Taking Off with Phoenix

@scrogson / https://github.com/scrogson

Phoenix

https://github.com/phoenixframework

Phoenix is a framework for building modern web apps, API backends, and distributed systems. Written in Elixir, you get beautiful syntax, productive tooling, and a fast and efficient runtime.

- MVC framework.
- Familiar to those with experience in other web frameworks similar to Ruby on Rails or Python's Django.
- Channels for implementing realtime features.
- Easy to test.
- No compromise - high productivity and high performance.

I hear Phoenix is pretty fast.

How fast?

Framework	Throughput (req/s)	Latency (ms)	Consistency (σ ms)
Plug	198328.21	0.63ms	2.22ms
Phoenix 0.13.1	179685.94	0.61ms	1.04ms
Gin	176156.41	0.65ms	0.57ms
Play	171236.03	1.89ms	14.17ms
Phoenix 0.9.0-dev	169030.24	0.59ms	0.30ms
Express Cluster	92064.94	1.24ms	1.07ms
Martini	32077.24	3.35ms	2.52ms
Sinatra	30561.95	3.50ms	2.53ms
Rails	11903.48	8.50ms	4.07ms

https://gist.github.com/omnibs/e5e72b31e6bd25caf39a

You down with OTP?

Yeah you know me!

Phoenix is an **OTP** application that provides functionality to your **OTP** application.

Building Blocks

Phoenix is the top layer of a multi-layer system designed to be modular and flexible.

- **Plug** - A simple abstraction for dealing with different web servers.
- **Cowboy** - Small, fast, modular HTTP server written in Erlang.
- **Ecto** - a DSL for writing queries and interacting with databases.

Getting Started

Installing

```
λ mix archive.install https://github.com/phoenixframework/phoenix/releases/
 download/v1.0.3/phoenix_new-1.0.3.ez
```

Installing

```
λ mix archive.install https://github.com/phoenixframework/phoenix/releases/
 download/v1.0.3/phoenix_new-1.0.3.ez
Are you sure you want to install archive "https://github.com/phoenixframework/
phoenix/releases/download/v1.0.3/phoenix_new-1.0.3.ez"? [Yn]
  *  creating .mix/archives/phoenix_new-1.0.3.ez

λ
```

Generating Your App

```
λ mix phoenix.new myapp
```

Generating Your App

```
λ mix phoenix.new myapp
* creating myapp/config/config.exs
* creating myapp/config/dev.exs
* creating myapp/config/prod.exs
* creating myapp/config/prod.secret.exs
* creating myapp/config/test.exs
* creating myapp/lib/myapp.ex
* creating myapp/lib/myapp/endpoint.ex
* creating myapp/test/views/error_view_test.exs
* creating myapp/test/support/conn_case.ex
* creating myapp/test/support/channel_case.ex
* creating myapp/test/test_helper.exs
* creating myapp/web/channels/user_socket.ex
* creating myapp/web/router.ex
* creating myapp/web/views/error_view.ex
* creating myapp/web/web.ex
* creating myapp/mix.exs
* creating myapp/README.md
* creating myapp/lib/myapp/repo.ex
* creating myapp/test/support/model_case.ex
* creating myapp/priv/repo/seeds.exs
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get

We are all set! Run your Phoenix application:

    $ cd myapp
    $ mix ecto.create
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get

We are all set! Run your Phoenix application:

    $ cd myapp
    $ mix ecto.create
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get

We are all set! Run your Phoenix application:

    $ cd myapp
    $ mix ecto.create
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Create Your Database

```
λ mix ecto.create
The database for Myapp.Repo has been created.
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get

We are all set! Run your Phoenix application:

    $ cd myapp
    $ mix ecto.create
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Generating Your App

```
* creating myapp/test/views/page_view_test.exs
* creating myapp/web/controllers/page_controller.ex
* creating myapp/web/templates/layout/app.html.eex
* creating myapp/web/templates/page/index.html.eex
* creating myapp/web/views/layout_view.ex
* creating myapp/web/views/page_view.ex

Fetch and install dependencies? [Yn]
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get

We are all set! Run your Phoenix application:

    $ cd myapp
    $ mix ecto.create
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Starting Your App

```
λ iex -S mix phoenix.server
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false] [dtrace]

[info] Running Myapp.Endpoint with Cowboy on http://localhost:4000
Interactive Elixir (1.1.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 25 Nov 23:57:09 - info: compiled 5 files into 2 files, copied 3 in
2284ms
```

localhost:4000

Phoenix Framework

Get Started

Welcome to Phoenix!

Phoenix is an Elixir Web Framework targeting full-featured, fault tolerant applications with realtime functionality.

Resources

- Docs
- Source

Help

- Issues
- #elixir-lang on freenode IRC
- @chris_mccord

phoenixframework.org

Go to “http://www.phoenixframework.org/docs”

Server Logs

```
λ iex -S mix phoenix.server
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false] [dtrace]

[info] Running Myapp.Endpoint with Cowboy on http://localhost:4000
Interactive Elixir (1.1.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 25 Nov 23:57:09 - info: compiled 5 files into 2 files, copied 3 in
2284ms

[info] GET /
[debug] Processing by Myapp.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 889µs
```

File Structure

```
λ tree -L 1
.
├── README.md
├── _build
├── brunch-config.js
├── config
├── deps
├── lib
├── mix.exs
├── mix.lock
├── node_modules
├── package.json
├── priv
├── test
└── web

8 directories, 5 files
```

File Structure

```
λ tree -L 1
.
├── README.md
├── _build
├── brunch-config.js
├── config
├── deps
├── lib
├── mix.exs
├── mix.lock
├── node_modules
├── package.json
├── priv
├── test
└── web

8 directories, 5 files
```

File Structure

```
λ tree -L 1
.
├── README.md
├── _build
├── brunch-config.js
├── config
├── deps
├── lib
├── mix.exs
├── mix.lock
├── node_modules
├── package.json
├── priv
├── test
└── web

8 directories, 5 files
```

File Structure

```
λ tree -L 1
.
├── README.md
├── _build
├── brunch-config.js
├── config
├── deps
├── lib
├── mix.exs
├── mix.lock
├── node_modules
├── package.json
├── priv
├── test
└── web

8 directories, 5 files
```

File Structure

```
λ tree -L 1
.
├── README.md
├── _build
├── brunch-config.js
├── config
├── deps
├── lib
├── mix.exs
├── mix.lock
├── node_modules
├── package.json
├── priv
├── test
└── web

8 directories, 5 files
```

Web

```
λ tree -L 1 web
web
├── channels
├── controllers
├── models
├── router.ex
├── static
├── templates
├── views
└── web.ex

6 directories, 2 files
```

Lib

```
λ tree -L 2 lib
lib
├── myapp
|   ├── endpoint.ex
|   └── repo.ex
└── myapp.ex

1 directory, 3 files
```

Test

```
λ tree -L 2 test
test
├── channels
├── controllers
|   └── page_controller_test.exs
├── models
├── support
|   ├── channel_case.ex
|   ├── conn_case.ex
|   └── model_case.ex
├── test_helper.exs
└── views
    ├── error_view_test.exs
    ├── layout_view_test.exs
    └── page_view_test.exs

5 directories, 8 files
```

Running the Tests

```
λ mix test
....

