Phoenix

v1.7.21

Table of contents

- Deployment
  - Introduction to Deployment
  - Deploying with Releases
  - Deploying on Gigalixir
  - Deploying on Fly.io
  - Deploying on Heroku
- Cheatsheets
  - Routing cheatsheet
- How-to's
  - Custom Error Pages
  - File Uploads
  - Using SSL
  - Writing a Channels Client
- Modules
  - Phoenix
  - Phoenix.Channel
  - Phoenix.Controller
  - Phoenix.Endpoint
  - Phoenix.Flash
  - Phoenix.Logger
  - Phoenix.Naming
  - Phoenix.Param
  - Phoenix.Presence
  - Phoenix.Router
  - Phoenix.Socket
  - Phoenix.Token
  - Phoenix.VerifiedRoutes
  - Testing
    - Phoenix.ChannelTest
    - Phoenix.ConnTest
  - Adapters and Plugs
    - Phoenix.CodeReloader
    - Phoenix.Endpoint.Cowboy2Adapter
    - Phoenix.Endpoint.SyncCodeReloadPlug
  - Digester
    - Phoenix.Digester.Compressor

        - Phoenix.Digester.Gzip
    - Socket
        - Phoenix.Socket.Broadcast
        - Phoenix.Socket.Message
        - Phoenix.Socket.Reply
        - Phoenix.Socket.Serializer
        - Phoenix.Socket.Transport
    - Exceptions
        - Phoenix.ActionClauseError
        - Phoenix.MissingParamError
        - Phoenix.NotAcceptableError
        - Phoenix.Router.MalformedURIError
        - Phoenix.Router.NoRouteError
        - Phoenix.Socket.InvalidMessageError
- Mix Tasks
    - mix local.phx
    - mix phx
    - mix phx.digest
    - mix phx.digest.clean
    - mix phx.gen
    - mix phx.gen.auth
    - mix phx.gen.cert
    - mix phx.gen.channel
    - mix phx.gen.context
    - mix phx.gen.embedded
    - mix phx.gen.html
    - mix phx.gen.json
    - mix phx.gen.live
    - mix phx.gen.notifier
    - mix phx.gen.presence
    - mix phx.gen.release
    - mix phx.gen.schema
    - mix phx.gen.secret
    - mix phx.gen.socket
    - mix phx.new
    - mix phx.new.ecto

- mix phx.new.web
- mix phx.routes
- mix phx.server

Changelog for v1.7

See the upgrade guide to upgrade from Phoenix 1.6.x.

Phoenix v1.7 requires Elixir v1.11+ & Erlang v22.1+.

Introduction of Verified Routes

Phoenix 1.7 includes a new `Phoenix.VerifiedRoutes` feature which provides `~p` for route generation with compile-time verification.

Use of the `sigil_p` macro allows paths and URLs throughout your application to be compile-time verified against your Phoenix router(s). For example the following path and URL usages:

```
<.link href={~p"/sessions/new"} method="post">Log
in</.link>

redirect(to: url(~p"/posts/#{post}"))
```

Will be verified against your standard `Phoenix.Router` definitions:

```
get "/posts/:post_id", PostController, :show
post "/sessions/new", SessionController, :create
```

Unmatched routes will issue compiler warnings:

```
warning: no route path for AppWeb.Router matches
"/postz/#{post}"
  lib/app_web/controllers/post_controller.ex:100:
AppWeb.PostController.show/2
```

Note: Elixir v1.14+ is required for comprehensive warnings. Older versions will work properly and warn on new compilations, but changes to the router file will not issue new warnings.

This feature replaces the `Helpers` module generated in your Phoenix router, but helpers will continue to work and be generated. You can disable router helpers by passing the `helpers: false` option to `use Phoenix.Router`.

phx.new revamp

The `phx.new` application generator has been improved to rely on function components for both Controller and LiveView rendering, ultimately simplifying the rendering stack of Phoenix applications and providing better reuse.

New applications come with a collection of well-documented and accessible core components, styled with Tailwind CSS by default. You can opt-out of Tailwind CSS with the `--no-tailwind` flag (the Tailwind CSS classes are kept in the generated components as reference for future styling).

1.7.21 (2025-03-27)

Bug fixes

- Fix socket sometimes not reconnecting after pagehide/pageshow (#6103)
- Check if priv folder exists before re-linking in CodeReloader (#6124)

Enhancements

- Relax LiveView dependency for new projects

1.7.20 (2025-02-20)

Enhancements

- Add `[:phoenix, :socket_drain]` telemetry event to track socket draining and use it for logging
- Address Elixir 1.18 warnings in phx.new
- Add `PHX_NEW_CACHE_DIR` env var for cached `phx.new` builds

Bug fixes

- Fix code reloader error when `mix.lock` is touched without its content changing

1.7.19 (2025-01-31)

Enhancements

- [phx.new] - bind to `0.0.0.0` in `dev.exs` if phx.new is being run inside a docker container. This exposes the container's phoenix server to the host so that it is accessible over port forwarding.

1.7.18 (2024-12-10)

Enhancements

- Use new interpolation syntax in generators
- Update gettext in generators to 0.26

1.7.17 (2024-12-03)

Enhancements

- Use LiveView 1.0.0 for newly generated applications

1.7.16 (2024-12-03)

Bug fixes

- Fix required Elixir version in mix.exs

1.7.15 (2024-12-02)

Enhancements

- Support phoenixframework.org installer

1.7.14 (2024-06-18)

Bug fixes

- Revert "Add `follow_redirect/2` to Phoenix.ConnTest" (#5797) as this conflicts with `follow_redirect/2` in LiveView, which is imported with ConnTest by default

1.7.13 (2024-06-18)

Bug fixes

- Fix Elixir 1.17 warning in Cowboy2Adapter
- Fix verified routes emitting diagnostics without file and position

JavaScript Client Bug Fixes

- Fix error when `sessionStorage` is not available on global namespace

Enhancements

- Add `follow_redirect/2` to Phoenix.ConnTest
- Use LiveView 1.0.0-rc for newly generated applications
- Use new `Phoenix.Component.used_input?` for form errors in generated `core_components.ex`
- Allow `mix ecto.setup` from the umbrella root

- Bump Endpoint static cache manifest on `config_change` callback

1.7.12 (2024-04-11)

JavaScript Client Bug Fixes

- Fix all unjoined channels from being removed from the socket when channel leave is called on any single unjoined channel instance

Enhancements

- [phx.gen.auth] Add enhanced session fixation protection. For applications whichs previously used `phx.gen.auth`, the following line can be added to the `renew_session` function in the auth module:

```
  defp renew_session(conn) do
+   delete_csrf_token()

    conn
    |> configure_session(renew: true)
    |> clear_session()
```

  Note: because the session id is in a http-only cookie by default, the only way to perform this attack prior to this change is if your application was already vulnerable to an XSS attack, which itself grants more escalated "privileges" than the CSRF fixation.

JavaScript Client Enhancements

- Only memorize longpoll fallback for browser session if WebSocket never had a successful connection

1.7.11 (2024-02-01)

Enhancements

- [phx.new] Default to the Bandit webserver for newly generated applications
- [phx.new] Enable longpoll transport by default and auto fallback when websocket fails for newly generated applications

JavaScript Client Enhancements

- Support new `longPollFallbackMs` option to auto fallback when websocket fails to connect
- Support new `debug` option to enable verbose logging

Deprecations

- Deprecate the `c:init/2` callback in endpoints in favor of `config/runtime.exs` or in favor of `{Phoenix.Endpoint, options}`

1.7.10 (2023-11-03)

Bug fixes

- [phx.new] Fix `CoreComponents.flash` generating incorrect id's causing flash messages to fail to be closed when clicked

Enhancements

- [Phoenix.Endpoint] Support dynamic port for `Endpoint.url/0`

1.7.9 (2023-10-11)

Bug fixes

- [Phoenix.CodeReloader] - Fix error in code reloader causing compilation errors

- [phx.new] – fix LiveView debug heex configuration being generated when `--no-html` pas passed

1.7.8 (2023-10-09)

Bug fixes

- [Phoenix.ChannelTest] Stringify lists when pushing data
- [Phoenix.Controller] Fix filename when sending downloads with non-ascii names
- [Phoenix.CodeReloader] Remove duplicate warnings on recent Elixir versions
- [Phoenix.CodeReloader] Do not crash code reloader if file information is missing from diagnostic
- [Phoenix.Logger] Do not crash when status is atom
- [phx.gen.release] Fix `mix phx.gen.release --docker` failing with `:http_util` error on Elixir v1.15
- [phx.gen.*] Skip map inputs in generated forms as there is no trivial matching input
- [phx.new] Fix tailwind/esbuild config and paths in umbrella projects
- [phx.new] Do not render `th` for actions if actions are empty

Enhancements

- [Phoenix] Allow latest `plug_crypto`
- [Phoenix.Endpoint] Support dynamic socket drainer configuration
- [Phoenix.Logger] Change socket serializer/version logs to warning
- [Phoenix.VerifiedRoutes] Add support for static resources with fragments in `~p`
- [phx.gen.schema] Support `--repo` and `--migration-dir` flags
- [phx.new] Allow `<.input type="checkbox">` without `value` attr in core components
- [phx.new] Allow UTC datetimes in the generators
- [phx.new] Automatically migrate when release starts when using sqlite 3

- [phx.new] Allow ID to be assigned in flash component
- [phx.new] Add `--adapter` flag for generating application with bandit
- [phx.new] Include DNSCluster for simple clustering
- [phx.routes] Support `--method` option

1.7.7 (2023-07-10)

Enhancements

- Support incoming binary payloads to channels over longpoll transport

1.7.6 (2023-06-16)

Bug Fixes

- Support websock_adapter 0.5.3

Enhancements

- Allow using Phoenix.ChannelTest socket/connect in another process

1.7.5 (2023-06-15)

Bug Fixes

- Fix LongPoll error when draining connections

1.7.4 (2023-06-15)

Bug Fixes

- Fix the WebSocket draining sending incorrect close code when draining causing LiveViews to reload the page instead of reconnecting

1.7.3 (2023-05-30)

Enhancements

- Use LiveView 0.19 for new apps

Bug Fixes

- Fix compilation error page on plug debugger showing obscure error when app fails to compile
- Fix warnings being printed twice in route verification

1.7.2 (2023-03-20)

Enhancements

- [Endpoint] Add socket draining for batched and orchestrated Channel/LiveView socket shutdown
- [code reloader] Improve the compilation error page to remove horizontal scrolling and include all warnings and errors from compilation
- [phx.new] Support the `--no-tailwind` and `--no-esbuild` flags
- [phx.new] Move heroicons to assets/vendor
- [phx.new] Simplify core modal to use the new JS.exec instruction to reduce footprint
- [sockets] Allow custom csrf_token_keys in WebSockets

1.7.1 (2023-03-02)

Enhancements

- [phx.new] Embed heroicons in app.css bundle to optimize usage

1.7.0 (2023-02-24)

Bug Fixes

- Fix race conditions in the longpoll transport by batching messages

1.7.0-rc.3 (2023-02-15)

Enhancements

- Use stream based collections for `phx.gen.live` generators
- Update `phx.gen.live` generators to use `Phoenix.Component.to_form`

1.7.0-rc.2 (2023-01-13)

Bug Fixes

- [Router] Fix routing bug causing incorrect matching order on similar routes
- [phx.new] Fix installation hanging in some cases

1.7.0-rc.1 (2023-01-06)

Enhancements

- Raise if using verified routes outside of functions
- Add tailwind.install/esbuild.install to mix setup

Bug Fixes

- [Presence] fix task shutdown match causing occasional presence errors
- [VerifiedRoutes] Fix expansion causing more compile-time deps than necessary
- [phx.gen.auth] Add password inputs to password reset edit form

- [phx.gen.embedded] Fixes missing :references generation to phx.gen.embedded
- Fix textarea rendering in core components
- Halt all sockets on intercept to fix longpoll response already sent error

1.7.0-rc.0 (2022-11-07)

Deprecations

- `Phoenix.Controller.get_flash` has been deprecated in favor of the new `Phoenix.Flash` module, which provides unified flash access

Enhancements

- [Router] Add `Phoenix.VerifiedRoutes` for `~p`-based route generation with compile-time verification.
- [Router] Support `helpers: false` to `use Phoenix.Router` to disable helper generation
- [Router] Add `--info [url]` switch to `phx.routes` to get route information about a url/path
- [Flash] Add `Phoenix.Flash` for unfied flash access

JavaScript Client Bug Fixes

- Fix heartbeat being sent after disconnect and causing abnormal disconnects

v1.6

The CHANGELOG for v1.6 releases can be found in the v1.6 branch.

Overview

Phoenix is a web development framework written in Elixir which implements the server-side Model View Controller (MVC) pattern. Many of its components and concepts will seem familiar to those of us with experience in other web frameworks like Ruby on Rails or Python's Django.

Phoenix provides the best of both worlds - high developer productivity *and* high application performance. It also has some interesting new twists like channels for implementing realtime features and pre-compiled templates for blazing speed.

If you are already familiar with Elixir, great! If not, there are a number of places to learn. The Elixir guides and the Elixir learning resources page are two great places to start.

The guides that you are currently looking at provide an overview of all parts that make Phoenix. Here is a rundown of what they provide:

- Introduction - the guides you are currently reading. They will cover how to get your first application up and running
- Guides - in-depth guides covering the main components in Phoenix and Phoenix applications
- Authentication - in-depth guide covering how to use `mix phx.gen.auth`
- Real-time components - in-depth guides covering Phoenix's built-in real-time components
- Testing - in-depth guides about testing
- Deployment - in-depth guides about deployment

- How-to's - a collection of articles on how to achieve certain things with Phoenix

If you would prefer to read these guides as an EPUB, click here!

Note, these guides are not a step-by-step introduction to Phoenix. If you want a more structured approach to learning the framework, we have a large community and many books, courses, and screencasts available. See our community page for a complete list.

Let's get Phoenix installed.

Installation

In order to build a Phoenix application, we will need a few dependencies installed in our Operating System:

- the Erlang VM and the Elixir programming language
- a database - Phoenix recommends PostgreSQL, but you can pick others or not use a database at all
- and other optional packages.

Please take a look at this list and make sure to install anything necessary for your system. Having dependencies installed in advance can prevent frustrating problems later on.

Elixir 1.14 or later

Phoenix is written in Elixir, and our application code will also be written in Elixir. We won't get far in a Phoenix app without it! The Elixir site maintains a great Installation Page to help.

Erlang 24 or later

Elixir code compiles to Erlang byte code to run on the Erlang virtual machine. Without Erlang, Elixir code has no virtual machine to run on, so we need to install Erlang as well.

When we install Elixir using instructions from the Elixir Installation Page, we will usually get Erlang too. If Erlang was not installed along with Elixir, please see the Erlang Instructions section of the Elixir Installation Page for instructions.

Phoenix

To check that we are on Elixir 1.14 and Erlang 24 or later, run:

```
elixir -v
Erlang/OTP 24 [erts-12.0] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false] [dtrace]

Elixir 1.14.0
```

Once we have Elixir and Erlang, we are ready to install the Phoenix application generator:

```
$ mix archive.install hex phx_new
```

The `phx.new` generator is now available to generate new applications in the next guide, called Up and Running. The flags mentioned below are command line options to the generator; see all available options by calling `mix help phx.new`.

PostgreSQL

PostgreSQL is a relational database server. Phoenix configures applications to use it by default, but we can switch to MySQL, MSSQL, or SQLite3 by passing the `--database` flag when creating a new application.

In order to talk to databases, Phoenix applications use another Elixir package, called Ecto. If you don't plan to use databases in your application, you can pass the `--no-ecto` flag.

However, if you are just getting started with Phoenix, we recommend you to install PostgreSQL and make sure it is running. The PostgreSQL wiki has installation guides for a number of different systems.

inotify-tools (for Linux users)

Phoenix provides a very handy feature called Live Reloading. As you change your views or your assets, it automatically reloads the page in the browser. In order for this functionality to work, you need a filesystem watcher.

macOS and Windows users already have a filesystem watcher, but Linux users must install inotify-tools. Please consult the inotify-tools wiki for distribution-specific installation instructions.

Summary

At the end of this section, you must have installed Elixir, Hex, Phoenix, and PostgreSQL. Now that we have everything installed, let's create our first Phoenix application and get up and running.

Up and Running

There are two mechanisms to start a new Phoenix application: the express option, supported on some OSes, and via `mix phx.new`. Let's check it out.

Phoenix Express

A single command will get you up and running in seconds:

For macOS/Ubuntu:

```
$ curl https://new.phoenixframework.org/myapp | sh
```

For Windows PowerShell:

```
> curl.exe -fsSO
https://new.phoenixframework.org/myapp.bat; .\myapp.bat
```

The above will install Erlang, Elixir, and Phoenix, and generate a fresh Phoenix application. It will also automatically pick one of PostgreSQL or MySQL as the database, and fallback to SQLite if none of them are available. Once the command above, it will open up a Phoenix application, with the steps necessary to complete your installation.

> Your Phoenix application name is taken from the path.

If your operating system is not supported, or the command above fails, don't fret! You can still start your Phoenix application using `mix phx.new`.

Via `mix phx.new`

In order to create a new Phoenix application, you will need to install Erlang, Elixir, and Phoenix. See the Installation Guide for more information. If you share your application with someone, they will also need to follow the Installation Guide steps to set it all up.

Once you are ready, you can run `mix phx.new` from any directory in order to bootstrap our Phoenix application. Phoenix will accept either an absolute or relative path for the directory of our new project. Assuming that the name of our application is `hello`, let's run the following command:

```
$ mix phx.new hello
```

By default, `mix phx.new` includes a number of optional dependencies, for example:

- Ecto for communicating with a data store, such as PostgreSQL, MySQL, and others. You can skip this with `--no-ecto`.
- Phoenix.HTML, TailwindCSS, and Esbuild for HTML applications. You can skip them with the `--no-html` and `--no-assets` flags.
- Phoenix.LiveView for building realtime and interactive web applications. You can skip this with `--no-live`.

Read the Mix Tasks Guide for the full list of things that can be excluded, among other options.

```
mix phx.new hello
* creating hello/config/config.exs
* creating hello/config/dev.exs
* creating hello/config/prod.exs
...

Fetch and install dependencies? [Yn]
```

Phoenix generates the directory structure and all the files we will need for our application.

> Phoenix promotes the usage of git as version control software: among the generated files we find a `.gitignore`. We can `git init` our repository, and immediately add and commit all that hasn't been marked ignored.

When it's done, it will ask us if we want it to install our dependencies for us. Let's say yes to that.

```
Fetch and install dependencies? [Yn] Y
* running mix deps.get
* running mix assets.setup
* running mix deps.compile

We are almost there! The following steps are missing:

    $ cd hello

Then configure your database in config/dev.exs and run:

    $ mix ecto.create

Start your Phoenix app with:

    $ mix phx.server

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

    $ iex -S mix phx.server
```

Once our dependencies are installed, the task will prompt us to change into our project directory and start our application.

Phoenix assumes that our PostgreSQL database will have a `postgres` user account with the correct permissions and a password of "postgres". If that isn't the case, please see the Mix Tasks Guide to learn more about the `mix ecto.create` task.

Ok, let's give it a try. First, we'll `cd` into the `hello/` directory we've just created:

```
$ cd hello
```

Now we'll create our database:

```
$ mix ecto.create
Compiling 13 files (.ex)
Generated hello app
The database for Hello.Repo has been created
```

In case the database could not be created, see the guides for the `mix ecto.create` for general troubleshooting.

> Note: if this is the first time you are running this command, Phoenix may also ask to install Rebar. Go ahead with the installation as Rebar is used to build Erlang packages.

And finally, we'll start the Phoenix server:

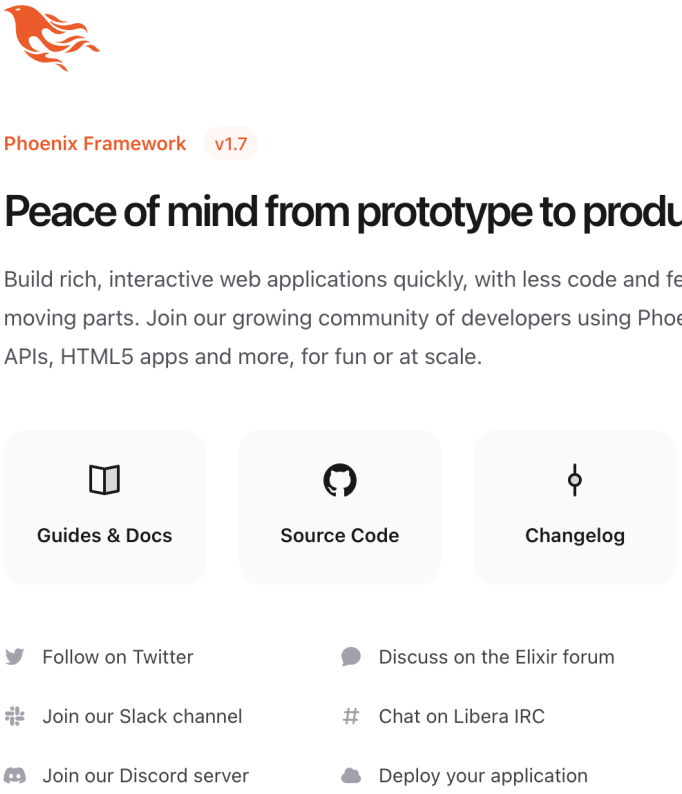
```
$ mix phx.server
[info] Running HelloWeb.Endpoint with cowboy 2.9.0 at
127.0.0.1:4000 (http)
[info] Access HelloWeb.Endpoint at http://localhost:4000
[watch] build finished, watching for changes...
...
```

If we choose not to have Phoenix install our dependencies when we generate a new application, the `mix phx.new` task will prompt us to take the necessary steps when we do want to install them.

```
Fetch and install dependencies? [Yn] n

We are almost there! The following steps are missing:
```

```
    $ cd hello
    $ mix deps.get

Then configure your database in config/dev.exs and run:

    $ mix ecto.create

Start your Phoenix app with:

    $ mix phx.server

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

    $ iex -S mix phx.server
```

By default, Phoenix accepts requests on port 4000. If we point our favorite web browser at http://localhost:4000, we should see the Phoenix Framework welcome page.



If your screen looks like the image above, congratulations! You now have a working Phoenix application. In case you can't see the page above, try accessing it via http://127.0.0.1:4000 and later make sure your OS has defined "localhost" as "127.0.0.1".

To stop it, we hit `ctrl-c` twice.

Now you are ready to explore the world provided by Phoenix! See our community page for books, screencasts, courses, and more.

Alternatively, you can continue reading these guides to have a quick introduction into all the parts that make your Phoenix application. If that's the

case, you can read the guides in any order or start with our guide that explains the Phoenix directory structure.

Community

The Elixir and Phoenix communities are friendly and welcoming. All questions and comments are valuable, so please come join the discussion!

There are a number of places to connect with community members at all experience levels.

- We're on Libera IRC in the #elixir channel.
- Feel free to join and check out the #phoenix channel on Discord.
- Read about bug reports or open an issue in the Phoenix issue tracker.
- Ask or answer questions about Phoenix on Elixir Forum or Stack Overflow.
- Follow the Phoenix Framework on Twitter.

The Security Working Group of the Erlang Ecosystem Foundation also publishes in-depth documents about security best practices for Erlang, Elixir, and Phoenix.

Books

- Programming Phoenix LiveView - Interactive Elixir Web Programming Without Writing Any JavaScript - 2023 (by Bruce Tate and Sophie DeBenedetto)
- Phoenix Tutorial (Phoenix 1.6) - Free to read online
- Real-Time Phoenix - Build Highly Scalable Systems with Channels (by Stephen Bussey - 2020)

- Programming Phoenix 1.4 (by Bruce Tate, Chris McCord, and José Valim - 2019)
- Phoenix in Action (by Geoffrey Lessel - 2019)
- Phoenix Inside Out - Book Series (by Shankar Dhanasekaran - 2017). First book of the series Mastering Phoenix Framework is free to read online
- Functional Web Development with Elixir, OTP, and Phoenix Rethink the Modern Web App (by Lance Halvorsen - 2017)

Screencasts/Courses

- Phoenix LiveView Free Course (by The Pragmatic Studio - 2023)
- Build It With Phoenix video course (by Geoffrey Lessel - 2023)
- Free Crash Course: Phoenix LiveView (by Productive Programmer - 2023)
- Phoenix on Rails: Elixir and Phoenix for Ruby on Rails developers (by George Arrowsmith - 2023)
- Groxio LiveView: Self Study Program (by Bruce Tate - 2020)
- Alchemist Camp: Learn Elixir and Phoenix by building (2018-2022)
- The Complete Elixir and Phoenix Bootcamp Master Functional Programming Techniques with Elixir and Phoenix while Learning to Build Compelling Web Applications (by Stephen Grider - 2017)
- Discover Elixir & Phoenix (by Tristan Edwards - 2017)
- Phoenix Framework Tutorial (by Tensor Programming - 2017)
- Getting Started with Phoenix (by Pluralsight - 2017)

- LearnPhoenix.tv: Learn how to Build Fast, Dependable Web Apps with Phoenix (2017)
- LearnPhoenix.io: Build Scalable, Real-Time Apps with Phoenix, React, and React Native (2016)

Packages Glossary

By default, Phoenix applications depend on several packages with different purposes. This page is a quick reference of the different packages you may work with as a Phoenix developer.

The main packages are:

- Ecto - a language integrated query and database wrapper
- Phoenix - the Phoenix web framework (these docs)
- Phoenix LiveView - build rich, real-time user experiences with server-rendered HTML. The LiveView project also defines `Phoenix.Component` and the HEEx template engine, used for rendering HTML content in both regular and real-time applications
- Plug - specification and conveniences for building composable modules web applications. This is the package responsible for the connection abstraction and the regular request- response life-cycle

You will also work with the following:

- ExUnit - Elixir's built-in test framework
- Gettext - internationalization and localization through `gettext`
- Swoosh - a library for composing, delivering and testing emails, also used by `mix phx.gen.auth`

When peeking under the covers, you will find those libraries play an important role in Phoenix applications:

- Phoenix HTML - building blocks for working with HTML and forms safely

- Phoenix Ecto - plugs and protocol implementations for using phoenix with ecto
- Phoenix PubSub - a distributed pub/sub system with presence support

When it comes to instrumentation and monitoring, check out:

- Phoenix LiveDashboard - real-time performance monitoring and debugging tools for Phoenix developers
- Telemetry Metrics - common interface for defining metrics based on Telemetry events

Directory structure

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.

When we use `mix phx.new` to generate a new Phoenix application, it builds a top-level directory structure like this:

```
├── _build
├── assets
├── config
├── deps
├── lib
│   ├── hello
│   ├── hello.ex
│   ├── hello_web
│   └── hello_web.ex
├── priv
└── test
```

We will go over those directories one by one:

- `_build` - a directory created by the `mix` command line tool that ships as part of Elixir that holds all compilation artifacts. As we have seen in "Up and Running", `mix` is the main interface to your application. We use Mix to compile our code, create databases, run our server, and more. This directory must not be checked into version control and it can be removed at any time. Removing it will force Mix to rebuild your application from scratch.

- `assets` - a directory that keeps source code for your front-end assets, typically JavaScript and CSS. These sources are automatically bundled by the `esbuild` tool. Static files like images and fonts go in `priv/static`.

- `config` - a directory that holds your project configuration. The `config/config.exs` file is the entry point for your configuration. At the end of the `config/config.exs`, it imports environment specific configuration, which can be found in `config/dev.exs`, `config/test.exs`, and `config/prod.exs`. Finally, `config/runtime.exs` is executed and it is the best place to read secrets and other dynamic configuration.

- `deps` - a directory with all of our Mix dependencies. You can find all dependencies listed in the `mix.exs` file, inside the `defp deps do` function definition. This directory must not be checked into version control and it can be removed at any time. Removing it will force Mix to download all deps from scratch.

- `lib` - a directory that holds your application source code. This directory is broken into two subdirectories, `lib/hello` and `lib/hello_web`. The `lib/hello` directory will be responsible to host all of your business logic and business domain. It typically interacts directly with the database - it is the "Model" in Model-View-Controller (MVC) architecture. `lib/hello_web` is responsible for exposing your business domain to the world, in this case, through a web application. It holds both the View and Controller from MVC. We will discuss the contents of these directories with more detail in the next sections.

- `priv` - a directory that keeps all resources that are necessary in production but are not directly part of your source code. You typically keep database scripts, translation files, images, and more in here. Generated assets, created from files in the `assets` directory, are placed in `priv/static/assets` by default.

- `test` - a directory with all of our application tests. It often mirrors the same structure found in `lib`.

The lib/hello directory

The `lib/hello` directory hosts all of your business domain. Since our project does not have any business logic yet, the directory is mostly empty. You will only find three files:

```
lib/hello
├── application.ex
├── mailer.ex
└── repo.ex
```

The `lib/hello/application.ex` file defines an Elixir application named `Hello.Application`. That's because at the end of the day Phoenix applications are simply Elixir applications. The `Hello.Application` module defines which services are part of our application:

```
children = [
  HelloWeb.Telemetry,
  Hello.Repo,
  {Phoenix.PubSub, name: Hello.PubSub},
  HelloWeb.Endpoint
]
```

If it is your first time with Phoenix, you don't need to worry about the details right now. For now, suffice it to say our application starts a database repository, a PubSub system for sharing messages across processes and nodes, and the application endpoint, which effectively serves HTTP requests. These services are started in the order they are defined and, whenever shutting down your application, they are stopped in the reverse order.

You can learn more about applications in Elixir's official docs for Application.

The `lib/hello/mailer.ex` file holds the `Hello.Mailer` module, which defines the main interface to deliver e-mails:

```
defmodule Hello.Mailer do
  use Swoosh.Mailer, otp_app: :hello
end
```

In the same `lib/hello` directory, we will find a `lib/hello/repo.ex`. It defines a `Hello.Repo` module which is our main interface to the database. If you are using Postgres (the default database), you will see something like this:

```
defmodule Hello.Repo do
  use Ecto.Repo,
    otp_app: :hello,
    adapter: Ecto.Adapters.Postgres
end
```

And that's it for now. As you work on your project, we will add files and modules to this directory.

The lib/hello_web directory

The `lib/hello_web` directory holds the web-related parts of our application. It looks like this when expanded:

```
lib/hello_web
├── controllers
│   ├── page_controller.ex
│   ├── page_html.ex
│   ├── error_html.ex
│   ├── error_json.ex
│   └── page_html
│       └── home.html.heex
├── components
│   ├── core_components.ex
│   ├── layouts.ex
│   └── layouts
│       ├── app.html.heex
│       └── root.html.heex
```

```
├── endpoint.ex
├── gettext.ex
├── router.ex
└── telemetry.ex
```

All of the files which are currently in the `controllers` and `components` directories are there to create the "Welcome to Phoenix!" page we saw in the "Up and running" guide.

By looking at `controller` and `components` directories, we can see Phoenix provides features for handling layouts and HTML and error pages out of the box.

Besides the directories mentioned, `lib/hello_web` has four files at its root. `lib/hello_web/endpoint.ex` is the entry-point for HTTP requests. Once the browser accesses http://localhost:4000, the endpoint starts processing the data, eventually leading to the router, which is defined in `lib/hello_web/router.ex`. The router defines the rules to dispatch requests to "controllers", which calls a view module to render HTML pages back to clients. We explore these layers in length in other guides, starting with the "Request life-cycle" guide coming next.

Through *Telemetry*, Phoenix is able to collect metrics and send monitoring events of your application. The `lib/hello_web/telemetry.ex` file defines the supervisor responsible for managing the telemetry processes. You can find more information on this topic in the Telemetry guide.

Finally, there is a `lib/hello_web/gettext.ex` file which provides internationalization through Gettext. If you are not worried about internationalization, you can safely skip this file and its contents.

The assets directory

The `assets` directory contains source files related to front-end assets, such as JavaScript and CSS. Since Phoenix v1.6, we use `esbuild` to compile assets, which is managed by the `esbuild` Elixir package. The integration

with `esbuild` is baked into your app. The relevant config can be found in your `config/config.exs` file.

Your other static assets are placed in the `priv/static` folder, where `priv/static/assets` is kept for generated assets. Everything in `priv/static` is served by the `Plug.Static` plug configured in `lib/hello_web/endpoint.ex`. When running in dev mode (`MIX_ENV=dev`), Phoenix watches for any changes you make in the `assets` directory, and then takes care of updating your front end application in your browser as you work.

Note that when you first create your Phoenix app using `mix phx.new` it is possible to specify options that will affect the presence and layout of the `assets` directory. In fact, Phoenix apps can bring their own front end tools or not have a front-end at all (handy if you're writing an API for example). For more information you can run `mix help phx.new` or see the documentation in Mix tasks.

If the default esbuild integration does not cover your needs, for example because you want to use another build tool, you can switch to a custom assets build.

As for CSS, Phoenix ships with the Tailwind CSS Framework, providing a base setup for projects. You may move to any CSS framework of your choice. Additional references can be found in the asset management guide.

Request life-cycle

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.

The goal of this guide is to talk about Phoenix's request life-cycle. This guide will take a practical approach where we will learn by doing: we will add two new pages to our Phoenix project and comment on how the pieces fit together along the way.

Let's get on with our first new Phoenix page!

Adding a new page

When your browser accesses http://localhost:4000/, it sends a HTTP request to whatever service is running on that address, in this case our Phoenix application. The HTTP request is made of a verb and a path. For example, the following browser requests translate into:

Browser address bar	Verb Path
http://localhost:4000/	GET /
http://localhost:4000/hello	GET /hello
http://localhost:4000/hello/world	GET /hello/world

There are other HTTP verbs. For example, submitting a form typically uses the POST verb.

Web applications typically handle requests by mapping each verb/path pair into a specific part of your application. This matching in Phoenix is done by the router. For example, we may map "/articles" to a portion of our application that shows all articles. Therefore, to add a new page, our first task is to add a new route.

A new route

The router maps unique HTTP verb/path pairs to controller/action pairs which will handle them. Controllers in Phoenix are simply Elixir modules. Actions are functions that are defined within these controllers.

Phoenix generates a router file for us in new applications at `lib/hello_web/router.ex`. This is where we will be working for this section.

The route for our "Welcome to Phoenix!" page from the previous Up And Running Guide looks like this.

```
  get "/", PageController, :home
```

Let's digest what this route is telling us. Visiting http://localhost:4000/ issues an HTTP `GET` request to the root path. All requests like this will be handled by the `home/2` function in the `HelloWeb.PageController` module defined in `lib/hello_web/controllers/page_controller.ex`.

The page we are going to build will say "Hello World, from Phoenix!" when we point our browser to http://localhost:4000/hello.

The first thing we need to do is to create the page route for a new page. Let's open up `lib/hello_web/router.ex` in a text editor. For a brand new application, it looks like this:

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.Layouts, :root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
```

```
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :home
  end

  # Other scopes may use custom stacks.
  # scope "/api", HelloWeb do
  #   pipe_through :api
  # end

  # ...
end
```

For now, we'll ignore the pipelines and the use of `scope` here and just focus on adding a route. We will discuss those in the Routing guide.

Let's add a new route to the router that maps a `GET` request for `/hello` to the `index` action of a soon-to-be-created `HelloWeb.HelloController` inside the `scope "/" do` block of the router:

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

  get "/", PageController, :home
  get "/hello", HelloController, :index
end
```

A new controller

Controllers are Elixir modules, and actions are Elixir functions defined in them. The purpose of actions is to gather the data and perform the tasks needed for rendering. Our route specifies that we need a `HelloWeb.HelloController` module with an `index/2` function.

To make the `index` action happen, let's create a new `lib/hello_web/controllers/hello_controller.ex` file, and make it look like the following:

```
defmodule HelloWeb.HelloController do
  use HelloWeb, :controller

  def index(conn, _params) do
    render(conn, :index)
  end
end
```

We'll save a discussion of `use HelloWeb, :controller` for the Controllers guide. For now, let's focus on the `index` action.

All controller actions take two arguments. The first is `conn`, a struct which holds a ton of data about the request. The second is `params`, which are the request parameters. Here, we are not using `params`, and we avoid compiler warnings by prefixing it with `_`.

The core of this action is `render(conn, :index)`. It tells Phoenix to render the `index` template. The modules responsible for rendering are called views. By default, Phoenix views are named after the controller (`HelloController`) and format (`HTML` in this case), so Phoenix is expecting a `HelloWeb.HelloHTML` to exist and define an `index/1` function.

A new view

Phoenix views act as the presentation layer. For example, we expect the output of rendering `index` to be a complete HTML page. To make our lives easier, we often use templates for creating those HTML pages.

Let's create a new view. Create `lib/hello_web/controllers/hello_html.ex` and make it look like this:

```
defmodule HelloWeb.HelloHTML do
  use HelloWeb, :html
```

```
  end
```

To add templates to this view, we can define them as function components in the module or in separate files.

Let's start by defining a function component:

```
defmodule HelloWeb.HelloHTML do
  use HelloWeb, :html

  def index(assigns) do
    ~H"""
    Hello!
    """
  end
end
```

We defined a function that receives `assigns` as arguments and use the `~H` sigil to put the contents we want to render. Inside the `~H` sigil, we use a templating language called HEEx, which stands for "HTML+EEx". `EEx` is a library for embedding Elixir that ships as part of Elixir itself. "HTML+EEx" is a Phoenix extension of EEx that is HTML aware, with support for HTML validation, components, and automatic escaping of values. The latter protects you from security vulnerabilities like Cross-Site-Scripting with no extra work on your part.

A template file works in the same way. Function components are great for smaller templates and separate files are a good choice when you have a lot of markup or your functions start to feel unmanageable.

Let's give it a try by defining a template in its own file. First delete our `def index(assigns)` function from above and replace it with an `embed_templates` declaration:

```
defmodule HelloWeb.HelloHTML do
  use HelloWeb, :html
```

```
  embed_templates "hello_html/*"
end
```

Here we are telling `Phoenix.Component` to embed all `.heex` templates found in the sibling `hello_html` directory into our module as function definitions.

Next, we need to add files to the `lib/hello_web/controllers/hello_html` directory.

Note the controller name (`HelloController`), the view name (`HelloHTML`), and the template directory (`hello_html`) all follow the same naming convention and are named after each other. They are also collocated together in the directory tree:

> **Note**: We can rename the `hello_html` directory to whatever we want and put it in a subdirectory of `lib/hello_web/controllers`, as long as we update the `embed_templates` setting accordingly. However, it's best to keep the same naming convention to prevent any confusion.

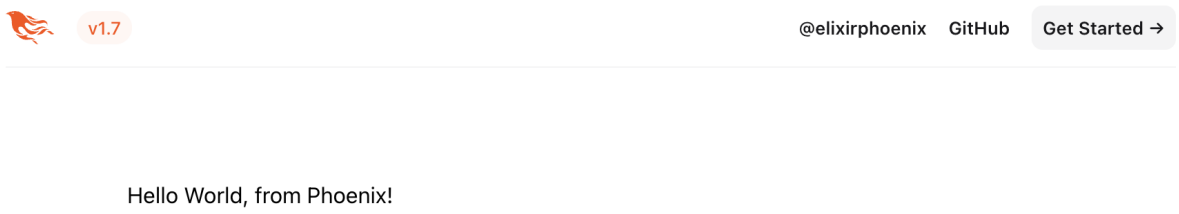
```
lib/hello_web
├── controllers
│   ├── hello_controller.ex
│   ├── hello_html.ex
│   └── hello_html
│       └── index.html.heex
```

A template file has the following structure: `NAME.FORMAT.TEMPLATING_LANGUAGE`. In our case, let's create an `index.html.heex` file at `lib/hello_web/controllers/hello_html/index.html.heex`:

```
<section>
  <h2>Hello World, from Phoenix!</h2>
</section>
```

Template files are compiled into the module as function components themselves, there is no runtime or performance difference between the two styles.

Now that we've got the route, controller, view, and template, we should be able to point our browsers at http://localhost:4000/hello and see our greeting from Phoenix! (In case you stopped the server along the way, the task to restart it is `mix phx.server`.)



There are a couple of interesting things to notice about what we just did. We didn't need to stop and restart the server while we made these changes. Yes, Phoenix has hot code reloading! Also, even though our `index.html.heex` file consists of only a single `section` tag, the page we get is a full HTML document. Our index template is actually rendered into layouts: first it renders `lib/hello_web/components/layouts/root.html.heex` which renders `lib/hello_web/components/layouts/app.html.heex` which finally includes our contents. If you open those files, you'll see a line that looks like this at the bottom:

```
{@inner_content}
```

Which injects our template into the layout before the HTML is sent off to the browser. We will talk more about layouts in the Controllers guide.

> A note on hot code reloading: Some editors with their automatic linters may prevent hot code reloading from working. If it's not working for you,

please see the discussion in this issue.

From endpoint to views

As we built our first page, we could start to understand how the request life-cycle is put together. Now let's take a more holistic look at it.

All HTTP requests start in our application endpoint. You can find it as a module named `HelloWeb.Endpoint` in `lib/hello_web/endpoint.ex`. Once you open up the endpoint file, you will see that, similar to the router, the endpoint has many calls to `plug`. `Plug` is a library and a specification for stitching web applications together. It is an essential part of how Phoenix handles requests and we will discuss it in detail in the Plug guide coming next.

For now, it suffices to say that each plug defines a slice of request processing. In the endpoint you will find a skeleton roughly like this:

```
defmodule HelloWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :hello

  plug Plug.Static, ...
  plug Plug.RequestId
  plug Plug.Telemetry, ...
  plug Plug.Parsers, ...
  plug Plug.MethodOverride
  plug Plug.Head
  plug Plug.Session, ...
  plug HelloWeb.Router
end
```

Each of these plugs have a specific responsibility that we will learn later. The last plug is precisely the `HelloWeb.Router` module. This allows the endpoint to delegate all further request processing to the router. As we now know, its main responsibility is to map verb/path pairs to controllers. The controller then tells a view to render a template.

At this moment, you may be thinking this can be a lot of steps to simply render a page. However, as our application grows in complexity, we will see that each layer serves a distinct purpose:

- endpoint (`Phoenix.Endpoint`) - the endpoint contains the common and initial path that all requests go through. If you want something to happen on all requests, it goes to the endpoint.

- router (`Phoenix.Router`) - the router is responsible for dispatching verb/path to controllers. The router also allows us to scope functionality. For example, some pages in your application may require user authentication, others may not.

- controller (`Phoenix.Controller`) - the job of the controller is to retrieve request information, talk to your business domain, and prepare data for the presentation layer.

- view - the view handles the structured data from the controller and converts it to a presentation to be shown to users. Views are often named after the content format they are rendering.

Let's do a quick recap on how the last three components work together by adding another page.

Another new page

Let's add just a little complexity to our application. We're going to add a new page that will recognize a piece of the URL, label it as a "messenger" and pass it through the controller into the template so our messenger can say hello.

As we did last time, the first thing we'll do is create a new route.

Another new route

For this exercise, we're going to reuse `HelloController` created at the previous step and add a new `show` action. We'll add a line just below our last route, like this:

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

    get "/", PageController, :home
    get "/hello", HelloController, :index
    get "/hello/:messenger", HelloController, :show
  end
```

Notice that we use the `:messenger` syntax in the path. Phoenix will take whatever value that appears in that position in the URL and convert it into a parameter. For example, if we point the browser at: `http://localhost:4000/hello/Frank`, the value of `"messenger"` will be `"Frank"`.

Another new action

Requests to our new route will be handled by the `HelloWeb.HelloController show` action. We already have the controller at `lib/hello_web/controllers/hello_controller.ex`, so all we need to do is edit that controller and add a `show` action to it. This time, we'll need to extract the messenger from the parameters so that we can pass it (the messenger) to the template. To do that, we add this show function to the controller:

```
  def show(conn, %{"messenger" => messenger}) do
    render(conn, :show, messenger: messenger)
  end
```

Within the body of the `show` action, we also pass a third argument to the render function, a key-value pair where `:messenger` is the key, and the `messenger` variable is passed as the value.

If the body of the action needs access to the full map of parameters bound to the `params` variable, in addition to the bound messenger variable, we could define `show/2` like this:

```
def show(conn, %{"messenger" => messenger} = params) do
  ...
end
```

It's good to remember that the keys of the `params` map will always be strings, and that the equals sign does not represent assignment, but is instead a pattern match assertion.

Another new template

For the last piece of this puzzle, we'll need a new template. Since it is for the `show` action of `HelloController`, it will go into the `lib/hello_web/controllers/hello_html` directory and be called `show.html.heex`. It will look surprisingly like our `index.html.heex` template, except that we will need to display the name of our messenger.

To do that, we'll use the special HEEx tags for executing Elixir expressions: `{...}` and `<%= %>`. Notice that EEx tag has an equals sign like this: `<%=` . That means that any Elixir code that goes between those tags will be executed, and the resulting value will replace the tag in the HTML output. If the equals sign were missing, the code would still be executed, but the value would not appear on the page.

Remember our templates are written in HEEx (HTML+EEx). HEEx is a superset of EEx, and thereby supports the EEx `<%= %>` interpolation syntax for interpolating arbitrary blocks of code. In general, the HEEx `{...}` interpolation syntax is preferred anytime there is HTML-aware intepolation to be done – such as within attributes or inline values with a body.

The only times `EEx` `<%= %>` interpolation is necessary is for interpolationg arbitrary blocks of markup, such as branching logic that inects separate markup trees, or for interpolating values within `<script>` or `<style>` tags.

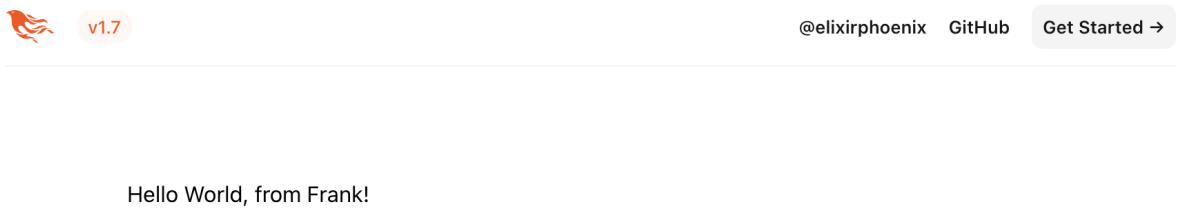
This is what the `hello_html/show.html.heex` template should look like:

```
<section>
  <h2>Hello World, from {@messenger}!</h2>
</section>
```

Our messenger appears as `@messenger`.

The values we passed to the view from the controller are collectively called our "assigns". We could access our messenger value via `assigns.messenger` but through some metaprogramming, Phoenix gives us the much cleaner `@` syntax for use in templates.

We're done. If you point your browser to http://localhost:4000/hello/Frank, you should see a page that looks like this:



Play around a bit. Whatever you put after `/hello/` will appear on the page as your messenger.

Plug

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Request life-cycle guide.

Plug lives at the heart of Phoenix's HTTP layer, and Phoenix puts Plug front and center. We interact with plugs at every step of the request life-cycle, and the core Phoenix components like endpoints, routers, and controllers are all just plugs internally. Let's jump in and find out just what makes Plug so special.

Plug is a specification for composable modules in between web applications. It is also an abstraction layer for connection adapters of different web servers. The basic idea of Plug is to unify the concept of a "connection" that we operate on. This differs from other HTTP middleware layers such as Rack, where the request and response are separated in the middleware stack.

At the simplest level, the Plug specification comes in two flavors: *function plugs* and *module plugs*.

Function plugs

In order to act as a plug, a function needs to:

1. accept a connection struct (`%Plug.Conn{}`) as its first argument, and connection options as its second one;
2. return a connection struct.

Any function that meets these two criteria will do. Here's an example.

```
def introspect(conn, _opts) do
  IO.puts """
  Verb: #{inspect(conn.method)}
  Host: #{inspect(conn.host)}
  Headers: #{inspect(conn.req_headers)}
  """

  conn
end
```

This function does the following:

1. It receives a connection and options (that we do not use)
2. It prints some connection information to the terminal
3. It returns the connection

Pretty simple, right? Let's see this function in action by adding it to our endpoint in `lib/hello_web/endpoint.ex`. We can plug it anywhere, so let's do it by inserting `plug :introspect` right before we delegate the request to the router:

```
defmodule HelloWeb.Endpoint do
  ...

  plug :introspect
  plug HelloWeb.Router

  def introspect(conn, _opts) do
    IO.puts """
    Verb: #{inspect(conn.method)}
    Host: #{inspect(conn.host)}
    Headers: #{inspect(conn.req_headers)}
    """

    conn
  end
end
```

Function plugs are plugged by passing the function name as an atom. To try the plug out, go back to your browser and fetch http://localhost:4000. You should see something like this printed in your shell terminal:

```
Verb: "GET"
Host: "localhost"
Headers: [...]
```

Our plug simply prints information from the connection. Although our initial plug is very simple, you can do virtually anything you want inside of it. To learn about all fields available in the connection and all of the functionality associated to it, see the documentation for `Plug.Conn`.

Now let's look at the other plug variant, the module plugs.

Module plugs

Module plugs are another type of plug that let us define a connection transformation in a module. The module only needs to implement two functions:

- `init/1` which initializes any arguments or options to be passed to `call/2`
- `call/2` which carries out the connection transformation. `call/2` is just a function plug that we saw earlier

To see this in action, let's write a module plug that puts the `:locale` key and value into the connection for downstream use in other plugs, controller actions, and our views. Put the contents below in a file named `lib/hello_web/plugs/locale.ex`:

```
defmodule HelloWeb.Plugs.Locale do
  import Plug.Conn

  @locales ["en", "fr", "de"]
```

```
  def init(default), do: default

  def call(%Plug.Conn{params: %{"locale" => loc}} = conn,
_default) when loc in @locales do
    assign(conn, :locale, loc)
  end

  def call(conn, default) do
    assign(conn, :locale, default)
  end
end
```

To give it a try, let's add this module plug to our router, by appending `plug HelloWeb.Plugs.Locale, "en"` to our `:browser` pipeline in `lib/hello_web/router.ex`:

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
    plug HelloWeb.Plugs.Locale, "en"
  end
  ...
```

In the `init/1` callback, we pass a default locale to use if none is present in the params. We also use pattern matching to define multiple `call/2` function heads to validate the locale in the params, and fall back to `"en"` if there is no match. The `assign/3` is a part of the `Plug.Conn` module and it's how we store values in the `conn` data structure.

To see the assign in action, go to the template in `lib/hello_web/controllers/page_html/home.html.heex` and add the following code after the closing of the `</h1>` tag:

```
<p>Locale: {@locale}</p>
```

Go to http://localhost:4000/ and you should see the locale exhibited. Visit http://localhost:4000/?locale=fr and you should see the assign changed to `"fr"`. Someone can use this information alongside Gettext to provide a fully internationalized web application.

That's all there is to Plug. Phoenix embraces the plug design of composable transformations all the way up and down the stack. Let's see some examples!

Where to plug

The endpoint, router, and controllers in Phoenix accept plugs.

Endpoint plugs

Endpoints organize all the plugs common to every request, and apply them before dispatching into the router with its custom pipelines. We added a plug to the endpoint like this:

```
defmodule HelloWeb.Endpoint do
  ...

  plug :introspect
  plug HelloWeb.Router
```

The default endpoint plugs do quite a lot of work. Here they are in order:

- `Plug.Static` - serves static assets. Since this plug comes before the logger, requests for static assets are not logged.

- `Phoenix.LiveDashboard.RequestLogger` - sets up the *Request Logger* for Phoenix LiveDashboard, this will allow you to have the option to either pass a query parameter to stream requests logs or to enable/disable a cookie that streams requests logs from your dashboard.

- `Plug.RequestId` - generates a unique request ID for each request.
- `Plug.Telemetry` - adds instrumentation points so Phoenix can log the request path, status code and request time by default.
- `Plug.Parsers` - parses the request body when a known parser is available. By default, this plug can handle URL-encoded, multipart and JSON content (with `Jason`). The request body is left untouched if the request content-type cannot be parsed.
- `Plug.MethodOverride` - converts the request method to PUT, PATCH or DELETE for POST requests with a valid `_method` parameter.
- `Plug.Head` - converts HEAD requests to GET requests.
- `Plug.Session` - a plug that sets up session management. Note that `fetch_session/2` must still be explicitly called before using the session, as this plug just sets up how the session is fetched.

In the middle of the endpoint, there is also a conditional block:

```
  if code_reloading? do
    socket "/phoenix/live_reload/socket",
Phoenix.LiveReloader.Socket
    plug Phoenix.LiveReloader
    plug Phoenix.CodeReloader
    plug Phoenix.Ecto.CheckRepoStatus, otp_app: :hello
  end
```

This block is only executed in development. It enables:

- live reloading - if you change a CSS file, they are updated in-browser without refreshing the page;
- code reloading - so we can see changes to our application without restarting the server;
- check repo status - which makes sure our database is up to date, raising a readable and actionable error otherwise.

Router plugs

In the router, we can declare plugs inside pipelines:

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.LayoutView,
:root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
    plug HelloWeb.Plugs.Locale, "en"
  end

  scope "/", HelloWeb do
    pipe_through :browser

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

Routes are defined inside scopes and scopes may pipe through multiple pipelines. Once a route matches, Phoenix invokes all plugs defined in all pipelines associated to that route. For example, accessing "/" will pipe through the `:browser` pipeline, consequently invoking all of its plugs.

As we will see in the routing guide, the pipelines themselves are plugs. There, we will also discuss all plugs in the `:browser` pipeline.

Controller plugs

Finally, controllers are plugs too, so we can do:

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller
```

```
  plug HelloWeb.Plugs.Locale, "en"
```

In particular, controller plugs provide a feature that allows us to execute plugs only within certain actions. For example, you can do:

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller

  plug HelloWeb.Plugs.Locale, "en" when action in
[:index]
```

And the plug will only be executed for the `index` action.

Plugs as composition

By abiding by the plug contract, we turn an application request into a series of explicit transformations. It doesn't stop there. To really see how effective Plug's design is, let's imagine a scenario where we need to check a series of conditions and then either redirect or halt if a condition fails. Without plug, we would end up with something like this:

```
defmodule HelloWeb.MessageController do
  use HelloWeb, :controller

  def show(conn, params) do
    case Authenticator.find_user(conn) do
      {:ok, user} ->
        case find_message(params["id"]) do
          nil ->
            conn |> put_flash(:info, "That message wasn't
found") |> redirect(to: ~p"/")
          message ->
            if Authorizer.can_access?(user, message) do
              render(conn, :show, page: message)
            else
              conn |> put_flash(:info, "You can't access
that page") |> redirect(to: ~p"/")
```

```
            end
        end
      :error ->
        conn |> put_flash(:info, "You must be logged in")
|> redirect(to: ~p"/")
    end
  end
end
```

Notice how just a few steps of authentication and authorization require complicated nesting and duplication? Let's improve this with a couple of plugs.

```
defmodule HelloWeb.MessageController do
  use HelloWeb, :controller

  plug :authenticate
  plug :fetch_message
  plug :authorize_message

  def show(conn, params) do
    render(conn, :show, page: conn.assigns[:message])
  end

  defp authenticate(conn, _) do
    case Authenticator.find_user(conn) do
      {:ok, user} ->
        assign(conn, :user, user)
      :error ->
        conn |> put_flash(:info, "You must be logged in")
|> redirect(to: ~p"/") |> halt()
    end
  end

  defp fetch_message(conn, _) do
    case find_message(conn.params["id"]) do
      nil ->
        conn |> put_flash(:info, "That message wasn't
found") |> redirect(to: ~p"/") |> halt()
      message ->
        assign(conn, :message, message)
    end
```

```
  end

  defp authorize_message(conn, _) do
    if Authorizer.can_access?(conn.assigns[:user],
conn.assigns[:message]) do
      conn
    else
      conn |> put_flash(:info, "You can't access that
page") |> redirect(to: ~p"/") |> halt()
    end
  end
end
```

To make this all work, we converted the nested blocks of code and used `halt(conn)` whenever we reached a failure path. The `halt(conn)` functionality is essential here: it tells Plug that the next plug should not be invoked.

At the end of the day, by replacing the nested blocks of code with a flattened series of plug transformations, we are able to achieve the same functionality in a much more composable, clear, and reusable way.

To learn more about plugs, see the documentation for the Plug project, which provides many built-in plugs and functionalities.

Routing

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Request life-cycle guide.

Routers are the main hubs of Phoenix applications. They match HTTP requests to controller actions, wire up real-time channel handlers, and define a series of pipeline transformations scoped to a set of routes.

The router file that Phoenix generates, `lib/hello_web/router.ex`, will look something like this one:

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.Layouts,
:root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :home
  end
```

```
  # Other scopes may use custom stacks.
  # scope "/api", HelloWeb do
  #   pipe_through :api
  # end
  # ...
end
```

Both the router and controller module names will be prefixed with the name you gave your application suffixed with `Web`.

The first line of this module, `use HelloWeb, :router`, simply makes Phoenix router functions available in our particular router.

Scopes have their own section in this guide, so we won't spend time on the `scope "/", HelloWeb do` block here. The `pipe_through :browser` line will get a full treatment in the "Pipelines" section of this guide. For now, you only need to know that pipelines allow a set of plugs to be applied to different sets of routes.

Inside the scope block, however, we have our first actual route:

```
get "/", PageController, :home
```

`get` is a Phoenix macro that corresponds to the HTTP verb GET. Similar macros exist for other HTTP verbs, including POST, PUT, PATCH, DELETE, OPTIONS, CONNECT, TRACE, and HEAD.

> **Why the macros?**
>
> Phoenix does its best to keep the usage of macros low. You may have noticed, however, that the `Phoenix.Router` relies heavily on macros. Why is that?
>
> We use `get`, `post`, `put`, and `delete` to define your routes. We use macros for two purposes:
>
> - They define the routing engine, used on every request, to choose which controller to dispatch the request to. Thanks to macros, Phoenix compiles all of your routes to a huge case-statement with pattern matching rules, which is heavily optimized by the Erlang VM
> - For each route you define, we also define metadata to implement `Phoenix.VerifiedRoutes`. As we will soon learn, verified routes allow us to reference any route as if it were a plain looking string, except that it is verified by the compiler to be valid (making it much harder to ship broken links, forms, mails, etc to production)
>
> In other words, the router relies on macros to build applications that are faster and safer. Also remember that macros in Elixir are compile-time only, which gives plenty of stability after the code is compiled. As we will learn next, Phoenix also provides introspection for all defined routes via `mix phx.routes`.

Examining routes

Phoenix provides an excellent tool for investigating routes in an application: `mix phx.routes`.

Let's see how this works. Go to the root of a newly-generated Phoenix application and run `mix phx.routes`. You should see something like the following, generated with all routes you currently have:

```
$ mix phx.routes
GET  /  HelloWeb.PageController :home
...
```

The route above tells us that any HTTP GET request for the root of the application will be handled by the `home` action of the `HelloWeb.PageController`.

Resources

The router supports other macros besides those for HTTP verbs like `get`, `post`, and `put`. The most important among them is `resources`. Let's add a resource to our `lib/hello_web/router.ex` file like this:

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

  get "/", PageController, :home
  resources "/users", UserController
  ...
end
```

For now it doesn't matter that we don't actually have a `HelloWeb.UserController`.

Run `mix phx.routes` once again at the root of your project. You should see something like the following:

```
...
GET     /users            HelloWeb.UserController :index
GET     /users/:id/edit   HelloWeb.UserController :edit
GET     /users/new        HelloWeb.UserController :new
GET     /users/:id        HelloWeb.UserController :show
POST    /users            HelloWeb.UserController :create
PATCH   /users/:id        HelloWeb.UserController :update
PUT     /users/:id        HelloWeb.UserController :update
DELETE  /users/:id        HelloWeb.UserController :delete
...
```

This is the standard matrix of HTTP verbs, paths, and controller actions. For a while, this was known as RESTful routes, but most consider this a misnomer nowadays. Let's look at them individually.

- A GET request to `/users` will invoke the `index` action to show all the users.

- A GET request to `/users/:id/edit` will invoke the `edit` action with an ID to retrieve an individual user from the data store and present the information in a form for editing.
- A GET request to `/users/new` will invoke the `new` action to present a form for creating a new user.
- A GET request to `/users/:id` will invoke the `show` action with an id to show an individual user identified by that ID.
- A POST request to `/users` will invoke the `create` action to save a new user to the data store.
- A PATCH request to `/users/:id` will invoke the `update` action with an ID to save the updated user to the data store.
- A PUT request to `/users/:id` will also invoke the `update` action with an ID to save the updated user to the data store.
- A DELETE request to `/users/:id` will invoke the `delete` action with an ID to remove the individual user from the data store.

If we don't need all these routes, we can be selective using the `:only` and `:except` options to filter specific actions.

Let's say we have a read-only posts resource. We could define it like this:

```
resources "/posts", PostController, only: [:index, :show]
```

Running `mix phx.routes` shows that we now only have the routes to the index and show actions defined.

```
GET     /posts      HelloWeb.PostController :index
GET     /posts/:id  HelloWeb.PostController :show
```

Similarly, if we have a comments resource, and we don't want to provide a route to delete one, we could define a route like this.

```
resources "/comments", CommentController, except:
[:delete]
```

Running `mix phx.routes` now shows that we have all the routes except the DELETE request to the delete action.

```
GET    /comments           HelloWeb.CommentController
:index
GET    /comments/:id/edit  HelloWeb.CommentController
:edit
GET    /comments/new       HelloWeb.CommentController
:new
GET    /comments/:id       HelloWeb.CommentController
:show
POST   /comments           HelloWeb.CommentController
:create
PATCH  /comments/:id       HelloWeb.CommentController
:update
PUT    /comments/:id       HelloWeb.CommentController
:update
```

The `Phoenix.Router.resources/4` macro describes additional options for customizing resource routes.

Verified Routes

Phoenix includes `Phoenix.VerifiedRoutes` module which provides compile-time checks of router paths against your router by using the `~p` sigil. For example, you can write paths in controllers, tests, and templates and the compiler will make sure those actually match routes defined in your router.

Let's see it in action. Run `iex -S mix` at the root of the project. We'll define a throwaway example module that builds a couple `~p` route paths.

```
iex> defmodule RouteExample do
...>   use HelloWeb, :verified_routes
...>
...>   def example do
...>     ~p"/comments"
...>     ~p"/unknown/123"
...>   end
```

```
...> end
warning: no route path for HelloWeb.Router matches
"/unknown/123"
  iex:5: RouteExample.example/0

{:module, RouteExample, ...}
iex>
```

Notice how the first call to an existing route, `~p"/comments"` gave no warning, but a bad route path `~p"/unknown/123"` produced a compiler warning, just as it should. This is significant because it allows us to write otherwise hard-coded paths in our application and the compiler will let us know whenever we write a bad route or change our routing structure.

Phoenix projects are set up out of the box to allow use of verified routes throughout your web layer, including tests. For example in your templates you can render `~p` links:

```
<.link href={~p"/"}>Welcome Page!</.link>
<.link href={~p"/comments"}>View Comments</.link>
```

Or in a controller, issue a redirect:

```
redirect(conn, to: ~p"/comments/#{comment}")
```

Using `~p` for route paths ensures our application paths and URLs stay up to date with the router definitions. The compiler will catch bugs for us, and let us know when we change routes that are referenced elsewhere in our application.

More on verified routes

What about paths with query strings? You can either add query string key values directly, or provide a dictionary of key-value pairs, for example:

```
~p"/users/17?admin=true&active=false"
"/users/17?admin=true&active=false"

~p"/users/17?#{[admin: true]}"
"/users/17?admin=true"
```

What if we need a full URL instead of a path? Just wrap your path with a call to `Phoenix.VerifiedRoutes.url/1`, which is imported everywhere that `~p` is available:

```
url(~p"/users")
"http://localhost:4000/users"
```

The `url` calls will get the host, port, proxy port, and SSL information needed to construct the full URL from the configuration parameters set for each environment. We'll talk about configuration in more detail in its own guide. For now, you can take a look at `config/dev.exs` file in your own project to see those values.

Nested resources

It is also possible to nest resources in a Phoenix router. Let's say we also have a `posts` resource that has a many-to-one relationship with `users`. That is to say, a user can create many posts, and an individual post belongs to only one user. We can represent that by adding a nested route in `lib/hello_web/router.ex` like this:

```
resources "/users", UserController do
  resources "/posts", PostController
end
```

When we run `mix phx.routes` now, in addition to the routes we saw for `users` above, we get the following set of routes:

```
...
GET     /users/:user_id/posts
HelloWeb.PostController :index
GET     /users/:user_id/posts/:id/edit
HelloWeb.PostController :edit
GET     /users/:user_id/posts/new
HelloWeb.PostController :new
GET     /users/:user_id/posts/:id
HelloWeb.PostController :show
POST    /users/:user_id/posts
HelloWeb.PostController :create
PATCH   /users/:user_id/posts/:id
HelloWeb.PostController :update
PUT     /users/:user_id/posts/:id
HelloWeb.PostController :update
DELETE  /users/:user_id/posts/:id
HelloWeb.PostController :delete
...
```

We see that each of these routes scopes the posts to a user ID. For the first one, we will invoke `PostController`'s `index` action, but we will pass in a `user_id`. This implies that we would display all the posts for that individual user only. The same scoping applies for all these routes.

When building paths for nested routes, we will need to interpolate the IDs where they belong in route definition. For the following `show` route, `42` is the `user_id`, and `17` is the `post_id`.

```
user_id = 42
post_id = 17
~p"/users/#{user_id}/posts/#{post_id}"
"/users/42/posts/17"
```

Verified routes also support the `Phoenix.Param` protocol, but we don't need to concern ourselves with Elixir protocols just yet. Just know that once we start building our application with structs like `%User{}` and `%Post{}`, we'll be able to interpolate those data structures directly into our `~p` paths and Phoenix will pluck out the correct fields to use in the route.

```
~p"/users/#{user}/posts/#{post}"
"/users/42/posts/17"
```

Notice how we didn't need to interpolate `user.id` or `post.id`? This is particularly nice if we decide later we want to make our URLs a little nicer and start using slugs instead. We don't need to change any of our `~p`'s!

Scoped routes

Scopes are a way to group routes under a common path prefix and scoped set of plugs. We might want to do this for admin functionality, APIs, and especially for versioned APIs. Let's say we have user-generated reviews on a site, and that those reviews first need to be approved by an administrator. The semantics of these resources are quite different, and they might not share the same controller. Scopes enable us to segregate these routes.

The paths to the user-facing reviews would look like a standard resource.

```
/reviews
/reviews/1234
/reviews/1234/edit
...
```

The administration review paths can be prefixed with `/admin`.

```
/admin/reviews
/admin/reviews/1234
/admin/reviews/1234/edit
...
```

We accomplish this with a scoped route that sets a path option to `/admin` like this one. We can nest this scope inside another scope, but instead, let's set it by itself at the root, by adding to `lib/hello_web/router.ex` the following:

```
scope "/admin", HelloWeb.Admin do
  pipe_through :browser

  resources "/reviews", ReviewController
end
```

We define a new scope where all routes are prefixed with `/admin` and all controllers are under the `HelloWeb.Admin` namespace.

Running `mix phx.routes` again, in addition to the previous set of routes we get the following:

```
...
GET     /admin/reviews
HelloWeb.Admin.ReviewController :index
GET     /admin/reviews/:id/edit
HelloWeb.Admin.ReviewController :edit
GET     /admin/reviews/new
HelloWeb.Admin.ReviewController :new
GET     /admin/reviews/:id
HelloWeb.Admin.ReviewController :show
POST    /admin/reviews
HelloWeb.Admin.ReviewController :create
PATCH   /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
PUT     /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
DELETE  /admin/reviews/:id
HelloWeb.Admin.ReviewController :delete
...
```

This looks good, but there is a problem here. Remember that we wanted both user-facing review routes `/reviews` and the admin ones `/admin/reviews`. If we now include the user-facing reviews in our router under the root scope like this:

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

```
  ...
  resources "/reviews", ReviewController
end

scope "/admin", HelloWeb.Admin do
  pipe_through :browser

  resources "/reviews", ReviewController
end
```

and we run `mix phx.routes`, we get output for each scoped route:

```
...
GET     /reviews
HelloWeb.ReviewController :index
GET     /reviews/:id/edit
HelloWeb.ReviewController :edit
GET     /reviews/new
HelloWeb.ReviewController :new
GET     /reviews/:id
HelloWeb.ReviewController :show
POST    /reviews
HelloWeb.ReviewController :create
PATCH   /reviews/:id
HelloWeb.ReviewController :update
PUT     /reviews/:id
HelloWeb.ReviewController :update
DELETE  /reviews/:id
HelloWeb.ReviewController :delete
...
GET     /admin/reviews
HelloWeb.Admin.ReviewController :index
GET     /admin/reviews/:id/edit
HelloWeb.Admin.ReviewController :edit
GET     /admin/reviews/new
HelloWeb.Admin.ReviewController :new
GET     /admin/reviews/:id
HelloWeb.Admin.ReviewController :show
POST    /admin/reviews
HelloWeb.Admin.ReviewController :create
PATCH   /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
```

```
PUT     /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
DELETE  /admin/reviews/:id
HelloWeb.Admin.ReviewController :delete
```

What if we had a number of resources that were all handled by admins? We could put all of them inside the same scope like this:

```
scope "/admin", HelloWeb.Admin do
  pipe_through :browser

  resources "/images",  ImageController
  resources "/reviews", ReviewController
  resources "/users",   UserController
end
```

Here's what `mix phx.routes` tells us:

```
...
GET     /admin/images
HelloWeb.Admin.ImageController :index
GET     /admin/images/:id/edit
HelloWeb.Admin.ImageController :edit
GET     /admin/images/new
HelloWeb.Admin.ImageController :new
GET     /admin/images/:id
HelloWeb.Admin.ImageController :show
POST    /admin/images
HelloWeb.Admin.ImageController :create
PATCH   /admin/images/:id
HelloWeb.Admin.ImageController :update
PUT     /admin/images/:id
HelloWeb.Admin.ImageController :update
DELETE  /admin/images/:id
HelloWeb.Admin.ImageController :delete
GET     /admin/reviews
HelloWeb.Admin.ReviewController :index
GET     /admin/reviews/:id/edit
HelloWeb.Admin.ReviewController :edit
GET     /admin/reviews/new
HelloWeb.Admin.ReviewController :new
```

```
GET     /admin/reviews/:id
HelloWeb.Admin.ReviewController :show
POST    /admin/reviews
HelloWeb.Admin.ReviewController :create
PATCH   /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
PUT     /admin/reviews/:id
HelloWeb.Admin.ReviewController :update
DELETE  /admin/reviews/:id
HelloWeb.Admin.ReviewController :delete
GET     /admin/users
HelloWeb.Admin.UserController :index
GET     /admin/users/:id/edit
HelloWeb.Admin.UserController :edit
GET     /admin/users/new
HelloWeb.Admin.UserController :new
GET     /admin/users/:id
HelloWeb.Admin.UserController :show
POST    /admin/users
HelloWeb.Admin.UserController :create
PATCH   /admin/users/:id
HelloWeb.Admin.UserController :update
PUT     /admin/users/:id
HelloWeb.Admin.UserController :update
DELETE  /admin/users/:id
HelloWeb.Admin.UserController :delete
```

This is great, exactly what we want. Note how every route and controller is properly namespaced.

Scopes can also be arbitrarily nested, but you should do it carefully as nesting can sometimes make our code confusing and less clear. With that said, suppose that we had a versioned API with resources defined for images, reviews, and users. Then technically, we could set up routes for the versioned API like this:

```
scope "/api", HelloWeb.Api, as: :api do
  pipe_through :api

  scope "/v1", V1, as: :v1 do
    resources "/images",  ImageController
```

```
    resources "/reviews", ReviewController
    resources "/users",   UserController
  end
end
```

You can run `mix phx.routes` to see how these definitions will look like.

Interestingly, we can use multiple scopes with the same path as long as we are careful not to duplicate routes. The following router is perfectly fine with two scopes defined for the same path:

```
defmodule HelloWeb.Router do
  use Phoenix.Router
  ...
  scope "/", HelloWeb do
    pipe_through :browser

    resources "/users", UserController
  end

  scope "/", AnotherAppWeb do
    pipe_through :browser

    resources "/posts", PostController
  end
  ...
end
```

If we do duplicate a route — which means two routes having the same path — we'll get this familiar warning:

```
warning: this clause cannot match because a previous
clause at line 16 always matches
```

Pipelines

We have come quite a long way in this guide without talking about one of the first lines we saw in the router: `pipe_through :browser`. It's time to fix that.

Pipelines are a series of plugs that can be attached to specific scopes. If you are not familiar with plugs, we have an in-depth guide about them.

