting our error flag we redirect back to displaying the create view.

Listing 3.11: Switching albums error checking

```
1     my $dir = $c->app->home->rel_dir("albums");
2     my $album_name = $c->param("album_name");
3
4     unless ($album_name) {
5         $c->flash("error" => "No album name given");
6
7         my $url = $c->url_for('/album/create');
8         return($c->redirect_to($url));
9     }
```

After the input checks are done and the album created, we switch to that album and redirect back to the landing page - which will detect the selected album and act appropriately.

Listing 3.12: Switching albums logic

```
1     warn("album_name: $album_name");
2     $c->session(album => $album_name);
3
4     my $url = $c->url_for('/');
5     return($c->redirect_to($url));
```

Also, one thing to see that we have a "warn" in the code there. It's main purpose is to show that this is possible and will go the "console". For example, if running through morbo, then the message will be intermixed in the morbo output and not in the log file.

So, after the above, we go back to the landing page - which redirects to the album viewer. The action for this is Photo::Controller::Album::show. This sub is responsible for shimmying data from our model (a bunch of JSON files) to the view. Below is how that is accomplished.

It should be noted that we are able to pass an object to the view.

Listing 3.13: Viewing albums

```
1     $c->stash(album => $album);
2     $c->stash(slots => $album->slots);
3     $c->stash(albums => $albums->all);
```

In addition, we put some debug information in the log to demonstrate how that operates.

Listing 3.14: Debugging example

```
1     $c->app->log->debug("album: " . $album->name);
```

Our template for this action (album/show.html.ep) does several things. In the HTML body we have our error handling, form to handle a new photo upload, button that allows for switching between albums, and the picture viewing code.

The picture code is in the template twice. The first time is if there are no photos to view and the second is when we have a photo.

The form code is shown below.

Listing 3.15: Form setup

```
1  <form action="/album/upload" method="post" enctype="multipart/form-data"> %#
2    <fieldset>
3      <legend>
4          <%= stash('album')->name %> - %#
5          <div class="btn-group">
6              <a class="btn btn-inverse dropdown-toggle" data-toggle="dropdown">
7                Albums
8                <span class="caret"></span>
9              </a>
10             <ul class="dropdown-menu">
11                 <li><a href="/album/create">Create</a></li>
12                 <li class="divider"></li>
13                 % foreach my $album (@{ stash('albums') }) {
14                   <li><a href="/album/switch/<%= $album->name %>"><%= $album->name
                        %></a></li>
15                 % }
16             </ul>
17         </div>
18     </legend>
19     <label>Label</label>
20     <input type="text" placeholder="Label" name="label">
21
22     <label>Description</label>
23     <input type="text" placeholder="Description" name="descr">
24
25     <label>File</label>
26     <input type="file" name="photo">
27   </fieldset>
28 <br>
29 <button type="submit" class="btn">Upload</button>
30 </form>
```

At line 1 we specify action that will happen when we upload a file. Line 4 is pretty cool because we're accessing a value in the stash that is an object. Lines 13 to 15 process our album array to generate a pulldown for the user to select an album. They are shown below.

Listing 3.16: Album links

```
1                  % foreach my $album (@{ stash('albums') }) {
2                    <li><a href="/album/switch/<%= $album->name %>"><%= $album->name
                        %></a></li>
3                  % }
```

The ability to intermix Perl loop and control structures with plain html is very powerful. As a specific example, the above allows us to iterate over an array of objects in the stash to create a portion of our html document.

After our form, we process the HTML to display the album. This is shown below.

Listing 3.17: Display album

```
1     % if (0 == @{ stash('slots') }) {
2         <div class="row">
3             <div class="span12">
4                 No photos found
5             </div>
6         </div>
7     % }
8
9     % if (0 != @{ stash('slots') }) {
10    ...
11    % }
```

Note that there are two sections. One is for when there are no photos in the album and the other is for when there are. The code for when there are photos is much more interesting than when there aren't, so lets look at a few snippets from it:

This snippet processes our album array and is used to give us a 'lil dot to click on to maneuver through the pics. Of note is how the $active variable is used to populate our class.