Finished in 0.5 seconds (0.4s on load, 0.05s on tests)
4 tests, 0 failures

Randomized with seed 413182
```

Mix Tasks

```
λ mix help | grep phoenix
mix phoenix.digest       # Digests and compress static files
mix phoenix.gen.channel # Generates a Phoenix channel
mix phoenix.gen.html     # Generates files for an HTML based resource
mix phoenix.gen.json     # Generates files for a JSON based resource
mix phoenix.gen.model    # Generates an Ecto model
mix phoenix.gen.secret   # Generates a secret
mix phoenix.new          # Create a new Phoenix v1.0.3 application
mix phoenix.routes       # Prints all routes
mix phoenix.server       # Starts applications and their servers
```

Plug

https://github.com/elixir-lang/plug

A specification for constructing **composable** modules to build web applications. Plugs are reusable modules or functions built to that specification.

Plugs can be written to handle almost anything, from authentication to parameter pre-processing, and rendering.

Provides adapters to HTTP servers which will ultimately deliver application content to our users.

The Plug Specification

There are two kind of plugs:

- function plugs
- module plugs

Function Plugs

A function plug is any function that receives a connection and a set of options and returns a connection. Its type signature must be:

```
(Plug.Conn.t, Plug.opts) :: Plug.Conn.t
```

Function Plug Example

```elixir
def json_header_plug(conn, _opts) do
  conn
  |> put_resp_content_type("application/json")
end
```

Module Plugs

A module plug is an extension of the function plug.
It is a module that must export:

- `init/1` - takes a set of options and initializes it.
- `call/2` - takes the connection and options, returns the connection

The result returned by `init/1` is passed as second argument to `call/2`.

Module Plug Example

```elixir
defmodule JSONHeaderPlug do
  import Plug.Conn

  def init(opts), do: opts

  def call(conn, _opts) do
    conn
    |> put_resp_content_type("application/json")
  end
end
```

```
defstruct adapter:          {Plug.Conn, nil},
          assigns:          %{},
          before_send:      [],
          body_params:      %Unfetched{aspect: :body_params},
          cookies:          %Unfetched{aspect: :cookies},
          halted:           false,
          host:             "www.example.com",
          method:           "GET",
          owner:            nil,
          params:           %Unfetched{aspect: :params},
          path_info:        [],
          port:             0,
          private:          %{},
          query_params:     %Unfetched{aspect: :query_params},
          query_string:     "",
          peer:             nil,
          remote_ip:        nil,
          req_cookies:      %Unfetched{aspect: :cookies},
          req_headers:      [],
          request_path:     "",
          resp_body:        nil,
          resp_cookies:     %{},
          resp_headers:     [{"cache-control", "max-age=0, private, must-revalidate"}],
          scheme:           :http,
          script_name:      [],
          secret_key_base:  nil,
          state:            :unset,
          status:           nil
```

Request Fields

These fields contain request information:

* `host` - the requested host as a binary, example: `"www.example.com"`
* `method` - the request method as a binary, example: `"GET"`
* `path_info` - the path split into segments, example: `["hello", "world"]`
* `script_name` - the initial portion of the URL's path that corresponds to the application routing, as segments, example: ["sub","app"].
* `request_path` - the requested path, example: `/trailing/and//double//slashes/`
* `port` - the requested port as an integer, example: `80`
* `peer` - the actual TCP peer that connected, example: `{{127, 0, 0, 1}, 12345}`. Often this is not the actual IP and port of the client, but rather of a load-balancer or request-router.
* `remote_ip` - the IP of the client, example: `{151, 236, 219, 228}`. This field is meant to be overwritten by plugs that understand e.g. the `X-Forwarded-For` header or HAProxy's PROXY protocol. It defaults to peer's IP.
* `req_headers` - the request headers as a list, example: `[{"content-type", "text/plain"}]`
* `scheme` - the request scheme as an atom, example: `:http`
* `query_string` - the request query string as a binary, example: `"foo=bar"`

Fetchable Fields

The request information in these fields is not populated until it is fetched using the associated `fetch_` function. For example, the `cookies` field uses `fetch_cookies/2`.

If you access these fields before fetching them, they will be returned as `Plug.Conn.Unfetched` structs.

* `cookies`- the request cookies with the response cookies
* `query_params` - the request query params
* `params` - the request params. Usually populated by a plug, like `Plug.Parsers`
* `req_cookies` - the request cookies (without the response ones)

Response Fields

These fields contain response information:

* `resp_body` - the response body, by default is an empty string. It is set
  to nil after the response is set, except for test connections.
* `resp_charset` - the response charset, defaults to "utf-8"
* `resp_cookies` - the response cookies with their name and options
* `resp_headers` - the response headers as a dict, by default `cache-control`
  is set to `"max-age=0, private, must-revalidate"`
* `status` - the response status

Furthermore, the `before_send` field stores callbacks that are invoked
before the connection is sent. Callbacks are invoked in the reverse order
they are registered (callbacks registered first are invoked last) in order
to reproduce a pipeline ordering.

Connection Fields

```
* `assigns` - shared user data as a dict
* `owner` - the Elixir process that owns the connection
* `halted` - the boolean status on whether the pipeline was halted
* `secret_key_base` - a secret key used to verify and encrypt cookies.
  the field must be set manually whenever one of those features are used.
  This data must be kept in the connection and never used directly, always
  use `Plug.Crypto.KeyGenerator.generate/3` to derive keys from it
* `state` - the connection state