Routes are defined inside scopes and scopes may pipe through multiple pipelines. Once a route matches, Phoenix invokes all plugs defined in all pipelines associated to that route. For example, accessing `/` will pipe through the `:browser` pipeline, consequently invoking all of its plugs.

Phoenix defines two pipelines by default, `:browser` and `:api`, which can be used for a number of common tasks. In turn we can customize them as well as create new pipelines to meet our needs.

The `:browser` and `:api` pipelines

As their names suggest, the `:browser` pipeline prepares for routes which render requests for a browser, and the `:api` pipeline prepares for routes which produce data for an API.

The `:browser` pipeline has six plugs: The `plug :accepts, ["html"]` defines the accepted request format or formats. `:fetch_session`, which, naturally, fetches the session data and makes it available in the connection. `:fetch_live_flash`, which fetches any flash messages from LiveView and merges them with the controller flash messages. Then, the plug `:put_root_layout` will store the root layout for rendering purposes. Later `:protect_from_forgery` and `:put_secure_browser_headers`, protects form posts from cross-site forgery.

Currently, the `:api` pipeline only defines `plug :accepts, ["json"]`.

The router invokes a pipeline on a route defined within a scope. Routes outside of a scope have no pipelines. Although the use of nested scopes is discouraged (see above the versioned API example), if we call

`pipe_through` within a nested scope, the router will invoke all `pipe_through`'s from parent scopes, followed by the nested one.

Those are a lot of words bunched up together. Let's take a look at some examples to untangle their meaning.

Here's another look at the router from a newly generated Phoenix application, this time with the `/api` scope uncommented back in and a route added.

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.Layouts,
:root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :home
  end

  # Other scopes may use custom stacks.
  scope "/api", HelloWeb do
    pipe_through :api

    resources "/reviews", ReviewController
  end
  # ...
end
```

When the server accepts a request, the request will always first pass through the plugs in our endpoint, after which it will attempt to match on the path and HTTP verb.

Let's say that the request matches our first route: a GET to `/`. The router will first pipe that request through the `:browser` pipeline - which will fetch the session data, fetch the flash, and execute forgery protection - before it dispatches the request to `PageController`'s `home` action.

Conversely, suppose the request matches any of the routes defined by the `resources/2` macro. In that case, the router will pipe it through the `:api` pipeline — which currently only performs content negotiation — before it dispatches further to the correct action of the `HelloWeb.ReviewController`.

If no route matches, no pipeline is invoked and a 404 error is raised.

Creating new pipelines

Phoenix allows us to create our own custom pipelines anywhere in the router. To do so, we call the `pipeline/2` macro with these arguments: an atom for the name of our new pipeline and a block with all the plugs we want in it.

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.Layouts,
:root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :auth do
    plug HelloWeb.Authentication
  end
```

```
  scope "/reviews", HelloWeb do
    pipe_through [:browser, :auth]

    resources "/", ReviewController
  end
end
```

The above assumes there is a plug called `HelloWeb.Authentication` that performs authentication and is now part of the `:auth` pipeline.

Note that pipelines themselves are plugs, so we can plug a pipeline inside another pipeline. For example, we could rewrite the `auth` pipeline above to automatically invoke `browser`, simplifying the downstream pipeline call:

```
pipeline :auth do
  plug :browser
  plug :ensure_authenticated_user
  plug :ensure_user_owns_review
end

scope "/reviews", HelloWeb do
  pipe_through :auth

  resources "/", ReviewController
end
```

How to organize my routes?

In Phoenix, we tend to define several pipelines, that provide specific functionality. For example, the `:browser` and `:api` pipelines are meant to be accessed by specific clients, browsers and http clients respectively.

Perhaps more importantly, it is also very common to define pipelines specific to authentication and authorization. For example, you might have a pipeline that requires all users are authenticated. Another pipeline may enforce only admin users can access certain routes.

Once your pipelines are defined, you reuse the pipelines in the desired scopes, grouping your routes around their pipelines. For example, going back to our reviews example. Let's say anyone can read a review, but only authenticated users can create them. Your routes could look like this:

```
pipeline :browser do
  ...
end

pipeline :auth do
  plug HelloWeb.Authentication
end

scope "/" do
  pipe_through [:browser]

  get "/reviews", PostController, :index
  get "/reviews/:id", PostController, :show
end

scope "/" do
  pipe_through [:browser, :auth]

  get "/reviews/new", PostController, :new
  post "/reviews", PostController, :create
end
```

Note in the above how the routes are split across different scopes. While the separation can be confusing at first, it has one big upside: it is very easy to inspect your routes and see all routes that, for example, require authentication and which ones do not. This helps with auditing and making sure your routes have the proper scope.

You can create as few or as many scopes as you want. Because pipelines are reusable across scopes, they help encapsulate common functionality and you can compose them as necessary on each scope you define.

Forward

The `Phoenix.Router.forward/4` macro can be used to send all requests that start with a particular path to a particular plug. Let's say we have a part of our system that is responsible (it could even be a separate application or library) for running jobs in the background, it could have its own web interface for checking the status of the jobs. We can forward to this admin interface using:

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  ...

  scope "/", HelloWeb do
    ...
  end

  forward "/jobs", BackgroundJob.Plug
end
```

This means that all routes starting with `/jobs` will be sent to the `HelloWeb.BackgroundJob.Plug` module. Inside the plug, you can match on subroutes, such as `/pending` and `/active` that shows the status of certain jobs.

We can even mix the `forward/4` macro with pipelines. If we wanted to ensure that the user was authenticated and was an administrator in order to see the jobs page, we could use the following in our router.

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  ...

  scope "/" do
    pipe_through [:authenticate_user, :ensure_admin]
    forward "/jobs", BackgroundJob.Plug
  end
end
```

This means the plugs in the `authenticate_user` and `ensure_admin` pipelines will be called before the `BackgroundJob.Plug` allowing them to send an appropriate response and halt the request accordingly.

The `opts` that are received in the `init/1` callback of the Module Plug can be passed as a third argument. For example, maybe the background job lets you set the name of your application to be displayed on the page. This could be passed with:

```
forward "/jobs", BackgroundJob.Plug, name: "Hello
Phoenix"
```

There is a fourth `router_opts` argument that can be passed. These options are outlined in the `Phoenix.Router.scope/2` documentation.

`BackgroundJob.Plug` can be implemented as any Module Plug discussed in the Plug guide. Note though it is not advised to forward to another Phoenix endpoint. This is because plugs defined by your app and the forwarded endpoint would be invoked twice, which may lead to errors.

Summary

Routing is a big topic, and we have covered a lot of ground here. The important points to take away from this guide are:

- Routes which begin with an HTTP verb name expand to a single clause of the match function.
- Routes declared with `resources` expand to 8 clauses of the match function.
- Resources may restrict the number of match function clauses by using the `only:` or `except:` options.
- Any of these routes may be nested.
- Any of these routes may be scoped to a given path.
- Using verified routes with `~p` for compile-time route checks

Controllers

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the request life-cycle guide.

Phoenix controllers act as intermediary modules. Their functions — called actions — are invoked from the router in response to HTTP requests. The actions, in turn, gather all the necessary data and perform all the necessary steps before invoking the view layer to render a template or returning a JSON response.

Phoenix controllers also build on the Plug package, and are themselves plugs. Controllers provide the functions to do almost anything we need to in an action. If we do find ourselves looking for something that Phoenix controllers don't provide, we might find what we're looking for in Plug itself. Please see the Plug guide or the Plug documentation for more information.

A newly generated Phoenix app will have a single controller named `PageController`, which can be found at `lib/hello_web/controllers/page_controller.ex` which looks like this:

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller

  def home(conn, _params) do
    render(conn, :home, layout: false)
  end
end
```

The first line below the module definition invokes the `__using__/1` macro of the `HelloWeb` module, which imports some useful modules.

`PageController` gives us the `home` action to display the Phoenix welcome page associated with the default route Phoenix defines in the router.

Actions

Controller actions are just functions. We can name them anything we like as long as they follow Elixir's naming rules. The only requirement we must fulfill is that the action name matches a route defined in the router.

For example, in `lib/hello_web/router.ex` we could change the action name in the default route that Phoenix gives us in a new app from `home`:

```
get "/", PageController, :home
```

to `index`:

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

as long as we change the action name in `PageController` to `index` as well, the welcome page will load as before.

```
defmodule HelloWeb.PageController do
  ...

  def index(conn, _params) do
    render(conn, :index)
  end
end
```

While we can name our actions whatever we like, there are conventions for action names which we should follow whenever possible. We went over

these in the routing guide, but we'll take another quick look here.

- index - renders a list of all items of the given resource type
- show - renders an individual item by ID
- new - renders a form for creating a new item
- create - receives parameters for one new item and saves it in a data store
- edit - retrieves an individual item by ID and displays it in a form for editing
- update - receives parameters for one edited item and saves the item to a data store
- delete - receives an ID for an item to be deleted and deletes it from a data store

Each of these actions takes two parameters, which will be provided by Phoenix behind the scenes.

The first parameter is always `conn`, a struct which holds information about the request such as the host, path elements, port, query string, and much more. `conn` comes to Phoenix via Elixir's Plug middleware framework. More detailed information about `conn` can be found in the Plug.Conn documentation.

The second parameter is `params`. Not surprisingly, this is a map which holds any parameters passed along in the HTTP request. It is a good practice to pattern match against parameters in the function signature to provide data in a simple package we can pass on to rendering. We saw this in the request life-cycle guide when we added a messenger parameter to our `show` route in `lib/hello_web/controllers/hello_controller.ex`.

```
defmodule HelloWeb.HelloController do
  ...

  def show(conn, %{"messenger" => messenger}) do
    render(conn, :show, messenger: messenger)
  end
end
```

In some cases — often in `index` actions, for instance — we don't care about parameters because our behavior doesn't depend on them. In those cases, we don't use the incoming parameters, and simply prefix the variable name with an underscore, calling it `_params`. This will keep the compiler from complaining about the unused variable while still keeping the correct arity.

Rendering

Controllers can render content in several ways. The simplest is to render some plain text using the `text/2` function which Phoenix provides.

For example, let's rewrite the `show` action from `HelloController` to return text instead. For that, we could do the following.

```
def show(conn, %{"messenger" => messenger}) do
  text(conn, "From messenger #{messenger}")
end
```

Now `/hello/Frank` in your browser should display `From messenger Frank` as plain text without any HTML.

A step beyond this is rendering pure JSON with the `json/2` function. We need to pass it something that the Jason library can decode into JSON, such as a map. (Jason is one of Phoenix's dependencies.)

```
def show(conn, %{"messenger" => messenger}) do
  json(conn, %{id: messenger})
end
```

If we again visit `/hello/Frank` in the browser, we should see a block of JSON with the key `id` mapped to the string `"Frank"`.

```
{"id": "Frank"}
```

The `json/2` function is useful for writing APIs and there is also the `html/2` function for rendering HTML, but most of the times we use Phoenix views to build our responses. For this, Phoenix includes the `render/3` function. It is specially important for HTML responses, as Phoenix Views provide performance and security benefits.

Let's rollback our `show` action to what we originally wrote in the request life-cycle guide:

```
defmodule HelloWeb.HelloController do
  use HelloWeb, :controller

  def show(conn, %{"messenger" => messenger}) do
    render(conn, :show, messenger: messenger)
  end
end
```

In order for the `render/3` function to work correctly, the controller and view must share the same root name (in this case `Hello`), and the `HelloHTML` module must include an `embed_templates` definition specifying where its templates live. By default the controller, view module, and templates are collocated together in the same controller directory. In other words, `HelloController` requires `HelloHTML`, and `HelloHTML` requires the existence of the `lib/hello_web/controllers/hello_html/` directory, which must contain the `show.html.heex` template.

`render/3` will also pass the value which the `show` action received for `messenger` from the parameters as an assign.

If we need to pass values into the template when using `render`, that's easy. We can pass a keyword like we've seen with `messenger: messenger`, or we can use `Plug.Conn.assign/3`, which conveniently returns `conn`.

```
  def show(conn, %{"messenger" => messenger}) do
    conn
    |> Plug.Conn.assign(:messenger, messenger)
```

```
    |> render(:show)
  end
```

Note: Using `Phoenix.Controller` imports `Plug.Conn`, so shortening the call to `assign/3` works just fine.

Passing more than one value to our template is as simple as connecting `assign/3` functions together:

```
  def show(conn, %{"messenger" => messenger}) do
    conn
    |> assign(:messenger, messenger)
    |> assign(:receiver, "Dweezil")
    |> render(:show)
  end
```

Or you can pass the assigns directly to `render` instead:

```
  def show(conn, %{"messenger" => messenger}) do
    render(conn, :show, messenger: messenger, receiver:
"Dweezil")
  end
```

Generally speaking, once all assigns are configured, we invoke the view layer. The view layer (`HelloWeb.HelloHTML`) then renders `show.html` alongside the layout and a response is sent back to the browser.

Components and HEEx templates have their own guide, so we won't spend much time on them here. What we will look at is how to render different formats from inside a controller action.

New rendering formats

Rendering HTML through a template is fine, but what if we need to change the rendering format on the fly? Let's say that sometimes we need HTML,

sometimes we need plain text, and sometimes we need JSON. Then what?

The view's job is not only to render HTML templates. Views are about data presentation. Given a bag of data, the view's purpose is to present that in a meaningful way given some format, be it HTML, JSON, CSV, or others. Many web apps today return JSON to remote clients, and Phoenix views are *great* for JSON rendering.

As an example, let's take `PageController`'s `home` action from a newly generated app. Out of the box, this has the right view `PageHTML`, the embedded templates from (`lib/hello_web/controllers/page_html`), and the right template for rendering HTML (`home.html.heex`.)

```
def home(conn, _params) do
  render(conn, :home, layout: false)
end
```

What it doesn't have is a view for rendering JSON. Phoenix Controller hands off to a view module to render templates, and it does so per format. We already have a view for the HTML format, but we need to instruct Phoenix how to render the JSON format as well. By default, you can see which formats your controllers support in `lib/hello_web.ex`:

```
def controller do
  quote do
    use Phoenix.Controller,
      formats: [:html, :json],
      layouts: [html: HelloWeb.Layouts]
    ...
  end
end
```

So out of the box Phoenix will look for a `HTML` and `JSON` view modules based on the request format and the controller name. We can also explicitly tell Phoenix in our controller which view(s) to use for each format. For

example, what Phoenix does by default can be explicitly set with the following in your controller:

```
plug :put_view, html: HelloWeb.PageHTML, json:
HelloWeb.PageJSON
```

Let's add a `PageJSON` view module at `lib/hello_web/controllers/page_json.ex`:

```
defmodule HelloWeb.PageJSON do
  def home(_assigns) do
    %{message: "this is some JSON"}
  end
end
```

Since the Phoenix View layer is simply a function that the controller renders, passing connection assigns, we can define a regular `home/1` function and return a map to be serialized as JSON.

There are just a few more things we need to do to make this work. Because we want to render both HTML and JSON from the same controller, we need to tell our router that it should accept the `json` format. We do that by adding `json` to the list of accepted formats in the `:browser` pipeline. Let's open up `lib/hello_web/router.ex` and change `plug :accepts` to include `json` as well as `html` like this.

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html", "json"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, html: {HelloWeb.LayoutView,
:root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
```

```
  end
...
```

Phoenix allows us to change formats on the fly with the `_format` query string parameter. If we go to `http://localhost:4000/?_format=json`, we will see `%{"message": "this is some JSON"}`.

In practice, however, applications that need to render both formats typically use two distinct pipelines for each, such as the `pipeline :api` already defined in your router file. To learn more, see our JSON and APIs guide.

Sending responses directly

If none of the rendering options above quite fits our needs, we can compose our own using some of the functions that `Plug` gives us. Let's say we want to send a response with a status of "201" and no body whatsoever. We can do that with the `Plug.Conn.send_resp/3` function.

Edit the `home` action of `PageController` in `lib/hello_web/controllers/page_controller.ex` to look like this:

```
def home(conn, _params) do
  send_resp(conn, 201, "")
end
```

Reloading http://localhost:4000 should show us a completely blank page. The network tab of our browser's developer tools should show a response status of "201" (Created). Some browsers (Safari) will download the response, as the content type is not set.

To be specific about the content type, we can use `put_resp_content_type/2` in conjunction with `send_resp/3`.

```
def home(conn, _params) do
  conn
  |> put_resp_content_type("text/plain")
```

```
    |> send_resp(201, "")
  end
```

Using `Plug` functions in this way, we can craft just the response we need.

Setting the content type

Analogous to the `_format` query string param, we can render any sort of format we want by modifying the HTTP Content-Type Header and providing the appropriate template.

If we wanted to render an XML version of our `home` action, we might implement the action like this in `lib/hello_web/page_controller.ex.`

```
def home(conn, _params) do
  conn
  |> put_resp_content_type("text/xml")
  |> render(:home, content: some_xml_content)
end
```

We would then need to provide an `home.xml.eex` template which created valid XML, and we would be done.

For a list of valid content mime-types, please see the `MIME` library.

Setting the HTTP Status

We can also set the HTTP status code of a response similarly to the way we set the content type. The `Plug.Conn` module, imported into all controllers, has a `put_status/2` function to do this.

`Plug.Conn.put_status/2` takes `conn` as the first parameter and as the second parameter either an integer or a "friendly name" used as an atom for the status code we want to set. The list of status code atom representations can be found in `Plug.Conn.Status.code/1` documentation.

Let's change the status in our `PageController home` action.

```
def home(conn, _params) do
  conn
  |> put_status(202)
  |> render(:home, layout: false)
end
```

The status code we provide must be a valid number.

Redirection

Often, we need to redirect to a new URL in the middle of a request. A successful `create` action, for instance, will usually redirect to the `show` action for the resource we just created. Alternately, it could redirect to the `index` action to show all the things of that same type. There are plenty of other cases where redirection is useful as well.

Whatever the circumstance, Phoenix controllers provide the handy `redirect/2` function to make redirection easy. Phoenix differentiates between redirecting to a path within the application and redirecting to a URL — either within our application or external to it.

In order to try out `redirect/2`, let's create a new route in `lib/hello_web/router.ex`.

```
defmodule HelloWeb.Router do
  ...

  scope "/", HelloWeb do
    ...
    get "/", PageController, :home
    get "/redirect_test", PageController, :redirect_test
    ...
  end
end
```

Then we'll change `PageController`'s `home` action of our controller to do nothing but to redirect to our new route.

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller

  def home(conn, _params) do
    redirect(conn, to: ~p"/redirect_test")
  end
end
```

We made use of `Phoenix.VerifiedRoutes.sigil_p/2` to build our redirect path, which is the preferred approach to reference any path within our application. We learned about verified routes in the routing guide.

Finally, let's define in the same file the action we redirect to, which simply renders the home, but now under a new address:

```
def redirect_test(conn, _params) do
  render(conn, :home, layout: false)
end
```

When we reload our welcome page, we see that we've been redirected to `/redirect_test` which shows the original welcome page. It works!

If we care to, we can open up our developer tools, click on the network tab, and visit our root route again. We see two main requests for this page - a get to `/` with a status of `302`, and a get to `/redirect_test` with a status of `200`.

Notice that the redirect function takes `conn` as well as a string representing a relative path within our application. For security reasons, the `:to` option can only redirect to paths within your application. If you want to redirect to a fully-qualified path or an external URL, you should use `:external` instead:

```
def home(conn, _params) do
  redirect(conn, external: "https://elixir-lang.org/")
end
```

Flash messages

Sometimes we need to communicate with users during the course of an action. Maybe there was an error updating a schema, or maybe we just want to welcome them back to the application. For this, we have flash messages.

The `Phoenix.Controller` module provides the `put_flash/3` to set flash messages as a key-value pair and placing them into a `@flash` assign in the connection. Let's set two flash messages in our `HelloWeb.PageController` to try this out.

To do this we modify the `home` action as follows:

```
defmodule HelloWeb.PageController do
  ...
  def home(conn, _params) do
    conn
    |> put_flash(:error, "Let's pretend we have an
error.")
    |> render(:home, layout: false)
  end
end
```

In order to see our flash messages, we need to be able to retrieve them and display them in a template layout. We can do that using `Phoenix.Flash.get/2` which takes the flash data and the key we care about. It then returns the value for that key.

For our convenience, a `flash_group` component is already available and added to the beginning of our welcome page

```
<.flash_group flash={@flash} />
```

When we reload the welcome page, our message should appear in the top right corner of the page.

The flash functionality is handy when mixed with redirects. Perhaps you want to redirect to a page with some extra information. If we reuse the redirect action from the previous section, we can do:

```
  def home(conn, _params) do
    conn
    |> put_flash(:error, "Let's pretend we have an
error.")
    |> redirect(to: ~p"/redirect_test")
  end
```

Now if you reload the welcome page, you will be redirected and the flash message will be shown once more.

Besides `put_flash/3`, the `Phoenix.Controller` module has another useful function worth knowing about. `clear_flash/1` takes only `conn` and removes any flash messages which might be stored in the session.

Phoenix does not enforce which keys are stored in the flash. As long as we are internally consistent, all will be well. `:info` and `:error`, however, are common and are handled by default in our templates.

Error pages

Phoenix has two views called `ErrorHTML` and `ErrorJSON` which live in `lib/hello_web/controllers/`. The purpose of these views is to handle errors in a general way for incoming HTML or JSON requests. Similar to the views we built in this guide, error views can return both HTML and JSON responses. See the Custom Error Pages How-To for more information.

Components and HEEx

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the request life-cycle guide.

The Phoenix endpoint pipeline takes a request, routes it to a controller, and calls a view module to render a template. The view interface from the controller is simple – the controller calls a view function with the connections assigns, and the function's job is to return a HEEx template. We call any function that accepts an `assigns` parameter and returns a HEEx template a *function component*. Function components are defined with the help of the `Phoenix.Component` module.

Function components are the essential building block for any kind of markup-based template rendering you'll perform in Phoenix. They serve as a shared abstraction for the standard MVC controller-based applications, LiveView applications, layouts, and smaller UI definitions you'll use throughout other templates.

In this chapter, we will recap how components were used in previous chapters and find new use cases for them.

Function components

At the end of the Request life-cycle chapter, we created a template at `lib/hello_web/controllers/hello_html/show.html.heex`, let's open it up:

```
<section>
  <h2>Hello World, from {@messenger}!</h2>
</section>
```

This template, is embedded as part of `HelloHTML`, at `lib/hello_web/controllers/hello_html.ex`:

```
defmodule HelloWeb.HelloHTML do
  use HelloWeb, :html

  embed_templates "hello_html/*"
end
```

That's simple enough. There's only two lines, `use HelloWeb, :html`. This line calls the `html/0` function defined in `HelloWeb` which sets up the basic imports and configuration for our function components and templates.

All of the imports and aliases we make in our module will also be available in our templates. That's because templates are effectively compiled into functions inside their respective module. For example, if you define a function in your module, you will be able to invoke it directly from the template. Let's see this in practice.

Imagine we want to refactor our `show.html.heex` to move the rendering of `<h2>Hello World, from {@messenger}!</h2>` to its own function. We can move it to a function component inside `HelloHTML`, let's do so:

```
defmodule HelloWeb.HelloHTML do
  use HelloWeb, :html

  embed_templates "hello_html/*"

  attr :messenger, :string, required: true

  def greet(assigns) do
    ~H"""
    <h2>Hello World, from {@messenger}!</h2>
    """
  end
end
```

We declared the attributes we accept via the `attr/3` macro provided by `Phoenix.Component`, then we defined our `greet/1` function which returns the HEEx template.

Next we need to update `show.html.heex`:

```
<section>
  <.greet messenger={@messenger} />
</section>
```

When we reload `http://localhost:4000/hello/Frank`, we should see the same content as before.

Since templates are embedded inside the `HelloHTML` module, we were able to invoke the view function simply as `<.greet messenger="..." />`.

If the component was defined elsewhere, we can also type `<HelloWeb.HelloHTML.greet messenger="..." />`.

By declaring attributes as required, Phoenix will warn at compile time if we call the `<.greet />` component without passing attributes. If an attribute is optional, you can specify the `:default` option with a value:

```
attr :messenger, :string, default: nil
```

Although this is a quick example, it shows the different roles function components play in Phoenix:

- Function components can be defined as functions that receive `assigns` as argument and call the `~H` sigil, as we did in `greet/1`
- Function components can be embedded from template files, that's how we load `show.html.heex` into `HelloWeb.HelloHTML`
- Function components can declare which attributes are expected, which are validated at compilation time

- Function components can be directly rendered from controllers
- Function components can be directly rendered from other function components, as we called `<.greet messenger={@messenger} />` from `show.html.heex`

And there's more. Before we go deeper, let's fully understand the expressive power behind the HEEx template language.

HEEx

Function components and templates files are powered by the HEEx template language, which stands for "HTML+EEx". EEx is an Elixir library that uses `<%= expression %>` to execute Elixir expressions and interpolate their results into arbitrary text templates. HEEx extends EEx for writing HTML templates mixed with Elixir interpolation. We can write Elixir code inside `{...}` for HTML-aware interpolation inside tag attributes and the body. We can also interpolate arbitrary HEEx blocks using EEx interpolation (`<%= ... %>`). We use `@name` to access the key `name` defined inside `assigns`.

This is frequently used to display assigns we have set by way of the `@` shortcut. In your controller, if you invoke:

```
render(conn, :show, username: "joe")
```

Then you can access said username in the templates as `{@username}`. In addition to displaying assigns and functions, we can use pretty much any Elixir expression. For example, in order to have conditionals:

```
<%= if some_condition? do %>
  <p>Some condition is true for user: {@username}</p>
<% else %>
  <p>Some condition is false for user: {@username}</p>
<% end %>
```

or even loops:

```
<table>
  <tr>
    <th>Number</th>
    <th>Power</th>
  </tr>
  <%= for number <- 1..10 do %>
    <tr>
      <td>{number}</td>
      <td>{number * number}</td>
    </tr>
  <% end %>
</table>
```

Did you notice the use of `<%= %>` versus `<% %>` above? All expressions that output something to the template **must** use the equals sign (=). If this is not included the code will still be executed but nothing will be inserted into the template.

HEEx also comes with handy HTML extensions we will learn next.

HTML extensions

Besides allowing interpolation of Elixir expressions via `<%= %>`, `.heex` templates come with HTML-aware extensions. For example, let's see what happens if you try to interpolate a value with "<" or ">" in it, which would lead to HTML injection:

```
{"<b>Bold?</b>"}
```

Once you render the template, you will see the literal `<b>` on the page. This means users cannot inject HTML content on the page. If you want to allow them to do so, you can call `raw`, but do so with extreme care:

```
{raw "<b>Bold?</b>"}
```

Another super power of HEEx templates is validation of HTML and interpolation syntax of attributes. You can write:

```
<div title="My div" class={@class}>
  <p>Hello {@username}</p>
</div>
```

Notice how you could simply use `key={value}`. HEEx will automatically handle special values such as `false` to remove the attribute or a list of classes.

To interpolate a dynamic number of attributes in a keyword list or map, do:

```
<div title="My div" {@many_attributes}>
  <p>Hello {@username}</p>
</div>
```

Also, try removing the closing `</div>` or renaming it to `</div-typo>`. HEEx templates will let you know about your error.

HEEx also supports shorthand syntax for `if` and `for` expressions via the special `:if` and `:for` attributes. For example, rather than this:

```
<%= if @some_condition do %>
  <div>...</div>
<% end %>
```

You can write:

```
<div :if={@some_condition}>...</div>
```

Likewise, for comprehensions may be written as:

```
<ul>
  <li :for={item <- @items}>{item.name}</li>
</ul>
```

Layouts

Layouts are just function components. They are defined in a module, just like all other function component templates. In a newly generated app, this is `lib/hello_web/components/layouts.ex`. You will also find a `layouts` folder with two built-in layouts generated by Phoenix. The default *root layout* is called `root.html.heex`, and it is the layout into which all templates will be rendered by default. The second is the *app layout*, called `app.html.heex`, which is rendered within the root layout and includes our contents.

You may be wondering how the string resulting from a rendered view ends up inside a layout. That's a great question! If we look at `lib/hello_web/components/layouts/root.html.heex`, just about at the end of the `<body>`, we will see this.

```
{@inner_content}
```

In other words, after rendering your page, the result is placed in the `@inner_content` assign.

Phoenix provides all kinds of conveniences to control which layout should be rendered. For example, the `Phoenix.Controller` module provides the `put_root_layout/2` function for us to switch *root layouts*. This takes `conn` as its first argument and a keyword list of formats and their layouts. You can set it to `false` to disable the layout altogether.

You can edit the `index` action of `HelloController` in `lib/hello_web/controllers/hello_controller.ex` to look like this.

```
def index(conn, _params) do
  conn
  |> put_root_layout(html: false)
  |> render(:index)
end
```

After reloading http://localhost:4000/hello, we should see a very different page, one with no title or CSS styling at all.

To customize the application layout, we invoke a similar function named `put_layout/2`. Let's actually create another layout and render the index template into it. As an example, let's say we had a different layout for the admin section of our application which didn't have the logo image. To do this, copy the existing `app.html.heex` to a new file `admin.html.heex` in the same directory `lib/hello_web/components/layouts`. Then remove everything inside the `<header>...</header>` tags (or change it to whatever you desire) in the new file.

Now, in the `index` action of the controller of `lib/hello_web/controllers/hello_controller.ex`, add the following:

```
def index(conn, _params) do
  conn
  |> put_layout(html: :admin)
  |> render(:index)
end
```

When we load the page, we should be rendering the admin layout without the header (or a custom one that you wrote).

At this point, you may be wondering, why does Phoenix have two layouts?

First of all, it gives us flexibility. In practice, we will hardly have multiple root layouts, as they often contain only HTML headers. This allows us to focus on different application layouts with only the parts that changes

between them. Second of all, Phoenix ships with a feature called LiveView, which allows us to build rich and real-time user experiences with server-rendered HTML. LiveView is capable of dynamically changing the contents of the page, but it only ever changes the app layout, never the root layout. Check out the LiveView documentation to learn more.

CoreComponents

In a new Phoenix application, you will also find a `core_components.ex` module inside the `components` folder. This module is a great example of defining function components to be reused throughout our application. This guarantees that, as our application evolves, our components will look consistent.

If you look inside `def html` in `HelloWeb` placed at `lib/hello_web.ex`, you will see that `CoreComponents` are automatically imported into all HTML views via `use HelloWeb, :html`. This is also the reason why `CoreComponents` itself performs `use Phoenix.Component` instead `use HelloWeb, :html` at the top: doing the latter would cause a deadlock as we would try to import `CoreComponents` into itself.

CoreComponents also play an important role in Phoenix code generators, as the code generators assume those components are available in order to quickly scaffold your application. In case you want to learn more about all of these pieces, you may:

- Explore the generated `CoreComponents` module to learn more from practical examples
- Read the official documentation for `Phoenix.Component`
- Read the official documentation for HEEx and the ~H sigils

Ecto

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.

Most web applications today need some form of data validation and persistence. In the Elixir ecosystem, we have `Ecto` to enable this. Before we jump into building database-backed web features, we're going to focus on the finer details of Ecto to give a solid base to build our web features on top of. Let's get started!

Phoenix uses Ecto to provide builtin support to the following databases:

- PostgreSQL (via `postgrex`)
- MySQL (via `myxql`)
- MSSQL (via `tds`)
- ETS (via `etso`)
- SQLite3 (via `ecto_sqlite3`)

Newly generated Phoenix projects include Ecto with the PostgreSQL adapter by default. You can pass the `--database` option to change or `--no-ecto` flag to exclude this.

Ecto also provides support for other databases and it has many learning resources available. Please check out Ecto's README for general information.

This guide assumes that we have generated our new application with Ecto integration and that we will be using PostgreSQL. The introductory guides cover how to get your first application up and running. For using other databases, see the Using other databases section.

Using the schema and migration generator

Once we have Ecto and PostgreSQL installed and configured, the easiest way to use Ecto is to generate an Ecto *schema* through the `phx.gen.schema` task. Ecto schemas are a way for us to specify how Elixir data types map to and from external sources, such as database tables. Let's generate a `User` schema with `name`, `email`, `bio`, and `number_of_pets` fields.

```
$ mix phx.gen.schema User users name:string email:string
\
bio:string number_of_pets:integer

* creating ./lib/hello/user.ex
* creating
priv/repo/migrations/20170523151118_create_users.exs

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

A couple of files were generated with this task. First, we have a `user.ex` file, containing our Ecto schema with our schema definition of the fields we passed to the task. Next, a migration file was generated inside `priv/repo/migrations/` which will create our database table that our schema maps to.

With our files in place, let's follow the instructions and run our migration:

```
$ mix ecto.migrate
Compiling 1 file (.ex)
Generated hello app

[info] == Running
Hello.Repo.Migrations.CreateUsers.change/0 forward

[info] create table users

[info] == Migrated in 0.0s
```

Mix assumes that we are in the development environment unless we tell it otherwise with `MIX_ENV=prod mix ecto.migrate`.

If we log in to our database server, and connect to our `hello_dev` database, we should see our `users` table. Ecto assumes that we want an integer column called `id` as our primary key, so we should see a sequence generated for that as well.

```
$ psql -U postgres

Type "help" for help.

postgres=# \connect hello_dev
You are now connected to database "hello_dev" as user
"postgres".
hello_dev=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | schema_migrations | table    | postgres
 public | users             | table    | postgres
 public | users_id_seq      | sequence | postgres
(3 rows)
hello_dev=# \q
```

If we take a look at the migration generated by `phx.gen.schema` in `priv/repo/migrations/`, we'll see that it will add the columns we specified. It will also add timestamp columns for `inserted_at` and `updated_at` which come from the `timestamps/1` function.

```
defmodule Hello.Repo.Migrations.CreateUsers do
  use Ecto.Migration

  def change do
    create table(:users) do
      add :name, :string
      add :email, :string
      add :bio, :string
      add :number_of_pets, :integer
```

```
      timestamps()
    end
  end
end
```

And here's what that translates to in the actual `users` table.

```
$ psql
hello_dev=# \d users
Table "public.users"
Column          |            Type             | Modifiers
----------------+-----------------------------+-----------
----------------------------------------
id              | bigint                      | not null
default nextval('users_id_seq'::regclass)
name            | character varying(255)      |
email           | character varying(255)      |
bio             | character varying(255)      |
number_of_pets  | integer                     |
inserted_at     | timestamp without time zone | not null
updated_at      | timestamp without time zone | not null
Indexes:
"users_pkey" PRIMARY KEY, btree (id)
```

Notice that we do get an `id` column as our primary key by default, even though it isn't listed as a field in our migration.

Repo configuration

Our `Hello.Repo` module is the foundation we need to work with databases in a Phoenix application. Phoenix generated it for us in `lib/hello/repo.ex`, and this is what it looks like.

```
defmodule Hello.Repo do
  use Ecto.Repo,
    otp_app: :hello,
    adapter: Ecto.Adapters.Postgres
end
```

It begins by defining the repository module. Then it configures our `otp_app` name, and the `adapter – Postgres`, in our case.

Our repo has three main tasks - to bring in all the common query functions from [`Ecto.Repo`], to set the `otp_app` name equal to our application name, and to configure our database adapter. We'll talk more about how to use `Hello.Repo` in a bit.

When `phx.new` generated our application, it included some basic repository configuration as well. Let's look at `config/dev.exs`.

```
...
# Configure your database
config :hello, Hello.Repo,
  username: "postgres",
  password: "postgres",
  hostname: "localhost",
  database: "hello_dev",
  show_sensitive_data_on_connection_error: true,
  pool_size: 10
...
```

We also have similar configuration in `config/test.exs` and `config/runtime.exs` (formerly `config/prod.secret.exs`) which can also be changed to match your actual credentials.

The schema

Ecto schemas are responsible for mapping Elixir values to external data sources, as well as mapping external data back into Elixir data structures. We can also define relationships to other schemas in our applications. For example, our `User` schema might have many posts, and each post would belong to a user. Ecto also handles data validation and type casting with changesets, which we'll discuss in a moment.

Here's the `User` schema that Phoenix generated for us.

```
defmodule Hello.User do
  use Ecto.Schema
  import Ecto.Changeset

  schema "users" do
    field :bio, :string
    field :email, :string
    field :name, :string
    field :number_of_pets, :integer

    timestamps()
  end

  @doc false
  def changeset(user, attrs) do
    user
    |> cast(attrs, [:name, :email, :bio,
:number_of_pets])
    |> validate_required([:name, :email, :bio,
:number_of_pets])
  end
end
```

Ecto schemas at their core are simply Elixir structs. Our `schema` block is what tells Ecto how to cast our `%User{}` struct fields to and from the external `users` table. Often, the ability to simply cast data to and from the database isn't enough and extra data validation is required. This is where Ecto changesets come in. Let's dive in!

Changesets and validations

Changesets define a pipeline of transformations our data needs to undergo before it will be ready for our application to use. These transformations might include type-casting, user input validation, and filtering out any extraneous parameters. Often we'll use changesets to validate user input before writing it to the database. Ecto repositories are also changeset-aware, which allows them not only to refuse invalid data, but also perform the minimal database updates possible by inspecting the changeset to know which fields have changed.

Let's take a closer look at our default changeset function.

```
def changeset(user, attrs) do
  user
  |> cast(attrs, [:name, :email, :bio, :number_of_pets])
  |> validate_required([:name, :email, :bio,
:number_of_pets])
end
```

Right now, we have two transformations in our pipeline. In the first call, we invoke `Ecto.Changeset.cast/3`, passing in our external parameters and marking which fields are required for validation.

`cast/3` first takes a struct, then the parameters (the proposed updates), and then the final field is the list of columns to be updated. `cast/3` also will only take fields that exist in the schema.

Next, `Ecto.Changeset.validate_required/3` checks that this list of fields is present in the changeset that `cast/3` returns. By default with the generator, all fields are required.

We can verify this functionality in `IEx`. Let's fire up our application inside IEx by running `iex -S mix`. In order to minimize typing and make this easier to read, let's alias our `Hello.User` struct.

```
$ iex -S mix

iex> alias Hello.User
Hello.User
```

Next, let's build a changeset from our schema with an empty `User` struct, and an empty map of parameters.

```
iex> changeset = User.changeset(%User{}, %{})
#Ecto.Changeset<
  action: nil,
```

```
  changes: %{},
  errors: [
    name: {"can't be blank", [validation: :required]},
    email: {"can't be blank", [validation: :required]},
    bio: {"can't be blank", [validation: :required]},
    number_of_pets: {"can't be blank", [validation:
:required]}
  ],
  data: #Hello.User<>,
  valid?: false
>
```

Once we have a changeset, we can check if it is valid.

```
iex> changeset.valid?
false
```

Since this one is not valid, we can ask it what the errors are.

```
iex> changeset.errors
[
  name: {"can't be blank", [validation: :required]},
  email: {"can't be blank", [validation: :required]},
  bio: {"can't be blank", [validation: :required]},
  number_of_pets: {"can't be blank", [validation:
:required]}
]
```

Now, let's make `number_of_pets` optional. In order to do this, we simply remove it from the list in the `changeset/2` function, in `Hello.User`.

```
    |> validate_required([:name, :email, :bio])
```

Now casting the changeset should tell us that only `name`, `email`, and `bio` can't be blank. We can test that by running `recompile()` inside IEx and then rebuilding our changeset.

```
iex> recompile()
Compiling 1 file (.ex)
:ok

iex> changeset = User.changeset(%User{}, %{})
#Ecto.Changeset<
  action: nil,
  changes: %{},
  errors: [
    name: {"can't be blank", [validation: :required]},
    email: {"can't be blank", [validation: :required]},
    bio: {"can't be blank", [validation: :required]}
  ],
  data: #Hello.User<>,
  valid?: false
>

iex> changeset.errors
[
  name: {"can't be blank", [validation: :required]},
  email: {"can't be blank", [validation: :required]},
  bio: {"can't be blank", [validation: :required]}
]
```

What happens if we pass a key-value pair that is neither defined in the schema nor required?

Inside our existing IEx shell, let's create a `params` map with valid values plus an extra `random_key: "random value"`.

```
iex> params = %{name: "Joe Example", email:
"joe@example.com", bio: "An example to all",
number_of_pets: 5, random_key: "random value"}
%{
  bio: "An example to all",
  email: "joe@example.com",
  name: "Joe Example",
  number_of_pets: 5,
  random_key: "random value"
}
```

Next, let's use our new `params` map to create another changeset.

```
iex> changeset = User.changeset(%User{}, params)
#Ecto.Changeset<
  action: nil,
  changes: %{
    bio: "An example to all",
    email: "joe@example.com",
    name: "Joe Example",
    number_of_pets: 5
  },
  errors: [],
  data: #Hello.User<>,
  valid?: true
>
```

Our new changeset is valid.

```
iex> changeset.valid?
true
```

We can also check the changeset's changes - the map we get after all of the transformations are complete.

```
iex(9)> changeset.changes
%{bio: "An example to all", email: "joe@example.com",
name: "Joe Example",
  number_of_pets: 5}
```

Notice that our `random_key` key and `"random_value"` value have been removed from the final changeset. Changesets allow us to cast external data, such as user input on a web form or data from a CSV file into valid data into our system. Invalid parameters will be stripped and bad data that is unable to be cast according to our schema will be highlighted in the changeset errors.

We can validate more than just whether a field is required or not. Let's take a look at some finer-grained validations.

What if we had a requirement that all biographies in our system must be at least two characters long? We can do this easily by adding another transformation to the pipeline in our changeset which validates the length of the `bio` field.

```
def changeset(user, attrs) do
  user
  |> cast(attrs, [:name, :email, :bio, :number_of_pets])
  |> validate_required([:name, :email, :bio,
:number_of_pets])
  |> validate_length(:bio, min: 2)
end
```

Now, if we try to cast data containing a value of `"A"` for our user's `bio`, we should see the failed validation in the changeset's errors.

```
iex> recompile()

iex> changeset = User.changeset(%User{}, %{bio: "A"})

iex> changeset.errors[:bio]
{"should be at least %{count} character(s)",
 [count: 2, validation: :length, kind: :min, type:
:string]}
```

If we also have a requirement for the maximum length that a bio can have, we can simply add another validation.

```
def changeset(user, attrs) do
  user
  |> cast(attrs, [:name, :email, :bio, :number_of_pets])
  |> validate_required([:name, :email, :bio,
:number_of_pets])
  |> validate_length(:bio, min: 2)
  |> validate_length(:bio, max: 140)
end
```

Let's say we want to perform at least some rudimentary format validation on the `email` field. All we want to check for is the presence of the `@`. The `Ecto.Changeset.validate_format/3` function is just what we need.

```
def changeset(user, attrs) do
  user
  |> cast(attrs, [:name, :email, :bio, :number_of_pets])
  |> validate_required([:name, :email, :bio,
:number_of_pets])
  |> validate_length(:bio, min: 2)
  |> validate_length(:bio, max: 140)
  |> validate_format(:email, ~r/@/)
end
```

If we try to cast a user with an email of `"example.com"`, we should see an error message like the following:

```
iex> recompile()

iex> changeset = User.changeset(%User{}, %{email:
"example.com"})

iex> changeset.errors[:email]
{"has invalid format", [validation: :format]}
```

There are many more validations and transformations we can perform in a changeset. Please see the Ecto Changeset documentation for more information.

Data persistence

We've explored migrations and schemas, but we haven't yet persisted any of our schemas or changesets. We briefly looked at our repository module in `lib/hello/repo.ex` earlier, and now it's time to put it to use.

Ecto repositories are the interface into a storage system, be it a database like PostgreSQL or an external service like a RESTful API. The `Repo` module's

purpose is to take care of the finer details of persistence and data querying for us. As the caller, we only care about fetching and persisting data. The `Repo` module takes care of the underlying database adapter communication, connection pooling, and error translation for database constraint violations.

Let's head back over to IEx with `iex -S mix`, and insert a couple of users into the database.

```
iex> alias Hello.{Repo, User}
[Hello.Repo, Hello.User]

iex> Repo.insert(%User{email: "user1@example.com"})
[debug] QUERY OK db=6.5ms queue=0.5ms idle=1358.3ms
INSERT INTO "users" ("email","inserted_at","updated_at")
VALUES ($1,$2,$3) RETURNING "id" ["user1@example.com",
~N[2021-02-25 01:58:55], ~N[2021-02-25 01:58:55]]
{:ok,
 %Hello.User{
   __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
   bio: nil,
   email: "user1@example.com",
   id: 1,
   inserted_at: ~N[2021-02-25 01:58:55],
   name: nil,
   number_of_pets: nil,
   updated_at: ~N[2021-02-25 01:58:55]
 }}

iex> Repo.insert(%User{email: "user2@example.com"})
[debug] QUERY OK db=1.3ms idle=1402.7ms
INSERT INTO "users" ("email","inserted_at","updated_at")
VALUES ($1,$2,$3) RETURNING "id" ["user2@example.com",
~N[2021-02-25 02:03:28], ~N[2021-02-25 02:03:28]]
{:ok,
 %Hello.User{
   __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
   bio: nil,
   email: "user2@example.com",
   id: 2,
   inserted_at: ~N[2021-02-25 02:03:28],
   name: nil,
   number_of_pets: nil,
```

```
    updated_at: ~N[2021-02-25 02:03:28]
  }}
```

We started by aliasing our `User` and `Repo` modules for easy access. Next, we called `Repo.insert/2` with a User struct. Since we are in the `dev` environment, we can see the debug logs for the query our repository performed when inserting the underlying `%User{}` data. We received a two-element tuple back with `{:ok, %User{}}`, which lets us know the insertion was successful.

We could also insert a user by passing a changeset to `Repo.insert/2`. If the changeset is valid, the repository will use an optimized database query to insert the record, and return a two-element tuple back, as above. If the changeset is not valid, we receive a two-element tuple consisting of `:error` plus the invalid changeset.

With a couple of users inserted, let's fetch them back out of the repo.

```
iex> Repo.all(User)
[debug] QUERY OK source="users" db=5.8ms queue=1.4ms
idle=1672.0ms
SELECT u0."id", u0."bio", u0."email", u0."name",
u0."number_of_pets", u0."inserted_at", u0."updated_at"
FROM "users" AS u0 []
[
  %Hello.User{
    __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
    bio: nil,
    email: "user1@example.com",
    id: 1,
    inserted_at: ~N[2021-02-25 01:58:55],
    name: nil,
    number_of_pets: nil,
    updated_at: ~N[2021-02-25 01:58:55]
  },
  %Hello.User{
    __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
    bio: nil,
    email: "user2@example.com",
    id: 2,
```

```
    inserted_at: ~N[2021-02-25 02:03:28],
    name: nil,
    number_of_pets: nil,
    updated_at: ~N[2021-02-25 02:03:28]
  }
]
```

That was easy! `Repo.all/1` takes a data source, our `User` schema in this case, and translates that to an underlying SQL query against our database. After it fetches the data, the Repo then uses our Ecto schema to map the database values back into Elixir data structures according to our `User` schema. We're not just limited to basic querying – Ecto includes a full-fledged query DSL for advanced SQL generation. In addition to a natural Elixir DSL, Ecto's query engine gives us multiple great features, such as SQL injection protection and compile-time optimization of queries. Let's try it out.

```
iex> import Ecto.Query
Ecto.Query

iex> Repo.all(from u in User, select: u.email)
[debug] QUERY OK source="users" db=0.8ms queue=0.9ms
idle=1634.0ms
SELECT u0."email" FROM "users" AS u0 []
["user1@example.com", "user2@example.com"]
```

First, we imported [`Ecto.Query`], which imports the `from/2` macro of Ecto's Query DSL. Next, we built a query which selects all the email addresses in our users table. Let's try another example.

```
iex> Repo.one(from u in User, where: ilike(u.email,
"%1%"),
                                select: count(u.id))
[debug] QUERY OK source="users" db=1.6ms SELECT
count(u0."id") FROM "users" AS u0 WHERE (u0."email" ILIKE
'%1%') []
1
```

Now we're starting to get a taste of Ecto's rich querying capabilities. We used `Repo.one/2` to fetch the count of all users with an email address containing `1`, and received the expected count in return. This just scratches the surface of Ecto's query interface, and much more is supported such as sub-querying, interval queries, and advanced select statements. For example, let's build a query to fetch a map of all user id's to their email addresses.

```
iex> Repo.all(from u in User, select: %{u.id => u.email})
[debug] QUERY OK source="users" db=0.9ms
SELECT u0."id", u0."email" FROM "users" AS u0 []
[
  %{1 => "user1@example.com"},
  %{2 => "user2@example.com"}
]
```

That little query packed a big punch. It both fetched all user emails from the database and efficiently built a map of the results in one go. You should browse the Ecto.Query documentation to see the breadth of supported query features.

In addition to inserts, we can also perform updates and deletes with `Repo.update/2` and `Repo.delete/2` to update or delete a single schema. Ecto also supports bulk persistence with the `Repo.insert_all/3`, `Repo.update_all/3`, and `Repo.delete_all/2` functions.

There is quite a bit more that Ecto can do and we've only barely scratched the surface. With a solid Ecto foundation in place, we're now ready to continue building our app and integrate the web-facing application with our backend persistence. Along the way, we'll expand our Ecto knowledge and learn how to properly isolate our web interface from the underlying details of our system. Please take a look at the Ecto documentation for the rest of the story.

In our contexts guide, we'll find out how to wrap up our Ecto access and business logic behind modules that group related functionality. We'll see how

Phoenix helps us design maintainable applications, and we'll find out about other neat Ecto features along the way.

Using other databases

Phoenix applications are configured to use PostgreSQL by default, but what if we want to use another database, such as MySQL? In this section, we'll walk through changing that default whether we are about to create a new application, or whether we have an existing one configured for PostgreSQL.

If we are about to create a new application, configuring our application to use MySQL is easy. We can simply pass the `--database mysql` flag to `phx.new` and everything will be configured correctly.

```
$ mix phx.new hello_phoenix --database mysql
```

This will set up all the correct dependencies and configuration for us automatically. Once we install those dependencies with `mix deps.get`, we'll be ready to begin working with Ecto in our application.

If we have an existing application, all we need to do is switch adapters and make some small configuration changes.

To switch adapters, we need to remove the Postgrex dependency and add a new one for MyXQL instead.

Let's open up our `mix.exs` file and do that now.

```
defmodule HelloPhoenix.MixProject do
  use Mix.Project

  . . .
  # Specifies your project dependencies.
  #
  # Type `mix help deps` for examples and options.
  defp deps do
```

```
      [
        {:phoenix, "~> 1.4.0"},
        {:phoenix_ecto, "~> 4.4"},
        {:ecto_sql, "~> 3.10"},
        {:myxql, ">= 0.0.0"},
        ...
      ]
    end
  end
```

Next, we need to configure our adapter to use the default MySQL credentials by updating `config/dev.exs`:

```
config :hello_phoenix, HelloPhoenix.Repo,
  username: "root",
  password: "",
  database: "hello_phoenix_dev"
```

If we have an existing configuration block for our `HelloPhoenix.Repo`, we can simply change the values to match our new ones. You also need to configure the correct values in the `config/test.exs` and `config/runtime.exs` (formerly `config/prod.secret.exs`) files as well.

The last change is to open up `lib/hello_phoenix/repo.ex` and make sure to set the `:adapter` to `Ecto.Adapters.MyXQL`.

Now all we need to do is fetch our new dependency, and we'll be ready to go.

```
$ mix deps.get
```

With our new adapter installed and configured, we're ready to create our database.

```
$ mix ecto.create
```

The database for HelloPhoenix.Repo has been created. We're also ready to run any migrations, or do anything else with Ecto that we may choose.

```
$ mix ecto.migrate
[info] == Running
HelloPhoenix.Repo.Migrations.CreateUser.change/0 forward
[info] create table users
[info] == Migrated in 0.2s
```

Other options

While Phoenix uses the `Ecto` project to interact with the data access layer, there are many other data access options, some even built into the Erlang standard library. ETS – available in Ecto via `etso` – and DETS are key-value data stores built into OTP. OTP also provides a relational database called Mnesia with its own query language called QLC. Both Elixir and Erlang also have a number of libraries for working with a wide range of popular data stores.

The data world is your oyster, but we won't be covering these options in these guides.

Contexts

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Request life-cycle guide.
>
> **Requirement**: This guide expects that you have gone through the Ecto guide.

So far, we've built pages, wired up controller actions through our routers, and learned how Ecto allows data to be validated and persisted. Now it's time to tie it all together by writing web-facing features that interact with our greater Elixir application.

When building a Phoenix project, we are first and foremost building an Elixir application. Phoenix's job is to provide a web interface into our Elixir application. Naturally, we compose our applications with modules and functions, but we often assign specific responsibilities to certain modules and give them names: such as controllers, routers, and live views.

As everything else, contexts in Phoenix are modules, but with the distinct reponsibility of drawing boundaries and grouping functionality. In other words, they allow us to reason and discuss about application design.

Thinking about contexts

Contexts are dedicated modules that expose and group related functionality. For example, anytime you call Elixir's standard library, be it `Logger.info/1` or `Stream.map/2`, you are accessing different contexts. Internally, Elixir's logger is made of multiple modules, but we never interact with those modules directly. We call the `Logger` module the context, exactly because it exposes and groups all of the logging functionality.

By giving modules that expose and group related functionality the name **contexts**, we help developers identify these patterns and talk about them. At the end of the day, contexts are just modules, as are your controllers, views, etc.

In Phoenix, contexts often encapsulate data access and data validation. They often talk to a database or APIs. Overall, think of them as boundaries to decouple and isolate parts of your application. Let's use these ideas to build out our web application. Our goal is to build an ecommerce system where we can showcase products, allow users to add products to their cart, and complete their orders.

> How to read this guide: Using the context generators is a great way for beginners and intermediate Elixir programmers alike to get up and running quickly while thoughtfully writing their applications. This guide focuses on those readers.

Adding a Catalog Context

An ecommerce platform has wide-reaching coupling across a codebase so it's important to think about writing well-defined modules. With that in mind, our goal is to build a product catalog API that handles creating, updating, and deleting the products available in our system. We'll start off with the basic features of showcasing our products, and we will add shopping cart features later. We'll see how starting with a solid foundation with isolated boundaries allows us to grow our application naturally as we add functionality.

Phoenix includes the `mix phx.gen.html`, `mix phx.gen.json`, `mix phx.gen.live`, and `mix phx.gen.context` generators that apply the ideas of isolating functionality in our applications into contexts. These generators are a great way to hit the ground running while Phoenix nudges you in the right direction to grow your application. Let's put these tools to use for our new product catalog context.

In order to run the context generators, we need to come up with a module name that groups the related functionality that we're building. In the Ecto

guide, we saw how we can use Changesets and Repos to validate and persist user schemas, but we didn't integrate this with our application at large. In fact, we didn't think about where a "user" in our application should live at all. Let's take a step back and think about the different parts of our system. We know that we'll have products to showcase on pages for sale, along with descriptions, pricing, etc. Along with selling products, we know we'll need to support carting, order checkout, and so on. While the products being purchased are related to the cart and checkout processes, showcasing a product and managing the *exhibition* of our products is distinctly different than tracking what a user has placed in their cart or how an order is placed. A `Catalog` context is a natural place for the management of our product details and the showcasing of those products we have for sale.

> **Naming things is hard**
>
> If you're stuck when trying to come up with a context name when the grouped functionality in your system isn't yet clear, you can simply use the plural form of the resource you're creating. For example, a `Products` context for managing products. As you grow your application and the parts of your system become clear, you can simply rename the context to a more refined one.

To jump-start our catalog context, we'll use `mix phx.gen.html` which creates a context module that wraps up Ecto access for creating, updating, and deleting products, along with web files like controllers and templates for the web interface into our context. Run the following command at your project root:

```
$ mix phx.gen.html Catalog Product products title:string
\
description:string price:decimal views:integer

* creating
lib/hello_web/controllers/product_controller.ex
* creating
lib/hello_web/controllers/product_html/edit.html.heex
* creating
lib/hello_web/controllers/product_html/index.html.heex
* creating
lib/hello_web/controllers/product_html/new.html.heex
```

```
* creating
lib/hello_web/controllers/product_html/show.html.heex
* creating
lib/hello_web/controllers/product_html/product_form.html.
heex
* creating lib/hello_web/controllers/product_html.ex
* creating
test/hello_web/controllers/product_controller_test.exs
* creating lib/hello/catalog/product.ex
* creating
priv/repo/migrations/20210201185747_create_products.exs
* creating lib/hello/catalog.ex
* injecting lib/hello/catalog.ex
* creating test/hello/catalog_test.exs
* injecting test/hello/catalog_test.exs
* creating test/support/fixtures/catalog_fixtures.ex
* injecting test/support/fixtures/catalog_fixtures.ex

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

    resources "/products", ProductController


Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

Phoenix generated the web files as expected in `lib/hello_web/`. We can also see our context files were generated inside a `lib/hello/catalog.ex` file and our product schema in the directory of the same name. Note the difference between `lib/hello` and `lib/hello_web`. We have a `Catalog` module to serve as the public API for product catalog functionality, as well as a `Catalog.Product` struct, which is an Ecto schema for casting and validating product data. Phoenix also provided web and context tests for us, it also included test helpers for creating entities via the `Hello.Catalog` context, which we'll look at later. For now, let's follow the instructions and add the route according to the console instructions, in `lib/hello_web/router.ex`:

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

    get "/", PageController, :index
+   resources "/products", ProductController
  end
```

With the new route in place, Phoenix reminds us to update our repo by running `mix ecto.migrate`, but first we need to make a few tweaks to the generated migration in `priv/repo/migrations/*_create_products.exs`:

```
  def change do
    create table(:products) do
      add :title, :string
      add :description, :string
-     add :price, :decimal
+     add :price, :decimal, precision: 15, scale: 6,
null: false
-     add :views, :integer
+     add :views, :integer, default: 0, null: false

      timestamps()
    end
```

We modified our price column to a specific precision of 15, scale of 6, along with a not-null constraint. This ensures we store currency with proper precision for any mathematical operations we may perform. Next, we added a default value and not-null constraint to our views count. With our changes in place, we're ready to migrate up our database. Let's do that now:

```
$ mix ecto.migrate
14:09:02.260 [info] == Running 20210201185747
Hello.Repo.Migrations.CreateProducts.change/0 forward

14:09:02.262 [info] create table products

14:09:02.273 [info] == Migrated 20210201185747 in 0.0s
```

Before we jump into the generated code, let's start the server with `mix phx.server` and visit http://localhost:4000/products. Let's follow the "New Product" link and click the "Save" button without providing any input. We should be greeted with the following output:

```
Oops, something went wrong! Please check the errors
below.
```

When we submit the form, we can see all the validation errors inline with the inputs. Nice! Out of the box, the context generator included the schema fields in our form template and we can see our default validations for required inputs are in effect. Let's enter some example product data and resubmit the form:

```
Product created successfully.

Title: Metaprogramming Elixir
Description: Write Less Code, Get More Done (and Have
Fun!)
Price: 15.000000
Views: 0
```

If we follow the "Back" link, we get a list of all products, which should contain the one we just created. Likewise, we can update this record or delete it. Now that we've seen how it works in the browser, it's time to take a look at the generated code.

Starting with generators

That little `mix phx.gen.html` command packed a surprising punch. We got a lot of functionality out-of-the-box for creating, updating, and deleting products in our catalog. This is far from a full-featured app, but remember, generators are first and foremost learning tools and a starting point for you to begin building real features. Code generation can't solve all your problems,

but it will teach you the ins and outs of Phoenix and nudge you towards the proper mindset when designing your application.

Let's first check out the `ProductController` that was generated in `lib/hello_web/controllers/product_controller.ex`:

```
defmodule HelloWeb.ProductController do
  use HelloWeb, :controller

  alias Hello.Catalog
  alias Hello.Catalog.Product

  def index(conn, _params) do
    products = Catalog.list_products()
    render(conn, :index, products: products)
  end

  def new(conn, _params) do
    changeset = Catalog.change_product(%Product{})
    render(conn, :new, changeset: changeset)
  end

  def create(conn, %{"product" => product_params}) do
    case Catalog.create_product(product_params) do
      {:ok, product} ->
        conn
        |> put_flash(:info, "Product created
successfully.")
        |> redirect(to: ~p"/products/#{product}")

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

  def show(conn, %{"id" => id}) do
    product = Catalog.get_product!(id)
    render(conn, :show, product: product)
  end
  ...
end
```

We've seen how controllers work in our controller guide, so the code probably isn't too surprising. What is worth noticing is how our controller calls into the `Catalog` context. We can see that the `index` action fetches a list of products with `Catalog.list_products/0`, and how products are persisted in the `create` action with `Catalog.create_product/1`. We haven't yet looked at the catalog context, so we don't yet know how product fetching and creation is happening under the hood – *but that's the point*. Our Phoenix controller is the web interface into our greater application. It shouldn't be concerned with the details of how products are fetched from the database or persisted into storage. We only care about telling our application to perform some work for us. This is great because our business logic and storage details are decoupled from the web layer of our application. If we move to a full-text storage engine later for fetching products instead of a SQL query, our controller doesn't need to be changed. Likewise, we can reuse our context code from any other interface in our application, be it a channel, mix task, or long-running process importing CSV data.

In the case of our `create` action, when we successfully create a product, we use `Phoenix.Controller.put_flash/3` to show a success message, and then we redirect to the router's product show page. Conversely, if `Catalog.create_product/1` fails, we render our `"new.html"` template and pass along the Ecto changeset for the template to lift error messages from.

Next, let's dig deeper and check out our `Catalog` context in `lib/hello/catalog.ex`:

```
defmodule Hello.Catalog do
  @moduledoc """
  The Catalog context.
  """

  import Ecto.Query, warn: false
  alias Hello.Repo

  alias Hello.Catalog.Product

  @doc """
```

```
  Returns the list of products.

  ## Examples

      iex> list_products()
      [%Product{}, ...]

  """
  def list_products do
    Repo.all(Product)
  end
  ...
end
```

This module will be the public API for all product catalog functionality in our system. For example, in addition to product detail management, we may also handle product category classification and product variants for things like optional sizing, trims, etc. If we look at the `list_products/0` function, we can see the private details of product fetching. And it's super simple. We have a call to `Repo.all(Product)`. We saw how Ecto repo queries worked in the Ecto guide, so this call should look familiar. Our `list_products` function is a generalized function name specifying the *intent* of our code – namely to list products. The details of that intent where we use our Repo to fetch the products from our PostgreSQL database is hidden from our callers. This is a common theme we'll see re-iterated as we use the Phoenix generators. Phoenix will push us to think about where we have different responsibilities in our application, and then to wrap up those different areas behind well-named modules and functions that make the intent of our code clear, while encapsulating the details.

Now we know how data is fetched, but how are products persisted? Let's take a look at the `Catalog.create_product/1` function:

```
  @doc """
  Creates a product.

  ## Examples

      iex> create_product(%{field: value})
```

```
      {:ok, %Product{}}

      iex> create_product(%{field: bad_value})
      {:error, %Ecto.Changeset{}}

  """
  def create_product(attrs \\ %{}) do
    %Product{}
    |> Product.changeset(attrs)
    |> Repo.insert()
  end
```

There's more documentation than code here, but a couple of things are important to highlight. First, we can see again that our Ecto Repo is used under the hood for database access. You probably also noticed the call to `Product.changeset/2`. We talked about changesets before, and now we see them in action in our context.

If we open up the `Product` schema in `lib/hello/catalog/product.ex`, it will look immediately familiar:

```
defmodule Hello.Catalog.Product do
  use Ecto.Schema
  import Ecto.Changeset

  schema "products" do
    field :description, :string
    field :price, :decimal
    field :title, :string
    field :views, :integer

    timestamps()
  end

  @doc false
  def changeset(product, attrs) do
    product
    |> cast(attrs, [:title, :description, :price,
:views])
    |> validate_required([:title, :description, :price,
:views])
```

```
  end
end
```

This is just what we saw before when we ran `mix phx.gen.schema`, except here we see a `@doc false` above our `changeset/2` function. This tells us that while this function is publicly callable, it's not part of the public context API. Callers that build changesets do so via the context API. For example, `Catalog.create_product/1` calls into our `Product.changeset/2` to build the changeset from user input. Callers, such as our controller actions, do not access `Product.changeset/2` directly. All interaction with our product changesets is done through the public `Catalog` context.

Adding Catalog functions

As we've seen, your context modules are dedicated modules that expose and group related functionality. Phoenix generates generic functions, such as `list_products` and `update_product`, but they only serve as a basis for you to grow your business logic and application from. Let's add one of the basic features of our catalog by tracking product page view count.

For any ecommerce system, the ability to track how many times a product page has been viewed is essential for marketing, suggestions, ranking, etc. While we could try to use the existing `Catalog.update_product` function, along the lines of `Catalog.update_product(product, %{views: product.views + 1})`, this would not only be prone to race conditions, but it would also require the caller to know too much about our Catalog system. To see why the race condition exists, let's walk through the possible execution of events:

Intuitively, you would assume the following events:

1. User 1 loads the product page with count of 13
2. User 1 saves the product page with count of 14
3. User 2 loads the product page with count of 14
4. User 2 saves the product page with count of 15

While in practice this would happen:

1. User 1 loads the product page with count of 13
2. User 2 loads the product page with count of 13
3. User 1 saves the product page with count of 14
4. User 2 saves the product page with count of 14

The race conditions would make this an unreliable way to update the existing table since multiple callers may be updating out of date view values. There's a better way.

Let's think of a function that describes what we want to accomplish. Here's how we would like to use it:

```
product = Catalog.inc_page_views(product)
```

That looks great. Our callers will have no confusion over what this function does, and we can wrap up the increment in an atomic operation to prevent race conditions.

Open up your catalog context (`lib/hello/catalog.ex`), and add this new function:

```
  def inc_page_views(%Product{} = product) do
    {1, [%Product{views: views}]} =
      from(p in Product, where: p.id == ^product.id,
select: [:views])
      |> Repo.update_all(inc: [views: 1])

    put_in(product.views, views)
  end
```

We built a query for fetching the current product given its ID which we pass to `Repo.update_all` . Ecto's `Repo.update_all` allows us to perform batch updates against the database, and is perfect for atomically updating values, such as incrementing our views count. The result of the repo

operation returns the number of updated records, along with the selected schema values specified by the `select` option. When we receive the new product views, we use `put_in(product.views, views)` to place the new view count within the product struct.

With our context function in place, let's make use of it in our product controller. Update your `show` action in `lib/hello_web/controllers/product_controller.ex` to call our new function:

```
def show(conn, %{"id" => id}) do
  product =
    id
    |> Catalog.get_product!()
    |> Catalog.inc_page_views()

  render(conn, :show, product: product)
end
```

We modified our `show` action to pipe our fetched product into `Catalog.inc_page_views/1`, which will return the updated product. Then we rendered our template just as before. Let's try it out. Refresh one of your product pages a few times and watch the view count increase.

We can also see our atomic update in action in the ecto debug logs:

```
[debug] QUERY OK source="products" db=0.5ms idle=834.5ms
UPDATE "products" AS p0 SET "views" = p0."views" + $1
WHERE (p0."id" = $2) RETURNING p0."views" [1, 1]
```

Good work!

As we've seen, designing with contexts gives you a solid foundation to grow your application from. Using discrete, well-defined APIs that expose the intent of your system allows you to write more maintainable applications

with reusable code. Now that we know how to start extending our context API, lets explore handling relationships within a context.

In-context relationships

Our basic catalog features are nice, but let's take it up a notch by categorizing products. Many ecommerce solutions allow products to be categorized in different ways, such as a product being marked for fashion, power tools, and so on. Starting with a one-to-one relationship between product and categories will cause major code changes later if we need to start supporting multiple categories. Let's set up a category association that will allow us to start off tracking a single category per product, but easily support more later as we grow our features.

For now, categories will contain only textual information. Our first order of business is to decide where categories live in the application. We have our `Catalog` context, which manages the exhibition of our products. Product categorization is a natural fit here. Phoenix is also smart enough to generate code inside an existing context, which makes adding new resources to a context a breeze. Run the following command at your project root:

> Sometimes it may be tricky to determine if two resources belong to the same context or not. In those cases, prefer distinct contexts per resource and refactor later if necessary. Otherwise you can easily end up with large contexts of loosely related entities. Also keep in mind that the fact two resources are related does not necessarily mean they belong to the same context, otherwise you would quickly end up with one large context, as the majority of resources in an application are connected to each other. To sum it up: if you are unsure, you should prefer separate modules (contexts).

```
$ mix phx.gen.context Catalog Category categories \
title:string:unique

You are generating into an existing context.
...
```

```
Would you like to proceed? [Yn] y
* creating lib/hello/catalog/category.ex
* creating
priv/repo/migrations/20210203192325_create_categories.exs
* injecting lib/hello/catalog.ex
* injecting test/hello/catalog_test.exs
* injecting test/support/fixtures/catalog_fixtures.ex

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

This time around, we used `mix phx.gen.context`, which is just like `mix phx.gen.html`, except it doesn't generate the web files for us. Since we already have controllers and templates for managing products, we can integrate the new category features into our existing web form and product show page. We can see we now have a new `Category` schema alongside our product schema at `lib/hello/catalog/category.ex`, and Phoenix told us it was *injecting* new functions in our existing Catalog context for the category functionality. The injected functions will look very familiar to our product functions, with new functions like `create_category`, `list_categories`, and so on. Before we migrate up, we need to do a second bit of code generation. Our category schema is great for representing an individual category in the system, but we need to support a many-to-many relationship between products and categories. Fortunately, ecto allows us to do this simply with a join table, so let's generate that now with the `ecto.gen.migration` command:

```
$ mix ecto.gen.migration create_product_categories

* creating
priv/repo/migrations/20210203192958_create_product_catego
ries.exs
```

Next, let's open up the new migration file and add the following code to the `change` function:

```
defmodule Hello.Repo.Migrations.CreateProductCategories
do
  use Ecto.Migration

  def change do
    create table(:product_categories, primary_key: false)
do
      add :product_id, references(:products, on_delete:
:delete_all)
      add :category_id, references(:categories,
on_delete: :delete_all)
    end

    create index(:product_categories, [:product_id])
    create unique_index(:product_categories,
[:category_id, :product_id])
  end
end
```

We created a `product_categories` table and used the `primary_key: false` option since our join table does not need a primary key. Next we defined our `:product_id` and `:category_id` foreign key fields, and passed `on_delete: :delete_all` to ensure the database prunes our join table records if a linked product or category is deleted. By using a database constraint, we enforce data integrity at the database level, rather than relying on ad-hoc and error-prone application logic.

Next, we created indexes for our foreign keys, one of which is a unique index to ensure a product cannot have duplicate categories. Note that we do not necessarily need single-column index for `category_id` because it is in the leftmost prefix of multicolumn index, which is enough for the database optimizer. Adding a redundant index, on the other hand, only adds overhead on write.

With our migrations in place, we can migrate up.