Listing 3.18: Dot code

```
1     % foreach my $slot (@{ stash('slots') }) {
2         % my $active = $slot->{idx} ? "" : "active";
3         <li data-target="#myCarousel" data-slide-to="<%= $slot->{idx} %>" class="
              <%= $active %>"></li>
4     % }
```

The next snippet displays the pictures. The line at 7 provides the src for the img tag. Without it, it would be difficult to have pictures.

Listing 3.19: Display pictures

```
1  <div class="carousel-inner">
2
3      % foreach my $slot (@{ stash('slots') }) {
4          % my $active = $slot->{idx} ? "" : "active";
5
6          <div class="item <%= $active %>">
7              <img src="<%= url_for->path("/album/photo/$slot->{idx}") %>"> %#
8              <div class="carousel-caption">
```

```
                    <h4><%= $slot->{label} %></h4>

                <p><%= $slot->{descr} %></p>
            </div>
        </div>
    % }

</div>
```

Finally, we the left and right arrows over the image if there is more than photo.

Listing 3.20: Navigation

```
% if (1 != @{ stash('slots')}) {
<a class="left carousel-control" href="#myCarousel" data-slide="prev">◀</a>
<a class="right carousel-control" href="#myCarousel" data-slide="next">▶</a>
% }
```

OK! We've done a lot and have a bit more to go. We should talk about the file upload and img src tag routes. Below is the form code that will POST to our upload route.

Listing 3.21: Upload form

```
<form action="/album/upload" method="post" enctype="multipart/form-data"> %#
```

The upload action is this sub: Photo::Controller::Album::upload. This does some validation, creates a photo album object, and then stores the photo in a new "slot". A slot is merely a way to refer to a set of pictures. Each slot is comprised of two files, one being the metadata for the image (a JSON file) and the other being the image itself.

The img src tag route, which is the Photo::Controller::Album::photo sub is shown below.

Listing 3.22: Display photo

```
sub photo {
    my $c = shift;

    my $dir = $c->app->home->rel_dir("albums");
    my $album = SiteCode::Album->new(path => "$dir/" . $c->session->{album}, name
       => $c->session->{album}); @

    my $slot = $c->param("slot");

    my $filename = $album->photo($slot);

    $c->reply->asset(Mojo::Asset::File->new(path => $filename));
}
```

This retrieves our selected album, gets the photo in a particular slot, and then sets our reply to use that file. At line 5 is where the album object is created - this shows one way to access session variables. The slot is given to us from the template and 9 gets the filename for the photo in that slot. It's an absolute path.

This path is then served at line 11.

The last bit of code is used to manually switch to a different album. It's main utility is when the session expires and the user has to choose an album, again. The code is below.

Listing 3.23: Switch albums

```
1  sub switch {
2      my $c = shift;
3
4      if (defined $c->param("name")) {
5          my $album_name = $c->param("name");
6
7          $c->session(album => $album_name);
8
9          my $url = $c->url_for('/');
10         return($c->redirect_to($url));
11     }
12
13     my $dir = $c->app->home->rel_dir("albums");
14     my $all = SiteCode::Albums->new(path => $dir)->all;
15
16     $c->stash(albums => $all);
17
18     return($c->render);
19 }
```

If we've been given a name from the view, then we save the album name in the session and redirect back to the landing page. This should then do another redirect back to the album viewer.

3.2 Blogging app

Next, we will focus on a new app that was heavily adapted from Mojo::Pg (another one of Sebastian Riedel's many creations). We will look at adding helpers and database usage into our bag of tricks.

The code is available via git:

```
$ cd /opt
$ git clone git@github.com:brianmed/mojo_book.git
```

The app is runnable via:

```
$ cd blog
$ morbo -v script/blog
```

This app should feel similar to the Photo app; so, we won't repeat very much of the information that can be retrieved from there. Below we start examining the unique aspects of the startup routine.

3.2.1 Startup

Given that Mojolicious initializes our app via the startup sub, then that's a natural place for adding our database set up. This is accomplished with helpers. These are code blocks that are available to the controller and templates.

Listing 3.24: Helpers

```
1  # Model
2  $self->helper(sql => sub { state $sql = Mojo::SQLite->new('sqlite:_blog.sqlite')
       });
3  $self->helper(
4    posts => sub { state $posts = Blog::Model::Posts->new(sql => shift->sql) });
```

In Line 2 we use the app object and create a helper. The state variable will persist in the process and a Mojo::SQLite object will be available. Given this, at least one database connection per process is required.

