An Apache/CRO Web Server

by

Tony O'Dell
2022-09-23
[https://deathbykeystroke.com]

Part 1

Building a Cro App - Part 1 building-a-cro-app-part-1 rakudo, programming

This whole article will encompass building a Cro application (with DB and templating), configuring your apache web server, and ensuring your app is running with systemd.

Part 1 - Building the Cro app

We're going to build a little 90s style forum, like Bytamin-C. This allows users to log in and participate in some chats and have a profile, or surf around and view some of the chats others are having.

Let's set up our directory structure first:

```
.
 bin
    app
 lib
 META6.json
 resources
 templates
```

In `bin/app` we'll have nothing for the time being, this will eventually be the script we use to start the webserver. In `META6.json` lets add the following:

```
{
  "name":         "Demo",
  "auth":         "zef:tony-o",
  "version":      "0.0.1",
  "api":          "0",
  "description": "A demo cro app",
  "depends":      ["cro", "Cro::WebApp::Template"],
  "provides":     {}
}
```

After doing this, you can run `zef install --/test --depsonly .` and you should end up with our app's dependencies so we can start filling out the rest.

The `bin/app`

Typically stuff inside of the bin directory doesn't gain the benefit of precompilation. To gain some start up speed and benefits, the `bin/app` file will just be stubbed to start our app. So let's create the bit we want precompiled in `lib/Demo/App.rakumod` add the following:

```
unit module Demo::App;

use Cro::HTTP::Router;
use Cro::HTTP::Server;
use Config;

my $application = route {
  get -> 'greet', $name {
    content 'text/plain', "Hello, $name!";
  }
}

my $host = config<listen-ip>   // '0.0.0.0';
my $port = config<listen-port> // 10000;

my Cro::Service $service = Cro::HTTP::Server.new(:$host, :$port, :$application);
```

```
$service.start;

say "Listening: $host:$port";

react whenever signal(SIGINT) {
  $service.stop;
  exit;
}
```

You'll see we're including something called `Config` , we haven't created this yet so let's add this to `lib/Config.rakumod` :

```
unit module Config;

sub config is export {
  {
    listen-ip   => '127.0.0.1',
    listen-port => 8666,
  };
}
```

Now, as we add things that need configuration we can add those to this file. There are plenty of ways to include config and if you're storing passwords and secrets then this module should retrieve those in a secure way. We still can't run this app to test it until we modify the `provides` in our `META6.json` :

```
{
  ...
  "provides": {
    "Demo::App": "lib/Demo/App.rakumod",
    "Config":    "lib/Config.rakumod"
  }
}
```

Now, when you run: `raku -I. -e 'use Demo::App'` you'll get the message of where your server is listening and if you hit that ip/port + `/greet/world` with your browser (`http://127.0.0.1:8666` if you didn't modify the config) then you should get a nice message of `Hello, world!`. This means our server is working. Let's stub it out now, in `bin/app` :

```
#!/usr/bin/env raku
use Demo::App;
```

Now your server can run with `raku -I. bin/app` and you can hit the same endpoint you did above for great success.

Building Templates and Static Files

Static Files

Serving static files is fairly simple in cro, add the following routes to `lib/Demo/App.rakumod` :

```
get -> 'style', *@path {
  static 'static/style', @path;
}

get -> 'script', *@path {
  static 'static/script', @path;
}
```

Cro protects us against a query like `/script/../db.sqlite` so we don't need to worry about path resolution ending up outside of our static file directories. You now need to create the directories `static/style` and `static/script`

you should have the directory structure of:

```
.
 bin
    app
 lib
    Config.rakumod
    Demo
        App.rakumod
 META6.json
 resources
 static
    script
    style
 templates
```

That's it for static files.

Templates

Cro has the concept of partials so we're going to modify our `greet/$name` route to work off of a template.