```
$ mix ecto.migrate
```

```
18:20:36.489 [info] == Running 20210222231834
Hello.Repo.Migrations.CreateCategories.change/0 forward

18:20:36.493 [info] create table categories

18:20:36.508 [info] create index categories_title_index

18:20:36.512 [info] == Migrated 20210222231834 in 0.0s

18:20:36.547 [info] == Running 20210222231930
Hello.Repo.Migrations.CreateProductCategories.change/0
forward

18:20:36.547 [info] create table product_categories

18:20:36.557 [info] create index
product_categories_product_id_index

18:20:36.560 [info]  create index
product_categories_category_id_product_id_index

18:20:36.562 [info] == Migrated 20210222231930 in 0.0s
```

Now that we have a `Catalog.Product` schema and a join table to associate products and categories, we're nearly ready to start wiring up our new features. Before we dive in, we first need real categories to select in our web UI. Let's quickly seed some new categories in the application. Add the following code to your seeds file in `priv/repo/seeds.exs`:

```
for title <- ["Home Improvement", "Power Tools",
"Gardening", "Books", "Education"] do
  {:ok, _} = Hello.Catalog.create_category(%{title:
title})
end
```

We simply enumerate over a list of category titles and use the generated `create_category/1` function of our catalog context to persist the new records. We can run the seeds with `mix run`:

```
$ mix run priv/repo/seeds.exs

[debug] QUERY OK db=3.1ms decode=1.1ms queue=0.7ms
idle=2.2ms
INSERT INTO "categories"
("title","inserted_at","updated_at") VALUES ($1,$2,$3)
RETURNING "id" ["Home Improvement", ~N[2021-02-03
19:39:53], ~N[2021-02-03 19:39:53]]
[debug] QUERY OK db=1.2ms queue=1.3ms idle=12.3ms
INSERT INTO "categories"
("title","inserted_at","updated_at") VALUES ($1,$2,$3)
RETURNING "id" ["Power Tools", ~N[2021-02-03 19:39:53],
~N[2021-02-03 19:39:53]]
[debug] QUERY OK db=1.1ms queue=1.1ms idle=15.1ms
INSERT INTO "categories"
("title","inserted_at","updated_at") VALUES ($1,$2,$3)
RETURNING "id" ["Gardening", ~N[2021-02-03 19:39:53],
~N[2021-02-03 19:39:53]]
[debug] QUERY OK db=2.4ms queue=1.0ms idle=17.6ms
INSERT INTO "categories"
("title","inserted_at","updated_at") VALUES ($1,$2,$3)
RETURNING "id" ["Books", ~N[2021-02-03 19:39:53],
~N[2021-02-03 19:39:53]]
```

Perfect. Before we integrate categories in the web layer, we need to let our context know how to associate products and categories. First, open up `lib/hello/catalog/product.ex` and add the following association:

```
+ alias Hello.Catalog.Category

  schema "products" do
    field :description, :string
    field :price, :decimal
    field :title, :string
    field :views, :integer

+   many_to_many :categories, Category, join_through:
"product_categories", on_replace: :delete

    timestamps()
  end
```

We used `Ecto.Schema`'s `many_to_many` macro to let Ecto know how to associate our product to multiple categories through the `"product_categories"` join table. We also used the `on_replace: :delete` option to declare that any existing join records should be deleted when we are changing our categories.

With our schema associations set up, we can implement the selection of categories in our product form. To do so, we need to translate the user input of catalog IDs from the front-end to our many-to-many association. Fortunately Ecto makes this a breeze now that our schema is set up. Open up your catalog context and make the following changes:

```
+ alias Hello.Catalog.Category

- def get_product!(id), do: Repo.get!(Product, id)
+ def get_product!(id) do
+   Product |> Repo.get!(id) |> Repo.preload(:categories)
+ end

  def create_product(attrs \\ %{}) do
    %Product{}
-   |> Product.changeset(attrs)
+   |> change_product(attrs)
    |> Repo.insert()
  end

  def update_product(%Product{} = product, attrs) do
    product
-   |> Product.changeset(attrs)
+   |> change_product(attrs)
    |> Repo.update()
  end

  def change_product(%Product{} = product, attrs \\ %{})
do
-   Product.changeset(product, attrs)
+   categories =
list_categories_by_id(attrs["category_ids"])

+   product
+   |> Repo.preload(:categories)
```

```
+    |> Product.changeset(attrs)
+    |> Ecto.Changeset.put_assoc(:categories, categories)
   end

+ def list_categories_by_id(nil), do: []
+ def list_categories_by_id(category_ids) do
+   Repo.all(from c in Category, where: c.id in
^category_ids)
+ end
```

First, we added `Repo.preload` to preload our categories when we fetch a product. This will allow us to reference `product.categories` in our controllers, templates, and anywhere else we want to make use of category information. Next, we modified our `create_product` and `update_product` functions to call into our existing `change_product` function to produce a changeset. Within `change_product` we added a lookup to find all categories if the `"category_ids"` attribute is present. Then we preloaded categories and called `Ecto.Changeset.put_assoc` to place the fetched categories into the changeset. Finally, we implemented the `list_categories_by_id/1` function to query the categories matching the category IDs, or return an empty list if no `"category_ids"` attribute is present. Now our `create_product` and `update_product` functions receive a changeset with the category associations all ready to go once we attempt an insert or update against our repo.

Next, let's expose our new feature to the web by adding the category input to our product form. To keep our form template tidy, let's write a new function to wrap up the details of rendering a category select input for our product. Open up your `ProductHTML` view in `lib/hello_web/controllers/product_html.ex` and key this in:

```
  def category_opts(changeset) do
    existing_ids =
      changeset
      |> Ecto.Changeset.get_change(:categories, [])
      |> Enum.map(& &1.data.id)

    for cat <- Hello.Catalog.list_categories(),
```

```
        do: [key: cat.title, value: cat.id, selected:
cat.id in existing_ids]
  end
```

We added a new `category_opts/1` function which generates the select options for a multiple select tag we will add soon. We calculated the existing category IDs from our changeset, then used those values when we generate the select options for the input tag. We did this by enumerating over all of our categories and returning the appropriate `key`, `value`, and `selected` values. We marked an option as selected if the category ID was found in those category IDs in our changeset.

With our `category_opts` function in place, we can open up `lib/hello_web/controllers/product_html/product_form.html.heex` and add:

```
  ...
  <.input field={f[:views]} type="number" label="Views"
/>

+ <.input field={f[:category_ids]} type="select"
multiple={true} options={category_opts(@changeset)} />

  <:actions>
    <.button>Save Product</.button>
  </:actions>
```

We added a `category_select` above our save button. Now let's try it out. Next, let's show the product's categories in the product show template. Add the following code to the list in `lib/hello_web/controllers/product_html/show.html.heex`:

```
<.list>
  ...
+ <:item title="Categories">
+   <ul>
+     <li :for={cat <- @product.categories}>{cat.title}
```

```
</li>
+    </ul>
+ </:item>
</.list>
```

Now if we start the server with `mix phx.server` and visit http://localhost:4000/products/new, we'll see the new category multiple select input. Enter some valid product details, select a category or two, and click save.

```
Title: Elixir Flashcards
Description: Flash card set for the Elixir programming
language
Price: 5.000000
Views: 0
Categories:
Education
Books
```

It's not much to look at yet, but it works! We added relationships within our context complete with data integrity enforced by the database. Not bad. Let's keep building!

Cross-context dependencies

Now that we have the beginnings of our product catalog features, let's begin to work on the other main features of our application – carting products from the catalog. In order to properly track products that have been added to a user's cart, we'll need a new place to persist this information, along with point-in-time product information like the price at time of carting. This is necessary so we can detect product price changes in the future. We know what we need to build, but now we need to decide where the cart functionality lives in our application.

If we take a step back and think about the isolation of our application, the exhibition of products in our catalog distinctly differs from the

responsibilities of managing a user's cart. A product catalog shouldn't care about the rules of our shopping cart system, and vice-versa. There's a clear need here for a separate context to handle the new cart responsibilities. Let's call it `ShoppingCart`.

Let's create a `ShoppingCart` context to handle basic cart duties. Before we write code, let's imagine we have the following feature requirements:

1. Add products to a user's cart from the product show page
2. Store point-in-time product price information at time of carting
3. Store and update quantities in cart
4. Calculate and display sum of cart prices

From the description, it's clear we need a `Cart` resource for storing the user's cart, along with a `CartItem` to track products in the cart. With our plan set, let's get to work. Run the following command to generate our new context:

```
$ mix phx.gen.context ShoppingCart Cart carts
user_uuid:uuid:unique

* creating lib/hello/shopping_cart/cart.ex
* creating
priv/repo/migrations/20210205203128_create_carts.exs
* creating lib/hello/shopping_cart.ex
* injecting lib/hello/shopping_cart.ex
* creating test/hello/shopping_cart_test.exs
* injecting test/hello/shopping_cart_test.exs
* creating
test/support/fixtures/shopping_cart_fixtures.ex
* injecting
test/support/fixtures/shopping_cart_fixtures.ex

Some of the generated database columns are unique. Please
provide
unique implementations for the following fixture
function(s) in
test/support/fixtures/shopping_cart_fixtures.ex:

    def unique_cart_user_uuid do
```

```
      raise "implement the logic to generate a unique
cart user_uuid"
    end

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

We generated our new context `ShoppingCart`, with a new `ShoppingCart.Cart` schema to tie a user to their cart which holds cart items. We don't have real users yet, so for now our cart will be tracked by an anonymous user UUID that we'll add to our plug session in a moment. With our cart in place, let's generate our cart items:

```
$ mix phx.gen.context ShoppingCart CartItem cart_items \
cart_id:references:carts product_id:references:products \
price_when_carted:decimal quantity:integer

You are generating into an existing context.
...
Would you like to proceed? [Yn] y
* creating lib/hello/shopping_cart/cart_item.ex
* creating
priv/repo/migrations/20210205213410_create_cart_items.exs
* injecting lib/hello/shopping_cart.ex
* injecting test/hello/shopping_cart_test.exs
* injecting
test/support/fixtures/shopping_cart_fixtures.ex

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

We generated a new resource inside our `ShoppingCart` named `CartItem`. This schema and table will hold references to a cart and product, along with the price at the time we added the item to our cart, and the quantity the user wishes to purchase. Let's touch up the generated migration file in `priv/repo/migrations/*_create_cart_items.ex`:

```
    create table(:cart_items) do
-     add :price_when_carted, :decimal
+     add :price_when_carted, :decimal, precision: 15,
scale: 6, null: false
      add :quantity, :integer
-     add :cart_id, references(:carts, on_delete:
:nothing)
+     add :cart_id, references(:carts, on_delete:
:delete_all)
-     add :product_id, references(:products, on_delete:
:nothing)
+     add :product_id, references(:products, on_delete:
:delete_all)

      timestamps()
    end

-   create index(:cart_items, [:cart_id])
    create index(:cart_items, [:product_id])
+   create unique_index(:cart_items, [:cart_id,
:product_id])
```

We used the `:delete_all` strategy again to enforce data integrity. This way, when a cart or product is deleted from the application, we don't have to rely on application code in our `ShoppingCart` or `Catalog` contexts to worry about cleaning up the records. This keeps our application code decoupled and the data integrity enforcement where it belongs – in the database. We also added a unique constraint to ensure a duplicate product is not allowed to be added to a cart. As with the `product_categories` table, using a multi-column index lets us remove the separate index for the leftmost field (`cart_id`). With our database tables in place, we can now migrate up:

```
$ mix ecto.migrate

16:59:51.941 [info] == Running 20210205203342
Hello.Repo.Migrations.CreateCarts.change/0 forward

16:59:51.945 [info] create table carts

16:59:51.949 [info] create index carts_user_uuid_index
```

```
16:59:51.952 [info] == Migrated 20210205203342 in 0.0s

16:59:51.988 [info] == Running 20210205213410
Hello.Repo.Migrations.CreateCartItems.change/0 forward

16:59:51.988 [info] create table cart_items

16:59:51.998 [info] create index cart_items_cart_id_index

16:59:52.000 [info] create index
cart_items_product_id_index

16:59:52.001 [info] create index
cart_items_cart_id_product_id_index

16:59:52.002 [info] == Migrated 20210205213410 in 0.0s
```

Our database is ready to go with new `carts` and `cart_items` tables, but now we need to map that back into application code. You may be wondering how we can mix database foreign keys across different tables and how that relates to the context pattern of isolated, grouped functionality. Let's jump in and discuss the approaches and their tradeoffs.

Cross-context data

So far, we've done a great job isolating the two main contexts of our application from each other, but now we have a necessary dependency to handle.

Our `Catalog.Product` resource serves to keep the responsibilities of representing a product inside the catalog, but ultimately for an item to exist in the cart, a product from the catalog must be present. Given this, our `ShoppingCart` context will have a data dependency on the `Catalog` context. With that in mind, we have two options. One is to expose APIs on the `Catalog` context that allows us to efficiently fetch product data for use in the `ShoppingCart` system, which we would manually stitch together. Or we can use database joins to fetch the dependent data. Both are valid options given your tradeoffs and application size, but joining data from the database

when you have a hard data dependency is just fine for a large class of applications and is the approach we will take here.

Now that we know where our data dependencies exist, let's add our schema associations so we can tie shopping cart items to products. First, let's make a quick change to our cart schema in `lib/hello/shopping_cart/cart.ex` to associate a cart to its items:

```
    schema "carts" do
      field :user_uuid, Ecto.UUID

+     has_many :items, Hello.ShoppingCart.CartItem

      timestamps()
    end
```

Now that our cart is associated to the items we place in it, let's set up the cart item associations inside `lib/hello/shopping_cart/cart_item.ex`:

```
    schema "cart_items" do
      field :price_when_carted, :decimal
      field :quantity, :integer
-     field :cart_id, :id
-     field :product_id, :id

+     belongs_to :cart, Hello.ShoppingCart.Cart
+     belongs_to :product, Hello.Catalog.Product

      timestamps()
    end

    @doc false
    def changeset(cart_item, attrs) do
      cart_item
      |> cast(attrs, [:price_when_carted, :quantity])
      |> validate_required([:price_when_carted, :quantity])
+     |> validate_number(:quantity,
greater_than_or_equal_to: 0, less_than: 100)
    end
```

First, we replaced the `cart_id` field with a standard `belongs_to` pointing at our `ShoppingCart.Cart` schema. Next, we replaced our `product_id` field by adding our first cross-context data dependency with a `belongs_to` for the `Catalog.Product` schema. Here, we intentionally coupled the data boundaries because it provides exactly what we need: an isolated context API with the bare minimum knowledge necessary to reference a product in our system. Next, we added a new validation to our changeset. With `validate_number/3`, we ensure any quantity provided by user input is between 0 and 100.

With our schemas in place, we can start integrating the new data structures and `ShoppingCart` context APIs into our web-facing features.

Adding Shopping Cart functions

As we mentioned before, the context generators are only a starting point for our application. We can and should write well-named, purpose built functions to accomplish the goals of our context. We have a few new features to implement. First, we need to ensure every user of our application is granted a cart if one does not yet exist. From there, we can then allow users to add items to their cart, update item quantities, and calculate cart totals. Let's get started!

We won't focus on a real user authentication system at this point, but by the time we're done, you'll be able to naturally integrate one with what we've written here. To simulate a current user session, open up your `lib/hello_web/router.ex` and key this in:

```
    pipeline :browser do
      plug :accepts, ["html"]
      plug :fetch_session
      plug :fetch_live_flash
      plug :put_root_layout, html: {HelloWeb.LayoutView,
  :root}
      plug :protect_from_forgery
      plug :put_secure_browser_headers
  +   plug :fetch_current_user
```

```
+   plug :fetch_current_cart
  end

+ defp fetch_current_user(conn, _) do
+   if user_uuid = get_session(conn, :current_uuid) do
+     assign(conn, :current_uuid, user_uuid)
+   else
+     new_uuid = Ecto.UUID.generate()
+
+     conn
+     |> assign(:current_uuid, new_uuid)
+     |> put_session(:current_uuid, new_uuid)
+   end
+ end

+ alias Hello.ShoppingCart
+
+ defp fetch_current_cart(conn, _opts) do
+   if cart =
ShoppingCart.get_cart_by_user_uuid(conn.assigns.current_u
uid) do
+     assign(conn, :cart, cart)
+   else
+     {:ok, new_cart} =
ShoppingCart.create_cart(conn.assigns.current_uuid)
+     assign(conn, :cart, new_cart)
+   end
+ end
```

We added a new `:fetch_current_user` and `:fetch_current_cart` plug to our browser pipeline to run on all browser-based requests. Next, we implemented the `fetch_current_user` plug which simply checks the session for a user UUID that was previously added. If we find one, we add a `current_uuid` assign to the connection and we're done. In the case we haven't yet identified this visitor, we generate a unique UUID with `Ecto.UUID.generate()`, then we place that value in the `current_uuid` assign, along with a new session value to identify this visitor on future requests. A random, unique ID isn't much to represent a user, but it's enough for us to track and identify a visitor across requests, which is all we need for now. Later as our application becomes more complete, you'll be ready to migrate to a complete user authentication solution. With a guaranteed current

user, we then implemented the `fetch_current_cart` plug which either finds a cart for the user UUID or creates a cart for the current user and assigns the result in the connection assigns. We'll need to implement our `ShoppingCart.get_cart_by_user_uuid/1` and modify the create cart function to accept a UUID, but let's add our routes first.

We'll need to implement a cart controller for handling cart operations like viewing a cart, updating quantities, and initiating the checkout process, as well as a cart items controller for adding and removing individual items to and from the cart. Add the following routes to your router in `lib/hello_web/router.ex`:

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

      get "/", PageController, :index
      resources "/products", ProductController

+     resources "/cart_items", CartItemController, only:
[:create, :delete]

+     get "/cart", CartController, :show
+     put "/cart", CartController, :update
    end
```

We added a `resources` declaration for a `CartItemController`, which will wire up the routes for a create and delete action for adding and removing individual cart items. Next, we added two new routes pointing at a `CartController`. The first route, a GET request, will map to our show action, to show the cart contents. The second route, a PUT request, will handle the submission of a form for updating our cart quantities.

With our routes in place, let's add the ability to add an item to our cart from the product show page. Create a new file at `lib/hello_web/controllers/cart_item_controller.ex` and key this in:

```
defmodule HelloWeb.CartItemController do
  use HelloWeb, :controller

  alias Hello.ShoppingCart

  def create(conn, %{"product_id" => product_id}) do
    case ShoppingCart.add_item_to_cart(conn.assigns.cart,
product_id) do
      {:ok, _item} ->
        conn
        |> put_flash(:info, "Item added to your cart")
        |> redirect(to: ~p"/cart")

      {:error, _changeset} ->
        conn
        |> put_flash(:error, "There was an error adding
the item to your cart")
        |> redirect(to: ~p"/cart")
    end
  end

  def delete(conn, %{"id" => product_id}) do
    {:ok, _cart} =
ShoppingCart.remove_item_from_cart(conn.assigns.cart,
product_id)
    redirect(conn, to: ~p"/cart")
  end
end
```

We defined a new `CartItemController` with the create and delete actions that we declared in our router. For `create`, we call a `ShoppingCart.add_item_to_cart/2` function which we'll implement in a moment. If successful, we show a flash successful message and redirect to the cart show page; else, we show a flash error message and redirect to the cart show page. For `delete`, we'll call a `remove_item_from_cart` function which we'll implement on our `ShoppingCart` context and then redirect back to the cart show page. We haven't implemented these two shopping cart functions yet, but notice how their names scream their intent: `add_item_to_cart` and `remove_item_from_cart` make it obvious what

we are accomplishing here. It also allows us to spec out our web layer and context APIs without thinking about all the implementation details at once.

Let's implement the new interface for the `ShoppingCart` context API in `lib/hello/shopping_cart.ex`:

```
+  alias Hello.Catalog
-  alias Hello.ShoppingCart.Cart
+  alias Hello.ShoppingCart.{Cart, CartItem}

+  def get_cart_by_user_uuid(user_uuid) do
+    Repo.one(
+      from(c in Cart,
+        where: c.user_uuid == ^user_uuid,
+        left_join: i in assoc(c, :items),
+        left_join: p in assoc(i, :product),
+        order_by: [asc: i.inserted_at],
+        preload: [items: {i, product: p}]
+      )
+    )
+  end

- def create_cart(attrs \\ %{}) do
-   %Cart{}
-   |> Cart.changeset(attrs)
+ def create_cart(user_uuid) do
+   %Cart{user_uuid: user_uuid}
+   |> Cart.changeset(%{})
    |> Repo.insert()
+   |> case do
+     {:ok, cart} -> {:ok, reload_cart(cart)}
+     {:error, changeset} -> {:error, changeset}
+   end
  end

+  defp reload_cart(%Cart{} = cart), do:
get_cart_by_user_uuid(cart.user_uuid)
+
+  def add_item_to_cart(%Cart{} = cart, product_id) do
+    product = Catalog.get_product!(product_id)
+
+    %CartItem{quantity: 1, price_when_carted:
product.price}
```

```
+    |> CartItem.changeset(%{})
+    |> Ecto.Changeset.put_assoc(:cart, cart)
+    |> Ecto.Changeset.put_assoc(:product, product)
+    |> Repo.insert(
+      on_conflict: [inc: [quantity: 1]],
+      conflict_target: [:cart_id, :product_id]
+    )
+  end
+
+  def remove_item_from_cart(%Cart{} = cart, product_id)
do
+    {1, _} =
+      Repo.delete_all(
+        from(i in CartItem,
+          where: i.cart_id == ^cart.id,
+          where: i.product_id == ^product_id
+        )
+      )
+
+    {:ok, reload_cart(cart)}
+  end
```

We started by implementing `get_cart_by_user_uuid/1` which fetches our cart and joins the cart items, and their products so that we have the full cart populated with all preloaded data. Next, we modified our `create_cart` function to accept a user UUID instead of attributes, which we used to populate the `user_uuid` field. If the insert is successful, we reload the cart contents by calling a private `reload_cart/1` function, which simply calls `get_cart_by_user_uuid/1` to refetch data.

Next, we wrote our new `add_item_to_cart/2` function which accepts a cart struct and a product id. We proceed to fetch the product with `Catalog.get_product!/1`, showing how contexts can naturally invoke other contexts if required. You could also have chosen to receive the product as argument and you would achieve similar results. Then we used an upsert operation against our repo to either insert a new cart item into the database, or increase the quantity by one if it already exists in the cart. This is accomplished via the `on_conflict` and `conflict_target` options, which tells our repo how to handle an insert conflict.

Finally, we implemented `remove_item_from_cart/2` where we simply issue a `Repo.delete_all` call with a query to delete the cart item in our cart that matches the product ID. Finally, we reload the cart contents by calling `reload_cart/1`.

With our new cart functions in place, we can now expose the "Add to cart" button on the product catalog show page. Open up your template in `lib/hello_web/controllers/product_html/show.html.heex` and make the following changes:

```
...
      <.link href={~p"/products/#{@product}/edit"}>
        <.button>Edit product</.button>
      </.link>
+     <.link href={~p"/cart_items?product_id=#
{@product.id}"} method="post">
+       <.button>Add to cart</.button>
+     </.link>
...
```

The `link` function component from `Phoenix.Component` accepts a `:method` attribute to issue an HTTP verb when clicked, instead of the default GET request. With this link in place, the "Add to cart" link will issue a POST request, which will be matched by the route we defined in router which dispatches to the `CartItemController.create/2` function.

Let's try it out. Start your server with `mix phx.server` and visit a product page. If we try clicking the add to cart link, we'll be greeted by an error page with the following logs in the console:

```
[info] POST /cart_items
[debug] Processing with
HelloWeb.CartItemController.create/2
  Parameters: %{"_method" => "post", "product_id" => "1",
...}
  Pipelines: [:browser]
INSERT INTO "cart_items" ...
[info] Sent 302 in 24ms
```

```
[info] GET /cart
[debug] Processing with HelloWeb.CartController.show/2
  Parameters: %{}
  Pipelines: [:browser]
[debug] QUERY OK source="carts" db=1.9ms idle=1798.5ms

[error] #PID<0.856.0> running HelloWeb.Endpoint
(connection #PID<0.841.0>, stream id 5) terminated
Server: localhost:4000 (http)
Request: GET /cart
** (exit) an exception was raised:
    ** (UndefinedFunctionError) function
HelloWeb.CartController.init/1 is undefined
       (module HelloWeb.CartController is not available)
       ...
```

It's working! Kind of. If we follow the logs, we see our POST to the `/cart_items` path. Next, we can see our `ShoppingCart.add_item_to_cart` function successfully inserted a row into the `cart_items` table, and then we issued a redirect to `/cart`. Before our error, we also see a query to the `carts` table, which means we're fetching the current user's cart. So far so good. We know our `CartItem` controller and new `ShoppingCart` context functions are doing their jobs, but we've hit our next unimplemented feature when the router attempts to dispatch to a nonexistent cart controller. Let's create the cart controller, view, and template to display and manage user carts.

Create a new file at `lib/hello_web/controllers/cart_controller.ex` and key this in:

```
defmodule HelloWeb.CartController do
  use HelloWeb, :controller

  alias Hello.ShoppingCart

  def show(conn, _params) do
    render(conn, :show, changeset:
ShoppingCart.change_cart(conn.assigns.cart))
  end
end
```

We defined a new cart controller to handle the `get "/cart"` route. For showing a cart, we render a `"show.html"` template which we'll create in a moment. We know we need to allow the cart items to be changed by quantity updates, so right away we know we'll need a cart changeset. Fortunately, the context generator included a `ShoppingCart.change_cart/1` function, which we'll use. We pass it our cart struct which is already in the connection assigns thanks to the `fetch_current_cart` plug we defined in the router.

Next, we can implement the view and template. Create a new view file at `lib/hello_web/controllers/cart_html.ex` with the following content:

```
defmodule HelloWeb.CartHTML do
  use HelloWeb, :html

  alias Hello.ShoppingCart

  embed_templates "cart_html/*"

  def currency_to_str(%Decimal{} = val), do: "$#
{Decimal.round(val, 2)}"
end
```

We created a view to render our `show.html` template and aliased our `ShoppingCart` context so it will be in scope for our template. We'll need to display the cart prices like product item price, cart total, etc, so we defined a `currency_to_str/1` which takes our decimal struct, rounds it properly for display, and prepends a USD dollar sign.

Next we can create the template at `lib/hello_web/controllers/cart_html/show.html.heex`:

```
<.header>
  My Cart
  <:subtitle :if={@cart.items == []}>Your cart is
empty</:subtitle>
</.header>
```

```
<div :if={@cart.items !== []}>
  <.simple_form :let={f} for={@changeset} action=
{~p"/cart"}>
    <.inputs_for :let={%{data: item} = item_form} field=
{f[:items]}>
      <.input field={item_form[:quantity]} type="number"
label={item.product.title} />

{currency_to_str(ShoppingCart.total_item_price(item))}
    </.inputs_for>
    <:actions>
      <.button>Update cart</.button>
    </:actions>
  </.simple_form>
  <b>Total</b>:
{currency_to_str(ShoppingCart.total_cart_price(@cart))}
</div>

<.back navigate={~p"/products"}>Back to products</.back>
```

We started by showing the empty cart message if our preloaded `cart.items` is empty. If we have items, we use the `simple_form` component provided by our `HelloWeb.CoreComponents` to take our cart changeset that we assigned in the `CartController.show/2` action and create a form which maps to our cart controller `update/2` action. Within the form, we use the `inputs_for` component to render inputs for the nested cart items. This will allow us to map item inputs back together when the form is submitted. Next, we display a number input for the item quantity and label it with the product title. We finish the item form by converting the item price to string. We haven't written the `ShoppingCart.total_item_price/1` function yet, but again we employed the idea of clear, descriptive public interfaces for our contexts. After rendering inputs for all the cart items, we show an "update cart" submit button, along with the total price of the entire cart. This is accomplished with another new `ShoppingCart.total_cart_price/1` function which we'll implement in a moment. Finally, we added a `back` component to go back to our products page.

We're almost ready to try out our cart page, but first we need to implement our new currency calculation functions. Open up your shopping cart context at `lib/hello/shopping_cart.ex` and add these new functions:

```
def total_item_price(%CartItem{} = item) do
  Decimal.mult(item.product.price, item.quantity)
end

def total_cart_price(%Cart{} = cart) do
  Enum.reduce(cart.items, 0, fn item, acc ->
    item
    |> total_item_price()
    |> Decimal.add(acc)
  end)
end
```

We implemented `total_item_price/1` which accepts a `%CartItem{}` struct. To calculate the total price, we simply take the preloaded product's price and multiply it by the item's quantity. We used `Decimal.mult/2` to take our decimal currency struct and multiply it with proper precision. Similarly for calculating the total cart price, we implemented a `total_cart_price/1` function which accepts the cart and sums the preloaded product prices for items in the cart. We again make use of the `Decimal` functions to add our decimal structs together.

Now that we can calculate price totals, let's try it out! Visit `http://localhost:4000/cart` and you should already see your first item in the cart. Going back to the same product and clicking "add to cart" will show our upsert in action. Your quantity should now be two. Nice work!

Our cart page is almost complete, but submitting the form will yield yet another error.

```
Request: POST /cart
** (exit) an exception was raised:
    ** (UndefinedFunctionError) function
HelloWeb.CartController.update/2 is undefined or private
```

Let's head back to our `CartController` at `lib/hello_web/controllers/cart_controller.ex` and implement the update action:

```
  def update(conn, %{"cart" => cart_params}) do
    case ShoppingCart.update_cart(conn.assigns.cart,
cart_params) do
      {:ok, _cart} ->
        redirect(conn, to: ~p"/cart")

      {:error, _changeset} ->
        conn
        |> put_flash(:error, "There was an error updating
your cart")
        |> redirect(to: ~p"/cart")
    end
  end
```

We started by plucking out the cart params from the form submit. Next, we call our existing `ShoppingCart.update_cart/2` function which was added by the context generator. We'll need to make some changes to this function, but the interface is good as is. If the update is successful, we redirect back to the cart page, otherwise we show a flash error message and send the user back to the cart page to fix any mistakes. Out-of-the-box, our `ShoppingCart.update_cart/2` function only concerned itself with casting the cart params into a changeset and updates it against our repo. For our purposes, we now need it to handle nested cart item associations, and most importantly, business logic for how to handle quantity updates like zero-quantity items being removed from the cart.

Head back over to your shopping cart context in `lib/hello/shopping_cart.ex` and replace your `update_cart/2` function with the following implementation:

```
  def update_cart(%Cart{} = cart, attrs) do
    changeset =
      cart
      |> Cart.changeset(attrs)
```

```
      |> Ecto.Changeset.cast_assoc(:items, with:
&CartItem.changeset/2)

    Ecto.Multi.new()
    |> Ecto.Multi.update(:cart, changeset)
    |> Ecto.Multi.delete_all(:discarded_items, fn %{cart:
cart} ->
      from(i in CartItem, where: i.cart_id == ^cart.id
and i.quantity == 0)
    end)
    |> Repo.transaction()
    |> case do
      {:ok, %{cart: cart}} -> {:ok, cart}
      {:error, :cart, changeset, _changes_so_far} ->
{:error, changeset}
    end
  end
```

We started much like how our out-of-the-box code started – we take the cart struct and cast the user input to a cart changeset, except this time we use `Ecto.Changeset.cast_assoc/3` to cast the nested item data into `CartItem` changesets. Remember the `<.inputs_for />` call in our cart form template? That hidden ID data is what allows Ecto's `cast_assoc` to map item data back to existing item associations in the cart. Next we use `Ecto.Multi.new/0`, which you may not have seen before. Ecto's `Multi` is a feature that allows lazily defining a chain of named operations to eventually execute inside a database transaction. Each operation in the multi chain receives the values from the previous steps and executes until a failed step is encountered. When an operation fails, the transaction is rolled back and an error is returned, otherwise the transaction is committed.

For our multi operations, we start by issuing an update of our cart, which we named `:cart`. After the cart update is issued, we perform a multi `delete_all` operation, which takes the updated cart and applies our zero-quantity logic. We prune any items in the cart with zero quantity by returning an ecto query that finds all cart items for this cart with an empty quantity. Calling `Repo.transaction/1` with our multi will execute the operations in a new transaction and we return the success or failure result to the caller just like the original function.

Let's head back to the browser and try it out. Add a few products to your cart, update the quantities, and watch the values changes along with the price calculations. Setting any quantity to 0 will also remove the item. Pretty neat!

Adding an Orders context

With our `Catalog` and `ShoppingCart` contexts, we're seeing first-hand how our well-considered modules and function names are yielding clear and maintainable code. Our last order of business is to allow the user to initiate the checkout process. We won't go as far as integrating payment processing or order fulfillment, but we'll get you started in that direction. Like before, we need to decide where code for completing an order should live. Is it part of the catalog? Clearly not, but what about the shopping cart? Shopping carts are related to orders – after all, the user has to add items in order to purchase any products – but should the order checkout process be grouped here?

If we stop and consider the order process, we'll see that orders involve related, but distinctly different data from the cart contents. Also, business rules around the checkout process are much different than carting. For example, we may allow a user to add a back-ordered item to their cart, but we could not allow an order with no inventory to be completed. Additionally, we need to capture point-in-time product information when an order is completed, such as the price of the items *at payment transaction time*. This is essential because a product price may change in the future, but the line items in our order must always record and display what we charged at time of purchase. For these reasons, we can start to see that ordering can stand on its own with its own data concerns and business rules.

Naming wise, `Orders` clearly defines the scope of our context, so let's get started by again taking advantage of the context generators. Run the following command in your console:

```
$ mix phx.gen.context Orders Order orders user_uuid:uuid
total_price:decimal

* creating lib/hello/orders/order.ex
```

```
* creating
priv/repo/migrations/20210209214612_create_orders.exs
* creating lib/hello/orders.ex
* injecting lib/hello/orders.ex
* creating test/hello/orders_test.exs
* injecting test/hello/orders_test.exs
* creating test/support/fixtures/orders_fixtures.ex
* injecting test/support/fixtures/orders_fixtures.ex

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

We generated an `Orders` context. We added a `user_uuid` field to associate our placeholder current user to an order, along with a `total_price` column. With our starting point in place, let's open up the newly created migration in `priv/repo/migrations/*_create_orders.exs` and make the following changes:

```
  def change do
    create table(:orders) do
      add :user_uuid, :uuid
-     add :total_price, :decimal
+     add :total_price, :decimal, precision: 15, scale:
6, null: false

      timestamps()
    end
  end
```

Like we did previously, we gave appropriate precision and scale options for our decimal column which will allow us to store currency without precision loss. We also added a not-null constraint to enforce all orders to have a price.

The orders table alone doesn't hold much information, but we know we'll need to store point-in-time product price information of all the items in the order. For that, we'll add an additional struct for this context named

`LineItem`. Line items will capture the price of the product *at payment transaction time*. Please run the following command:

```
$ mix phx.gen.context Orders LineItem order_line_items \
price:decimal quantity:integer \
order_id:references:orders product_id:references:products

You are generating into an existing context.
...
Would you like to proceed? [Yn] y
* creating lib/hello/orders/line_item.ex
* creating
priv/repo/migrations/20210209215050_create_order_line_ite
ms.exs
* injecting lib/hello/orders.ex
* injecting test/hello/orders_test.exs
* injecting test/support/fixtures/orders_fixtures.ex

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

We used the `phx.gen.context` command to generate the `LineItem` Ecto schema and inject supporting functions into our orders context. Like before, let's modify the migration in `priv/repo/migrations/*_create_order_line_items.exs` and make the following decimal field changes:

```
  def change do
    create table(:order_line_items) do
-     add :price, :decimal
+     add :price, :decimal, precision: 15, scale: 6,
null: false
      add :quantity, :integer
      add :order_id, references(:orders, on_delete:
:nothing)
      add :product_id, references(:products, on_delete:
:nothing)

      timestamps()
```

```
    end

    create index(:order_line_items, [:order_id])
    create index(:order_line_items, [:product_id])
  end
```

With our migration in place, let's wire up our orders and line items associations in `lib/hello/orders/order.ex`:

```
  schema "orders" do
    field :total_price, :decimal
    field :user_uuid, Ecto.UUID

+   has_many :line_items, Hello.Orders.LineItem
+   has_many :products, through: [:line_items, :product]

    timestamps()
  end
```

We used `has_many :line_items` to associate orders and line items, just like we've seen before. Next, we used the `:through` feature of `has_many`, which allows us to instruct ecto how to associate resources across another relationship. In this case, we can associate products of an order by finding all products through associated line items. Next, let's wire up the association in the other direction in `lib/hello/orders/line_item.ex`:

```
  schema "order_line_items" do
    field :price, :decimal
    field :quantity, :integer
-   field :order_id, :id
-   field :product_id, :id

+   belongs_to :order, Hello.Orders.Order
+   belongs_to :product, Hello.Catalog.Product

    timestamps()
  end
```

We used `belongs_to` to associate line items to orders and products. With our associations in place, we can start integrating the web interface into our order process. Open up your router `lib/hello_web/router.ex` and add the following line:

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

      ...
+     resources "/orders", OrderController, only: [:create,
:show]
    end
```

We wired up `create` and `show` routes for our generated `OrderController`, since these are the only actions we need at the moment. With our routes in place, we can now migrate up:

```
$ mix ecto.migrate

17:14:37.715 [info] == Running 20210209214612
Hello.Repo.Migrations.CreateOrders.change/0 forward

17:14:37.720 [info] create table orders

17:14:37.755 [info] == Migrated 20210209214612 in 0.0s

17:14:37.784 [info] == Running 20210209215050
Hello.Repo.Migrations.CreateOrderLineItems.change/0
forward

17:14:37.785 [info] create table order_line_items

17:14:37.795 [info] create index
order_line_items_order_id_index

17:14:37.796 [info] create index
order_line_items_product_id_index

17:14:37.798 [info] == Migrated 20210209215050 in 0.0s
```

Before we render information about our orders, we need to ensure our order data is fully populated and can be looked up by a current user. Open up your orders context in `lib/hello/orders.ex` and replace your `get_order!/1` function by a new `get_order!/2` definition:

```
  def get_order!(user_uuid, id) do
    Order
    |> Repo.get_by!(id: id, user_uuid: user_uuid)
    |> Repo.preload([line_items: [:product]])
  end
```

We rewrote the function to accept a user UUID and query our repo for an order matching the user's ID for a given order ID. Then we populated the order by preloading our line item and product associations.

To complete an order, our cart page can issue a POST to the `OrderController.create` action, but we need to implement the operations and logic to actually complete an order. Like before, we'll start at the web interface. Create a new file at `lib/hello_web/controllers/order_controller.ex` and key this in:

```
defmodule HelloWeb.OrderController do
  use HelloWeb, :controller

  alias Hello.Orders

  def create(conn, _) do
    case Orders.complete_order(conn.assigns.cart) do
      {:ok, order} ->
        conn
        |> put_flash(:info, "Order created
successfully.")
        |> redirect(to: ~p"/orders/#{order}")

      {:error, _reason} ->
        conn
        |> put_flash(:error, "There was an error
processing your order")
        |> redirect(to: ~p"/cart")
```

```
    end
  end
end
```

We wrote the `create` action to call an as-yet-implemented `Orders.complete_order/1` function. Our code is technically "creating" an order, but it's important to step back and consider the naming of your interfaces. The act of *completing* an order is extremely important in our system. Money changes hands in a transaction, physical goods could be automatically shipped, etc. Such an operation deserves a better, more obvious function name, such as `complete_order`. If the order is completed successfully we redirect to the show page, otherwise a flash error is shown as we redirect back to the cart page.

Here is also a good opportunity to highlight that contexts can naturally work with data defined by other contexts too. This will be especially common with data that is used throughout the application, such as the cart here (but it can also be the current user or the current project, and so forth, depending on your project).

Now we can implement our `Orders.complete_order/1` function. To complete an order, our job will require a few operations:

1. A new order record must be persisted with the total price of the order
2. All items in the cart must be transformed into new order line items records with quantity and point-in-time product price information
3. After successful order insert (and eventual payment), items must be pruned from the cart

From our requirements alone, we can start to see why a generic `create_order` function doesn't cut it. Let's implement this new function in `lib/hello/orders.ex`:

```
  alias Hello.Orders.LineItem
  alias Hello.ShoppingCart

  def complete_order(%ShoppingCart.Cart{} = cart) do
```

```
    line_items =
      Enum.map(cart.items, fn item ->
        %{product_id: item.product_id, price:
item.product.price, quantity: item.quantity}
      end)

    order =
      Ecto.Changeset.change(%Order{},
        user_uuid: cart.user_uuid,
        total_price: ShoppingCart.total_cart_price(cart),
        line_items: line_items
      )

    Ecto.Multi.new()
    |> Ecto.Multi.insert(:order, order)
    |> Ecto.Multi.run(:prune_cart, fn _repo, _changes ->
      ShoppingCart.prune_cart_items(cart)
    end)
    |> Repo.transaction()
    |> case do
      {:ok, %{order: order}} -> {:ok, order}
      {:error, name, value, _changes_so_far} -> {:error,
{name, value}}
    end
  end
```

We started by mapping the `%ShoppingCart.CartItem{}`'s in our shopping cart into a map of order line items structs. The job of the order line item record is to capture the price of the product *at payment transaction time*, so we reference the product's price here. Next, we create a bare order changeset with `Ecto.Changeset.change/2` and associate our user UUID, set our total price calculation, and place our order line items in the changeset. With a fresh order changeset ready to be inserted, we can again make use of `Ecto.Multi` to execute our operations in a database transaction. We start by inserting the order, followed by a `run` operation. The `Ecto.Multi.run/3` function allows us to run any code in the function which must either succeed with `{:ok, result}` or error, which halts and rolls back the transaction. Here, we simply call into our shopping cart context and ask it to prune all items in a cart. Running the transaction will execute the multi as before and we return the result to the caller.

To close out our order completion, we need to implement the `ShoppingCart.prune_cart_items/1` function in `lib/hello/shopping_cart.ex`:

```
  def prune_cart_items(%Cart{} = cart) do
    {_, _} = Repo.delete_all(from(i in CartItem, where:
i.cart_id == ^cart.id))
    {:ok, reload_cart(cart)}
  end
```

Our new function accepts the cart struct and issues a `Repo.delete_all` which accepts a query of all items for the provided cart. We return a success result by simply reloading the pruned cart to the caller. With our context complete, we now need to show the user their completed order. Head back to your order controller and add the `show/2` action:

```
  def show(conn, %{"id" => id}) do
    order = Orders.get_order!(conn.assigns.current_uuid,
id)
    render(conn, :show, order: order)
  end
```

We added the show action to pass our `conn.assigns.current_uuid` to `get_order!` which authorizes orders to be viewable only by the owner of the order. Next, we can implement the view and template. Create a new view file at `lib/hello_web/controllers/order_html.ex` with the following content:

```
defmodule HelloWeb.OrderHTML do
  use HelloWeb, :html

  embed_templates "order_html/*"
end
```

Next we can create the template at `lib/hello_web/controllers/order_html/show.html.heex`:

```
<.header>
  Thank you for your order!
  <:subtitle>
     <strong>User uuid: </strong>{@order.user_uuid}
  </:subtitle>
</.header>

<.table id="items" rows={@order.line_items}>
  <:col :let={item} label="Title">{item.product.title}
</:col>
  <:col :let={item} label="Quantity">{item.quantity}
</:col>
  <:col :let={item} label="Price">
    {HelloWeb.CartHTML.currency_to_str(item.price)}
  </:col>
</.table>

<strong>Total price:</strong>
{HelloWeb.CartHTML.currency_to_str(@order.total_price)}

<.back navigate={~p"/products"}>Back to products</.back>
```

To show our completed order, we displayed the order's user, followed by the line item listing with product title, quantity, and the price we "transacted" when completing the order, along with the total price.

Our last addition will be to add the "complete order" button to our cart page to allow completing an order. Add the following button to the <.header> of the cart show template in `lib/hello_web/controllers/cart_html/show.html.heex`:

```
   <.header>
     My Cart
+    <:actions>
+      <.link href={~p"/orders"} method="post">
+        <.button>Complete order</.button>
+      </.link>
+    </:actions>
   </.header>
```

We added a link with `method="post"` to send a POST request to our `OrderController.create` action. If we head back to our cart page at http://localhost:4000/cart and complete an order, we'll be greeted by our rendered template:

```
Thank you for your order!

User uuid: 08964c7c-908c-4a55-bcd3-9811ad8b0b9d
Title                   Quantity Price
Metaprogramming Elixir  2        $15.00

Total price: $30.00
```

Nice work! We haven't added payments, but we can already see how our `ShoppingCart` and `Orders` context splitting is driving us towards a maintainable solution. With our cart items separated from our order line items, we are well equipped in the future to add payment transactions, cart price detection, and more.

Great work!

FAQ

When to use code generators?

In this guide, we have used code generators for schemas, contexts, controllers, and more. If you are happy to move forward with Phoenix defaults, feel free to rely on generators to scaffold large parts of your application. When using Phoenix generators, the main question you need to answer is: does this new functionality (with its schema, table, and fields) belong to one of the existing contexts or a new one?

This way, Phoenix generators guide you to use contexts to group related functionality, instead of having several dozens of schemas laying around without any structure. And remember: if you're stuck when trying to come up

with a context name, you can simply use the plural form of the resource you're creating.

How do I structure code inside contexts?

You may wonder how to organize the code inside contexts. For example, should you define a module for changesets (such as ProductChangesets) and another module for queries (such as ProductQueries)?

One important benefit of contexts is that this decision does not matter much. The context is your public API, the other modules are private. Contexts isolate these modules into small groups so the surface area of your application is the context and not *all of your code*.

So while you and your team could establish patterns for organizing these private modules, it is also our opinion it is completely fine for them to be different. The major focus should be on how the contexts are defined and how they interact with each other (and with your web application).

Think about it as a well-kept neighbourhood. Your contexts are houses, you want to keep them well-preserved, well-connected, etc. Inside the houses, they may all be a little bit different, and that's fine.

Returning Ecto structures from context APIs

As we explored the context API, you might have wondered:

> If one of the goals of our context is to encapsulate Ecto Repo access, why does `create_user/1` return an `Ecto.Changeset` struct when we fail to create a user?

Although Changesets are part of Ecto, they are not tied to the database, and they can be used to map data from and to any source, which makes it a general and useful data structure for tracking field changes, perform validations, and generate error messages.

For those reasons, `%Ecto.Changeset{}` is a good choice to model the data changes between your contexts and your web layer - regardless if you are talking to an API or the database.

Finally, note that your controllers and views are not hardcoded to work exclusively with Ecto either. Instead, Phoenix defines protocols such as `Phoenix.Param` and `Phoenix.HTML.FormData`, which allow any library to extend how Phoenix generates URL parameters or renders forms. Conveniently for us, the `phoenix_ecto` project implements those protocols, but you could as well bring your own data structures and implement them yourself.

JSON and APIs

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Controllers guide.

You can also use the Phoenix Framework to build Web APIs. By default Phoenix supports JSON but you can bring any other rendering format you desire.

The JSON API

For this guide let's create a simple JSON API to store our favourite links, that will support all the CRUD (Create, Read, Update, Delete) operations out of the box.

For this guide, we will use Phoenix generators to scaffold our API infrastructure:

```
mix phx.gen.json Urls Url urls link:string title:string
* creating lib/hello_web/controllers/url_controller.ex
* creating lib/hello_web/controllers/url_json.ex
* creating lib/hello_web/controllers/changeset_json.ex
* creating
test/hello_web/controllers/url_controller_test.exs
* creating
lib/hello_web/controllers/fallback_controller.ex
* creating lib/hello/urls/url.ex
* creating
priv/repo/migrations/20221129120234_create_urls.exs
* creating lib/hello/urls.ex
* injecting lib/hello/urls.ex
* creating test/hello/urls_test.exs
* injecting test/hello/urls_test.exs
```

```
* creating test/support/fixtures/urls_fixtures.ex
* injecting test/support/fixtures/urls_fixtures.ex
```

We will break those files into four categories:

- Files in `lib/hello_web` responsible for effectively rendering JSON
- Files in `lib/hello` responsible for defining our context and logic to persist links to the database
- Files in `priv/repo/migrations` responsible for updating our database
- Files in `test` to test our controllers and contexts

In this guide, we will explore only the first category of files. To learn more about how Phoenix stores and manage data, check out the Ecto guide and the Contexts guide for more information. We also have a whole section dedicated to testing.

At the end, the generator asks us to add the `/url` resource to our `:api` scope in `lib/hello_web/router.ex`:

```
scope "/api", HelloWeb do
  pipe_through :api
  resources "/urls", UrlController, except: [:new, :edit]
end
```

The API scope uses the `:api` pipeline, which will run specific steps such as ensuring the client can handle JSON responses.

Then we need to update our repository by running migrations:

```
mix ecto.migrate
```

Trying out the JSON API

Before we go ahead and change those files, let's take a look at how our API behaves from the command line.

First, we need to start the server:

```
mix phx.server
```

Next, let's make a smoke test to check our API is working with:

```
curl -i http://localhost:4000/api/urls
```

If everything went as planned we should get a `200` response:

```
HTTP/1.1 200 OK
cache-control: max-age=0, private, must-revalidate
content-length: 11
content-type: application/json; charset=utf-8
date: Fri, 06 May 2022 21:22:42 GMT
server: Cowboy
x-request-id: Fuyg-wMl4S-hAfsAAAUk

{"data":[]}
```

We didn't get any data because we haven't populated the database with any yet. So let's add some links:

```
curl -iX POST http://localhost:4000/api/urls \
   -H 'Content-Type: application/json' \
   -d '{"url": {"link":"https://phoenixframework.org",
"title":"Phoenix Framework"}}'

curl -iX POST http://localhost:4000/api/urls \
   -H 'Content-Type: application/json' \
   -d '{"url": {"link":"https://elixir-lang.org",
"title":"Elixir"}}'
```

Now we can retrieve all links:

```
curl -i http://localhost:4000/api/urls
```

Or we can just retrieve a link by its `id`:

```
curl -i http://localhost:4000/api/urls/1
```

Next, we can update a link with:

```
curl -iX PUT http://localhost:4000/api/urls/2 \
   -H 'Content-Type: application/json' \
   -d '{"url": {"title":"Elixir Programming Language"}}'
```

The response should be a `200` with the updated link in the body.

Finally, we need to try out the removal of a link:

```
curl -iX DELETE http://localhost:4000/api/urls/2 \
   -H 'Content-Type: application/json'
```

A `204` response should be returned to indicate the successful removal of the link.

Rendering JSON

To understand how to render JSON, let's start with the `index` action from `UrlController` defined at `lib/hello_web/controllers/url_controller.ex`:

```
  def index(conn, _params) do
    urls = Urls.list_urls()
```

```
    render(conn, :index, urls: urls)
  end
```

As we can see, this is not any different from how Phoenix renders HTML templates. We call `render/3`, passing the connection, the template we want our views to render (`:index`), and the data we want to make available to our views.

Phoenix typically uses one view per rendering format. When rendering HTML, we would use `UrlHTML`. Now that we are rendering JSON, we will find a `UrlJSON` view collocated with the template at `lib/hello_web/controllers/url_json.ex`. Let's open it up:

```
defmodule HelloWeb.UrlJSON do
  alias Hello.Urls.Url

  @doc """
  Renders a list of urls.
  """
  def index(%{urls: urls}) do
    %{data: for(url <- urls, do: data(url))}
  end

  @doc """
  Renders a single url.
  """
  def show(%{url: url}) do
    %{data: data(url)}
  end

  defp data(%Url{} = url) do
    %{
      id: url.id,
      link: url.link,
      title: url.title
    }
  end
end
```

This view is very simple. The `index` function receives all URLs, and converts them into a list of maps. Those maps are placed inside the data key at the root, exactly as we saw when interfacing with our application from `cURL`. In other words, our JSON view converts our complex data into simple Elixir data-structures. Once our view layer returns, Phoenix uses the `Jason` library to encode JSON and send the response to the client.

If you explore the remaining the controller, you will learn the `show` action is similar to the `index` one. For `create`, `update`, and `delete` actions, Phoenix uses one other important feature, called "Action fallback".

Action fallback

Action fallback allows us to centralize error handling code in plugs, which are called when a controller action fails to return a `%Plug.Conn{}` struct. These plugs receive both the `conn` which was originally passed to the controller action along with the return value of the action.

Let's say we have a `show` action which uses `with` to fetch a blog post and then authorize the current user to view that blog post. In this example we might expect `fetch_post/1` to return `{:error, :not_found}` if the post is not found and `authorize_user/3` might return `{:error, :unauthorized}` if the user is unauthorized. We could use our `ErrorHTML` and `ErrorJSON` views which are generated by Phoenix for every new application to handle these error paths accordingly:

```
defmodule HelloWeb.MyController do
  use Phoenix.Controller

  def show(conn, %{"id" => id}, current_user) do
    with {:ok, post} <- fetch_post(id),
         :ok <- authorize_user(current_user, :view, post)
do
      render(conn, :show, post: post)
    else
      {:error, :not_found} ->
        conn
```

```
        |> put_status(:not_found)
        |> put_view(html: HelloWeb.ErrorHTML, json:
HelloWeb.ErrorJSON)
        |> render(:"404")

      {:error, :unauthorized} ->
        conn
        |> put_status(403)
        |> put_view(html: HelloWeb.ErrorHTML, json:
HelloWeb.ErrorJSON)
        |> render(:"403")
    end
  end
end
```

Now imagine you may need to implement similar logic for every controller and action handled by your API. This would result in a lot of repetition.

Instead we can define a module plug which knows how to handle these error cases specifically. Since controllers are module plugs, let's define our plug as a controller:

```
defmodule HelloWeb.MyFallbackController do
  use Phoenix.Controller

  def call(conn, {:error, :not_found}) do
    conn
    |> put_status(:not_found)
    |> put_view(json: HelloWeb.ErrorJSON)
    |> render(:"404")
  end

  def call(conn, {:error, :unauthorized}) do
    conn
    |> put_status(403)
    |> put_view(json: HelloWeb.ErrorJSON)
    |> render(:"403")
  end
end
```

Then we can reference our new controller as the `action_fallback` and simply remove the `else` block from our `with`:

```
defmodule HelloWeb.MyController do
  use Phoenix.Controller

  action_fallback HelloWeb.MyFallbackController

  def show(conn, %{"id" => id}, current_user) do
    with {:ok, post} <- fetch_post(id),
         :ok <- authorize_user(current_user, :view, post)
do
      render(conn, :show, post: post)
    end
  end
end
```

Whenever the `with` conditions do not match, `HelloWeb.MyFallbackController` will receive the original `conn` as well as the result of the action and respond accordingly.

FallbackController and ChangesetJSON

With this knowledge in hand, we can explore the `FallbackController` (`lib/hello_web/controllers/fallback_controller.ex`) generated by `mix phx.gen.json`. In particular, it handles one clause (the other is generated as an example):

```
  def call(conn, {:error, %Ecto.Changeset{} = changeset})
do
    conn
    |> put_status(:unprocessable_entity)
    |> put_view(json: HelloWeb.ChangesetJSON)
    |> render(:error, changeset: changeset)
  end
```

The goal of this clause is to handle the `{:error, changeset}` return types from the `HelloWeb.Urls` context and render them into rendered errors via the `ChangesetJSON` view. Let's open up `lib/hello_web/controllers/changeset_json.ex` to learn more:

```
defmodule HelloWeb.ChangesetJSON do
  @doc """
  Renders changeset errors.
  """
  def error(%{changeset: changeset}) do
    # When encoded, the changeset returns its errors
    # as a JSON object. So we just pass it forward.
    %{errors: Ecto.Changeset.traverse_errors(changeset,
&translate_error/1)}
  end
end
```

As we can see, it will convert the errors into a data structure, which will be rendered as JSON. The changeset is a data structure responsible for casting and validating data. For our example, it is defined in `Hello.Urls.Url.changeset/1`. Let's open up `lib/hello/urls/url.ex` and see its definition:

```
  @doc false
  def changeset(url, attrs) do
    url
    |> cast(attrs, [:link, :title])
    |> validate_required([:link, :title])
  end
```

As you can see, the changeset requires both link and title to be given. This means we can try posting a url with no link and title and see how our API responds:

```
curl -iX POST http://localhost:4000/api/urls \
   -H 'Content-Type: application/json' \
   -d '{"url": {}}'
```

```
{"errors": {"link": ["can't be blank"], "title": ["can't
be blank"]}}
```

Feel free to modify the `changeset` function and see how your API behaves.

API-only applications

In case you want to generate a Phoenix application exclusively for APIs, you can pass several options when invoking `mix phx.new`. Let's check which `--no-*` flags we need to use to not generate the scaffolding that isn't necessary on our Phoenix application for the REST API.

From your terminal run:

```
mix help phx.new
```

The output should contain the following:

```
  • --no-assets - equivalent to --no-esbuild and --no-
tailwind
  • --no-dashboard - do not include Phoenix.LiveDashboard
  • --no-ecto - do not generate Ecto files
  • --no-esbuild - do not include esbuild dependencies
and
    assets. We do not recommend setting this option,
unless for API
    only applications, as doing so requires you to
manually add and
    track JavaScript dependencies
  • --no-gettext - do not generate gettext files
  • --no-html - do not generate HTML views
  • --no-live - comment out LiveView socket setup in your
Endpoint
    and assets/js/app.js. Automatically disabled if --no-
html is given
  • --no-mailer - do not generate Swoosh mailer files
  • --no-tailwind - do not include tailwind dependencies
```

```
and
    assets. The generated markup will still include
Tailwind CSS
    classes, those are left-in as reference for the
subsequent
    styling of your layout and components
```

The `--no-html` is the obvious one we want to use when creating any Phoenix application for an API in order to leave out all the unnecessary HTML scaffolding. You may also pass `--no-assets`, if you don't want any of the asset management bit, `--no-gettext` if you don't support internationalization, and so on.

Also bear in mind that nothing stops you to have a backend that supports simultaneously the REST API and a Web App (HTML, assets, internationalization and sockets).

Mix tasks

There are currently a number of built-in Phoenix-specific and Ecto-specific Mix tasks available to us within a newly-generated application. We can also create our own application specific tasks.

> Note to learn more about `mix`, you can read Elixir's official Introduction to Mix.

Phoenix tasks

```
$ mix help --search "phx"
mix local.phx           # Updates the Phoenix project
generator locally
mix phx                 # Prints Phoenix help information
mix phx.digest          # Digests and compresses static
files
mix phx.digest.clean    # Removes old versions of static
assets.
mix phx.gen.auth        # Generates authentication logic
for a resource
mix phx.gen.cert        # Generates a self-signed
certificate for HTTPS testing
mix phx.gen.channel     # Generates a Phoenix channel
mix phx.gen.context     # Generates a context with
functions around an Ecto schema
mix phx.gen.embedded    # Generates an embedded Ecto
schema file
mix phx.gen.html        # Generates controller, views, and
context for an HTML resource
mix phx.gen.json        # Generates controller, views, and
context for a JSON resource
mix phx.gen.live        # Generates LiveView, templates,
and context for a resource
mix phx.gen.notifier    # Generates a notifier that
delivers emails by default
mix phx.gen.presence    # Generates a Presence tracker
mix phx.gen.schema      # Generates an Ecto schema and
```

```
migration file
mix phx.gen.secret      # Generates a secret
mix phx.gen.socket      # Generates a Phoenix socket
handler
mix phx.new             # Creates a new Phoenix
application
mix phx.new.ecto        # Creates a new Ecto project
within an umbrella project
mix phx.new.web         # Creates a new Phoenix web
project within an umbrella project
mix phx.routes          # Prints all routes
mix phx.server          # Starts applications and their
servers
```

We have seen all of these at one point or another in the guides, but having all the information about them in one place seems like a good idea.

We will cover all Phoenix Mix tasks, except `phx.new`, `phx.new.ecto`, and `phx.new.web`, which are part of the Phoenix installer. You can learn more about them or any other task by calling `mix help TASK`.

mix phx.gen.html

Phoenix offers the ability to generate all the code to stand up a complete HTML resource — Ecto migration, Ecto context, controller with all the necessary actions, view, and templates. This can be a tremendous time saver. Let's take a look at how to make this happen.

The `mix phx.gen.html` task takes the following arguments: the module name of the context, the module name of the schema, the resource name, and a list of column_name:type attributes. The module name we pass in must conform to the Elixir rules of module naming, following proper capitalization.

```
$ mix phx.gen.html Blog Post posts body:string
word_count:integer
* creating lib/hello_web/controllers/post_controller.ex
* creating
lib/hello_web/controllers/post_html/edit.html.heex
```

```
* creating
lib/hello_web/controllers/post_html/post_form.html.heex
* creating
lib/hello_web/controllers/post_html/index.html.heex
* creating
lib/hello_web/controllers/post_html/new.html.heex
* creating
lib/hello_web/controllers/post_html/show.html.heex
* creating lib/hello_web/controllers/post_html.ex
* creating
test/hello_web/controllers/post_controller_test.exs
* creating lib/hello/blog/post.ex
* creating
priv/repo/migrations/20211001233016_create_posts.exs
* creating lib/hello/blog.ex
* injecting lib/hello/blog.ex
* creating test/hello/blog_test.exs
* injecting test/hello/blog_test.exs
* creating test/support/fixtures/blog_fixtures.ex
* injecting test/support/fixtures/blog_fixtures.ex
```

When `mix phx.gen.html` is done creating files, it helpfully tells us that we need to add a line to our router file as well as run our Ecto migrations.

```
Add the resource to your browser scope in
lib/hello_web/router.ex:

    resources "/posts", PostController

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

Important: If we don't do this, we will see the following warnings in our logs, and our application will error when compiling.

```
$ mix phx.server
Compiling 17 files (.ex)

warning: no route path for HelloWeb.Router matches
```

```
\"/posts\"
  lib/hello_web/controllers/post_controller.ex:22:
HelloWeb.PostController.index/2
```

If we don't want to create a context or schema for our resource we can use the `--no-context` flag. Note that this still requires a context module name as a parameter.

```
$ mix phx.gen.html Blog Post posts body:string
word_count:integer --no-context
* creating lib/hello_web/controllers/post_controller.ex
* creating
lib/hello_web/controllers/post_html/edit.html.heex
* creating
lib/hello_web/controllers/post_html/post_form.html.heex
* creating
lib/hello_web/controllers/post_html/index.html.heex
* creating
lib/hello_web/controllers/post_html/new.html.heex
* creating
lib/hello_web/controllers/post_html/show.html.heex
* creating lib/hello_web/controllers/post_html.ex
* creating
test/hello_web/controllers/post_controller_test.exs
```

It will tell us we need to add a line to our router file, but since we skipped the context, it won't mention anything about `ecto.migrate`.

```
Add the resource to your browser scope in
lib/hello_web/router.ex:

    resources "/posts", PostController
```

Similarly, if we want a context created without a schema for our resource we can use the `--no-schema` flag.

```
$ mix phx.gen.html Blog Post posts body:string
word_count:integer --no-schema
```

```
* creating lib/hello_web/controllers/post_controller.ex
* creating
lib/hello_web/controllers/post_html/edit.html.heex
* creating
lib/hello_web/controllers/post_html/post_form.html.heex
* creating
lib/hello_web/controllers/post_html/index.html.heex
* creating
lib/hello_web/controllers/post_html/new.html.heex
* creating
lib/hello_web/controllers/post_html/show.html.heex
* creating lib/hello_web/controllers/post_html.ex
* creating
test/hello_web/controllers/post_controller_test.exs
* creating lib/hello/blog.ex
* injecting lib/hello/blog.ex
* creating test/hello/blog_test.exs
* injecting test/hello/blog_test.exs
* creating test/support/fixtures/blog_fixtures.ex
* injecting test/support/fixtures/blog_fixtures.ex
```

It will tell us we need to add a line to our router file, but since we skipped the schema, it won't mention anything about `ecto.migrate`.

`mix phx.gen.json`

Phoenix also offers the ability to generate all the code to stand up a complete JSON resource — Ecto migration, Ecto schema, controller with all the necessary actions and view. This command will not create any template for the app.

The `mix phx.gen.json` task takes the following arguments: the module name of the context, the module name of the schema, the resource name, and a list of column_name:type attributes. The module name we pass in must conform to the Elixir rules of module naming, following proper capitalization.

```
$ mix phx.gen.json Blog Post posts title:string
content:string
```

```
* creating lib/hello_web/controllers/post_controller.ex
* creating lib/hello_web/controllers/post_json.ex
* creating
test/hello_web/controllers/post_controller_test.exs
* creating lib/hello_web/controllers/changeset_json.ex
* creating
lib/hello_web/controllers/fallback_controller.ex
* creating lib/hello/blog/post.ex
* creating
priv/repo/migrations/20170906153323_create_posts.exs
* creating lib/hello/blog.ex
* injecting lib/hello/blog.ex
* creating test/hello/blog/blog_test.exs
* injecting test/hello/blog/blog_test.exs
* creating test/support/fixtures/blog_fixtures.ex
* injecting test/support/fixtures/blog_fixtures.ex
```

When `mix phx.gen.json` is done creating files, it helpfully tells us that we need to add a line to our router file as well as run our Ecto migrations.

```
Add the resource to the "/api" scope in
lib/hello_web/router.ex:

    resources "/posts", PostController, except: [:new,
:edit]

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

Important: If we don't do this, we'll get the following warning in our logs and the application will error when attempting to compile:

```
$ mix phx.server
Compiling 19 files (.ex)

warning: no route path for HelloWeb.Router matches
\"/posts\"
  lib/hello_web/controllers/post_controller.ex:22:
HelloWeb.PostController.index/2
```

`mix phx.gen.json` also supports `--no-context`, `--no-schema`, and others, as in `mix phx.gen.html`.

mix phx.gen.context

If we don't need a complete HTML/JSON resource and only need a context, we can use the `mix phx.gen.context` task. It will generate a context, a schema, a migration and a test case.

The `mix phx.gen.context` task takes the following arguments: the module name of the context, the module name of the schema, the resource name, and a list of column_name:type attributes.

```
$ mix phx.gen.context Accounts User users name:string
age:integer
* creating lib/hello/accounts/user.ex
* creating
priv/repo/migrations/20170906161158_create_users.exs
* creating lib/hello/accounts.ex
* injecting lib/hello/accounts.ex
* creating test/hello/accounts/accounts_test.exs
* injecting test/hello/accounts/accounts_test.exs
* creating test/support/fixtures/accounts_fixtures.ex
* injecting test/support/fixtures/accounts_fixtures.ex
```

Note: If we need to namespace our resource we can simply namespace the first argument of the generator.

```
$ mix phx.gen.context Admin.Accounts User users
name:string age:integer
* creating lib/hello/admin/accounts/user.ex
* creating
priv/repo/migrations/20170906161246_create_users.exs
* creating lib/hello/admin/accounts.ex
* injecting lib/hello/admin/accounts.ex
* creating test/hello/admin/accounts_test.exs
* injecting test/hello/admin/accounts_test.exs
* creating
test/support/fixtures/admin/accounts_fixtures.ex
```

```
* injecting
test/support/fixtures/admin/accounts_fixtures.ex
```

mix phx.gen.schema

If we don't need a complete HTML/JSON resource and are not interested in generating or altering a context we can use the `mix phx.gen.schema` task. It will generate a schema, and a migration.

The `mix phx.gen.schema` task takes the following arguments: the module name of the schema (which may be namespaced), the resource name, and a list of column_name:type attributes.

```
$ mix phx.gen.schema Accounts.Credential credentials
email:string:unique user_id:references:users
* creating lib/hello/accounts/credential.ex
* creating
priv/repo/migrations/20170906162013_create_credentials.ex
s
```

mix phx.gen.auth

Phoenix also offers the ability to generate all of the code to stand up a complete authentication system — Ecto migration, phoenix context, controllers, templates, etc. This can be a huge time saver, allowing you to quickly add authentication to your system and shift your focus back to the primary problems your application is trying to solve.

The `mix phx.gen.auth` task takes the following arguments: the module name of the context, the module name of the schema, and a plural version of the schema name used to generate database tables and route paths.

Here is an example version of the command:

```
$ mix phx.gen.auth Accounts User users
* creating
```

```
priv/repo/migrations/20201205184926_create_users_auth_tab
les.exs
* creating lib/hello/accounts/user_notifier.ex
* creating lib/hello/accounts/user.ex
* creating lib/hello/accounts/user_token.ex
* creating lib/hello_web/controllers/user_auth.ex
* creating test/hello_web/controllers/user_auth_test.exs
* creating
lib/hello_web/controllers/user_confirmation_html.ex
* creating
lib/hello_web/templates/user_confirmation/new.html.heex
* creating
lib/hello_web/templates/user_confirmation/edit.html.heex
* creating
lib/hello_web/controllers/user_confirmation_controller.ex
* creating
test/hello_web/controllers/user_confirmation_controller_t
est.exs
* creating
lib/hello_web/templates/user_registration/new.html.heex
* creating
lib/hello_web/controllers/user_registration_controller.ex
* creating
test/hello_web/controllers/user_registration_controller_t
est.exs
* creating
lib/hello_web/controllers/user_registration_html.ex
* creating
lib/hello_web/controllers/user_reset_password_html.ex
* creating
lib/hello_web/controllers/user_reset_password_controller.
ex
* creating
test/hello_web/controllers/user_reset_password_controller
_test.exs
* creating
lib/hello_web/templates/user_reset_password/edit.html.hee
x
* creating
lib/hello_web/templates/user_reset_password/new.html.heex
* creating lib/hello_web/controllers/user_session_html.ex
* creating
lib/hello_web/controllers/user_session_controller.ex
* creating
test/hello_web/controllers/user_session_controller_test.e
```

```
xs
* creating
lib/hello_web/templates/user_session/new.html.heex
* creating
lib/hello_web/controllers/user_settings_html.ex
* creating
lib/hello_web/templates/user_settings/edit.html.heex
* creating
lib/hello_web/controllers/user_settings_controller.ex
* creating
test/hello_web/controllers/user_settings_controller_test.
exs
* creating lib/hello/accounts.ex
* injecting lib/hello/accounts.ex
* creating test/hello/accounts_test.exs
* injecting test/hello/accounts_test.exs
* creating test/support/fixtures/accounts_fixtures.ex
* injecting test/support/fixtures/accounts_fixtures.ex
* injecting test/support/conn_case.ex
* injecting config/test.exs
* injecting mix.exs
* injecting lib/hello_web/router.ex
* injecting lib/hello_web/router.ex - imports
* injecting lib/hello_web/router.ex - plug
* injecting lib/hello_web/templates/layout/root.html.heex
```

When `mix phx.gen.auth` is done creating files, it helpfully tells us that we need to re-fetch our dependencies as well as run our Ecto migrations.

```
Please re-fetch your dependencies with the following
command:

    mix deps.get

Remember to update your repository by running migrations:

  $ mix ecto.migrate

Once you are ready, visit "/users/register"
to create your account and then access to "/dev/mailbox"
to
see the account confirmation email.
```

A more complete walk-through of how to get started with this generator is available in the `mix phx.gen.auth` authentication guide.

`mix phx.gen.channel` and `mix phx.gen.socket`

This task will generate a basic Phoenix channel, the socket to power the channel (if you haven't created one yet), as well a test case for it. It takes the module name for the channel as the only argument:

```
$ mix phx.gen.channel Room
* creating lib/hello_web/channels/room_channel.ex
* creating test/hello_web/channels/room_channel_test.exs
```

If your application does not have a `UserSocket` yet, it will ask if you want to create one:

```
The default socket handler - HelloWeb.UserSocket - was
not found
in its default location.

Do you want to create it? [Y/n]
```

By confirming, a channel will be created, then you need to connect the socket in your endpoint:

```
Add the socket handler to your
`lib/hello_web/endpoint.ex`, for example:

    socket "/socket", HelloWeb.UserSocket,
      websocket: true,
      longpoll: false

For the front-end integration, you need to import the
`user_socket.js`
in your `assets/js/app.js` file:

    import "./user_socket.js"
```

In case a `UserSocket` already exists or you decide to not create one, the `channel` generator will tell you to add it to the Socket manually:

```
Add the channel to your
`lib/hello_web/channels/user_socket.ex` handler, for
example:

    channel "rooms:lobby", HelloWeb.RoomChannel
```

You can also create a socket any time by invoking `mix phx.gen.socket`.

mix phx.gen.presence

This task will generate a presence tracker. The module name can be passed as an argument, `Presence` is used if no module name is passed.

```
$ mix phx.gen.presence Presence
* lib/hello_web/channels/presence.ex

Add your new module to your supervision tree,
in lib/hello/application.ex:

    children = [
      ...
      HelloWeb.Presence
    ]
```

mix phx.routes

This task has a single purpose, to show us all the routes defined for a given router. We saw it used extensively in the routing guide.

If we don't specify a router for this task, it will default to the router Phoenix generated for us.

```
$ mix phx.routes
GET  /  TaskTester.PageController.index/2
```

We can also specify an individual router if we have more than one for our application.

```
$ mix phx.routes TaskTesterWeb.Router
GET  /  TaskTesterWeb.PageController.index/2
```

mix phx.server

This is the task we use to get our application running. It takes no arguments at all. If we pass any in, they will be silently ignored.

```
$ mix phx.server
[info] Running TaskTesterWeb.Endpoint with Cowboy on port
4000 (http)
```

It will silently ignore our `DoesNotExist` argument:

```
$ mix phx.server DoesNotExist
[info] Running TaskTesterWeb.Endpoint with Cowboy on port
4000 (http)
```

If we would like to start our application and also have an `IEx` session open to it, we can run the Mix task within `iex` like this, `iex -S mix phx.server`.

```
$ iex -S mix phx.server
Erlang/OTP 17 [erts-6.4] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false] [dtrace]

[info] Running TaskTesterWeb.Endpoint with Cowboy on port
4000 (http)
Interactive Elixir (1.0.4) - press Ctrl+C to exit (type
h() ENTER for help)
iex(1)>
```

`mix phx.digest`

This task does two things, it creates a digest for our static assets and then compresses them.

"Digest" here refers to an MD5 digest of the contents of an asset which gets added to the filename of that asset. This creates a sort of fingerprint for it. If the digest doesn't change, browsers and CDNs will use a cached version. If it does change, they will re-fetch the new version.

Before we run this task let's inspect the contents of two directories in our hello application.

First `priv/static/` which should look similar to this:

```
├── assets
│   ├── app.css
│   └── app.js
├── favicon.ico
└── robots.txt
```

And then `assets/` which should look similar to this:

```
├── css
│   └── app.css
├── js
│   └── app.js
├── tailwind.config.js
└── vendor
    └── topbar.js
```

All of these files are our static assets. Now let's run the `mix phx.digest` task.