The connection state is used to track the connection lifecycle. It starts
as `:unset` but is changed to `:set` (via `Plug.Conn.resp/3`) or `:file`
(when invoked via `Plug.Conn.send_file/3`). Its final result is
`:sent` or `:chunked` depending on the response model.
```

Private Fields

These fields are reserved for libraries/framework usage.

* `adapter` - holds the adapter information in a tuple
* `private` - shared library data as a dict

Plugs in Phoenix

- Endpoint
- Router
- Controllers

Endpoint

The endpoint is the boundary where all requests to your web application start. It is also the interface your application provides to the underlying web servers.

Overall, an endpoint has three responsibilities:

- provides a wrapper for starting and stopping the endpoint as part of a supervision tree;
- defines an initial plug pipeline where requests are sent through;
- hosts web specific configuration for your application.

Endpoint

```
defmodule Myapp.Endpoint do
  use Phoenix.Endpoint, otp_app: :myapp

  socket "/socket", Myapp.UserSocket

  plug Plug.Static,
    at: "/", from: :myapp, gzip: false,
    only: ~w(css fonts images js favicon.ico robots.txt)

  if code_reloading? do
    socket "/phoenix/live_reload/socket", Phoenix.LiveReloader.Socket
    plug Phoenix.LiveReloader
    plug Phoenix.CodeReloader
  end

  plug Plug.RequestId
  plug Plug.Logger
```

Endpoint

```
  plug Plug.Parsers,
    parsers: [:urlencoded, :multipart, :json],
    pass: ["*/*"],
    json_decoder: Poison

  plug Plug.MethodOverride
  plug Plug.Head

  plug Plug.Session,
    store: :cookie,
    key: "_myapp_key",
    signing_salt: "bKBYQ8PF"

  plug Myapp.Router
end
```

Router

The router provides a set of macros for generating routes that dispatch to specific controllers and actions. Those macros are named after HTTP verbs.

Router

```
defmodule Myapp.Router do
  use Myapp.Web, :router

  scope "/", Myapp do

    get "/", PageController, :index

    resources "/rooms", RoomController do
      resources "/users", UserController
    end
  end
end
```

Router

Leverages the BEAM's pattern-matching at run-time

```
defp match(conn, "GET", ["rooms"], _)
defp match(conn, "GET", ["rooms", id, "edit"], _)
defp match(conn, "GET", ["rooms", "new"], _)
defp match(conn, "GET", ["rooms", id], _)
defp match(conn, "POST", ["rooms"], _)
defp match(conn, "PATCH", ["rooms", id], _)
defp match(conn, "PUT", ["rooms", id], _)
defp match(conn, "DELETE", ["rooms", id], _)
defp match(conn, "GET", ["rooms", post_id, "users"], _)
defp match(conn, "GET", ["rooms", post_id, "users", id, "edit"], _)
defp match(conn, "GET", ["rooms", post_id, "users", "new"], _)
defp match(conn, "GET", ["rooms", post_id, "users", id], _)
defp match(conn, "POST", ["rooms", post_id, "users"], _)
defp match(conn, "PATCH", ["rooms", post_id, "users", id], _)
defp match(conn, "PUT", ["rooms", post_id, "users", id], _)
defp match(conn, "DELETE", ["rooms", post_id, "users", id], _)
```

Router

This

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

Compiles to this (showing full function definition)

```
defp match(conn, "GET", [], _) do
  conn
  |> Plug.Conn.put_private(:phoenix_pipelines, [:browser])
  |> Plug.Conn.put_private(:phoenix_route, fn conn ->
    opts = Myapp.PageController.init(:index)
    Myapp.PageController.call(conn, opts)
  end)
  |> browser([])
end
```

Router

Mix Task

```
λ mix phoenix.routes
     page_path  GET     /                                Myapp.PageController :index
     room_path  GET     /rooms                           Myapp.RoomController :index
     room_path  GET     /rooms/:id/edit                  Myapp.RoomController :edit
     room_path  GET     /rooms/new                       Myapp.RoomController :new
     room_path  GET     /rooms/:id                       Myapp.RoomController :show
     room_path  POST    /rooms                           Myapp.RoomController :create
     room_path  PATCH   /rooms/:id                       Myapp.RoomController :update
                PUT     /rooms/:id                       Myapp.RoomController :update
     room_path  DELETE  /rooms/:id                       Myapp.RoomController :delete
room_user_path  GET     /rooms/:room_id/users            Myapp.UserController :index
room_user_path  GET     /rooms/:room_id/users/:id/edit   Myapp.UserController :edit
room_user_path  GET     /rooms/:room_id/users/new        Myapp.UserController :new
room_user_path  GET     /rooms/:room_id/users/:id        Myapp.UserController :show
room_user_path  POST    /rooms/:room_id/users            Myapp.UserController :create
room_user_path  PATCH   /rooms/:room_id/users/:id        Myapp.UserController :update
                PUT     /rooms/:room_id/users/:id        Myapp.UserController :update
room_user_path  DELETE  /rooms/:room_id/users/:id        Myapp.UserController :delete
```

Pipelines

Pipelines group functions together
to handle common tasks.

```
send_resp(controller(router(
endpoint(connection))))
```

```
handle_request(Conn) ->
  Conn1 = endpoint(Conn),
  Conn2 = router(Conn1),
  Conn3 = controller(Conn2),
  send_resp(Conn3).
```

```
cat ~/.ssh/id_rsa.pub | pbcopy
```

```
connection
|> endpoint
|> router
|> controller
|> send_resp
```

Router Pipelines

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

Router Pipelines

```
scope "/", Myapp do
  pipe_through :browser

  get "/", PageController, :index

  resources "/rooms", RoomController do
    resources "/users", UserController
  end
end
```

Generators