First, change our `use` block in `lib/Demo/App.rakumod` to include `Cro::WebApp::Template` :

```
unit module Demo::App;

use Cro::HTTP::Router;
use Cro::HTTP::Server;
use Cro::WebApp::Template;

use Config;
```

And modify our application's route block to be:

```
my $application = route {
  template-location 'templates/';

  get -> 'greet', $name {
    template 'main.crotmp', {:$name};
  }
}
```

With these changes we're telling cro to look in the `templates/` directory for whatever `.crotmp` first and we're telling our greeter route to use a template giving it access to the `$name` variable. Now we need to create the template `templates/main.crotmp` :

```
<!DOCTYPE html>
<html>
  <head />
  <body>
    Greetings, <.name>!
  </body>
</html>
```

Now firing up our application and visiting our server at `/greet/world` we'll get the same message but this time complete with html! Now let's build out our authorization and functionality.

Building Our Application's Functionality

If we want to let users post and sign up, we're going to want a database. Let's define our table schemas (using SQLite for this demo), in `schema.sql` :

```
CREATE TABLE users (
  id         UUID NOT NULL PRIMARY KEY,
  name       TEXT NOT NULL,
  foreign_id TEXT
);

CREATE TABLE posts (
  id      UUID NOT NULL PRIMARY KEY,
  author  UUID NOT NULL,
  title   TEXT NOT NULL,
  body    TEXT NOT NULL,
  parent  UUID,

  FOREIGN KEY(author) REFERENCES users(id)
);
```

Now run `sqlite3 resources/db.sqlite < schema.sql`, you should now have a `db.sqlite` database with our posts and users ready to use. Let's make sure we keep our `META6.json` up to date, add a `resources` key:

```
{
  ...
  "provides": {...},
  "resources": [ "db.sqlite" ]
}
```

We're going to morph our greeter route to just be our app's main route and allow people to log in. In `lib/Demo/App.rakumod` make the modifications to our `$application` :

```
my $application = route {
  template-location 'templates/';

  get -> {
    template 'main.crotmp';
  }
}
```

Now we're at a break point. For this article we're going to use github's oauth so you'll need to take a break from this article and create a github oauth app (at the time of writing this, the path in github is `Settings/Developer Settings`) with the following info:

Github Screenshot

Now we need the secret you generated and the client id in `lib/Config.rakumod` :

```
sub config is export {
  {
    listen-ip   => '127.0.0.1',
    listen-port => 8666,
    gh-secret   => 'd4.....................6bb9d3',
    gh-client   => '85c...fc50decdf74c8',
  };
}
```

Okay! Let's build out an OAuth flow. Essentially what happens in OAuth is:

- Your app directs the user to the OAuth provider with a list of grants (what resources we want access to)
- The user succeeds or fails at the login (if fails then this process stops)
- The provider gives our server an authorization grant
- Our server then requests an access token from the provider
- We can use that access token to request information from the OAuth provider within the scope of our grants

We're going to modify our route to our application:

```
class SessionVar does Cro::HTTP::Auth {
  has $.user         is rw;
  has $.user-id      is rw;
  has $.login-state  is rw;
}

my $client      = HTTP::Tinyish.new;
my $db          = DB::SQLite.new(filename => %?RESOURCES<db.sqlite>.absolute);
my $application = route {
  template-location 'templates/';

  before Cro::HTTP::Session::InMemory[SessionVar].new(
    expiration  => Duration.new(60*60),
    cookie-name => 'demo-app',
  );

  get -> SessionVar \session {
    template 'main.crotmp', {
      user      => session.user,
      logged-in => (session.user-id ?? True !! False),
    };
  }

  get -> SessionVar \session, 'oauth' {
    session.login-state = ('a' .. 'z').pick(24).sort.join('');
    redirect 'https://github.com/login/oauth/authorize'
           ~ "?client_id={encode-percents: config<gh-client>}"
           ~ "&redirect_uri={encode-percents: "http://localhost:8666/oauth2"}"
           ~ "&state={encode-percents: session.login-state}";
  }

  get -> SessionVar \session, 'oauth2', :$state! is query, Str :$code! is query, :$error?
is query {
    redirect '/' if $error.defined;
    my $resp = $client.post(
      headers => {Accept => 'application/json'},
      'https://github.com/login/oauth/access_token'
      ~ "?client_id={encode-percents: config<gh-client> }"
      ~ "&client_secret={encode-percents: config<gh-secret> }"
      ~ "&code={encode-percents: $code }",
    );
    redirect '/' unless 200 <= $resp<status> <= 201;
    redirect '/?error=state-mismatch' unless $state eq session.login-state;

    # Now get the user info so we can create a user
    my $json = from-json($resp<content>);
    $resp = $client.get(
      headers => {Authorization => "token {$json<access_token>}"},
      "https://api.github.com/user",
    );
```

```
$json = from-json($resp<content>);

    # determine if user exists
    my $user = $db.query('select * from users where foreign_id = ?',  $json<id>).hash;
    if ($user<foreign_id>//'') ne $json<id> {
      # create them
      my $id = UUID.new.Str;
      $db.query('insert into users (id, name, foreign_id) values (?, ?, ?);', $id,
$json<name>, $json<id>);
      $user = $db.query('select * from users where id = ?',  $id).hash;
    }

    # if the db failed for whatever reason, don't set the user session
    if ($user<foreign_id>//'') eq $json<id> {
      session.user = $user;
      session.user-id = $user<id>;
    }
    redirect '/';
  }
}
```