The next helper is our Model object. This is the glue between our controller, data (the Model), and view.

The code below will create and version control a database. Super sweet. A very minimal, yet versatile file format is used for this versioning and creation.

Listing 3.25: Migration Logic

```
  # Migrate to latest version if necessary
  my $path = $self->home->rel_file('migrations/blog.sql');
  $self->sql->migrations->name('blog')->from_file($path)->migrate;
```

Below is the actual migration file contents.

Listing 3.26: Migration SQL

```
-- 1 up
create table if not exists posts (
  id    serial primary key,
  title text,
  body  text
);

-- 1 down
drop table if exists posts;
```

This syntax uses SQL comments with metadata for versioning. The *1 up* is used when going from version 0 –> 1. And, the *1 down* is used when going from 1 –> 0.

The routes are very similar to the Blog app and are listed below.

Listing 3.27: Blog routes

```
  # Controller
  my $r = $self->routes;
  $r->get('/' => sub { shift->redirect_to('posts') });
  $r->get('/posts')->to('posts#index');
  $r->get('/posts/create')->to('posts#create')->name('create_post');
  $r->post('/posts')->to('posts#store')->name('store_post');
  $r->get('/posts/:id')->to('posts#show')->name('show_post');
  $r->get('/posts/:id/edit')->to('posts#edit')->name('edit_post');
  $r->put('/posts/:id')->to('posts#update')->name('update_post');
  $r->delete('/posts/:id')->to('posts#remove')->name('remove_post');
```

There are; however, a few key differences. The / route (index) shows how an anonymous sub can be used instead of a package. Also, the succinct format of 'controller#action' is used with naming routes.

These names can be referenced later via url_for and redirect_to - in fact, the / route uses *redirect->to('**posts**')*.

3.2.2 Controller

Our blog begins here:

Listing 3.28: Initial route

```
1 sub index {
2   my $self = shift;
3   $self->render(posts => $self->posts->all);
4 }
```

Line 3 is where the magic happens. We use the *posts* helper and call the *all* sub in the Blog::Model::Posts package.

This returns an array of posts which the template then processes.

Listing 3.29: Posts SQL

```
1 sub all { shift->sql->db->query('select * from posts')->hashes->to_array }
```

The template is below and uses a layout for making templates with a common structure easier (e.g. HTML documents).

Listing 3.30: Post template

```
1 % layout 'blog', title => 'Blog';
2 % for my $post (@$posts) {
3   <p>
4     <h2><%= link_to $post->{title} => show_post => {id => $post->{id}} %></h2>
5     %= $post->{body}
6   </p>
7 % }
8 %= link_to 'New post' => 'create_post'
```

As you can tell, line 2 processes our array and creates the post listings. Our layout, listed below, allows for rapid and concise construction of a plethora of templates.

Listing 3.31: Blog layout

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title><%= title %></title>
5      <style>
6        a, body { color: #2a2a2a }
7        body { font: 0.9em 'Helvetica Neue', Helvetica, sans-serif }
8        input[type=text], textarea { width: 600px }
9        input.field-with-error, textarea.field-with-error {
10         border: 4px solid #f00;
11       }
12       textarea { height: 300px }
13     </style>
14   </head>
15   <body>
16     <h1><%= link_to 'Blog' => 'posts' %></h1>
17     %= content
18   </body>
19 </html>
```

Line 17 is where the content from the *calling* template is inserted.

Further inspection of the template shows that at line 4 we have a tag helper that uses the named route **show_post**. This creates the blog listing.

All that in one line 'o code:

Listing 3.32: Blog posts rendering

```
1    $self->render(posts => $self->posts->all);
```

3.3 Chat app

Wow, we have a photo app and a blogging app. What's next is a chat app. It's a Mojolicious::Lite app that in around 80 lines has...

- Real-time websocket chat
- Database Schema Set up
- Database Data Reset
- Records number of users
- Sends status messages every 10 seconds
- All templates included

The code is available via git:

```
$ cd /opt
$ git clone git@github.com:brianmed/mojo_book.git
```

The app is runnable via:

```
$ cd chat
$ morbo -v chat.pl
```

Starting from the top, we use a helper in exactly the same way as our blogging app and then use the super sweet migration features of Mojo::SQLite and resets the number of connected users.

This is reflected in the lines below.

Listing 3.33: Chat database

```
1 helper sql => sub { state $sql = Mojo::SQLite->new("sqlite:_chat.sqlite") };
2
3 # Setup and reset database
4 app->sql->migrations->from_data("main", "migrations")->migrate;
5 app->sql->db->query("DELETE FROM connected");
```

Next, our index page is configured via..

Listing 3.34: Index route

```
1 get "/" => "chat";
```

And, uses the template here:

Listing 3.35: Chat logic

```
@@ chat.html.ep
<form onsubmit="sendChat(this.children[0]); return false"><input></form>
<div id="log"></div>
<script>
  var ws  = new WebSocket('<%= url_for('channel')->to_abs %>');
  ws.onmessage = function (e) {
    document.getElementById('log').innerHTML += '<p>' + e.data + '</p>';
  };
  function sendChat(input) { ws.send(input.value); input.value = '' }
</script>
```

This is a very succinct way to express which template goes with a route that needs no app logic executed in the controller.

The heart of our app is the websocket connection. This does several things:

- Set up the connection
- Identify the connection
- Sends status notifications
- Forwards messages via pubsub
- Receives messages via pubsub
- Gracefully cleanup
- Records presence

First, we configure the connection and then we identify the connection uniquely. The unique connection is identified with the monotonic time that steady_time provides.

Listing 3.36: Websocket route

```
1 websocket "/channel" => sub {
2   my $c = shift;
3
4   # Setup connection
5   $c->inactivity_timeout(3600);
6
7   # Identify connection
8   $c->stash("unique", md5_sum(steady_time));
```

Next we set up a recurring timer that will send a message every 10 seconds to the websocket connection. This type of app logic is very powerful and there are a variety of possibilities with websockets, event loops (IOLoop), databases, and the async feature set of Mojolicious which will allow realizing amazing apps.

Listing 3.37: Notifications

```
  # Send FYI notifications
  my $id = Mojo::IOLoop->recurring(10 => sub {
    my $loop = shift;

    my $human = "people";
    my $word = "are";

    my $connected = $c->sql->db->query(qq(
      SELECT COUNT(person) as count
      FROM connected
      WHERE person != ?),

      $c->stash("unique")
    )->hash->{count};

    if ($connected) {
      $human = "person" if 1 == $connected;
      $word = "is" if 1 == $connected;
    }

    $c->send(sprintf("The time is now: %s, $connected other $human $word connected"
      ,
      scalar(localtime)));
  });
```

The next bit 'o code will blast out and receive the chat messages via the pubsub capabilities of Mojo::SQLite and friends.

Listing 3.38: PubSub

```
  # Forward messages from the browser to SQLite
  $c->on(message => sub { shift->sql->pubsub->notify(mojochat => shift) });

  # Forward messages from SQLite to the browser
  my $cb = $c->sql->pubsub->listen(mojochat => sub { $c->send(pop) });
```

The amount of expressiveness that Mojolicious provides is super cool. In a couple lines of code we have realized the bulk of our chat app.

The next bit gracefully closes our connection. We stop listening for chat messages; remove ourselves from the pool of connected users; and stop the recurring timer.

Listing 3.39: Cleanup

```
  # Gracefully cleanup
  $c->on(finish => sub {
    my $c = shift;
```

```
4
5       $c->sql->pubsub->unlisten(mojochat => $cb);
6
7       $c->sql->db->query("DELETE FROM connected WHERE person = ?", $c->stash("unique"
            ));
8
9       Mojo::IOLoop->remove($id);
10    });
```

Finally, our app announces its presensce via a DB call.

Listing 3.40: Presence

```
1   # Record our presence
2   $c->sql->db->query("INSERT INTO connected VALUES (?, CURRENT_TIMESTAMP)", $c->
        stash("unique"));
```

One last thing that should be noted is that our current database calls block.

Examples

4.1 Basic Authentication

Authentication and authorization are necessary for content protection. There are several ways they can be realized. One is with Basic authentication and another is with sessions. This first example is with Basic that returns a 401 to the client.

We check for the proper username and send back a 401 if not found.

Listing 4.1: Authentication logic

