Building and testing a Phoenix JSON API

Paul Smith – August 2, 2016 UPDATED ON March 6, 2019

ELIXIR, TESTING, PHOENIX

Note: This blog post is for Phoenix 1.2. If you're on Phoenix >= 1.3 check out the Phoenix guides.

Since writing the first version of this guide a lot has changed in Phoenix. In this post we'll show you how to build and test the index and show actions of a JSON API with Phoenix 1.2. We'll show you some new techniques we've learned since the last time. You'll learn:

- How to use the built-in Phoenix test helpers
- How to use views to render JSON
- How to unit test your views
- How to use those unit tests to reduce duplication in your controller tests.

Setup

To get started, make sure you follow the Phoenix installation instructions, and then

run `mix phoenix.new todos` to create the project. After Phoenix is setup, install ExMachina in your project. Remember to include `import Todos.Factory` in your `Todos.ConnCase` so that you have access to the functions in your factory.

Add a Factory and TodoControllerTest

We want to show a list of todos from our API, so let's start off with a controller test and an ExMachina factory for inserting todos.

This factory will make it so that we can easily generate test data.

```
# Create the factory in test/support/factory.ex
defmodule Todos.Factory do
  use ExMachina.Ecto, repo: Todos.Repo

  def todo_factory do
    %Todos.Todo{
      title: "Something I need to do",
      description: "List of steps I need to complete"
    }
  end
end
```

```
# Create this test in test/controllers/todo_controller_test.exs
defmodule Todos.TodoControllerTest do
  use Todos.ConnCase

  test "#index renders a list of todos" do
    conn = build_conn()
    todo = insert(:todo)

    conn = get conn, todo_path(conn, :index)

    assert json_response(conn, 200) == %{
      "todos" => [%{
```

```
          "title" => todo.title,
          "description" => todo.description,
          "inserted_at" => Ecto.DateTime.to_iso8601(todo.inserted_a
          "updated_at" => Ecto.DateTime.to_iso8601(todo.updated_at)
        }]
      }
    end
  end
```

We `use Todos.ConnCase` which is an automatically generated module in `test/support/conn_case.ex`. It handles importing and setting up the functions you need to write controller tests. Open up that file to view more about what it does.

In our test we first call `build_conn/0` to create a new connection. Then we create a todo in the database. After that we perform a GET request to the todo index path. Later we assert that the returned connection had a successful status of 200 and had a JSON response body.

We call `Ecto.DateTime.to_iso8601` to make sure the date times are formatted in a standard way. ISO8601 is a standardized date format that can be used from most programming languages so it makes an ideal fit for JSON APIs.

If we run `mix test` now, it will fail to compile because we are missing the Todos.Todo model, and the todo_path in the router.

Add the model and a migration for the database.

First let's add a model for our todos. You could use `mix phoenix.gen.model` for this, but we're going to generate a migration and write the model ourselves. This will help us see what's going on, and makes sure we only write what we need.

Let's add a schema for `Todos.Todo` at `web/models/todo.ex`:

```
defmodule Todos.Todo do
  use Todos.Web, :model

  schema "todos" do
    field :title
    field :description

    timestamps
  end
end
```

And then we create a migration with `mix ecto.gen.migration create_todos`.

```
# In the newly generated migration file
defmodule Todos.Repo.Migrations.CreateTodos do
  use Ecto.Migration

  def change do
    create table(:todos) do
      add :title, :string
      add :description, :text

      timestamps
    end
  end
end
```

Now when you run `mix test` you will still see the error about the missing `todo_path` helper. Let's take care of that now.

Add a route and a controller

Let's add a new todos route to our router:

```
# In web/router.ex

defmodule Todos.Router do
  # Add this scope for handling API requests
  scope "/api", Todos do
    pipe_through :api

    resources "/todos", TodoController, only: [:index]
  end
end
```

The `resources` macro generates RESTful routes for you. In this case, we just need an index action, but we'll add more later. Piping it through the default `:api` scope will make sure that the response is encoded as JSON.

Next we'll add the `TodoController`.

```
# In web/controllers/todo_controller.ex
defmodule Todos.TodoController do
  use Todos.Web, :controller

  alias Todos.Todo

  def index(conn, _params) do
    todos = Repo.all(Todo)
    render conn, "index.json", todos: todos
  end
end
```

If we run `mix test` again we'll see that we have not defined a `Todos.TodoView` to render our todos.

