

IF140303-Web Application Development

## **Session-12:**

# **Applying a Plug in Elixir and Phoenix**

PRU/SPMI/FR-BM-18/0222

**Alfa Yohannis**



# Introduction to Plugs in Elixir

- A plug is a specification for composing web applications.
- Plugs are the building blocks of Elixir web applications like Phoenix.
- There are two types of plugs:
  - **Function Plug:** A simple plug, defined as a function.
  - **Module Plug:** A more complex plug, defined as a module.

# Creating a Module Plug for Authentication

- We will create a module plug to check if a user is logged in.
- This plug checks if a user ID is assigned in the connection object.
- If the user ID exists, it fetches the user from the database and assigns it to the connection object.
- The plug consists of two main functions:
  - `init/1`: Used for setup, called once.
  - `call/2`: Called with the connection and returns a connection.
- The `assign/3` function is used to store a value in the connection's assigns.

# Creating the Plug Module

```
1  defmodule Discuss.Plugs.SetUser do
2    import Plug.Conn
3    import Phoenix.Controller
4
5    alias Discuss.Repo
6    alias Discuss.User
7    alias Discuss.Router.Helpers
8
9    def init(\_params) do
10     end
```

# Creating the Plug Module

```
1  def call(conn, \_params) do
2    user\_id = get\_session(conn, :user\_id)
3    cond do
4      user = user\_id \&\& Repo.get(User, user\_id) ->
5        assign(conn, :user, user)
6      true ->
7        assign(conn, :user, nil)
8    end
9  end
10 end
```

- init/1: Placeholder function for initialization.
- call/2: Checks session for user\_id, assigns user if found.
- assign/3 stores user data in conn.assigns for later use.

# Using the Plug in the Router

- The created plug should be added to the router's pipeline.
- Modify `router.ex` to include the plug in the browser pipeline.

```
1 pipeline :browser do
2   plug Discuss.Plugs.SetUser
3   end
```

# Adding a Login Button to the Header

```
1 <body>
2 <nav class="light-blue">
3 <div class="nav-wrapper container">
4 <a href="/" class="brand-logo">Logo</a>
5 <ul class="right">
6 <%= if @conn.assigns[:user] do %>
7 <li>
8 <%= link "Logout", to: session_path(@conn, :signout) %>
9 </li>
```

# Adding a Login Button to the Header

- We add a login button to the application header.
- The button checks if a user is logged in and displays the appropriate option.
- If the user is logged in, a logout button is shown; otherwise, a login button appears.



# Adding a Login Button to the Header

```
1 <\% else \%>
2 <li>
3 <\%= link "Sign in with Github", to: session_path(@conn, :
   request, "github") \%>
4 </li>
5 <\% end \%>
6 </ul>
7 </div>
8 </nav>
```

- `@conn.assigns[:user]` is checked to see if a user is logged in.
- The `link/2` function creates the login/logout link.

# Updating the Router for Signout

- We add a signout route to the router.
- This allows users to log out of the application.

```
1   scope "/auth", Discuss do
2     pipe\_through :browser
3     get "/signout", AuthController, :signout
4     get "[:provider]", AuthController, :request
5     get "[:provider/callback]", AuthController, :request
6   end
```

# Updating the AuthController for Signout

- We add a signout function to the AuthController.
- This function clears the session and redirects the user to the home page.

```
1  ...  
2  def signout(conn, \_params) do  
3  conn  
4  |> configure\_session(drop: true)  
5  |> redirect(to: topic\_path(conn, :index))  
6  end  
7  ...
```

# Adding a Logout Button to the Header

- The logout button is dynamically displayed in the header.
- When clicked, it triggers the signout route and logs the user out.

```
1 <body>
2 <nav class="light-blue">
3 <div class="nav-wrapper container">
4 <a href="/" class="brand-logo">Logo</a>
5 <ul class="right">
6 <%= if @conn.assigns[:user] do %>
7 <li>
8 <%= link "Logout", to: session_path(@conn, :signout) %>
9 </li>
```

# Adding a Logout Button to the Header

```
1 <\% else \%>
2 <li>
3 <\%= link "Sign in with Github", to: session_path(@conn, :
   request, "github") \%>
4 </li>
5 <\% end \%>
6 </ul>
7 </div>
8 </nav>
```

- `link/2` creates a hyperlink to the logout route.
- The button is displayed based on the user's login status.

# Adding Authorization: Restricting Actions Based on User Authentication

- We need to ensure only signed-in users can post, edit, or delete topics.
- To enforce this, we'll create a new plug in 'Web > controllers > plug > require\_auth.ex'.

# Creating the 'RequireAuth' Plug

```
1  defmodule Discuss.Plugs.RequireAuth do
2    use Plug.Conn
3    use Phoenix.Controller
4
5    alias Discuss.Router.Helpers
6
7    def init(_params) do
8      end
9
10   def call(conn, _params) do
11     if conn.assigns[:user] do
12       conn
```

# Creating the 'RequireAuth' Plug

```
1  else
2  conn
3    |> put_flash(:error, "You must be logged in.")
4    |> redirect(to: Helpers.topic_path(conn, :index))
5    |> halt()
6  end
7  end
8  end
```

- `halt/0` stops the connection processing immediately.
- This ensures that unauthorized users are redirected before further actions.