```
1  under (sub {
2    my $c = shift;
3
4    # Check for username "Bender" and password "rocks"
5    if (secure_compare($c->req->url->to_abs->userinfo // "", 'Bender:rocks')) {
6      return 1;
7    }
8
9    # Require authentication
10   $c->res->headers->www_authenticate('Basic');
11   $c->render(text => 'Authentication required!', status => 401);
12
13   return undef;
14 });
```

If Bender is found, then one of the GET requests are served..

Listing 4.2: Protected content

```
1  get '/' => sub {
2    my $c = shift;
3
4    return $c->render(text => 'Hello Bender!');
5  };
6
7  get '/time' => sub {
8    my $c = shift;
9
10   return $c->render(text => scalar(localtime));
11 };
```

The main thing to denote from this example is that Mojolicious makes your life easy with enough primitives that get out of your way so that development can happen faster.

4.2 Session Authentication

Our next example is with session authentication. We have..

- A Landing page
- Login page view
- Login page processing
- Logout processing
- Session authentication
- Member area
- Private file retrieval

Below is our landing page. If we are logged in, then we proceed onto the member area, if not we get the login page. The landing page app logic is a non-trivial decision where we determine if the login page should be given for an already logged in person, the user gets redirected to the member area, or they get a real landing page.

Listing 4.3: Initial route

```
1 # Landing page
2 get '/' => sub {
3   my $c = shift;
4
5   return $c->redirect_to("/time") if $c->session("username");
6   return $c->redirect_to("/login");
7 };
```

Our login logic is below. The login GET request is just a template and processing via POST is where we do username authentication. We don't really do any authorization; however, using the power of our minds we can imagine different routes that require a specific set of usernames, or we can add groups onto our user scheme and show different templates based on that.

The options are really almost limitless.

Listing 4.4: Login logic

```
# Show login
get '/login' => 'login';

# Process login
post '/login' => sub {
  my $c = shift;

  # Authentication
  unless ("Bender" eq $c->param("username")) {
    return $c->redirect_to("/login");
  }
  if ("rocks" ne $c->param("password")) {
    return $c->redirect_to("/login");
  }

  ### The session persists across requests via cookies
  $c->session(username => $c->param("username"));

  # Expiration date in seconds from now (persists between requests)
  #
  # This is how long they are logged in
  $c->session(expiration => 604800);

  return $c->redirect_to("/time");
};
```

The final piece for our login scheme is that session expiration can be set programmatically. This will give us the ability to set the time our user can stay logged in.

An important piece of our session management is the ability to logout. This is done via HTML links. Below we set the session expiration to a previous time and this resets the session.

Listing 4.5: Logout logic

```
# Exit member area
get '/logout' => sub {
  my $c = shift;

  # Delete whole session by setting an expiration date in the past
  $c->session(expires => 1);

  $c->redirect_to("/login");
};
```

Our actual session validation is done via an under route. Since this is a Mojolicious::Lite app, then all routes after the under will be subject to authentication. The session authentication merely verifies that a username is present in the session.

If no username, then we go back to the login screen. If so, we continue our march into the route in question.

Listing 4.6: Authentication logic

```
# Session authentication
under (sub {
  my $c = shift;

  # Already logged in?
  if ($c->session("username")) {
    return 1;
  }

  $c->redirect_to("/login");

  return undef;
});
```

The main new feature of our member area is that "private" files are protected. We use Mojo::Asset::File and present the "private" file (i.e. any file with an absolute path) for the user. These lines are below.

Listing 4.7: Protected content

```
  $c->res->headers->content_type('text/plain');
  $c->reply->asset(Mojo::Asset::File->new(path => '/etc/passwd'));
```

4.3 JSON API

Moving along, our next example utilizes a JSON API that abstracts out a key/value store written using Mojo::SQLite.

In our under route which allows for the JSON API authentication. The under does several things, which are:

- `Validate JSON`
- `Verify username`
- `Verify API key`
- `Authenticate user`

Below is the validation. Note how the controller object ($c) has a req object which includes the JSON already parsed. Also, our under renders and returns undef so that the user is presented with their JSON and the child routes are not processed.

Listing 4.8: Authentication start