Oof, this is a huge change, let's break it down. First big modification is the introduction of a session to our app. We're going to keep our user data in the session so we don't need to load it every request so we can tell cro to hold that in a session object resembling the `SessionVar` class we've designed.

```
# This will hold our session data
class SessionVar does Cro::HTTP::Auth {
  has $.user    is rw;
  has $.user-id is rw;
}
...

  # This tells Cro we want an InMemory session resembling SessionVar
  before Cro::HTTP::Session::InMemory[SessionVar].new(
    expiration  => Duration.new(60*60),
    cookie-name => 'demo-app',
  );
```

Now we need to set up our redirect. This could also be done in a link rather than as an endpoint for our API but this is also fine.

```
  get -> SessionVar \session, 'oauth' {
    session.login-state = ('a' .. 'z').pick(24).sort.join('');
    redirect 'https://github.com/login/oauth/authorize'
           ~ "?client_id={encode-percents: config<gh-client>}"
           ~ "&redirect_uri={encode-percents: "http://localhost:8666/oauth2"}"  # take
note of this url, this is where we'll handle the login
           ~ "&state={encode-percents: session.login-state}";
  }
```

Now, if you were to fire up your server and have your github OAuth settings correct, you could hit `/oauth` and you'd be presented with the nice github authorization screen.

Now, once a user accepts or rejects the login request, we need to handle it:

```
  get -> SessionVar \session, 'oauth2', :$state! is query, Str :$code! is query, :$error?
is query {
    # If the user declined oauth, redirect home
    redirect '/' if $error.defined;

    # Here we're cashing in the code we got back from our redirect so that we can
    # later request information about the user.
```

```
my $resp = $client.post(
      headers => {Accept => 'application/json'},
      'https://github.com/login/oauth/access_token'
      ~ "?client_id={encode-percents: config<gh-client> }"
      ~ "&client_secret={encode-percents: config<gh-secret> }"
      ~ "&code={encode-percents: $code }",
    );

    # Go home if this code exchange failed.
    redirect '/' unless 200 <= $resp<status> <= 201;
    # Go home if what we go back from the server is not what we expected
    redirect '/?error=state-mismatch' unless $state eq session.login-state;

    # Now get the user info so we can create a user
    my $json = from-json($resp<content>);
    $resp = $client.get(
      headers => {Authorization => "token {$json<access_token>}"},
      "https://api.github.com/user",
    );
    $json = from-json($resp<content>);

    # determine if user exists
    my $user = $db.query('select * from users where foreign_id = ?',  $json<id>).hash;
    if ($user<foreign_id>//'') ne $json<id> {
      # create them
      my $id = UUID.new.Str;
      $db.query('insert into users (id, name, foreign_id) values (?, ?, ?);', $id,
$json<name>, $json<id>);
      $user = $db.query('select * from users where id = ?',  $id).hash;
    }

    # if the db failed for whatever reason, don't set the user session
    if ($user<foreign_id>//'') eq $json<id> {
      session.user = $user;
      session.user-id = $user<id>;
    }
    redirect '/';
  }
```