```
$ mix phx.digest
Check your digested files at 'priv/static'.
```

We can now do as the task suggests and inspect the contents of `priv/static/` directory. We'll see that all files from `assets/` have been copied over to `priv/static/` and also each file now has a couple of versions. Those versions are:

- the original file
- a compressed file with gzip
- a file containing the original file name and its digest
- a compressed file containing the file name and its digest

We can optionally determine which files should be gzipped by using the `:gzippable_exts` option in the config file:

```
config :phoenix, :gzippable_exts, ~w(.js .css)
```

> Note: We can specify a different output folder where `mix phx.digest` will put processed files. The first argument is the path where the static files are located.

```
$ mix phx.digest priv/static/ -o www/public/
Check your digested files at 'www/public/'
```

> Note: You can use `mix phx.digest.clean` to prune stale versions of the assets. If you want to remove all produced files, run `mix phx.digest.clean --all`.

Ecto tasks

Newly generated Phoenix applications now include Ecto and Postgrex as dependencies by default (which is to say, unless we use `mix phx.new` with

the `--no-ecto` flag). With those dependencies come Mix tasks to take care of common Ecto operations. Let's see which tasks we get out of the box.

```
$ mix help --search "ecto"
mix ecto                # Prints Ecto help information
mix ecto.create         # Creates the repository storage
mix ecto.drop           # Drops the repository storage
mix ecto.dump           # Dumps the repository database
structure
mix ecto.gen.migration # Generates a new migration for
the repo
mix ecto.gen.repo       # Generates a new repository
mix ecto.load           # Loads previously dumped database
structure
mix ecto.migrate        # Runs the repository migrations
mix ecto.migrations     # Displays the repository
migration status
mix ecto.reset          # Alias defined in mix.exs
mix ecto.rollback       # Rolls back the repository
migrations
mix ecto.setup          # Alias defined in mix.exs
```

Note: We can run any of the tasks above with the `--no-start` flag to execute the task without starting the application.

mix ecto.create

This task will create the database specified in our repo. By default it will look for the repo named after our application (the one generated with our app unless we opted out of Ecto), but we can pass in another repo if we want.

Here's what it looks like in action.

```
$ mix ecto.create
The database for Hello.Repo has been created.
```

There are a few things that can go wrong with `ecto.create`. If our Postgres database doesn't have a "postgres" role (user), we'll get an error like this

one.

```
$ mix ecto.create
** (Mix) The database for Hello.Repo couldn't be created,
reason given: psql: FATAL:  role "postgres" does not
exist
```

We can fix this by creating the "postgres" role in the `psql` console with the permissions needed to log in and create a database.

```
=# CREATE ROLE postgres LOGIN CREATEDB;
CREATE ROLE
```

If the "postgres" role does not have permission to log in to the application, we'll get this error.

```
$ mix ecto.create
** (Mix) The database for Hello.Repo couldn't be created,
reason given: psql: FATAL:  role "postgres" is not
permitted to log in
```

To fix this, we need to change the permissions on our "postgres" user to allow login.

```
=# ALTER ROLE postgres LOGIN;
ALTER ROLE
```

If the "postgres" role does not have permission to create a database, we'll get this error.

```
$ mix ecto.create
** (Mix) The database for Hello.Repo couldn't be created,
reason given: ERROR:  permission denied to create
database
```

To fix this, we need to change the permissions on our "postgres" user in the `psql` console to allow database creation.

```
=# ALTER ROLE postgres CREATEDB;
ALTER ROLE
```

If the "postgres" role is using a password different from the default "postgres", we'll get this error.

```
$ mix ecto.create
** (Mix) The database for Hello.Repo couldn't be created,
reason given: psql: FATAL:  password authentication
failed for user "postgres"
```

To fix this, we can change the password in the environment specific configuration file. For the development environment the password used can be found at the bottom of the `config/dev.exs` file.

Finally, if we happen to have another repo called `OurCustom.Repo` that we want to create the database for, we can run this.

```
$ mix ecto.create -r OurCustom.Repo
The database for OurCustom.Repo has been created.
```

mix ecto.drop

This task will drop the database specified in our repo. By default it will look for the repo named after our application (the one generated with our app unless we opted out of Ecto). It will not prompt us to check if we're sure we want to drop the database, so do exercise caution.

```
$ mix ecto.drop
The database for Hello.Repo has been dropped.
```

If we happen to have another repo that we want to drop the database for, we can specify it with the `-r` flag.

```
$ mix ecto.drop -r OurCustom.Repo
The database for OurCustom.Repo has been dropped.
```

mix ecto.gen.repo

Many applications require more than one data store. For each data store, we'll need a new repo, and we can generate them automatically with `ecto.gen.repo`.

If we name our repo `OurCustom.Repo`, this task will create it here `lib/our_custom/repo.ex`.

```
$ mix ecto.gen.repo -r OurCustom.Repo
* creating lib/our_custom
* creating lib/our_custom/repo.ex
* updating config/config.exs
Don't forget to add your new repo to your supervision
tree
(typically in lib/hello/application.ex):

    {OurCustom.Repo, []}
```

Notice that this task has updated `config/config.exs`. If we take a look, we'll see this extra configuration block for our new repo.

```
. . .
config :hello, OurCustom.Repo,
  username: "user",
  password: "pass",
  hostname: "localhost",
  database: "hello_repo",
. . .
```

Of course, we'll need to change the login credentials to match what our database expects. We'll also need to change the config for other environments.

We certainly should follow the instructions and add our new repo to our supervision tree. In our `Hello` application, we would open up `lib/hello/application.ex`, and add our repo as a worker to the `children` list.

```
. . .
children = [
  Hello.Repo,
  # Our custom repo
  OurCustom.Repo,
  # Start the endpoint when the application starts
  HelloWeb.Endpoint,
]
. . .
```

mix ecto.gen.migration

Migrations are a programmatic, repeatable way to affect changes to a database schema. Migrations are also just modules, and we can create them with the `ecto.gen.migration` task. Let's walk through the steps to create a migration for a new comments table.

We simply need to invoke the task with a `snake_case` version of the module name that we want. Preferably, the name will describe what we want the migration to do.

```
$ mix ecto.gen.migration add_comments_table
* creating priv/repo/migrations
* creating
priv/repo/migrations/20150318001628_add_comments_table.ex
s
```

Notice that the migration's filename begins with a string representation of the date and time the file was created.

Let's take a look at the file `ecto.gen.migration` has generated for us at `priv/repo/migrations/20150318001628_add_comments_table.exs`.

```
defmodule Hello.Repo.Migrations.AddCommentsTable do
  use Ecto.Migration

  def change do
  end
end
```

Notice that there is a single function `change/0` which will handle both forward migrations and rollbacks. We'll define the schema changes that we want using Ecto's handy DSL, and Ecto will figure out what to do depending on whether we are rolling forward or rolling back. Very nice indeed.

What we want to do is create a `comments` table with a `body` column, a `word_count` column, and timestamp columns for `inserted_at` and `updated_at`.

```
. . .
def change do
  create table(:comments) do
    add :body, :string
    add :word_count, :integer
    timestamps()
  end
end
. . .
```

Again, we can run this task with the `-r` flag and another repo if we need to.

```
$ mix ecto.gen.migration -r OurCustom.Repo add_users
* creating priv/repo/migrations
```

```
* creating
priv/repo/migrations/20150318172927_add_users.exs
```

For more information on how to modify your database schema please refer to the Ecto's migration DSL docs. For example, to alter an existing schema see the documentation on Ecto's `alter/2` function.

That's it! We're ready to run our migration.

`mix ecto.migrate`

Once we have our migration module ready, we can simply run `mix ecto.migrate` to have our changes applied to the database.

```
$ mix ecto.migrate
[info] == Running
Hello.Repo.Migrations.AddCommentsTable.change/0 forward
[info] create table comments
[info] == Migrated in 0.1s
```

When we first run `ecto.migrate`, it will create a table for us called `schema_migrations`. This will keep track of all the migrations which we run by storing the timestamp portion of the migration's filename.

Here's what the `schema_migrations` table looks like.

```
hello_dev=# select * from schema_migrations;
version        |     inserted_at
---------------+---------------------
20150317170448 | 2015-03-17 21:07:26
20150318001628 | 2015-03-18 01:45:00
(2 rows)
```

When we roll back a migration, `ecto.rollback` will remove the record representing this migration from `schema_migrations`.

By default, `ecto.migrate` will execute all pending migrations. We can exercise more control over which migrations we run by specifying some options when we run the task.

We can specify the number of pending migrations we would like to run with the `-n` or `--step` options.

```
$ mix ecto.migrate -n 2
[info] == Running
Hello.Repo.Migrations.CreatePost.change/0 forward
[info] create table posts
[info] == Migrated in 0.0s
[info] == Running
Hello.Repo.Migrations.AddCommentsTable.change/0 forward
[info] create table comments
[info] == Migrated in 0.0s
```

The `--step` option will behave the same way.

```
mix ecto.migrate --step 2
```

The `--to` option will run all migrations up to and including given version.

```
mix ecto.migrate --to 20150317170448
```

mix ecto.rollback

The `ecto.rollback` task will reverse the last migration we have run, undoing the schema changes. `ecto.migrate` and `ecto.rollback` are mirror images of each other.

```
$ mix ecto.rollback
[info] == Running
Hello.Repo.Migrations.AddCommentsTable.change/0 backward
```

```
[info] drop table comments
[info] == Migrated in 0.0s
```

`ecto.rollback` will handle the same options as `ecto.migrate`, so `-n`, `--step`, `-v`, and `--to` will behave as they do for `ecto.migrate`.

Creating our own Mix task

As we've seen throughout this guide, both Mix itself and the dependencies we bring in to our application provide a number of really useful tasks for free. Since neither of these could possibly anticipate all our individual application's needs, Mix allows us to create our own custom tasks. That's exactly what we are going to do now.

The first thing we need to do is create a `mix/tasks/` directory inside of `lib/`. This is where any of our application specific Mix tasks will go.

```
$ mkdir -p lib/mix/tasks/
```

Inside that directory, let's create a new file, `hello.greeting.ex`, that looks like this.

```
defmodule Mix.Tasks.Hello.Greeting do
  use Mix.Task

  @shortdoc "Sends a greeting to us from Hello Phoenix"

  @moduledoc """
  This is where we would put any long form documentation
and doctests.
  """

  @impl Mix.Task
  def run(_args) do
    Mix.shell().info("Greetings from the Hello Phoenix
Application!")
  end
```

```
  # We can define other functions as needed here.
end
```

Let's take a quick look at the moving parts involved in a working Mix task.

The first thing we need to do is name our module. All tasks must be defined in the `Mix.Tasks` namespace. We'd like to invoke this as `mix hello.greeting`, so we complete the module name with `Hello.Greeting`.

The `use Mix.Task` line brings in functionality from Mix that makes this module behave as a Mix task.

The `@shortdoc` module attribute holds a string which will describe our task when users invoke `mix help`.

`@moduledoc` serves the same function that it does in any module. It's where we can put long-form documentation and doctests, if we have any.

The `run/1` function is the critical heart of any Mix task. It's the function that does all the work when users invoke our task. In ours, all we do is send a greeting from our app, but we can implement our `run/1` function to do whatever we need it to. Note that `Mix.shell().info/1` is the preferred way to print text back out to the user.

Of course, our task is just a module, so we can define other private functions as needed to support our `run/1` function.

Now that we have our task module defined, our next step is to compile the application.

```
$ mix compile
Compiled lib/tasks/hello.greeting.ex
Generated hello.app
```

Now our new task should be visible to `mix help`.

```
$ mix help --search hello
mix hello.greeting # Sends a greeting to us from Hello
Phoenix
```

Notice that `mix help` displays the text we put into the `@shortdoc` along with the name of our task.

So far, so good, but does it work?

```
$ mix hello.greeting
Greetings from the Hello Phoenix Application!
```

Indeed it does.

If you want to make your new Mix task to use your application's infrastructure, you need to make sure the application is started and configure when Mix task is being executed. This is particularly useful if you need to access your database from within the Mix task. Thankfully, Mix makes it really easy for us via the `@requirements` module attribute:

```
  @requirements ["app.start"]

  @impl Mix.Task
  def run(_args) do
    Mix.shell().info("Now I have access to Repo and other
goodies!")
    Mix.shell().info("Greetings from the Hello Phoenix
Application!")
  end
```

Telemetry

In this guide, we will show you how to instrument and report on `:telemetry` events in your Phoenix application.

> `te·lem·e·try` - the process of recording and transmitting the readings of an instrument.

As you follow along with this guide, we will introduce you to the core concepts of Telemetry, you will initialize a reporter to capture your application's events as they occur, and we will guide you through the steps to properly instrument your own functions using `:telemetry`. Let's take a closer look at how Telemetry works in your application.

Overview

The `[:telemetry]` library allows you to emit events at various stages of an application's lifecycle. You can then respond to these events by, among other things, aggregating them as metrics and sending the metrics data to a reporting destination.

Telemetry stores events by their name in an ETS table, along with the handler for each event. Then, when a given event is executed, Telemetry looks up its handler and invokes it.

Phoenix's Telemetry tooling provides you with a supervisor that uses `Telemetry.Metrics` to define the list of Telemetry events to handle and how to handle those events, i.e. how to structure them as a certain type of metric. This supervisor works together with Telemetry reporters to respond to the specified Telemetry events by aggregating them as the appropriate metric and sending them to the correct reporting destination.

The Telemetry supervisor

Since v1.5, new Phoenix applications are generated with a Telemetry supervisor. This module is responsible for managing the lifecycle of your Telemetry processes. It also defines a `metrics/0` function, which returns a list of `Telemetry.Metrics` that you define for your application.

By default, the supervisor also starts `:telemetry_poller`. By simply adding `:telemetry_poller` as a dependency, you can receive VM-related events on a specified interval.

If you are coming from an older version of Phoenix, install the `:telemetry_metrics` and `:telemetry_poller` packages:

```
{:telemetry_metrics, "~> 1.0"},
{:telemetry_poller, "~> 1.0"}
```

and create your Telemetry supervisor at `lib/my_app_web/telemetry.ex`:

```
# lib/my_app_web/telemetry.ex
defmodule MyAppWeb.Telemetry do
  use Supervisor
  import Telemetry.Metrics

  def start_link(arg) do
    Supervisor.start_link(__MODULE__, arg, name:
__MODULE__)
  end

  def init(_arg) do
    children = [
      {:telemetry_poller, measurements:
periodic_measurements(), period: 10_000}
      # Add reporters as children of your supervision
tree.
      # {Telemetry.Metrics.ConsoleReporter, metrics:
metrics()}
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
```

```
  def metrics do
    [
      # Phoenix Metrics
      summary("phoenix.endpoint.stop.duration",
        unit: {:native, :millisecond}
      ),
      summary("phoenix.router_dispatch.stop.duration",
        tags: [:route],
        unit: {:native, :millisecond}
      ),
      # VM Metrics
      summary("vm.memory.total", unit: {:byte,
:kilobyte}),
      summary("vm.total_run_queue_lengths.total"),
      summary("vm.total_run_queue_lengths.cpu"),
      summary("vm.total_run_queue_lengths.io")
    ]
  end

  defp periodic_measurements do
    [
      # A module, function and arguments to be invoked
periodically.
      # This function must call :telemetry.execute/3 and
a metric must be added above.
      # {MyApp, :count_users, []}
    ]
  end
end
```

Make sure to replace MyApp by your actual application name.

Then add to your main application's supervision tree (usually in `lib/my_app/application.ex`):

```
children = [
  MyAppWeb.Telemetry,
  MyApp.Repo,
  MyAppWeb.Endpoint,
  ...
]
```

Telemetry Events

Many Elixir libraries (including Phoenix) are already using the `:telemetry` package as a way to give users more insight into the behavior of their applications, by emitting events at key moments in the application lifecycle.

A Telemetry event is made up of the following:

- `name` - A string (e.g. `"my_app.worker.stop"`) or a list of atoms that uniquely identifies the event.
- `measurements` - A map of atom keys (e.g. `:duration`) and numeric values.
- `metadata` - A map of key-value pairs that can be used for tagging metrics.

A Phoenix Example

Here is an example of an event from your endpoint:

- `[:phoenix, :endpoint, :stop]` - dispatched by `Plug.Telemetry`, one of the default plugs in your endpoint, whenever the response is sent
  - Measurement: `%{duration: native_time}`
  - Metadata: `%{conn: Plug.Conn.t}`

This means that after each request, `Plug`, via `:telemetry`, will emit a "stop" event, with a measurement of how long it took to get the response:

```
:telemetry.execute([:phoenix, :endpoint, :stop], %
{duration: duration}, %{conn: conn})
```

Phoenix Telemetry Events

A full list of all Phoenix telemetry events can be found in `Phoenix.Logger`

Metrics

> Metrics are aggregations of Telemetry events with a specific name, providing a view of the system's behaviour over time.
>
> — `Telemetry.Metrics`

The Telemetry.Metrics package provides a common interface for defining metrics. It exposes a set of five metric type functions that are responsible for structuring a given Telemetry event as a particular measurement.

The package does not perform any aggregation of the measurements itself. Instead, it provides a reporter with the Telemetry event-as-measurement definition and the reporter uses that definition to perform aggregations and report them.

We will discuss reporters in the next section.

Let's take a look at some examples.

Using `Telemetry.Metrics`, you can define a counter metric, which counts how many HTTP requests were completed:

```
Telemetry.Metrics.counter("phoenix.endpoint.stop.duration
")
```

or you could use a distribution metric to see how many requests were completed in particular time buckets:

```
Telemetry.Metrics.distribution("phoenix.endpoint.stop.dur
ation")
```

This ability to introspect HTTP requests is really powerful -- and this is but one of *many* telemetry events emitted by the Phoenix framework! We'll discuss more of these events, as well as specific patterns for extracting valuable data from Phoenix/Plug events in the Phoenix Metrics section later in this guide.

> The full list of `:telemetry` events emitted from Phoenix, along with their measurements and metadata, is available in the "Instrumentation" section of the `Phoenix.Logger` module documentation.

An Ecto Example

Like Phoenix, Ecto ships with built-in Telemetry events. This means that you can gain introspection into your web and database layers using the same tools.

Here is an example of a Telemetry event executed by Ecto when an Ecto repository starts:

- `[:ecto, :repo, :init]` - dispatched by `Ecto.Repo`
  - Measurement: `%{system_time: native_time}`
  - Metadata: `%{repo: Ecto.Repo, opts: Keyword.t()}`

This means that whenever the `Ecto.Repo` starts, it will emit an event, via `:telemetry`, with a measurement of the time at start-up.

```
:telemetry.execute([:ecto, :repo, :init], %{system_time:
System.system_time()}, %{repo: repo, opts: opts})
```

Additional Telemetry events are executed by Ecto adapters.

One such adapter-specific event is the `[:my_app, :repo, :query]` event. For instance, if you want to graph query execution time, you can use the `Telemetry.Metrics.summary/2` function to instruct your reporter to

calculate statistics of the `[:my_app, :repo, :query]` event, like maximum, mean, percentiles etc.:

```
Telemetry.Metrics.summary("my_app.repo.query.query_time",
  unit: {:native, :millisecond}
)
```

Or you could use the `Telemetry.Metrics.distribution/2` function to define a histogram for another adapter-specific event: `[:my_app, :repo, :query, :queue_time]`, thus visualizing how long queries spend queued:

```
Telemetry.Metrics.distribution("my_app.repo.query.queue_time",
  unit: {:native, :millisecond}
)
```

> You can learn more about Ecto Telemetry in the "Telemetry Events" section of the `Ecto.Repo` module documentation.

So far we have seen some of the Telemetry events common to Phoenix applications, along with some examples of their various measurements and metadata. With all of this data just waiting to be consumed, let's talk about reporters.

Reporters

Reporters subscribe to Telemetry events using the common interface provided by `Telemetry.Metrics`. They then aggregate the measurements (data) into metrics to provide meaningful information about your application.

For example, if the following `Telemetry.Metrics.summary/2` call is added to the `metrics/0` function of your Telemetry supervisor:

```
summary("phoenix.endpoint.stop.duration",
  unit: {:native, :millisecond}
```

```
)
```

Then the reporter will attach a listener for the `"phoenix.endpoint.stop.duration"` event and will respond to this event by calculating a summary metric with the given event metadata and reporting on that metric to the appropriate source.

Phoenix.LiveDashboard

For developers interested in real-time visualizations for their Telemetry metrics, you may be interested in installing `LiveDashboard`. LiveDashboard acts as a Telemetry.Metrics reporter to render your data as beautiful, real-time charts on the dashboard.

Telemetry.Metrics.ConsoleReporter

`Telemetry.Metrics` ships with a `ConsoleReporter` that can be used to print events and metrics to the terminal. You can use this reporter to experiment with the metrics discussed in this guide.

Uncomment or add the following to this list of children in your Telemetry supervision tree (usually in `lib/my_app_web/telemetry.ex`):

```
{Telemetry.Metrics.ConsoleReporter, metrics: metrics()}
```

> There are numerous reporters available, for services like StatsD, Prometheus, and more. You can find them by searching for "telemetry_metrics" on hex.pm.

Phoenix Metrics

Earlier we looked at the "stop" event emitted by `Plug.Telemetry`, and used it to count the number of HTTP requests. In reality, it's only somewhat helpful

to be able to see just the total number of requests. What if you wanted to see the number of requests per route, or per route *and* method?

Let's take a look at another event emitted during the HTTP request lifecycle, this time from `Phoenix.Router`:

- `[:phoenix, :router_dispatch, :stop]` - dispatched by Phoenix.Router after successfully dispatching to a matched route
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{conn: Plug.Conn.t, route: binary, plug: module, plug_opts: term, path_params: map, pipe_through: [atom]}`

Let's start by grouping these events by route. Add the following (if it does not already exist) to the `metrics/0` function of your Telemetry supervisor (usually in `lib/my_app_web/telemetry.ex`):

```
# lib/my_app_web/telemetry.ex
def metrics do
  [
    ...metrics...
    summary("phoenix.router_dispatch.stop.duration",
      tags: [:route],
      unit: {:native, :millisecond}
    )
  ]
end
```

Restart your server, and then make requests to a page or two. In your terminal, you should see the ConsoleReporter print logs for the Telemetry events it received as a result of the metrics definitions you provided.

The log line for each request contains the specific route for that request. This is due to specifying the `:tags` option for the summary metric, which takes care of our first requirement; we can use `:tags` to group metrics by route.

Note that reporters will necessarily handle tags differently depending on the underlying service in use.

Looking more closely at the Router "stop" event, you can see that the `Plug.Conn` struct representing the request is present in the metadata, but how do you access the properties in `conn`?

Fortunately, `Telemetry.Metrics` provides the following options to help you classify your events:

- `:tags` - A list of metadata keys for grouping;
- `:tag_values` - A function which transforms the metadata into the desired shape; Note that this function is called for each event, so it's important to keep it fast if the rate of events is high.

Learn about all the available metrics options in the `Telemetry.Metrics` module documentation.

Let's find out how to extract more tags from events that include a `conn` in their metadata.

Extracting tag values from Plug.Conn

Let's add another metric for the route event, this time to group by route and method:

```
summary("phoenix.router_dispatch.stop.duration",
  tags: [:method, :route],
  tag_values: &get_and_put_http_method/1,
  unit: {:native, :millisecond}
)
```

We've introduced the `:tag_values` option here, because we need to perform a transformation on the event metadata in order to get to the values we need.

Add the following private function to your Telemetry module to lift the `:method` value from the `Plug.Conn` struct:

```
# lib/my_app_web/telemetry.ex
defp get_and_put_http_method(%{conn: %{method: method}} =
metadata) do
  Map.put(metadata, :method, method)
end
```

Restart your server and make some more requests. You should begin to see logs with tags for both the HTTP method and the route.

Note the `:tags` and `:tag_values` options can be applied to all `Telemetry.Metrics` types.

Renaming value labels using tag values

Sometimes when displaying a metric, the value label may need to be transformed to improve readability. Take for example the following metric that displays the duration of the each LiveView's `mount/3` callback by `connected?` status.

```
summary("phoenix.live_view.mount.stop.duration",
  unit: {:native, :millisecond},
  tags: [:view, :connected?],
  tag_values: &live_view_metric_tag_values/1
)
```

The following function lifts `metadata.socket.view` and `metadata.socket.connected?` to be top-level keys on `metadata`, as we did in the previous example.

```
# lib/my_app_web/telemetry.ex
defp live_view_metric_tag_values(metadata) do
  metadata
  |> Map.put(:view, metadata.socket.view)
```

```
    |> Map.put(:connected?, Phoenix.LiveView.connected?
  (metadata.socket))
  end
```

However, when rendering these metrics in LiveDashboard, the value label is output as `"Elixir.Phoenix.LiveDashboard.MetricsLive true"`.

To make the value label easier to read, we can update our private function to generate more user friendly names. We'll run the value of the `:view` through `inspect/1` to remove the `Elixir.` prefix and call another private function to convert the `connected?` boolean into human readable text.

```
# lib/my_app_web/telemetry.ex
defp live_view_metric_tag_values(metadata) do
  metadata
  |> Map.put(:view, inspect(metadata.socket.view))
  |> Map.put(:connected?,
get_connection_status(Phoenix.LiveView.connected?
(metadata.socket)))
end

defp get_connection_status(true), do: "Connected"
defp get_connection_status(false), do: "Disconnected"
```

Now the value label will be rendered like `"Phoenix.LiveDashboard.MetricsLive Connected"`.

Hopefully, this gives you some inspiration on how to use the `:tag_values` option. Just remember to keep this function fast since it is called on every event.

Periodic measurements

You might want to periodically measure key-value pairs within your application. Fortunately the `:telemetry_poller` package provides a mechanism for custom measurements, which is useful for retrieving process information or for performing custom measurements periodically.

Add the following to the list in your Telemetry supervisor's `periodic_measurements/0` function, which is a private function that returns a list of measurements to take on a specified interval.

```
# lib/my_app_web/telemetry.ex
defp periodic_measurements do
  [
    {MyApp, :measure_users, []},
    {:process_info,
      event: [:my_app, :my_server],
      name: MyApp.MyServer,
      keys: [:message_queue_len, :memory]}
  ]
end
```

where `MyApp.measure_users/0` could be written like this:

```
# lib/my_app.ex
defmodule MyApp do
  def measure_users do
    :telemetry.execute([:my_app, :users], %{total:
MyApp.users_count()}, %{})
  end
end
```

Now with measurements in place, you can define the metrics for the events above:

```
# lib/my_app_web/telemetry.ex
def metrics do
  [
    ...metrics...
    # MyApp Metrics
    last_value("my_app.users.total"),
    last_value("my_app.my_server.memory", unit: :byte),
    last_value("my_app.my_server.message_queue_len")
    summary("my_app.my_server.call.stop.duration"),
    counter("my_app.my_server.call.exception")
```

```
    ]
  end
```

You will implement MyApp.MyServer in the Custom Events section.

Libraries using Telemetry

Telemetry is quickly becoming the de-facto standard for package instrumentation in Elixir. Here is a list of libraries currently emitting `:telemetry` events.

Library authors are actively encouraged to send a PR adding their own (in alphabetical order, please):

- Absinthe - Events
- Ash Framework - Events
- Broadway - Events
- Ecto - Events
- Oban - Events
- Phoenix - Events
- Plug - Events
- Tesla - Events

Custom Events

If you need custom metrics and instrumentation in your application, you can utilize the `:telemetry` package (https://hexdocs.pm/telemetry) just like your favorite frameworks and libraries.

Here is an example of a simple GenServer that emits telemetry events. Create this file in your app at `lib/my_app/my_server.ex`:

```
# lib/my_app/my_server.ex
defmodule MyApp.MyServer do
  @moduledoc """
  An example GenServer that runs arbitrary functions and
```

```
emits telemetry events when called.
  """
  use GenServer

  # A common prefix for :telemetry events
  @prefix [:my_app, :my_server, :call]

  def start_link(fun) do
    GenServer.start_link(__MODULE__, fun, name:
__MODULE__)
  end

  @doc """
  Runs the function contained within this server.

  ## Events

  The following events may be emitted:

    * `[:my_app, :my_server, :call, :start]` - Dispatched
      immediately before invoking the function. This
event
      is always emitted.

      * Measurement: `%{system_time: system_time}`

      * Metadata: `%{}`

    * `[:my_app, :my_server, :call, :stop]` - Dispatched
      immediately after successfully invoking the
function.

      * Measurement: `%{duration: native_time}`

      * Metadata: `%{}`

    * `[:my_app, :my_server, :call, :exception]` -
Dispatched
      immediately after invoking the function, in the
event
      the function throws or raises.

      * Measurement: `%{duration: native_time}`

      * Metadata: `%{kind: kind, reason: reason,
```

```
stacktrace: stacktrace}`
  """
  def call!, do: GenServer.call(__MODULE__, :called)

  @impl true
  def init(fun) when is_function(fun, 0), do: {:ok, fun}

  @impl true
  def handle_call(:called, _from, fun) do
    # Wrap the function invocation in a "span"
    result = telemetry_span(fun)

    {:reply, result, fun}
  end

  # Emits telemetry events related to invoking the
function
  defp telemetry_span(fun) do
    start_time = emit_start()

    try do
      fun.()
    catch
      kind, reason ->
        stacktrace = System.stacktrace()
        duration = System.monotonic_time() - start_time
        emit_exception(duration, kind, reason,
stacktrace)
        :erlang.raise(kind, reason, stacktrace)
    else
      result ->
        duration = System.monotonic_time() - start_time
        emit_stop(duration)
        result
    end
  end

  defp emit_start do
    start_time_mono = System.monotonic_time()

    :telemetry.execute(
      @prefix ++ [:start],
      %{system_time: System.system_time()},
      %{}
    )
```

```
      start_time_mono
    end

    defp emit_stop(duration) do
      :telemetry.execute(
        @prefix ++ [:stop],
        %{duration: duration},
        %{}
      )
    end

    defp emit_exception(duration, kind, reason, stacktrace)
  do
      :telemetry.execute(
        @prefix ++ [:exception],
        %{duration: duration},
        %{
          kind: kind,
          reason: reason,
          stacktrace: stacktrace
        }
      )
    end
  end
```

and add it to your application's supervisor tree (usually in `lib/my_app/application.ex`), giving it a function to invoke when called:

```
  # lib/my_app/application.ex
  children = [
    # Start a server that greets the world
    {MyApp.MyServer, fn -> "Hello, world!" end},
  ]
```

Now start an IEx session and call the server:

```
  iex> MyApp.MyServer.call!
```

and you should see something like the following output:

```
[Telemetry.Metrics.ConsoleReporter] Got new event!
Event name: my_app.my_server.call.stop
All measurements: %{duration: 4000}
All metadata: %{}

Metric measurement: #Function<2.111777250/1 in
Telemetry.Metrics.maybe_convert_measurement/2> (summary)
With value: 0.004 millisecond
Tag values: %{}

"Hello, world!"
```

Asset Management

Beside producing HTML, most web applications have various assets (JavaScript, CSS, images, fonts and so on).

From Phoenix v1.7, new applications use esbuild to prepare assets via the Elixir esbuild wrapper, and tailwindcss via the Elixir tailwindcss wrapper for CSS. The direct integration with `esbuild` and `tailwind` means that newly generated applications do not have dependencies on Node.js or an external build system (e.g. Webpack).

Your JavaScript is typically placed at "assets/js/app.js" and `esbuild` will extract it to "priv/static/assets/app.js". In development, this is done automatically via the `esbuild` watcher. In production, this is done by running `mix assets.deploy`.

`esbuild` can also handle your CSS files, but by default `tailwind` handles all CSS building.

Finally, all other assets, that usually don't have to be preprocessed, go directly to "priv/static".

Third-party JS packages

If you want to import JavaScript dependencies, you have at least three options to add them to your application:

1. Vendor those dependencies inside your project and import them in your "assets/js/app.js" using a relative path:

```
import topbar from "../vendor/topbar"
```

2. Call `npm install topbar --save` inside your assets directory and `esbuild` will be able to automatically pick them up:

```
import topbar from "topbar"
```

3. Use Mix to track the dependency from a source repository:

```
# mix.exs
{:topbar, github: "buunguyen/topbar", app: false,
compile: false}
```

Run `mix deps.get` to fetch the dependency and then import it:

```
import topbar from "topbar"
```

New applications use this third approach to import Heroicons, avoiding vendoring a copy of all icons when you may only use a few or even none, avoiding Node.js and `npm`, and tracking an explicit version that is easy to update thanks to Mix. It is important to note that git dependencies cannot be used by Hex packages, so if you intend to publish your project to Hex, consider vendoring the files instead.

Images, fonts, and external files

If you reference an external file in your CSS or JavaScript files, `esbuild` will attempt to validate and manage them, unless told otherwise.

For example, imagine you want to reference `priv/static/images/bg.png`, served at `/images/bg.png`, from your CSS file:

```
body {
  background-image: url(/images/bg.png);
```

```
}
```

The above may fail with the following message:

```
error: Could not resolve "/images/bg.png" (mark it as
external to exclude it from the bundle)
```

Given the images are already managed by Phoenix, you need to mark all resources from `/images` (and also `/fonts`) as external, as the error message says. This is what Phoenix does by default for new apps since v1.6.1+. In your `config/config.exs`, you will find:

```
args: ~w(js/app.js --bundle --target=es2017 --
outdir=../priv/static/assets --external:/fonts/* --
external:/images/*),
```

If you need to reference other directories, you need to update the arguments above accordingly. Note running `mix phx.digest` will create digested files for all of the assets in `priv/static`, so your images and fonts are still cache-busted.

Esbuild plugins

Phoenix's default configuration of `esbuild` (via the Elixir wrapper) does not allow you to use esbuild plugins. If you want to use an esbuild plugin, for example to compile SASS files to CSS, you can replace the default build system with a custom build script.

The following is an example of a custom build using esbuild via Node.JS. First of all, you'll need to install Node.js in development and make it available for your production build step.

Then you'll need to add `esbuild` to your Node.js packages and the Phoenix packages. Inside the `assets` directory, run:

```
$ npm install esbuild --save-dev
$ npm install ../deps/phoenix ../deps/phoenix_html
../deps/phoenix_live_view --save
```

or, for Yarn:

```
$ yarn add --dev esbuild
$ yarn add ../deps/phoenix ../deps/phoenix_html
../deps/phoenix_live_view
```

Next, add a custom JavaScript build script. We'll call the example `assets/build.js`:

```
const esbuild = require("esbuild");

const args = process.argv.slice(2);
const watch = args.includes('--watch');
const deploy = args.includes('--deploy');

const loader = {
  // Add loaders for images/fonts/etc, e.g. { '.svg':
'file' }
};

const plugins = [
  // Add and configure plugins here
];

// Define esbuild options
let opts = {
  entryPoints: ["js/app.js"],
  bundle: true,
  logLevel: "info",
  target: "es2017",
  outdir: "../priv/static/assets",
  external: ["*.css", "fonts/*", "images/*"],
  nodePaths: ["../deps"],
  loader: loader,
  plugins: plugins,
};
```

```
if (deploy) {
  opts = {
    ...opts,
    minify: true,
  };
}

if (watch) {
  opts = {
    ...opts,
    sourcemap: "inline",
  };
  esbuild
    .context(opts)
    .then((ctx) => {
      ctx.watch();
    })
    .catch((_error) => {
      process.exit(1);
    });
} else {
  esbuild.build(opts);
}
```

This script covers following use cases:

- `node build.js`: builds for development & testing (useful on CI)
- `node build.js --watch`: like above, but watches for changes continuously
- `node build.js --deploy`: builds minified assets for production

Modify `config/dev.exs` so that the script runs whenever you change files, replacing the existing `:esbuild` configuration under `watchers`:

```
config :hello, HelloWeb.Endpoint,
  ...
  watchers: [
    node: ["build.js", "--watch", cd:
Path.expand("../assets", __DIR__)]
```

```
    ],
    ...
```

Modify the `aliases` task in `mix.exs` to install `npm` packages during `mix setup` and use the new `esbuild` on `mix assets.deploy`:

```
  defp aliases do
    [
      setup: ["deps.get", "ecto.setup", "cmd --cd assets
npm install"],
      ...,
      "assets.deploy": ["cmd --cd assets node build.js --
deploy", "phx.digest"]
    ]
  end
```

Finally, remove the `esbuild` configuration from `config/config.exs` and remove the dependency from the `deps` function in your `mix.exs`, and you are done!

Alternative JS build tools

If you are writing an API or you want to use another asset build tool, you may want to remove the `esbuild` Hex package (see steps below). Then you must follow the additional steps required by the third-party tool.

Remove esbuild

1. Remove the `esbuild` configuration in `config/config.exs` and `config/dev.exs`,
2. Remove the `assets.deploy` task defined in `mix.exs`,
3. Remove the `esbuild` dependency from `mix.exs`,
4. Unlock the `esbuild` dependency:

```
$ mix deps.unlock esbuild
```

Alternative CSS frameworks

By default, Phoenix generates CSS with the `tailwind` library and its default plugins.

If you want to use external `tailwind` plugins or another CSS framework, you should replace the `tailwind` Hex package (see steps below). Then you can use an `esbuild` plugin (as outlined above) or even bring a separate framework altogether.

Remove tailwind

1. Remove the `tailwind` configuration in `config/config.exs` and `config/dev.exs`,
2. Remove the `assets.deploy` task defined in `mix.exs`,
3. Remove the `tailwind` dependency from `mix.exs`,
4. Unlock the `tailwind` dependency:

```
$ mix deps.unlock tailwind
```

You may optionally remove and delete the `heroicons` dependency as well.

mix phx.gen.auth

The `mix phx.gen.auth` command generates a flexible, pre-built authentication system into your Phoenix app. This generator allows you to quickly move past the task of adding authentication to your codebase and stay focused on the real-world problem your application is trying to solve.

Getting started

> Before running this command, consider committing your work as it generates multiple files.

Let's start by running the following command from the root of our app (or `apps/my_app_web` in an umbrella app):

```
$ mix phx.gen.auth Accounts User users

An authentication system can be created in two different
ways:
- Using Phoenix.LiveView (default)
- Using Phoenix.Controller only

Do you want to create a LiveView based authentication
system? [Y/n] Y
```

The authentication generators support Phoenix LiveView, for enhanced UX, so we'll answer `Y` here. You may also answer `n` for a controller based authentication system.

Either approach will create an `Accounts` context with an `Accounts.User` schema module. The final argument is the plural version of the schema module, which is used for generating database table names and route paths. The `mix phx.gen.auth` generator is similar to `mix phx.gen.html` except

it does not accept a list of additional fields to add to the schema, and it generates many more context functions.

Since this generator installed additional dependencies in `mix.exs`, let's fetch those:

```
$ mix deps.get
```

Now we need to verify the database connection details for the development and test environments in `config/` so the migrator and tests can run properly. Then run the following to create the database:

```
$ mix ecto.setup
```

Let's run the tests to make sure our new authentication system works as expected.

```
$ mix test
```

And finally, let's start our Phoenix server and try it out.

```
$ mix phx.server
```

Developer responsibilities

Since Phoenix generates this code into your application instead of building these modules into Phoenix itself, you now have complete freedom to modify the authentication system, so it works best with your use case. The one caveat with using a generated authentication system is it will not be updated after it's been generated. Therefore, as improvements are made to the output of `mix phx.gen.auth`, it becomes your responsibility to determine if these changes need to be ported into your application. Security-related and other important

improvements will be explicitly and clearly marked in the `CHANGELOG.md` file and upgrade notes.

Generated code

The following are notes about the generated authentication system.

Password hashing

The password hashing mechanism defaults to `bcrypt` for Unix systems and `pbkdf2` for Windows systems. Both systems use the Comeonin interface.

The password hashing mechanism can be overridden with the `--hashing-lib` option. The following values are supported:

- `bcrypt` - bcrypt_elixir
- `pbkdf2` - pbkdf2_elixir
- `argon2` - argon2_elixir

We recommend developers to consider using `argon2`, which is the most robust of all 3. The downside is that `argon2` is quite CPU and memory intensive, and you will need more powerful instances to run your applications on.

For more information about choosing these libraries, see the Comeonin project.

Forbidding access

The generated code ships with an authentication module with a handful of plugs that fetch the current user, require authentication and so on. For instance, in an app named Demo which had `mix phx.gen.auth Accounts User users` run on it, you will find a module named `DemoWeb.UserAuth` with plugs such as:

- `fetch_current_user` - fetches the current user information if available
- `require_authenticated_user` - must be invoked after `fetch_current_user` and requires that a current user exists and is authenticated
- `redirect_if_user_is_authenticated` - used for the few pages that must not be available to authenticated users

Confirmation

The generated functionality ships with an account confirmation mechanism, where users have to confirm their account, typically by email. However, the generated code does not forbid users from using the application if their accounts have not yet been confirmed. You can add this functionality by customizing the `require_authenticated_user` in the `Auth` module to check for the `confirmed_at` field (and any other property you desire).

Notifiers

The generated code is not integrated with any system to send SMSes or emails for confirming accounts, resetting passwords, etc. Instead, it simply logs a message to the terminal. It is your responsibility to integrate with the proper system after generation.

Note that if you generated your Phoenix project with `mix phx.new`, your project is configured to use Swoosh mailer by default. To view notifier emails during development with Swoosh, navigate to `/dev/mailbox`.

Tracking sessions

All sessions and tokens are tracked in a separate table. This allows you to track how many sessions are active for each account. You could even expose this information to users if desired.

Note that whenever the password changes (either via reset password or directly), all tokens are deleted, and the user has to log in again on all

devices.

User Enumeration attacks

A user enumeration attack allows someone to check if an email is registered in the application. The generated authentication code does not attempt to protect from such checks. For instance, when you register an account, if the email is already registered, the code will notify the user the email is already registered.

If your application is sensitive to enumeration attacks, you need to implement your own workflows, which tends to be very different from most applications, as you need to carefully balance security and user experience.

Furthermore, if you are concerned about enumeration attacks, beware of timing attacks too. For example, registering a new account typically involves additional work (such as writing to the database, sending emails, etc) compared to when an account already exists. Someone could measure the time taken to execute those additional tasks to enumerate emails. This applies to all endpoints (registration, confirmation, password recovery, etc.) that may send email, in-app notifications, etc.

Case sensitiveness

The email lookup is made to be case-insensitive. Case-insensitive lookups are the default in MySQL and MSSQL. In SQLite3 we use `COLLATE NOCASE` in the column definition to support it. In PostgreSQL, we use the `citext` extension.

Note `citext` is part of PostgreSQL itself and is bundled with it in most operating systems and package managers. `mix phx.gen.auth` takes care of creating the extension and no extra work is necessary in the majority of cases. If by any chance your package manager splits `citext` into a separate package, you will get an error while migrating, and you can most likely solve it by installing the `postgres-contrib` package.

Concurrent tests

The generated tests run concurrently if you are using a database that supports concurrent tests, which is the case of PostgreSQL.

More about `mix phx.gen.auth`

Check out `mix phx.gen.auth` for more details, such as using a different password hashing library, customizing the web module namespace, generating binary id type, configuring the default options, and using custom table names.

Additional resources

The following links have more information regarding the motivation and design of the code this generates.

- Berenice Medel's blog post on generating LiveViews for authentication (rather than conventional Controllers & Views) - Bringing Phoenix Authentication to Life
- José Valim's blog post - An upcoming authentication solution for Phoenix
- The original `phx_gen_auth` repo (for Phoenix 1.5 applications) - This is a great resource to see discussions around decisions that have been made in earlier versions of the project.
- Original pull request on bare Phoenix app
- Original design spec

API Authentication

> **Requirement**: This guide expects that you have gone through the `mix phx.gen.auth` guide.

This guide shows how to add API authentication on top of `mix phx.gen.auth`. Since the authentication generator already includes a token table, we use it to store API tokens too, following the best security practices.

We will break this guide in two parts: augmenting the context and the plug implementation. We will assume that the following `mix phx.gen.auth` command was executed:

```
$ mix phx.gen.auth Accounts User users
```

If you ran something else, it should be trivial to adapt the names.

Adding API functions to the context

Our authentication system will require two functions. One to create the API token and another to verify it. Open up `lib/my_app/accounts.ex` and add these two new functions:

```
  ## API

  @doc """
  Creates a new api token for a user.

  The token returned must be saved somewhere safe.
  This token cannot be recovered from the database.
  """
  def create_user_api_token(user) do
    {encoded_token, user_token} =
UserToken.build_email_token(user, "api-token")
```

```
    Repo.insert!(user_token)
    encoded_token
  end

  @doc """
  Fetches the user by API token.
  """
  def fetch_user_by_api_token(token) do
    with {:ok, query} <-
UserToken.verify_email_token_query(token, "api-token"),
         %User{} = user <- Repo.one(query) do
      {:ok, user}
    else
      _ -> :error
    end
  end
```

The new functions use the existing `UserToken` functionality to store a new type of token called "api-token". Because this is an email token, if the user changes their email, the tokens will be expired.

Also notice we called the second function `fetch_user_by_api_token`, instead of `get_user_by_api_token`. Because we want to render different status codes in our API, depending if a user was found or not, we return `{:ok, user}` or `:error`. Elixir's convention is to call these functions `fetch_*`, instead of `get_*` which would usually return `nil` instead of tuples.

To make sure our new functions work, let's write tests. Open up `test/my_app/accounts_test.exs` and add this new describe block:

```
  describe "create_user_api_token/1 and
fetch_user_by_api_token/1" do
    test "creates and fetches by token" do
      user = user_fixture()
      token = Accounts.create_user_api_token(user)
      assert Accounts.fetch_user_by_api_token(token) ==
{:ok, user}
      assert Accounts.fetch_user_by_api_token("invalid")
== :error
```

```
    end
  end
```

If you run the tests, they will actually fail. Something similar to this:

```
1) test create_user_api_token/1 and
fetch_user_by_api_token/1 creates and verify token
(Demo.AccountsTest)
   test/demo/accounts_test.exs:21
   ** (FunctionClauseError) no function clause matching
in Demo.Accounts.UserToken.days_for_context/1

   The following arguments were given to
Demo.Accounts.UserToken.days_for_context/1:

       # 1
       "api-token"

   Attempted function clauses (showing 2 out of 2):

       defp days_for_context("confirm")
       defp days_for_context("reset_password")

   code: assert Accounts.verify_api_token(token) == {:ok,
user}
   stacktrace:
     (demo 0.1.0) lib/demo/accounts/user_token.ex:129:
Demo.Accounts.UserToken.days_for_context/1
     (demo 0.1.0) lib/demo/accounts/user_token.ex:114:
Demo.Accounts.UserToken.verify_email_token_query/2
     (demo 0.1.0) lib/demo/accounts.ex:301:
Demo.Accounts.verify_api_token/1
     test/demo/accounts_test.exs:24: (test)
```

If you prefer, try looking at the error and fixing it yourself. The explanation will come next.

The `UserToken` module expects us to declare the validity of each token and we haven't defined one for "api-token". The length is going to depend on your

application and how sensitive it is in terms of security. For this example, let's say the token is valid for 365 days.

Open up `lib/my_app/accounts/user_token.ex`, find where `defp days_for_context` is defined, and add a new clause, like this:

```
    defp days_for_context("api-token"), do: 365
    defp days_for_context("confirm"), do:
  @confirm_validity_in_days
    defp days_for_context("reset_password"), do:
  @reset_password_validity_in_days
```

Now tests should pass and we are ready to move forward!

API authentication plug

The last part is to add authentication to our API.

When we ran `mix phx.gen.auth`, it generated a `MyAppWeb.UserAuth` module with several plugs, which are small functions that receive the `conn` and customize our request/response life-cycle. Open up `lib/my_app_web/user_auth.ex` and add this new function:

```
  def fetch_api_user(conn, _opts) do
    with ["Bearer " <> token] <- get_req_header(conn,
  "authorization"),
         {:ok, user} <-
  Accounts.fetch_user_by_api_token(token) do
      assign(conn, :current_user, user)
    else
      _ ->
        conn
        |> send_resp(:unauthorized, "No access for you")
        |> halt()
    end
  end
```

Our function receives the connection and checks if the "authorization" header has been set with "Bearer TOKEN", where "TOKEN" is the value returned by `Accounts.create_user_api_token/1`. In case the token is not valid or there is no such user, we abort the request.

Finally, we need to add this `plug` to our pipeline. Open up `lib/my_app_web/router.ex` and you will find a pipeline for API. Let's add our new plug under it, like this:

```
pipeline :api do
  plug :accepts, ["json"]
  plug :fetch_api_user
end
```

Now you are ready to receive and validate API requests. Feel free to open up `test/my_app_web/user_auth_test.exs` and write your own test. You can use the tests for other plugs as templates!

Your turn

The overall API authentication flow will depend on your application.

If you want to use this token in a JavaScript client, you will need to slightly alter the `UserSessionController` to invoke `Accounts.create_user_api_token/1` and return a JSON response and include the token returned it.

If you want to provide APIs for 3rd-party users, you will need to allow them to create tokens, and show the result of `Accounts.create_user_api_token/1` to them. They must save these tokens somewhere safe and include them as part of their requests using the "authorization" header.

Channels

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.

Channels are an exciting part of Phoenix that enable soft real-time communication with and between millions of connected clients.

Some possible use cases include:

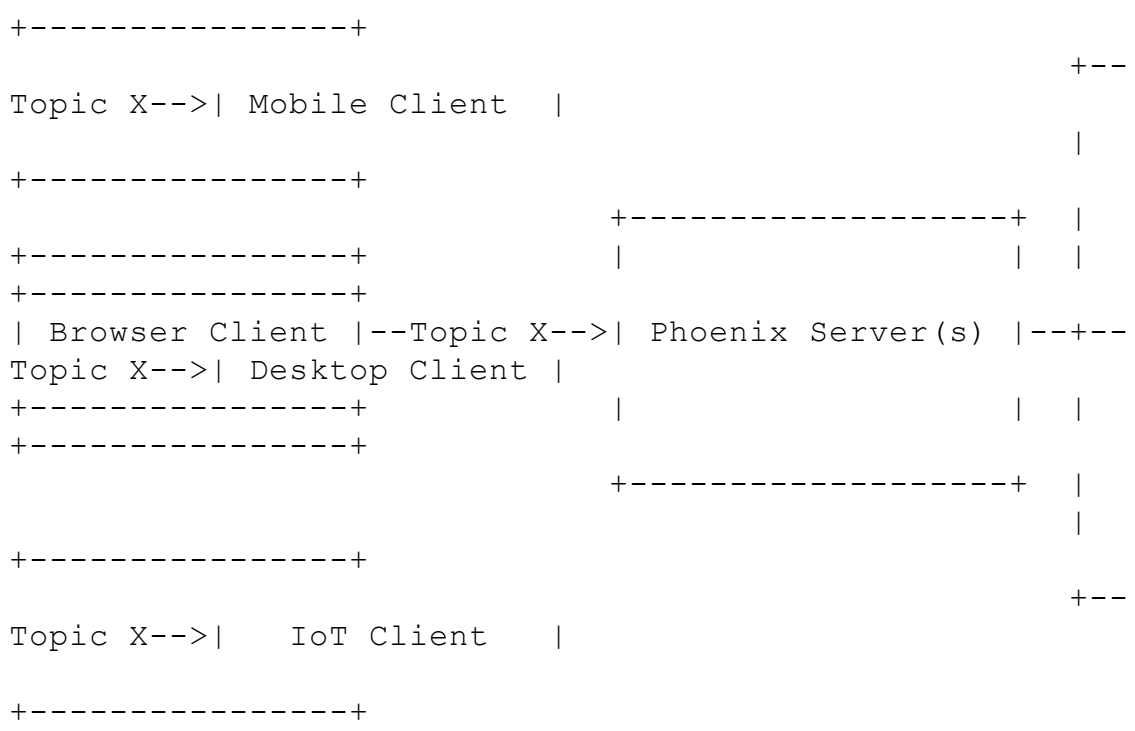
- Chat rooms and APIs for messaging apps
- Breaking news, like "a goal was scored" or "an earthquake is coming"
- Tracking trains, trucks, or race participants on a map
- Events in multiplayer games
- Monitoring sensors and controlling lights
- Notifying a browser that a page's CSS or JavaScript has changed (this is handy in development)

Conceptually, Channels are pretty simple.

First, clients connect to the server using some transport, like WebSocket. Once connected, they join one or more topics. For example, to interact with a public chat room clients may join a topic called `public_chat`, and to receive updates from a product with ID 7, they may need to join a topic called `product_updates:7`.

Clients can push messages to the topics they've joined, and can also receive messages from them. The other way around, Channel servers receive messages from their connected clients, and can push messages to them too.

Servers are able to broadcast messages to all clients subscribed to a certain topic. This is illustrated in the following diagram:



Broadcasts work even if the application runs on several nodes/computers. That is, if two clients have their socket connected to different application nodes and are subscribed to the same topic `T`, both of them will receive messages broadcasted to `T`. That is possible thanks to an internal PubSub mechanism.

Channels can support any kind of client: a browser, native app, smart watch, embedded device, or anything else that can connect to a network. All the client needs is a suitable library; see the Client Libraries section below. Each client library communicates using one of the "transports" that Channels understand. Currently, that's either Websockets or long polling, but other transports may be added in the future.

Unlike stateless HTTP connections, Channels support long-lived connections, each backed by a lightweight BEAM process, working in parallel and maintaining its own state.

This architecture scales well; Phoenix Channels can support millions of subscribers with reasonable latency on a single box, passing hundreds of thousands of messages per second. And that capacity can be multiplied by adding more nodes to the cluster.

The Moving Parts

Although Channels are simple to use from a client perspective, there are a number of components involved in routing messages to clients across a cluster of servers. Let's take a look at them.

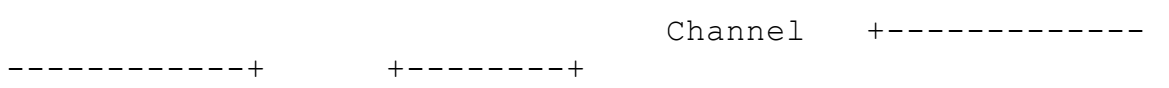
Overview

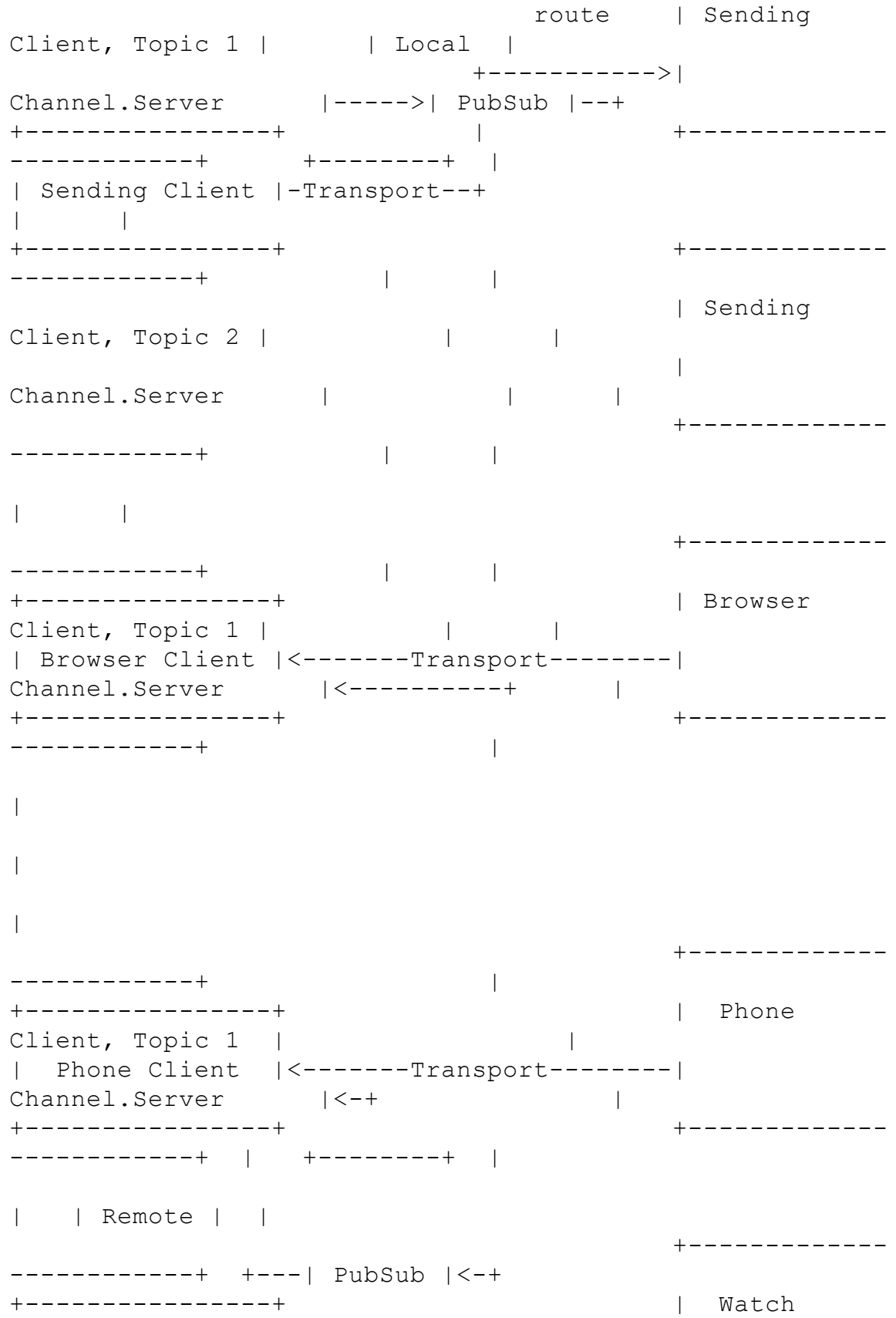
To start communicating, a client connects to a node (a Phoenix server) using a transport (e.g., Websockets or long polling) and joins one or more channels using that single network connection. One channel server lightweight process is created per client, per topic. Each channel holds onto the `%Phoenix.Socket{}` and can maintain any state it needs within its `socket.assigns`.

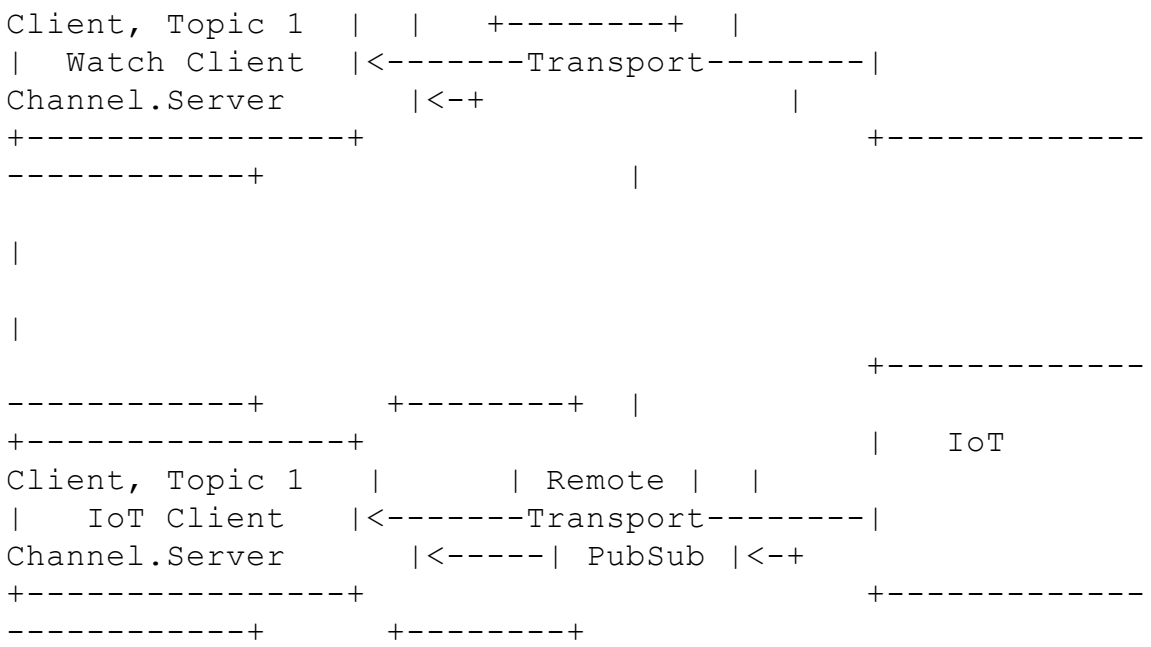
Once the connection is established, each incoming message from a client is routed, based on its topic, to the correct channel server. If the channel server asks to broadcast a message, that message is sent to the local PubSub, which sends it out to any clients connected to the same server and subscribed to that topic.

If there are other nodes in the cluster, the local PubSub also forwards the message to their PubSubs, which send it out to their own subscribers. Because only one message has to be sent per additional node, the performance cost of adding nodes is negligible, while each new node supports many more subscribers.

The message flow looks something like this:




route
Sending Client, Topic 1
Local
Channel.Server
PubSub
Sending Client
Transport
Sending Client, Topic 2
Channel.Server
Browser Client, Topic 1
Browser Client
Transport
Channel.Server
Phone Client, Topic 1
Phone Client
Transport
Channel.Server
Remote
PubSub
Watch



Endpoint

In your Phoenix app's `Endpoint` module, a `socket` declaration specifies which socket handler will receive connections on a given URL.

```
socket "/socket", HelloWeb.UserSocket,
  websocket: true,
  longpoll: false
```

Phoenix comes with two default transports: websocket and longpoll. You can configure them directly via the `socket` declaration.

Socket Handlers

On the client side, you will establish a socket connection to the route above:

```
let socket = new Socket("/socket", {params: {token:
window.userToken}})
```

On the server, Phoenix will invoke `HelloWeb.UserSocket.connect/2`, passing your parameters and the initial socket state. Within the socket, you can authenticate and identify a socket connection and set default socket assigns. The socket is also where you define your channel routes.

Channel Routes

Channel routes match on the topic string and dispatch matching requests to the given Channel module.

The star character `*` acts as a wildcard matcher, so in the following example route, requests for `room:lobby` and `room:123` would both be dispatched to the `RoomChannel`. In your `UserSocket`, you would have:

```
channel "room:*", HelloWeb.RoomChannel
```

Channels

Channels handle events from clients, so they are similar to Controllers, but there are two key differences. Channel events can go both directions - incoming and outgoing. Channel connections also persist beyond a single request/response cycle. Channels are the highest level abstraction for real-time communication components in Phoenix.

Each Channel will implement one or more clauses of each of these four callback functions - `join/3`, `terminate/2`, `handle_in/3`, and `handle_out/3`.

Topics

Topics are string identifiers - names that the various layers use in order to make sure messages end up in the right place. As we saw above, topics can use wildcards. This allows for a useful `"topic:subtopic"` convention. Often, you'll compose topics using record IDs from your application layer, such as `"users:123"`.

Messages

The `Phoenix.Socket.Message` module defines a struct with the following keys which denotes a valid message. From the Phoenix.Socket.Message docs.

- `topic` - The string topic or `"topic:subtopic"` pair namespace, such as `"messages"` or `"messages:123"`
- `event` - The string event name, for example `"phx_join"`
- `payload` - The message payload
- `ref` - The unique string ref

PubSub

PubSub is provided by the `Phoenix.PubSub` module. Interested parties can receive events by subscribing to topics. Other processes can broadcast events to certain topics.

This is useful to broadcast messages on channel and also for application development in general. For instance, letting all connected live views to know that a new comment has been added to a post.

The PubSub system takes care of getting messages from one node to another so that they can be sent to all subscribers across the cluster. By default, this is done using Phoenix.PubSub.PG2, which uses native BEAM messaging.

If your deployment environment does not support distributed Elixir or direct communication between servers, Phoenix also ships with a Redis Adapter that uses Redis to exchange PubSub data. Please see the Phoenix.PubSub docs for more information.

Client Libraries

Any networked device can connect to Phoenix Channels as long as it has a client library. The following libraries exist today, and new ones are always

welcome; to write your own, see our how-to guide Writing a Channels Client.

Official

Phoenix ships with a JavaScript client that is available when generating a new Phoenix project. The documentation for the JavaScript module is available at https://hexdocs.pm/phoenix/js/; the code is in multiple js files.

3rd Party

- Swift (iOS)
  - SwiftPhoenix
- Java (Android)
  - JavaPhoenixChannels
- Kotlin (Android)
  - JavaPhoenixClient
- C#
  - PhoenixSharp
- Elixir
  - phoenix_gen_socket_client
  - slipstream
- GDScript (Godot Game Engine)
  - GodotPhoenixChannels

Tying it all together

Let's tie all these ideas together by building a simple chat application. Make sure you created a new Phoenix application and now we are ready to generate the `UserSocket`.

Generating a socket

Let's invoke the socket generator to get started:

```
$ mix phx.gen.socket User
```

It will create two files, the client code in `assets/js/user_socket.js` and the server counter-part in `lib/hello_web/channels/user_socket.ex`. After running, the generator will also ask to add the following line to `lib/hello_web/endpoint.ex`:

```
defmodule HelloWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :hello

  socket "/socket", HelloWeb.UserSocket,
    websocket: true,
    longpoll: false

  ...
end
```

The generator also asks us to import the client code, we will do that later.

Next, we will configure our socket to ensure messages get routed to the correct channel. To do that, we'll uncomment the `"room:*"` channel definition:

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

  ## Channels
  channel "room:*", HelloWeb.RoomChannel
  ...
```

Now, whenever a client sends a message whose topic starts with `"room:"`, it will be routed to our RoomChannel. Next, we'll define a `HelloWeb.RoomChannel` module to manage our chat room messages.

Joining Channels

The first priority of your channels is to authorize clients to join a given topic. For authorization, we must implement `join/3` in `lib/hello_web/channels/room_channel.ex`.

```
defmodule HelloWeb.RoomChannel do
  use Phoenix.Channel

  def join("room:lobby", _message, socket) do
    {:ok, socket}
  end

  def join("room:" <> _private_room_id, _params, _socket)
do
    {:error, %{reason: "unauthorized"}}
  end
end
```

For our chat app, we'll allow anyone to join the `"room:lobby"` topic, but any other room will be considered private and special authorization, say from a database, will be required. (We won't worry about private chat rooms for this exercise, but feel free to explore after we finish.)

With our channel in place, let's get the client and server talking.

The generated `assets/js/user_socket.js` defines a simple client based on the socket implementation that ships with Phoenix.

We can use that library to connect to our socket and join our channel, we just need to set our room name to `"room:lobby"` in that file.

```
// assets/js/user_socket.js
// ...
socket.connect()

// Now that you are connected, you can join channels with
a topic:
let channel = socket.channel("room:lobby", {})
channel.join()
  .receive("ok", resp => { console.log("Joined
```

```
successfully", resp) })
  .receive("error", resp => { console.log("Unable to
join", resp) })

export default socket
```

After that, we need to make sure `assets/js/user_socket.js` gets imported into our application JavaScript file. To do that, uncomment this line in `assets/js/app.js`.

```
// ...
import "./user_socket.js"
```

Save the file and your browser should auto refresh, thanks to the Phoenix live reloader. If everything worked, we should see "Joined successfully" in the browser's JavaScript console. Our client and server are now talking over a persistent connection. Now let's make it useful by enabling chat.

In `lib/hello_web/controllers/page_html/home.html.heex`, we'll replace the existing code with a container to hold our chat messages, and an input field to send them:

```
<div id="messages" role="log" aria-live="polite"></div>
<input id="chat-input" type="text">
```

Now let's add a couple of event listeners to `assets/js/user_socket.js`:

```
// ...
let channel           = socket.channel("room:lobby", {})
let chatInput         = document.querySelector("#chat-
input")
let messagesContainer =
document.querySelector("#messages")

chatInput.addEventListener("keypress", event => {
  if(event.key === 'Enter'){
    channel.push("new_msg", {body: chatInput.value})
```

```
    chatInput.value = ""
  }
})

channel.join()
  .receive("ok", resp => { console.log("Joined
successfully", resp) })
  .receive("error", resp => { console.log("Unable to
join", resp) })

export default socket
```

All we had to do is detect that enter was pressed and then `push` an event over the channel with the message body. We named the event `"new_msg"`. With this in place, let's handle the other piece of a chat application, where we listen for new messages and append them to our messages container.

```
// ...
let channel           = socket.channel("room:lobby", {})
let chatInput         = document.querySelector("#chat-
input")
let messagesContainer =
document.querySelector("#messages")

chatInput.addEventListener("keypress", event => {
  if(event.key === 'Enter'){
    channel.push("new_msg", {body: chatInput.value})
    chatInput.value = ""
  }
})

channel.on("new_msg", payload => {
  let messageItem = document.createElement("p")
  messageItem.innerText = `[${Date()}] ${payload.body}`
  messagesContainer.appendChild(messageItem)
})

channel.join()
  .receive("ok", resp => { console.log("Joined
successfully", resp) })
  .receive("error", resp => { console.log("Unable to
join", resp) })
```

```
export default socket
```

We listen for the `"new_msg"` event using `channel.on`, and then append the message body to the DOM. Now let's handle the incoming and outgoing events on the server to complete the picture.

Incoming Events

We handle incoming events with `handle_in/3`. We can pattern match on the event names, like `"new_msg"`, and then grab the payload that the client passed over the channel. For our chat application, we simply need to notify all other `room:lobby` subscribers of the new message with `broadcast!/3`.

```
defmodule HelloWeb.RoomChannel do
  use Phoenix.Channel

  def join("room:lobby", _message, socket) do
    {:ok, socket}
  end

  def join("room:" <> _private_room_id, _params, _socket)
do
    {:error, %{reason: "unauthorized"}}
  end

  def handle_in("new_msg", %{"body" => body}, socket) do
    broadcast!(socket, "new_msg", %{body: body})
    {:noreply, socket}
  end
end
```

`broadcast!/3` will notify all joined clients on this `socket`'s topic and invoke their `handle_out/3` callbacks. `handle_out/3` isn't a required callback, but it allows us to customize and filter broadcasts before they reach each client. By default, `handle_out/3` is implemented for us and simply pushes the message on to the client. Hooking into outgoing events allows for powerful message customization and filtering. Let's see how.

Intercepting Outgoing Events

We won't implement this for our application, but imagine our chat app allowed users to ignore messages about new users joining a room. We could implement that behavior like this, where we explicitly tell Phoenix which outgoing event we want to intercept and then define a `handle_out/3` callback for those events. (Of course, this assumes that we have an `Accounts` context with an `ignoring_user?/2` function, and that we pass a user in via the `assigns` map). It is important to note that the `handle_out/3` callback will be called for every recipient of a message, so more expensive operations like hitting the database should be considered carefully before being included in `handle_out/3`.

```
intercept ["user_joined"]

def handle_out("user_joined", msg, socket) do
  if Accounts.ignoring_user?(socket.assigns[:user],
msg.user_id) do
    {:noreply, socket}
  else
    push(socket, "user_joined", msg)
    {:noreply, socket}
  end
end
```

That's all there is to our basic chat app. Fire up multiple browser tabs and you should see your messages being pushed and broadcasted to all windows!

Using Token Authentication

When we connect, we'll often need to authenticate the client. Fortunately, this is a 4-step process with Phoenix.Token.

Step 1 - Assign a Token in the Connection

Let's say we have an authentication plug in our app called `OurAuth`. When `OurAuth` authenticates a user, it sets a value for the `:current_user` key in `conn.assigns`. Since the `current_user` exists, we can simply assign the user's token in the connection for use in the layout. We can wrap that behavior up in a private function plug, `put_user_token/2`. This could also be put in its own module as well. To make this all work, we just add `OurAuth` and `put_user_token/2` to the browser pipeline.

```
pipeline :browser do
  ...
  plug OurAuth
  plug :put_user_token
end

defp put_user_token(conn, _) do
  if current_user = conn.assigns[:current_user] do
    token = Phoenix.Token.sign(conn, "user socket",
current_user.id)
    assign(conn, :user_token, token)
  else
    conn
  end
end
```

Now our `conn.assigns` contains the `current_user` and `user_token`.

