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

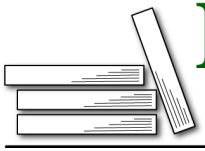
edited by Jacquelyn Carter

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

The Anatomy of a Plug

Before we build our plug, let's take a deep dive into the Plug library and learn how plugs work from the inside. There are two kinds of plugs: *module plugs* and *function plugs*. A function plug is a single function. A module plug is a module that provides two functions with some configuration details. Either way, they work the same.

We have seen both kinds of plugs in use. From the endpoint module in lib/rumbl_web/endpoint.ex, you can see an example of a module plug:

```
plug Plug.RequestId
```

You specify a module plug by providing the module name. In the router, you can see an example of a function plug:

```
plug :protect_from_forgery
```

You specify a function plug with the name of the function as an atom. Because a module is just a collection of functions, it strengthens the idea that plugs are just functions.

For our first plug, we'll write a module plug that encapsulates all the authentication logic in one place.

Module Plugs

Sometimes you might want to share a plug across more than one module. In that case, you can use a module plug. To satisfy the Plug specification, a module plug must have two functions, named init and call.

The simplest possible module plug returns the given options on init and the given connection on call. This plug does nothing:

```
defmodule NothingPlug do
  def init(opts) do
    opts
  end

  def call(conn, _opts) do
    conn
  end
end
```

Remember, a typical plug transforms a connection. The main work of a module plug happens in call. In our NothingPlug, we simply pass the connection through without changes. The call will happen *at runtime*.

Sometimes, you might want to let the programmer change the behavior of a plug. We can do that work in the second argument to call, options. In our NothingPlug, we don't need any more information to do our job, so we ignore the options.

Sometimes, you might need Phoenix to do some heavy lifting to transform options. That's the job of the init function. Plug uses the result of init as the second argument to call. In development mode, Phoenix calls init at runtime, but in production mode, init is called only once, at *compile time*. This strategy makes init the perfect place to validate and transform options without slowing down every request so call can be as fast as possible. Since call is the workhorse of Plug, we want it to do as little work as possible.

For both module and function plugs, the request interface is the same. conn, the first argument, is the data we pass through every plug. It has the details for any request, and we morph it in tiny steps until we eventually send a response. All plugs take a conn and return a conn.

You'll see piped functions using a common data structure over and over in Elixir. The trick that makes this tactic work is having the right common data structure. Since Plug works with web APIs, our data structure will specify the typical details of the web server's domain.

In Phoenix, you'll see connections, usually abbreviated conn, literally everywhere. At the end of the day, the conn is only a Plug.Conn struct, and it forms the foundation for Plug.

Plug.Conn Fields

You can find great online documentation for Plug.Conn.[3] This structure has the various fields that web applications need to understand about web requests and responses. Let's look at some of the supported fields.

Request fields contain information about the inbound request. They're parsed by the adapter for the web server you're using. Cowboy is the default web server that Phoenix uses, but you can also choose to plug in your own. These fields contain strings, except where otherwise specified:

host

The requested host. For example, www.pragprog.com.

method

The request method. For example, GET or POST.

3. http://hexdocs.pm/plug/Plug.Conn.html

path_info
: The path, split into a List of segments. For example, ["admin", "users"].

req_headers
: A list of request headers. For example, [{"content-type", "text/plain"}].

scheme
: The request protocol as an atom. For example, :https.

You can get other information as well, such as the query string, the remote IP address, the port, and the like. For Phoenix, if a web request's information is available from the web server's adapter, it's in Plug.Conn.

Next comes a set of *fetchable fields*. A fetchable field is empty until you explicitly request it. These fields require a little time to process, so they're left out of the connection by default until you want to explicitly fetch them:

cookies
: These are the request cookies with the response cookies.

params
: These are the request parameters. Some plugs help to parse these parameters from the query string, or from the request body.

Next are a series of fields that are used to process web requests and keep information about the plug pipeline. Here are some of the fields you'll encounter:

assigns
: This user-defined map contains anything you want to put in it. For instance, this is where we will keep the authenticated user for the current request.

halted
: Sometimes a connection must be halted, such as a failed authorization. In this case, the halting plug sets this flag.

You can also find a secret_key_base for everything related to encryption.

Since the Plug framework handles the whole life cycle of a request, including both the request and the response, Plug.Conn provides fields for the response:

resp_body
: Initially an empty string, the response body will contain the HTTP response string when it's available.

resp_cookies
: The resp_cookies has the outbound cookies for the response.

resp_headers
: These headers follow the HTTP specification and contain information such as the response type and caching rules.

status
: The response code generally contains 200–299 for success, 300–399 for redirects, 400–499 for bad client requests such as not-found, and 500+ for server errors.