```
λ mix help | grep phoenix.gen
mix phoenix.gen.channel # Generates a Phoenix channel
mix phoenix.gen.html    # Generates files for an HTML based resource
mix phoenix.gen.json    # Generates files for a JSON based resource
mix phoenix.gen.model   # Generates an Ecto model
mix phoenix.gen.secret  # Generates a secret
```

Generate a Post resource

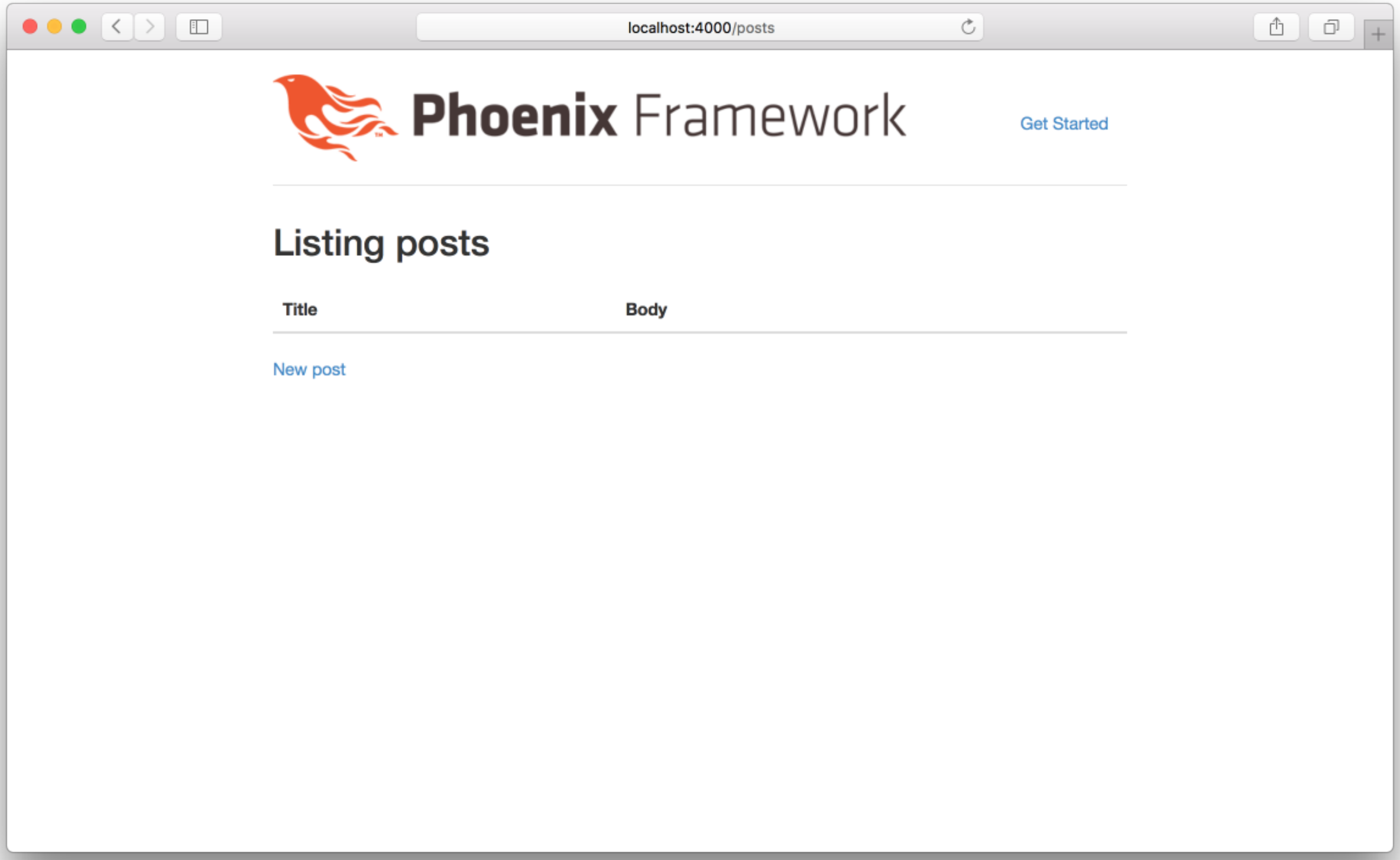
```
λ mix phoenix.gen.html Post posts title body:text
* creating web/controllers/post_controller.ex
* creating web/templates/post/edit.html.eex
* creating web/templates/post/form.html.eex
* creating web/templates/post/index.html.eex
* creating web/templates/post/new.html.eex
* creating web/templates/post/show.html.eex
* creating web/views/post_view.ex
* creating test/controllers/post_controller_test.exs
* creating priv/repo/migrations/20151130222956_create_post.exs
* creating web/models/post.ex
* creating test/models/post_test.exs

Add the resource to your browser scope in web/router.ex:

    resources "/posts", PostController

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

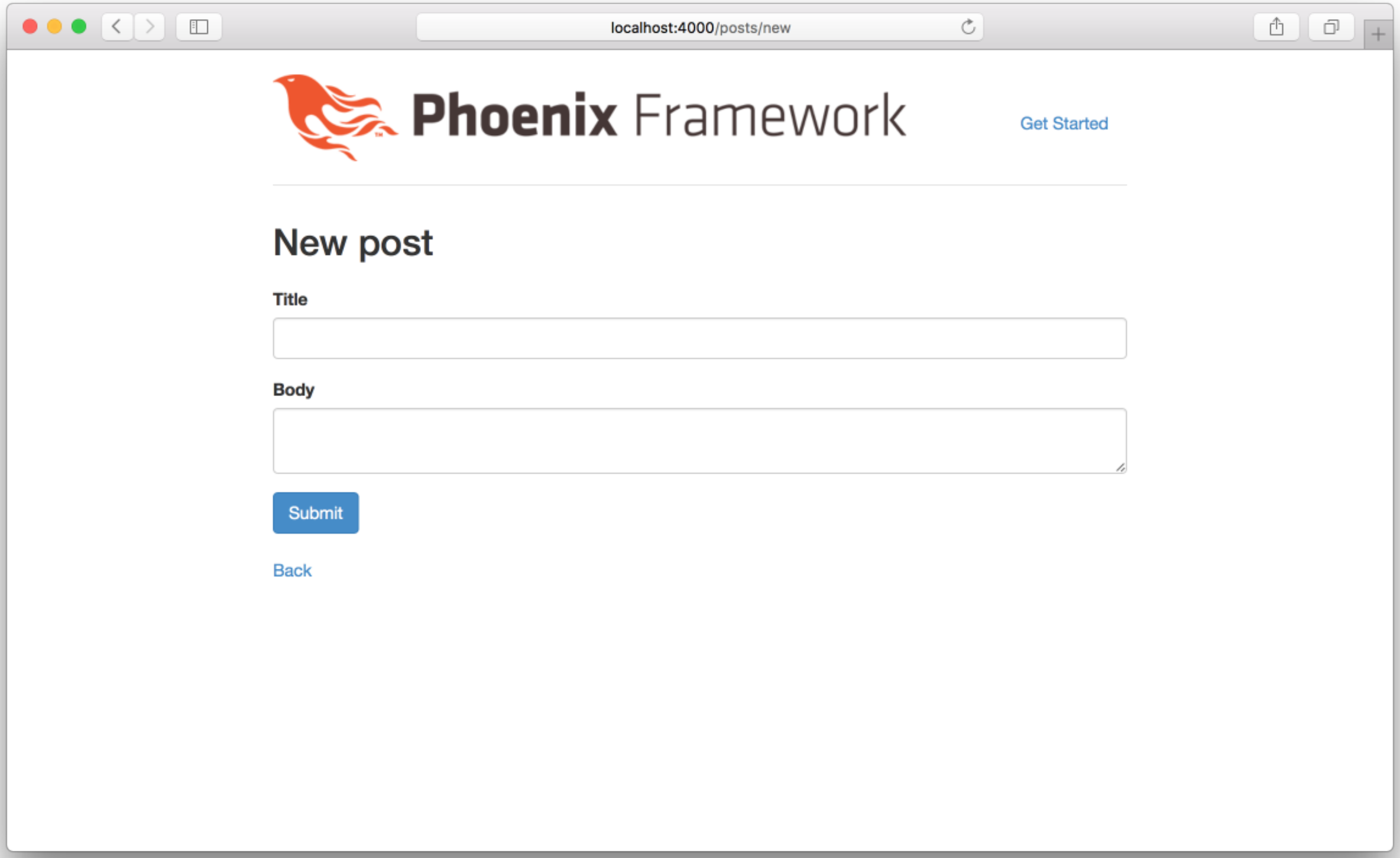
Add your routes

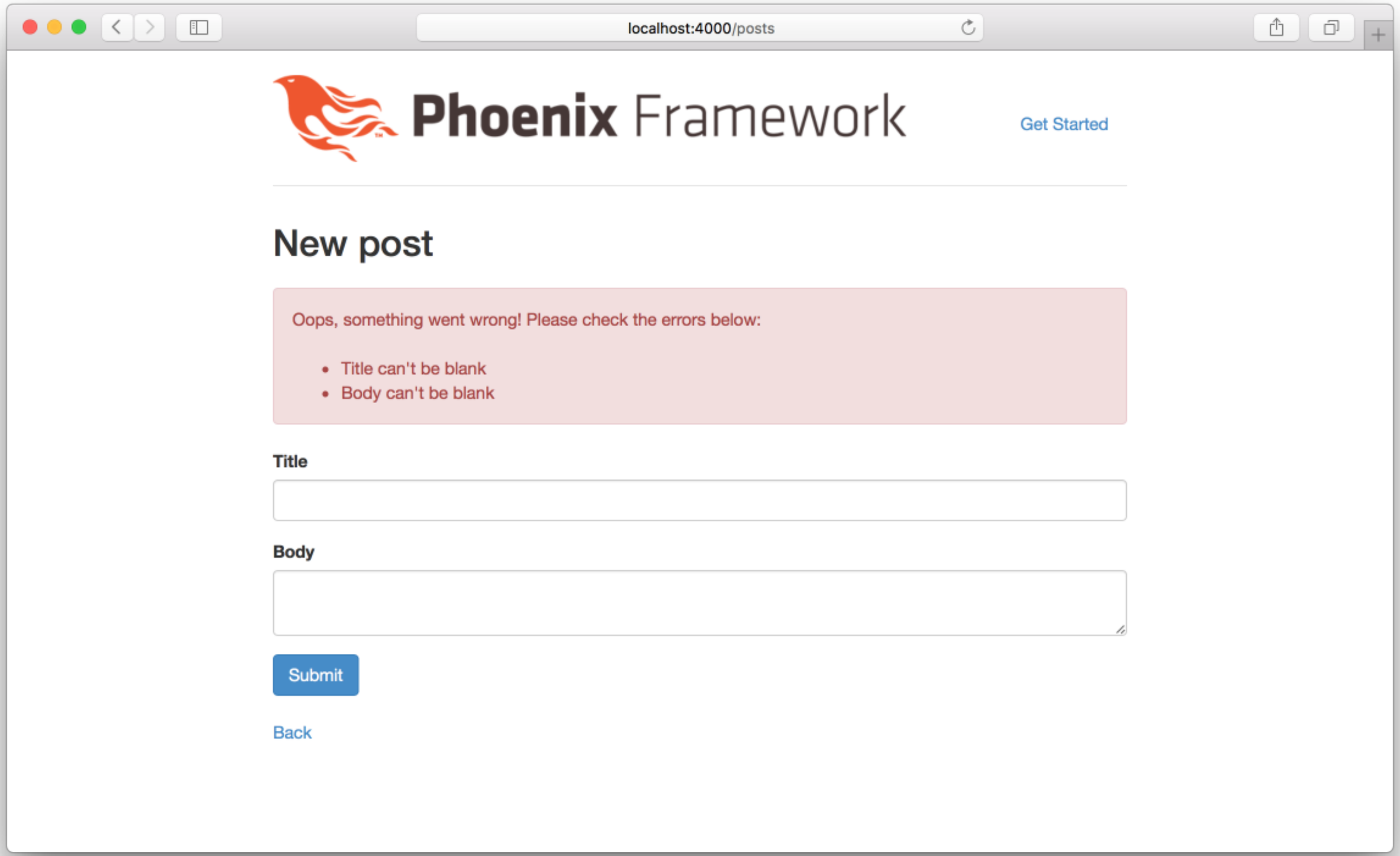
```
scope "/", Myapp do
  pipe_through :browser

  get "/", PageController, :index

  resources "/posts", PostController

  resources "/rooms", RoomController do
    resources "/users", UserController
  end
end
```