Step 2 - Pass the Token to the JavaScript

Next, we need to pass this token to JavaScript. We can do so inside a script tag in `lib/hello_web/components/layouts/app.html.heex` right above the app.js script, as follows:

```
<script>window.userToken = "<%= assigns[:user_token] %>";
</script>
<script src={~p"/assets/app.js"}></script>
```

Step 3 - Pass the Token to the Socket Constructor and Verify

We also need to pass the `:params` to the socket constructor and verify the user token in the `connect/3` function. To do so, edit `lib/hello_web/channels/user_socket.ex`, as follows:

```
def connect(%{"token" => token}, socket, _connect_info)
do
  # max_age: 1209600 is equivalent to two weeks in
seconds
  case Phoenix.Token.verify(socket, "user socket", token,
max_age: 1209600) do
    {:ok, user_id} ->
      {:ok, assign(socket, :current_user, user_id)}
    {:error, reason} ->
      :error
  end
end
```

In our JavaScript, we can use the token set previously when constructing the Socket:

```
let socket = new Socket("/socket", {params: {token:
window.userToken}})
```

We used `Phoenix.Token.verify/4` to verify the user token provided by the client. `Phoenix.Token.verify/4` returns either `{:ok, user_id}` or `{:error, reason}`. We can pattern match on that return in a `case` statement. With a verified token, we set the user's id as the value to `:current_user` in the socket. Otherwise, we return `:error`.

Step 4 - Connect to the socket in JavaScript

With authentication set up, we can connect to sockets and channels from JavaScript.

```
let socket = new Socket("/socket", {params: {token:
window.userToken}})
socket.connect()
```

Now that we are connected, we can join channels with a topic:

```
let channel = socket.channel("topic:subtopic", {})
channel.join()
  .receive("ok", resp => { console.log("Joined
successfully", resp) })
  .receive("error", resp => { console.log("Unable to
join", resp) })

export default socket
```

Note that token authentication is preferable since it's transport agnostic and well-suited for long running-connections like channels, as opposed to using sessions or other authentication approaches.

Fault Tolerance and Reliability Guarantees

Servers restart, networks split, and clients lose connectivity. In order to design robust systems, we need to understand how Phoenix responds to these events and what guarantees it offers.

Handling Reconnection

Clients subscribe to topics, and Phoenix stores those subscriptions in an in-memory ETS table. If a channel crashes, the clients will need to reconnect to the topics they had previously subscribed to. Fortunately, the Phoenix JavaScript client knows how to do this. The server will notify all the clients of the crash. This will trigger each client's `Channel.onError` callback. The clients will attempt to reconnect to the server using an exponential backoff strategy. Once they reconnect, they'll attempt to rejoin the topics they had previously subscribed to. If they are successful, they'll start receiving messages from those topics as before.

Resending Client Messages

Channel clients queue outgoing messages into a `PushBuffer`, and send them to the server when there is a connection. If no connection is available, the client holds on to the messages until it can establish a new connection. With no connection, the client will hold the messages in memory until it establishes a connection, or until it receives a `timeout` event. The default timeout is set to 5000 milliseconds. The client won't persist the messages in the browser's local storage, so if the browser tab closes, the messages will be gone.

Resending Server Messages

Phoenix uses an at-most-once strategy when sending messages to clients. If the client is offline and misses the message, Phoenix won't resend it. Phoenix doesn't persist messages on the server. If the server restarts, unsent messages will be gone. If our application needs stronger guarantees around message delivery, we'll need to write that code ourselves. Common approaches involve persisting messages on the server and having clients request missing messages. For an example, see Chris McCord's Phoenix training: client code and server code.

Example Application

To see an example of the application we just built, checkout the project phoenix_chat_example.

Presence

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Channels guide.

Phoenix Presence is a feature which allows you to register process information on a topic and replicate it transparently across a cluster. It's a combination of both a server-side and client-side library, which makes it simple to implement. A simple use-case would be showing which users are currently online in an application.

Phoenix Presence is special for a number of reasons. It has no single point of failure, no single source of truth, relies entirely on the standard library with no operational dependencies and self-heals.

Setting up

We are going to use Presence to track which users are connected on the server and send updates to the client as users join and leave. We will deliver those updates via Phoenix Channels. Therefore, let's create a `RoomChannel`, as we did in the channels guides:

```
$ mix phx.gen.channel Room
```

Follow the steps after the generator and you are ready to start tracking presence.

The Presence generator

To get started with Presence, we'll first need to generate a presence module. We can do this with the `mix phx.gen.presence` task:

```
$ mix phx.gen.presence
* creating lib/hello_web/channels/presence.ex

Add your new module to your supervision tree,
in lib/hello/application.ex:

    children = [
      ...
      HelloWeb.Presence,
    ]

You're all set! See the Phoenix.Presence docs for more
details:
https://hexdocs.pm/phoenix/Phoenix.Presence.html
```

If we open up the `lib/hello_web/channels/presence.ex` file, we will see the following line:

```
use Phoenix.Presence,
  otp_app: :hello,
  pubsub_server: Hello.PubSub
```

This sets up the module for presence, defining the functions we require for tracking presences. As mentioned in the generator task, we should add this module to our supervision tree in `application.ex`:

```
children = [
  ...
  HelloWeb.Presence,
]
```

Usage With Channels and JavaScript

Next, we will create the channel that we'll communicate presence over. After a user joins, we can push the list of presences down the channel and then track the connection. We can also provide a map of additional information to track.

```
defmodule HelloWeb.RoomChannel do
  use Phoenix.Channel
  alias HelloWeb.Presence

  def join("room:lobby", %{"name" => name}, socket) do
    send(self(), :after_join)
    {:ok, assign(socket, :name, name)}
  end

  def handle_info(:after_join, socket) do
    {:ok, _} =
      Presence.track(socket, socket.assigns.name, %{
        online_at: inspect(System.system_time(:second))
      })

    push(socket, "presence_state", Presence.list(socket))
    {:noreply, socket}
  end
end
```

Finally, we can use the client-side Presence library included in `phoenix.js` to manage the state and presence diffs that come down the socket. It listens for the `"presence_state"` and `"presence_diff"` events and provides a simple callback for you to handle the events as they happen, with the `onSync` callback.

The `onSync` callback allows you to easily react to presence state changes, which most often results in re-rendering an updated list of active users. You can use the `list` method to format and return each individual presence based on the needs of your application.

To iterate users, we use the `presences.list()` function which accepts a callback. The callback will be called for each presence item with 2 arguments, the presence id and a list of metas (one for each presence for that

presence id). We use this to display the users and the number of devices they are online with.

We can see presence working by adding the following to `assets/js/app.js`:

```
import {Socket, Presence} from "phoenix"

let socket = new Socket("/socket", {params: {token:
window.userToken}})
let channel = socket.channel("room:lobby", {name:
window.location.search.split("=")[1]})
let presence = new Presence(channel)

function renderOnlineUsers(presence) {
  let response = ""

  presence.list((id, {metas: [first, ...rest]}) => {
    let count = rest.length + 1
    response += `<br>${id} (count: ${count})</br>`
  })

  document.querySelector("main").innerHTML = response
}

socket.connect()

presence.onSync(() => renderOnlineUsers(presence))

channel.join()
```

We can ensure this is working by opening 3 browser tabs. If we navigate to http://localhost:4000/?name=Alice on two browser tabs and http://localhost:4000/?name=Bob then we should see:

```
Alice (count: 2)
Bob (count: 1)
```

If we close one of the Alice tabs, then the count should decrease to 1. If we close another tab, the user should disappear from the list entirely.

Making it safe

In our initial implementation, we are passing the name of the user as part of the URL. However, in many systems, you want to allow only logged in users to access the presence functionality. To do so, you should set up token authentication, as detailed in the token authentication section of the channels guide.

With token authentication, you should access `socket.assigns.user_id`, set in `UserSocket`, instead of `socket.assigns.name` set from parameters.

Usage With LiveView

Whilst Phoenix does ship with a JavaScript API for dealing with presence, it is also possible to extend the `HelloWeb.Presence` module to support LiveView.

One thing to keep in mind when dealing with LiveView, is that each LiveView is a stateful process, so if we keep the presence state in the LiveView, each LiveView process will contain the full list of online users in memory. Instead, we can keep track of the online users within the `Presence` process, and pass separate events to the LiveView, which can use a stream to update the online list.

To start with, we need to update the `lib/hello_web/channels/presence.ex` file to add some optional callbacks to the `HelloWeb.Presence` module.

Firstly, we add the `init/1` callback. This allows us to keep track of the presence state within the process.

```
  def init(_opts) do
    {:ok, %{}}
```

```
  end
```

The presence module also allows a `fetch/2` callback, this allows the data fetched from the presence to be modified, allowing us to define the shape of the response. In this case we are adding an `id` and a `user` map.

```
  def fetch(_topic, presences) do
    for {key, %{metas: [meta | metas]}} <- presences,
into: %{} do
      # user can be populated here from the database here
we populate
      # the name for demonstration purposes
      {key, %{metas: [meta | metas], id: meta.id, user: %
{name: meta.id}}}
    end
  end
```

The final thing to add is the `handle_metas/4` callback. This callback updates the state that we keep track of in `HelloWeb.Presence` based on the user leaves and joins.

```
  def handle_metas(topic, %{joins: joins, leaves:
leaves}, presences, state) do
    for {user_id, presence} <- joins do
      user_data = %{id: user_id, user: presence.user,
metas: Map.fetch!(presences, user_id)}
      msg = {__MODULE__, {:join, user_data}}
      Phoenix.PubSub.local_broadcast(Hello.PubSub,
"proxy:#{topic}", msg)
    end

    for {user_id, presence} <- leaves do
      metas =
        case Map.fetch(presences, user_id) do
          {:ok, presence_metas} -> presence_metas
          :error -> []
        end

      user_data = %{id: user_id, user: presence.user,
metas: metas}
```

```
      msg = {__MODULE__, {:leave, user_data}}
      Phoenix.PubSub.local_broadcast(Hello.PubSub,
"proxy:#{topic}", msg)
    end

    {:ok, state}
  end
```

You can see that we are broadcasting events for the joins and leaves. These will be listened to by the LiveView process. You'll also see that we use "proxy" channel when broadcasting the joins and leaves. This is because we don't want our LiveView process to receive the presence events directly. We can add a few helper functions so that this particular implementation detail is abstracted from the LiveView module.

```
  def list_online_users(), do: list("online_users") |>
Enum.map(fn {_id, presence} -> presence end)

  def track_user(name, params), do: track(self(),
"online_users", name, params)

  def subscribe(), do:
Phoenix.PubSub.subscribe(Hello.PubSub,
"proxy:online_users")
```

Now that we have our presence module set up and broadcasting events, we can create a LiveView. Create a new file `lib/hello_web/live/online/index.ex` with the following contents:

```
defmodule HelloWeb.OnlineLive do
  use HelloWeb, :live_view

  def mount(params, _session, socket) do
    socket = stream(socket, :presences, [])
    socket =
    if connected?(socket) do
      HelloWeb.Presence.track_user(params["name"], %{id:
params["name"]})
      HelloWeb.Presence.subscribe()
```

```
      stream(socket, :presences,
HelloWeb.Presence.list_online_users())
    else
       socket
    end

    {:ok, socket}
  end

  def render(assigns) do
    ~H"""
    <ul id="online_users" phx-update="stream">
      <li :for={{dom_id, %{id: id, metas: metas}} <-
@streams.presences} id={dom_id}>{id} ({length(metas)})
</li>
    </ul>
    """
  end

  def handle_info({HelloWeb.Presence, {:join, presence}},
socket) do
    {:noreply, stream_insert(socket, :presences,
presence)}
  end

  def handle_info({HelloWeb.Presence, {:leave,
presence}}, socket) do
    if presence.metas == [] do
      {:noreply, stream_delete(socket, :presences,
presence)}
    else
      {:noreply, stream_insert(socket, :presences,
presence)}
    end
  end
end
```

If we add this route to the `lib/hello_web/router.ex`:

```
    live "/online/:name", OnlineLive, :index
```

Then we can navigate to http://localhost:4000/online/Alice in one tab, and http://localhost:4000/online/Bob in another, you'll see that the presences are tracked, along with the number of presences per user. Opening and closing tabs with various users will update the presence list in real-time.

Introduction to Testing

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.

Testing has become integral to the software development process, and the ability to easily write meaningful tests is an indispensable feature for any modern web framework. Phoenix takes this seriously, providing support files to make all the major components of the framework easy to test. It also generates test modules with real-world examples alongside any generated modules to help get us going.

Elixir ships with a built-in testing framework called ExUnit. ExUnit strives to be clear and explicit, keeping magic to a minimum. Phoenix uses ExUnit for all of its testing, and we will use it here as well.

Running tests

When Phoenix generates a web application for us, it also includes tests. To run them, simply type `mix test`:

```
$ mix test
....

Finished in 0.09 seconds
5 tests, 0 failures

Randomized with seed 652656
```

We already have five tests!

In fact, we already have a directory structure completely set up for testing, including a test helper and support files.

```
test
├── hello_web
│   └── controllers
│       ├── error_html_test.exs
│       ├── error_json_test.exs
│       └── page_controller_test.exs
├── support
│   ├── conn_case.ex
│   └── data_case.ex
└── test_helper.exs
```

The test cases we get for free include those from `test/hello_web/controllers/`. They are testing our controllers and views. If you haven't read the guides for controllers and views, now is a good time.

Understanding test modules

We are going to use the next sections to get acquainted with Phoenix testing structure. We will start with the three test files generated by Phoenix.

The first test file we'll look at is `test/hello_web/controllers/page_controller_test.exs`.

```
defmodule HelloWeb.PageControllerTest do
  use HelloWeb.ConnCase

  test "GET /", %{conn: conn} do
    conn = get(conn, ~p"/")
    assert html_response(conn, 200) =~ "Peace of mind
from prototype to production"
  end
end
```

There are a couple of interesting things happening here.

Our test files simply define modules. At the top of each module, you will find a line such as:

```
use HelloWeb.ConnCase
```

If you were to write an Elixir library, outside of Phoenix, instead of `use HelloWeb.ConnCase` you would write `use ExUnit.Case`. However, Phoenix already ships with a bunch of functionality for testing controllers and `HelloWeb.ConnCase` builds on top of `ExUnit.Case` to bring these functionalities in. We will explore the `HelloWeb.ConnCase` module soon.

Then we define each test using the `test/3` macro. The `test/3` macro receives three arguments: the test name, the testing context that we are pattern matching on, and the contents of the test. In this test, we access the root page of our application by a "GET" HTTP request on the path "/" with the `get/2` macro. Then we **assert** that the rendered page contains the string "Peace of mind from prototype to production".

When writing tests in Elixir, we use assertions to check that something is true. In our case, `assert html_response(conn, 200) =~ "Peace of mind from prototype to production"` is doing a couple things:

- It asserts that `conn` has rendered a response
- It asserts that the response has the 200 status code (which means OK in HTTP parlance)
- It asserts that the type of the response is HTML
- It asserts that the result of `html_response(conn, 200)`, which is an HTML response, has the string "Peace of mind from prototype to production" in it

However, from where does the `conn` we use on `get` and `html_response` come from? To answer this question, let's take a look at `HelloWeb.ConnCase`.

The ConnCase

If you open up `test/support/conn_case.ex`, you will find this (with comments removed):

```
defmodule HelloWeb.ConnCase do
  use ExUnit.CaseTemplate

  using do
    quote do
      # The default endpoint for testing
      @endpoint HelloWeb.Endpoint

      use HelloWeb, :verified_routes

      # Import conveniences for testing with connections
      import Plug.Conn
      import Phoenix.ConnTest
      import HelloWeb.ConnCase
    end
  end

  setup tags do
    Hello.DataCase.setup_sandbox(tags)
    {:ok, conn: Phoenix.ConnTest.build_conn()}
  end
end
```

There is a lot to unpack here.

The second line says this is a case template. This is a ExUnit feature that allows developers to replace the built-in `use ExUnit.Case` by their own case. This line is pretty much what allows us to write `use HelloWeb.ConnCase` at the top of our controller tests.

Now that we have made this module a case template, we can define callbacks that are invoked on certain occasions. The `using` callback defines code to be injected on every module that calls `use HelloWeb.ConnCase`. In this case, it starts by setting the `@endpoint` module attribute with the name of our endpoint.

Next, it wires up `:verified_routes` to allow us to use `~p` based paths in our test just like we do in the rest of our application to easily generate paths and URLs in our tests.

Finally, we import `Plug.Conn`, so all of the connection helpers available in controllers are also available in tests, and then imports `Phoenix.ConnTest`. You can consult these modules to learn all functionality available.

Then our case template defines a `setup` block. The `setup` block will be called before test. Most of the setup block is on setting up the SQL Sandbox, which we will talk about later. In the last line of the `setup` block, we will find this:

```
{:ok, conn: Phoenix.ConnTest.build_conn()}
```

The last line of `setup` can return test metadata that will be available in each test. The metadata we are passing forward here is a newly built `Plug.Conn`. In our test, we extract the connection out of this metadata at the very beginning of our test:

```
test "GET /", %{conn: conn} do
```

And that's where the connection comes from! At first, the testing structure does come with a bit of indirection, but this indirection pays off as our test suite grows, since it allows us to cut down the amount of boilerplate.

View tests

The other test files in our application are responsible for testing our views.

The error view test case, `test/hello_web/controllers/error_html_test.exs`, illustrates a few interesting things of its own.

```
defmodule HelloWeb.ErrorHTMLTest do
  use HelloWeb.ConnCase, async: true

  # Bring render_to_string/4 for testing custom views
  import Phoenix.Template

  test "renders 404.html" do
    assert render_to_string(HelloWeb.ErrorHTML, "404",
"html", []) == "Not Found"
  end

  test "renders 500.html" do
    assert render_to_string(HelloWeb.ErrorHTML, "500",
"html", []) == "Internal Server Error"
  end
end
```

`HelloWeb.ErrorHTMLTest` sets `async: true` which means that this test case will be run in parallel with other test cases. While individual tests within the case still run serially, this can greatly increase overall test speeds.

It also imports `Phoenix.Template` in order to use the `render_to_string/4` function. With that, all the assertions can be simple string equality tests.

Running tests per directory/file

Now that we have an idea what our tests are doing, let's look at different ways to run them.

As we saw near the beginning of this guide, we can run our entire suite of tests with `mix test`.

```
$ mix test
....

Finished in 0.2 seconds
5 tests, 0 failures
```

```
Randomized with seed 540755
```

If we would like to run all the tests in a given directory, `test/hello_web/controllers` for instance, we can pass the path to that directory to `mix test`.

```
$ mix test test/hello_web/controllers/
.

Finished in 0.2 seconds
5 tests, 0 failures

Randomized with seed 652376
```

In order to run all the tests in a specific file, we can pass the path to that file into `mix test`.

```
$ mix test test/hello_web/controllers/error_html_test.exs
...

Finished in 0.2 seconds
2 tests, 0 failures

Randomized with seed 220535
```

And we can run a single test in a file by appending a colon and a line number to the filename.

Let's say we only wanted to run the test for the way `HelloWeb.ErrorHTML` renders `500.html`. The test begins on line 11 of the file, so this is how we would do it.

```
$ mix test
test/hello_web/controllers/error_html_test.exs:11
Including tags: [line: "11"]
Excluding tags: [:test]
```

```
.

Finished in 0.1 seconds
2 tests, 0 failures, 1 excluded

Randomized with seed 288117
```

We chose to run this specifying the first line of the test, but actually, any line of that test will do. These line numbers would all work - `:11`, `:12`, or `:13`.

Running tests using tags

ExUnit allows us to tag our tests individually or for the whole module. We can then choose to run only the tests with a specific tag, or we can exclude tests with that tag and run everything else.

Let's experiment with how this works.

First, we'll add a `@moduletag` to `test/hello_web/controllers/error_html_test.exs`.

```
defmodule HelloWeb.ErrorHTMLTest do
  use HelloWeb.ConnCase, async: true

  @moduletag :error_view_case
  ...
end
```

If we use only an atom for our module tag, ExUnit assumes that it has a value of `true`. We could also specify a different value if we wanted.

```
defmodule HelloWeb.ErrorHTMLTest do
  use HelloWeb.ConnCase, async: true

  @moduletag error_view_case: "some_interesting_value"
```

```
  ...
end
```

For now, let's leave it as a simple atom `@moduletag :error_view_case`.

We can run only the tests from the error view case by passing `--only error_view_case` into `mix test`.

```
$ mix test --only error_view_case
Including tags: [:error_view_case]
Excluding tags: [:test]

...

Finished in 0.1 seconds
5 tests, 0 failures, 3 excluded

Randomized with seed 125659
```

> Note: ExUnit tells us exactly which tags it is including and excluding for each test run. If we look back to the previous section on running tests, we'll see that line numbers specified for individual tests are actually treated as tags.

```
$ mix test
test/hello_web/controllers/error_html_test.exs:11
Including tags: [line: "11"]
Excluding tags: [:test]

.

Finished in 0.2 seconds
2 tests, 0 failures, 1 excluded

Randomized with seed 364723
```

Specifying a value of `true` for `error_view_case` yields the same results.

```
$ mix test --only error_view_case:true
Including tags: [error_view_case: "true"]
Excluding tags: [:test]

...

Finished in 0.1 seconds
5 tests, 0 failures, 3 excluded

Randomized with seed 833356
```

Specifying `false` as the value for `error_view_case`, however, will not run any tests because no tags in our system match `error_view_case: false`.

```
$ mix test --only error_view_case:false
Including tags: [error_view_case: "false"]
Excluding tags: [:test]



Finished in 0.1 seconds
5 tests, 0 failures, 5 excluded

Randomized with seed 622422
The --only option was given to "mix test" but no test
executed
```

We can use the `--exclude` flag in a similar way. This will run all of the tests except those in the error view case.

```
$ mix test --exclude error_view_case
Excluding tags: [:error_view_case]

.

Finished in 0.2 seconds
5 tests, 0 failures, 2 excluded
```

```
Randomized with seed 682868
```

Specifying values for a tag works the same way for `--exclude` as it does for `--only`.

We can tag individual tests as well as full test cases. Let's tag a few tests in the error view case to see how this works.

```
defmodule HelloWeb.ErrorHTMLTest do
  use HelloWeb.ConnCase, async: true

  @moduletag :error_view_case

  # Bring render/4 and render_to_string/4 for testing
custom views
  import Phoenix.Template

  @tag individual_test: "yup"
  test "renders 404.html" do
    assert render_to_string(HelloWeb.ErrorView, "404",
"html", []) ==
           "Not Found"
  end

  @tag individual_test: "nope"
  test "renders 500.html" do
    assert render_to_string(HelloWeb.ErrorView, "500",
"html", []) ==
           "Internal Server Error"
  end
end
```

If we would like to run only tests tagged as `individual_test`, regardless of their value, this will work.

```
$ mix test --only individual_test
Including tags: [:individual_test]
Excluding tags: [:test]
```

```
..

Finished in 0.1 seconds
5 tests, 0 failures, 3 excluded

Randomized with seed 813729
```

We can also specify a value and run only tests with that.

```
$ mix test --only individual_test:yup
Including tags: [individual_test: "yup"]
Excluding tags: [:test]

.

Finished in 0.1 seconds
5 tests, 0 failures, 4 excluded

Randomized with seed 770938
```

Similarly, we can run all tests except for those tagged with a given value.

```
$ mix test --exclude individual_test:nope
Excluding tags: [individual_test: "nope"]

...

Finished in 0.2 seconds
5 tests, 0 failures, 1 excluded

Randomized with seed 539324
```

We can be more specific and exclude all the tests from the error view case except the one tagged with `individual_test` that has the value "yup".

```
$ mix test --exclude error_view_case --include
individual_test:yup
Including tags: [individual_test: "yup"]
Excluding tags: [:error_view_case]
```

```
..

Finished in 0.2 seconds
5 tests, 0 failures, 1 excluded

Randomized with seed 61472
```

Finally, we can configure ExUnit to exclude tags by default. The default ExUnit configuration is done in the `test/test_helper.exs` file:

```
ExUnit.start(exclude: [error_view_case: true])

Ecto.Adapters.SQL.Sandbox.mode(Hello.Repo, :manual)
```

Now when we run `mix test`, it only runs the specs from our `page_controller_test.exs` and `error_json_test.exs`.

```
$ mix test
Excluding tags: [error_view_case: true]

.

Finished in 0.2 seconds
5 tests, 0 failures, 2 excluded

Randomized with seed 186055
```

We can override this behavior with the `--include` flag, telling `mix test` to include tests tagged with `error_view_case`.

```
$ mix test --include error_view_case
Including tags: [:error_view_case]
Excluding tags: [error_view_case: true]

....

Finished in 0.2 seconds
```

```
5 tests, 0 failures

Randomized with seed 748424
```

This technique can be very useful to control very long running tests, which you may only want to run in CI or in specific scenarios.

Randomization

Running tests in random order is a good way to ensure that our tests are truly isolated. If we notice that we get sporadic failures for a given test, it may be because a previous test changes the state of the system in ways that aren't cleaned up afterward, thereby affecting the tests which follow. Those failures might only present themselves if the tests are run in a specific order.

ExUnit will randomize the order tests run in by default, using an integer to seed the randomization. If we notice that a specific random seed triggers our intermittent failure, we can re-run the tests with that same seed to reliably recreate that test sequence in order to help us figure out what the problem is.

```
$ mix test --seed 401472
....

Finished in 0.2 seconds
5 tests, 0 failures

Randomized with seed 401472
```

Concurrency and partitioning

As we have seen, ExUnit allows developers to run tests concurrently. This allows developers to use all of the power in their machine to run their test suites as fast as possible. Couple this with Phoenix performance, most test suites compile and run in a fraction of the time compared to other frameworks.

While developers usually have powerful machines available to them during development, this may not always be the case in your Continuous Integration servers. For this reason, ExUnit also supports out of the box test partitioning in test environments. If you open up your `config/test.exs`, you will find the database name set to:

```
database: "hello_test#
{System.get_env("MIX_TEST_PARTITION")}",
```

By default, the `MIX_TEST_PARTITION` environment variable has no value, and therefore it has no effect. But in your CI server, you can, for example, split your test suite across machines by using four distinct commands:

```
$ MIX_TEST_PARTITION=1 mix test --partitions 4
$ MIX_TEST_PARTITION=2 mix test --partitions 4
$ MIX_TEST_PARTITION=3 mix test --partitions 4
$ MIX_TEST_PARTITION=4 mix test --partitions 4
```

That's all you need to do and ExUnit and Phoenix will take care of all rest, including setting up the database for each distinct partition with a distinct name.

Going further

While ExUnit is a simple test framework, it provides a really flexible and robust test runner through the `mix test` command. We recommend you to run `mix help test` or read the docs online

We've seen what Phoenix gives us with a newly generated app. Furthermore, whenever you generate a new resource, Phoenix will generate all appropriate tests for that resource too. For example, you can create a complete scaffold with schema, context, controllers, and views by running the following command at the root of your application:

```
$ mix phx.gen.html Blog Post posts title body:text
* creating lib/hello_web/controllers/post_controller.ex
* creating
lib/hello_web/controllers/post_html/edit.html.heex
* creating
lib/hello_web/controllers/post_html/index.html.heex
* creating
lib/hello_web/controllers/post_html/new.html.heex
* creating
lib/hello_web/controllers/post_html/show.html.heex
* creating
lib/hello_web/controllers/post_html/post_form.html.heex
* creating lib/hello_web/controllers/post_html.ex
* creating
test/hello_web/controllers/post_controller_test.exs
* creating lib/hello/blog/post.ex
* creating
priv/repo/migrations/20211001233016_create_posts.exs
* creating lib/hello/blog.ex
* injecting lib/hello/blog.ex
* creating test/hello/blog_test.exs
* injecting test/hello/blog_test.exs
* creating test/support/fixtures/blog_fixtures.ex
* injecting test/support/fixtures/blog_fixtures.ex

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

    resources "/posts", PostController


Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

Now let's follow the directions and add the new resources route to our `lib/hello_web/router.ex` file and run the migrations.

When we run `mix test` again, we see that we now have twenty-one tests!

```
$ mix test
.................

Finished in 0.1 seconds
21 tests, 0 failures

Randomized with seed 537537
```

At this point, we are at a great place to transition to the rest of the testing guides, in which we'll examine these tests in much more detail, and add some of our own.

Testing Contexts

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Introduction to Testing guide.
>
> **Requirement**: This guide expects that you have gone through the Contexts guide.

At the end of the Introduction to Testing guide, we generated an HTML resource for posts using the following command:

```
$ mix phx.gen.html Blog Post posts title body:text
```

This gave us a number of modules for free, including a Blog context and a Post schema, alongside their respective test files. As we have learned in the Context guide, the Blog context is simply a module with functions to a particular area of our business domain, while Post schema maps to a particular table in our database.

In this guide, we are going to explore the tests generated for our contexts and schemas. Before we do anything else, let's run `mix test` to make sure our test suite runs cleanly.

```
$ mix test
.................

Finished in 0.6 seconds
21 tests, 0 failures

Randomized with seed 638414
```

Great. We've got twenty-one tests and they are all passing!

Testing posts

If you open up `test/hello/blog_test.exs`, you will see a file with the following:

```
defmodule Hello.BlogTest do
  use Hello.DataCase

  alias Hello.Blog

  describe "posts" do
    alias Hello.Blog.Post

    import Hello.BlogFixtures

    @invalid_attrs %{body: nil, title: nil}

    test "list_posts/0 returns all posts" do
      post = post_fixture()
      assert Blog.list_posts() == [post]
    end

    ...
```

As the top of the file we import `Hello.DataCase`, which as we will see soon, it is similar to `HelloWeb.ConnCase`. While `HelloWeb.ConnCase` sets up helpers for working with connections, which is useful when testing controllers and views, `Hello.DataCase` provides functionality for working with contexts and schemas.

Next, we define an alias, so we can refer to `Hello.Blog` simply as `Blog`.

Then we start a `describe "posts"` block. A `describe` block is a feature in ExUnit that allows us to group similar tests. The reason why we have grouped all post related tests together is because contexts in Phoenix are

capable of grouping multiple schemas together. For example, if we ran this command:

```
$ mix phx.gen.html Blog Comment comments
post_id:references:posts body:text
```

We will get a bunch of new functions in the `Hello.Blog` context, plus a whole new `describe "comments"` block in our test file.

The tests defined for our context are very straight-forward. They call the functions in our context and assert on their results. As you can see, some of those tests even create entries in the database:

```
test "create_post/1 with valid data creates a post" do
  valid_attrs = %{body: "some body", title: "some title"}

  assert {:ok, %Post{} = post} =
Blog.create_post(valid_attrs)
  assert post.body == "some body"
  assert post.title == "some title"
end
```

At this point, you may wonder: how can Phoenix make sure the data created in one of the tests do not affect other tests? We are glad you asked. To answer this question, let's talk about the `DataCase`.

The DataCase

If you open up `test/support/data_case.ex`, you will find the following:

```
defmodule Hello.DataCase do
  use ExUnit.CaseTemplate

  using do
    quote do
      alias Hello.Repo
```

```
      import Ecto
      import Ecto.Changeset
      import Ecto.Query
      import Hello.DataCase
    end
  end

  setup tags do
    Hello.DataCase.setup_sandbox(tags)
    :ok
  end

  def setup_sandbox(tags) do
    pid = Ecto.Adapters.SQL.Sandbox.start_owner!
(Hello.Repo, shared: not tags[:async])
    on_exit(fn ->
Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  end

  def errors_on(changeset) do
    ...
  end
end
```

`Hello.DataCase` is another `ExUnit.CaseTemplate`. In the `using` block, we can see all of the aliases and imports `DataCase` brings into our tests. The `setup` chunk for `DataCase` is very similar to the one from `ConnCase`. As we can see, most of the `setup` block revolves around setting up a SQL Sandbox.

The SQL Sandbox is precisely what allows our tests to write to the database without affecting any of the other tests. In a nutshell, at the beginning of every test, we start a transaction in the database. When the test is over, we automatically rollback the transaction, effectively erasing all of the data created in the test.

Furthermore, the SQL Sandbox allows multiple tests to run concurrently, even if they talk to the database. This feature is provided for PostgreSQL

databases and it can be used to further speed up your contexts and controllers tests by adding a `async: true` flag when using them:

```
use Hello.DataCase, async: true
```

There are some considerations you need to have in mind when running asynchronous tests with the sandbox, so please refer to the `Ecto.Adapters.SQL.Sandbox` for more information.

Finally at the end of the of the `DataCase` module we can find a function named `errors_on` with some examples of how to use it. This function is used for testing any validation we may want to add to our schemas. Let's give it a try by adding our own validations and then testing them.

Testing schemas

When we generate our HTML Post resource, Phoenix generated a Blog context and a Post schema. It generated a test file for the context, but no test file for the schema. However, this doesn't mean we don't need to test the schema, it just means we did not have to test the schema so far.

You may be wondering then: when do we test the context directly and when do we test the schema directly? The answer to this question is the same answer to the question of when do we add code to a context and when do we add it to the schema?

The general guideline is to keep all side-effect free code in the schema. In other words, if you are simply working with data structures, schemas and changesets, put it in the schema. The context will typically have the code that creates and updates schemas and then write them to a database or an API.

We'll be adding additional validations to the schema module, so that's a great opportunity to write some schema specific tests. Open up `lib/hello/blog/post.ex` and add the following validation to `def changeset`:

```
def changeset(post, attrs) do
  post
  |> cast(attrs, [:title, :body])
  |> validate_required([:title, :body])
  |> validate_length(:title, min: 2)
end
```

The new validation says the title needs to have at least 2 characters. Let's write a test for this. Create a new file at `test/hello/blog/post_test.exs` with this:

```
defmodule Hello.Blog.PostTest do
  use Hello.DataCase, async: true
  alias Hello.Blog.Post

  test "title must be at least two characters long" do
    changeset = Post.changeset(%Post{}, %{title: "I"})
    assert %{title: ["should be at least 2
character(s)"]} = errors_on(changeset)
  end
end
```

And that's it. As our business domain grows, we have well-defined places to test our contexts and schemas.

Testing Controllers

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Introduction to Testing guide.

At the end of the Introduction to Testing guide, we generated an HTML resource for posts using the following command:

```
$ mix phx.gen.html Blog Post posts title body:text
```

This gave us a number of modules for free, including a PostController and the associated tests. We are going to explore those tests to learn more about testing controllers in general. At the end of the guide, we will generate a JSON resource, and explore how our API tests look like.

HTML controller tests

If you open up `test/hello_web/controllers/post_controller_test.exs`, you will find the following:

```
defmodule HelloWeb.PostControllerTest do
  use HelloWeb.ConnCase

  import Hello.BlogFixtures

  @create_attrs %{body: "some body", title: "some title"}
  @update_attrs %{body: "some updated body", title: "some
updated title"}
  @invalid_attrs %{body: nil, title: nil}
```

```
  describe "index" do
    test "lists all posts", %{conn: conn} do
      conn = get(conn, ~p"/posts")
      assert html_response(conn, 200) =~ "Listing Posts"
    end
  end

  ...
```

Similar to the `PageControllerTest` that ships with our application, this controller tests uses `use HelloWeb.ConnCase` to setup the testing structure. Then, as usual, it defines some aliases, some module attributes to use throughout testing, and then it starts a series of `describe` blocks, each of them to test a different controller action.

The index action

The first describe block is for the `index` action. The action itself is implemented like this in `lib/hello_web/controllers/post_controller.ex`:

```
def index(conn, _params) do
  posts = Blog.list_posts()
  render(conn, :index, posts: posts)
end
```

It gets all posts and renders the "index.html" template. The template can be found in `lib/hello_web/templates/page/index.html.heex`.

The test looks like this:

```
describe "index" do
  test "lists all posts", %{conn: conn} do
    conn = get(conn, ~p"/posts")
    assert html_response(conn, 200) =~ "Listing Posts"
  end
end
```

The test for the `index` page is quite straight-forward. It uses the `get/2` helper to make a request to the `"/posts"` page, which is verified against our router in the test thanks to `~p`, then we assert we got a successful HTML response and match on its contents.

The create action

The next test we will look at is the one for the `create` action. The `create` action implementation is this:

```
def create(conn, %{"post" => post_params}) do
  case Blog.create_post(post_params) do
    {:ok, post} ->
      conn
      |> put_flash(:info, "Post created successfully.")
      |> redirect(to: ~p"/posts/#{post}")

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

Since there are two possible outcomes for the `create`, we will have at least two tests:

```
describe "create post" do
  test "redirects to show when data is valid", %{conn:
conn} do
    conn = post(conn, ~p"/posts", post: @create_attrs)

    assert %{id: id} = redirected_params(conn)
    assert redirected_to(conn) == ~p"/posts/#{id}"

    conn = get(conn, ~p"/posts/#{id}")
    assert html_response(conn, 200) =~ "Post #{id}"
  end

  test "renders errors when data is invalid", %{conn:
conn} do
```

```
    conn = post(conn, ~p"/posts", post: @invalid_attrs)
    assert html_response(conn, 200) =~ "New Post"
  end
end
```

The first test starts with a `post/2` request. That's because once the form in the `/posts/new` page is submitted, it becomes a POST request to the create action. Because we have supplied valid attributes, the post should have been successfully created and we should have redirected to the show action of the new post. This new page will have an address like `/posts/ID`, where ID is the identifier of the post in the database.

We then use `redirected_params(conn)` to get the ID of the post and then match that we indeed redirected to the show action. Finally, we do request a `get` request to the page we redirected to, allowing us to verify that the post was indeed created.

For the second test, we simply test the failure scenario. If any invalid attribute is given, it should re-render the "New Post" page.

One common question is: how many failure scenarios do you test at the controller level? For example, in the Testing Contexts guide, we introduced a validation to the `title` field of the post:

```
def changeset(post, attrs) do
  post
  |> cast(attrs, [:title, :body])
  |> validate_required([:title, :body])
  |> validate_length(:title, min: 2)
end
```

In other words, creating a post can fail for the following reasons:

- the title is missing
- the body is missing
- the title is present but is less than 2 characters

Should we test all of these possible outcomes in our controller tests?

The answer is no. All of the different rules and outcomes should be verified in your context and schema tests. The controller works as the integration layer. In the controller tests we simply want to verify, in broad strokes, that we handle both success and failure scenarios.

The test for `update` follows a similar structure as `create`, so let's skip to the `delete` test.

The delete action

The `delete` action looks like this:

```
def delete(conn, %{"id" => id}) do
  post = Blog.get_post!(id)
  {:ok, _post} = Blog.delete_post(post)

  conn
  |> put_flash(:info, "Post deleted successfully.")
  |> redirect(to: ~p"/posts")
end
```

The test is written like this:

```
  describe "delete post" do
    setup [:create_post]

    test "deletes chosen post", %{conn: conn, post: post}
do
      conn = delete(conn, ~p"/posts/#{post}")
      assert redirected_to(conn) == ~p"/posts"

      assert_error_sent 404, fn ->
        get(conn, ~p"/posts/#{post}")
      end
    end
  end
```

```
  defp create_post(_) do
    post = post_fixture()
    %{post: post}
  end
```

First of all, `setup` is used to declare that the `create_post` function should run before every test in this `describe` block. The `create_post` function simply creates a post and stores it in the test metadata. This allows us to, in the first line of the test, match on both the post and the connection:

```
test "deletes chosen post", %{conn: conn, post: post} do
```

The test uses `delete/2` to delete the post and then asserts that we redirected to the index page. Finally, we check that it is no longer possible to access the show page of the deleted post:

```
assert_error_sent 404, fn ->
  get(conn, ~p"/posts/#{post}")
end
```

`assert_error_sent` is a testing helper provided by `Phoenix.ConnTest`. In this case, it verifies that:

1. An exception was raised
2. The exception has a status code equivalent to 404 (which stands for Not Found)

This pretty much mimics how Phoenix handles exceptions. For example, when we access `/posts/12345` where `12345` is an ID that does not exist, we will invoke our `show` action:

```
def show(conn, %{"id" => id}) do
  post = Blog.get_post!(id)
  render(conn, :show, post: post)
end
```

When an unknown post ID is given to `Blog.get_post!/1`, it raises an `Ecto.NotFoundError`. If your application raises any exception during a web request, Phoenix translates those requests into proper HTTP response codes. In this case, 404.

We could, for example, have written this test as:

```
assert_raise Ecto.NotFoundError, fn ->
  get(conn, ~p"/posts/#{post}")
end
```

However, you may prefer the implementation Phoenix generates by default as it ignores the specific details of the failure, and instead verifies what the browser would actually receive.

The tests for `new`, `edit`, and `show` actions are simpler variations of the tests we have seen so far. You can check the action implementation and their respective tests yourself. Now we are ready to move to JSON controller tests.

JSON controller tests

So far we have been working with a generated HTML resource. However, let's take a look at how our tests look like when we generate a JSON resource.

First of all, run this command:

```
$ mix phx.gen.json News Article articles title body
```

We chose a very similar concept to the Blog context <-> Post schema, except we are using a different name, so we can study these concepts in isolation.

After you run the command above, do not forget to follow the final steps output by the generator. Once all is done, we should run `mix test` and now

have 35 passing tests:

```
$ mix test
.................

Finished in 0.6 seconds
35 tests, 0 failures

Randomized with seed 618478
```

You may have noticed that this time the scaffold controller has generated fewer tests. Previously it generated 16 (we went from 5 to 21) and now it generated 14 (we went from 21 to 35). That's because JSON APIs do not need to expose the `new` and `edit` actions. We can see this is the case in the resource we have added to the router at the end of the `mix phx.gen.json` command:

```
resources "/articles", ArticleController, except: [:new,
:edit]
```

`new` and `edit` are only necessary for HTML because they basically exist to assist users in creating and updating resources. Besides having less actions, we will notice the controller and view tests and implementations for JSON are drastically different from the HTML ones.

The only thing that is pretty much the same between HTML and JSON is the contexts and the schema, which, once you think about it, it makes total sense. After all, your business logic should remain the same, regardless if you are exposing it as HTML or JSON.

With the differences in hand, let's take a look at the controller tests.

The index action

Open up `test/hello_web/controllers/article_controller_test.exs`. The

initial structure is quite similar to `post_controller_test.exs`. So let's take a look at the tests for the `index` action. The `index` action itself is implemented in `lib/hello_web/controllers/article_controller.ex` like this:

```
def index(conn, _params) do
  articles = News.list_articles()
  render(conn, :index, articles: articles)
end
```

The action gets all articles and renders the index template. Since we are talking about JSON, we don't have a `index.json.heex` template. Instead, the code that converts `articles` into JSON can be found directly in the ArticleJSON module, defined at `lib/hello_web/controllers/article_json.ex` like this:

```
defmodule HelloWeb.ArticleJSON do
  alias Hello.News.Article

  def index(%{articles: articles}) do
    %{data: for(article <- articles, do: data(article))}
  end

  def show(%{article: article}) do
    %{data: data(article)}
  end

  defp data(%Article{} = article) do
    %{
      id: article.id,
      title: article.title,
      body: article.body
    }
  end
end
```

Since a controller render is a regular function call, we don't need any extra features to render JSON. We simply define functions for our `index` and

`show` actions that return the map of JSON for articles.

Let's take a look at the test for the `index` action then:

```
describe "index" do
  test "lists all articles", %{conn: conn} do
    conn = get(conn, ~p"/api/articles")
    assert json_response(conn, 200)["data"] == []
  end
end
```

It simply accesses the `index` path, asserts we got a JSON response with status 200 and that it contains a "data" key with an empty list, as we have no articles to return.

That was quite boring. Let's look at something more interesting.

The `create` action

The `create` action is defined like this:

```
def create(conn, %{"article" => article_params}) do
  with {:ok, %Article{} = article} <-
News.create_article(article_params) do
    conn
    |> put_status(:created)
    |> put_resp_header("location", ~p"/api/articles/#
{article}")
    |> render(:show, article: article)
  end
end
```

As we can see, it checks if an article could be created. If so, it sets the status code to `:created` (which translates to 201), it sets a "location" header with the location of the article, and then renders "show.json" with the article.

This is precisely what the first test for the `create` action verifies:

```
describe "create article" do
  test "renders article when data is valid", %{conn:
conn} do
    conn = post(conn, ~p"/articles", article:
@create_attrs)
    assert %{"id" => id} = json_response(conn, 201)
["data"]

    conn = get(conn, ~p"/api/articles/#{id}")

    assert %{
             "id" => ^id,
             "body" => "some body",
             "title" => "some title"
           } = json_response(conn, 200)["data"]
  end
```

The test uses `post/2` to create a new article and then we verify that the article returned a JSON response, with status 201, and that it had a "data" key in it. We pattern match the "data" on `%{"id" => id}`, which allows us to extract the ID of the new article. Then we perform a `get/2` request on the `show` route and verify that the article was successfully created.

Inside `describe "create article"`, we will find another test, which handles the failure scenario. Can you spot the failure scenario in the `create` action? Let's recap it:

```
def create(conn, %{"article" => article_params}) do
  with {:ok, %Article{} = article} <-
News.create_article(article_params) do
```

The `with` special form that ships as part of Elixir allows us to check explicitly for the happy paths. In this case, we are interested only in the scenarios where `News.create_article(article_params)` returns `{:ok, article}`, if it returns anything else, the other value will simply be returned directly and none of the contents inside the `do/end` block will be executed. In other words, if `News.create_article/1` returns `{:error,`

`changeset}`, we will simply return `{:error, changeset}` from the action.

However, this introduces an issue. Our actions do not know how to handle the `{:error, changeset}` result by default. Luckily, we can teach Phoenix Controllers to handle it with the Action Fallback controller. At the top of `ArticleController`, you will find:

```
action_fallback HelloWeb.FallbackController
```

This line says: if any action does not return a `%Plug.Conn{}`, we want to invoke `FallbackController` with the result. You will find `HelloWeb.FallbackController` at `lib/hello_web/controllers/fallback_controller.ex` and it looks like this:

```
defmodule HelloWeb.FallbackController do
  use HelloWeb, :controller

  def call(conn, {:error, %Ecto.Changeset{} = changeset})
do
    conn
    |> put_status(:unprocessable_entity)
    |> put_view(json: HelloWeb.ChangesetJSON)
    |> render(:error, changeset: changeset)
  end

  def call(conn, {:error, :not_found}) do
    conn
    |> put_status(:not_found)
    |> put_view(html: HelloWeb.ErrorHTML, json:
HelloWeb.ErrorJSON)
    |> render(:"404")
  end
end
```

You can see how the first clause of the `call/2` function handles the `{:error, changeset}` case, setting the status code to unprocessable entity

(422), and then rendering "error.json" from the changeset view with the failed changeset.

With this in mind, let's look at our second test for `create`:

```
test "renders errors when data is invalid", %{conn: conn}
do
  conn = post(conn, ~p"/api/articles", article:
@invalid_attrs)
  assert json_response(conn, 422)["errors"] != %{}
end
```

It simply posts to the `create` path with invalid parameters. This makes it return a JSON response, with status code 422, and a response with a non-empty "errors" key.

The `action_fallback` can be extremely useful to reduce boilerplate when designing APIs. You can learn more about the "Action Fallback" in the Controllers guide.

The `delete` action

Finally, the last action we will study is the `delete` action for JSON. Its implementation looks like this:

```
def delete(conn, %{"id" => id}) do
  article = News.get_article!(id)

  with {:ok, %Article{}} <- News.delete_article(article)
do
    send_resp(conn, :no_content, "")
  end
end
```

The new action simply attempts to delete the article and, if it succeeds, it returns an empty response with status code `:no_content` (204).

The test looks like this:

```
describe "delete article" do
  setup [:create_article]

  test "deletes chosen article", %{conn: conn, article:
article} do
    conn = delete(conn, ~p"/api/articles/#{article}")
    assert response(conn, 204)

    assert_error_sent 404, fn ->
      get(conn, ~p"/api/articles/#{article}")
    end
  end
end

defp create_article(_) do
  article = article_fixture()
  %{article: article}
end
```

It setups a new article, then in the test it invokes the `delete` path to delete it, asserting on a 204 response, which is neither JSON nor HTML. Then it verifies that we can no longer access said article.

That's all!

Now that we understand how the scaffolded code and their tests work for both HTML and JSON APIs, we are prepared to move forward in building and maintaining our web applications!

Testing Channels

> **Requirement**: This guide expects that you have gone through the introductory guides and got a Phoenix application up and running.
>
> **Requirement**: This guide expects that you have gone through the Introduction to Testing guide.
>
> **Requirement**: This guide expects that you have gone through the Channels guide.

In the Channels guide, we saw that a "Channel" is a layered system with different components. Given this, there would be cases when writing unit tests for our Channel functions may not be enough. We may want to verify that its different moving parts are working together as we expect. This integration testing would assure us that we correctly defined our channel route, the channel module, and its callbacks; and that the lower-level layers such as the PubSub and Transport are configured correctly and are working as intended.

Generating channels

As we progress through this guide, it would help to have a concrete example we could work off of. Phoenix comes with a Mix task for generating a basic channel and tests. These generated files serve as a good reference for writing channels and their corresponding tests. Let's go ahead and generate our Channel:

```
$ mix phx.gen.channel Room
* creating lib/hello_web/channels/room_channel.ex
* creating test/hello_web/channels/room_channel_test.exs
* creating test/support/channel_case.ex

The default socket handler - HelloWeb.UserSocket - was
not found.
```

```
Do you want to create it? [Yn]
* creating lib/hello_web/channels/user_socket.ex
* creating assets/js/user_socket.js

Add the socket handler to your
`lib/hello_web/endpoint.ex`, for example:

    socket "/socket", HelloWeb.UserSocket,
      websocket: true,
      longpoll: false

For the front-end integration, you need to import the
`user_socket.js`
in your `assets/js/app.js` file:

    import "./user_socket.js"
```

This creates a channel, its test and instructs us to add a channel route in `lib/hello_web/channels/user_socket.ex`. It is important to add the channel route or our channel won't function at all!

The ChannelCase

Open up `test/hello_web/channels/room_channel_test.exs` and you will find this:

```
defmodule HelloWeb.RoomChannelTest do
  use HelloWeb.ChannelCase
```

Similar to `ConnCase` and `DataCase`, we now have a `ChannelCase`. All three of them have been generated for us when we started our Phoenix application. Let's take a look at it. Open up `test/support/channel_case.ex`:

```
defmodule HelloWeb.ChannelCase do
  use ExUnit.CaseTemplate

  using do
```

```
    quote do
      # Import conveniences for testing with channels
      import Phoenix.ChannelTest
      import HelloWeb.ChannelCase

      # The default endpoint for testing
      @endpoint HelloWeb.Endpoint
    end
  end

  setup _tags do
    Hello.DataCase.setup_sandbox(tags)
    :ok
  end
end
```

It is very straight-forward. It sets up a case template that imports all of `Phoenix.ChannelTest` on use. In the `setup` block, it starts the SQL Sandbox, which we discussed in the Testing contexts guide.

Subscribe and joining

Now that we know that Phoenix provides with a custom Test Case just for channels and what it provides, we can move on to understanding the rest of `test/hello_web/channels/room_channel_test.exs`.

First off, is the setup block:

```
setup do
  {:ok, _, socket} =
    HelloWeb.UserSocket
    |> socket("user_id", %{some: :assign})
    |> subscribe_and_join(HelloWeb.RoomChannel,
"room:lobby")

  %{socket: socket}
end
```

The `setup` block sets up a `Phoenix.Socket` based on the `UserSocket` module, which you can find at `lib/hello_web/channels/user_socket.ex`. Then it says we want to subscribe and join the `RoomChannel`, accessible as `"room:lobby"` in the `UserSocket`. At the end of the test, we return the `%{socket: socket}` as metadata, so we can reuse it on every test.

In a nutshell, `subscribe_and_join/3` emulates the client joining a channel and subscribes the test process to the given topic. This is a necessary step since clients need to join a channel before they can send and receive events on that channel.

Testing a synchronous reply

The first test block in our generated channel test looks like:

```
test "ping replies with status ok", %{socket: socket} do
  ref = push(socket, "ping", %{"hello" => "there"})
  assert_reply ref, :ok, %{"hello" => "there"}
end
```

This tests the following code in our `HelloWeb.RoomChannel`:

```
# Channels can be used in a request/response fashion
# by sending replies to requests from the client
def handle_in("ping", payload, socket) do
  {:reply, {:ok, payload}, socket}
end
```

As is stated in the comment above, we see that a `reply` is synchronous since it mimics the request/response pattern we are familiar with in HTTP. This synchronous reply is best used when we only want to send an event back to the client when we are done processing the message on the server. For example, when we save something to the database and then send a message to the client only once that's done.

In the `test "ping replies with status ok", %{socket: socket} do` line, we see that we have the map `%{socket: socket}`. This gives us access to the `socket` in the setup block.

We emulate the client pushing a message to the channel with `push/3`. In the line `ref = push(socket, "ping", %{"hello" => "there"})`, we push the event `"ping"` with the payload `%{"hello" => "there"}` to the channel. This triggers the `handle_in/3` callback we have for the `"ping"` event in our channel. Note that we store the `ref` since we need that on the next line for asserting the reply. With `assert_reply ref, :ok, %{"hello" => "there"}`, we assert that the server sends a synchronous reply `:ok, %{"hello" => "there"}`. This is how we check that the `handle_in/3` callback for the `"ping"` was triggered.

Testing a Broadcast

It is common to receive messages from the client and broadcast to everyone subscribed to a current topic. This common pattern is simple to express in Phoenix and is one of the generated `handle_in/3` callbacks in our `HelloWeb.RoomChannel`.

```
def handle_in("shout", payload, socket) do
  broadcast(socket, "shout", payload)
  {:noreply, socket}
end
```

Its corresponding test looks like:

```
test "shout broadcasts to room:lobby", %{socket: socket}
do
  push(socket, "shout", %{"hello" => "all"})
  assert_broadcast "shout", %{"hello" => "all"}
end
```

We notice that we access the same `socket` that is from the setup block. How handy! We also do the same `push/3` as we did in the synchronous reply test.

So we `push` the `"shout"` event with the payload `%{"hello" => "all"}`.

Since the `handle_in/3` callback for the `"shout"` event just broadcasts the same event and payload, all subscribers in the `"room:lobby"` should receive the message. To check that, we do `assert_broadcast "shout", %{"hello" => "all"}`.

NOTE: `assert_broadcast/3` tests that the message was broadcast in the PubSub system. For testing if a client receives a message, use `assert_push/3`.

Testing an asynchronous push from the server

The last test in our `HelloWeb.RoomChannelTest` verifies that broadcasts from the server are pushed to the client. Unlike the previous tests discussed, we are indirectly testing that the channel's `handle_out/3` callback is triggered. By default, `handle_out/3` is implemented for us and simply pushes the message on to the client.

Since the `handle_out/3` event is only triggered when we call `broadcast/3` from our channel, we will need to emulate that in our test. We do that by calling `broadcast_from` or `broadcast_from!`. Both serve the same purpose with the only difference of `broadcast_from!` raising an error when broadcast fails.

The line `broadcast_from!(socket, "broadcast", %{"some" => "data"})` will trigger the `handle_out/3` callback which pushes the same event and payload back to the client. To test this, we do `assert_push "broadcast", %{"some" => "data"}`.

That's it. Now you are ready to develop and fully test real-time applications. To learn more about other functionality provided when testing channels, check out the documentation for `Phoenix.ChannelTest`.

Introduction to Deployment

Once we have a working application, we're ready to deploy it. If you're not quite finished with your own application, don't worry. Just follow the Up and Running Guide to create a basic application to work with.

When preparing an application for deployment, there are three main steps:

- Handling of your application secrets
- Compiling your application assets
- Starting your server in production

In this guide, we will learn how to get the production environment running locally. You can use the same techniques in this guide to run your application in production, but depending on your deployment infrastructure, extra steps will be necessary.

As an example of deploying to other infrastructures, we also discuss four different approaches in our guides: using Elixir's releases with `mix release`, using Gigalixir, using Fly, and using Heroku. We've also included links to deploying Phoenix on other platforms under Community Deployment Guides. Finally, the release guide has a sample Dockerfile you can use if you prefer to deploy with container technologies.

Let's explore those steps above one by one.

Handling of your application secrets

All Phoenix applications have data that must be kept secure, for example, the username and password for your production database, and the secret Phoenix uses to sign and encrypt important information. The general recommendation is to keep those in environment variables and load them into your application. This is done in `config/runtime.exs` (formerly `config/prod.secret.exs` or `config/releases.exs`), which is

responsible for loading secrets and configuration from environment variables.

Therefore, you need to make sure the proper relevant variables are set in production:

```
$ mix phx.gen.secret
REALLY_LONG_SECRET
$ export SECRET_KEY_BASE=REALLY_LONG_SECRET
$ export DATABASE_URL=ecto://USER:PASS@HOST/database
```

Do not copy those values directly, set `SECRET_KEY_BASE` according to the result of `mix phx.gen.secret` and `DATABASE_URL` according to your database address.

If for some reason you do not want to rely on environment variables, you can hard code the secrets in your `config/runtime.exs` but make sure not to check the file into your version control system.

With your secret information properly secured, it is time to configure assets!

Before taking this step, we need to do one bit of preparation. Since we will be readying everything for production, we need to do some setup in that environment by getting our dependencies and compiling.

```
$ mix deps.get --only prod
$ MIX_ENV=prod mix compile
```

Compiling your application assets

This step is required only if you have compilable assets like JavaScript and stylesheets. By default, Phoenix uses `esbuild` but everything is encapsulated in a single `mix assets.deploy` task defined in your `mix.exs`:

```
$ MIX_ENV=prod mix assets.deploy
Check your digested files at "priv/static".
```

And that is it! The Mix task by default builds the assets and then generates digests with a cache manifest file so Phoenix can quickly serve assets in production.

> Note: if you run the task above in your local machine, it will generate many digested assets in `priv/static`. You can prune them by running `mix phx.digest.clean --all`.

Keep in mind that, if you by any chance forget to run the steps above, Phoenix will show an error message:

```
$ PORT=4001 MIX_ENV=prod mix phx.server
10:50:18.732 [info] Running MyAppWeb.Endpoint with Cowboy
on http://example.com
10:50:18.735 [error] Could not find static manifest at
"my_app/_build/prod/lib/foo/priv/static/cache_manifest.js
on". Run "mix phx.digest" after building your static
files or remove the configuration from "config/prod.exs".
```

The error message is quite clear: it says Phoenix could not find a static manifest. Just run the commands above to fix it or, if you are not serving or don't care about assets at all, you can just remove the `cache_static_manifest` configuration from your config.

Starting your server in production

To run Phoenix in production, we need to set the `PORT` and `MIX_ENV` environment variables when invoking `mix phx.server`:

```
$ PORT=4001 MIX_ENV=prod mix phx.server
10:59:19.136 [info] Running MyAppWeb.Endpoint with Cowboy
on http://example.com
```

To run in detached mode so that the Phoenix server does not stop and continues to run even if you close the terminal:

```
$ PORT=4001 MIX_ENV=prod elixir --erl "-detached" -S mix
phx.server
```

In case you get an error message, please read it carefully, and open up a bug report if it is still not clear how to address it.

You can also run your application inside an interactive shell:

```
$ PORT=4001 MIX_ENV=prod iex -S mix phx.server
10:59:19.136 [info] Running MyAppWeb.Endpoint with Cowboy
on http://example.com
```

Putting it all together

The previous sections give an overview about the main steps required to deploy your Phoenix application. In practice, you will end-up adding steps of your own as well. For example, if you are using a database, you will also want to run `mix ecto.migrate` before starting the server to ensure your database is up to date.

Overall, here is a script you can use as a starting point:

```
# Initial setup
$ mix deps.get --only prod
$ MIX_ENV=prod mix compile

# Compile assets
$ MIX_ENV=prod mix assets.deploy

# Custom tasks (like DB migrations)
$ MIX_ENV=prod mix ecto.migrate
```

```
# Finally run the server
$ PORT=4001 MIX_ENV=prod mix phx.server
```

And that's it. Next, you can use one of our official guides to deploy:

- with Elixir's releases
- to Gigalixir, an Elixir-centric Platform as a Service (PaaS)
- to Fly.io, a PaaS that deploys your servers close to your users with built-in distribution support
- and to Heroku, one of the most popular PaaS.

Community Deployment Guides

- Render has first class support for Phoenix applications. There are guides for hosting Phoenix with Mix releases, Distillery, and as a Distributed Elixir Cluster.

Deploying with Releases

What we'll need

The only thing we'll need for this guide is a working Phoenix application. For those of us who need a simple application to deploy, please follow the Up and Running guide.

Goals

Our main goal for this guide is to package your Phoenix application into a self-contained directory that includes the Erlang VM, Elixir, all of your code and dependencies. This package can then be dropped into a production machine.

Releases, assemble!

If you are not familiar with Elixir releases yet, we recommend you to read Elixir's excellent docs before continuing.

Once that is done, you can assemble a release by going through all of the steps in our general deployment guide with `mix release` at the end. Let's recap.

First set the environment variables:

```
$ mix phx.gen.secret
REALLY_LONG_SECRET
$ export SECRET_KEY_BASE=REALLY_LONG_SECRET
$ export DATABASE_URL=ecto://USER:PASS@HOST/database
```

Then load dependencies to compile code and assets:

```
# Initial setup
$ mix deps.get --only prod
$ MIX_ENV=prod mix compile

# Compile assets
$ MIX_ENV=prod mix assets.deploy
```

And now run `mix phx.gen.release`:

```
$ mix phx.gen.release
==> my_app
* creating rel/overlays/bin/server
* creating rel/overlays/bin/server.bat
* creating rel/overlays/bin/migrate
* creating rel/overlays/bin/migrate.bat
* creating lib/my_app/release.ex

Your application is ready to be deployed in a release!

    # To start your system
    _build/dev/rel/my_app/bin/my_app start

    # To start your system with the Phoenix server
running
    _build/dev/rel/my_app/bin/server

    # To run migrations
    _build/dev/rel/my_app/bin/migrate

Once the release is running:

    # To connect to it remotely
    _build/dev/rel/my_app/bin/my_app remote

    # To stop it gracefully (you may also send
SIGINT/SIGTERM)
    _build/dev/rel/my_app/bin/my_app stop

To list all commands:

    _build/dev/rel/my_app/bin/my_app
```

The `phx.gen.release` task generated a few files for us to assist in releases. First, it created `server` and `migrate` *overlay* scripts for conveniently running the phoenix server inside a release or invoking migrations from a release. The files in the `rel/overlays` directory are copied into every release environment. Next, it generated a `release.ex` file which is used to invoke Ecto migrations without a dependency on `mix` itself.

Note: If you are a Docker user, you can pass the `--docker` flag to `mix phx.gen.release` to generate a Dockerfile ready for deployment.

Next, we can invoke `mix release` to build the release:

```
$ MIX_ENV=prod mix release
Generated my_app app
* assembling my_app-0.1.0 on MIX_ENV=prod
* using config/runtime.exs to configure the release at
runtime

Release created at _build/prod/rel/my_app!

    # To start your system
    _build/prod/rel/my_app/bin/my_app start

...
```

You can start the release by calling `_build/prod/rel/my_app/bin/my_app start`, or boot your webserver by calling `_build/prod/rel/my_app/bin/server`, where you have to replace `my_app` by your current application name.

Now you can get all of the files under the `_build/prod/rel/my_app` directory, package it, and run it in any production machine with the same OS and architecture as the one that assembled the release. For more details, check the docs for `mix release`.

But before we finish this guide, there is one more feature from releases that most Phoenix application will use, so let's talk about that.

Ecto migrations and custom commands

A common need in production systems is to execute custom commands required to set up the production environment. One of such commands is precisely migrating the database. Since we don't have `Mix`, a *build* tool, inside releases, which are a production artifact, we need to bring said commands directly into the release.

The `phx.gen.release` command created the following `release.ex` file in your project `lib/my_app/release.ex`, with the following content:

```
defmodule MyApp.Release do
  @app :my_app

  def migrate do
    load_app()

    for repo <- repos() do
      {:ok, _, _} = Ecto.Migrator.with_repo(repo,
&Ecto.Migrator.run(&1, :up, all: true))
    end
  end

  def rollback(repo, version) do
    load_app()
    {:ok, _, _} = Ecto.Migrator.with_repo(repo,
&Ecto.Migrator.run(&1, :down, to: version))
  end

  defp repos do
    Application.fetch_env!(@app, :ecto_repos)
  end

  defp load_app do
    Application.load(@app)
  end
end
```

Where you replace the first two lines by your application names.

Now you can assemble a new release with `MIX_ENV=prod mix release` and you can invoke any code, including the functions in the module above, by calling the `eval` command:

```
$ _build/prod/rel/my_app/bin/my_app eval
"MyApp.Release.migrate"
```

And that's it! If you peek inside the `migrate` script, you'll see it wraps exactly this invocation.

You can use this approach to create any custom command to run in production. In this case, we used `load_app`, which calls `Application.load/1` to load the current application without starting it. However, you may want to write a custom command that starts the whole application. In such cases, `Application.ensure_all_started/1` must be used. Keep in mind, starting the application will start all processes for the current application, including the Phoenix endpoint. This can be circumvented by changing your supervision tree to not start certain children under certain conditions. For example, in the release commands file you could do:

```
defp start_app do
  load_app()
  Application.put_env(@app, :minimal, true)
  Application.ensure_all_started(@app)
end
```

And then in your application you check `Application.get_env(@app, :minimal)` and start only part of the children when it is set.

Containers

Elixir releases work well with container technologies, such as Docker. The idea is that you assemble the release inside the Docker container and then build an image based on the release artifacts.

If you call `mix phx.gen.release --docker` you'll see a new file with these contents:

```
# Find eligible builder and runner images on Docker Hub.
We use Ubuntu/Debian
# instead of Alpine to avoid DNS resolution issues in
production.
#
# https://hub.docker.com/r/hexpm/elixir/tags?
page=1&name=ubuntu
# https://hub.docker.com/_/ubuntu?tab=tags
#
# This file is based on these images:
#
#   - https://hub.docker.com/r/hexpm/elixir/tags - for
the build image
#   - https://hub.docker.com/_/debian?
tab=tags&page=1&name=bullseye-20230612-slim - for the
release image
#   - https://pkgs.org/ - resource for finding needed
packages
#   - Ex: hexpm/elixir:1.14.5-erlang-25.3.2.4-debian-
bullseye-20230612-slim
#
ARG ELIXIR_VERSION=1.14.5
ARG OTP_VERSION=25.3.2.4
ARG DEBIAN_VERSION=bullseye-20230612-slim

ARG BUILDER_IMAGE="hexpm/elixir:${ELIXIR_VERSION}-
erlang-${OTP_VERSION}-debian-${DEBIAN_VERSION}"
ARG RUNNER_IMAGE="debian:${DEBIAN_VERSION}"

FROM ${BUILDER_IMAGE} as builder

# install build dependencies
RUN apt-get update -y && apt-get install -y build-
essential git \
    && apt-get clean && rm -f /var/lib/apt/lists/*_*

# prepare build dir
WORKDIR /app

# install hex + rebar
RUN mix local.hex --force && \
```

```
    mix local.rebar --force

# set build ENV
ENV MIX_ENV="prod"

# install mix dependencies
COPY mix.exs mix.lock ./
RUN mix deps.get --only $MIX_ENV
RUN mkdir config

# copy compile-time config files before we compile
dependencies
# to ensure any relevant config change will trigger the
dependencies
# to be re-compiled.
COPY config/config.exs config/${MIX_ENV}.exs config/
RUN mix deps.compile

COPY priv priv

COPY lib lib

COPY assets assets

# compile assets
RUN mix assets.deploy

# Compile the release
RUN mix compile

# Changes to config/runtime.exs don't require recompiling
the code
COPY config/runtime.exs config/

COPY rel rel
RUN mix release

# start a new build stage so that the final image will
only contain
# the compiled release and other runtime necessities
FROM ${RUNNER_IMAGE}

RUN apt-get update -y && \
  apt-get install -y libstdc++6 openssl libncurses5
locales ca-certificates \
```

```
  && apt-get clean && rm -f /var/lib/apt/lists/*_*

# Set the locale
RUN sed -i '/en_US.UTF-8/s/^# //g' /etc/locale.gen &&
locale-gen

ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8

WORKDIR "/app"
RUN chown nobody /app

# set runner ENV
ENV MIX_ENV="prod"

# Only copy the final release from the build stage
COPY --from=builder --chown=nobody:root
/app/_build/${MIX_ENV}/rel/my_app ./

USER nobody

# If using an environment that doesn't automatically reap
zombie processes, it is
# advised to add an init process such as tini via `apt-
get install`
# above and adding an entrypoint. See
https://github.com/krallin/tini for details
# ENTRYPOINT ["/tini", "--"]

CMD ["/app/bin/server"]
```

Where `my_app` is the name of your app. At the end, you will have an application in `/app` ready to run as `/app/bin/server`.

A few points about configuring a containerized application:

- If you run your app in a container, the `Endpoint` needs to be configured to listen on a "public" `:ip` address (like `0.0.0.0`) so that the app can be reached from outside the container. Whether the host should publish the container's ports to its own public IP or to localhost depends on your needs.

- The more configuration you can provide at runtime (using `config/runtime.exs`), the more reusable your images will be across environments. In particular, secrets like database credentials and API keys should not be compiled into the image, but rather should be provided when creating containers based on that image. This is why the `Endpoint`'s `:secret_key_base` is configured in `config/runtime.exs` by default.
- If possible, any environment variables that are needed at runtime should be read in `config/runtime.exs`, not scattered throughout your code. Having them all visible in one place will make it easier to ensure the containers get what they need, especially if the person doing the infrastructure work does not work on the Elixir code. Libraries in particular should never directly read environment variables; all their configuration should be handed to them by the top-level application, preferably without using the application environment.

Deploying on Gigalixir

What we'll need

The only thing we'll need for this guide is a working Phoenix application. For those of us who need a simple application to deploy, please follow the Up and Running guide.

Goals

Our main goal for this guide is to get a Phoenix application running on Gigalixir.

Steps

Let's separate this process into a few steps, so we can keep track of where we are.

- Initialize Git repository
- Install the Gigalixir CLI
- Sign up for Gigalixir
- Create and set up Gigalixir application
- Provision a database
- Make our project ready for Gigalixir
- Deploy time!
- Useful Gigalixir commands

Initializing Git repository

If you haven't already, we'll need to commit our files to git. We can do so by running the following commands in our project directory:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Installing the Gigalixir CLI

Follow the instructions here to install the command-line interface for your platform.

Signing up for Gigalixir

We can sign up for an account at gigalixir.com or with the CLI. Let's use the CLI.

```
$ gigalixir signup
```

Gigalixir's free tier does not require a credit card and comes with 1 app instance and 1 PostgreSQL database for free, but please consider upgrading to a paid plan if you are running a production application.

Next, let's login

```
$ gigalixir login
```

And verify

```
$ gigalixir account
```

Creating and setting up our Gigalixir application

There are three different ways to deploy a Phoenix app on Gigalixir: with mix, with Elixir's releases, or with Distillery. In this guide, we'll be using Mix because it is the easiest to get up and running, but you won't be able to connect a remote observer or hot upgrade. For more information, see Mix vs Distillery vs Elixir Releases. If you want to deploy with another method, follow the Getting Started Guide.

Creating a Gigalixir application

Let's create a Gigalixir application

```
$ gigalixir create -n "your_app_name"
```

Note: the app name cannot be changed afterwards. A random name is used if you do not provide one.

Verify the app was created

```
$ gigalixir apps
```

Verify that a git remote was created

```
$ git remote -v
```

Specifying versions

The buildpacks we use default to Elixir, Erlang, and Node.js versions that are quite old and it's generally a good idea to run the same version in production as you do in development, so let's do that.

```
$ echo 'elixir_version=1.14.3' > elixir_buildpack.config
$ echo 'erlang_version=24.3' >> elixir_buildpack.config
```

```
$ echo 'node_version=12.16.3' >
phoenix_static_buildpack.config
```

Phoenix v1.6 uses `esbuild` to compile your assets, but all Gigalixir images come with `npm`, so we will configure `npm` directly to deploy our assets. Add a `assets/package.json` file if you don't have any with the following:

```
{
  "scripts": {
    "deploy": "cd .. && mix assets.deploy && rm -f
_build/esbuild*"
  }
}
```

Finally, don't forget to commit:

```
$ git add elixir_buildpack.config
phoenix_static_buildpack.config assets/package.json
$ git commit -m "Set Elixir, Erlang, and Node version"
```

Making our Project ready for Gigalixir

There's nothing we need to do to get our app running on Gigalixir, but for a production app, you probably want to enforce SSL. To do that, see Force SSL

You may also want to use SSL for your database connection. For that, uncomment the line `ssl: true` in your `Repo` config.

Provisioning a database

Let's provision a database for our app

```
$ gigalixir pg:create --free
```

Verify the database was created

```
$ gigalixir pg
```

Verify that a `DATABASE_URL` and `POOL_SIZE` were created

```
$ gigalixir config
```

Deploy Time!