Add a view to render the todos

Views are used in Phoenix to organize rendering. You can use a view to render HTML templates, JSON (as we are about to do), or both. Views are made up of regular Elixir functions, which makes them easy to test, and easy to understand.

Here's the view we'll use to render our Todos.

```
defmodule Todos.TodoView do
  use Todos.Web, :view

  def render("index.json", %{todos: todos}) do
    %{
      todos: Enum.map(todos, &todo_json/1)
    }
  end

  def todo_json(todo) do
    %{
      title: todo.title,
      description: todo.description,
      inserted_at: todo.inserted_at,
      updated_at: todo.updated_at
    }
  end
end
```

In the render call for `"index.json"` we nest our list of todos inside of a `todos` key. Then we render each todo using the `todo_json/1` function for `"todo.json"`. The syntax `&todo_json/1` is called function capturing. This is telling the program to pass each todo as the first argument to the `todo_json/1` function. You could have done the same thing like this `Enum.map(todos, fn(todo) -> todo_json(todo) end)`.

You'll also notice that we don't need to do anything special with `inserted_at`

and `updated_at`. That's because those fields are `Ecto.DateTime`s and those structs are automatically serialized using `Ecto.DateTime.to_iso8601`.

Checking serialization with unit tests

Our test should pass now, but what happens when we add another endpoint we want to test?

```
defmodule Todos.TodoControllerTest do
  use Todos.ConnCase

  test "#index renders a list of todos" do
    conn = build_conn()
    todo = insert(:todo)

    conn = get conn, todo_path(conn, :index)

    assert json_response(conn, 200) == %{
      "todos" => [%{
        "title" => todo.title,
        "description" => todo.description,
        "inserted_at" => Ecto.DateTime.to_iso8601(todo.inserted_a
        "updated_at" => Ecto.DateTime.to_iso8601(todo.updated_at)
      }]
    }
  end

  test "#show renders a single todo" do
    conn = build_conn()
    todo = insert(:todo)

    conn = get conn, todo_path(conn, :show, todo)

    assert json_response(conn, 200) == %{
      "todo" => %{
        "title" => todo.title,
        "description" => todo.description,
        "inserted_at" => Ecto.DateTime.to_iso8601(todo.inserted_a
```

```
        "updated_at" => Ecto.DateTime.to_iso8601(todo.updated_at)
      }
    }
  end
end
```

Now we're duplicating the assertion in two places. We could extract a function for this, but instead we'll do something to clean this up and make future updates easier.

Write a view test for TodoView

Instead of testing the output in the controller test, we'll do it in a view test.

```
# In test/views/todo_view_test.exs
defmodule Todos.TodoViewTest do
  use ModelCase
  import Todos.Factory
  alias Todos.TodoView

  test "todo_json" do
    todo = insert(:todo)

    rendered_todo = TodoView.todo_json(todo)

    assert rendered_todo == %{
      title: todo.title,
      description: todo.description,
      inserted_at: todo.inserted_at,
      updated_at: todo.updated_at
    }
  end

  test "index.json" do
    todo = insert(:todo)

    rendered_todos = TodoView.render("index.json", %{todos: [todo
```

```
    assert rendered_todos == %{
      todos: [TodoView.todo_json(todo)]
    }
  end

  test "show.json" do
    todo = insert(:todo)

    rendered_todo = TodoView.render("show.json", %{todo: todo})

    assert rendered_todo == %{
      todo: TodoView.todo_json(todo)
    }
  end
end
```

Notice that we don't call `Ecto.DateTime.to_iso8601(todo.inserted_at)` or `Ecto.DateTime.to_iso8601(todo.updated_at)` when we test the rendered todo. That's because the view returns an Elixir map, and has not been encoded to JSON. At some point later on in a request, Phoenix will encode the Map as JSON and will convert the `Ecto.DateTime` into an ISO8601 string for us automatically.

Adding the show action to the controller and view

Let's add a show action to the controller:

```
defmodule Todos.TodoController do
  # Omitted for brevity

  def show(conn, %{"id" => id}) do
    todo = Repo.get!(Todo, id)
    render(conn, "show.json", todo: todo)
  end
end
```

This is very similar to the index action, except that we get a specific record by id, and we render `"show.json"`.