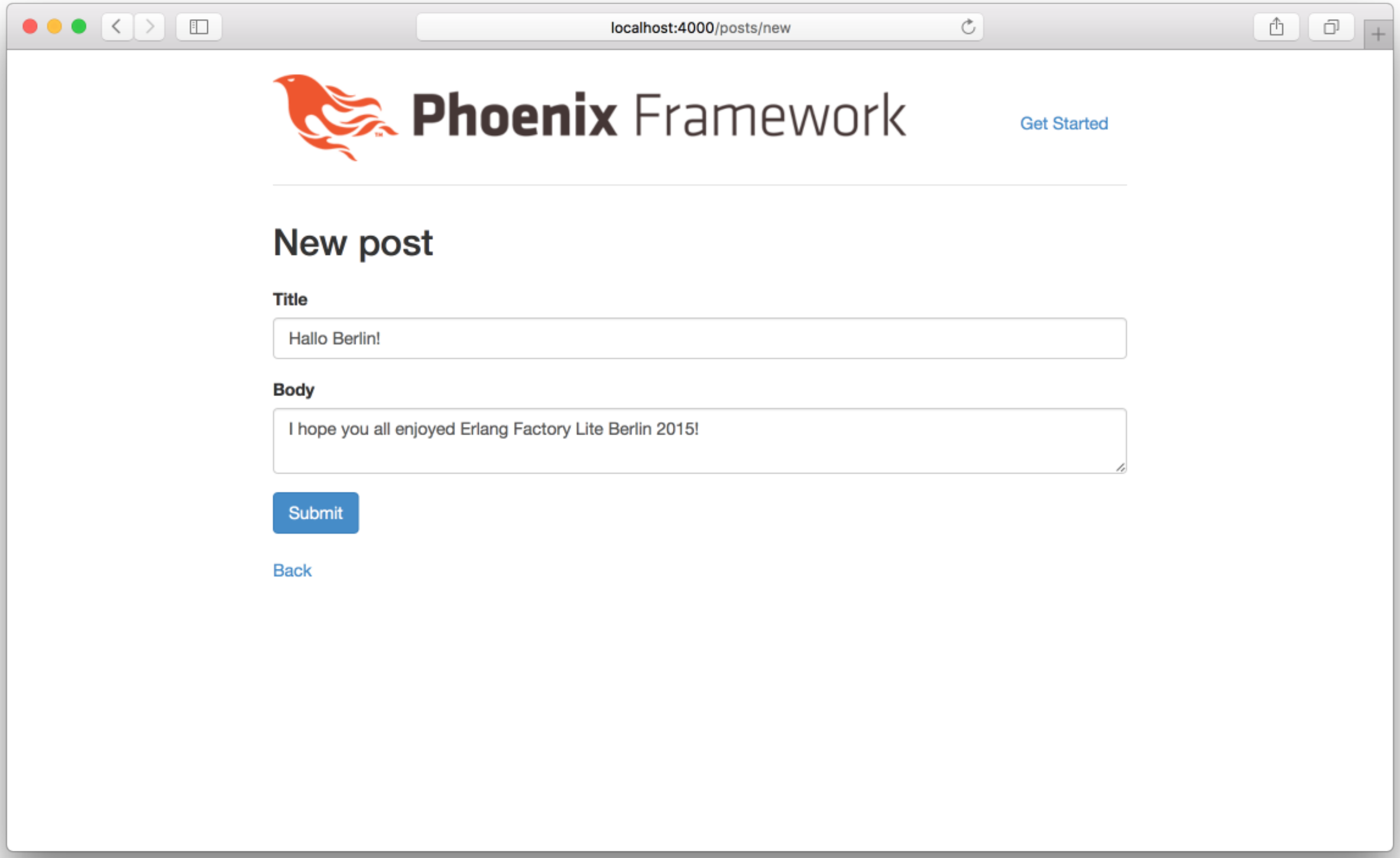
Migrate the Database

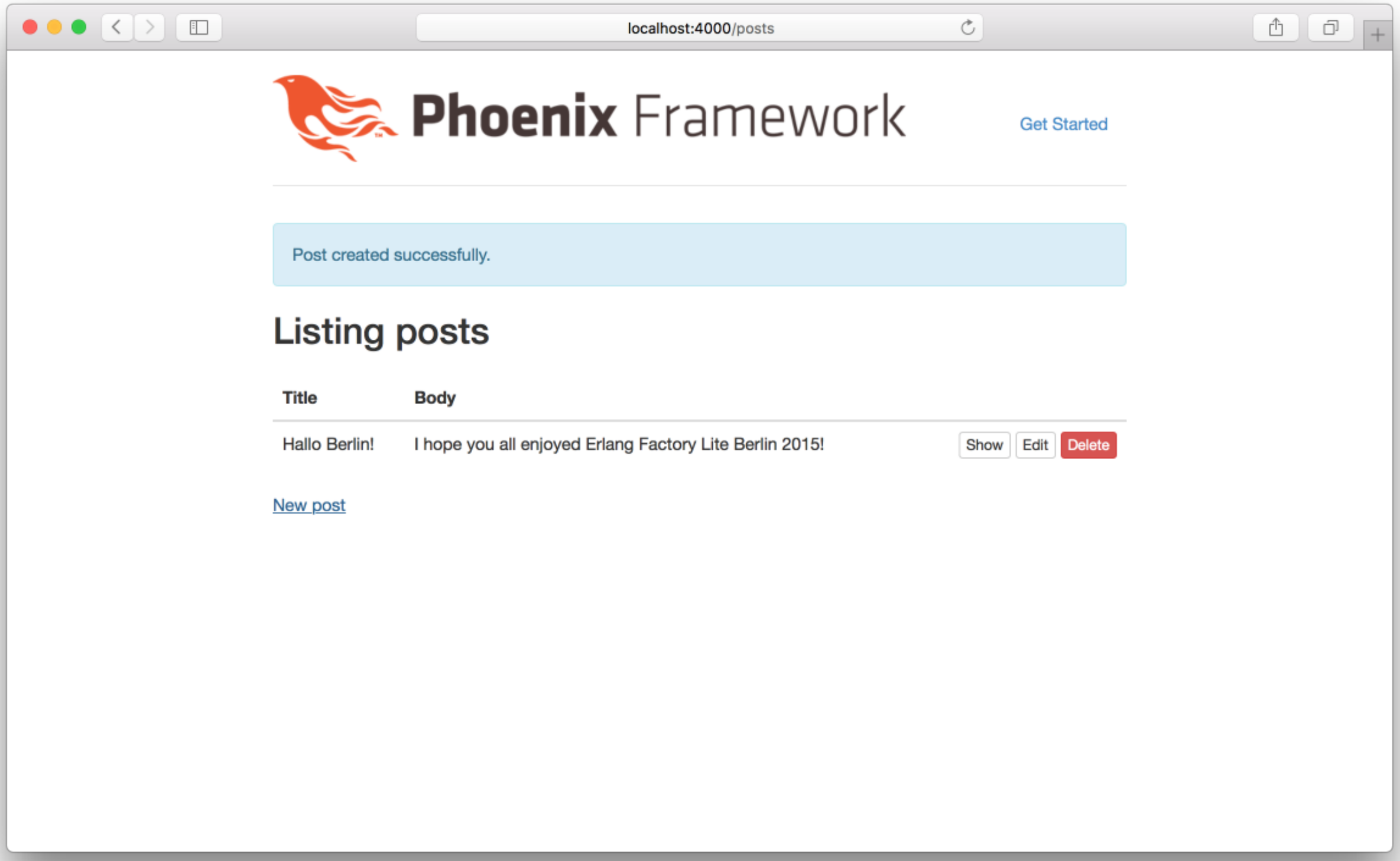
```
λ mix ecto.migrate
Compiled web/models/post.ex
Compiled web/views/page_view.ex
Compiled web/views/layout_view.ex
Compiled web/views/error_view.ex
Compiled web/controllers/page_controller.ex
Compiled web/controllers/post_controller.ex
Compiled web/views/post_view.ex
Compiled web/router.ex
Compiled lib/myapp/endpoint.ex
Generated myapp app

23:37:07.095 [info]  == Running Myapp.Repo.Migrations.CreatePost.change/0 forward

23:37:07.096 [info]  create table posts

23:37:07.113 [info]  == Migrated in 0.1s
```