Along with all this functionality, we also need to update our dependencies and project meta. The new `use` block:

```
use Cro::HTTP::Router;
use Cro::HTTP::Server;
use Cro::HTTP::Session::InMemory;
use Cro::WebApp::Template;
use Cro::Uri :encode-percents;
use HTTP::Tinyish;
use JSON::Fast;
use DB::SQLite;
use UUID;
```

and our new `META6.json` :

```
{
  "name":        "Demo",
  "auth":        "zef:tony-o",
  "version":     "0.0.1",
  "api":         "0",
  "description": "A demo cro app",
  "depends":     ["cro", "Cro::WebApp::Template", "HTTP::Tinyish", "DB::SQLite", "UUID"],
  "provides":    {
    "Demo::App": "lib/Demo/App.rakumod",
    "Config":    "lib/Config.rakumod"
```

```
},
  "resources":  [ "db.sqlite" ]
}
```

Now, last thing before we can test. Our `/` route has changed to use our session rather than just lounging around. Let's update our template to take advantage of our new powers.

```
<!DOCTYPE html>
<html>
  <head />
  <body>
    <!.logged-in> <!-- if we're not logged in !-->
      <a href="/oauth">Login</a>
    </!>
    <?.logged-in> <!-- if we're logged in !-->
      Welcome, <.user.name>
    </?>
  </body>
</html>
```

Fire that bad boy up with `raku -I. bin/app` and now when you first visit your site you'll get a `login` link and if you decline then you'll just be shown the login link, if you choose to accept fate then you should see a nice little display of your github name for the subsequent requests.

At this point, if you were doing this in an effort to build something robust, you'd likely start look at real SQL servers and the available connection pools for them so you don't have one DB reference. You'd also probably start looking at session backends that are more persistent.

Members Only, Public Pages, and a Mix of Both

Now all that's left is to make a couple of pages.

- Create a new topic (only available if logged in)
- View a topic
- * List hierarchy of replies
- * Show respond form if logged in

First let's show a link to make a post only if the user is logged in. In `templates/main.crotmp` :

```
<!DOCTYPE html>
<html>
  <head />
  <body>
    <!.logged-in>
      <a href="/oauth">Login</a>
    </!>
    <?.logged-in>
      Welcome, <.user.name> | <a href="/post">New Post</a>
    </?>
  </body>
</html>
```

The new route (in `lib/Demo/App.rakumod`):

```
  # Anything requiring this subset will not match unless their session `user-id` is
defined
```

```
subset LoggedIn of SessionVar where *.user-id ?? True !! False;

  # The actual `create post` route
  get -> LoggedIn \session, 'post' {
    # This is necessary because we provide no other handlers if the user isn't logged in
    redirect '/oauth' unless session.user-id;

    template 'post.crotmp', { title => '', body => '', errors => [] };
  }

  # Handle the POST request to make the post
  post -> LoggedIn \session, 'post' {
    # This is necessary because we provide no other handlers if the user isn't logged in
    redirect '/oauth' unless session.user-id;

    request-body -> (:$title, :$body) {
      # Handle all of the errors so the user isn't clicking back and forth to fix every
single
      # issue.
      my @errors;
      @errors.push('Title cannot be fewer than five characters') if $title.chars <= 5;
      @errors.push('Post cannot be fewer than fifty characters') if $body.chars <= 50;

      if @errors {
        template 'post.crotmp', {
          :$title,
          :$body,
          :@errors,
        };
        last;
      }


      my $id = UUID.new.Str;
      my $ok = $db.query('insert into posts (id, title, body, author) values (?, ?, ?,
?);', $id, $title, $body, session.user-id);
      if $ok {
        # The post was successful, let's show the user
        redirect "/view/{$id}";
      } else {
        # The insert failed, let's show the user
        @errors.push('An error occurred while saving your post, please try again later');
        template 'post.crotmp', { :$title, :$body, :@errors };
      }
    }
  }
```

Okay, now we have a working endpoint. When you fire up the app you can log in and you can create posts. So, what's left for this tiny demo? Let's make the `/view/:post-id` endpoint so we can see our posts. We'll take the format of the old bulletin boards where we show the body and then a threaded view of responses below it.