Our project is now ready to be deployed on Gigalixir.

```
$ git push gigalixir
```

Check the status of your deploy and wait until the app is `Healthy`

```
$ gigalixir ps
```

Run migrations

```
$ gigalixir run mix ecto.migrate
```

Check your app logs

```
$ gigalixir logs
```

If everything looks good, let's take a look at your app running on Gigalixir

```
$ gigalixir open
```

Useful Gigalixir Commands

Open a remote console

```
$ gigalixir account:ssh_keys:add "$(cat
~/.ssh/id_rsa.pub)"
$ gigalixir ps:remote_console
```

To open a remote observer, see Remote Observer

To set up clustering, see Clustering Nodes

To hot upgrade, see Hot Upgrades

For custom domains, scaling, jobs and other features, see the Gigalixir Documentation

Troubleshooting

See Troubleshooting

Also, don't hesitate to email help@gigalixir.com or request an invitation and join the #gigalixir channel on Slack.

Deploying on Fly.io

Fly.io maintains their own guide for Elixir/Phoenix here: Fly.io/docs/elixir/getting-started/ we will keep this guide up but for the latest and greatest check with them!

What we'll need

The only thing we'll need for this guide is a working Phoenix application. For those of us who need a simple application to deploy, please follow the Up and Running guide.

You can just:

```
$ mix phx.new my_app
```

Goals

The main goal for this guide is to get a Phoenix application running on Fly.io.

Sections

Let's separate this process into a few steps, so we can keep track of where we are.

- Install the Fly.io CLI
- Sign up for Fly.io
- Deploy the app to Fly.io
- Extra Fly.io tips
- Helpful Fly.io resources

Installing the Fly.io CLI

Follow the instructions here to install Flyctl, the command-line interface for the Fly.io platform.

Sign up for Fly.io

We can sign up for an account using the CLI.

```
$ fly auth signup
```

Or sign in.

```
$ flyctl auth login
```

Fly has a free tier for most applications. A credit card is required when setting up an account to help prevent abuse. See the pricing page for more details.

Deploy the app to Fly.io

To tell Fly about your application, run `fly launch` in the directory with your source code. This creates and configures a Fly.io app.

```
$ fly launch
```

This scans your source, detects the Phoenix project, and runs `mix phx.gen.release --docker` for you! This creates a Dockerfile for you.

The `fly launch` command walks you through a few questions.

- You can name the app or have it generate a random name for you.

- Choose an organization (defaults to `personal`). Organizations are a way of sharing applications and resources between Fly.io users.
- Choose a region to deploy to. Defaults to the nearest Fly.io region. You can check out the complete list of regions here.
- Sets up a Postgres DB for you.
- Builds the Dockerfile.
- Deploys your application!

The `fly launch` command also created a `fly.toml` file for you. This is where you can set ENV values and other config.

Storing secrets on Fly.io

You may also have some secrets you'd like to set on your app.

Use `fly secrets` to configure those.

```
$ fly secrets set MY_SECRET_KEY=my_secret_value
```

Deploying again

When you want to deploy changes to your application, use `fly deploy`.

```
$ fly deploy
```

Note: On Apple Silicon (M1) computers, docker runs cross-platform builds using qemu which might not always work. If you get a segmentation fault error like the following:

```
 => [build  7/17] RUN mix deps.get --only
 => => # qemu: uncaught target signal 11 (Segmentation
fault) - core dumped
```

You can use fly's remote builder by adding the `--remote-only` flag:

```
$ fly deploy --remote-only
```

You can always check on the status of a deploy

```
$ fly status
```

Check your app logs

```
$ fly logs
```

If everything looks good, open your app on Fly

```
$ fly open
```

Extra Fly.io tips

Getting an IEx shell into a running node

Elixir supports getting a IEx shell into a running production node.

There are a couple prerequisites, we first need to establish an SSH Shell to our machine on Fly.io.

This step sets up a root certificate for your account and then issues a certificate.

```
$ fly ssh issue --agent
```

With SSH configured, let's open a console.

```
$ fly ssh console
Connecting to my-app-1234.internal... complete
/ #
```

If all has gone smoothly, then you have a shell into the machine! Now we just need to launch our remote IEx shell. The deployment Dockerfile was configured to pull our application into `/app`. So the command for an app named `my_app` looks like this:

```
$ app/bin/my_app remote
Erlang/OTP 23 [erts-11.2.1] [source] [64-bit] [smp:1:1]
[ds:1:1:10] [async-threads:1]

Interactive Elixir (1.11.2) - press Ctrl+C to exit (type
h() ENTER for help)
iex(my_app@fdaa:0:1da8:a7b:ac4:b204:7e29:2)1>
```

Now we have a running IEx shell into our node! You can safely disconnect using CTRL+C, CTRL+C.

Clustering your application

Elixir and the BEAM have the incredible ability to be clustered together and pass messages seamlessly between nodes. This portion of the guide walks you through clustering your Elixir application.

There are 2 parts to getting clustering quickly setup on Fly.io.

- Installing and using `libcluster`
- Scaling the application to multiple instances

Adding `libcluster`

The widely adopted library libcluster helps here.

There are multiple strategies that `libcluster` can use to find and connect with other nodes. The strategy we'll use on Fly.io is `DNSPoll`.

After installing `libcluster`, add it to the application like this:

```
defmodule MyApp.Application do
  use Application

  def start(_type, _args) do
    topologies = Application.get_env(:libcluster,
:topologies) || []

    children = [
      # ...
      # setup for clustering
      {Cluster.Supervisor, [topologies, [name:
MyApp.ClusterSupervisor]]}
    ]

    # ...
  end

  # ...
end
```

Our next step is to add the `topologies` configuration to `config/runtime.exs`.

```
  app_name =
    System.get_env("FLY_APP_NAME") ||
      raise "FLY_APP_NAME not available"

  config :libcluster,
    topologies: [
      fly6pn: [
        strategy: Cluster.Strategy.DNSPoll,
        config: [
          polling_interval: 5_000,
          query: "#{app_name}.internal",
          node_basename: app_name
        ]
```

```
      ]
    ]
```

This configures `libcluster` to use the `DNSPoll` strategy and look for other deployed apps using the `$FLY_APP_NAME` on the `.internal` private network.

Controlling the name for our node

We need to control the naming of our Elixir nodes. To help them connect up, we'll name them using this pattern: `your-fly-app-name@the.ipv6.address.on.fly`. To do this, we'll generate the release config.

```
$ mix release.init
```

Then edit the generated `rel/env.sh.eex` file and add the following lines:

```
ip=$(grep fly-local-6pn /etc/hosts | cut -f 1)
export RELEASE_DISTRIBUTION=name
export RELEASE_NODE=$FLY_APP_NAME@$ip
```

After making the change, deploy your app!

```
$ fly deploy
```

For our app to be clustered, we have to have multiple instances. Next we'll add an additional node instance.

Running multiple instances

There are two ways to run multiple instances.

1. Scale our application to have multiple instances in one region.
2. Add an instance to another region (multiple regions).

Let's first start with a baseline of our single deployment.

```
$ fly status
...
Instances
ID       VERSION REGION DESIRED STATUS  HEALTH CHECKS
RESTARTS CREATED
f9014bf7 26      sea    run     running 1 total, 1
passing 0        1h8m ago
```

Scaling in a single region

Let's scale up to 2 instances in our current region.

```
$ fly scale count 2
Count changed to 2
```

Checking the status, we can see what happened.

```
$ fly status
...
Instances
ID       VERSION REGION DESIRED STATUS  HEALTH CHECKS
RESTARTS CREATED
eb4119d3 27      sea    run     running 1 total, 1
passing 0        39s ago
f9014bf7 27      sea    run     running 1 total, 1
passing 0        1h13m ago
```

We now have two instances in the same region.

Let's make sure they are clustered together. We can check the logs:

```
$ fly logs
...
app[eb4119d3] sea [info] 21:50:21.924 [info]
[libcluster:fly6pn] connected to :"my-app-
1234@fdaa:0:1da8:a7b:ac2:f901:4bf7:2"
...
```

But that's not as rewarding as seeing it from inside a node. From an IEx shell, we can ask the node we're connected to, what other nodes it can see.

```
$ fly ssh console -C "/app/bin/my_app remote"
```

```
iex(my-app-1234@fdaa:0:1da8:a7b:ac2:f901:4bf7:2)1>
Node.list
[:"my-app-1234@fdaa:0:1da8:a7b:ac4:eb41:19d3:2"]
```

The IEx prompt is included to help show the IP address of the node we are connected to. Then getting the `Node.list` returns the other node. Our two instances are connected and clustered!

Scaling to multiple regions

Fly makes it easy to deploy instances closer to your users. Through the magic of DNS, users are directed to the nearest region where your application is located. You can read more about Fly.io regions here.

Starting back from our baseline of a single instance running in `sea` which is Seattle, Washington (US), let's add the region `ewr` which is Parsippany, NJ (US). This puts an instance on both coasts of the US.

```
$ fly regions add ewr
Region Pool:
ewr
sea
Backup Region:
```

```
iad
lax
sjc
vin
```

Looking at the status shows that we're only in 1 region because our count is set to 1.

```
$ fly status
...
Instances
ID       VERSION REGION DESIRED STATUS  HEALTH CHECKS
RESTARTS CREATED
cdf6c422 29      sea    run     running 1 total, 1
passing 0        58s ago
```

Let's add a 2nd instance and see it deploy to `ewr`.

```
$ fly scale count 2
Count changed to 2
```

Now the status shows we have two instances spread across 2 regions!

```
$ fly status
...
Instances
ID       VERSION REGION DESIRED STATUS  HEALTH CHECKS
RESTARTS CREATED
0a8e6666 30      ewr    run     running 1 total, 1
passing 0        16s ago
cdf6c422 30      sea    run     running 1 total, 1
passing 0        6m47s ago
```

Let's ensure they are clustered together.

```
$ fly ssh console -C "/app/bin/my_app remote"
```

```
iex(my-app-1234@fdaa:0:1da8:a7b:ac2:cdf6:c422:2)1>
Node.list
[:"my-app-1234@fdaa:0:1da8:a7b:ab2:a8e:6666:2"]
```

We have two instances of our application deployed to the West and East coasts of the North American continent and they are clustered together! Our users will automatically be directed to the server nearest them.

The Fly.io platform has built-in distribution support making it easy to cluster distributed Elixir nodes in multiple regions.

Helpful Fly.io resources

Open the Dashboard for your account

```
$ fly dashboard
```

Deploy your application

```
$ fly deploy
```

Show the status of your deployed application

```
$ fly status
```

Access and tail the logs

```
$ fly logs
```

Scaling your application up or down

```
$ fly scale count 2
```

Refer to the Fly.io Elixir documentation for additional information.

Working with Fly.io applications covers things like:

- Status and logs
- Custom domains
- Certificates

Troubleshooting

See Troubleshooting and Elixir Troubleshooting

Visit the Fly.io Community to find solutions and ask questions.

Deploying on Heroku

What we'll need

The only thing we'll need for this guide is a working Phoenix application. For those of us who need a simple application to deploy, please follow the Up and Running guide.

Goals

Our main goal for this guide is to get a Phoenix application running on Heroku.

Limitations

Heroku is a great platform and Elixir performs well on it. However, you may run into limitations if you plan to leverage advanced features provided by Elixir and Phoenix, such as:

- Connections are limited.
    - Heroku limits the number of simultaneous connections as well as the duration of each connection. It is common to use Elixir for real-time apps which need lots of concurrent, persistent connections, and Phoenix is capable of handling over 2 million connections on a single server.
- Distributed clustering is not possible.
    - Heroku firewalls dynos off from one another. This means things like distributed Phoenix channels and distributed tasks will need to rely on something like Redis instead of Elixir's built-in distribution.

- In-memory state such as those in Agents, GenServers, and ETS will be lost every 24 hours.
    - Heroku restarts dynos every 24 hours regardless of whether the node is healthy.
- The built-in observer can't be used with Heroku.
    - Heroku does allow for connection into your dyno, but you won't be able to use the observer to watch the state of your dyno.

If you are just getting started, or you don't expect to use the features above, Heroku should be enough for your needs. For instance, if you are migrating an existing application running on Heroku to Phoenix, keeping a similar set of features, Elixir will perform just as well or even better than your current stack.

If you want a platform-as-a-service without these limitations, try Gigalixir. If you would rather deploy to a cloud platform, such as EC2, Google Cloud, etc, consider using `mix release`.

Steps

Let's separate this process into a few steps, so we can keep track of where we are.

- Initialize Git repository
- Sign up for Heroku
- Install the Heroku Toolbelt
- Create and set up Heroku application
- Make our project ready for Heroku
- Deploy time!
- Useful Heroku commands

Initializing Git repository

Git is a popular decentralized revision control system and is also used to deploy apps to Heroku.

Before we can push to Heroku, we'll need to initialize a local Git repository and commit our files to it. We can do so by running the following commands in our project directory:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Heroku offers some great information on how it is using Git here.

Signing up for Heroku

Signing up to Heroku is very simple, just head over to https://signup.heroku.com/ and fill in the form.

The Free plan will give us one web dyno and one worker dyno, as well as a PostgreSQL and Redis instance for free.

These are meant to be used for testing and development, and come with some limitations. In order to run a production application, please consider upgrading to a paid plan.

Installing the Heroku Toolbelt

Once we have signed up, we can download the correct version of the Heroku Toolbelt for our system here.

The Heroku CLI, part of the Toolbelt, is useful to create Heroku applications, list currently running dynos for an existing application, tail logs or run one-off commands (mix tasks for instance).

Create and Set Up Heroku Application

There are two different ways to deploy a Phoenix app on Heroku. We could use Heroku buildpacks or their container stack. The difference between these two approaches is in how we tell Heroku to treat our build. In buildpack case, we need to update our apps configuration on Heroku to use Phoenix/Elixir specific buildpacks. On container approach, we have more control on how we want to set up our app, and we can define our container image using `Dockerfile` and `heroku.yml`. This section will explore the buildpack approach. In order to use Dockerfile, it is often recommended to convert our app to use releases, which we will describe later on.

Create Application

A buildpack is a convenient way of packaging framework and/or runtime support. Phoenix requires 2 buildpacks to run on Heroku, the first adds basic Elixir support and the second adds Phoenix specific commands.

With the Toolbelt installed, let's create the Heroku application. We will do so using the latest available version of the Elixir buildpack:

```
$ heroku create --buildpack hashnuke/elixir
Creating app... done, ⬢ mysterious-meadow-6277
Setting buildpack to hashnuke/elixir... done
https://mysterious-meadow-6277.herokuapp.com/ |
https://git.heroku.com/mysterious-meadow-6277.git
```

> Note: the first time we use a Heroku command, it may prompt us to log in. If this happens, just enter the email and password you specified during signup.
>
> Note: the name of the Heroku application is the random string after "Creating" in the output above (mysterious-meadow-6277). This will be unique, so expect to see a different name from "mysterious-meadow-6277".
>
> Note: the URL in the output is the URL to our application. If we open it in our browser now, we will get the default Heroku welcome page.

> Note: if we hadn't initialized our Git repository before we ran the `heroku create` command, we wouldn't have our Heroku remote repository properly set up at this point. We can set that up manually by running: `heroku git:remote -a [our-app-name]`.

The buildpack uses a predefined Elixir and Erlang version, but to avoid surprises when deploying, it is best to explicitly list the Elixir and Erlang version we want in production to be the same we are using during development or in your continuous integration servers. This is done by creating a config file named `elixir_buildpack.config` in the root directory of your project with your target version of Elixir and Erlang:

```
# Elixir version
elixir_version=1.14.0

# Erlang version
# https://github.com/HashNuke/heroku-buildpack-elixir-
otp-builds/blob/master/otp-versions
erlang_version=24.3

# Invoke assets.deploy defined in your mix.exs to deploy
assets with esbuild
# Note we nuke the esbuild executable from the image
hook_post_compile="eval mix assets.deploy && rm -f
_build/esbuild*"
```

Finally, let's tell the build pack how to start our webserver. Create a file named `Procfile` at the root of your project:

```
web: mix phx.server
```

Optional: Node, npm, and the Phoenix Static buildpack

By default, Phoenix uses `esbuild` and manages all assets for you. However, if you are using `node` and `npm`, you will need to install the Phoenix Static buildpack to handle them:

```
$ heroku buildpacks:add
https://github.com/gjaldon/heroku-buildpack-phoenix-
static.git
Buildpack added. Next release on mysterious-meadow-6277
will use:
  1. https://github.com/HashNuke/heroku-buildpack-
elixir.git
  2. https://github.com/gjaldon/heroku-buildpack-phoenix-
static.git
```

When using this buildpack, you want to delegate all asset bundling to `npm`. So you must remove the `hook_post_compile` configuration from your `elixir_buildpack.config` and move it to the deploy script of your `assets/package.json`. Something like this:

```
{
  ...
  "scripts": {
    "deploy": "cd .. && mix assets.deploy && rm -f
_build/esbuild*"
  }
  ...
}
```

The Phoenix Static buildpack uses a predefined Node.js version, but to avoid surprises when deploying, it is best to explicitly list the Node.js version we want in production to be the same we are using during development or in your continuous integration servers. This is done by creating a config file named `phoenix_static_buildpack.config` in the root directory of your project with your target version of Node.js:

```
# Node.js version
node_version=10.20.1
```

Please refer to the configuration section for full details. You can make your own custom build script, but for now we will use the default one provided.

Finally, note that since we are using multiple buildpacks, you might run into an issue where the sequence is out of order (the Elixir buildpack needs to run before the Phoenix Static buildpack). Heroku's docs explain this better, but you will need to make sure the Phoenix Static buildpack comes last.

Making our Project ready for Heroku

Every new Phoenix project ships with a config file `config/runtime.exs` (formerly `config/prod.secret.exs`) which loads configuration and secrets from environment variables. This aligns well with Heroku best practices (12-factor apps), so the only work left for us to do is to configure URLs and SSL.

First let's tell Phoenix to only use the SSL version of the website. Find the endpoint config in your `config/prod.exs`:

```
config :scaffold, ScaffoldWeb.Endpoint,
  url: [port: 443, scheme: "https"],
```

... and add `force_ssl`

```
config :scaffold, ScaffoldWeb.Endpoint,
  url: [port: 443, scheme: "https"],
  force_ssl: [rewrite_on: [:x_forwarded_proto]],
```

`force_ssl` need to be set here because it is a *compile* time config. It will not work when set from `runtime.exs`.

Then in your `config/runtime.exs` (formerly `config/prod.secret.exs`):

... add `host`

```
config :scaffold, ScaffoldWeb.Endpoint,
  url: [host: host, port: 443, scheme: "https"]
```

and uncomment the `# ssl: true,` line in your repository configuration. It will look like this:

```
config :hello, Hello.Repo,
  ssl: true,
  url: database_url,
  pool_size:
String.to_integer(System.get_env("POOL_SIZE") || "10")
```

Finally, if you plan on using websockets, then we will need to decrease the timeout for the websocket transport in `lib/hello_web/endpoint.ex`. If you do not plan on using websockets, then leaving it set to false is fine. You can find further explanation of the options available at the documentation.

```
defmodule HelloWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :hello

  socket "/socket", HelloWeb.UserSocket,
    websocket: [timeout: 45_000]

  ...
end
```

Also set the host in Heroku:

```
$ heroku config:set PHX_HOST="mysterious-meadow-
6277.herokuapp.com"
```

This ensures that any idle connections are closed by Phoenix before they reach Heroku's 55-second timeout window.

Creating Environment Variables in Heroku

The `DATABASE_URL` config var is automatically created by Heroku when we add the Heroku Postgres add-on. We can create the database via the Heroku toolbelt:

```
$ heroku addons:create heroku-postgresql:mini
```

Now we set the `POOL_SIZE` config var:

```
$ heroku config:set POOL_SIZE=18
```

This value should be just under the number of available connections, leaving a couple open for migrations and mix tasks. The mini database allows 20 connections, so we set this number to 18. If additional dynos will share the database, reduce the `POOL_SIZE` to give each dyno an equal share.

When running a mix task later (after we have pushed the project to Heroku) you will also want to limit its pool size like so:

```
$ heroku run "POOL_SIZE=2 mix hello.task"
```

So that Ecto does not attempt to open more than the available connections.

We still have to create the `SECRET_KEY_BASE` config based on a random string. First, use `mix phx.gen.secret` to get a new secret:

```
$ mix phx.gen.secret
xvafzY4y01jYuzLm3ecJqo008dVnU3CN4f+MamNd1Zue4pXvfvUjbiXT8
akaIF53
```

Your random string will be different; don't use this example value.

Now set it in Heroku:

```
$ heroku config:set
SECRET_KEY_BASE="xvafzY4y01jYuzLm3ecJqo008dVnU3CN4f+MamNd
1Zue4pXvfvUjbiXT8akaIF53"
Setting config vars and restarting mysterious-meadow-
6277... done, v3
SECRET_KEY_BASE:
xvafzY4y01jYuzLm3ecJqo008dVnU3CN4f+MamNd1Zue4pXvfvUjbiXT8
akaIF53
```

Deploy Time!

Our project is now ready to be deployed on Heroku.

Let's commit all our changes:

```
$ git add elixir_buildpack.config
$ git commit -a -m "Use production config from Heroku ENV
variables and decrease socket timeout"
```

And deploy:

```
$ git push heroku main
Counting objects: 55, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (49/49), done.
Writing objects: 100% (55/55), 48.48 KiB | 0 bytes/s,
done.
Total 55 (delta 1), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Multipack app detected
remote: -----> Fetching custom git buildpack... done
remote: -----> elixir app detected
remote: -----> Checking Erlang and Elixir versions
remote:        WARNING: elixir_buildpack.config wasn't
found in the app
remote:        Using default config from Elixir buildpack
remote:        Will use the following versions:
```

```
remote:        * Stack cedar-14
remote:        * Erlang 17.5
remote:        * Elixir 1.0.4
remote:        Will export the following config vars:
remote:        * Config vars DATABASE_URL
remote:        * MIX_ENV=prod
remote: -----> Stack changed, will rebuild
remote: -----> Fetching Erlang 17.5
remote: -----> Installing Erlang 17.5 (changed)
remote:
remote: -----> Fetching Elixir v1.0.4
remote: -----> Installing Elixir v1.0.4 (changed)
remote: -----> Installing Hex
remote: 2015-07-07 00:04:00
URL:https://s3.amazonaws.com/s3.hex.pm/installs/1.0.0/hex
.ez [262010/262010] ->
"/app/.mix/archives/hex.ez" [1]
remote: * creating /app/.mix/archives/hex.ez
remote: -----> Installing rebar
remote: * creating /app/.mix/rebar
remote: -----> Fetching app dependencies with mix
remote: Running dependency resolution
remote: Dependency resolution completed successfully
remote: [...]
remote: -----> Compiling
remote: [...]
remote: Generated phoenix_heroku app
remote: [...]
remote: Consolidated protocols written to
_build/prod/consolidated
remote: -----> Creating .profile.d with env vars
remote: -----> Fetching custom git buildpack... done
remote: -----> Phoenix app detected
remote:
remote: -----> Loading configuration and environment
remote:        Loading config...
remote:        [...]
remote:        Will export the following config vars:
remote:        * Config vars DATABASE_URL
remote:        * MIX_ENV=prod
remote:
remote: -----> Compressing... done, 82.1MB
remote: -----> Launching... done, v5
remote:        https://mysterious-meadow-
6277.herokuapp.com/ deployed to Heroku
```

```
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/mysterious-meadow-6277.git
 * [new branch]      master -> master
```

Typing `heroku open` in the terminal should launch a browser with the Phoenix welcome page opened. In the event that you are using Ecto to access a database, you will also need to run migrations after the first deploy:

```
$ heroku run "POOL_SIZE=2 mix ecto.migrate"
```

And that's it!

Deploying to Heroku using the container stack

Create Heroku application

Set the stack of your app to `container`, this allows us to use `Dockerfile` to define our app setup.

```
$ heroku create
Creating app... done, ⬢ mysterious-meadow-6277
$ heroku stack:set container
```

Add a new `heroku.yml` file to your root folder. In this file you can define addons used by your app, how to build the image and what configs are passed to the image. You can learn more about Heroku's `heroku.yml` options here. Here is a sample:

```
setup:
  addons:
    - plan: heroku-postgresql
      as: DATABASE
build:
  docker:
```

```
    web: Dockerfile
  config:
    MIX_ENV: prod
    SECRET_KEY_BASE: $SECRET_KEY_BASE
    DATABASE_URL: $DATABASE_URL
```

Set up releases and Dockerfile

Now we need to define a `Dockerfile` at the root folder of your project that contains your application. We recommend to use releases when doing so, as the release will allow us to build a container with only the parts of Erlang and Elixir we actually use. Follow the releases docs. At the end of the guide, there is a sample Dockerfile file you can use.

Once you have the image definition set up, you can push your app to heroku and you can see it starts building the image and deploy it.

Useful Heroku Commands

We can look at the logs of our application by running the following command in our project directory:

```
$ heroku logs # use --tail if you want to tail them
```

We can also start an IEx session attached to our terminal for experimenting in our app's environment:

```
$ heroku run "POOL_SIZE=2 iex -S mix"
```

In fact, we can run anything using the `heroku run` command, like the Ecto migration task from above:

```
$ heroku run "POOL_SIZE=2 mix ecto.migrate"
```

Connecting to your dyno

Heroku gives you the ability to connect to your dyno with an IEx shell which allows running Elixir code such as database queries.

- Modify the `web` process in your Procfile to run a named node:

```
web: elixir --sname server -S mix phx.server
```

- Redeploy to Heroku
- Connect to the dyno with `heroku ps:exec` (if you have several applications on the same repository you will need to specify the app name or the remote name with `--app APP_NAME` or `--remote REMOTE_NAME`)
- Launch an iex session with `iex --sname console --remsh server`

You have an iex session into your dyno!

Troubleshooting

Compilation Error

Occasionally, an application will compile locally, but not on Heroku. The compilation error on Heroku will look something like this:

```
remote: == Compilation error on file
lib/postgrex/connection.ex ==
remote: could not compile dependency :postgrex, "mix
compile" failed. You can recompile this dependency with
"mix deps.compile postgrex", update it with "mix
deps.update postgrex" or clean it with "mix deps.clean
postgrex"
remote: ** (CompileError) lib/postgrex/connection.ex:207:
Postgrex.Connection.__struct__/0 is undefined, cannot
```

```
expand struct Postgrex.Connection
remote:     (elixir) src/elixir_map.erl:58:
:elixir_map.translate_struct/4
remote:     (stdlib) lists.erl:1353: :lists.mapfoldl/3
remote:     (stdlib) lists.erl:1354: :lists.mapfoldl/3
remote:
remote:
remote:  !     Push rejected, failed to compile elixir
app
remote:
remote: Verifying deploy...
remote:
remote: !   Push rejected to mysterious-meadow-6277.
remote:
To https://git.heroku.com/mysterious-meadow-6277.git
```

This has to do with stale dependencies which are not getting recompiled properly. It's possible to force Heroku to recompile all dependencies on each deploy, which should fix this problem. The way to do it is to add a new file called `elixir_buildpack.config` at the root of the application. The file should contain this line:

```
always_rebuild=true
```

Commit this file to the repository and try to push again to Heroku.

Connection Timeout Error

If you are constantly getting connection timeouts while running `heroku run` this could mean that your internet provider has blocked port number 5000:

```
heroku run "POOL_SIZE=2 mix myapp.task"
Running POOL_SIZE=2 mix myapp.task on mysterious-meadow-
6277... !
ETIMEDOUT: connect ETIMEDOUT 50.19.103.36:5000
```

You can overcome this by adding `detached` option to run command:

```
heroku run:detached "POOL_SIZE=2 mix ecto.migrate"
Running POOL_SIZE=2 mix ecto.migrate on mysterious-
meadow-6277... done, run.8089 (Free)
```

Routing cheatsheet

Those need to be declared in the correct router module and scope.

A quick reference to the common routing features' syntax. For an exhaustive overview, refer to the routing guides.

Routing declaration

Single route

```
get "/users", UserController, :index
patch "/users/:id", UserController, :update


# generated routes
~p"/users"
~p"/users/9" # user_id is 9
```

Also accepts `put`, `patch`, `options`, `delete` and `head`.

Resources

Simple

```
resources "/users", UserController
```

Generates `:index`, `:edit`, `:new`, `:show`, `:create`, `:update` and `:delete`.

Options

```
resources "/users", UserController, only: [:show]
resources "/users", UserController, except: [:create,
:delete]
resources "/users", UserController, as: :person #
~p"/person"
```

Nested

```
resources "/users", UserController do
  resources "/posts", PostController
end


# generated routes
~p"/users/3/posts" # user_id is 3
~p"/users/3/posts/17" # user_id is 3 and post_id = 17
```

For more info check the resources docs.

Scopes

Simple

```
scope "/admin", HelloWeb.Admin do
  pipe_through :browser

  resources "/users",   UserController
end


# generated path helpers
~p"/admin/users"
```

Nested

```
scope "/api", HelloWeb.Api, as: :api do
  pipe_through :api

  scope "/v1", V1, as: :v1 do
    resources "/users", UserController
  end
end


# generated path helpers
~p"/api/v1/users"
```

For more info check the scoped routes docs.

Custom Error Pages

New Phoenix projects have two error views called `ErrorHTML` and `ErrorJSON`, which live in `lib/hello_web/controllers/`. The purpose of these views is to handle errors in a general way for each format, from one centralized location.

The Error Views

For new applications, the `ErrorHTML` and `ErrorJSON` views looks like this:

```
defmodule HelloWeb.ErrorHTML do
  use HelloWeb, :html

  # If you want to customize your error pages,
  # uncomment the embed_templates/1 call below
  # and add pages to the error directory:
  #
  #   * lib/<%= @lib_web_name
%>/controllers/error_html/404.html.heex
  #   * lib/<%= @lib_web_name
%>/controllers/error_html/500.html.heex
  #
  # embed_templates "error_html/*"

  # The default is to render a plain text page based on
  # the template name. For example, "404.html" becomes
  # "Not Found".
  def render(template, _assigns) do

Phoenix.Controller.status_message_from_template(template)
  end
end

defmodule HelloWeb.ErrorJSON do
  # If you want to customize a particular status code,
  # you may add your own clauses, such as:
```

```
  #
  # def render("500.json", _assigns) do
  #   %{errors: %{detail: "Internal Server Error"}}
  # end

  # By default, Phoenix returns the status message from
  # the template name. For example, "404.json" becomes
  # "Not Found".
  def render(template, _assigns) do
    %{errors: %{detail:
Phoenix.Controller.status_message_from_template(template)
}}
  end
end
```

Before we dive into this, let's see what the rendered `404 Not Found` message looks like in a browser. In the development environment, Phoenix will debug errors by default, showing us a very informative debugging page. What we want here, however, is to see what page the application would serve in production. In order to do that, we need to set `debug_errors: false` in `config/dev.exs`.

```
import Config

config :hello, HelloWeb.Endpoint,
  http: [port: 4000],
  debug_errors: false,
  code_reloader: true,
  . . .
```

After modifying our config file, we need to restart our server in order for this change to take effect. After restarting the server, let's go to http://localhost:4000/such/a/wrong/path for a running local application and see what we get.

Ok, that's not very exciting. We get the bare string "Not Found", displayed without any markup or styling.

The first question is, where does that error string come from? The answer is right in `ErrorHTML`.

```
def render(template, _assigns) do

Phoenix.Controller.status_message_from_template(template)
end
```

Great, so we have this `render/2` function that takes a template and an `assigns` map, which we ignore. When you call `render(conn, :some_template)` from the controller, Phoenix first looks for a `some_template/1` function on the view module. If no function exists, it falls back to calling `render/2` with the template and format name, such as `"some_template.html"`.

In other words, to provide custom error pages, we could simply define a proper `render/2` function clause in `HelloWeb.ErrorHTML`.

```
  def render("404.html", _assigns) do
    "Page Not Found"
  end
```

But we can do even better.

Phoenix generates an `ErrorHTML` for us, but it doesn't give us a `lib/hello_web/controllers/error_html` directory. Let's create one now. Inside our new directory, let's add a template named `404.html.heex` and give it some markup – a mixture of our application layout and a new `<div>` with our message to the user.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width,
initial-scale=1"/>
```

```
    <title>Welcome to Phoenix!</title>
    <link rel="stylesheet" href="/assets/app.css"/>
    <script defer type="text/javascript"
src="/assets/app.js"></script>
  </head>
  <body>
    <header>
      <section class="container">
        <nav>
          <ul>
            <li><a
href="https://hexdocs.pm/phoenix/overview.html">Get
Started</a></li>
          </ul>
        </nav>
        <a href="https://phoenixframework.org/"
class="phx-logo">
          <img src="/images/logo.svg" alt="Phoenix
Framework Logo"/>
        </a>
      </section>
    </header>
    <main class="container">
      <section class="phx-hero">
        <p>Sorry, the page you are looking for does not
exist.</p>
      </section>
    </main>
  </body>
</html>
```

After you define the template file, remember to remove the equivalent `render/2` clause for that template, as otherwise the function overrides the template. Let's do so for the 404.html clause we have previously introduced in `lib/hello_web/controllers/error_html.ex`. We also need to tell Phoenix to embed our templates into the module:

```
+ embed_templates "error_html/*"

- def render("404.html", _assigns) do
```

```
-   "Page Not Found"
- end
```

Now, when we go back to http://localhost:4000/such/a/wrong/path, we should see a much nicer error page. It is worth noting that we did not render our `404.html.heex` template through our application layout, even though we want our error page to have the look and feel of the rest of our site. This is to avoid circular errors. For example, what happens if our application failed due to an error in the layout? Attempting to render the layout again will just trigger another error. So ideally we want to minimize the amount of dependencies and logic in our error templates, sharing only what is necessary.

Custom exceptions

Elixir provides a macro called `defexception/1` for defining custom exceptions. Exceptions are represented as structs, and structs need to be defined inside of modules.

In order to create a custom exception, we need to define a new module. Conventionally, this will have "Error" in the name. Inside that module, we need to define a new exception with `defexception/1`, the file `lib/hello_web.ex` seems like a good place for it.

```
defmodule HelloWeb.SomethingNotFoundError do
  defexception [:message]
end
```

You can raise your new exception like this:

```
raise HelloWeb.SomethingNotFoundError, "oops"
```

By default, Plug and Phoenix will treat all exceptions as 500 errors. However, Plug provides a protocol called `Plug.Exception` where we are

able to customize the status and add actions that exception structs can return on the debug error page.

If we wanted to supply a status of 404 for an `HelloWeb.SomethingNotFoundError` error, we could do it by defining an implementation for the `Plug.Exception` protocol like this, in `lib/hello_web.ex`:

```
defimpl Plug.Exception, for:
HelloWeb.SomethingNotFoundError do
  def status(_exception), do: 404
  def actions(_exception), do: []
end
```

Alternatively, you could define a `plug_status` field directly in the exception struct:

```
defmodule HelloWeb.SomethingNotFoundError do
  defexception [:message, plug_status: 404]
end
```

However, implementing the `Plug.Exception` protocol by hand can be convenient in certain occasions, such as when providing actionable errors.

Actionable errors

Exception actions are functions that can be triggered from the error page, and they're basically a list of maps defining a `label` and a `handler` to be executed. As an example, Phoenix will display an error if you have pending migrations and will provide a button on the error page to perform the pending migrations.

When `debug_errors` is `true`, they are rendered in the error page as a collection of buttons and follow the format of:

```
[
  %{
    label: String.t(),
    handler: {module(), function :: atom(), args :: []}
  }
]
```

If we wanted to return some actions for an `HelloWeb.SomethingNotFoundError` we would implement `Plug.Exception` like this:

```
defimpl Plug.Exception, for:
HelloWeb.SomethingNotFoundError do
  def status(_exception), do: 404

  def actions(_exception) do
    [
      %{
        label: "Run seeds",
        handler: {Code, :eval_file,
["priv/repo/seeds.exs"]}
      }
    ]
  end
end
```

File Uploads

One common task for web applications is uploading files. These files might be images, videos, PDFs, or files of any other type. In order to upload files through an HTML interface, we need a `file` input tag in a multipart form.

> **Looking for the LiveView Uploads guide?**
>
> This guide explains multipart HTTP file uploads via `Plug.Upload`. For more information about LiveView file uploads, including direct-to-cloud external uploads on the client, refer to the LiveView Uploads guide.

Plug provides a `Plug.Upload` struct to hold the data from the `file` input. A `Plug.Upload` struct will automatically appear in your request parameters if a user has selected a file when they submit the form.

In this guide you will do the following:

1. Configure a multipart form
2. Add a file input element to the form
3. Verify your upload params
4. Manage your uploaded files

In the `Contexts guide`, we generated an HTML resource for products. We can reuse the form we generated there in order to demonstrate how file uploads work in Phoenix. Please refer to that guide for instructions on generating the product resource you will be using here.

Configure a multipart form

The first thing you need to do is change your form into a multipart form. The `HelloWeb.CoreComponents simple_form/1` component accepts a `multipart` attribute where you can specify this.

Here is the form from `lib/hello_web/controllers/product_html/product_form.html.heex` with that change in place:

```
<.simple_form :let={f} for={@changeset} action={@action}
multipart>
. . .
```

Add a file input

Once you have a multipart form, you need a `file` input. Here's how you would do that, also in `product_form.html.heex`:

```
. . .
  <.input field={f[:photo]} type="file" label="Photo" />

  <:actions>
    <.button>Save Product</.button>
  </:actions>
</.simple_form>
```

When rendered, here is the HTML for the default `HelloWeb.CoreComponents input/1` component:

```
<div>
  <label for="product_photo" class="block text-
sm...">Photo</label>
  <input type="file" name="product[photo]"
id="product_photo" class="mt-2 block w-full...">
</div>
```

Note the `name` attribute of your `file` input. This will create the `"photo"` key in the `product_params` map which will be available in your controller action.

This is all from the form side. Now when users submit the form, a `POST` request will route to your `HelloWeb.ProductController create/2` action.

> **Should I add photo to my Ecto schema?**
>
> The photo input does not need to be part of your schema for it to come across in the `product_params`. If you want to persist any properties of the photo in a database, however, you would need to add it to your `Hello.Product` schema.

Verify your upload params

Since you generated an HTML resource, you can now start your server with `mix phx.server`, visit http://localhost:4000/products/new, and create a new product with a photo.

Before you begin, add `IO.inspect product_params` to the top of your `ProductController.create/2` action in `lib/hello_web/controllers/product_controller.ex`. This will show the `product_params` in your development log so you can get a better sense of what's happening.

```
. . .
  def create(conn, %{"product" => product_params}) do
    IO.inspect product_params
. . .
```

When you do that, this is what your `product_params` will output in the log:

```
%{"title" => "Metaprogramming Elixir", "description" =>
"Write Less Code, Get More Done (and Have Fun!)", "price"
=> "15.000000", "views" => "0",
```

```
"photo" => %Plug.Upload{content_type: "image/png",
filename: "meta-cover.png", path:
"/var/folders/_6/xbsnn7tx6g9dblyx149nrvbw0000gn/T//plug-
1434/multipart-558399-917557-1"}}
```

You have a `"photo"` key which maps to the pre-populated `Plug.Upload` struct representing your uploaded photo.

To make this easier to read, focus on the struct itself:

```
%Plug.Upload{content_type: "image/png", filename: "meta-
cover.png", path:
"/var/folders/_6/xbsnn7tx6g9dblyx149nrvbw0000gn/T//plug-
1434/multipart-558399-917557-1"}
```

`Plug.Upload` provides the file's content type, original filename, and path to the temporary file which Plug created for you. In this case, `"/var/folders/_6/xbsnn7tx6g9dblyx149nrvbw0000gn/T//plug-1434/"` is the directory created by Plug in which to put uploaded files. The directory will persist across requests. `"multipart-558399-917557-1"` is the name Plug gave to your uploaded file. If you had multiple `file` inputs and if the user selected photos for all of them, you would have multiple files scattered in temporary directories. Plug will make sure all the filenames are unique.

> **Plug.Upload files are temporary**
>
> Plug removes uploads from its directory as the request completes. If you need to do anything with this file, you need to do it before then (or give it away, but that is outside the scope of this guide).

Manage your uploaded files

Once you have the `Plug.Upload` struct available in your controller, you can perform any operation on it you want. For example, you may want to do one or more of the following:

- Check to make sure the file exists with `File.exists?/1`
- Copy the file somewhere else on the filesystem with `File.cp/2`
- Give the file away to another Elixir process with `Plug.Upload.give_away/3`
- Send it to S3 with an external library
- Send it back to the client with `Plug.Conn.send_file/5`

In a production system, you may want to copy the file to a root directory, such as `/media`. When doing so, it is important to guarantee the names are unique. For instance, if you are allowing users to upload product cover images, you could use the product id to generate a unique name:

```
if upload = product_params["photo"] do
  extension = Path.extname(upload.filename)
  File.cp(upload.path, "/media/#{product.id}-cover#
{extension}")
end
```

Then a `Plug.Static` plug could be added in your `lib/my_app_web/endpoint.ex` to serve the files at `"/media"`:

```
plug Plug.Static, at: "/uploads", from: "/media"
```

The uploaded file can now be accessed from your browsers using a path such as `"/uploads/1-cover.jpg"`. In practice, there are other concerns you want to handle when uploading files, such validating extensions, encoding names, and so on. Many times, using a library that already handles such cases is preferred.

Finally, notice that when there is no data from the `file` input, you get neither the `"photo"` key nor a `Plug.Upload` struct. Here are the `product_params` from the log.

```
%{"title" => "Metaprogramming Elixir", "description" =>
"Write Less Code, Get More Done (and Have Fun!)", "price"
=> "15.000000", "views" => "0"}
```

Configuring upload limits

The conversion from the data being sent by the form to an actual `Plug.Upload` is done by the `Plug.Parsers` plug which you can find inside `HelloWeb.Endpoint`:

```
# lib/hello_web/endpoint.ex
plug Plug.Parsers,
  parsers: [:urlencoded, :multipart, :json],
  pass: ["*/*"],
  json_decoder: Phoenix.json_library()
```

Besides the options above, `Plug.Parsers` accepts other options to control data upload:

- `:length` - sets the max body length to read, defaults to `8_000_000` bytes
- `:read_length` - set the amount of bytes to read at one time, defaults to `1_000_000` bytes
- `:read_timeout` - set the timeout for each chunk received, defaults to `15_000` ms

The first option configures the maximum data allowed. The remaining ones configure how much data we expect to read and its frequency. If the client cannot push data fast enough, the connection will be terminated. Phoenix ships with reasonable defaults but you may want to customize it under special circumstances, for example, if you are expecting really slow clients to send large chunks of data.

It is also worth pointing out those limits are important as a security mechanism. For example, if you don't set a limit for data upload, attackers

could open up thousands of connections to your application and send one byte every 2 minutes, which would take very long to complete while using up all connections to your server. The limits above expect at least a reasonable amount of progress, making attackers' lives a bit harder.

Using SSL

To prepare an application to serve requests over SSL, we need to add a little bit of configuration and two environment variables. In order for SSL to actually work, we'll need a key file and certificate file from a certificate authority. The environment variables that we'll need are paths to those two files.

The configuration consists of a new `https:` key for our endpoint whose value is a keyword list of port, path to the key file, and path to the cert (PEM) file. If we add the `otp_app:` key whose value is the name of our application, Plug will begin to look for them at the root of our application. We can then put those files in our `priv` directory and set the paths to `priv/our_keyfile.key` and `priv/our_cert.crt`.

Here's an example configuration from `config/runtime.exs`.

```
import Config

config :hello, HelloWeb.Endpoint,
  http: [port: {:system, "PORT"}],
  url: [host: "example.com"],
  cache_static_manifest:
"priv/static/cache_manifest.json",
  https: [
    port: 443,
    cipher_suite: :strong,
    otp_app: :hello,
    keyfile: System.get_env("SOME_APP_SSL_KEY_PATH"),
    certfile: System.get_env("SOME_APP_SSL_CERT_PATH"),
    # OPTIONAL Key for intermediate certificates:
    cacertfile:
System.get_env("INTERMEDIATE_CERTFILE_PATH")
  ]
```

Without the `otp_app:` key, we need to provide absolute paths to the files wherever they are on the filesystem in order for Plug to find them.

```
Path.expand("../../../some/path/to/ssl/key.pem", __DIR__)
```

The options under the `https:` key are passed to the Plug adapter, typically `Bandit`, which in turn uses `Plug.SSL` to select the TLS socket options. Please refer to the documentation for Plug.SSL.configure/1 for more information on the available options and their defaults. The Plug HTTPS Guide and the Erlang/OTP ssl documentation also provide valuable information.

SSL in Development

If you would like to use HTTPS in development, a self-signed certificate can be generated by running: `mix phx.gen.cert`. This requires Erlang/OTP 20 or later.

With your self-signed certificate, your development configuration in `config/dev.exs` can be updated to run an HTTPS endpoint:

```
config :my_app, MyAppWeb.Endpoint,
  ...
  https: [
    port: 4001,
    cipher_suite: :strong,
    keyfile: "priv/cert/selfsigned_key.pem",
    certfile: "priv/cert/selfsigned.pem"
  ]
```

This can replace your `http` configuration, or you can run HTTP and HTTPS servers on different ports.

Force SSL

In many cases, you'll want to force all incoming requests to use SSL by redirecting HTTP to HTTPS. This can be accomplished by setting the `:force_ssl` option in your endpoint configuration. It expects a list of options which are forwarded to `Plug.SSL`. By default, it sets the "strict-transport-security" header in HTTPS requests, forcing browsers to always use HTTPS. If an unsafe (HTTP) request is sent, it redirects to the HTTPS version using the `:host` specified in the `:url` configuration. For example:

```
config :my_app, MyAppWeb.Endpoint,
  force_ssl: [rewrite_on: [:x_forwarded_proto]]
```

To dynamically redirect to the `host` of the current request, set `:host` in the `:force_ssl` configuration to `nil`.

```
config :my_app, MyAppWeb.Endpoint,
  force_ssl: [rewrite_on: [:x_forwarded_proto], host:
nil]
```

In these examples, the `rewrite_on:` key specifies the HTTP header used by a reverse proxy or load balancer in front of the application to indicate whether the request was received over HTTP or HTTPS. For more information on the implications of offloading TLS to an external element, in particular relating to secure cookies, refer to the Plug HTTPS Guide. Keep in mind that the options passed to `Plug.SSL` in that document should be set using the `force_ssl:` endpoint option in a Phoenix application.

It is important to note that `force_ssl:` is a *compile* time config, so it normally is set in `prod.exs`, it will not work when set from `runtime.exs`.

HSTS

HSTS, short for 'HTTP Strict-Transport-Security', is a mechanism that allows websites to declare themselves as accessible exclusively through a secure connection (HTTPS). It was introduced to prevent man-in-the-middle

attacks that strip SSL/TLS encryption. HSTS causes web browsers to redirect from HTTP to HTTPS and to refuse to connect unless the connection uses SSL/TLS.

With `force_ssl: [hsts: true]` set, the `Strict-Transport-Security` header is added with a max-age that defines the duration for which the policy is valid. Modern web browsers will respond to this by redirecting from HTTP to HTTPS, among other consequences. RFC6797, which defines HSTS, also specifies that **the browser should keep track of a host's policy and apply it until it expires.** It further specifies that **traffic on any port other than 80 is assumed to be encrypted** as per the policy.

While HSTS is recommended in production, it can lead to unexpected behavior when accessing applications on localhost. For instance, accessing an application with HSTS enabled at `https://localhost:4000` leads to a situation where all subsequent traffic from localhost, except for port 80, is expected to be encrypted. This can disrupt traffic to other local servers or proxies running on your computer that are unrelated to your Phoenix application and may not support encrypted traffic.

If you inadvertently enable HSTS for localhost, you may need to reset your browser's cache before it will accept HTTP traffic from localhost again.

For Chrome:

1. Open the Developer Tools Panel.
2. Click and hold the reload icon next to the address bar to reveal a dropdown menu.
3. Select "Empty Cache and Hard Reload".

For Safari:

1. Clear your browser cache.
2. Remove the entry from `~/Library/Cookies/HSTS.plist` or delete the file entirely.
3. Restart Safari.

For other browsers, please consult the documentation for HSTS.

Alternatively, setting the `:expires` option on `force_ssl` to `0` should expire the entry and disable HSTS.

For more information on HSTS options, see Plug.SSL.

Writing a Channels Client

Client libraries for Phoenix Channels already exist in several languages, but if you want to write your own, this guide should get you started. It may also be useful as a guide for manual testing with a WebSocket client.

Overview

Because WebSockets are bidirectional, messages can flow in either direction at any time. For this reason, clients typically use callbacks to handle incoming messages whenever they come.

A client must join at least one topic to begin sending and receiving messages, and may join any number of topics using the same connection.

Connecting

To establish a WebSocket connection to Phoenix Channels, first make note of the `socket` declaration in the application's `Endpoint` module. For example, if you see: `socket "/mobile", MyAppWeb.MobileSocket`, the path for the initial HTTP request is:

```
[host]:[port]/mobile/websocket?vsn=2.0.0
```

Passing `&vsn=2.0.0` specifies `Phoenix.Socket.V2.JSONSerializer`, which is built into Phoenix, and which expects and returns messages in the form of lists.

You also need to include the standard header fields for upgrading an HTTP request to a WebSocket connection or use an HTTP library that handles this for you; in Elixir, mint_web_socket is an example.

Other parameters or headers may be expected or required by the specific `connect/3` function in the application's socket module (in the example above, `MyAppWeb.MobileSocket.connect/3`).

Message Format

The message format is determined by the serializer configured for the application. For these examples, `Phoenix.Socket.V2.JSONSerializer` is assumed.

The general format for messages a client sends to a Phoenix Channel is as follows:

```
[join_reference, message_reference, topic_name,
event_name, payload]
```

- The `join_reference` is also chosen by the client and should also be a unique value. It only needs to be sent for a `"phx_join"` event; for other messages it can be `null`. It is used as a message reference for `push` messages from the server, meaning those that are not replies to a specific client message. For example, imagine something like "a new user just joined the chat room".
- The `message_reference` is chosen by the client and should be a unique value. The server includes it in its reply so that the client knows which message the reply is for.
- The `topic_name` must be a known topic for the socket endpoint, and a client must join that topic before sending any messages on it.
- The `event_name` must match the first argument of a `handle_in` function on the server channel module.
- The `payload` should be a map and is passed as the second argument to that `handle_in` function.

There are three events that are understood by every Phoenix application.

First, `phx_join` is used join a channel. For example, to join the `miami:weather` channel:

```
["0", "0", "miami:weather", "phx_join", {"some":
"param"}]
```

Second, `phx_leave` is used to leave a channel. For example, to leave the `miami:weather` channel:

```
[null, "1", "miami:weather", "phx_leave", {}]
```

Third, `heartbeat` is used to maintain the WebSocket connection. For example:

```
[null, "2", "phoenix", "heartbeat", {}]
```

The `heartbeat` message is only needed when no other messages are being sent and prevents Phoenix from closing the connection; the exact `:timeout` is configured in the application's `Endpoint` module.

Other allowed messages depend on the Phoenix application.

For example, if the Channel serving the `miami:weather` can handle a `report_emergency` event:

```
def handle_in("report_emergency", payload, socket) do
  MyApp.Emergencies.report(payload) # or whatever
  {:reply, :ok, socket}
end
```

...a client could send:

[null, "3", "miami:weather", "report_emergency",
{"category": "sharknado"}]

Phoenix

This is the documentation for the Phoenix project.

To get started, see our overview guides.

Summary

Functions

Functions

json_library()

Returns the configured JSON encoding library for Phoenix.

To customize the JSON library, including the following in your `config/config.exs`:

```
config :phoenix, :json_library, AlternativeJsonLibrary
```

plug_init_mode()

Returns the `:plug_init_mode` that controls when plugs are initialized.

We recommend to set it to `:runtime` in development for compilation time improvements. It must be `:compile` in production (the default).

This option is passed as the `:init_mode` to `Plug.Builder.compile/3`.

Phoenix.Channel behaviour

Defines a Phoenix Channel.

Channels provide a means for bidirectional communication from clients that integrate with the `Phoenix.PubSub` layer for soft-realtime functionality.

For a conceptual overview, see the Channels guide.

Topics & Callbacks

Every time you join a channel, you need to choose which particular topic you want to listen to. The topic is just an identifier, but by convention it is often made of two parts: `"topic:subtopic"`. Using the `"topic:subtopic"` approach pairs nicely with the `Phoenix.Socket.channel/3` allowing you to match on all topics starting with a given prefix by using a splat (the `*` character) as the last character in the topic pattern:

```
channel "room:*", MyAppWeb.RoomChannel
```

Any topic coming into the router with the `"room:"` prefix would dispatch to `MyAppWeb.RoomChannel` in the above example. Topics can also be pattern matched in your channels' `join/3` callback to pluck out the scoped pattern:

```
# handles the special `"lobby"` subtopic
def join("room:lobby", _payload, socket) do
  {:ok, socket}
end

# handles any other subtopic as the room ID, for
example `"room:12"`, `"room:34"`
```

```
def join("room:" <> room_id, _payload, socket) do
  {:ok, socket}
end
```

Authorization

Clients must join a channel to send and receive PubSub events on that channel. Your channels must implement a `join/3` callback that authorizes the socket for the given topic. For example, you could check if the user is allowed to join that particular room.

To authorize a socket in `join/3`, return `{:ok, socket}`. To refuse authorization in `join/3`, return `{:error, reply}`.

Incoming Events

After a client has successfully joined a channel, incoming events from the client are routed through the channel's `handle_in/3` callbacks. Within these callbacks, you can perform any action. Incoming callbacks must return the `socket` to maintain ephemeral state.

Typically you'll either forward a message to all listeners with `broadcast!/3` or reply directly to a client event for request/response style messaging.

General message payloads are received as maps:

```
def handle_in("new_msg", %{"uid" => uid, "body" =>
body}, socket) do
  ...
  {:reply, :ok, socket}
end
```

Binary data payloads are passed as a `{:binary, data}` tuple:

```
def handle_in("file_chunk", {:binary, chunk}, socket)
do
  ...
  {:reply, :ok, socket}
end
```

Broadcasts

Here's an example of receiving an incoming `"new_msg"` event from one client, and broadcasting the message to all topic subscribers for this socket.

```
def handle_in("new_msg", %{"uid" => uid, "body" =>
body}, socket) do
  broadcast!(socket, "new_msg", %{uid: uid, body:
body})
  {:noreply, socket}
end
```

Replies

Replies are useful for acknowledging a client's message or responding with the results of an operation. A reply is sent only to the client connected to the current channel process. Behind the scenes, they include the client message `ref`, which allows the client to correlate the reply it receives with the message it sent.

For example, imagine creating a resource and replying with the created record:

```
def handle_in("create:post", attrs, socket) do
  changeset = Post.changeset(%Post{}, attrs)

  if changeset.valid? do
    post = Repo.insert!(changeset)
    response = MyAppWeb.PostView.render("show.json", %
{post: post})
```

```
      {:reply, {:ok, response}, socket}
    else
      response =
MyAppWeb.ChangesetView.render("errors.json", %
{changeset: changeset})
      {:reply, {:error, response}, socket}
    end
  end
```

Or you may just want to confirm that the operation succeeded:

```
  def handle_in("create:post", attrs, socket) do
    changeset = Post.changeset(%Post{}, attrs)

    if changeset.valid? do
      Repo.insert!(changeset)
      {:reply, :ok, socket}
    else
      {:reply, :error, socket}
    end
  end
```

Binary data is also supported with replies via a `{:binary, data}` tuple:

```
  {:reply, {:ok, {:binary, bin}}, socket}
```

If you don't want to send a reply to the client, you can return:

```
  {:noreply, socket}
```

One situation when you might do this is if you need to reply later; see `reply/2`.

Pushes

Calling `push/3` allows you to send a message to the client which is not a reply to a specific client message. Because it is not a reply, a pushed message does not contain a client message `ref`; there is no prior client message to relate it to.

Possible use cases include notifying a client that:

- You've auto-saved the user's document
- The user's game is ending soon
- The IoT device's settings should be updated

For example, you could `push/3` a message to the client in `handle_info/3` after receiving a `PubSub` message relevant to them.

```
alias Phoenix.Socket.Broadcast
def handle_info(%Broadcast{topic: _, event: event,
payload: payload}, socket) do
  push(socket, event, payload)
  {:noreply, socket}
end
```

Push data can be given in the form of a map or a tagged `{:binary, data}` tuple:

```
# client asks for their current rank. reply contains
it, and client
# is also pushed a leader board and a badge image
def handle_in("current_rank", _, socket) do
  push(socket, "leaders", %{leaders:
Game.get_leaders(socket.assigns.game_id)})
  push(socket, "badge", {:binary, File.read!
(socket.assigns.badge_path)})
  {:reply, %{val:
Game.get_rank(socket.assigns[:user])}, socket}
end
```

Note that in this example, `push/3` is called from `handle_in/3`; in this way you can essentially reply N times to a single message from the client.

See `reply/2` for why this may be desirable.

Intercepting Outgoing Events

When an event is broadcasted with `broadcast/3`, each channel subscriber can choose to intercept the event and have their `handle_out/3` callback triggered. This allows the event's payload to be customized on a socket by socket basis to append extra information, or conditionally filter the message from being delivered. If the event is not intercepted with `Phoenix.Channel.intercept/1`, then the message is pushed directly to the client:

```
intercept ["new_msg", "user_joined"]

# for every socket subscribing to this topic, append an
`is_editable`
# value for client metadata.
def handle_out("new_msg", msg, socket) do
  push(socket, "new_msg", Map.merge(msg,
    %{is_editable: User.can_edit_message?
(socket.assigns[:user], msg)}
  ))
  {:noreply, socket}
end

# do not send broadcasted `"user_joined"` events if
this socket's user
# is ignoring the user who joined.
def handle_out("user_joined", msg, socket) do
  unless User.ignoring?(socket.assigns[:user],
msg.user_id) do
    push(socket, "user_joined", msg)
  end
  {:noreply, socket}
end
```

Broadcasting to an external topic

In some cases, you will want to broadcast messages without the context of a `socket`. This could be for broadcasting from within your channel to an

external topic, or broadcasting from elsewhere in your application like a controller or another process. Such can be done via your endpoint:

```
# within channel
def handle_in("new_msg", %{"uid" => uid, "body" =>
body}, socket) do
  ...
  broadcast_from!(socket, "new_msg", %{uid: uid, body:
body})
  MyAppWeb.Endpoint.broadcast_from!(self(),
"room:superadmin",
    "new_msg", %{uid: uid, body: body})
  {:noreply, socket}
end

# within controller
def create(conn, params) do
  ...
  MyAppWeb.Endpoint.broadcast!("room:" <> rid,
"new_msg", %{uid: uid, body: body})
  MyAppWeb.Endpoint.broadcast!("room:superadmin",
"new_msg", %{uid: uid, body: body})
  redirect(conn, to: "/")
end
```

Terminate

On termination, the channel callback `terminate/2` will be invoked with the error reason and the socket.

If we are terminating because the client left, the reason will be `{:shutdown, :left}`. Similarly, if we are terminating because the client connection was closed, the reason will be `{:shutdown, :closed}`.

If any of the callbacks return a `:stop` tuple, it will also trigger terminate with the reason given in the tuple.

`terminate/2`, however, won't be invoked in case of errors nor in case of exits. This is the same behaviour as you find in Elixir abstractions like `GenServer` and others. Similar to `GenServer`, it would also be possible to `:trap_exit` to guarantee that `terminate/2` is invoked. This practice is not encouraged though.

Generally speaking, if you want to clean something up, it is better to monitor your channel process and do the clean up from another process. All channel callbacks, including `join/3`, are called from within the channel process. Therefore, `self()` in any of them returns the PID to be monitored.

Exit reasons when stopping a channel

When the channel callbacks return a `:stop` tuple, such as:

```
{:stop, :shutdown, socket}
{:stop, {:error, :enoent}, socket}
```

the second argument is the exit reason, which follows the same behaviour as standard `GenServer` exits.

You have three options to choose from when shutting down a channel:

- `:normal` - in such cases, the exit won't be logged and linked processes do not exit
- `:shutdown` or `{:shutdown, term}` - in such cases, the exit won't be logged and linked processes exit with the same reason unless they're trapping exits
- any other term - in such cases, the exit will be logged and linked processes exit with the same reason unless they're trapping exits

Subscribing to external topics

Sometimes you may need to programmatically subscribe a socket to external topics in addition to the internal `socket.topic`. For example, imagine you have a bidding system where a remote client dynamically sets preferences on products they want to receive bidding notifications on. Instead of requiring a unique channel process and topic per preference, a more efficient and simple approach would be to subscribe a single channel to relevant notifications via your endpoint. For example:

```
defmodule MyAppWeb.Endpoint.NotificationChannel do
  use Phoenix.Channel

  def join("notification:" <> user_id, %{"ids" => ids},
socket) do
    topics = for product_id <- ids, do: "product:#
{product_id}"

    {:ok, socket
          |> assign(:topics, [])
          |> put_new_topics(topics)}
  end

  def handle_in("watch", %{"product_id" => id}, socket)
do
    {:reply, :ok, put_new_topics(socket, ["product:#
{id}"])}
  end

  def handle_in("unwatch", %{"product_id" => id},
socket) do
    {:reply, :ok,
MyAppWeb.Endpoint.unsubscribe("product:#{id}")}
  end

  defp put_new_topics(socket, topics) do
    Enum.reduce(topics, socket, fn topic, acc ->
      topics = acc.assigns.topics
      if topic in topics do
        acc
      else
        :ok = MyAppWeb.Endpoint.subscribe(topic)
        assign(acc, :topics, [topic | topics])
      end
```

```
    end)
  end
end
```

Note: the caller must be responsible for preventing duplicate subscriptions. After calling `subscribe/1` from your endpoint, the same flow applies to handling regular Elixir messages within your channel. Most often, you'll simply relay the `%Phoenix.Socket.Broadcast{}` event and payload:

```
alias Phoenix.Socket.Broadcast
def handle_info(%Broadcast{topic: _, event: event,
payload: payload}, socket) do
  push(socket, event, payload)
  {:noreply, socket}
end
```

Hibernation

From Erlang/OTP 20, channels automatically hibernate to save memory after 15_000 milliseconds of inactivity. This can be customized by passing the `:hibernate_after` option to `use Phoenix.Channel`:

```
use Phoenix.Channel, hibernate_after: 60_000
```

You can also set it to `:infinity` to fully disable it.

Shutdown

You can configure the shutdown behavior of each channel used when your application is shutting down by setting the `:shutdown` value on use:

```
use Phoenix.Channel, shutdown: 5_000
```

It defaults to 5_000. The supported values are described under the in the `Supervisor` module docs.

Logging

By default, channel `"join"` and `"handle_in"` events are logged, using the level `:info` and `:debug`, respectively. You can change the level used for each event, or disable logs, per event type by setting the `:log_join` and `:log_handle_in` options when using `Phoenix.Channel`. For example, the following configuration logs join events as `:info`, but disables logging for incoming events:

```
use Phoenix.Channel, log_join: :info, log_handle_in:
false
```

Note that changing an event type's level doesn't affect what is logged, unless you set it to `false`, it affects the associated level.

Summary

Types

Callbacks

Handle regular GenServer call messages.

handle_cast(msg, socket)

Handle regular GenServer cast messages.

handle_in(event, payload, socket)

Handle incoming `event`s.

handle_info(msg, socket)

Handle regular Elixir process messages.

handle_out(event, payload, socket)

Intercepts outgoing `event`s.

join(topic, payload, socket)

Handle channel joins by `topic`.

terminate(reason, t)

Invoked when the channel process is about to exit.

Functions

broadcast(socket, event, message)

Broadcast an event to all subscribers of the socket topic.

broadcast!(socket, event, message)

Same as `broadcast/3`, but raises if broadcast fails.

broadcast_from(socket, event, message)

Broadcast event from pid to all subscribers of the socket topic.

broadcast_from!(socket, event, message)

Same as `broadcast_from/3`, but raises if broadcast fails.

intercept(events)

Defines which Channel events to intercept for `handle_out/3` callbacks.

push(socket, event, message)

Sends an event directly to the connected client without requiring a prior message from the client.

reply(socket_ref, status)

Replies asynchronously to a socket push.

socket_ref(socket)

Generates a `socket_ref` for an async reply.

Types

payload()

```
@type payload() :: map() | term() | {:binary, binary()}
```

reply()

```
@type reply() :: status :: atom() | {status :: atom(),
response :: payload()}
```

socket_ref()

```
@type socket_ref() ::
  {transport_pid :: Pid, serializer :: module(), topic ::
binary(),
   ref :: binary(), join_ref :: binary()}
```

Callbacks

handle_call(msg, from, socket)

(optional)

```
@callback handle_call(
  msg :: term(),
  from :: {pid(), tag :: term()},
  socket :: Phoenix.Socket.t()
) ::
  {:reply, response :: term(), Phoenix.Socket.t()}
  | {:noreply, Phoenix.Socket.t()}
  | {:stop, reason :: term(), Phoenix.Socket.t()}
```

Handle regular GenServer call messages.

See GenServer.handle_call/3.

handle_cast(msg, socket)

(optional)

```
@callback handle_cast(msg :: term(), socket ::
Phoenix.Socket.t()) ::
  {:noreply, Phoenix.Socket.t()} | {:stop, reason ::
term(), Phoenix.Socket.t()}
```

Handle regular GenServer cast messages.

See `GenServer.handle_cast/2`.

handle_in(event, payload, socket)

(optional)

```
@callback handle_in(
  event :: String.t(),
  payload :: payload(),
  socket :: Phoenix.Socket.t()
) ::
  {:noreply, Phoenix.Socket.t()}
  | {:noreply, Phoenix.Socket.t(), timeout() | :hibernate}
  | {:reply, reply(), Phoenix.Socket.t()}
  | {:stop, reason :: term(), Phoenix.Socket.t()}
  | {:stop, reason :: term(), reply(), Phoenix.Socket.t()}
```

Handle incoming `event`s.

Payloads are serialized before sending with the configured serializer.

Example

```
  def handle_in("ping", payload, socket) do
    {:reply, {:ok, payload}, socket}
  end
```

handle_info(msg, socket)

(optional)

```
@callback handle_info(msg :: term(), socket ::
Phoenix.Socket.t()) ::
  {:noreply, Phoenix.Socket.t()} | {:stop, reason ::
term(), Phoenix.Socket.t()}
```

Handle regular Elixir process messages.

See GenServer.handle_info/2.

handle_out(event, payload, socket)

(optional)

```
@callback handle_out(
  event :: String.t(),
  payload :: payload(),
  socket :: Phoenix.Socket.t()
) ::
  {:noreply, Phoenix.Socket.t()}
  | {:noreply, Phoenix.Socket.t(), timeout() | :hibernate}
  | {:stop, reason :: term(), Phoenix.Socket.t()}
```

Intercepts outgoing `event`s.

See `intercept/1`.

join(topic, payload, socket)

```
@callback join(topic :: binary(), payload :: payload(),
socket :: Phoenix.Socket.t()) ::
  {:ok, Phoenix.Socket.t()}
  | {:ok, reply :: payload(), Phoenix.Socket.t()}
  | {:error, reason :: map()}
```

Handle channel joins by `topic`.

To authorize a socket, return `{:ok, socket}` or `{:ok, reply, socket}`. To refuse authorization, return `{:error, reason}`.

Payloads are serialized before sending with the configured serializer.

Example

```
def join("room:lobby", payload, socket) do
  if authorized?(payload) do
    {:ok, socket}
  else
    {:error, %{reason: "unauthorized"}}
  end
end
```

terminate(reason, t)

(optional)

```
@callback terminate(
  reason :: :normal | :shutdown | {:shutdown, :left |
:closed | term()},
  Phoenix.Socket.t()
) :: term()
```

Invoked when the channel process is about to exit.

See GenServer.terminate/2.

Functions

broadcast(socket, event, message)

Broadcast an event to all subscribers of the socket topic.

The event's message must be a serializable map or a tagged `{:binary, data}` tuple where `data` is binary data.

Examples

```
iex> broadcast(socket, "new_message", %{id: 1, content:
"hello"})
:ok

iex> broadcast(socket, "new_message", {:binary,
"hello"})
:ok
```

broadcast!(socket, event, message)

Same as `broadcast/3`, but raises if broadcast fails.

broadcast_from(socket, event, message)

Broadcast event from pid to all subscribers of the socket topic.

The channel that owns the socket will not receive the published message. The event's message must be a serializable map or a tagged `{:binary, data}` tuple where `data` is binary data.

Examples

```
iex> broadcast_from(socket, "new_message", %{id: 1,
content: "hello"})
:ok

iex> broadcast_from(socket, "new_message", {:binary,
"hello"})
:ok
```

broadcast_from!(socket, event, message)

Same as `broadcast_from/3`, but raises if broadcast fails.

intercept(events)

(macro)

Defines which Channel events to intercept for `handle_out/3` callbacks.

By default, broadcasted events are pushed directly to the client, but intercepting events gives your channel a chance to customize the event for the client to append extra information or filter the message from being delivered.

Note: intercepting events can introduce significantly more overhead if a large number of subscribers must customize a message since the broadcast will be encoded N times instead of a single shared encoding across all subscribers.

Examples

```
intercept ["new_msg"]

def handle_out("new_msg", payload, socket) do
  push(socket, "new_msg", Map.merge(payload,
    is_editable: User.can_edit_message?
(socket.assigns[:user], payload)
  ))
  {:noreply, socket}
end
```

`handle_out/3` callbacks must return one of:

```
{:noreply, Socket.t} |
{:noreply, Socket.t, timeout | :hibernate} |
{:stop, reason :: term, Socket.t}
```

push(socket, event, message)

Sends an event directly to the connected client without requiring a prior message from the client.

The event's message must be a serializable map or a tagged `{:binary, data}` tuple where `data` is binary data.

Note that unlike some in client libraries, this server-side `push/3` does not return a reference. If you need to get a reply from the client and to correlate that reply with the message you pushed, you'll need to include a unique identifier in the message, track it in the Channel's state, have the client include it in its reply, and examine the ref when the reply comes to `handle_in/3`.

Examples

```
iex> push(socket, "new_message", %{id: 1, content:
"hello"})
:ok

iex> push(socket, "new_message", {:binary, "hello"})
:ok
```

reply(socket_ref, status)

```
@spec reply(socket_ref(), reply()) :: :ok
```

Replies asynchronously to a socket push.

The usual way of replying to a client's message is to return a tuple from `handle_in/3` like:

```
{:reply, {status, payload}, socket}
```

But sometimes you need to reply to a push asynchronously - that is, after your `handle_in/3` callback completes. For example, you might need to perform work in another process and reply when it's finished.

You can do this by generating a reference to the socket with `socket_ref/1` and calling `reply/2` with that ref when you're ready to reply.

Note: A `socket_ref` is required so the `socket` itself is not leaked outside the channel. The `socket` holds information such as assigns and transport configuration, so it's important to not copy this information outside of the channel that owns it.

Technically, `reply/2` will allow you to reply multiple times to the same client message, and each reply will include the client message `ref`. But the client may expect only one reply; in that case, `push/3` would be preferable for the additional messages.

Payloads are serialized before sending with the configured serializer.

Examples

```
def handle_in("work", payload, socket) do
  Worker.perform(payload, socket_ref(socket))
  {:noreply, socket}
end

def handle_info({:work_complete, result, ref}, socket)
do
  reply(ref, {:ok, result})
  {:noreply, socket}
end
```

socket_ref(socket)

```
@spec socket_ref(Phoenix.Socket.t()) :: socket_ref()
```

Generates a `socket_ref` for an async reply.

See `reply/2` for example usage.

Phoenix.Controller

Controllers are used to group common functionality in the same (pluggable) module.

For example, the route:

```
get "/users/:id", MyAppWeb.UserController, :show
```

will invoke the `show/2` action in the `MyAppWeb.UserController`:

```
defmodule MyAppWeb.UserController do
  use MyAppWeb, :controller

  def show(conn, %{"id" => id}) do
    user = Repo.get(User, id)
    render(conn, :show, user: user)
  end
end
```

An action is a regular function that receives the connection and the request parameters as arguments. The connection is a `Plug.Conn` struct, as specified by the Plug library.

Then we invoke `render/3`, passing the connection, the template to render (typically named after the action), and the `user: user` as assigns. We will explore all of those concepts next.

Connection

A controller by default provides many convenience functions for manipulating the connection, rendering templates, and more.

Those functions are imported from two modules:

- `Plug.Conn` - a collection of low-level functions to work with the connection

- `Phoenix.Controller` - functions provided by Phoenix to support rendering, and other Phoenix specific behaviour

If you want to have functions that manipulate the connection without fully implementing the controller, you can import both modules directly instead of `use Phoenix.Controller`.

Rendering and layouts

One of the main features provided by controllers is the ability to perform content negotiation and render templates based on information sent by the client.

There are two ways to render content in a controller. One option is to invoke format-specific functions, such as `html/2` and `json/2`.

However, most commonly controllers invoke custom modules called views. Views are modules capable of rendering a custom format. This is done by specifying the option `:formats` when defining the controller:

```
use Phoenix.Controller,
  formats: [:html, :json]
```

Now, when invoking `render/3`, a controller named `MyAppWeb.UserController` will invoke `MyAppWeb.UserHTML` and `MyAppWeb.UserJSON` respectively when rendering each format:

```
def show(conn, %{"id" => id}) do
  user = Repo.get(User, id)
  # Will invoke UserHTML.show(%{user: user}) for html
requests
  # Will invoke UserJSON.show(%{user: user}) for json
requests
```

```
    render(conn, :show, user: user)
  end
```

Some formats are also handy to have layouts, which render content shared across all pages. We can also specify layouts on `use`:

```
  use Phoenix.Controller,
    formats: [:html, :json],
    layouts: [html: MyAppWeb.Layouts]
```

You can also specify formats and layouts to render by calling `put_view/2` and `put_layout/2` directly with a connection. The line above can also be written directly in your actions as:

```
  conn
  |> put_view(html: MyAppWeb.UserHTML, json:
  MyAppWeb.UserJSON)
  |> put_layout(html: MyAppWeb.Layouts)
```

Backwards compatibility

In previous Phoenix versions, a controller you always render `MyApp.UserView`. This behaviour can be explicitly retained by passing a suffix to the formats options:

```
  use Phoenix.Controller,
    formats: [html: "View", json: "View"],
    layouts: [html: MyAppWeb.Layouts]
```

Options

When used, the controller supports the following options to customize template rendering:

- `:formats` - the formats this controller will render by default. For example, specifying `formats: [:html, :json]` for a controller named `MyAppWeb.UserController` will invoke `MyAppWeb.UserHTML` and `MyAppWeb.UserJSON` when respectively rendering each format. If `:formats` is not set, the default view is set to `MyAppWeb.UserView`

- `:layouts` - which layouts to render for each format, for example: `[html: DemoWeb.Layouts]`

Deprecated options:

- `:namespace` - sets the namespace for the layout. Use `:layouts` instead

- `:put_default_views` - controls whether the default view and layout should be set or not. Set `formats: []` and `layouts: []` instead

Plug pipeline

As with routers, controllers also have their own plug pipeline. However, different from routers, controllers have a single pipeline:

```
defmodule MyAppWeb.UserController do
  use MyAppWeb, :controller

  plug :authenticate, usernames: ["jose", "eric",
"sonny"]

  def show(conn, params) do
    # authenticated users only
  end

  defp authenticate(conn, options) do
    if get_session(conn, :username) in
options[:usernames] do
      conn
    else
```

```
      conn |> redirect(to: "/") |> halt()
    end
  end
end
```

The `:authenticate` plug will be invoked before the action. If the plug calls `Plug.Conn.halt/1` (which is by default imported into controllers), it will halt the pipeline and won't invoke the action.

Guards

`plug/2` in controllers supports guards, allowing a developer to configure a plug to only run in some particular action.