localhost:4000/posts
Phoenix Framework
Get Started
Listing posts
Title
Body
New post


localhost:4000/posts/new
Phoenix Framework
Get Started
New post
Title
Body
Submit
Back


localhost:4000/posts
Phoenix Framework
Get Started
New post
Oops, something went wrong! Please check the errors below:
Title can't be blank
Body can't be blank
Title
Body
Submit
Back


localhost:4000/posts/new
Phoenix Framework
Get Started
New post
Title
Hallo Berlin!
Body
I hope you all enjoyed Erlang Factory Lite Berlin 2015!
Submit
Back


localhost:4000/posts
Phoenix Framework
Get Started
Post created successfully.
Listing posts
Title
Body
Hallo Berlin!
I hope you all enjoyed Erlang Factory Lite Berlin 2015!
Show
Edit
Delete
New post

Controllers

```
defmodule Myapp.PostController do
  use Myapp.Web, :controller

  alias Myapp.Post

  plug :scrub_params, "post" when action in [:create, :update]

  def index(conn, _params) do
    posts = Repo.all(Post)
    render(conn, "index.html", posts: posts)
  end

  def new(conn, _params) do
    changeset = Post.changeset(%Post{})
    render(conn, "new.html", changeset: changeset)
  end
  …
end
```

Controller Pipelines

```
defmodule Myapp.PostController do
  use Myapp.Web, :controller

  alias Myapp.Post

  plug :scrub_params, "post" when action in [:create, :update]

  def index(conn, _params) do
    posts = Repo.all(Post)
    render(conn, "index.html", posts: posts)
  end

  def new(conn, _params) do
    changeset = Post.changeset(%Post{})
    render(conn, "new.html", changeset: changeset)
  end
  …
end
```

GET /posts

List all posts

```
def index(conn, _params) do
  posts = Repo.all(Post)
  render(conn, "index.html", posts: posts)
end
```

GET /posts/:id

Show a single post

```
def show(conn, %{"id" => id}) do
  post = Repo.get!(Post, id)
  render(conn, "show.html", post: post)
end
```

GET /posts/new

Show a form to create a new post

```elixir
def new(conn, _params) do
  changeset = Post.changeset(%Post{})
  render(conn, "new.html", changeset: changeset)
end
```

POST /posts

Create a new post

```
def create(conn, %{"post" => post_params}) do
  changeset = Post.changeset(%Post{}, post_params)

  case Repo.insert(changeset) do
    {:ok, _post} ->
      conn
      |> put_flash(:info, "Post created successfully.")
      |> redirect(to: post_path(conn, :index))
    {:error, changeset} ->
      render(conn, "new.html", changeset: changeset)
  end
end
```

GET /posts/:id/edit

Find a post and render an edit form

```
def edit(conn, %{"id" => id}) do
  post = Repo.get!(Post, id)
  changeset = Post.changeset(post)
  render(conn, "edit.html", post: post, changeset: changeset)
end
```

PUT/PATCH /posts/:id

Update a post

```
def update(conn, %{"id" => id, "post" => post_params}) do
  post = Repo.get!(Post, id)
  changeset = Post.changeset(post, post_params)

  case Repo.update(changeset) do
    {:ok, post} ->
      conn
      |> put_flash(:info, "Post updated successfully.")
      |> redirect(to: post_path(conn, :show, post))
    {:error, changeset} ->
      render(conn, "edit.html", post: post, changeset: changeset)
  end
end
```

DELETE /posts/:id

Delete a post

```
def delete(conn, %{"id" => id}) do
  post = Repo.get!(Post, id)

  Repo.delete!(post)

  conn
  |> put_flash(:info, "Post deleted successfully.")
  |> redirect(to: post_path(conn, :index))
end
```

Views & Templates

- Views render templates
- Views serve as a presentation layer
- Module hierarchy for shared context
- Templates are precompiled into views
- EEx & Haml engine support

View

```
defmodule Myapp.PostView do
  use Myapp.Web, :view
  alias Myapp.Post

  def title(%Post{title: title}) do
    String.upcase(title)
  end

  def render("index.json", %{posts: posts}) do
    %{data: render_many(posts, Myapp.PostView, "show.json")}
  end

  def render("show.json", %{post: post}) do
    %{id: post.id,
      title: post.title,
      body: post.body,
      inserted_at: post.inserted_at}
  end
end
```

Template

```
<h2>Show post</h2>
<ul>
  <li>
    <strong>Title:</strong>
    <%= title(@post) %>
  </li>
  <li>
    <strong>Body:</strong>
    <%= @post.body %>
  </li>
</ul>
<%= link "Edit", to: post_path(@conn, :edit, @post) %>
<%= link "Back", to: post_path(@conn, :index) %>
```

Channels

Channels allow you to route pub/sub events to channel handlers in your application. By default, Phoenix supports both WebSocket and LongPoller transports.

- WebSocket / PubSub Abstraction
- Similar to Controllers, but bi-directional
- Handle socket events and broadcast
- phoenix.js - JavaScript client

UserSocket

```
defmodule Chat.Endpoint do
  use Phoenix.Endpoint, otp_app: :chat

  socket "/socket", Chat.UserSocket

  # Serve at "/" the given assets from "priv/static" directory
  plug Plug.Static,
    at: "/", from: :chat,
    only: ~w(css images js favicon.ico robots.txt)

  # Code reloading will only work if the :code_reloader key of
  # the :phoenix application is set to true in your config file.
  if code_reloading? do
    socket "/phoenix/live_reload/socket", Phoenix.LiveReloader.Socket
    plug Phoenix.CodeReloader
    plug Phoenix.LiveReloader
  end
```

UserSocket

```
defmodule Chat.UserSocket do
  use Phoenix.Socket

  channel "rooms:*", Chat.RoomChannel

  transport :websocket, Phoenix.Transports.WebSocket
  transport :longpoll, Phoenix.Transports.LongPoll

  def connect(_params, socket) do
    {:ok, socket}
  end

  def id(_socket), do: nil
end
```