Hierarchal Post View

First let's get the easy part out of the way, in `templates/view.crotmp` :

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="/css/demo.css" />
  </head>
  <body>
    <h2><.post.title></h2>
```

```
<pre>
      <.post.body>
    </pre>
    <@.levels:$l>
      <div>
        <&HTML($l.level)>&gt; <a href="/view/<$l.id>"><$l.title></a>
      </div>
    </@>
    <!-- if we're logged in then show a response form! !-->
    <?.logged-in>
      <br/>
      <br/>
      <@.response.errors: $err>
        <div>Error: <$err></div>
      </@>
      <form method="POST" action="/view/<.post.id>">
        <div><input type="text" name="title" placeholder="Response Title"
value="<.response.title>" /></div>
        <div><textarea name="body" cols="40" rows="20"><.response.body></textarea></div>
        <div><input type="submit" value="Post" /></div>
      </form>
    </?>
  </body>
</html>
```

Cool. Now if you're logged in you'll see a form to respond to whatever post you're viewing. We're wrapping our post body in `pre` because we're kicking it old school for this post. Let's add our routes in `lib/Demo/App.rakumod` :

```
  get -> SessionVar \session, 'view', $id {
    # Build-post-hierarchy is doing some fun stuff to make our template code simple, more
after the break
    my $data = build-post-hierarchy($id);
    if !$data {
      # If the post doesn't exist, go home
      redirect '/';
    } else {
      template 'view.crotmp', {
        :post($data<post>),
        :levels($data<levels>),
        response => {title => '', body => '', parent => $id, errors => [] },
        logged-in => (session.user-id ?? True !! False),
      };
    }
  }

  post -> SessionVar \session, 'view', $id {
    # Build-post-hierarchy is doing some fun stuff to make our template code simple, more
after the break
    my $data = build-post-hierarchy($id);
    if !$data {
      redirect '/';
    } else {
      request-body -> (:$title, :$body) {
        my @errors;
        @errors.push('Title cannot be fewer than five characters') if $title.chars <= 5;
        @errors.push('Post cannot be fewer than fifty characters') if $body.chars <= 50;

        if @errors {
          template 'view.crotmp', {
            :post($data<post>),
            :levels($data<levels>),
            response => {title => $title, body => $body, parent => $id, errors => [] },
            logged-in => (session.user-id ?? True !! False),
            :@errors,
```

```
};
        } else {
          my $rid = UUID.new.Str;
          my $ok = $db.query('insert into posts (id, title, body, author, parent) values
(?, ?, ?, ?, ?);', $rid, $title, $body, session.user-id, $id);
          if $ok {
            # see-other is letting Cro know we need to do a `GET` request, otherwise
we'll get stuck in an infinite loop
            redirect :see-other, "/view/{$rid}";
          } else {
            @errors.push('An error occurred while saving your post, please try again
later');
            template 'view.crotmp', {
              :post($data<post>),
              :levels($data<levels>),
              response => {title => $title, body => $body, parent => $id, errors =>
@errors },
              logged-in => (session.user-id ?? True !! False),
              :@errors,
            };
          }
        }
      }
    }
  }
```

The `POST` route is very similar to the `post` route of the same method/name. It's doing essentially the same thing but adding a parent id to the post so that when we build the hierarchy we know exactly what to do. Let's take a look at that hierarchy builder (in `lib/Demo/App.rakumod`):

```
# This sub is declared outside of our routes so we can use it in the GET/POST version
sub build-post-hierarchy(Str:D $id) {
  # Ensure the post exists prior to building the hierarchy
  my %post    = $db.query('select p.id, p.title, p.body, u.name from posts p left join
users u on u.id = p.author where p.id = ?;', $id).hash;