Now let's add a show handler to the `Todos.TodoView`

```
defmodule Todos.TodoView do
  def render("show.json", %{todo: todo}) do
    %{todo: todo_json(todo)}
  end
end
```

Now our tests should pass. But let's clean them up so we don't have so much duplication in our tests.

Cleaning up the controller tests

Instead of individually testing the JSON returned by each action, let's use our views.

```
defmodule Todos.TodoControllerTest do
  use Todos.ConnCase

  test "#index renders a list of todos" do
    conn = build_conn()
    todo = insert(:todo)

    conn = get conn, todo_path(conn, :index)

    assert json_response(conn, 200) == render_json("index.json",
  end

  test "#show renders a single todo" do
    conn = build_conn()
    todo = insert(:todo)
```

```
    conn = get conn, todo_path(conn, :show, todo)

    assert json_response(conn, 200) == render_json("show.json", t
  end

  defp render_json(template, assigns) do
    assigns = Map.new(assigns)

    Todos.TodoView.render(template, assigns)
    |> Poison.encode!
    |> Poison.decode!
  end
end
```

Since we unit tested our views we can check that the JSON response matches the JSON generated by view. We encode and then decode the output from the view so that we get the same response back that we get from `json_response`.

You may be wondering, is this that much better? It seems very similar to what was there before, but it gives us a lot of power in the future. Let's see how in the next section.

Using our views elsewhere

Let's say we wanted to add an `/api/users` endpoint that renders a user's name, email and list of todos. Since we have unit tested our `TodoView`, testing this would be pretty simple. It's simple because views are made up of regular functions, so we can just call it from our unit test just as we would with any other function.

```
# This is just an example, so you don't need to save it anywhere
defmodule Todos.UserViewTest do
  use Todos.ModelCase
  alias Todos.UserView
```

```
  test "show.json" do
    todo = insert(:todo)
    user = insert(:user, todos: [todo])

    rendered_user = UserView.render("show.json", %{user: user})

    assert rendered_user == %{
      user: %{
        name: user.name,
        email: user.email,
        # TodoView.todo_json is tested, so we know it outputs wha
        todos: [Todos.TodoView.todo_json(todo)]
      }
    }
  end
end

defmodule Todos.UserControllerTest do
  use Todos.ConnCase

  test "#show renders a user" do
    conn = build_conn()
    user = insert(:user, todos: [insert(:todo)])

    conn = get conn, user_path(conn, :show, user.id)

    assert json_response(conn, 200) == render_json("show.json", u
  end

  defp render_json(template, assigns) do
    assigns = Map.new(assigns)

    Todos.UserView.render(template, assigns)
    |> Poison.encode!
    |> Poison.decode!
  end
end
```

Wrapping up

You may have noticed that the `render_json` could lead to a lot of duplication across your controller tests. Let's create a module that we import in all of our controller tests to help with this.

```
# Add this to test/support/conn_case_helper.ex
defmodule Todos.ConnCaseHelper do
  def render_json(view, template, assigns) do
    view.render(template, assigns) |> format_json
  end

  defp format_json(data) do
   data |> Poison.encode! |> Poison.decode!
 end
end

# In test/support/conn_case.ex
defmodule Todos.ConnCase do
  using do
    quote do
      # Add this to import the helpers in all your controller tes
      import Todos.ConnCaseHelper
    end
  end
end
```

Now in your tests you can write:

```
defmodule Todos.UserControllerTest do
  use Todos.ConnCase
  alias Todos.UserView

  test "#show renders a user" do
    conn = build_conn()
    user = insert(:user, todos: [insert(:todo)])

    conn = get conn, user_path(conn, :show, user.id)
```

```
      assert json_response(conn, 200) == render_json(UserView, "sho
    end
  end
```

Now this can be used in all your controller tests.

If you enjoyed this post, you might also like:

Make Phoenix Even Faster with a GenServer-backed Key Value Store

ExMachina for Elixir: Factories with a Functional Twist

Testing Elixir Plugs

thoughtbot loves Elixir. Check out our growing body of Elixir open source work and see why we think Phoenix is the next step for a good number of Rubyists.

Open Source

Argo
Bourbon
Capybara Webkit
Clearance
Dotfiles
Factory Bot
Laptop
Suspenders
More...

Locations

Austin, TX
Boston, MA
London, UK
New York, NY
Raleigh/Durham, NC
San Francisco, CA

Podcasts

The Bike Shed
Build Phase
Giant Robots
Tentative