Finally, Plug supports some private fields reserved for the adapter and frameworks:

adapter
: Information about the underlying web server is stored here.

private
: This field has a map for the private use of frameworks.

Initially, a conn comes in almost blank and is filled out progressively by different plugs in the pipeline. For example, the endpoint may parse parameters, and the application developer will set fields primarily in assigns. Functions that render set the response fields such as status, change the state, and so on.

Plug.Conn also defines many functions that directly manipulate those fields, which makes abstracting the work of doing more complex operations such as managing cookies or sending files straightforward.

Now that you have a little more knowledge, we're ready to transform the connection by writing our first plug.

Writing an Authentication Plug

The authentication process works in two stages. First, we'll store the user ID in the session every time a new user registers or a user logs in. Second, we'll check if there's a new user in the session and store it in conn.assigns for every incoming request, so it can be accessed in our controllers and views. Let's start with the second part because it's a little easier to follow.

Create a file called lib/rumbl_web/controllers/auth.ex that looks like this:

authentication/listings/rumbl/lib/rumbl_web/controllers/auth.ex

```
defmodule RumblWeb.Auth do
  import Plug.Conn

  def init(opts), do: opts

  def call(conn, _opts) do
    user_id = get_session(conn, :user_id)
    user = user_id && Rumbl.Accounts.get_user(user_id)
    assign(conn, :current_user, user)
  end
end
```

Don't let the init function throw you off. It's just a simple function to allow compile time options. Plugs allow data to flow through an application at *run time* through the context. Without init, our plug can't accept any *compile time* options.

call checks if a :user_id is stored in the session. If one exists, we look it up and assign the result in the connection. assign is a function imported from Plug.Conn that slightly transforms the connection—in this case, storing the user (or nil) in conn.assigns. That way, the :current_user will be available in all downstream functions including controllers and views.

Let's add our plug to the router, at the end of the browser pipeline:

authentication/listings/rumbl/lib/rumbl_web/router.change1.ex

```
pipeline :browser do
  plug :accepts, ["html"]
  plug :fetch_session
  plug :fetch_flash
  plug :protect_from_forgery
  plug :put_secure_browser_headers
  plug RumblWeb.Auth
end
```

With our plug in place, we can begin to use this information downstream.

Restricting Access

The RumblWeb.Auth plug processes the request information and transforms the conn, adding :current_user to conn.assigns. Now, downstream plugs can use it to find out if a user is logged in.

We'll use this information to restrict access to pages where we list or show user information. Specifically, we don't want to allow users to access the :index and :show actions of RumblWeb.UserController unless they're logged in.

Open up RumblWeb.UserController and add the following function:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change2.ex

```
defp authenticate(conn) do
  if conn.assigns.current_user do
    conn
  else
    conn
    |> put_flash(:error, "You must be logged in to access that page")
    |> redirect(to: Routes.page_path(conn, :index))
    |> halt()
  end
end
```

If there's a current user, we return the connection unchanged. Otherwise we store a flash message and redirect back to our application root. We use halt(conn) to stop any downstream transformations.

Let's invoke the authenticate function from index to try it out:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change2.ex

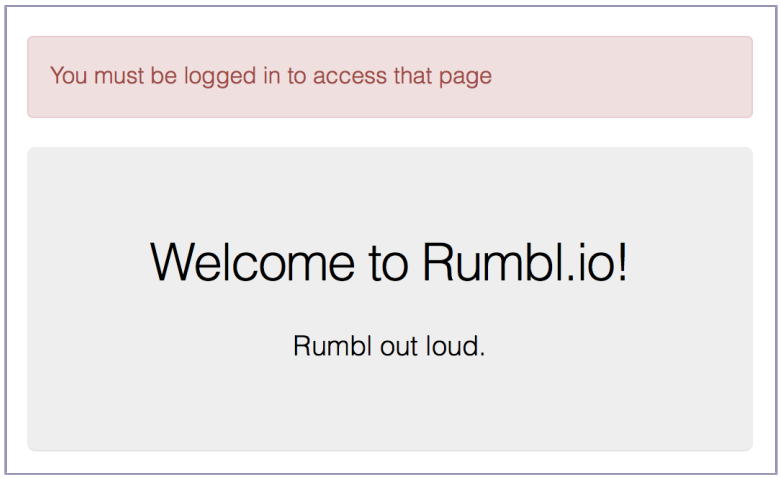
```
def index(conn, _params) do
  case authenticate(conn) do
    %Plug.Conn{halted: true} = conn ->
      conn

    conn ->
      users = Accounts.list_users()
      render(conn, "index.html", users: users)
  end
end
```

Now visit http://localhost:4000/users, where we're redirected back to the root with a message telling us to log in, as shown in the figure on page 11.