  # This query looks a little complicated and requires SQLite > 3.8.2
  # In `threads.*` we're following our top level to gather all nested sets of responses
  # In `threads.*` the query of `parents.*` is following whatever thread we're viewing
until
  # we find a post with parent = null and this is our top level post
  my $sql     = q:to/EOSQL/;
with recursive threads(level,id,title,parent) as (
        select 0, id, title, parent from posts where id = (
          with recursive parents(id,parent) as (
        select id, parent from posts where id = ?
        union all
        select p.id, p.parent from posts p, parents where p.id = parents.parent
          ) select id from parents where parent is null
        )
        union all
        select level+1, p.id, p.title, p.parent from posts p, threads where p.parent =
threads.id
) select * from threads order by level, parent;
EOSQL

  # Return the problem to the handler
  if (%post<id>//'') ne $id {
    return Nil
  }

  # This code makes it easy for templates to consume the posts as an ordered hierarchy
  # so that the template may just iterate an array
```

```
my @threads = $db.query($sql, $id).hashes;
  my %indexes;
  my @levels;
  for @threads -> $t {
    if %indexes{$t<parent>//''} {
      @levels.splice(%indexes{$t<parent>}, 0, { id => $t<id>, title => $t<title>, (level
=> ' ' x $t<level> * 4) });
      for %indexes.keys -> $k {
        %indexes{$k}++ if %indexes{$t<parent>} < %indexes{$k};
      }
      %indexes{$t<id>} = %indexes{$t<parent>} + 1;
    } else {
      @levels.push({ id => $t<id>, title => $t<title>, level => (' ' x $t<level> *
4) });
      %indexes{$t<id>} = @levels.elems;
    }
  }
  return {:%post, :@levels};
}
```

Whew. That's definitely a mouth full. Likely your use cases will be more straightforward but easy tutorials usually leave out some crucial bit and this is intended to show some complication.

All that's left at this point is making the homepage show top level posts so let's modify our home page's route (in `lib/Demo/App.rakumod`):

```
  get -> SessionVar \session {
    template 'main.crotmp', {
      user      => session.user,
      # Add posts !
      posts     => $db.query('select id, title from posts where parent is null').hashes,
      logged-in => (session.user-id ?? True !! False),
    };
  }
```

Our template to match (`templates/main.crotmp`):

```
<!DOCTYPE html>
<html>
  <head />
  <body>
    <div>
    <!.logged-in>
      <a href="/oauth">Login</a>
    </!>
    <?.logged-in>
      Welcome, <.user.name> | <a href="/post">New Post</a>
    </?>
    </div>
    <h2>Posts</h2>
    <@.posts:$p>
    <div>
      ° <a href="/view/<$p.id>"><$p.title></a>
    </@>
  </body>
</html>
```

Okay. That's it! You have a working cro app (or should)! Let's review what we've done in this app:

- Visitors can log in via OAuth

- Anyone can view posts/responses
- Registered users can create new posts and respond

Seems like a lot of code for these three basic functions (and it is) but bulletin boards can get even more complicated very quickly.

This is part one of building a cro application end to end including deployment and next week the topic will be configuring apache to reverse proxy to our cro app and configuring systemd to ensure our app is available.

Note: the full code for this demo can be found here

Part 2

Building a Cro App - Part B building-a-cro-app-part-b rakudo, programming, systems

We have our Cro app built, authenticating, and doing all kinds of fun stuff. Let's take a look at deploying this bad wamma jamma!

To do so we'll do the follow:

- Create a systemd service file
- Configure apache2 to serve our app
- Set up SSL with apache2 for our service (with Let's Encrypt)

Systemd

This section is fairly straight forward but it requires some consideration:

- Systemd is not loading your user's environment
- Systemd is running as a different user than your own so your app root must have the right file permissions, by default this user is `root`

For the purpose of this article we're just going to assume you've put your app root in `/opt/my-app` , you have a working rakudo, that you have all of the dependencies for your app installed, and that the app is runnable on the machine you're installing the systemd file on.

In your services file `/etc/systemd/system/my-app.service` :

```
[Unit]
Description=My Cro App - A Tutorial
DefaultDependencies=no
After=network.target

[Service]
Type=simple
WorkingDirectory=/opt/my-app
ExecStart=/usr/bin/raku -I. /opt/my-app/bin/app
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

This is the most basic systemd file you can make, there are a lot of other options you can put in here including things like a restart backoff if the app is not starting. It is important to note that you should use the full path to binaries in the service file, make no assumptions, and take frequent breaks.

