Elixir - Simple Phoenix App

Together we will write a simple elixir web app using Phoenix framework. Phoenix apps usually run on Postgres database, in our example it was removed in order to simplify the setup - so no DB required.
Phoenix supports MVC, yet all the layers can be easly removed - it is not uncommon to build “one controller just for authentication app” using GraphQL or remove View layer of API only applications. Phoenix is one of the most powerful framework for building web apps - coming with best socket support, ability to adapt, and high performance (empty response time is about 200 **µs**, most requests shouldn’t take more than few **ms** to process - and the choke point is database!)

1) Open project in your favourite text editor, open terminal, prepare browser

2) Run the following commands:

$ mix deps.get - download the project dependencies - u can look them up in mix.exs file
$ mix test - run the test located in test folder and check if they pass
$ mix phx.server - run the server on localhost:4000

3) Check app behaviour:

For simplicity the project uses ets local storage without supervisor, which is limited to process lifecycle. This means the data will be removed after application restart or upon ets process exception.

Right now app should be up and running
localhost:4000/api/v1/blog/get_all_posts should return list of posts

4) Project Root:

_build - binaries
config - configuration scripts (.exs files)
deps - downloaded packages
lib - code!

test - tests and related logic
mix.exs - our mix setup
mix.lock - mix generated package info

5) Project /lib:

test_app - contains main application logic
test_app_web - contains web related logic (Controllers, Channels, Router and Endpoint)

6) Go to /lib/test_app_web/controllers/blog_controller.ex

it contains all the methods which are called by pattern matching requests
conn - contains our connaction data
params - can be passed by querystring or as request params

Controller calles coresponding logic from test_app folder (**blog.ex**)

We use "**alias**" to alias **TestApp.Blog** as **Blog** module, and **TestApp.Blog.Post** as **Post**

7) Go to test_app/blog/post.ex

This is definition of our **%Post{}** struct

8) Go to test_app/blog.ex

This file can be treated as UI for our app Blog context

9) Add some posts:

You can add post by sending GET request to proper endpoint and passing params in Querystring. Simply visit following page:
localhost:4000/api/v1/blog/add_post**?title=title&body=body**

It returns post and its id - now you can lookup the post by visiting
localhost:4000/api/v1/blog/add_post**?id=<id-from-last-request>**

10) Request lifecycle in our app:

Request are handled by **Endpoint** (we will skip that) and passed to **Router.**
Router contains functions we want to call for request handling. It calls **Controller** function (note: scopes use pattern matching, so we are sure the controller will be last and only item on Stack, letting it return response).
Controller pattern match request params, calling proper logic and returning response.

10) Create delete post function:

0) Add empty **controller** function delete_post(conn, _params)
1) Bind controller function in **Router** - use get for now
2) Add pattern matching to controller - request should contain id - so replace params with %{“id” => id}
3) Call the corresponding Blog function (delete_post_by_id) from controller with id as param, it should return “:ok” if post was deleted. You can check it with “:ok=Blog.delete_post_by_id(id)”
4) Add logic to Blog.delete_post_by_id(id) function
5) Render response in controller - it should contain deleted post id

localhost:4000/api/v1/blog/delete_post**?id=<post-to-delete-id>**
should return deleted post id as json response **{id: <post-to-delete-id>}**

11) Add update post function:

localhost:4000/api/v1/blog/update_post**?id=<post-to-update-id>&title=newtitle&body=newbody**
should return updated post as response

12) Add tests in test/test_app/blog_test.exs for completed tasks:

usually tests are written before coding

For post delation:

```
  test "delete_post_by_id/1" do
        post = %Post{title: "Title", body: "Body"}

        {:ok, created_post} = Blog.add_post(post)
```

```
        assert :ok = Blog.delete_post_by_id(created_post.id)
  end
```

For post update:

```
  test "update_post/1" do
        post = %Post{title: "Title", body: "Body"}
        {:ok, created_post} = Blog.add_post(post)

        post_update = %Post{id: created_post.id, title: “NewTitle”, body: “NewBody”}

        assert {:ok, ^post_update} = Blog.update_post(post_update)
  end
```

---

assert {:ok, **^**post_update} = Blog.update_post(post_update)
“^” means we pattern match agains already existing variable, so this equals

```
assert {:ok,
%Post{id: created_post.id, title: “NewTitle”, body: “NewBody”}
} = Blog.update_post(post_update)
```

13*) Add tests for controller functions - use already existing test as guide

14*) Refactor app to use proper HTTP Request Method, change controller tests. From now it will be easier to check app behaviour by running tests, rather than making requests yourself. You can use curl or REST client for manual testing