We could make the same changes to the show action, invoking our plug and honoring halt. And we could do the same thing every time we require authentication. We'd also have code that's repetitive, ugly, and error prone. We need to *plug* the authenticate function for the actions to be protected. Let's do that.

Like endpoints and routers, controllers also have their own plug pipeline. Each plug in the controller pipeline is executed in order, before the action is invoked. The controller pipeline lets us explicitly choose which actions fire any given plug.

To plug the authenticate function, we must first make it a function plug. A function plug is any function that receives two arguments—the connection



and a set of options—and returns the connection. With a minor tweak, we can satisfy that contract. You need only add an options variable, which you'll ignore:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change3.ex

```
defp authenticate(conn, _opts) do
  if conn.assigns.current_user do
    conn
  else
    conn
    |> put_flash(:error, "You must be logged in to access that page")
    |> redirect(to: Routes.page_path(conn, :index))
    |> halt()
  end
end
```

Now let's plug it in our controller, right after alias Rumbl.Accounts.User:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change3.ex

```
plug :authenticate when action in [:index, :show]
```

Then, change the index action back to its previous state, like this:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change3.ex

```
def index(conn, _params) do
  users = Accounts.list_users()
  render(conn, "index.html", users: users)
end
```

Visit http://localhost:4000/users to see our plug in action. We redirect, exactly as we should.

Let's take a minute to appreciate the code we've written so far. A small change to our authentication lets us plug it before every action. We can also share it with any other controllers or even move it to a router pipeline, restricting whole sections of our application with minor changes. None of these features relies on magical inheritance mechanisms, only our explicit lists of functions in our plug pipelines.

At this point, you may also be wondering what happened with halt. When we changed the index action, we had to explicitly check if the connection halted or not, before acting on it. Plug pipelines explicitly check for halted: true between every plug invocation, so the halting concern is neatly solved by Plug.

In fact, you're seeing Elixir macro expansion in action. Let's take an arbitrary example. Suppose you write code like this:

```
plug :one
plug Two
plug :three, some: :option
```

It would compile to:

```
case one(conn, []) do
  %Plug.Conn{halted: true} = conn -> conn
  conn ->
    case Two.call(conn, Two.init([])) do
      %Plug.Conn{halted: true} = conn -> conn
      conn ->
        case three(conn, some: :option) do
          %Plug.Conn{halted: true} = conn -> conn
          conn -> conn
        end
    end
end
```

Elixir macros and macro expansion are beyond the scope of this book. What you need to know is that at some point in the compile process, Elixir would translate the first example to the second. Conceptually, not much is happening here, and that's exactly the beauty behind Plug. For each plug, we invoke it with the given options, check if the returned connection halted, and move forward if it didn't. It's a simple abstraction that allows us to express and compose both simple and complex functionality.

With all that said, we already have a mechanism for loading data from the session and using it to restrict user access. But we still don't have a mechanism to log the users in.

Logging In

Let's add a tiny function to RumblWeb.Auth that receives the connection and the user, and stores the user ID in the session:

authentication/listings/rumbl/lib/rumbl_web/controllers/auth.change1.ex

```
def login(conn, user) do
  conn
  |> assign(:current_user, user)
  |> put_session(:user_id, user.id)
  |> configure_session(renew: true)
end
```

As you recall, the Plug.Conn struct has a field called assigns. We call setting a value in that structure an *assign*. Our function stores the given user as the :current_user assign, puts the user ID in the session, and finally configures the session, setting the :renew option to true. The last step is extremely important and it protects us from session fixation attacks. It tells Plug to send the session cookie back to the client with a different identifier, in case an attacker knew, by any chance, the previous one.

Let's go back to the RumblWeb.UserController.create action and change it to call the login function after we insert the user in the database:

authentication/listings/rumbl/lib/rumbl_web/controllers/user_controller.change2.ex

```
def create(conn, %{"user" => user_params}) do
  case Accounts.register_user(user_params) do
    {:ok, user} ->
      conn
      |> RumblWeb.Auth.login(user)
      |> put_flash(:info, "#{user.name} created!")
      |> redirect(to: Routes.user_path(conn, :index))

    {:error, %Ecto.Changeset{} = changeset} ->
      render(conn, "new.html", changeset: changeset)
  end
end
```

Now visit http://localhost:4000/users/new, register a new user, and try to access the pages we restricted previously. As you can see, the user can finally access them.