Extracted from:

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Chris McCord,
Bruce Tate,
and José Valim

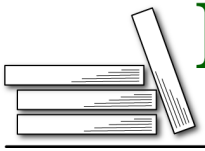
edited by Jacquelyn Carter

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Chris McCord
Bruce Tate
José Valim

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-226-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2019

Building a Feature

Our first feature won't be complicated. It'll print a string when you load a specific URL. To build that feature, we're going to use a small fraction of the files that mix phx.new created. Don't worry. You'll get a tour of the whole tree a little later. For now, everything we need is in the lib/hello_web subdirectory. We'll edit router.ex to point a URL to our code. We'll also add a controller to the lib/hello_web/controllers subdirectory, a view to lib/hello_web/views, and a template to lib/hello_web/templates.

First things first. We want to map requests coming in to a specific URL to the code that satisfies our request. We'll tie a URL to a function on a controller, and that function to a view. You'll do so in the routing layer, as you would for other web frameworks. Routes in Phoenix go in lib/hello_web/router.ex by default. The .ex extension is for compiled Elixir files. Take a look at that file now. Scroll to the bottom, and you'll find a block that looks like this:

getting_started/listings/hello/lib/hello_web/router.ex

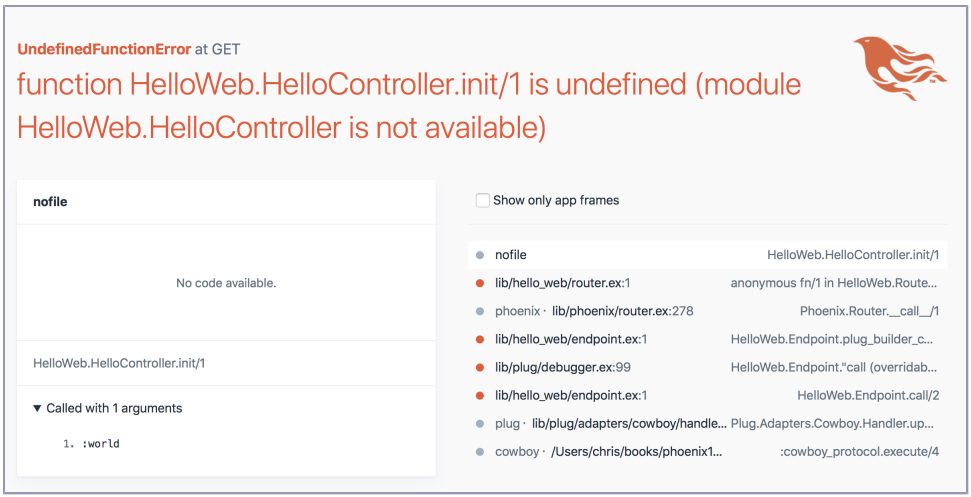
```
scope "/", HelloWeb do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
end
```

You can see a block of requests, scoped to /. That means that this group of routes will attempt to match all routes beginning with /. The pipe_through :browser macro handles some housekeeping for all common browser-style requests. You can see one route that takes requests that look like / and sends them to the :index action on the PageController. This looks like the right place to add our route. Add the following route *above the existing route*:

```
get "/hello", HelloController, :world
get "/", PageController, :index
```

This new code will match routes starting with /hello and send them to the :world function on the HelloController module. If you'd like, you can point your browser to localhost:4000/hello, but you'll get an error page because our controller module doesn't exist yet:



Before moving forward, let's briefly review the error page. At the top of the page we get the exception name: UndefinedFunctionError. Next, we see the error message. It seems HelloWeb.HelloController.init, which expects one argument, is undefined because the module does not exist. That's a good start and you can also see the detailed code related to the error.

The lefthand pane will usually show the relevant code snippets. Because the module in this case does not exist, there is no code snippet loaded by default, but you can populate this pane by clicking any of the stacktrace entries on the righthand side. In the stack trace, orange dots denote calls within the application and gray ones identify dependency code. Finally the bottom of the page has general request information, such as request headers, cookies, session, and the like.

Let's fix that error now. All controllers in Phoenix are in lib/hello_web/controllers. Create a lib/hello_web/controllers/hello_controller.ex file that looks like this:

getting_started/listings/hello/lib/hello_web/controllers/hello_controller.ex

```
defmodule HelloWeb.HelloController do
  use HelloWeb, :controller

  def world(conn, _params) do
    render(conn, "world.html")
  end
end
```

This controller is simple. If you're new to Elixir, you'll often see use SomeModule to introduce specific functionality to a module. The use HelloWeb, :controller call prepares us to use the Phoenix Controller API, including making some functions available that we'll want to use later. The router will call the world action on our controller, passing all of the information we need. We call the functions invoked by the router on our controller's actions, but don't get confused. They're just functions.