# Updating 'TopicController' with 'RequireAuth' Plug

- We will update the 'TopicController' to restrict specific actions to signed-in users.
- The plug is applied conditionally using 'when' with the 'action' keyword.

# Updated 'TopicController' with 'RequireAuth' Plug

```
1  defmodule Discuss.TopicController do
2    use Discuss.Web, :controller
3
4    alias Discuss.Topic
5
6    plug Discuss.Plugs.RequireAuth when action in [:new, :create
7      , :edit, :update, :delete]
8
9    ...
```

- Only the specified actions (new, create, edit, update, delete) will trigger the 'RequireAuth' plug.
- This ensures non-logged-in users cannot access these actions.

# Associating Users with Topics

- We'll associate users with topics using a foreign key relationship.
- This involves adding a 'user\_id' column to the 'topics' table.
- Migration will alter the existing table, leaving the 'user\_id' empty for previously created topics.

# Migration: Adding 'user\_id' to Topics

```
1  defmodule Discuss.Repo.Migrations.AddUserIdToTopics do
2    use Ecto.Migration
3
4    def change do
5      alter table(:topics) do
6        add :user_id, references(:users)
7      end
8    end
9  end
```

- `references(:users)` establishes the foreign key relationship.
- Migrate the database with `'mix ecto.migrate'`.

# Updating Models for Association

- Now, we need to associate the 'User' model with the 'Topic' model.
- This is done by adding the appropriate associations in both models.

# User Model: Adding 'has\_many' Association

```
1  defmodule Discuss.User do
2    use Discuss.Web, :model
3
4    schema "users" do
5      field :email, :string
6      field :provider, :string
7      field :token, :string
8      has_many :topics, Discuss.Topic
9      timestamps()
10   end
```

# User Model: Adding 'has\_many' Association

```
1  def changeset(struct, params \\ %{}) do
2    struct
3    |> cast(params, [:email, :provider, :token])
4    |> validate_required([:email, :provider, :token])
5  end
6  end
```

- `has_many :topics` sets up the one-to-many relationship.
- A user can have multiple topics.

# Topic Model: Adding 'belongs\_to' Association

```
1  defmodule Discuss.Topic do
2    use Discuss.Web, :model
3
4    schema "topics" do
5      field :title, :string
6      belongs_to :user, Discuss.User
7    end
```



# Topic Model: Adding 'belongs\_to' Association

```
1  def changeset(struct, params \\ %{}) do
2    struct
3    |> cast(params, [:title])
4    |> validate_required([:title])
5  end
6  end
```

- belongs\_to :user establishes the reverse relationship.
- Each topic is associated with a specific user.

# Updating 'TopicController' to Associate Topics with Users

- We need to update the 'create' function to associate a new topic with the current user.
- This involves using 'build\_assoc/2'.

# Updating the 'create' Function

```
1  def create(conn, %{"topic" => topic}) do
2    changeset = conn.assigns.user
3    |> build_assoc(:topics)
4    |> Topic.changeset(topic)
5
6    case Repo.insert(changeset) do
7      {:ok, _topic} ->
8        conn
9        |> put_flash(:info, "Topic Created")
10       |> redirect(to: topic_path(conn, :index))

```

# Updating the 'create' Function

```
1      {:error, changeset} ->  
2      render conn, "new.html", changeset: changeset  
3      end  
4      end
```

- build\_assoc/2 automatically associates the new topic with the current user.
- This ensures that the 'user\_id' field in the 'topics' table is populated.

# Restricting Edit/Delete Buttons Based on Own

- We will update the 'index.html.eex' template to ensure only the topic owner can see edit/delete buttons.
- This is done using an 'if' statement.

# Updating 'index.html.eex' Template

```
1 <ul class = "collection">
2   <%= for topic <- @topics do %>
3     <li class="collection-item">
4       <%= topic.title %>
5
6       <%= if @conn.assigns.user.id == topic.user_id do %>
7         <div class="right">
8           <%= link "Edit", to: topic_path(@conn, :edit, topic) %>
9           <%= link "Delete", to: topic_path(@conn, :delete , topic),
              method: :delete %>
10        </div>
11      <% end %>
```

# Updating 'index.html.eex' Template

```
1     </li>  
2     <%end%>  
3 </ul>
```

- The 'if' statement ensures that only the topic owner sees the edit/delete options.
- This provides a basic level of authorization on the front-end.

# Adding a Plug to Refuse Unauthorized Actions

- Even though unauthorized users cannot see the buttons, they might still access actions via URL.
- We'll create a plug to refuse unauthorized actions.



# Adding a Plug to Refuse Unauthorized Actions

```
1  def check_topic_owner(conn, _params) do
2    %{params: %{"id" => topic_id}} = conn
3    if Repo.get(Topic, topic_id).user_id == conn.assigns.user.id
4      do
5        conn
6      else
```

# Adding a Plug to Refuse Unauthorized Actions

```
1      conn
2      |> put_flash(:error, "You cannot edit that")
3      |> redirect(to: topic_path(conn, :index))
4      |> halt()
5      end
6      end
```

- We compare the 'user\_id' of the topic with the 'id' of the logged-in user.
- If they don't match, we refuse the action.

# Conclusion

- We implemented authorization to restrict actions based on user authentication.
- Users are now associated with their topics.
- These steps ensure better security and user management within our Discuss application.