```
1  under (sub {
2      my $c = shift;
3
4      unless ($c->req->json) {
5        $c->render(json => {
6          status => "error", data => { message => "No JSON found" }
7        });
8
9        return undef;
10     }
```

After validation, we authenticate the user. If there are errors, then a similar scenario exists where the JSON is rendered and undef is returned. However, when the authentication succeeds, then "1" is returned and the children routes can be processed.

Listing 4.9: Validate credentials

```
    unless ("fnord" eq $username) {
      $c->render(json => {
        status => "error", data => { message => "Credentials mis-match" }
      });

      return undef;
    }

    unless ("68b329da9893e34099c7d8ad5cb9c940" eq $api_key) {
      $c->render(json => {
        status => "error", data => { message => "Credentials mis-match" }
      });

      return undef;
    }

    return 1;
});
```

Next, we have helpers for our CRUD datastore and the GET/POST routes that allow for data retrieval and modifications.

Listing 4.10: CRUD helpers

```
helper insert => sub {
  my $c = shift;

  my $email = $c->req->json->{email};
  my $key = $c->req->json->{key};
  my $value = $c->req->json->{value};

  $c->sql->db->query(
"INSERT INTO keys (email, key, value) VALUES (?, ?, ?)",
     $email, $key, $value
  )->last_insert_id;

  return $c;
};

helper select => sub {
  my $c = shift;

  my $email = $c->req->json->{email};
  my $key = $c->req->json->{key};

  return $c->sql->db->query(
"SELECT * from keys WHERE email = ? and key = ?",
    $email, $key
  )->hash;
};
```

The above are the INSERT and SELECT helpers. Given how the insert helper returns $c and select returns $hash, we can do something like this: "$c->insert->select". The goal is a fluent interface to our datastore.

After our helpers, we have the actual routes that manipulate data. Note how straightforward it is to return JSON with something as simple as "$c->render(json => ...)".

Listing 4.11: CRUD routes

```
any '/v1/insert' => sub {
    my $c = shift;

    return($c->render(json => {
      status => "success",
      datum => $c->insert->select
    }));
};

any '/v1/select' => sub {
    my $c = shift;

    return($c->render(json => {
      status => "success",
      datum => $c->select
    }));
};
```

That's basically it. Next is a sample run via the command-line.

```
$ curl -v -X POST 'http://127.0.0.1:3000/v1/insert' --data '{"username":"fnord","api_key":"68
    b329da9893e34099c7d8ad5cb9c940","email":"a@a.com","key":"str","value":"hello_world"}'
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> POST /v1/insert HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 117
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 117 out of 117 bytes
< HTTP/1.1 200 OK
< Server: Mojolicious (Perl)
< Date: Sat, 23 Apr 2016 17:43:12 GMT
< Content-Type: application/json;charset=UTF-8
< Content-Length: 122
<
* Connection #0 to host 127.0.0.1 left intact
{"status":"success","datum":{"key":"str","value":"hello_world","inserted":"2016-04-23
    17:43:12","id":1,"email":"a@a.com"}}
$ curl -v -X GET 'http://127.0.0.1:3000/v1/select' --data '{"username":"fnord","api_key":"68b
    329da9893e34099c7d8ad5cb9c940","email":"a@a.com","key":"str"}'
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 3000 (#0)
> GET /v1/select HTTP/1.1
> Host: 127.0.0.1:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 95
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 95 out of 95 bytes
< HTTP/1.1 200 OK
< Server: Mojolicious (Perl)
< Date: Sat, 23 Apr 2016 17:43:23 GMT
< Content-Type: application/json;charset=UTF-8
< Content-Length: 122
<
* Connection #0 to host 127.0.0.1 left intact
{"datum":{"inserted":"2016-04-23 17:43:12","key":"str","value":"hello_world","email":"a@a.
    com","id":1},"status":"success"}
```

4.4 Synchronizing non-blocking operations

Our next example was lifted from the Mojolicious Cookbook. The cookbook has a plethora of examples and other goodness. The example we are focusing on is synchronizing api calls in a non-blocking fashion.

What we are going to do is search metacpan with a couple queries in a non-blocking fashion.

The first bit is the delay. This is methodology for orchestrating callbacks. Simply do Mojo::IOLoop->delay and you..