Once again, you might point your browser to localhost:4000/hello, but you'd find that it's still not working. We have yet to create our view, so Phoenix reports:

```
undefined function: HelloWeb.HelloView.render/2
    (module HelloWeb.HelloView is not available)
```

That makes sense. Let's easily fix that problem by creating a view called lib/hello_web/views/hello_view.ex with the following contents:

getting_started/listings/hello/lib/hello_web/views/hello_view.ex

```
defmodule HelloWeb.HelloView do
  use HelloWeb, :view
end
```

That file doesn't actually do any work beyond tying the view for world with some code to render a template. We'll rely on the defaults to render a template, which doesn't yet exist. One more time, you see an error when you point your browser to localhost:4000/hello:

```
Could not render "world.html" for HelloWeb.HelloView, please define a matching
clause for render/2 or define a template at "lib/hello_web/templates/hello".
  No templates were compiled for this module.
```

We are getting closer. Create the following template at lib/hello_web/templates/hello/world.html.eex, and we're done:

getting_started/listings/hello/lib/hello_web/templates/hello/world.html.eex

```
<h1>From template: Hello world!</h1>
```

As soon as you save your code, notice that the web page reloads! We have live reloading enabled, so whenever we touch templates or template assets, you'll see an automatic page reload.

The .eex extension denotes a template, which Phoenix will compile into a function. If you look closely, you can see the page we loaded has a header. We're implicitly using the layout defined in the lib/hello_web/views/layout_view.ex view and the template defined in lib/hello_web/templates/layout/app.html.eex. We'll work more with views a little later. For now, it's enough for you to know it's there.

Enjoy the results. It's not a fully operational death star, but you're well on your way.

Using Routes and Params

Right now, there's no dynamic information in our route, and we don't need any *yet*, but later we'll need to grab dynamic data from the URL to look up data from our database. Let's use our sandbox to see how that works. We'll use dynamic routes closely with Elixir's pattern matching. First, let's revise our route. Replace the first route in lib/hello_web/router.ex with this one:

```
get "/hello/:name", HelloController, :world
```

Notice that we're matching a URL pattern—/hello, as before—but we also add /:name to the route. The : tells Phoenix to create a parameter called :name in our route and pass that name as a parameter to the controller. Change the world function on lib/hello_web/controllers/hello_controller.ex to look like this:

```
def world(conn, %{"name" => name}) do
  render(conn, "world.html", name: name)
end
```

Since it's the first time we're using the shorthand hash notation, we'll give it a brief introduction. The code name: name is shorthand for :name => name. They are both shorthand notations for representing key-value pairs. [name: name] is shorthand for [{:name, name}]. Finally, since name: name is the last argument, of a function, we can omit the brackets. That means render(conn, "world.html", name: name) is shorthand for render(conn, "world.html", [name: name]). Whew. Now, on to the code.

Our new action uses the second argument, which is a map of inbound parameters. We match to capture the name key in the name variable, and pass the result to render in a keyword list. If you're new to Elixir, that function header looks a little different from what you might have seen before. Something special is happening, so let's look at it in a little more detail. If you already understand pattern matching, you can skip to the next section.

Pattern Matching in Functions

The Elixir language has an excellent feature called *pattern matching*. When Elixir encounters a = operator, it means "make the thing on the left match the thing on the right." You can use this feature in two different ways: to take data structures apart, or to test. Let's look at an example. Open up interactive Elixir by typing iex in your OS shell and follow this script:

```
iex> {first, second, third} = {:lions, :tigers, :bears}
{:lions, :tigers, :bears}

iex> first
:lions

iex> {first, second, :bears} = {:lions, :tigers, :bears}
{:lions, :tigers, :bears}

iex> {first, second, :armadillos} = {:lions, :tigers, :bears}
** (MatchError) no match of right hand side value: {:lions, :tigers, :bears}
```

In the first statement, we're matching a 3-tuple to {:lions, :tigers, :bears}. Elixir tries to make the expression on the left match, and it can do so by assigning first to :lions, and second to :tigers. In this case, we're using the pattern match to pick off pieces of the inside of the data structure.

In the third or fourth statement, we're doing something different. We're matching to do a test. When the interpreter tries to match the two, it succeeds and passes on, or fails and throws an exception.