```
plug :do_something when action in [:show, :edit]
```

Due to operator precedence in Elixir, if the second argument is a keyword list, we need to wrap the keyword in `[...]` when using `when`:

```
plug :authenticate, [usernames: ["jose", "eric",
"sonny"]] when action in [:show, :edit]
plug :authenticate, [usernames: ["admin"]] when not
action in [:index]
```

The first plug will run only when action is show or edit. The second plug will always run, except for the index action.

Those guards work like regular Elixir guards and the only variables accessible in the guard are `conn`, the `action` as an atom and the `controller` as an alias.

Controllers are plugs

Like routers, controllers are plugs, but they are wired to dispatch to a particular function which is called an action.

For example, the route:

```
get "/users/:id", UserController, :show
```

will invoke `UserController` as a plug:

```
UserController.call(conn, :show)
```

which will trigger the plug pipeline and which will eventually invoke the inner action plug that dispatches to the `show/2` function in `UserController`.

As controllers are plugs, they implement both `init/1` and `call/2`, and it also provides a function named `action/2` which is responsible for dispatching the appropriate action after the plug stack (and is also overridable).

Overriding `action/2` for custom arguments

Phoenix injects an `action/2` plug in your controller which calls the function matched from the router. By default, it passes the conn and params. In some cases, overriding the `action/2` plug in your controller is a useful way to inject arguments into your actions that you would otherwise need to repeatedly fetch off the connection. For example, imagine if you stored a `conn.assigns.current_user` in the connection and wanted quick access to the user for every action in your controller:

```
def action(conn, _) do
  args = [conn, conn.params, conn.assigns.current_user]
  apply(__MODULE__, action_name(conn), args)
end

def index(conn, _params, user) do
  videos = Repo.all(user_videos(user))
  # ...
end
```

```
def delete(conn, %{"id" => id}, user) do
  video = Repo.get!(user_videos(user), id)
  # ...
end
```

Summary

Types

layout()
view()

Functions

accepts(conn, accepted)

Performs content negotiation based on the available formats.

action_fallback(plug)

Registers the plug to call as a fallback to the controller action.

action_name(conn)

Returns the action name as an atom, raises if unavailable.

allow_jsonp(conn, opts \\ [])

A plug that may convert a JSON response into a JSONP one.

clear_flash(conn)

Clears all flash messages.

controller_module(conn)

Returns the controller module as an atom, raises if unavailable.

current_path(conn)

Returns the current request path with its default query parameters

current_path(conn, params)

Returns the current path with the given query parameters.

current_url(conn)

Returns the current request url with its default query parameters

current_url(conn, params)

Returns the current request URL with query params.

delete_csrf_token()

Deletes the CSRF token from the process dictionary.

endpoint_module(conn)

Returns the endpoint module as an atom, raises if unavailable.

fetch_flash(conn, opts \\ [])

Fetches the flash storage.

get_csrf_token()

Gets or generates a CSRF token.

get_flash(conn) deprecated

Returns a map of previously set flash messages or an empty map.

get_flash(conn, key) deprecated

Returns a message from flash by `key` (or `nil` if no message is available for `key`).

get_format(conn)

Returns the request format, such as "json", "html".

html(conn, data)

Sends html response.

json(conn, data)

Sends JSON response.

layout(conn, format \\ nil)

Retrieves the current layout for the given format.

layout_formats(conn) deprecated

Retrieves current layout formats.

merge_flash(conn, enumerable)

Merges a map into the flash.

protect_from_forgery(conn, opts \\ [])

Enables CSRF protection.

put_flash(conn, key, message)

Persists a value in flash.

put_format(conn, format)

Puts the format in the connection.

put_layout(conn, layout)

Stores the layout for rendering.

put_layout_formats(conn, formats) deprecated

Sets which formats have a layout when rendering.

put_new_layout(conn, layout)

Stores the layout for rendering if one was not stored yet.

put_new_view(conn, formats)

Stores the view for rendering if one was not stored yet.

put_root_layout(conn, layout)

Stores the root layout for rendering.

put_router_url(conn, uri)

Puts the url string or `%URI{}` to be used for route generation.

put_secure_browser_headers(conn, headers \\ %{})

Put headers that improve browser security.

put_static_url(conn, uri)

Puts the URL or `%URI{}` to be used for the static url generation.

put_view(conn, formats)

Stores the view for rendering.

redirect(conn, opts)

Sends redirect response to the given url.

render(conn, template_or_assigns \\ [])

Render the given template or the default template specified by the current action with the given assigns.

render(conn, template, assigns)

Renders the given `template` and `assigns` based on the `conn` information.

root_layout(conn, format \\ nil)

Retrieves the current root layout for the given format.

router_module(conn)

Returns the router module as an atom, raises if unavailable.

scrub_params(conn, required_key)

Scrubs the parameters from the request.

send_download(conn, kind, opts \\ [])

Sends the given file or binary as a download.

status_message_from_template(template)

Generates a status message from the template name.

text(conn, data)

Sends text response.

view_module(conn, format \\ nil)

Retrieves the current view for the given format.

view_template(conn)

Returns the template name rendered in the view as a string (or nil if no template was rendered).

Types

layout()

```
@type layout() :: {module(), layout_name :: atom()} |
atom() | false
```

view()

```
@type view() :: atom()
```

Functions

accepts(conn, accepted)

```
@spec accepts(Plug.Conn.t(), [binary()]) :: Plug.Conn.t()
```

Performs content negotiation based on the available formats.

It receives a connection, a list of formats that the server is capable of rendering and then proceeds to perform content negotiation based on the request information. If the client accepts any of the given formats, the request proceeds.

If the request contains a "_format" parameter, it is considered to be the format desired by the client. If no "_format" parameter is available, this function will parse the "accept" header and find a matching format accordingly.

This function is useful when you may want to serve different content-types (such as JSON and HTML) from the same routes. However, if you always have distinct routes, you can also disable content negotiation and simply hardcode your format of choice in your route pipelines:

```
plug :put_format, "html"
```

It is important to notice that browsers have historically sent bad accept headers. For this reason, this function will default to "html" format whenever:

- the accepted list of arguments contains the "html" format
- the accept header specified more than one media type preceded or followed by the wildcard media type "`*/*`"

This function raises `Phoenix.NotAcceptableError`, which is rendered with status 406, whenever the server cannot serve a response in any of the formats expected by the client.

Examples

`accepts/2` can be invoked as a function:

```
iex> accepts(conn, ["html", "json"])
```

or used as a plug:

```
plug :accepts, ["html", "json"]
plug :accepts, ~w(html json)
```

Custom media types

It is possible to add custom media types to your Phoenix application. The first step is to teach Plug about those new media types in your `config/config.exs` file:

```
config :mime, :types, %{
  "application/vnd.api+json" => ["json-api"]
}
```

The key is the media type, the value is a list of formats the media type can be identified with. For example, by using "json-api", you will be able to use templates with extension "index.json-api" or to force a particular format in a given URL by sending "?_format=json-api".

After this change, you must recompile plug:

```
$ mix deps.clean mime --build
$ mix deps.get
```

And now you can use it in accepts too:

```
plug :accepts, ["html", "json-api"]
```

action_fallback(plug)

(macro)

Registers the plug to call as a fallback to the controller action.

A fallback plug is useful to translate common domain data structures into a valid `%Plug.Conn{}` response. If the controller action fails to return a `%Plug.Conn{}`, the provided plug will be called and receive the controller's `%Plug.Conn{}` as it was before the action was invoked along with the value returned from the controller action.

Examples

```
defmodule MyController do
  use Phoenix.Controller

  action_fallback MyFallbackController

  def show(conn, %{"id" => id}, current_user) do
    with {:ok, post} <- Blog.fetch_post(id),
         :ok <- Authorizer.authorize(current_user,
:view, post) do

      render(conn, "show.json", post: post)
    end
  end
end
```

In the above example, `with` is used to match only a successful post fetch, followed by valid authorization for the current user. In the event either of those fail to match, `with` will not invoke the render block and instead return the unmatched value. In this case, imagine `Blog.fetch_post/2` returned `{:error, :not_found}` or `Authorizer.authorize/3` returned `{:error, :unauthorized}`. For cases where these data structures serve as return values across multiple boundaries in our domain, a single fallback module can be used to translate the value into a valid response. For example, you could write the following fallback controller to handle the above values:

```
defmodule MyFallbackController do
  use Phoenix.Controller

  def call(conn, {:error, :not_found}) do
    conn
    |> put_status(:not_found)
    |> put_view(MyErrorView)
    |> render(:"404")
  end

  def call(conn, {:error, :unauthorized}) do
    conn
    |> put_status(:forbidden)
    |> put_view(MyErrorView)
    |> render(:"403")
  end
end
```

action_name(conn)

```
@spec action_name(Plug.Conn.t()) :: atom()
```

Returns the action name as an atom, raises if unavailable.

allow_jsonp(conn, opts \\ [])

```
@spec allow_jsonp(Plug.Conn.t(), Keyword.t()) ::
Plug.Conn.t()
```

A plug that may convert a JSON response into a JSONP one.

In case a JSON response is returned, it will be converted to a JSONP as long as the callback field is present in the query string. The callback field itself defaults to "callback", but may be configured with the callback option.

In case there is no callback or the response is not encoded in JSON format, it is a no-op.

Only alphanumeric characters and underscore are allowed in the callback name. Otherwise an exception is raised.

Examples

```
# Will convert JSON to JSONP if callback=someFunction
is given
plug :allow_jsonp

# Will convert JSON to JSONP if cb=someFunction is
given
plug :allow_jsonp, callback: "cb"
```

clear_flash(conn)

Clears all flash messages.

controller_module(conn)

```
@spec controller_module(Plug.Conn.t()) :: atom()
```

Returns the controller module as an atom, raises if unavailable.

current_path(conn)

Returns the current request path with its default query parameters:

```
iex> current_path(conn)
"/users/123?existing=param"
```

See `current_path/2` to override the default parameters.

The path is normalized based on the `conn.script_name` and `conn.path_info`. For example, "/foo//bar/" will become "/foo/bar". If you want the original path, use `conn.request_path` instead.

current_path(conn, params)

Returns the current path with the given query parameters.

You may also retrieve only the request path by passing an empty map of params.

Examples

```
iex> current_path(conn)
"/users/123?existing=param"

iex> current_path(conn, %{new: "param"})
"/users/123?new=param"

iex> current_path(conn, %{filter: %{status: ["draft",
"published"]}})
"/users/123?filter[status][]=draft&filter[status]
[]=published"
```

```
iex> current_path(conn, %{})
"/users/123"
```

The path is normalized based on the `conn.script_name` and `conn.path_info`. For example, "/foo//bar/" will become "/foo/bar". If you want the original path, use `conn.request_path` instead.

current_url(conn)

Returns the current request url with its default query parameters:

```
iex> current_url(conn)
"https://www.example.com/users/123?existing=param"
```

See `current_url/2` to override the default parameters.

current_url(conn, params)

Returns the current request URL with query params.

The path will be retrieved from the currently requested path via `current_path/1`. The scheme, host and others will be received from the URL configuration in your Phoenix endpoint. The reason we don't use the host and scheme information in the request is because most applications are behind proxies and the host and scheme may not actually reflect the host and scheme accessed by the client. If you want to access the url precisely as requested by the client, see `Plug.Conn.request_url/1`.

Examples

```
iex> current_url(conn)
"https://www.example.com/users/123?existing=param"

iex> current_url(conn, %{new: "param"})
"https://www.example.com/users/123?new=param"

iex> current_url(conn, %{})
"https://www.example.com/users/123"
```

Custom URL Generation

In some cases, you'll need to generate a request's URL, but using a different scheme, different host, etc. This can be accomplished in two ways.

If you want to do so in a case-by-case basis, you can define a custom function that gets the endpoint URI configuration and changes it accordingly. For example, to get the current URL always in HTTPS format:

```
def current_secure_url(conn, params \\ %{}) do
  current_uri = MyAppWeb.Endpoint.struct_url()
  current_path = Phoenix.Controller.current_path(conn,
params)

Phoenix.VerifiedRoutes.unverified_url(%URI{current_uri
| scheme: "https"}, current_path)
end
```

However, if you want all generated URLs to always have a certain schema, host, etc, you may use `put_router_url/2`.

delete_csrf_token()

Deletes the CSRF token from the process dictionary.

Note: The token is deleted only after a response has been sent.

endpoint_module(conn)

```
@spec endpoint_module(Plug.Conn.t()) :: atom()
```

Returns the endpoint module as an atom, raises if unavailable.

fetch_flash(conn, opts \\ [])

Fetches the flash storage.

get_csrf_token()

Gets or generates a CSRF token.

If a token exists, it is returned, otherwise it is generated and stored in the process dictionary.

get_flash(conn)

This function is deprecated. get_flash/1 is deprecated. Use the @flash assign provided by the :fetch_flash plug.

Returns a map of previously set flash messages or an empty map.

Examples

```
iex> get_flash(conn)
%{}

iex> conn = put_flash(conn, :info, "Welcome Back!")
iex> get_flash(conn)
%{"info" => "Welcome Back!"}
```

get_flash(conn, key)

This function is deprecated. get_flash/2 is deprecated. Use Phoenix.Flash.get(@flash, key) instead.

Returns a message from flash by `key` (or `nil` if no message is available for `key`).

Examples

```
iex> conn = put_flash(conn, :info, "Welcome Back!")
iex> get_flash(conn, :info)
"Welcome Back!"
```

get_format(conn)

Returns the request format, such as "json", "html".

This format is used when rendering a template as an atom. For example, `render(conn, :foo)` will render `"foo.FORMAT"` where the format is the one set here. The default format is typically set from the negotiation done in `accepts/2`.

html(conn, data)

```
@spec html(Plug.Conn.t(), iodata()) :: Plug.Conn.t()
```

Sends html response.

Examples

```
iex> html(conn, "<html><head>...")
```

json(conn, data)

```
@spec json(Plug.Conn.t(), term()) :: Plug.Conn.t()
```

Sends JSON response.

It uses the configured `:json_library` under the `:phoenix` application for `:json` to pick up the encoder module.

Examples

```
iex> json(conn, %{id: 123})
```

layout(conn, format \\ nil)

```
@spec layout(Plug.Conn.t(), binary() | nil) :: {atom(), String.t() | atom()} | false
```

Retrieves the current layout for the given format.

If no format is given, takes the current one from the connection.

layout_formats(conn)

This function is deprecated. layout_formats/1 is deprecated, pass a keyword list to put_layout/put_root_layout instead.

```
@spec layout_formats(Plug.Conn.t()) :: [String.t()]
```

Retrieves current layout formats.

merge_flash(conn, enumerable)

Merges a map into the flash.

Returns the updated connection.

Examples

```
iex> conn = merge_flash(conn, info: "Welcome Back!")
iex> Phoenix.Flash.get(conn.assigns.flash, :info)
"Welcome Back!"
```

protect_from_forgery(conn, opts \\ [])

Enables CSRF protection.

Currently used as a wrapper function for `Plug.CSRFProtection` and mainly serves as a function plug in `YourApp.Router`.

Check `get_csrf_token/0` and `delete_csrf_token/0` for retrieving and deleting CSRF tokens.

put_flash(conn, key, message)

Persists a value in flash.

Returns the updated connection.

Examples

```
iex> conn = put_flash(conn, :info, "Welcome Back!")
iex> Phoenix.Flash.get(conn.assigns.flash, :info)
"Welcome Back!"
```

put_format(conn, format)

Puts the format in the connection.

This format is used when rendering a template as an atom. For example, `render(conn, :foo)` will render `"foo.FORMAT"` where the format is the one set here. The default format is typically set from the negotiation done in `accepts/2`.

See `get_format/1` for retrieval.

put_layout(conn, layout)

```
@spec put_layout(Plug.Conn.t(), [{format :: atom(), layout()}] | false) ::
  Plug.Conn.t()
```

Stores the layout for rendering.

The layout must be given as keyword list where the key is the request format the layout will be applied to (such as `:html`) and the value is one of:

- `{module, layout}` with the `module` the layout is defined and the name of the `layout` as an atom
- `layout` when the name of the layout. This requires a layout for the given format in the shape of `{module, layout}` to be previously given
- `false` which disables the layout

If `false` is given without a format, all layouts are disabled.

Examples

```
iex> layout(conn)
false

iex> conn = put_layout(conn, html: {AppView,
:application})
iex> layout(conn)
{AppView, :application}

iex> conn = put_layout(conn, html: :print)
iex> layout(conn)
{AppView, :print}
```

Raises [`Plug.Conn.AlreadySentError`](#) if `conn` is already sent.

put_layout_formats(conn, formats)

This function is deprecated. put_layout_formats/2 is deprecated, pass a keyword list to put_layout/put_root_layout instead.

```
@spec put_layout_formats(Plug.Conn.t(), [String.t()]) ::
Plug.Conn.t()
```

Sets which formats have a layout when rendering.

Examples

```
iex> layout_formats(conn)
["html"]

iex> put_layout_formats(conn, ["html", "mobile"])
iex> layout_formats(conn)
["html", "mobile"]
```

Raises `Plug.Conn.AlreadySentError` if `conn` is already sent.

put_new_layout(conn, layout)

```
@spec put_new_layout(Plug.Conn.t(), [{format :: atom(),
layout()}] | layout()) ::
  Plug.Conn.t()
```

Stores the layout for rendering if one was not stored yet.

See `put_layout/2` for more information.

Raises `Plug.Conn.AlreadySentError` if `conn` is already sent.

put_new_view(conn, formats)

```
@spec put_new_view(Plug.Conn.t(), [{format :: atom(), view()}] | view()) ::
  Plug.Conn.t()
```

Stores the view for rendering if one was not stored yet.

Raises `Plug.Conn.AlreadySentError` if `conn` is already sent.

put_root_layout(conn, layout)

```
@spec put_root_layout(Plug.Conn.t(), [{format :: atom(), layout()}] | false) ::
  Plug.Conn.t()
```

Stores the root layout for rendering.

The layout must be given as keyword list where the key is the request format the layout will be applied to (such as `:html`) and the value is one of:

- `{module, layout}` with the `module` the layout is defined and the name of the `layout` as an atom
- `layout` when the name of the layout. This requires a layout for the given format in the shape of `{module, layout}` to be previously given
- `false` which disables the layout

Examples

```
iex> root_layout(conn)
false

iex> conn = put_root_layout(conn, html: {AppView,
:root})
iex> root_layout(conn)
{AppView, :root}

iex> conn = put_root_layout(conn, html: :bare)
iex> root_layout(conn)
{AppView, :bare}
```

Raises `Plug.Conn.AlreadySentError` if `conn` is already sent.

put_router_url(conn, uri)

Puts the url string or `%URI{}` to be used for route generation.

This function overrides the default URL generation pulled from the `%Plug.Conn{}`'s endpoint configuration.

Examples

Imagine your application is configured to run on "example.com" but after the user signs in, you want all links to use "some_user.example.com". You can do so by setting the proper router url configuration:

```
def put_router_url_by_user(conn) do
  put_router_url(conn,
get_user_from_conn(conn).account_name <>
".example.com")
end
```

Now when you call `Routes.some_route_url(conn, ...)`, it will use the router url set above. Keep in mind that, if you want to generate routes to the *current* domain, it is preferred to use `Routes.some_route_path` helpers, as those are always relative.

put_secure_browser_headers(conn, headers \\ %{})

Put headers that improve browser security.

It sets the following headers:

- `referrer-policy` - only send origin on cross origin requests
- `x-frame-options` - set to SAMEORIGIN to avoid clickjacking through iframes unless in the same origin
- `x-content-type-options` - set to nosniff. This requires script and style tags to be sent with proper content type
- `x-download-options` - set to noopen to instruct the browser not to open a download directly in the browser, to avoid HTML files rendering inline and accessing the security context of the application (like critical domain cookies)
- `x-permitted-cross-domain-policies` - set to none to restrict Adobe Flash Player's access to data

A custom headers map may also be given to be merged with defaults. It is recommended for custom header keys to be in lowercase, to avoid sending duplicate keys in a request. Additionally, responses with mixed-case headers served over HTTP/2 are not considered valid by common clients, resulting in dropped responses.

put_static_url(conn, uri)

Puts the URL or `%URI{}` to be used for the static url generation.

Using this function on a `%Plug.Conn{}` struct tells `static_url/2` to use the given information for URL generation instead of the `%Plug.Conn{}`'s endpoint configuration (much like `put_router_url/2` but for static URLs).

put_view(conn, formats)

```
@spec put_view(Plug.Conn.t(), [{format :: atom(), view()}]
| view()) :: Plug.Conn.t()
```

Stores the view for rendering.

Raises `Plug.Conn.AlreadySentError` if `conn` is already sent.

Examples

```
# Use single view module
iex> put_view(conn, AppView)

# Use multiple view module for content negotiation
iex> put_view(conn, html: AppHTML, json: AppJSON)
```

redirect(conn, opts)

Sends redirect response to the given url.

For security, `:to` only accepts paths. Use the `:external` option to redirect to any URL.

The response will be sent with the status code defined within the connection, via `Plug.Conn.put_status/2`. If no status code is set, a 302 response is sent.

Examples

```
iex> redirect(conn, to: "/login")

iex> redirect(conn, external: "https://elixir-
lang.org")
```

render(conn, template_or_assigns \\ [])

```
@spec render(Plug.Conn.t(), Keyword.t() | map() | binary()
| atom()) :: Plug.Conn.t()
```

Render the given template or the default template specified by the current action with the given assigns.

See `render/3` for more information.

render(conn, template, assigns)

```
@spec render(Plug.Conn.t(), binary() | atom(), Keyword.t()
| map()) :: Plug.Conn.t()
```

Renders the given `template` and `assigns` based on the `conn` information.

Once the template is rendered, the template format is set as the response content type (for example, an HTML template will set "text/html" as response content type) and the data is sent to the client with default status of 200.

Arguments

- `conn` - the `Plug.Conn` struct
- `template` - which may be an atom or a string. If an atom, like `:index`, it will render a template with the same format as the one returned by `get_format/1`. For example, for an HTML request, it will render the "index.html" template. If the template is a string, it must contain the extension too, like "index.json"
- `assigns` - a dictionary with the assigns to be used in the view. Those assigns are merged and have higher precedence than the connection assigns (`conn.assigns`)

Examples

```
defmodule MyAppWeb.UserController do
  use Phoenix.Controller

  def show(conn, _params) do
    render(conn, "show.html", message: "Hello")
  end
end
```

The example above renders a template "show.html" from the `MyAppWeb.UserView` and sets the response content type to "text/html".

In many cases, you may want the template format to be set dynamically based on the request. To do so, you can pass the template name as an atom

(without the extension):

```
def show(conn, _params) do
  render(conn, :show, message: "Hello")
end
```

In order for the example above to work, we need to do content negotiation with the accepts plug before rendering. You can do so by adding the following to your pipeline (in the router):

```
plug :accepts, ["html"]
```

Views

By default, Controllers render templates in a view with a similar name to the controller. For example, `MyAppWeb.UserController` will render templates inside the `MyAppWeb.UserView`. This information can be changed any time by using the `put_view/2` function:

```
def show(conn, _params) do
  conn
  |> put_view(MyAppWeb.SpecialView)
  |> render(:show, message: "Hello")
end
```

`put_view/2` can also be used as a plug:

```
defmodule MyAppWeb.UserController do
  use Phoenix.Controller

  plug :put_view, html: MyAppWeb.SpecialView

  def show(conn, _params) do
    render(conn, :show, message: "Hello")
  end
end
```

Layouts

Templates are often rendered inside layouts. By default, Phoenix will render layouts for html requests. For example:

```
defmodule MyAppWeb.UserController do
  use Phoenix.Controller

  def show(conn, _params) do
    render(conn, "show.html", message: "Hello")
  end
end
```

will render the "show.html" template inside an "app.html" template specified in `MyAppWeb.LayoutView`. `put_layout/2` can be used to change the layout, similar to how `put_view/2` can be used to change the view.

root_layout(conn, format \\ nil)

```
@spec root_layout(Plug.Conn.t(), binary() | nil) ::
  {atom(), String.t() | atom()} | false
```

Retrieves the current root layout for the given format.

If no format is given, takes the current one from the connection.

router_module(conn)

```
@spec router_module(Plug.Conn.t()) :: atom()
```

Returns the router module as an atom, raises if unavailable.

scrub_params(conn, required_key)

```
@spec scrub_params(Plug.Conn.t(), String.t()) ::
Plug.Conn.t()
```

Scrubs the parameters from the request.

This process is two-fold:

- Checks to see if the `required_key` is present
- Changes empty parameters of `required_key` (recursively) to nils

This function is useful for removing empty strings sent via HTML forms. If you are providing an API, there is likely no need to invoke `scrub_params/2`.

If the `required_key` is not present, it will raise `Phoenix.MissingParamError`.

Examples

```
iex> scrub_params(conn, "user")
```

send_download(conn, kind, opts \\ [])

Sends the given file or binary as a download.

The second argument must be `{:binary, contents}`, where `contents` will be sent as download, or `{:file, path}`, where `path` is the filesystem location of the file to be sent. Be careful to not interpolate the path from external parameters, as it could allow traversal of the filesystem.

The download is achieved by setting "content-disposition" to attachment. The "content-type" will also be set based on the extension of the given filename but can be customized via the `:content_type` and `:charset` options.

Options

- `:filename` - the filename to be presented to the user as download
- `:content_type` - the content type of the file or binary sent as download. It is automatically inferred from the filename extension
- `:disposition` - specifies disposition type (`:attachment` or `:inline`). If `:attachment` was used, user will be prompted to save the file. If `:inline` was used, the browser will attempt to open the file. Defaults to `:attachment`.
- `:charset` - the charset of the file, such as "utf-8". Defaults to none
- `:offset` - the bytes to offset when reading. Defaults to `0`
- `:length` - the total bytes to read. Defaults to `:all`
- `:encode` - encodes the filename using `URI.encode/2`. Defaults to `true`. When `false`, disables encoding. If you disable encoding, you need to guarantee there are no special characters in the filename, such as quotes, newlines, etc. Otherwise you can expose your application to security attacks

Examples

To send a file that is stored inside your application priv directory:

```
path = Application.app_dir(:my_app,
"priv/prospectus.pdf")
send_download(conn, {:file, path})
```

When using `{:file, path}`, the filename is inferred from the given path but may also be set explicitly.

To allow the user to download contents that are in memory as a binary or string:

```
send_download(conn, {:binary, "world"}, filename:
"hello.txt")
```

See `Plug.Conn.send_file/3` and `Plug.Conn.send_resp/3` if you would like to access the low-level functions used to send files and responses via Plug.

status_message_from_template(template)

Generates a status message from the template name.

Examples

```
iex> status_message_from_template("404.html")
"Not Found"
iex> status_message_from_template("whatever.html")
"Internal Server Error"
```

text(conn, data)

```
@spec text(Plug.Conn.t(), String.Chars.t()) ::
Plug.Conn.t()
```

Sends text response.

Examples

```
iex> text(conn, "hello")

iex> text(conn, :implements_to_string)
```

view_module(conn, format \\ nil)

```
@spec view_module(Plug.Conn.t(), binary() | nil) :: atom()
```

Retrieves the current view for the given format.

If no format is given, takes the current one from the connection.

view_template(conn)

```
@spec view_template(Plug.Conn.t()) :: binary() | nil
```

Returns the template name rendered in the view as a string (or nil if no template was rendered).

Phoenix.Endpoint behaviour

Defines a Phoenix 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:

- to provide a wrapper for starting and stopping the endpoint as part of a supervision tree
- to define an initial plug pipeline for requests to pass through
- to host web specific configuration for your application

Endpoints

An endpoint is simply a module defined with the help of `Phoenix.Endpoint`. If you have used the `mix phx.new` generator, an endpoint was automatically generated as part of your application:

```
defmodule YourAppWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :your_app

  # plug ...
  # plug ...

  plug YourApp.Router
end
```

Endpoints must be explicitly started as part of your application supervision tree. Endpoints are added by default to the supervision tree in

generated applications. Endpoints can be added to the supervision tree as follows:

```
children = [
  YourAppWeb.Endpoint
]
```

Endpoint configuration

All endpoints are configured in your application environment. For example:

```
config :your_app, YourAppWeb.Endpoint,
  secret_key_base: "kjoy3o1zeidquwy1398juxzldjlksahdk3"
```

Endpoint configuration is split into two categories. Compile-time configuration means the configuration is read during compilation and changing it at runtime has no effect. The compile-time configuration is mostly related to error handling.

Runtime configuration, instead, is accessed during or after your application is started and can be read through the `config/2` function:

```
YourAppWeb.Endpoint.config(:port)
YourAppWeb.Endpoint.config(:some_config,
:default_value)
```

Compile-time configuration

Compile-time configuration may be set on `config/dev.exs`, `config/prod.exs` and so on, but has no effect on `config/runtime.exs`:

- `:code_reloader` - when `true`, enables code reloading functionality. For the list of code reloader configuration options see

`Phoenix.CodeReloader.reload/1`. Keep in mind code reloading is based on the file-system, therefore it is not possible to run two instances of the same app at the same time with code reloading in development, as they will race each other and only one will effectively recompile the files. In such cases, tweak your config files so code reloading is enabled in only one of the apps or set the MIX_BUILD environment variable to give them distinct build directories

- `:debug_errors` - when `true`, uses `Plug.Debugger` functionality for debugging failures in the application. Recommended to be set to `true` only in development as it allows listing of the application source code during debugging. Defaults to `false`

- `:force_ssl` - ensures no data is ever sent via HTTP, always redirecting to HTTPS. It expects a list of options which are forwarded to `Plug.SSL`. By default it sets the "strict-transport-security" header in HTTPS requests, forcing browsers to always use HTTPS. If an unsafe request (HTTP) is sent, it redirects to the HTTPS version using the `:host` specified in the `:url` configuration. To dynamically redirect to the `host` of the current request, set `:host` in the `:force_ssl` configuration to `nil`

Runtime configuration

The configuration below may be set on `config/dev.exs`, `config/prod.exs` and so on, as well as on `config/runtime.exs`. Typically, if you need to configure them with system environment variables, you set them in `config/runtime.exs`. These options may also be set when starting the endpoint in your supervision tree, such as `{MyApp.Endpoint, options}`.

- `:adapter` - which webserver adapter to use for serving web requests. See the "Adapter configuration" section below

- `:cache_static_manifest` - a path to a json manifest file that contains static files and their digested version. This is typically set to

"priv/static/cache_manifest.json" which is the file automatically generated by `mix phx.digest`. It can be either: a string containing a file system path or a tuple containing the application name and the path within that application.

- `:cache_static_manifest_latest` - a map of the static files pointing to their digest version. This is automatically loaded from `cache_static_manifest` on boot. However, if you have your own static handling mechanism, you may want to set this value explicitly. This is used by projects such as `LiveView` to detect if the client is running on the latest version of all assets.

- `:cache_manifest_skip_vsn` - when true, skips the appended query string "?vsn=d" when generating paths to static assets. This query string is used by `Plug.Static` to set long expiry dates, therefore, you should set this option to true only if you are not using `Plug.Static` to serve assets, for example, if you are using a CDN. If you are setting this option, you should also consider passing `--no-vsn` to `mix phx.digest`. Defaults to `false`.

- `:check_origin` - configure the default `:check_origin` setting for transports. See `socket/3` for options. Defaults to `true`.

- `:secret_key_base` - a secret key used as a base to generate secrets for encrypting and signing data. For example, cookies and tokens are signed by default, but they may also be encrypted if desired. Defaults to `nil` as it must be set per application

- `:server` - when `true`, starts the web server when the endpoint supervision tree starts. Defaults to `false`. The `mix phx.server` task automatically sets this to `true`

- `:url` - configuration for generating URLs throughout the app. Accepts the `:host`, `:scheme`, `:path` and `:port` options. All keys except `:path` can be changed at runtime. Defaults to:

```
[host: "localhost", path: "/"]
```

The `:port` option requires either an integer or string. The `:host` option requires a string.

The `:scheme` option accepts `"http"` and `"https"` values. Default value is inferred from top level `:http` or `:https` option. It is useful when hosting Phoenix behind a load balancer or reverse proxy and terminating SSL there.

The `:path` option can be used to override root path. Useful when hosting Phoenix behind a reverse proxy with URL rewrite rules

- `:static_url` - configuration for generating URLs for static files. It will fallback to `url` if no option is provided. Accepts the same options as `url`

- `:watchers` - a set of watchers to run alongside your server. It expects a list of tuples containing the executable and its arguments. Watchers are guaranteed to run in the application directory, but only when the server is enabled (unless `:force_watchers` configuration is set to `true`). For example, the watcher below will run the "watch" mode of the webpack build tool when the server starts. You can configure it to whatever build tool or command you want:

```
[
  node: [
    "node_modules/webpack/bin/webpack.js",
    "--mode",
    "development",
    "--watch",
    "--watch-options-stdin"
  ]
]
```

The `:cd` and `:env` options can be given at the end of the list to customize the watcher:

```
[node: [..., cd: "assets", env: [{"TAILWIND_MODE",
"watch"}]]]
```

A watcher can also be a module-function-args tuple that will be invoked accordingly:

```
[another: {Mod, :fun, [arg1, arg2]}]
```

- `:force_watchers` - when `true`, forces your watchers to start even when the `:server` option is set to `false`.

- `:live_reload` - configuration for the live reload option. Configuration requires a `:patterns` option which should be a list of file patterns to watch. When these files change, it will trigger a reload.

```
live_reload: [
  url: "ws://localhost:4000",
  patterns: [
    ~r"priv/static/(?!uploads/).*
(js|css|png|jpeg|jpg|gif|svg)$",
    ~r"lib/app_web/(live|views)/.*(ex)$",
    ~r"lib/app_web/templates/.*(eex)$"
  ]
]
```

- `:pubsub_server` - the name of the pubsub server to use in channels and via the Endpoint broadcast functions. The PubSub server is typically started in your supervision tree.

- `:render_errors` - responsible for rendering templates whenever there is a failure in the application. For example, if the application

crashes with a 500 error during a HTML request, `render("500.html", assigns)` will be called in the view given to `:render_errors`. A `:formats` list can be provided to specify a module per format to handle error rendering. Example:

```
[formats: [html: MyApp.ErrorHTML], layout: false,
log: :debug]
```

- `:log_access_url` - log the access url once the server boots

Note that you can also store your own configurations in the Phoenix.Endpoint. For example, Phoenix LiveView expects its own configuration under the `:live_view` key. In such cases, you should consult the documentation of the respective projects.

Adapter configuration

Phoenix allows you to choose which webserver adapter to use. Newly generated applications created via the `phx.new` Mix task use the `Bandit` webserver via the `Bandit.PhoenixAdapter` adapter. If not otherwise specified via the `adapter` option Phoenix will fall back to the `Phoenix.Endpoint.Cowboy2Adapter` for backwards compatibility with applications generated prior to Phoenix 1.7.8.

Both adapters can be configured in a similar manner using the following two top-level options:

- `:http` - the configuration for the HTTP server. It accepts all options as defined by either `Bandit` or `Plug.Cowboy` depending on your choice of adapter. Defaults to `false`

- `:https` - the configuration for the HTTPS server. It accepts all options as defined by either `Bandit` or `Plug.Cowboy` depending on your choice of adapter. Defaults to `false`

In addition, the connection draining can be configured for the Cowboy webserver via the following top-level option (this is not required for Bandit as it has connection draining built-in):

- `:drainer` - a drainer process waits for any on-going request to finish during application shutdown. It accepts the `:shutdown` and `:check_interval` options as defined by `Plug.Cowboy.Drainer`. Note the draining does not terminate any existing connection, it simply waits for them to finish. Socket connections run their own drainer before this one is invoked. That's because sockets are stateful and can be gracefully notified, which allows us to stagger them over a longer period of time. See the documentation for `socket/3` for more information

Endpoint API

In the previous section, we have used the `config/2` function that is automatically generated in your endpoint. Here's a list of all the functions that are automatically defined in your endpoint:

- for handling paths and URLs: `struct_url/0`, `url/0`, `path/1`, `static_url/0`, `static_path/1`, and `static_integrity/1`
- for gathering runtime information about the address and port the endpoint is running on: `server_info/1`
- for broadcasting to channels: `broadcast/3`, `broadcast!/3`, `broadcast_from/4`, `broadcast_from!/4`, `local_broadcast/3`, and `local_broadcast_from/4`
- for configuration: `start_link/1`, `config/2`, and `config_change/2`
- as required by the `Plug` behaviour: `Plug.init/1` and `Plug.call/2`

Summary

Types

Callbacks

Broadcasts a `msg` as `event` in the given `topic` to all nodes.

Broadcasts a `msg` as `event` in the given `topic` to all nodes.

Broadcasts a `msg` from the given `from` as `event` in the given `topic` to all nodes.

Broadcasts a `msg` from the given `from` as `event` in the given `topic` to all nodes.

Access the endpoint configuration given by key.

Reload the endpoint configuration on application upgrades.

host()

Returns the host from the :url configuration.

local_broadcast(topic, event, msg)

Broadcasts a `msg` as `event` in the given `topic` within the current node.

local_broadcast_from(from, topic, event, msg)

Broadcasts a `msg` from the given `from` as `event` in the given `topic` within the current node.

path(path)

Generates the path information when routing to this endpoint.

script_name()

Returns the script name from the :url configuration.

server_info(scheme)

Returns the address and port that the server is running on

start_link(keyword)

Starts the endpoint supervision tree.

static_integrity(path)

Generates an integrity hash to a static file in `priv/static`.

static_lookup(path)

Generates a two item tuple containing the `static_path` and `static_integrity`.

static_path(path)

Generates a route to a static file in `priv/static`.

static_url()

Generates the static URL without any path information.

struct_url()

Generates the endpoint base URL, but as a `URI` struct.

subscribe(topic, opts)

Subscribes the caller to the given topic.

unsubscribe(topic)

Unsubscribes the caller from the given topic.

url()

Generates the endpoint base URL without any path information.

Functions

server?(otp_app, endpoint)

Checks if Endpoint's web server has been configured to start.

socket(path, module, opts \\ [])

Defines a websocket/longpoll mount-point for a `socket`.

Types

event()

```
@type event() :: String.t()
```

msg()

```
@type msg() :: map() | {:binary, binary()}
```

topic()

```
@type topic() :: String.t()
```

Callbacks

broadcast(topic, event, msg)

```
@callback broadcast(topic(), event(), msg()) :: :ok |
{:error, term()}
```

Broadcasts a `msg` as `event` in the given `topic` to all nodes.

broadcast!(topic, event, msg)

```
@callback broadcast!(topic(), event(), msg()) :: :ok
```

Broadcasts a `msg` as `event` in the given `topic` to all nodes.

Raises in case of failures.

broadcast_from(from, topic, event, msg)

```
@callback broadcast_from(from :: pid(), topic(), event(),
msg()) :: :ok | {:error, term()}
```

Broadcasts a `msg` from the given `from` as `event` in the given `topic` to all nodes.

broadcast_from!(from, topic, event, msg)

```
@callback broadcast_from!(from :: pid(), topic(), event(),
msg()) :: :ok
```

Broadcasts a `msg` from the given `from` as `event` in the given `topic` to all nodes.

Raises in case of failures.

config key, default

```
@callback config(key :: atom(), default :: term()) ::
term()
```

Access the endpoint configuration given by key.

config_change(changed, removed)

```
@callback config_change(changed :: term(), removed ::
term()) :: term()
```

Reload the endpoint configuration on application upgrades.

host()

```
@callback host() :: String.t()
```

Returns the host from the :url configuration.

local_broadcast(topic, event, msg)

```
@callback local_broadcast(topic(), event(), msg()) :: :ok
```

Broadcasts a `msg` as `event` in the given `topic` within the current node.

local_broadcast_from(from, topic, event, msg)

```
@callback local_broadcast_from(from :: pid(), topic(),
event(), msg()) :: :ok
```

Broadcasts a `msg` from the given `from` as `event` in the given `topic` within the current node.

path(path)

```
@callback path(path :: String.t()) :: String.t()
```

Generates the path information when routing to this endpoint.

script_name()

```
@callback script_name() :: [String.t()]
```

Returns the script name from the :url configuration.

server_info(scheme)

```
@callback server_info(Plug.Conn.scheme()) ::
  {:ok,
```

```
    {:inet.ip_address(), :inet.port_number()} |
:inet.returned_non_ip_address()}
  | {:error, term()}
```

Returns the address and port that the server is running on

start_link(keyword)

```
@callback start_link(keyword()) :: Supervisor.on_start()
```

Starts the endpoint supervision tree.

Starts endpoint's configuration cache and possibly the servers for handling requests.

static_integrity(path)

```
@callback static_integrity(path :: String.t()) ::
String.t() | nil
```

Generates an integrity hash to a static file in `priv/static`.

static_lookup(path)

```
@callback static_lookup(path :: String.t()) ::
  {String.t(), String.t()} | {String.t(), nil}
```

Generates a two item tuple containing the `static_path` and `static_integrity`.

static_path(path)

```
@callback static_path(path :: String.t()) :: String.t()
```

Generates a route to a static file in `priv/static`.

static_url()

```
@callback static_url() :: String.t()
```

Generates the static URL without any path information.

struct_url()

```
@callback struct_url() :: URI.t()
```

Generates the endpoint base URL, but as a `URI` struct.

subscribe(topic, opts)

```
@callback subscribe(topic(), opts :: Keyword.t()) :: :ok |
{:error, term()}
```

Subscribes the caller to the given topic.

See `Phoenix.PubSub.subscribe/3` for options.

unsubscribe(topic)

```
@callback unsubscribe(topic()) :: :ok | {:error, term()}
```

Unsubscribes the caller from the given topic.

url()

```
@callback url() :: String.t()
```

Generates the endpoint base URL without any path information.

Functions

server?(otp_app, endpoint)

Checks if Endpoint's web server has been configured to start.

- `otp_app` - The OTP app running the endpoint, for example `:my_app`
- `endpoint` - The endpoint module, for example `MyAppWeb.Endpoint`

Examples

```
iex> Phoenix.Endpoint.server?(:my_app,
MyAppWeb.Endpoint)
true
```

socket(path, module, opts \\ [])

(macro)

Defines a websocket/longpoll mount-point for a `socket`.

It expects a `path`, a `socket` module, and a set of options. The socket module is typically defined with `Phoenix.Socket`.

Both websocket and longpolling connections are supported out of the box.

Options

- `:websocket` - controls the websocket configuration. Defaults to `true`. May be false or a keyword list of options. See "Common configuration" and "WebSocket configuration" for the whole list

- `:longpoll` - controls the longpoll configuration. Defaults to `false`. May be true or a keyword list of options. See "Common configuration" and "Longpoll configuration" for the whole list

- `:drainer` - a keyword list or a custom MFA function returning a keyword list, for example:

```
{MyAppWeb.Socket, :drainer_configuration, []}
```

configuring how to drain sockets on application shutdown. The goal is to notify all channels (and LiveViews) clients to reconnect. The supported options are:

- `:batch_size` - How many clients to notify at once in a given batch. Defaults to 10000.
- `:batch_interval` - The amount of time in milliseconds given for a batch to terminate. Defaults to 2000ms.
- `:shutdown` - The maximum amount of time in milliseconds allowed to drain all batches. Defaults to 30000ms.
- `:log` - the log level for drain actions. Defaults the `:log` option passed to `use Phoenix.Socket` or `:info`. Set it to `false` to disable logging.

For example, if you have 150k connections, the default values will split them into 15 batches of 10k connections. Each batch takes 2000ms before the next batch starts. In this case, we will do everything right under the maximum shutdown time of 30000ms. Therefore, as you increase the number of connections, remember to adjust the shutdown accordingly. Finally, after the socket drainer runs, the lower level HTTP/HTTPS connection drainer will still run, and apply to all connections. Set it to `false` to disable draining.

You can also pass the options below on `use Phoenix.Socket`. The values specified here override the value in `use Phoenix.Socket`.

Examples

```
socket "/ws", MyApp.UserSocket

socket "/ws/admin", MyApp.AdminUserSocket,
  longpoll: true,
  websocket: [compress: true]
```

Path params

It is possible to include variables in the path, these will be available in the `params` that are passed to the socket.

```
socket "/ws/:user_id", MyApp.UserSocket,
  websocket: [path: "/project/:project_id"]
```

Common configuration

The configuration below can be given to both `:websocket` and `:longpoll` keys:

- `:path` - the path to use for the transport. Will default to the transport name ("/websocket" or "/longpoll")
- `:serializer` - a list of serializers for messages. See `Phoenix.Socket` for more information
- `:transport_log` - if the transport layer itself should log and, if so, the level
- `:check_origin` - if the transport should check the origin of requests when the `origin` header is present. May be `true`, `false`, a list of hosts that are allowed, or a function provided as MFA tuple. Defaults to `:check_origin` setting at endpoint configuration.

  If `true`, the header is checked against `:host` in `YourAppWeb.Endpoint.config(:url)[:host]`.

  If `false` and you do not validate the session in your socket, your app is vulnerable to Cross-Site WebSocket Hijacking (CSWSH) attacks. Only use in development, when the host is truly unknown or when serving clients that do not send the `origin` header, such as mobile apps.

You can also specify a list of explicitly allowed origins. Wildcards are supported.

```
check_origin: [
  "https://example.com",
  "//another.com:888",
  "//*.other.com"
]
```

Or to accept any origin matching the request connection's host, port, and scheme:

```
check_origin: :conn
```

Or a custom MFA function:

```
check_origin: {MyAppWeb.Auth, :my_check_origin?,
[]}
```

The MFA is invoked with the request `%URI{}` as the first argument, followed by arguments in the MFA list, and must return a boolean.

- `:code_reloader` - enable or disable the code reloader. Defaults to your endpoint configuration
- `:connect_info` - a list of keys that represent data to be copied from the transport to be made available in the user socket `connect/3` callback. See the "Connect info" subsection for valid keys

Connect info

The valid keys are:

- `:peer_data` - the result of `Plug.Conn.get_peer_data/1`

- `:trace_context_headers` - a list of all trace context headers. Supported headers are defined by the W3C Trace Context Specification. These headers are necessary for libraries such as OpenTelemetry to extract trace propagation information to know this request is part of a larger trace in progress.

- `:x_headers` - all request headers that have an "x-" prefix

- `:uri` - a `%URI{}` with information from the conn

- `:user_agent` - the value of the "user-agent" request header

- `{:session, session_config}` - the session information from `Plug.Conn`. The `session_config` is typically an exact copy of the arguments given to `Plug.Session`. In order to validate the session, the "_csrf_token" must be given as request parameter when connecting the socket with the value of `URI.encode_www_form(Plug.CSRFProtection.get_csrf_token())`. The CSRF token request parameter can be modified via the `:csrf_token_key` option.

  Additionally, `session_config` may be a MFA, such as `{MyAppWeb.Auth, :get_session_config, []}`, to allow loading config in runtime.

Arbitrary keywords may also appear following the above valid keys, which is useful for passing custom connection information to the socket.

For example:

```
  socket "/socket", AppWeb.UserSocket,
      websocket: [
        connect_info: [:peer_data,
:trace_context_headers, :x_headers, :uri, session:
[store: :cookie]]
      ]
```

With arbitrary keywords:

```
socket "/socket", AppWeb.UserSocket,
    websocket: [
      connect_info: [:uri, custom_value: "abcdef"]
    ]
```

> **Where are my headers?**
>
> Phoenix only gives you limited access to the connection headers for security reasons. WebSockets are cross-domain, which means that, when a user "John Doe" visits a malicious website, the malicious website can open up a WebSocket connection to your application, and the browser will gladly submit John Doe's authentication/cookie information. If you were to accept this information as is, the malicious website would have full control of a WebSocket connection to your application, authenticated on John Doe's behalf.
>
> To safe-guard your application, Phoenix limits and validates the connection information your socket can access. This means your application is safe from these attacks, but you can't access cookies and other headers in your socket. You may access the session stored in the connection via the `:connect_info` option, provided you also pass a csrf token when connecting over WebSocket.

Websocket configuration

The following configuration applies only to `:websocket`.

- `:timeout` - the timeout for keeping websocket connections open after it last received data, defaults to 60_000ms
- `:max_frame_size` - the maximum allowed frame size in bytes, defaults to "infinity"
- `:fullsweep_after` - the maximum number of garbage collections before forcing a fullsweep for the socket process. You can set it to `0` to force more frequent cleanups of your websocket transport processes. Setting this option requires Erlang/OTP 24

- `:compress` - whether to enable per message compression on all data frames, defaults to false

- `:subprotocols` - a list of supported websocket subprotocols. Used for handshake `Sec-WebSocket-Protocol` response header, defaults to nil.

  For example:

  ```
  subprotocols: ["sip", "mqtt"]
  ```

- `:error_handler` - custom error handler for connection errors. If `Phoenix.Socket.connect/3` returns an `{:error, reason}` tuple, the error handler will be called with the error reason. For WebSockets, the error handler must be a MFA tuple that receives a `Plug.Conn`, the error reason, and returns a `Plug.Conn` with a response. For example:

  ```
  socket "/socket", MySocket,
      websocket: [
        error_handler: {MySocket, :handle_error, []}
      ]
  ```

  and a `{:error, :rate_limit}` return may be handled on `MySocket` as:

  ```
  def handle_error(conn, :rate_limit), do:
  Plug.Conn.send_resp(conn, 429, "Too many requests")
  ```

Longpoll configuration

The following configuration applies only to `:longpoll`:

- `:window_ms` - how long the client can wait for new messages in its poll request in milliseconds (ms). Defaults to `10_000`.

- `:pubsub_timeout_ms` - how long a request can wait for the pubsub layer to respond in milliseconds (ms). Defaults to `2000`.

- `:crypto` - options for verifying and signing the token, accepted by `Phoenix.Token`. By default tokens are valid for 2 weeks

Phoenix.Flash

Provides shared flash access.

Summary

Functions

get(flash, key)

Gets the key from the map of flash data.

Functions

get(flash, key)

Gets the key from the map of flash data.

Examples

```
<div id="info"><%= Phoenix.Flash.get(@flash, :info) %>
</div>
<div id="error"><%= Phoenix.Flash.get(@flash, :error)
%></div>
```

Phoenix.Logger

Instrumenter to handle logging of various instrumentation events.

Instrumentation

Phoenix uses the `:telemetry` library for instrumentation. The following events are published by Phoenix with the following measurements and metadata:

- `[:phoenix, :endpoint, :init]` - dispatched by `Phoenix.Endpoint` after your Endpoint supervision tree successfully starts
  - Measurement: `%{system_time: system_time}`
  - Metadata: `%{pid: pid(), config: Keyword.t(), module: module(), otp_app: atom()}`
  - Disable logging: This event is not logged
- `[:phoenix, :endpoint, :start]` - dispatched by `Plug.Telemetry` in your endpoint, usually after code reloading
  - Measurement: `%{system_time: system_time}`
  - Metadata: `%{conn: Plug.Conn.t, options: Keyword.t}`
  - Options: `%{log: Logger.level | false}`
  - Disable logging: In your endpoint `plug Plug.Telemetry, ..., log: Logger.level | false`
  - Configure log level dynamically: `plug Plug.Telemetry, ..., log: {Mod, Fun, Args}`
- `[:phoenix, :endpoint, :stop]` - dispatched by `Plug.Telemetry` in your endpoint whenever the response is sent

    - Measurement: `%{duration: native_time}`
    - Metadata: `%{conn: Plug.Conn.t, options: Keyword.t}`
    - Options: `%{log: Logger.level | false}`
    - Disable logging: In your endpoint `plug Plug.Telemetry, ..., log: Logger.level | false`
    - Configure log level dynamically: `plug Plug.Telemetry, ..., log: {Mod, Fun, Args}`

- `[:phoenix, :router_dispatch, :start]` - dispatched by `Phoenix.Router` before dispatching to a matched route
    - Measurement: `%{system_time: System.system_time}`
    - Metadata: `%{conn: Plug.Conn.t, route: binary, plug: module, plug_opts: term, path_params: map, pipe_through: [atom], log: Logger.level | false}`
    - Disable logging: Pass `log: false` to the router macro, for example: `get("/page", PageController, :index, log: false)`
    - Configure log level dynamically: `get("/page", PageController, :index, log: {Mod, Fun, Args})`

- `[:phoenix, :router_dispatch, :exception]` - dispatched by `Phoenix.Router` after exceptions on dispatching a route
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{conn: Plug.Conn.t, kind: :throw | :error | :exit, reason: term(), stacktrace: Exception.stacktrace()}`
    - Disable logging: This event is not logged

- `[:phoenix, :router_dispatch, :stop]` - dispatched by `Phoenix.Router` after successfully dispatching a matched route
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{conn: Plug.Conn.t, route: binary, plug: module, plug_opts: term, path_params: map, pipe_through: [atom], log: Logger.level | false}`
    - Disable logging: This event is not logged
- `[:phoenix, :error_rendered]` - dispatched at the end of an error view being rendered
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{conn: Plug.Conn.t, status: Plug.Conn.status, kind: Exception.kind, reason: term, stacktrace: Exception.stacktrace}`
    - Disable logging: Set `render_errors: [log: false]` on your endpoint configuration
- `[:phoenix, :socket_connected]` - dispatched by `Phoenix.Socket`, at the end of a socket connection
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{endpoint: atom, transport: atom, params: term, connect_info: map, vsn: binary, user_socket: atom, result: :ok | :error, serializer: atom, log: Logger.level | false}`
    - Disable logging: `use Phoenix.Socket, log: false` or `socket "/foo", MySocket, websocket: [log: false]` in your endpoint
- `[:phoenix, :socket_drain]` - dispatched by `Phoenix.Socket` when using the `:drainer` option

    - Measurement: `%{count: integer, total: integer, index: integer, rounds: integer}`
    - Metadata: `%{endpoint: atom, socket: atom, intervasl: integer, log: Logger.level | false}`
    - Disable logging: `use Phoenix.Socket, log: false` in your endpoint or pass `:log` option in the `:drainer` option

- `[:phoenix, :channel_joined]` - dispatched at the end of a channel join
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{result: :ok | :error, params: term, socket: Phoenix.Socket.t}`
    - Disable logging: This event cannot be disabled

- `[:phoenix, :channel_handled_in]` - dispatched at the end of a channel handle in
    - Measurement: `%{duration: native_time}`
    - Metadata: `%{event: binary, params: term, socket: Phoenix.Socket.t}`
    - Disable logging: This event cannot be disabled

To see an example of how Phoenix LiveDashboard uses these events to create metrics, visit https://hexdocs.pm/phoenix_live_dashboard/metrics.html.

Parameter filtering

When logging parameters, Phoenix can filter out sensitive parameters such as passwords and tokens. Parameters to be filtered can be added via the `:filter_parameters` option:

```
config :phoenix, :filter_parameters, ["password",
"secret"]
```

With the configuration above, Phoenix will filter any parameter that contains the terms `password` or `secret`. The match is case sensitive.

Phoenix's default is `["password"]`.

Phoenix can filter all parameters by default and selectively keep parameters. This can be configured like so:

```
config :phoenix, :filter_parameters, {:keep, ["id",
"order"]}
```

With the configuration above, Phoenix will filter all parameters, except those that match exactly `id` or `order`. If a kept parameter matches, all parameters nested under that one will also be kept.

Dynamic log level

In some cases you may wish to set the log level dynamically on a per-request basis. To do so, set the `:log` option to a tuple, `{Mod, Fun, Args}`. The `Plug.Conn.t()` for the request will be prepended to the provided list of arguments.

When invoked, your function must return a `Logger.level()` or `false` to disable logging for the request.

For example, in your Endpoint you might do something like this:

```
# lib/my_app_web/endpoint.ex
plug Plug.Telemetry,
  event_prefix: [:phoenix, :endpoint],
  log: {__MODULE__, :log_level, []}

# Disables logging for routes like /status/*
```

```
  def log_level(%{path_info: ["status" | _]}), do:
false
  def log_level(_), do: :info
```

Disabling

When you are using custom logging system it is not always desirable to enable `Phoenix.Logger` by default. You can always disable this in general by:

```
config :phoenix, :logger, false
```

Phoenix.Naming

Conveniences for inflecting and working with names in Phoenix.

Summary

Functions

Functions

camelize(value)

```
@spec camelize(String.t()) :: String.t()
```

Converts a string to camel case.

Takes an optional `:lower` flag to return lowerCamelCase.

Examples

```
iex> Phoenix.Naming.camelize("my_app")
"MyApp"

iex> Phoenix.Naming.camelize("my_app", :lower)
"myApp"
```

In general, `camelize` can be thought of as the reverse of `underscore`, however, in some cases formatting may be lost:

```
Phoenix.Naming.underscore "SAPExample"  #=>
"sap_example"
Phoenix.Naming.camelize   "sap_example" #=>
"SapExample"
```

camelize(value, atom)

```
@spec camelize(String.t(), :lower) :: String.t()
```

humanize(atom)

```
@spec humanize(atom() | String.t()) :: String.t()
```

Converts an attribute/form field into its humanize version.

Examples

```
iex> Phoenix.Naming.humanize(:username)
"Username"
iex> Phoenix.Naming.humanize(:created_at)
"Created at"
iex> Phoenix.Naming.humanize("user_id")
"User"
```

resource_name(alias, suffix \\ "")

```
@spec resource_name(String.Chars.t(), String.t()) ::
String.t()
```

Extracts the resource name from an alias.

Examples

```
iex> Phoenix.Naming.resource_name(MyApp.User)
"user"

iex> Phoenix.Naming.resource_name(MyApp.UserView,
"View")
"user"
```

underscore(value)

```
@spec underscore(String.t()) :: String.t()
```

Converts a string to underscore case.

Examples

```
iex> Phoenix.Naming.underscore("MyApp")
"my_app"
```

In general, `underscore` can be thought of as the reverse of `camelize`, however, in some cases formatting may be lost:

```
Phoenix.Naming.underscore "SAPExample"  #=>
"sap_example"
Phoenix.Naming.camelize   "sap_example" #=>
"SapExample"
```

unsuffix(value, suffix)

```
@spec unsuffix(String.t(), String.t()) :: String.t()
```

Removes the given suffix from the name if it exists.

Examples

```
iex> Phoenix.Naming.unsuffix("MyApp.User", "View")
"MyApp.User"
```

```
iex> Phoenix.Naming.unsuffix("MyApp.UserView", "View")
"MyApp.User"
```

Phoenix.Param protocol

A protocol that converts data structures into URL parameters.

This protocol is used by URL helpers and other parts of the Phoenix stack. For example, when you write:

```
user_path(conn, :edit, @user)
```

Phoenix knows how to extract the `:id` from `@user` thanks to this protocol.

By default, Phoenix implements this protocol for integers, binaries, atoms, and structs. For structs, a key `:id` is assumed, but you may provide a specific implementation.

Nil values cannot be converted to param.

Custom parameters

In order to customize the parameter for any struct, one can simply implement this protocol.

However, for convenience, this protocol can also be derivable. For example:

```
defmodule User do
  @derive Phoenix.Param
  defstruct [:id, :username]
end
```

By default, the derived implementation will also use the `:id` key. In case the user does not contain an `:id` key, the key can be specified with an

option:

```
defmodule User do
  @derive {Phoenix.Param, key: :username}
  defstruct [:username]
end
```

will automatically use `:username` in URLs.

When using Ecto, you must call `@derive` before your `schema` call:

```
@derive {Phoenix.Param, key: :username}
schema "users" do
```

Summary

Types

t()

All the types that implement this protocol.

Functions

to_param(term)

Types

t()

```
@type t() :: term()
```

All the types that implement this protocol.

Functions

to_param(term)

```
@spec to_param(term()) :: String.t()
```

Phoenix.Presence behaviour

Provides Presence tracking to processes and channels.

This behaviour provides presence features such as fetching presences for a given topic, as well as handling diffs of join and leave events as they occur in real-time. Using this module defines a supervisor and a module that implements the `Phoenix.Tracker` behaviour that uses `Phoenix.PubSub` to broadcast presence updates.

In case you want to use only a subset of the functionality provided by `Phoenix.Presence`, such as tracking processes but without broadcasting updates, we recommend that you look at the `Phoenix.Tracker` functionality from the `phoenix_pubsub` project.

Example Usage

Start by defining a presence module within your application which uses `Phoenix.Presence` and provide the `:otp_app` which holds your configuration, as well as the `:pubsub_server`.

```
defmodule MyAppWeb.Presence do
  use Phoenix.Presence,
    otp_app: :my_app,
    pubsub_server: MyApp.PubSub
end
```

The `:pubsub_server` must point to an existing pubsub server running in your application, which is included by default as `MyApp.PubSub` for new applications.

Next, add the new supervisor to your supervision tree in `lib/my_app/application.ex`. It must be after the PubSub child and before the endpoint:

```
children = [
  ...
  {Phoenix.PubSub, name: MyApp.PubSub},
  MyAppWeb.Presence,
  MyAppWeb.Endpoint
]
```

Once added, presences can be tracked in your channel after joining:

```
defmodule MyAppWeb.MyChannel do
  use MyAppWeb, :channel
  alias MyAppWeb.Presence

  def join("some:topic", _params, socket) do
    send(self(), :after_join)
    {:ok, assign(socket, :user_id, ...)}
  end

  def handle_info(:after_join, socket) do
    {:ok, _} = Presence.track(socket,
socket.assigns.user_id, %{
      online_at: inspect(System.system_time(:second))
    })

    push(socket, "presence_state",
Presence.list(socket))
    {:noreply, socket}
  end
end
```

In the example above, `Presence.track` is used to register this channel's process as a presence for the socket's user ID, with a map of metadata. Next, the current presence information for the socket's topic is pushed to the client as a `"presence_state"` event.

Finally, a diff of presence join and leave events will be sent to the client as they happen in real-time with the "presence_diff" event. The diff structure will be a map of `:joins` and `:leaves` of the form:

```
%{
  joins: %{"123" => %{metas: [%{status: "away",
phx_ref: ...}]}},
  leaves: %{"456" => %{metas: [%{status: "online",
phx_ref: ...}]}}
},
```

See `list/1` for more information on the presence data structure.

Fetching Presence Information

Presence metadata should be minimized and used to store small, ephemeral state, such as a user's "online" or "away" status. More detailed information, such as user details that need to be fetched from the database, can be achieved by overriding the `fetch/2` function.

The `fetch/2` callback is triggered when using `list/1` and on every update, and it serves as a mechanism to fetch presence information a single time, before broadcasting the information to all channel subscribers. This prevents N query problems and gives you a single place to group isolated data fetching to extend presence metadata.

The function must return a map of data matching the outlined Presence data structure, including the `:metas` key, but can extend the map of information to include any additional information. For example:

```
def fetch(_topic, presences) do
  users = presences |> Map.keys() |>
Accounts.get_users_map()

  for {key, %{metas: metas}} <- presences, into: %{} do
    {key, %{metas: metas, user:
users[String.to_integer(key)]}}
  end
end
```

Where `Account.get_users_map/1` could be implemented like:

```
def get_users_map(ids) do
  query =
    from u in User,
      where: u.id in ^ids,
      select: {u.id, u}

  query |> Repo.all() |> Enum.into(%{})
end
```

The `fetch/2` function above fetches all users from the database who have registered presences for the given topic. The presences information is then extended with a `:user` key of the user's information, while maintaining the required `:metas` field from the original presence data.

Using Elixir as a Presence Client

Presence is great for external clients, such as JavaScript applications, but it can also be used from an Elixir client process to keep track of presence changes as they happen on the server. This can be accomplished by implementing the optional `init/1` and `handle_metas/4` callbacks on your presence module. For example, the following callback receives presence metadata changes, and broadcasts to other Elixir processes about users joining and leaving:

```
defmodule MyApp.Presence do
  use Phoenix.Presence,
    otp_app: :my_app,
    pubsub_server: MyApp.PubSub

  def init(_opts) do
    {:ok, %{}} # user-land state
  end

  def handle_metas(topic, %{joins: joins, leaves:
leaves}, presences, state) do
    # fetch existing presence information for the
joined users and broadcast the
    # event to all subscribers
    for {user_id, presence} <- joins do
```

```
      user_data = %{user: presence.user, metas:
Map.fetch!(presences, user_id)}
      msg = {MyApp.PresenceClient, {:join, user_data}}
      Phoenix.PubSub.local_broadcast(MyApp.PubSub,
topic, msg)
    end

    # fetch existing presence information for the left
users and broadcast the
    # event to all subscribers
    for {user_id, presence} <- leaves do
      metas =
        case Map.fetch(presences, user_id) do
          {:ok, presence_metas} -> presence_metas
          :error -> []
        end

      user_data = %{user: presence.user, metas: metas}
      msg = {MyApp.PresenceClient, {:leave, user_data}}
      Phoenix.PubSub.local_broadcast(MyApp.PubSub,
topic, msg)
    end

    {:ok, state}
  end
end
```

The `handle_metas/4` callback receives the topic, presence diff, current presences for the topic with their metadata, and any user-land state accumulated from init and subsequent `handle_metas/4` calls. In our example implementation, we walk the `:joins` and `:leaves` in the diff, and populate a complete presence from our known presence information. Then we broadcast to the local node subscribers about user joins and leaves.

Testing with Presence

Every time the `fetch` callback is invoked, it is done from a separate process. Given those processes run asynchronously, it is often necessary to guarantee they have been shutdown at the end of every test. This can be done by using ExUnit's `on_exit` hook plus `fetchers_pids` function:

```
on_exit(fn ->
  for pid <- MyAppWeb.Presence.fetchers_pids() do
    ref = Process.monitor(pid)
    assert_receive {:DOWN, ^ref, _, _, _}, 1000
  end
end)
```

Summary

Types

Callbacks

Extend presence information with additional data.

Returns the map of presence metadata for a socket/topic-key pair.

Receives presence metadata changes.

Initializes the presence client state.

list(socket_or_topic)

Returns presences for a socket/topic.

track(socket, key, meta)

Track a channel's process as a presence.

track(pid, topic, key, meta)

Track an arbitrary process as a presence.

untrack(socket, key)

Stop tracking a channel's process.

untrack(pid, topic, key)

Stop tracking a process.

update(socket, key, meta)

Update a channel presence's metadata.

update(pid, topic, key, meta)

Update a process presence's metadata.

Types

presence()

```
@type presence() :: %{key: String.t(), meta: map()}
```

presences()

```
@type presences() :: %{required(String.t()) => %{metas:
[map()]}}
```

topic()

```
@type topic() :: String.t()
```

Callbacks

fetch(topic, presences)

```
@callback fetch(topic(), presences()) :: presences()
```

Extend presence information with additional data.

When `list/1` is used to list all presences of the given `topic`, this callback is triggered once to modify the result before it is broadcasted to all channel subscribers. This avoids N query problems and provides a single place to extend presence metadata. You must return a map of data matching the original result, including the `:metas` key, but can extend the map to include any additional information.

The default implementation simply passes `presences` through unchanged.

Example

```
def fetch(_topic, presences) do
  query =
    from u in User,
      where: u.id in ^Map.keys(presences),
      select: {u.id, u}

  users = query |> Repo.all() |> Enum.into(%{})
  for {key, %{metas: metas}} <- presences, into: %{} do
    {key, %{metas: metas, user: users[key]}}
  end
end
```

get_by_key(arg1, key)

```
@callback get_by_key(Phoenix.Socket.t() | topic(), key ::
String.t()) :: [presence()]
```

Returns the map of presence metadata for a socket/topic-key pair.

Examples

Uses the same data format as each presence in `list/1`, but only returns metadata for the presences under a topic and key pair. For example, a user with key `"user1"`, connected to the same chat room `"room:1"` from two devices, could return:

```
iex> MyPresence.get_by_key("room:1", "user1")
[%{name: "User 1", metas: [%{device: "Desktop"}, %
{device: "Mobile"}]}]
```

Like `list/1`, the presence metadata is passed to the `fetch` callback of your presence module to fetch any additional information.

handle_metas(topic, diff, presences, state)

(optional)

```
@callback handle_metas(
  topic :: String.t(),
  diff :: map(),
  presences :: map(),
  state :: term()
) ::
  {:ok, term()}
```

Receives presence metadata changes.

init(state)

(optional)

```
@callback init(state :: term()) :: {:ok, new_state ::
term()}
```

Initializes the presence client state.

Invoked when your presence module starts, allows dynamically providing initial state for handling presence metadata.

list(socket_or_topic)

```
@callback list(socket_or_topic :: Phoenix.Socket.t() |
topic()) :: presences()
```

Returns presences for a socket/topic.

Presence data structure

The presence information is returned as a map with presences grouped by key, cast as a string, and accumulated metadata, with the following form:

```
%{key => %{metas: [%{phx_ref: ..., ...}, ...]}}
```

For example, imagine a user with id `123` online from two different devices, as well as a user with id `456` online from just one device. The following presence information might be returned:

```
%{"123" => %{metas: [%{status: "away", phx_ref: ...},
                     %{status: "online", phx_ref:
...}]},
  "456" => %{metas: [%{status: "online", phx_ref:
...}]}}
```

The keys of the map will usually point to a resource ID. The value will contain a map with a `:metas` key containing a list of metadata for each resource. Additionally, every metadata entry will contain a `:phx_ref` key which can be used to uniquely identify metadata for a given key. In the event that the metadata was previously updated, a `:phx_ref_prev` key will be present containing the previous `:phx_ref` value.

track(socket, key, meta)

```
@callback track(socket :: Phoenix.Socket.t(), key ::
String.t(), meta :: map()) ::
  {:ok, ref :: binary()} | {:error, reason :: term()}
```

Track a channel's process as a presence.

Tracked presences are grouped by `key`, cast as a string. For example, to group each user's channels together, use user IDs as keys. Each presence can be associated with a map of metadata to store small, ephemeral state, such as a user's online status. To store detailed information, see `fetch/2`.

Example

```
alias MyApp.Presence
def handle_info(:after_join, socket) do
  {:ok, _} = Presence.track(socket,
socket.assigns.user_id, %{
    online_at: inspect(System.system_time(:second))
  })
  {:noreply, socket}
end
```

track(pid, topic, key, meta)

```
@callback track(pid(), topic(), key :: String.t(), meta ::
map()) ::
  {:ok, ref :: binary()} | {:error, reason :: term()}
```

Track an arbitrary process as a presence.

Same with `track/3`, except track any process by `topic` and `key`.

untrack(socket, key)

```
@callback untrack(socket :: Phoenix.Socket.t(), key ::
String.t()) :: :ok
```

Stop tracking a channel's process.

untrack(pid, topic, key)

```
@callback untrack(pid(), topic(), key :: String.t()) ::
:ok
```

Stop tracking a process.

update(socket, key, meta)

```
@callback update(
  socket :: Phoenix.Socket.t(),
  key :: String.t(),
  meta :: map() | (map() -> map())
) :: {:ok, ref :: binary()} | {:error, reason :: term()}
```

Update a channel presence's metadata.

Replace a presence's metadata by passing a new map or a function that takes the current map and returns a new one.

update(pid, topic, key, meta)

```
@callback update(pid(), topic(), key :: String.t(), meta
:: map() | (map() -> map())) ::
  {:ok, ref :: binary()} | {:error, reason :: term()}
```

Update a process presence's metadata.

Same as `update/3`, but with an arbitrary process.

Phoenix.Router

Defines a Phoenix 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. For example:

```
defmodule MyAppWeb.Router do
  use Phoenix.Router

  get "/pages/:page", PageController, :show
end
```

The `get/3` macro above accepts a request to `/pages/hello` and dispatches it to `PageController`'s `show` action with `%{"page" => "hello"}` in `params`.

Phoenix's router is extremely efficient, as it relies on Elixir pattern matching for matching routes and serving requests.

Routing

`get/3`, `post/3`, `put/3`, and other macros named after HTTP verbs are used to create routes.

The route:

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

matches a `GET` request to `/pages` and dispatches it to the `index` action in `PageController`.