```
Build Mojo::IOLoop::Delay object to manage callbacks and control the flow of
events for this event loop, which can help you avoid deep nested closures that
often result from continuation-passing style. Callbacks will be passed along to
"steps" in Mojo::IOLoop::Delay.
```

Below is what the code looks like that starts the delay.

Listing 4.12: Index route

```
1 # Search MetaCPAN for "mojo" and "minion"
2 get '/' => sub {
3   my $c = shift;
4
5   # Prepare response in two steps
6   $c->delay(
```

The steps needed by the delay are sub CODE blocks. The first one sets up our concurrent requests. This is shown below:

Listing 4.13: GET requests

```
1     # Concurrent requests
2     sub {
3       my $delay = shift;
4       my $url   = Mojo::URL->new('api.metacpan.org/v0/module/_search');
5       $url->query({sort => 'date:desc'});
6       $c->ua->get($url->clone->query({q => 'mojo'})   => $delay->begin);
7       $c->ua->get($url->clone->query({q => 'minion'}) => $delay->begin);
8     },
```

We use the embedded Mojo::UserAgent object and the begin methods of the delay that set up the number of requests that will happen concurrently. When the GET request is finished, then the UserAgent will execute the begin callback. This callback does the following:

```
Indicate an active event by incrementing the event counter, the returned
callback needs to be executed when the event has completed, to decrement the
```

```
event counter again. When all callbacks have been executed and the event
counter reached zero, "steps" will continue.
```

After the first step is finished, then the rendering takes place. The code for this is below.

Listing 4.14: Request output

```
1     # Delayed rendering
2     sub {
3       my ($delay, $mojo, $minion) = @_;
4       $c->render(json => {
5         mojo   => $mojo->res->json('/hits/hits/0/_source/release'),
6         minion => $minion->res->json('/hits/hits/0/_source/release')
7       });
8     }
```

The $c->render will display the json to the user. The point of this example is that the Mojo::UserAgent get requests were happening while other requests on the server were being served.

4.5 Mojo::UserAgent Command-Line

This is our first command-line example. We are retrieving the weather for a given IP address. There are two GET requests. One retrieve the latitude and longitude for an IP address, and the next takes that and gets the weather.

The below code sets up our script. We use Mojo::Base -strict for enabling Modern Perl features in our script. Also, the very succinct Mojo::IOLoop->delay allows us some freedom for running non-blocking code in a serial fashion.

Listing 4.15: Setup

```
1 #!perl
2 use Mojo::Base -strict;
3
4 use Mojo::IOLoop;
5 use Mojo::UserAgent;
6
7 my $ua = Mojo::UserAgent->new;
8 my $addr = shift // die("Please pass in an IP");
```

After this, the 1st GET request is done. Note how we use an array of CODE references for our delay method. The first sub initiates a non-blocking request with ip-api.com.

Listing 4.16: Lat/Lon

```
1 # Non-blocking requests (synchronized with a delay)
2 Mojo::IOLoop->delay(
3   sub {
4     my $delay = shift;
5
6     my $ip_query = sprintf("http://ip-api.com/json/%s", $addr);
7     $ua->get($ip_query => $delay->begin);
8   },
```

And then we set up the query for the final GET request that retrieves the weather. Note how the transaction from the previous request is available in this delay step.

Listing 4.17: Weather

```
1   sub {
2     my ($delay, $ip) = @_;
3
4     # Setup weather query
5     my $query = sprintf(
6         "lat=%s&lon=%s&unit=0&lg=english&FcstType=json",
7         $ip->res->json->{lat},
```

```
        $ip->res->json->{lon}
    );
    my $url = sprintf("http://forecast.weather.gov/MapClick.php?%s", $query);

    $ua->get($url => $delay->begin);
  },
```

In the last step we print the weather. Note the compact methodology of accessing the response and the embedded json embedded.

Listing 4.18: Output

```
  sub {
    my ($delay, $weather) = @_;

    # Talk about the weather
    my $j = $weather->res->json;
    say(sprintf(
        "$addr: %s: %s: %s",
        $j->{location}{areaDescription},
        $j->{time}{startPeriodName}[0],
        $j->{data}{text}[0]
    ));
  }
)->wait;
```

The final coolness that we use from Mojo::IOLoop::Delay is the wait method. This will..

```
Start "ioloop" and stop it again once an "error" or "finish" event gets
emitted, does nothing when "ioloop" is already running.
```