You can also use pattern-matching syntax within your function heads in both of these ways. Type the following into your console:

```
iex> austin = %{city: "Austin", state: "Tx"}
%{city: "Austin", state: "Tx"}

iex> defmodule Place do
...>   def city(%{city: city}), do: city
...>   def texas?(%{state: "Tx"}), do: true
...>   def texas?(_), do: false
...> end
```

This module uses pattern matching in two different ways. The first function uses pattern matching to destructure the data, or take it apart. We use it to extract the city. It grabs the value for the :city key from any map. Although this bit of destructuring is trivial, sometimes the data structures can be deep, and you can reach in and grab the attributes you need with a surgeon's precision.

The second function, texas?, is using a pattern match as a test. If the inbound map has a :state keyword that's set to Tx, it'll match. Otherwise, it'll fall through to the next function, returning false. If we wanted to, we could:

- Match all maps with a given key, as in has_state?(%{state: _}), where the underscore _ will match anything
- Use strings as keys instead of atoms, as in has_state?(%{"state" => "Tx"})
- Match a state, and assign the whole map to a variable, as in has_state?(%{"state" => "Tx"} = place)

The point is, pattern matching is a huge part of Elixir and Phoenix programming. We'll use it to grab only certain types of connections, and also to grab individual pieces of the connection, conveniently within the function heading.

With all of that in mind, let's look at our controller action again:

```
def world(conn, %{"name" => name}) do
  render(conn, "world.html", name: name)
end
```

That makes more sense now. We're grabbing the name field from the second argument, which contains the inbound parameters. Our controller then renders the world.html template, passing in the local data. The local data prepares a map of variables for use by the templates. Now our views can access the name variable we've specified.

Chris says:

Atom Keys vs. String Keys?

In the world action in our controllers, the external parameters have string keys, "name" => name, while internally we use name: name. That's a convention followed throughout Phoenix. External data can't safely be converted to atoms, because the atom table isn't garbage-collected. Instead, we explicitly match on the string keys, and then our application boundaries like controllers and channels will convert them into atom keys, which we'll rely on everywhere else inside Phoenix.

Using Assigns in Templates

Now, all that remains is to tweak our template in lib/hello_web/templates/hello/world.html.eex to make use of the value. You can access the name specified in the world action as @name, like this:

```
<h1>Hello <%= String.capitalize(@name) %>!</h1>
```

The <%= %> brackets surround the code we want to substitute into the rendered page. @name will have the value of the :name option that we passed to render. We've worked for this reward, so point your browser to localhost:4000/hello/phoenix. It's ALIVE!

We've done a lot in a short time. Some of this plumbing might seem like magic to you, but you'll find that Phoenix is marvelously explicit, so it's easy to understand exactly what's happening, when, for each request. It's time to make this magic more tangible.

Going Deeper: The Request Pipeline

When we created the `hello` project, Mix created a bunch of directories and files. It's time to take a more detailed look at what all of those files do and, by extension, how Phoenix helps you organize applications.

When you think about it, typical web applications are just big functions. Each web request is a function call taking a single formatted string—the URL—as an argument. That function returns a response that's nothing more than a formatted string. If you look at your application in this way, your goal is to understand how functions are composed to make the one big function call that handles each request. In some web frameworks, that task is easier said than done. Most frameworks have hidden functions that are only exposed to those with deep, intimate internal knowledge.

The Phoenix experience is different because it encourages breaking big functions down into smaller ones. Then, it provides a place to explicitly register each smaller function in a way that's easy to understand and replace. We'll tie all of these functions together with the Plug library.

Think of the Plug library as a specification for building applications that connect to the web. Each plug consumes and produces a common data structure called `Plug.Conn`. Remember, that struct represents *the whole universe for a given request*, because it has things that web applications need: the inbound request, the protocol, the parsed parameters, and so on.

Think of each individual plug as a function that takes a `conn`, does something small, and returns a slightly changed `conn`. The web server provides the initial data for our request, and then Phoenix calls one plug after another. Each plug can transform the `conn` in some small way until you eventually send a response back to the user.

Even responses are just transformations on the connection. When you hear words like *request* and *response*, you might be tempted to think that a request is a plug function call, and a response is the return value. That's not what happens. A response is just one more action on the `connection`, like this:

```
conn
|> ...
|> render_response()
```

The whole Phoenix framework is made up of organizing functions that do something small to connections, *even rendering the result*. Said another way…

Plugs are functions.

Your web applications are pipelines of plugs.