RoomChannel

```elixir
defmodule Chat.RoomChannel do
  use Phoenix.Channel

  def join("rooms:lobby", message, socket) do
    Process.flag(:trap_exit, true)
    :timer.send_interval(5000, :ping)
    send(self, {:after_join, message})

    {:ok, socket}
  end

  def join("rooms:" <> _private_subtopic, _message, _socket) do
    {:error, %{reason: "unauthorized"}}
  end

  def handle_info({:after_join, msg}, socket) do
    broadcast! socket, "user:entered", %{user: msg["user"]}
    push socket, "join", %{status: "connected"}
    {:noreply, socket}
  end
  def handle_info(:ping, socket) do
```

RoomChannel

```
  def handle_info({:after_join, msg}, socket) do
    broadcast! socket, "user:entered", %{user: msg["user"]}
    push socket, "join", %{status: "connected"}
    {:noreply, socket}
  end
  def handle_info(:ping, socket) do
    push socket, "new:msg", %{user: "SYSTEM", body: "ping"}
    {:noreply, socket}
  end

  def terminate(reason, _socket) do
    Logger.debug"> leave #{inspect reason}"
    :ok
  end

  def handle_in("new:msg", msg, socket) do
    broadcast! socket, "new:msg", %{user: msg["user"], body: msg["body"]}
    {:reply, {:ok, %{msg: msg["body"]}}, assign(socket, :user, msg["user"])}
  end
end
```

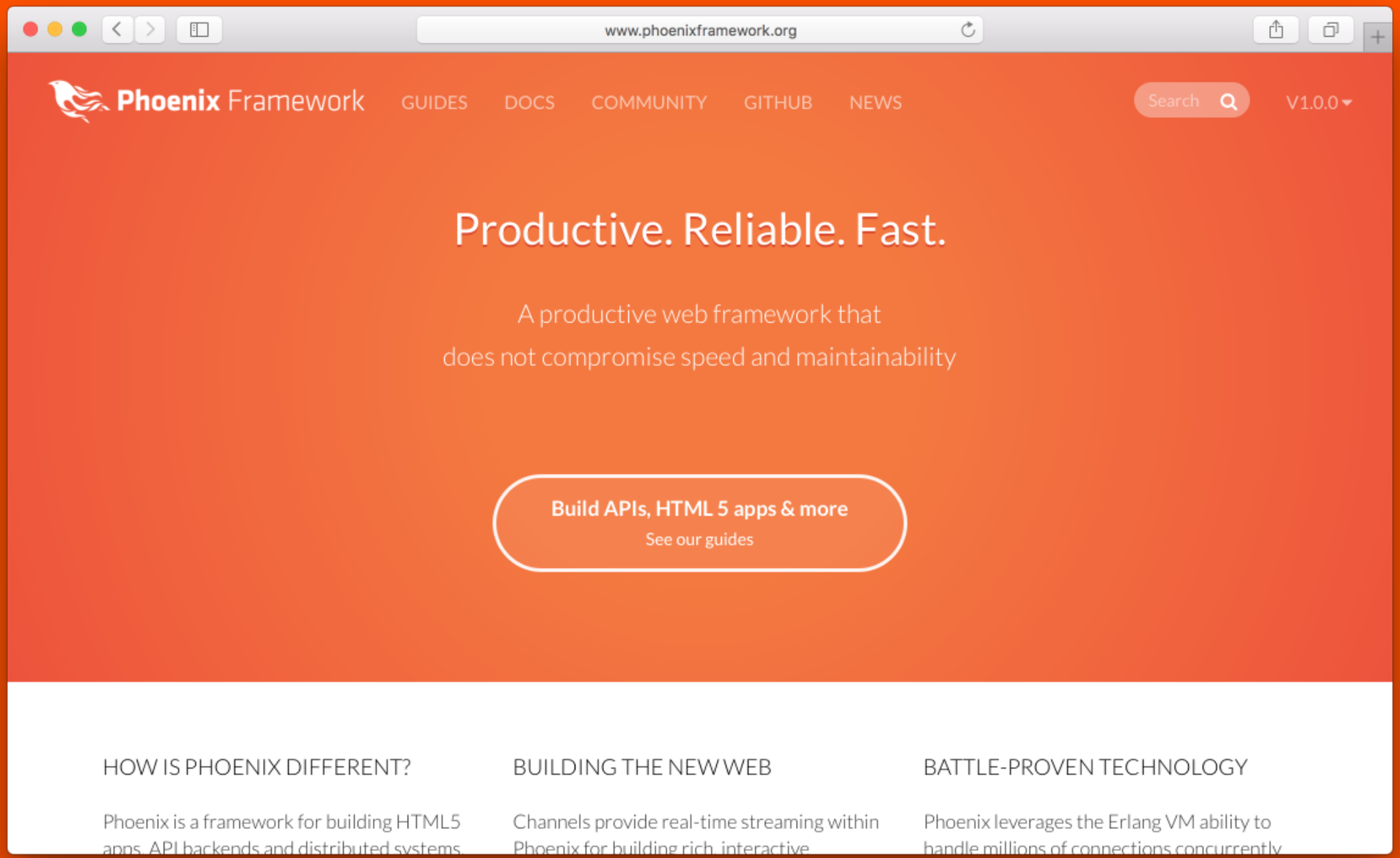
RoomChannel

```
  def handle_info({:after_join, msg}, socket) do
    broadcast! socket, "user:entered", %{user: msg["user"]}
    push socket, "join", %{status: "connected"}
    {:noreply, socket}
  end
  def handle_info(:ping, socket) do
    push socket, "new:msg", %{user: "SYSTEM", body: "ping"}
    {:noreply, socket}
  end

  def terminate(reason, _socket) do
    Logger.debug"> leave #{inspect reason}"
    :ok
  end

  def handle_in("new:msg", msg, socket) do
    broadcast! socket, "new:msg", %{user: msg["user"], body: msg["body"]}
    {:reply, {:ok, %{msg: msg["body"]}}, assign(socket, :user, msg["user"])}
  end
end
```

Phoenix.js

```
import {Socket} from “phoenix”

let socket = new Socket("/socket", {
  logger: ((kind, msg, data) => { console.log(`${kind}: ${msg}`, data) })
})

socket.connect({user_id: “123"})

var chan = socket.channel("rooms:lobby", {})

chan.join().receive("ignore", () => console.log("auth error"))
           .receive("ok", () => console.log("join ok"))
           .after(10000, () => console.log("Connection interruption”))

chan.onError(e => console.log("something went wrong", e))
chan.onClose(e => console.log("channel closed", e))

chan.push("new:msg", {user: "scrogson", body: “Hallo!"})

chan.on("new:msg", msg => {
  $messages.append(this.messageTemplate(msg))
})
```

Check out the guides!



http://phoenixframework.org

Danke!

@scrogson