Now, run `systemctl daemon-reload` so that systemd refreshes its cache, enable our service with `systemctl enable my-app`, and finally start the service with `systemctl start my-app`.

You should now be able to hit port `8666` (if you haven't modified anything from part 1 or cloned the repo). You can test this by running `curl localhost:8666` - you should get a response from our app.

If you're having issues and think that the app is not running, you can use `journalctl -u my-app.service` to see the logs and correct any errors the app is having getting started.

That's all there really is in a basic systemd service. If you'd like to delve into service files more, this is a hand resource. Onto to the tougher stuff.

Configuring Apache

Getting the certificates with `certbot` prior to setting up apache is much easier but it's a headache when you have to renew the certs so we're going to make this work with apache running so there's no downtime when you renew.

For this tutorial we're looking at Debian 11 so some of your install commands might be a bit different but installing apache2:

```
# apt install apache2
...
# a2enmod rewrite
...
# a2enmod proxy
...
# a2enmod proxy_http
...
# systemctl restart apache2
...
```

After getting apache installed we need to configure it to respond to our domain, so in your favorite version of vim load up `/etc/apache2/sites-available/000-default.conf` and make it look exactly like:

```
<VirtualHost *:80>
 ServerName pm6.dev
 ServerAlias www.pm6.dev
 DocumentRoot /var/www

 RewriteEngine On
 RewriteCond %{HTTPS} off
 RewriteCond %{REQUEST_URI} !^/\.well-known/
 RewriteRule (.*) https://www.pm6.dev/$1 [R,L]
</VirtualHost>

<VirtualHost *:443>
 ServerName pm6.dev
 ServerAlias www.pm6.dev
 ProxyRequests     Off
 ProxyPreserveHost On

 ErrorLog ${APACHE_LOG_DIR}/error.log
 CustomLog ${APACHE_LOG_DIR}/access.log combined

 <Proxy *>
  Order deny,allow
  Allow from all
 </Proxy>
 ProxyPass         /  http://localhost:8666/ nocanon
 ProxyPassReverse  /  http://localhost:8666/
</VirtualHost>
```

Now we're ready to get certbot involved. The configuration above is using the hostname `pm6.dev` , you should replace that with whatever hostname you're looking to use. The other thing it does is redirects standard `http` requests to `https` *unless* the request is from certbot.

```
# apt install certbot
...
```

Now we're ready to set up the certificates, grab that shiny domain name and replace `{EMAIL}` and `{DOMAIN}` with your own!

```
# systemctl restart apache2
...
# certbot certonly --webroot -w '/var/www/' -d 'www.{DOMAIN}' -d '{DOMAIN}' -n --email
'{EMAIL}' --agree-tos
```

So far we're looking fresh, if certbot succeeded then we're set up for a hands free renewal in the future. Let's put the certs to use by editing our apache2 config again in notepad++:

```
# ...
<VirtualHost *:443>
   ServerName pm6.dev
   ServerAlias www.pm6.dev
   ProxyRequests     Off
   ProxyPreserveHost On

   SSLCertificateFile /etc/letsencrypt/live/www.pm6.dev/fullchain.pem
   SSLCertificateKeyFile /etc/letsencrypt/live/www.pm6.dev/privkey.pem

   ErrorLog ${APACHE_LOG_DIR}/error.log
   CustomLog ${APACHE_LOG_DIR}/access.log combined

# ...
```

The two new lines are the ones starting with `SSLCertificate` . Enable `ssl` on apache2 and then restart apache2:

```
# a2enmod ssl
...
# systemctl restart apache2
```

Voila! That's it. That's all there is to it. If apache complains here then correct any errors and then restart this paragraph.