```
get "/pages/:page", PageController, :show
```

matches `/pages/hello` and dispatches to the `show` action with `%{"page" => "hello"}` in `params`.

```
defmodule PageController do
  def show(conn, params) do
    # %{"page" => "hello"} == params
  end
end
```

Partial and multiple segments can be matched. For example:

```
get "/api/v:version/pages/:id", PageController, :show
```

matches `/api/v1/pages/2` and puts `%{"version" => "1", "id" => "2"}` in `params`. Only the trailing part of a segment can be captured.

Routes are matched from top to bottom. The second route here:

```
get "/pages/:page", PageController, :show
get "/pages/hello", PageController, :hello
```

will never match `/pages/hello` because `/pages/:page` matches that first.

Routes can use glob-like patterns to match trailing segments.

```
get "/pages/*page", PageController, :show
```

matches `/pages/hello/world` and puts the globbed segments in `params["page"]`.

```
GET /pages/hello/world
%{"page" => ["hello", "world"]} = params
```

Globs cannot have prefixes nor suffixes, but can be mixed with variables:

```
get "/pages/he:page/*rest", PageController, :show
```

matches

```
GET /pages/hello
%{"page" => "llo", "rest" => []} = params

GET /pages/hey/there/world
%{"page" => "y", "rest" => ["there" "world"]} = params
```

> **Why the macros?**
>
> Phoenix does its best to keep the usage of macros low. You may have noticed, however, that the `Phoenix.Router` relies heavily on macros. Why is that?
>
> We use `get`, `post`, `put`, and `delete` to define your routes. We use macros for two purposes:
>
> - They define the routing engine, used on every request, to choose which controller to dispatch the request to. Thanks to macros, Phoenix compiles all of your routes to a single case-statement with pattern matching rules, which is heavily optimized by the Erlang VM
> - For each route you define, we also define metadata to implement `Phoenix.VerifiedRoutes`. As we will soon learn, verified routes allows to us to reference any route as if it is a plain looking string, except it is verified by the compiler to be valid (making it much harder to ship broken links, forms, mails, etc to production)
>
> In other words, the router relies on macros to build applications that are faster and safer. Also remember that macros in Elixir are compile-time only, which gives plenty of stability after the code is compiled. Phoenix also provides introspection for all defined routes via `mix phx.routes`.

Generating routes

For generating routes inside your application, see the `Phoenix.VerifiedRoutes` documentation for `~p` based route generation which is the preferred way to generate route paths and URLs with compile-time verification.

Phoenix also supports generating function helpers, which was the default mechanism in Phoenix v1.6 and earlier. We will explore it next.

Helpers (deprecated)

Phoenix generates a module `Helpers` inside your router by default, which contains named helpers to help developers generate and keep their routes up to date. Helpers can be disabled by passing `helpers: false` to `use Phoenix.Router`.

Helpers are automatically generated based on the controller name. For example, the route:

```
get "/pages/:page", PageController, :show
```

will generate the following named helper:

```
MyAppWeb.Router.Helpers.page_path(conn_or_endpoint,
:show, "hello")
"/pages/hello"

MyAppWeb.Router.Helpers.page_path(conn_or_endpoint,
:show, "hello", some: "query")
"/pages/hello?some=query"

MyAppWeb.Router.Helpers.page_url(conn_or_endpoint,
:show, "hello")
"http://example.com/pages/hello"

MyAppWeb.Router.Helpers.page_url(conn_or_endpoint,
```

```
:show, "hello", some: "query")
"http://example.com/pages/hello?some=query"
```

If the route contains glob-like patterns, parameters for those have to be given as list:

```
MyAppWeb.Router.Helpers.page_path(conn_or_endpoint,
:show, ["hello", "world"])
"/pages/hello/world"
```

The URL generated in the named URL helpers is based on the configuration for `:url`, `:http` and `:https`. However, if for some reason you need to manually control the URL generation, the url helpers also allow you to pass in a `URI` struct:

```
uri = %URI{scheme: "https", host: "other.example.com"}
MyAppWeb.Router.Helpers.page_url(uri, :show, "hello")
"https://other.example.com/pages/hello"
```

The named helper can also be customized with the `:as` option. Given the route:

```
get "/pages/:page", PageController, :show, as:
:special_page
```

the named helper will be:

```
MyAppWeb.Router.Helpers.special_page_path(conn, :show,
"hello")
"/pages/hello"
```

Scopes and Resources

It is very common in Phoenix applications to namespace all of your routes under the application scope:

```
scope "/", MyAppWeb do
  get "/pages/:id", PageController, :show
end
```

The route above will dispatch to `MyAppWeb.PageController`. This syntax is not only convenient for developers, since we don't have to repeat the `MyAppWeb.` prefix on all routes, but it also allows Phoenix to put less pressure on the Elixir compiler. If instead we had written:

```
get "/pages/:id", MyAppWeb.PageController, :show
```

The Elixir compiler would infer that the router depends directly on `MyAppWeb.PageController`, which is not true. By using scopes, Phoenix can properly hint to the Elixir compiler the controller is not an actual dependency of the router. This provides more efficient compilation times.

Scopes allow us to scope on any path or even on the helper name:

```
scope "/v1", MyAppWeb, host: "api." do
  get "/pages/:id", PageController, :show
end
```

For example, the route above will match on the path `"/api/v1/pages/1"` and the named route will be `api_v1_page_path`, as expected from the values given to `scope/2` option.

Like all paths you can define dynamic segments that will be applied as parameters in the controller:

```
scope "/api/:version", MyAppWeb do
  get "/pages/:id", PageController, :show
end
```

For example, the route above will match on the path `"/api/v1/pages/1"` and in the controller the `params` argument will have a map with the key `:version` with the value `"v1"`.

Phoenix also provides a `resources/4` macro that allows developers to generate "RESTful" routes to a given resource:

```
defmodule MyAppWeb.Router do
  use Phoenix.Router

  resources "/pages", PageController, only: [:show]
  resources "/users", UserController, except: [:delete]
end
```

Finally, Phoenix ships with a `mix phx.routes` task that nicely formats all routes in a given router. We can use it to verify all routes included in the router above:

```
$ mix phx.routes
page_path  GET    /pages/:id
PageController.show/2
user_path  GET    /users
UserController.index/2
user_path  GET    /users/:id/edit
UserController.edit/2
user_path  GET    /users/new       UserController.new/2
user_path  GET    /users/:id
UserController.show/2
user_path  POST   /users
UserController.create/2
user_path  PATCH  /users/:id
UserController.update/2
           PUT    /users/:id
UserController.update/2
```

One can also pass a router explicitly as an argument to the task:

```
$ mix phx.routes MyAppWeb.Router
```

Check `scope/2` and `resources/4` for more information.

Pipelines and plugs

Once a request arrives at the Phoenix router, it performs a series of transformations through pipelines until the request is dispatched to a desired route.

Such transformations are defined via plugs, as defined in the Plug specification. Once a pipeline is defined, it can be piped through per scope.

For example:

```
defmodule MyAppWeb.Router do
  use Phoenix.Router

  pipeline :browser do
    plug :fetch_session
    plug :accepts, ["html"]
  end

  scope "/" do
    pipe_through :browser

    # browser related routes and resources
  end
end
```

`Phoenix.Router` imports functions from both `Plug.Conn` and `Phoenix.Controller` to help define plugs. In the example above, `fetch_session/2` comes from `Plug.Conn` while `accepts/2` comes from `Phoenix.Controller`.

Note that router pipelines are only invoked after a route is found. No plug is invoked in case no matches were found.

How to organize my routes?

In Phoenix, we tend to define several pipelines, that provide specific functionality. For example, the `pipeline :browser` above includes plugs that are common for all routes that are meant to be accessed by a browser. Similarly, if you are also serving `:api` requests, you would have a separate `:api` pipeline that validates information specific to your endpoints.

Perhaps more importantly, it is also very common to define pipelines specific to authentication and authorization. For example, you might have a pipeline that requires all users are authenticated. Another pipeline may enforce only admin users can access certain routes. Since routes are matched top to bottom, it is recommended to place the authenticated/authorized routes before the less restricted routes to ensure they are matched first.

Once your pipelines are defined, you reuse the pipelines in the desired scopes, grouping your routes around their pipelines. For example, imagine you are building a blog. Anyone can read a post, but only authenticated users can create them. Your routes could look like this:

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

pipeline :auth do
  plug :ensure_authenticated
end

scope "/" do
  pipe_through [:browser, :auth]

  get "/posts/new", PostController, :new
```

```
  post "/posts", PostController, :create
end

scope "/" do
  pipe_through [:browser]

  get "/posts", PostController, :index
  get "/posts/:id", PostController, :show
end
```

Note in the above how the routes are split across different scopes. While the separation can be confusing at first, it has one big upside: it is very easy to inspect your routes and see all routes that, for example, require authentication and which ones do not. This helps with auditing and making sure your routes have the proper scope.

You can create as few or as many scopes as you want. Because pipelines are reusable across scopes, they help encapsulate common functionality and you can compose them as necessary on each scope you define.

Summary

Reflection

route_info(router, method, path, host)

Returns the compile-time route info and runtime path params for a request.

scoped_alias(router_module, alias)

Returns the full alias with the current scope's aliased prefix.

scoped_path(router_module, path)

Returns the full path with the current scope's path prefix.

Functions

connect(path, plug, plug_opts, options \\ [])

Generates a route to handle a connect request to the given path.

delete(path, plug, plug_opts, options \\ [])

Generates a route to handle a delete request to the given path.

forward(path, plug, plug_opts \\ [], router_opts \\ [])

Forwards a request at the given path to a plug.

get(path, plug, plug_opts, options \\ [])

Generates a route to handle a get request to the given path.

head(path, plug, plug_opts, options \\ [])

Generates a route to handle a head request to the given path.

match(verb, path, plug, plug_opts, options \\ [])

Generates a route match based on an arbitrary HTTP method.

options(path, plug, plug_opts, options \\ [])

Generates a route to handle a options request to the given path.

patch(path, plug, plug_opts, options \\ [])

Generates a route to handle a patch request to the given path.

pipe_through(pipes)

Defines a list of plugs (and pipelines) to send the connection through.

pipeline(plug, list)

Defines a plug pipeline.

plug(plug, opts \\ [])

Defines a plug inside a pipeline.

post(path, plug, plug_opts, options \\ [])

Generates a route to handle a post request to the given path.

put(path, plug, plug_opts, options \\ [])

Generates a route to handle a put request to the given path.

resources(path, controller)

See `resources/4`.

resources(path, controller, opts)

See `resources/4`.

resources(path, controller, opts, list)

Defines "RESTful" routes for a resource.

routes(router)

Returns all routes information from the given router.

scope(options, list)

Defines a scope in which routes can be nested.

scope(path, options, list)

Define a scope with the given path.

scope(path, alias, options, list)

Defines a scope with the given path and alias.

trace(path, plug, plug_opts, options \\ [])

Generates a route to handle a trace request to the given path.

Reflection

route_info(router, method, path, host)

Returns the compile-time route info and runtime path params for a request.

The `path` can be either a string or the `path_info` segments.

A map of metadata is returned with the following keys:

- `:log` - the configured log level. For example `:debug`
- `:path_params` - the map of runtime path params
- `:pipe_through` - the list of pipelines for the route's scope, for example `[:browser]`
- `:plug` - the plug to dispatch the route to, for example `AppWeb.PostController`
- `:plug_opts` - the options to pass when calling the plug, for example: `:index`
- `:route` - the string route pattern, such as `"/posts/:id"`

Examples

```
iex> Phoenix.Router.route_info(AppWeb.Router, "GET",
"/posts/123", "myhost")
%{
  log: :debug,
```

```
  path_params: %{"id" => "123"},
  pipe_through: [:browser],
  plug: AppWeb.PostController,
  plug_opts: :show,
  route: "/posts/:id",
}

iex> Phoenix.Router.route_info(MyRouter, "GET", "/not-
exists", "myhost")
:error
```

scoped_alias(router_module, alias)

Returns the full alias with the current scope's aliased prefix.

Useful for applying the same short-hand alias handling to other values besides the second argument in route definitions.

Examples

```
scope "/", MyPrefix do
  get "/", ProxyPlug, controller:
scoped_alias(__MODULE__, MyController)
end
```

scoped_path(router_module, path)

Returns the full path with the current scope's path prefix.

Functions

connect(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a connect request to the given path.

```
connect("/events/:id", EventController, :action)
```

See `match/5` for options.

delete(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a delete request to the given path.

```
delete("/events/:id", EventController, :action)
```

See `match/5` for options.

forward(path, plug, plug_opts \\ [], router_opts \\ [])

(macro)

Forwards a request at the given path to a plug.

All paths that match the forwarded prefix will be sent to the forwarded plug. This is useful for sharing a router between applications or even breaking a big router into smaller ones. The router pipelines will be invoked prior to forwarding the connection.

However, we don't advise forwarding to another endpoint. The reason is that plugs defined by your app and the forwarded endpoint would be invoked twice, which may lead to errors.

Examples

```
scope "/", MyApp do
  pipe_through [:browser, :admin]

  forward "/admin", SomeLib.AdminDashboard
  forward "/api", ApiRouter
end
```

get(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a get request to the given path.

```
get("/events/:id", EventController, :action)
```

See `match/5` for options.

head(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a head request to the given path.

```
head("/events/:id", EventController, :action)
```

See `match/5` for options.

match(verb, path, plug, plug_opts, options \\ [])

(macro)

Generates a route match based on an arbitrary HTTP method.

Useful for defining routes not included in the built-in macros.

The catch-all verb, `:*`, may also be used to match all HTTP methods.

Options

- `:as` - configures the named helper. If `nil`, does not generate a helper. Has no effect when using verified routes exclusively
- `:alias` - configure if the scope alias should be applied to the route. Defaults to true, disables scoping if false.
- `:log` - the level to log the route dispatching under, may be set to false. Defaults to `:debug`. Route dispatching contains information about how the route is handled (which controller action is called, what parameters are available and which pipelines are used) and is separate from the plug level logging. To alter the plug log level, please see https://hexdocs.pm/phoenix/Phoenix.Logger.html#module-dynamic-log-level.
- `:private` - a map of private data to merge into the connection when a route matches

- `:assigns` - a map of data to merge into the connection when a route matches
- `:metadata` - a map of metadata used by the telemetry events and returned by `route_info/4`
- `:warn_on_verify` - the boolean for whether matches to this route trigger an unmatched route warning for `Phoenix.VerifiedRoutes`. It is useful to ignore an otherwise catch-all route definition from being matched when verifying routes. Defaults `false`.

Examples

```
match(:move, "/events/:id", EventController, :move)

match(:*, "/any", SomeController, :any)
```

options(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a options request to the given path.

```
options("/events/:id", EventController, :action)
```

See `match/5` for options.

patch(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a patch request to the given path.

```
patch("/events/:id", EventController, :action)
```

See `match/5` for options.

pipe_through(pipes)

(macro)

Defines a list of plugs (and pipelines) to send the connection through.

Plugs are specified using the atom name of any imported 2-arity function which takes a `%Plug.Conn{}` and options and returns a `%Plug.Conn{}`; for example, `:require_authenticated_user`.

Pipelines are defined in the router; see `pipeline/2` for more information.

```
pipe_through [:my_imported_function, :my_pipeline]
```

pipeline(plug, list)

(macro)

Defines a plug pipeline.

Pipelines are defined at the router root and can be used from any scope.

Examples

```
pipeline :api do
  plug :token_authentication
  plug :dispatch
end
```

A scope may then use this pipeline as:

```
scope "/" do
  pipe_through :api
end
```

Every time `pipe_through/1` is called, the new pipelines are appended to the ones previously given.

plug(plug, opts \\ [])

(macro)

Defines a plug inside a pipeline.

See `pipeline/2` for more information.

post(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a post request to the given path.

```
post("/events/:id", EventController, :action)
```

See match/5 for options.

put(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a put request to the given path.

```
put("/events/:id", EventController, :action)
```

See match/5 for options.

resources(path, controller)

(macro)

See resources/4.

resources(path, controller, opts)

(macro)

See resources/4.

resources(path, controller, opts, list)

(macro)

Defines "RESTful" routes for a resource.

The given definition:

```
resources "/users", UserController
```

will include routes to the following actions:

- `GET /users => :index`
- `GET /users/new => :new`
- `POST /users => :create`
- `GET /users/:id => :show`
- `GET /users/:id/edit => :edit`
- `PATCH /users/:id => :update`
- `PUT /users/:id => :update`
- `DELETE /users/:id => :delete`

Options

This macro accepts a set of options:

- `:only` - a list of actions to generate routes for, for example: `[:show, :edit]`
- `:except` - a list of actions to exclude generated routes from, for example: `[:delete]`
- `:param` - the name of the parameter for this resource, defaults to `"id"`
- `:name` - the prefix for this resource. This is used for the named helper and as the prefix for the parameter in nested resources. The

default value is automatically derived from the controller name, i.e. `UserController` will have name `"user"`
- `:as` - configures the named helper. If `nil`, does not generate a helper. Has no effect when using verified routes exclusively
- `:singleton` - defines routes for a singleton resource that is looked up by the client without referencing an ID. Read below for more information

Singleton resources

When a resource needs to be looked up without referencing an ID, because it contains only a single entry in the given context, the `:singleton` option can be used to generate a set of routes that are specific to such single resource:

- `GET /user => :show`
- `GET /user/new => :new`
- `POST /user => :create`
- `GET /user/edit => :edit`
- `PATCH /user => :update`
- `PUT /user => :update`
- `DELETE /user => :delete`

Usage example:

```
resources "/account", AccountController, only: [:show],
singleton: true
```

Nested Resources

This macro also supports passing a nested block of route definitions. This is helpful for nesting children resources within their parents to generate nested routes.

The given definition:

```
resources "/users", UserController do
  resources "/posts", PostController
end
```

will include the following routes:

```
user_post_path  GET     /users/:user_id/posts
PostController :index
user_post_path  GET     /users/:user_id/posts/:id/edit
PostController :edit
user_post_path  GET     /users/:user_id/posts/new
PostController :new
user_post_path  GET     /users/:user_id/posts/:id
PostController :show
user_post_path  POST    /users/:user_id/posts
PostController :create
user_post_path  PATCH   /users/:user_id/posts/:id
PostController :update
                PUT     /users/:user_id/posts/:id
PostController :update
user_post_path  DELETE  /users/:user_id/posts/:id
PostController :delete
```

routes(router)

Returns all routes information from the given router.

scope(options, list)

(macro)

Defines a scope in which routes can be nested.

Examples

```
scope path: "/api/v1", alias: API.V1 do
  get "/pages/:id", PageController, :show
end
```

The generated route above will match on the path `"/api/v1/pages/:id"` and will dispatch to `:show` action in `API.V1.PageController`. A named helper `api_v1_page_path` will also be generated.

Options

The supported options are:

- `:path` - a string containing the path scope.
- `:as` - a string or atom containing the named helper scope. When set to false, it resets the nested helper scopes. Has no effect when using verified routes exclusively
- `:alias` - an alias (atom) containing the controller scope. When set to false, it resets all nested aliases.
- `:host` - a string or list of strings containing the host scope, or prefix host scope, ie `"foo.bar.com"`, `"foo."`
- `:private` - a map of private data to merge into the connection when a route matches
- `:assigns` - a map of data to merge into the connection when a route matches
- `:log` - the level to log the route dispatching under, may be set to false. Defaults to `:debug`. Route dispatching contains information about how the route is handled (which controller action is called, what parameters are available and which pipelines are used) and is separate from the plug level logging. To alter the plug log level, please see https://hexdocs.pm/phoenix/Phoenix.Logger.html#module-dynamic-log-level.

scope(path, options, list)

(macro)

Define a scope with the given path.

This function is a shortcut for:

```
scope path: path do
  ...
end
```

Examples

```
scope "/v1", host: "api." do
  get "/pages/:id", PageController, :show
end
```

scope(path, alias, options, list)

(macro)

Defines a scope with the given path and alias.

This function is a shortcut for:

```
scope path: path, alias: alias do
  ...
end
```

Examples

```
scope "/v1", API.V1, host: "api." do
  get "/pages/:id", PageController, :show
end
```

trace(path, plug, plug_opts, options \\ [])

(macro)

Generates a route to handle a trace request to the given path.

```
trace("/events/:id", EventController, :action)
```

See match/5 for options.

Phoenix.Socket behaviour

A socket implementation that multiplexes messages over channels.

`Phoenix.Socket` is used as a module for establishing a connection between client and server. Once the connection is established, the initial state is stored in the `Phoenix.Socket` struct.

The same socket can be used to receive events from different transports. Phoenix supports `websocket` and `longpoll` options when invoking `Phoenix.Endpoint.socket/3` in your endpoint. `websocket` is set by default and `longpoll` can also be configured explicitly.

```
socket "/socket", MyAppWeb.Socket, websocket: true,
longpoll: false
```

The command above means incoming socket connections can be made via a WebSocket connection. Incoming and outgoing events are routed to channels by topic:

```
channel "room:lobby", MyAppWeb.LobbyChannel
```

See `Phoenix.Channel` for more information on channels.

Socket Behaviour

Socket handlers are mounted in Endpoints and must define two callbacks:

- `connect/3` - receives the socket params, connection info if any, and authenticates the connection. Must return a `Phoenix.Socket` struct, often with custom assigns

- `id/1` - receives the socket returned by `connect/3` and returns the id of this connection as a string. The `id` is used to identify socket connections, often to a particular user, allowing us to force disconnections. For sockets requiring no authentication, `nil` can be returned

Examples

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

  channel "room:*", MyAppWeb.RoomChannel

  def connect(params, socket, _connect_info) do
    {:ok, assign(socket, :user_id, params["user_id"])}
  end

  def id(socket), do: "users_socket:#
{socket.assigns.user_id}"
end

# Disconnect all user's socket connections and their
multiplexed channels
MyAppWeb.Endpoint.broadcast("users_socket:" <> user.id,
"disconnect", %{})
```

Socket fields

- `:id` - The string id of the socket
- `:assigns` - The map of socket assigns, default: `%{}`
- `:channel` - The current channel module
- `:channel_pid` - The channel pid
- `:endpoint` - The endpoint module where this socket originated, for example: `MyAppWeb.Endpoint`
- `:handler` - The socket module where this socket originated, for example: `MyAppWeb.UserSocket`
- `:joined` - If the socket has effectively joined the channel
- `:join_ref` - The ref sent by the client when joining

- `:ref` - The latest ref sent by the client
- `:pubsub_server` - The registered name of the socket's pubsub server
- `:topic` - The string topic, for example `"room:123"`
- `:transport` - An identifier for the transport, used for logging
- `:transport_pid` - The pid of the socket's transport process
- `:serializer` - The serializer for socket messages

Using options

On `use Phoenix.Socket`, the following options are accepted:

- `:log` - the default level to log socket actions. Defaults to `:info`. May be set to `false` to disable it

- `:partitions` - each channel is spawned under a supervisor. This option controls how many supervisors will be spawned to handle channels. Defaults to the number of cores.

Garbage collection

It's possible to force garbage collection in the transport process after processing large messages. For example, to trigger such from your channels, run:

```
send(socket.transport_pid, :garbage_collect)
```

Alternatively, you can configure your endpoint socket to trigger more fullsweep garbage collections more frequently, by setting the `:fullsweep_after` option for websockets. See `Phoenix.Endpoint.socket/3` for more info.

Client-server communication

The encoding of server data and the decoding of client data is done according to a serializer, defined in `Phoenix.Socket.Serializer`. By

default, JSON encoding is used to broker messages to and from clients.

The serializer `decode!` function must return a `Phoenix.Socket.Message` which is forwarded to channels except:

- `"heartbeat"` events in the "phoenix" topic - should just emit an OK reply
- `"phx_join"` on any topic - should join the topic
- `"phx_leave"` on any topic - should leave the topic

Each message also has a `ref` field which is used to track responses.

The server may send messages or replies back. For messages, the ref uniquely identifies the message. For replies, the ref matches the original message. Both data-types also include a join_ref that uniquely identifies the currently joined channel.

The `Phoenix.Socket` implementation may also send special messages and replies:

- `"phx_error"` - in case of errors, such as a channel process crashing, or when attempting to join an already joined channel

- `"phx_close"` - the channel was gracefully closed

Phoenix ships with a JavaScript implementation of both websocket and long polling that interacts with Phoenix.Socket and can be used as reference for those interested in implementing custom clients.

Custom sockets and transports

See the `Phoenix.Socket.Transport` documentation for more information on writing your own socket that does not leverage channels or for writing your own transports that interacts with other sockets.

Custom channels

You can list any module as a channel as long as it implements a `child_spec/1` function. The `child_spec/1` function receives the caller as argument and it must return a child spec that initializes a process.

Once the process is initialized, it will receive the following message:

```
{Phoenix.Channel, auth_payload, from, socket}
```

A custom channel implementation MUST invoke `GenServer.reply(from, {:ok | :error, reply_payload})` during its initialization with a custom `reply_payload` that will be sent as a reply to the client. Failing to do so will block the socket forever.

A custom channel receives `Phoenix.Socket.Message` structs as regular messages from the transport. Replies to those messages and custom messages can be sent to the socket at any moment by building an appropriate `Phoenix.Socket.Reply` and `Phoenix.Socket.Message` structs, encoding them with the serializer and dispatching the serialized result to the transport.

For example, to handle "phx_leave" messages, which is recommended to be handled by all channel implementations, one may do:

```
def handle_info(
      %Message{topic: topic, event: "phx_leave"} =
message,
      %{topic: topic, serializer: serializer,
transport_pid: transport_pid} = socket
    ) do
  send transport_pid, serializer.encode!
(build_leave_reply(message))
  {:stop, {:shutdown, :left}, socket}
end
```

A special message delivered to all channels is a Broadcast with event "phx_drain", which is sent when draining the socket during application

shutdown. Typically it is handled by sending a drain message to the transport, causing it to shutdown:

```
def handle_info(
      %Broadcast{event: "phx_drain"},
      %{transport_pid: transport_pid} = socket
    ) do
  send(transport_pid, :socket_drain)
  {:stop, {:shutdown, :draining}, socket}
end
```

We also recommend all channels to monitor the `transport_pid` on `init` and exit if the transport exits. We also advise to rewrite `:normal` exit reasons (usually due to the socket being closed) to the `{:shutdown, :closed}` to guarantee links are broken on the channel exit (as a `:normal` exit does not break links):

```
def handle_info({:DOWN, _, _, transport_pid, reason}, %
{transport_pid: transport_pid} = socket) do
  reason = if reason == :normal, do: {:shutdown,
:closed}, else: reason
  {:stop, reason, socket}
end
```

Any process exit is treated as an error by the socket layer unless a `{:socket_close, pid, reason}` message is sent to the socket before shutdown.

Custom channel implementations cannot be tested with `Phoenix.ChannelTest`.

Summary

Types

t()

Callbacks

connect(params, t)

Shortcut version of `connect/3` which does not receive `connect_info`.

connect(params, t, connect_info)

Receives the socket params and authenticates the connection.

id(t)

Identifies the socket connection.

Functions

assign(socket, attrs)
assign(socket, key, value)

Adds key-value pairs to socket assigns.

channel(topic_pattern, module, opts \\ [])

Defines a channel matching the given topic and transports.

Types

t()

```
@type t() :: %Phoenix.Socket{
  assigns: map(),
  channel: atom(),
  channel_pid: pid(),
  endpoint: atom(),
  handler: atom(),
  id: String.t() | nil,
  join_ref: term(),
  joined: boolean(),
  private: map(),
  pubsub_server: atom(),
  ref: term(),
  serializer: atom(),
  topic: String.t(),
  transport: atom(),
  transport_pid: pid()
}
```

Callbacks

connect(params, t)

(optional)

```
@callback connect(params :: map(), t()) :: {:ok, t()} |
{:error, term()} | :error
```

Shortcut version of `connect/3` which does not receive `connect_info`.

Provided for backwards compatibility.

connect(params, t, connect_info)

(optional)

```
@callback connect(params :: map(), t(), connect_info ::
map()) ::
  {:ok, t()} | {:error, term()} | :error
```

Receives the socket params and authenticates the connection.

Socket params and assigns

Socket params are passed from the client and can be used to verify and authenticate a user. After verification, you can put default assigns into the socket that will be set for all channels, ie

```
  {:ok, assign(socket, :user_id, verified_user_id)}
```

To deny connection, return `:error` or `{:error, term}`. To control the response the client receives in that case, define an error handler in the websocket configuration.

See `Phoenix.Token` documentation for examples in performing token verification on connect.

id(t)

```
@callback id(t()) :: String.t() | nil
```

Identifies the socket connection.

Socket IDs are topics that allow you to identify all sockets for a given user:

```
def id(socket), do: "users_socket:#
{socket.assigns.user_id}"
```

Would allow you to broadcast a `"disconnect"` event and terminate all active sockets and channels for a given user:

```
MyAppWeb.Endpoint.broadcast("users_socket:" <> user.id,
"disconnect", %{})
```

Returning `nil` makes this socket anonymous.

Functions

assign(socket, attrs)

assign(socket, key, value)

Adds key-value pairs to socket assigns.

A single key-value pair may be passed, a keyword list or map of assigns may be provided to be merged into existing socket assigns.

Examples

```
iex> assign(socket, :name, "Elixir")
iex> assign(socket, name: "Elixir", logo: "💧")
```

channel(topic_pattern, module, opts \\ [])

(macro)

Defines a channel matching the given topic and transports.

- `topic_pattern` - The string pattern, for example `"room:*"`, `"users:*"`, or `"system"`
- `module` - The channel module handler, for example `MyAppWeb.RoomChannel`
- `opts` - The optional list of options, see below

Options

- `:assigns` - the map of socket assigns to merge into the socket on join

Examples

```
channel "topic1:*", MyChannel
```

Topic Patterns

The `channel` macro accepts topic patterns in two flavors. A splat (the `*` character) argument can be provided as the last character to indicate a `"topic:subtopic"` match. If a plain string is provided, only that topic will match the channel handler. Most use-cases will use the `"topic:*"` pattern to allow more versatile topic scoping.

See `Phoenix.Channel` for more information

Phoenix.Token

Conveniences to sign/encrypt data inside tokens for use in Channels, API authentication, and more.

The data stored in the token is signed to prevent tampering, and is optionally encrypted. This means that, so long as the key (see below) remains secret, you can be assured that the data stored in the token has not been tampered with by a third party. However, unless the token is encrypted, it is not safe to use this token to store private information, such as a user's sensitive identification data, as it can be trivially decoded. If the token is encrypted, its contents will be kept secret from the client, but it is still a best practice to encode as little secret information as possible, to minimize the impact of key leakage.

Example

When generating a unique token for use in an API or Channel it is advised to use a unique identifier for the user, typically the id from a database. For example:

```
iex> user_id = 1
iex> token = Phoenix.Token.sign(MyAppWeb.Endpoint,
"user auth", user_id)
iex> Phoenix.Token.verify(MyAppWeb.Endpoint, "user
auth", token, max_age: 86400)
{:ok, 1}
```

In that example we have a user's id, we generate a token and verify it using the secret key base configured in the given `endpoint`. We guarantee the token will only be valid for one day by setting a max age (recommended).

The first argument to `sign/4`, `verify/4`, `encrypt/4`, and `decrypt/4` can be one of:

- the module name of a Phoenix endpoint (shown above) - where the secret key base is extracted from the endpoint
- `Plug.Conn` - where the secret key base is extracted from the endpoint stored in the connection
- `Phoenix.Socket` or `Phoenix.LiveView.Socket` - where the secret key base is extracted from the endpoint stored in the socket
- a string, representing the secret key base itself. A key base with at least 20 randomly generated characters should be used to provide adequate entropy

The second argument is a cryptographic salt which must be the same in both calls to `sign/4` and `verify/4`, or both calls to `encrypt/4` and `decrypt/4`. For instance, it may be called "user auth" and treated as namespace when generating a token that will be used to authenticate users on channels or on your APIs.

The third argument can be any term (string, int, list, etc.) that you wish to codify into the token. Upon valid verification, this same term will be extracted from the token.

Usage

Once a token is signed, we can send it to the client in multiple ways.

One is via the meta tag:

```
<%= tag :meta, name: "channel_token",
               content: Phoenix.Token.sign(@conn, "user
auth", @current_user.id) %>
```

Or an endpoint that returns it:

```
def create(conn, params) do
  user = User.create(params)
  render(conn, "user.json",
         %{token: Phoenix.Token.sign(conn, "user auth",
user.id), user: user})
end
```

Once the token is sent, the client may now send it back to the server as an authentication mechanism. For example, we can use it to authenticate a user on a Phoenix channel:

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

  def connect(%{"token" => token}, socket,
_connect_info) do
    case Phoenix.Token.verify(socket, "user auth",
token, max_age: 86400) do
      {:ok, user_id} ->
        socket = assign(socket, :user, Repo.get!(User,
user_id))
        {:ok, socket}
      {:error, _} ->
        :error
    end
  end

  def connect(_params, _socket, _connect_info), do:
:error
end
```

In this example, the phoenix.js client will send the token in the `connect` command which is then validated by the server.

`Phoenix.Token` can also be used for validating APIs, handling password resets, e-mail confirmation and more.

Summary

Types

Functions

Decrypts the original data from the token and verifies its integrity.

Encodes, encrypts, and signs data into a token you can send to clients. Its usage is identical to that of `sign/4`, but the data is extracted using `decrypt/4`, rather than `verify/4`.

Encodes and signs data into a token you can send to clients.

Decodes the original data from the token and verifies its integrity.

Types

context()

```
@type context() ::
  Plug.Conn.t()
  | %{:endpoint => atom(), optional(atom()) => any()}
  | atom()
  | binary()
```

max_age_opt()

```
@type max_age_opt() :: {:max_age, pos_integer() |
:infinity}
```

shared_opt()

```
@type shared_opt() ::
  {:key_iterations, pos_integer()}
  | {:key_length, pos_integer()}
  | {:key_digest, :sha256 | :sha384 | :sha512}
```

signed_at_opt()

```
@type signed_at_opt() :: {:signed_at, pos_integer()}
```

Functions

decrypt(context, secret, token, opts \\ [])

```
@spec decrypt(context(), binary(), binary(), [shared_opt()
| max_age_opt()]) :: term()
```

Decrypts the original data from the token and verifies its integrity.

Its usage is identical to `verify/4` but for encrypted tokens.

Options

- `:key_iterations` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to 1000
- `:key_length` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to 32
- `:key_digest` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to `:sha256`
- `:max_age` - verifies the token only if it has been generated "max age" ago in seconds. Defaults to the max age signed in the token by `encrypt/4`.

encrypt(context, secret, data, opts \\ [])

```
@spec encrypt(context(), binary(), term(), [
  shared_opt() | max_age_opt() | signed_at_opt()
]) :: binary()
```

Encodes, encrypts, and signs data into a token you can send to clients. Its usage is identical to that of `sign/4`, but the data is extracted using

decrypt/4, rather than verify/4.

Options

- :key_iterations - option passed to Plug.Crypto.KeyGenerator when generating the encryption and signing keys. Defaults to 1000
- :key_length - option passed to Plug.Crypto.KeyGenerator when generating the encryption and signing keys. Defaults to 32
- :key_digest - option passed to Plug.Crypto.KeyGenerator when generating the encryption and signing keys. Defaults to :sha256
- :signed_at - set the timestamp of the token in seconds. Defaults to System.os_time(:millisecond)
- :max_age - the default maximum age of the token. Defaults to 86400 seconds (1 day) and it may be overridden on decrypt/4.

sign(context, salt, data, opts \\ [])

```
@spec sign(context(), binary(), term(), [
  shared_opt() | max_age_opt() | signed_at_opt()
]) :: binary()
```

Encodes and signs data into a token you can send to clients.

Options

- :key_iterations - option passed to Plug.Crypto.KeyGenerator when generating the encryption and signing keys. Defaults to 1000
- :key_length - option passed to Plug.Crypto.KeyGenerator when generating the encryption and signing keys. Defaults to 32

- `:key_digest` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to `:sha256`
- `:signed_at` - set the timestamp of the token in seconds. Defaults to `System.os_time(:millisecond)`
- `:max_age` - the default maximum age of the token. Defaults to 86400 seconds (1 day) and it may be overridden on `verify/4`.

verify(context, salt, token, opts \\ [])

```
@spec verify(context(), binary(), binary(), [shared_opt()
| max_age_opt()]) ::
  {:ok, term()} | {:error, :expired | :invalid | :missing}
```

Decodes the original data from the token and verifies its integrity.

Examples

In this scenario we will create a token, sign it, then provide it to a client application. The client will then use this token to authenticate requests for resources from the server. See `Phoenix.Token` summary for more info about creating tokens.

```
iex> user_id    = 99
iex> secret     = "kjoy3o1zeidquwy1398juxzldjlksahdk3"
iex> namespace  = "user auth"
iex> token      = Phoenix.Token.sign(secret, namespace,
user_id)
```

The mechanism for passing the token to the client is typically through a cookie, a JSON response body, or HTTP header. For now, assume the

client has received a token it can use to validate requests for protected resources.

When the server receives a request, it can use `verify/4` to determine if it should provide the requested resources to the client:

```
iex> Phoenix.Token.verify(secret, namespace, token,
max_age: 86400)
{:ok, 99}
```

In this example, we know the client sent a valid token because `verify/4` returned a tuple of type `{:ok, user_id}`. The server can now proceed with the request.

However, if the client had sent an expired token, an invalid token, or `nil`, `verify/4` would have returned an error instead:

```
iex> Phoenix.Token.verify(secret, namespace, expired,
max_age: 86400)
{:error, :expired}

iex> Phoenix.Token.verify(secret, namespace, invalid,
max_age: 86400)
{:error, :invalid}

iex> Phoenix.Token.verify(secret, namespace, nil,
max_age: 86400)
{:error, :missing}
```

Options

- `:key_iterations` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to 1000
- `:key_length` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to 32

- `:key_digest` - option passed to `Plug.Crypto.KeyGenerator` when generating the encryption and signing keys. Defaults to `:sha256`
- `:max_age` - verifies the token only if it has been generated "max age" ago in seconds. Defaults to the max age signed in the token by `sign/4`.

Phoenix.VerifiedRoutes

Provides route generation with compile-time verification.

Use of the `sigil_p` macro allows paths and URLs throughout your application to be compile-time verified against your Phoenix router(s). For example, the following path and URL usages:

```
<.link href={~p"/sessions/new"} method="post">Log
in</.link>

redirect(to: url(~p"/posts/#{post}"))
```

Will be verified against your standard `Phoenix.Router` definitions:

```
get "/posts/:post_id", PostController, :show
post "/sessions/new", SessionController, :create
```

Unmatched routes will issue compiler warnings:

```
warning: no route path for AppWeb.Router matches
"/postz/#{post}"
  lib/app_web/controllers/post_controller.ex:100:
AppWeb.PostController.show/2
```

Additionally, interpolated ~p values are encoded via the `Phoenix.Param` protocol. For example, a `%Post{}` struct in your application may derive the `Phoenix.Param` protocol to generate slug-based paths rather than ID based ones. This allows you to use `~p"/posts/#{post}"` rather than `~p"/posts/#{post.slug}"` throughout your application. See the `Phoenix.Param` documentation for more details.

Finally, query strings are also supported in verified routes, either in traditional form:

```
~p"/posts?page=#{page}"
```

Or as a keyword list or map of values:

```
params = %{page: 1, direction: "asc"}
~p"/posts?#{params}"
```

Like path segments, query strings params are proper URL encoded and may be interpolated directly into the ~p string.

What about named routes?

Many web frameworks, and early versions of Phoenix, provided a feature called "named routes". The idea is that, when you define routes in your web applications, you could give them names too. In Phoenix that was done as follows:

```
get "/login", SessionController, :create, as: :login
```

And now you could generate the route usnig the `login_path` function.

Named routes exist to avoid hardcoding routes in your templates, if you wrote `<a href="/login">` and then changed your router, the link would point to a page that no longer exist. By using `login_path`, we make sure it always points to a valid URL in our router. However, named routes come with the downsides of indirection: when you look at the code, it is not immediately clear which URL will be generated. Furthermore, if you have an existing URL and you want to add it to a template, you need to do a reverse lookup and find its name in the router. At the end of the day, named routes are arbitrary names that need to be memorized by developers, adding cognitive overhead.

Verified routes tackle this problem by allowing the routes to be written as we would read them in a browser, but using the `~p` sigil to guarantee they actually exist at compilation time. They remove the indirection of named routes while keeping their guarantees.

In any case, if part of your application requires features similar to named routes, then remember you can still leverage Elixir features to achieve the same result. For example, you can define several functions as named routes to be reused across modules:

```
def login_path, do: ~p"/login"
def user_home_path(user), do: ~p"/users/#
{user.username}"
```

Options

To verify routes in your application modules, such as controller, templates, and views, `use Phoenix.VerifiedRoutes`, which supports the following options:

- `:router` - The required router to verify ~p paths against
- `:endpoint` - The optional endpoint for ~p script_name and URL generation
- `:statics` - The optional list of static directories to treat as verified paths

For example:

```
use Phoenix.VerifiedRoutes,
  router: AppWeb.Router,
  endpoint: AppWeb.Endpoint,
  statics: ~w(images)
```

Usage

The majority of path and URL generation needs your application will be met with `~p` and `url/1`, where all information necessary to construct the path or URL is provided by the compile-time information stored in the Endpoint and Router passed to `use Phoenix.VerifiedRoutes`.

That said, there are some circumstances where `path/2`, `path/3`, `url/2`, and `url/3` are required:

- When the runtime values of the `%Plug.Conn{}`, `%Phoenix.LiveSocket{}`, or a `%URI{}` dictate the formation of the path or URL, which happens under the following scenarios:
    - `Phoenix.Controller.put_router_url/2` is used to override the endpoint's URL
    - `Phoenix.Controller.put_static_url/2` is used to override the endpoint's static URL
- When the Router module differs from the one passed to `use Phoenix.VerifiedRoutes`, such as library code, or application code that relies on multiple routers. In such cases, the router module can be provided explicitly to `path/3` and `url/3`.

Tracking Warnings

All static path segments must start with forward slash, and you must have a static segment between dynamic interpolations in order for a route to be verified without warnings. For example, the following path generates proper warnings

```
~p"/media/posts/#{post}"
```

While this one will not allow the compiler to see the full path:

```
type = "posts"
~p"/media/#{type}/#{post}"
```

In such cases, it's better to write a function such as `media_path/1` which branches on different `~p`'s to handle each type.

Like any other compilation warning, the Elixir compiler will warn any time the file that a ~p resides in changes, or if the router is changed. To view previously issued warnings for files that lack new changes, the `--all-warnings` flag may be passed to the `mix compile` task. For the following will show all warnings the compiler has previously encountered when compiling the current application code:

```
$ mix compile --all-warnings
```

*Note: Elixir >= 1.14.0 is required for comprehensive warnings. Older versions will compile properly, but no warnings will be issued.

Summary

Functions

path(conn_or_socket_or_endpoint_or_uri, sigil_p)

Generates the router path with route verification.

path(conn_or_socket_or_endpoint_or_uri, router, sigil_p)

Generates the router path with route verification.

sigil_p(route, extra)

Generates the router path with route verification.

static_integrity(conn_or_socket_or_endpoint_or_uri, path)

Generates an integrity hash to a static asset given its file path.

static_path(conn_or_socket_or_endpoint_or_uri, path)

Generates path to a static asset given its file path.

static_url(conn_or_socket_or_endpoint, path)

Generates url to a static asset given its file path.

unverified_path(conn_or_socket_or_endpoint_or_uri, router, path, params \\ %{})

Returns the path with relevant script name prefixes without verification.

unverified_url(conn_or_socket_or_endpoint_or_uri, path, params \\ %{})

Returns the URL for the endpoint from the path without verification.

url(sigil_p)

Generates the router url with route verification.

url(conn_or_socket_or_endpoint_or_uri, sigil_p)

Generates the router url with route verification from the connection, socket, or URI.

url(conn_or_socket_or_endpoint_or_uri, router, sigil_p)

Generates the url with route verification from the connection, socket, or URI and router.

Functions

path(conn_or_socket_or_endpoint_or_uri, sigil_p)

(macro)

Generates the router path with route verification.

See `sigil_p/2` for more information.

Warns when the provided path does not match against the router specified in `use Phoenix.VerifiedRoutes` or the `@router` module attribute.

Examples

```
import Phoenix.VerifiedRoutes

redirect(to: path(conn, ~p"/users/top"))

redirect(to: path(conn, ~p"/users/#{@user}"))

~H"""
<.link href={path(@uri, "/users?page=#
{@page}")}>profile</.link>
<.link href={path(@uri, "/users?#
{@params}")}>profile</.link>
"""
```

path(conn_or_socket_or_endpoint_or_uri, router, sigil_p)

(macro)

Generates the router path with route verification.

See `sigil_p/2` for more information.

Warns when the provided path does not match against the router specified in the router argument.

Examples

```
import Phoenix.VerifiedRoutes

redirect(to: path(conn, MyAppWeb.Router,
~p"/users/top"))

redirect(to: path(conn, MyAppWeb.Router, ~p"/users/#
{@user}"))

~H"""
<.link href={path(@uri, MyAppWeb.Router, "/users?page=#
{@page}")}>profile</.link>
<.link href={path(@uri, MyAppWeb.Router, "/users?#
{@params}")}>profile</.link>
"""
```

sigil_p(route, extra)

(macro)

Generates the router path with route verification.

Interpolated named parameters are encoded via the `Phoenix.Param` protocol.

Warns when the provided path does not match against the router specified in `use Phoenix.VerifiedRoutes` or the `@router` module attribute.

Examples

```
use Phoenix.VerifiedRoutes, endpoint:
MyAppWeb.Endpoint, router: MyAppWeb.Router

redirect(to: ~p"/users/top")
```

```
redirect(to: ~p"/users/#{@user}")

~H"""
<.link href={~p"/users?page=#{@page}"}>profile</.link>

<.link href={~p"/users?#{@params}"}>profile</.link>
"""
```

static_integrity(conn_or_socket_or_endpoint_or_uri, path)

Generates an integrity hash to a static asset given its file path.

See `Phoenix.Endpoint.static_integrity/1` for more information.

Examples

```
iex> static_integrity(conn, "/assets/app.js")
"813dfe33b5c7f8388bccaaa38eec8382"

iex> static_integrity(socket, "/assets/app.js")
"813dfe33b5c7f8388bccaaa38eec8382"

iex> static_integrity(AppWeb.Endpoint,
"/assets/app.js")
"813dfe33b5c7f8388bccaaa38eec8382"
```

static_path(conn_or_socket_or_endpoint_or_uri, path)

Generates path to a static asset given its file path.

See `Phoenix.Endpoint.static_path/1` for more information.

Examples

```
iex> static_path(conn, "/assets/app.js")
"/assets/app-813dfe33b5c7f8388bccaaa38eec8382.js"

iex> static_path(socket, "/assets/app.js")
"/assets/app-813dfe33b5c7f8388bccaaa38eec8382.js"

iex> static_path(AppWeb.Endpoint, "/assets/app.js")
"/assets/app-813dfe33b5c7f8388bccaaa38eec8382.js"

iex> static_path(%URI{path: "/subresource"},
"/assets/app.js")
"/subresource/assets/app-
813dfe33b5c7f8388bccaaa38eec8382.js"
```

static_url(conn_or_socket_or_endpoint, path)

Generates url to a static asset given its file path.

See `Phoenix.Endpoint.static_url/0` and `Phoenix.Endpoint.static_path/1` for more information.

Examples

```
iex> static_url(conn, "/assets/app.js")
"https://example.com/assets/app-
813dfe33b5c7f8388bccaaa38eec8382.js"

iex> static_url(socket, "/assets/app.js")
"https://example.com/assets/app-
813dfe33b5c7f8388bccaaa38eec8382.js"

iex> static_url(AppWeb.Endpoint, "/assets/app.js")
"https://example.com/assets/app-
813dfe33b5c7f8388bccaaa38eec8382.js"
```

unverified_path(conn_or_socket_or_endpoint_or_uri, router, path, params \\ %{})

Returns the path with relevant script name prefixes without verification.

Examples

```
iex> unverified_path(conn, AppWeb.Router, "/posts")
"/posts"

iex> unverified_path(conn, AppWeb.Router, "/posts",
page: 1)
"/posts?page=1"
```

unverified_url(conn_or_socket_or_endpoint_or_uri, path, params \\ %{})

Returns the URL for the endpoint from the path without verification.

Examples

```
iex> unverified_url(conn, "/posts")
"https://example.com/posts"

iex> unverified_url(conn, "/posts", page: 1)
"https://example.com/posts?page=1"
```

url(sigil_p)

(macro)

Generates the router url with route verification.

See `sigil_p/2` for more information.

Warns when the provided path does not match against the router specified in `use Phoenix.VerifiedRoutes` or the `@router` module attribute.

Examples

```
use Phoenix.VerifiedRoutes, endpoint:
MyAppWeb.Endpoint, router: MyAppWeb.Router

redirect(to: url(conn, ~p"/users/top"))

redirect(to: url(conn, ~p"/users/#{@user}"))

~H"""
<.link href={url(@uri, "/users?#{[page:
@page]}")}>profile</.link>
"""
```

The router may also be provided in cases where you want to verify routes for a router other than the one passed to `use Phoenix.VerifiedRoutes`:

```
redirect(to: url(conn, OtherRouter, ~p"/users"))
```

Forwarded routes are also resolved automatically. For example, imagine you have a forward path to an admin router in your main router:

```
defmodule AppWeb.Router do
  ...
  forward "/admin", AppWeb.AdminRouter
end

defmodule AppWeb.AdminRouter do
  ...
  get "/users", AppWeb.Admin.UserController
end
```

Forwarded paths in your main application router will be verified as usual, such as `~p"/admin/users"`.

url(conn_or_socket_or_endpoint_or_uri, sigil_p)

(macro)

Generates the router url with route verification from the connection, socket, or URI.

See `url/1` for more information.

url(conn_or_socket_or_endpoint_or_uri, router, sigil_p)

(macro)

Generates the url with route verification from the connection, socket, or URI and router.

See `url/1` for more information.

Phoenix.ChannelTest

Conveniences for testing Phoenix channels.

In channel tests, we interact with channels via process communication, sending and receiving messages. It is also common to subscribe to the same topic the channel subscribes to, allowing us to assert if a given message was broadcast or not.

Channel testing

To get started, define the module attribute `@endpoint` in your test case pointing to your application endpoint.

Then you can directly create a socket and `subscribe_and_join/4` topics and channels:

```
{:ok, _, socket} =
  socket(UserSocket, "user:id", %{some_assigns: 1})
  |> subscribe_and_join(RoomChannel, "room:lobby", %
{"id" => 3})
```

You usually want to set the same ID and assigns your `UserSocket.connect/3` callback would set. Alternatively, you can use the `connect/3` helper to call your `UserSocket.connect/3` callback and initialize the socket with the socket id:

```
{:ok, socket} = connect(UserSocket, %{"some" =>
"params"}, %{})
{:ok, _, socket} = subscribe_and_join(socket,
"room:lobby", %{"id" => 3})
```

Once called, `subscribe_and_join/4` will subscribe the current test process to the "room:lobby" topic and start a channel in another process. It returns `{:ok, reply, socket}` or `{:error, reply}`.

Now, in the same way the channel has a socket representing communication it will push to the client. Our test has a socket representing communication to be pushed to the server.

For example, we can use the `push/3` function in the test to push messages to the channel (it will invoke `handle_in/3`):

```
push(socket, "my_event", %{"some" => "data"})
```

Similarly, we can broadcast messages from the test itself on the topic that both test and channel are subscribed to, triggering `handle_out/3` on the channel:

```
broadcast_from(socket, "my_event", %{"some" => "data"})
```

> Note only `broadcast_from/3` and `broadcast_from!/3` are available in tests to avoid broadcast messages to be resent to the test process.

While the functions above are pushing data to the channel (server) we can use `assert_push/3` to verify the channel pushed a message to the client:

```
assert_push "my_event", %{"some" => "data"}
```

Or even assert something was broadcast into pubsub:

```
assert_broadcast "my_event", %{"some" => "data"}
```

Finally, every time a message is pushed to the channel, a reference is returned. We can use this reference to assert a particular reply was sent from the server:

```
ref = push(socket, "counter", %{})
assert_reply ref, :ok, %{"counter" => 1}
```

Checking side-effects

Often one may want to do side-effects inside channels, like writing to the database, and verify those side-effects during their tests.

Imagine the following `handle_in/3` inside a channel:

```
def handle_in("publish", %{"id" => id}, socket) do
  Repo.get!(Post, id) |> Post.publish() |> Repo.update!
()
  {:noreply, socket}
end
```

Because the whole communication is asynchronous, the following test would be very brittle:

```
push(socket, "publish", %{"id" => 3})
assert Repo.get_by(Post, id: 3, published: true)
```

The issue is that we have no guarantees the channel has done processing our message after calling `push/3`. The best solution is to assert the channel sent us a reply before doing any other assertion. First change the channel to send replies:

```
def handle_in("publish", %{"id" => id}, socket) do
  Repo.get!(Post, id) |> Post.publish() |> Repo.update!
()
```

```
    {:reply, :ok, socket}
  end
```

Then expect them in the test:

```
  ref = push(socket, "publish", %{"id" => 3})
  assert_reply ref, :ok
  assert Repo.get_by(Post, id: 3, published: true)
```

Leave and close

This module also provides functions to simulate leaving and closing a channel. Once you leave or close a channel, because the channel is linked to the test process on join, it will crash the test process:

```
  leave(socket)
  ** (EXIT from #PID<...>) {:shutdown, :leave}
```

You can avoid this by unlinking the channel process in the test:

```
  Process.unlink(socket.channel_pid)
```

Notice `leave/1` is async, so it will also return a reference which you can use to check for a reply:

```
  ref = leave(socket)
  assert_reply ref, :ok
```

On the other hand, close is always sync and it will return only after the channel process is guaranteed to have been terminated:

```
  :ok = close(socket)
```

This mimics the behaviour existing in clients.

To assert that your channel closes or errors asynchronously, you can monitor the channel process with the tools provided by Elixir, and wait for the `:DOWN` message. Imagine an implementation of the `handle_info/2` function that closes the channel when it receives `:some_message`:

```
def handle_info(:some_message, socket) do
  {:stop, :normal, socket}
end
```

In your test, you can assert that the close happened by:

```
Process.monitor(socket.channel_pid)
send(socket.channel_pid, :some_message)
assert_receive {:DOWN, _, _, _, :normal}
```

Summary

Functions

assert_broadcast(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

Asserts the channel has broadcast a message within `timeout`.

assert_push(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

Asserts the channel has pushed a message back to the client with the given event and payload within `timeout`.

assert_reply(ref, status, payload \\ Macro.escape(%{}), timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

Asserts the channel has replied to the given message within `timeout`.

broadcast_from(socket, event, message)

Broadcast event from pid to all subscribers of the socket topic.

broadcast_from!(socket, event, message)

Same as `broadcast_from/3`, but raises if broadcast fails.

close(socket, timeout \\ 5000)

Emulates the client closing the socket.

connect(handler, params, options \\ quote do [] end)

Initiates a transport connection for the socket handler.

join(socket, topic)

See `join/4`.

join(socket, topic, payload)

See `join/4`.

join(socket, channel, topic, payload \\ %{})

Joins the channel under the given topic and payload.

leave(socket)

Emulates the client leaving the channel.

push(socket, event, payload \\ %{})

Pushes a message into the channel.

refute_broadcast(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

Asserts the channel has not broadcast a message within `timeout`.

refute_push(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

Asserts the channel has not pushed a message to the client matching the given event and payload within `timeout`.

refute_reply(ref, status, payload \\ Macro.escape(%{}), timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

Asserts the channel has not replied with a matching payload within `timeout`.

socket(socket_module)

Builds a socket for the given `socket_module`.

socket(socket_module, socket_id, socket_assigns, options \\ [])

Builds a socket for the given `socket_module` with given id and assigns.

subscribe_and_join(socket, topic)

See `subscribe_and_join/4`.

subscribe_and_join(socket, topic, payload)

See `subscribe_and_join/4`.

subscribe_and_join(socket, channel, topic, payload \\ %{})

Subscribes to the given topic and joins the channel under the given topic and payload.

subscribe_and_join!(socket, topic)

See `subscribe_and_join!/4`.

subscribe_and_join!(socket, topic, payload)

See `subscribe_and_join!/4`.

subscribe_and_join!(socket, channel, topic, payload \\ %{})

Same as `subscribe_and_join/4`, but returns either the socket or throws an error.

Functions

assert_broadcast(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

(macro)

Asserts the channel has broadcast a message within `timeout`.

Before asserting anything was broadcast, we must first subscribe to the topic of the channel in the test process:

```
@endpoint.subscribe("foo:ok")
```

Now we can match on event and payload as patterns:

```
assert_broadcast "some_event", %{"data" => _}
```

In the assertion above, we don't particularly care about the data being sent, as long as something was sent.

The timeout is in milliseconds and defaults to the `:assert_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms).

assert_push(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

(macro)

Asserts the channel has pushed a message back to the client with the given event and payload within `timeout`.

Notice event and payload are patterns. This means one can write:

```
assert_push "some_event", %{"data" => _}
```

In the assertion above, we don't particularly care about the data being sent, as long as something was sent.

The timeout is in milliseconds and defaults to the `:assert_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms).

NOTE: Because event and payload are patterns, they will be matched. This means that if you wish to assert that the received payload is equivalent to an existing variable, you need to pin the variable in the assertion expression.

Good:

```
expected_payload = %{foo: "bar"}
assert_push "some_event", ^expected_payload
```

Bad:

```
expected_payload = %{foo: "bar"}
assert_push "some_event", expected_payload
# The code above does not assert the payload matches
the described map.
```

assert_reply(ref, status, payload \\ Macro.escape(%{}), timeout \\ Application.fetch_env!(:ex_unit, :assert_receive_timeout))

(macro)

Asserts the channel has replied to the given message within `timeout`.

Notice status and payload are patterns. This means one can write:

```
ref = push(channel, "some_event")
assert_reply ref, :ok, %{"data" => _}
```

In the assertion above, we don't particularly care about the data being sent, as long as something was replied.

The timeout is in milliseconds and defaults to the `:assert_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms).

broadcast_from(socket, event, message)

Broadcast event from pid to all subscribers of the socket topic.

The test process will not receive the published message. This triggers the `handle_out/3` callback in the channel.

Examples

```
iex> broadcast_from(socket, "new_message", %{id: 1,
content: "hello"})
:ok
```

broadcast_from!(socket, event, message)

Same as `broadcast_from/3`, but raises if broadcast fails.

close(socket, timeout \\ 5000)

Emulates the client closing the socket.

Closing socket is synchronous and has a default timeout of 5000 milliseconds.

connect(handler, params, options \\ quote do [] end)

(macro)

Initiates a transport connection for the socket handler.

Useful for testing UserSocket authentication. Returns the result of the handler's `connect/3` callback.

join(socket, topic)

See `join/4`.

join(socket, topic, payload)

See `join/4`.

join(socket, channel, topic, payload \\ %{})

Joins the channel under the given topic and payload.

The given channel is joined in a separate process which is linked to the test process.

It returns `{:ok, reply, socket}` or `{:error, reply}`.

leave(socket)

```
@spec leave(Phoenix.Socket.t()) :: reference()
```

Emulates the client leaving the channel.

push(socket, event, payload \\ %{})

```
@spec push(Phoenix.Socket.t(), String.t(), map()) ::
reference()
```

Pushes a message into the channel.

The triggers the `handle_in/3` callback in the channel.

Examples

```
iex> push(socket, "new_message", %{id: 1, content:
"hello"})
reference
```

refute_broadcast(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

(macro)

Asserts the channel has not broadcast a message within `timeout`.

Like `assert_broadcast`, the event and payload are patterns.

The timeout is in milliseconds and defaults to the `:refute_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms). Keep in mind this macro will block the test by the timeout value, so use it only when necessary as overuse will certainly slow down your test suite.

refute_push(event, payload, timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

(macro)

Asserts the channel has not pushed a message to the client matching the given event and payload within `timeout`.

Like `assert_push`, the event and payload are patterns.

The timeout is in milliseconds and defaults to the `:refute_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms). Keep in mind this macro will block the test by the timeout value, so use it only when necessary as overuse will certainly slow down your test suite.

refute_reply(ref, status, payload \\ Macro.escape(%{}), timeout \\ Application.fetch_env!(:ex_unit, :refute_receive_timeout))

(macro)

Asserts the channel has not replied with a matching payload within `timeout`.

Like `assert_reply`, the event and payload are patterns.

The timeout is in milliseconds and defaults to the `:refute_receive_timeout` set on the `:ex_unit` application (which defaults to 100ms). Keep in mind this macro will block the test by the timeout value, so use it only when necessary as overuse will certainly slow down your test suite.

socket(socket_module)

(macro)

Builds a socket for the given `socket_module`.

The socket is then used to subscribe and join channels. Use this function when you want to create a blank socket to pass to functions like `UserSocket.connect/3`.

Otherwise, use `socket/4` if you want to build a socket with existing id and assigns.

Examples

```
socket(MyApp.UserSocket)
```

socket(socket_module, socket_id, socket_assigns, options \\ [])

(macro)

Builds a socket for the given `socket_module` with given id and assigns.

Examples

```
socket(MyApp.UserSocket, "user_id", %{some: :assign})
```

If you need to access the socket in another process than the test process, you can give the `pid` of the test process in the 4th argument.

Examples

```
test "connect in a task" do
  pid = self()
  task = Task.async(fn ->
    socket = socket(MyApp.UserSocket, "user_id", %
{some: :assign}, test_process: pid)
    broadcast_from!(socket, "default", %{"foo" =>
"bar"})
    assert_push "default", %{"foo" => "bar"}
  end)
  Task.await(task)
end
```

subscribe_and_join(socket, topic)

See subscribe_and_join/4.

subscribe_and_join(socket, topic, payload)

See subscribe_and_join/4.

subscribe_and_join(socket, channel, topic, payload \\ %{})

Subscribes to the given topic and joins the channel under the given topic and payload.

By subscribing to the topic, we can use `assert_broadcast/3` to verify a message has been sent through the pubsub layer.

By joining the channel, we can interact with it directly. The given channel is joined in a separate process which is linked to the test process.

If no channel module is provided, the socket's handler is used to lookup the matching channel for the given topic.

It returns `{:ok, reply, socket}` or `{:error, reply}`.

subscribe_and_join!(socket, topic)

See `subscribe_and_join!/4`.

subscribe_and_join!(socket, topic, payload)

See `subscribe_and_join!/4`.

subscribe_and_join!(socket, channel, topic, payload \\ %{})

Same as `subscribe_and_join/4`, but returns either the socket or throws an error.

This is helpful when you are not testing joining the channel and just need the socket.

Phoenix.ConnTest

Conveniences for testing Phoenix endpoints and connection related helpers.

You likely want to use this module or make it part of your `ExUnit.CaseTemplate`. Once used, this module automatically imports all functions defined here as well as the functions in `Plug.Conn`.

Endpoint testing

`Phoenix.ConnTest` typically works against endpoints. That's the preferred way to test anything that your router dispatches to:

```
@endpoint MyAppWeb.Endpoint

test "says welcome on the home page" do
  conn = get(build_conn(), "/")
  assert conn.resp_body =~ "Welcome!"
end

test "logs in" do
  conn = post(build_conn(), "/login", [username:
"john", password: "doe"])
  assert conn.resp_body =~ "Logged in!"
end
```

The `@endpoint` module attribute contains the endpoint under testing, most commonly your application endpoint itself. If you are using the MyApp.ConnCase generated by Phoenix, it is automatically set for you.

As in your router and controllers, the connection is the main abstraction in testing. `build_conn()` returns a new connection and functions in this module can be used to manipulate the connection before dispatching to the endpoint.

For example, one could set the accepts header for json requests as follows:

```
build_conn()
|> put_req_header("accept", "application/json")
|> get("/")
```

You can also create your own helpers, such as `json_conn()` that uses `build_conn/0` and `put_req_header/3`, so you avoid repeating the connection setup throughout your tests.

Controller testing

The functions in this module can also be used for controller testing. While endpoint testing is preferred over controller testing, especially since the controller in Phoenix plays an integration role between your domain and your views, unit testing controllers may be helpful in some situations.

For such cases, you need to set the `@endpoint` attribute to your controller and pass an atom representing the action to dispatch:

```
@endpoint MyAppWeb.HomeController

test "says welcome on the home page" do
  conn = get(build_conn(), :index)
  assert conn.resp_body =~ "Welcome!"
end
```

Keep in mind that, once the `@endpoint` variable is set, all tests after setting it will be affected.

Views testing

Under other circumstances, you may be testing a view or another layer that requires a connection for processing. For such cases, a connection can be created using the `build_conn/3` helper:

```
MyApp.UserView.render("hello.html", conn:
build_conn(:get, "/"))
```

While `build_conn/0` returns a connection with no request information to it, `build_conn/3` returns a connection with the given request information already filled in.

Recycling

Browsers implement a storage by using cookies. When a cookie is set in the response, the browser stores it and sends it in the next request.

To emulate this behaviour, this module provides the idea of recycling. The `recycle/1` function receives a connection and returns a new connection, similar to the one returned by `build_conn/0` with all the response cookies from the previous connection defined as request headers. This is useful when testing multiple routes that require cookies or session to work.

Keep in mind Phoenix will automatically recycle the connection between dispatches. This usually works out well most times, but it may discard information if you are modifying the connection before the next dispatch:

```
# No recycling as the connection is fresh
conn = get(build_conn(), "/")

# The connection is recycled, creating a new one behind
the scenes
conn = post(conn, "/login")

# We can also recycle manually in case we want custom
headers
conn =
  conn
  |> recycle()
  |> put_req_header("x-special", "nice")
```

```
# No recycling as we did it explicitly
conn = delete(conn, "/logout")
```

Recycling also recycles the "accept" and "authorization" headers, as well as peer data information.

Summary

Functions

assert_error_sent(status_int_or_atom, func)

Asserts an error was wrapped and sent with the given status.

build_conn()

Creates a connection to be used in upcoming requests.

build_conn(method, path, params_or_body \\ nil)

Creates a connection to be used in upcoming requests with a preset method, path and body.

bypass_through(conn)

Calls the Endpoint and Router pipelines.

bypass_through(conn, router)

Calls the Endpoint and Router pipelines for the current route.

bypass_through(conn, router, pipelines)

Calls the Endpoint and the given Router pipelines.

clear_flash(conn)

Clears up the flash storage.

connect(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

delete(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

delete_req_cookie(conn, key)

Deletes a request cookie.

dispatch(conn, endpoint, method, path_or_action, params_or_body \\ nil)

Dispatches the connection to the given endpoint.

ensure_recycled(conn)

Ensures the connection is recycled if it wasn't already.

fetch_flash(conn)

Fetches the flash storage.

get(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

get_flash(conn) deprecated

Gets the whole flash storage.

get_flash(conn, key) deprecated

Gets the given key from the flash storage.

head(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

html_response(conn, status)

Asserts the given status code, that we have an html response and returns the response body if one was set or sent.

init_test_session(conn, session)

Inits a session used exclusively for testing.

json_response(conn, status)

Asserts the given status code, that we have a json response and returns the decoded JSON response if one was set or sent.

options(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

patch(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

path_params(conn, to)

Returns the matched params of the URL for the `%Plug.Conn{}`'s router.

post(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

put(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

put_flash(conn, key, value)

Puts the given value under key in the flash storage.

put_req_cookie(conn, key, value)

Puts a request cookie.

recycle(conn, headers \\ ~w(accept accept-language authorization))

Recycles the connection.

redirected_params(conn, status \\ 302)

Returns the matched params from the URL the connection was redirected to.

redirected_to(conn, status \\ 302)

Returns the location header from the given redirect response.

response(conn, given)

Asserts the given status code and returns the response body if one was set or sent.

response_content_type(conn, format)

Returns the content type as long as it matches the given format.

text_response(conn, status)

Asserts the given status code, that we have a text response and returns the response body if one was set or sent.

trace(conn, path_or_action, params_or_body \\ nil)

Dispatches to the current endpoint.

Functions

assert_error_sent(status_int_or_atom, func)

```
@spec assert_error_sent(integer() | atom(), function()) ::
{integer(), list(), term()}
```

Asserts an error was wrapped and sent with the given status.

Useful for testing actions that you expect raise an error and have the response wrapped in an HTTP status, with content usually rendered by your MyAppWeb.ErrorHTML view.

The function accepts a status either as an integer HTTP status or atom, such as `500` or `:internal_server_error`. The list of allowed atoms is available in `Plug.Conn.Status`. If an error is raised, a 3-tuple of the wrapped response is returned matching the status, headers, and body of the response:

```
{500, [{"content-type", "text/html"} | _], "Internal
Server Error"}
```

Examples

```
assert_error_sent :internal_server_error, fn ->
  get(build_conn(), "/broken/route")
end

response = assert_error_sent 500, fn ->
  get(build_conn(), "/broken/route")
end
assert {500, [_h | _t], "Internal Server Error"} =
response
```

This can also be used to test a route resulted in an error that was translated to a specific response by the `Plug.Status` protocol, such as

Ecto.NoResultsError:

```
assert_error_sent :not_found, fn ->
  get(build_conn(), "/something-that-raises-no-results-
error")
end
```

Note: for routes that don't raise an error, but instead return a status, you should test the response directly:

```
conn = get(build_conn(), "/users/not-found")
assert response(conn, 404)
```

build_conn()

```
@spec build_conn() :: Plug.Conn.t()
```

Creates a connection to be used in upcoming requests.

build_conn(method, path, params_or_body \\ nil)

```
@spec build_conn(atom() | binary(), binary(), binary() |
list() | map() | nil) ::
  Plug.Conn.t()
```

Creates a connection to be used in upcoming requests with a preset method, path and body.

This is useful when a specific connection is required for testing a plug or a particular function.

bypass_through(conn)

```
@spec bypass_through(Plug.Conn.t()) :: Plug.Conn.t()
```

Calls the Endpoint and Router pipelines.

Useful for unit testing Plugs where Endpoint and/or router pipeline plugs are required for proper setup.

Note the use of `get("/")` following `bypass_through` in the examples below. To execute the plug pipelines, you must issue a request against the router. Most often, you can simply send a GET request against the root path, but you may also specify a different method or path which your pipelines may operate against.

Examples

For example, imagine you are testing an authentication plug in isolation, but you need to invoke the Endpoint plugs and router pipelines to set up session and flash related dependencies. One option is to invoke an existing route that uses the proper pipelines. You can do so by passing the connection and the router name to `bypass_through`:

```
conn =
  conn
  |> bypass_through(MyAppWeb.Router)
  |> get("/some_url")
  |> MyApp.RequireAuthentication.call([])
assert conn.halted
```

You can also specify which pipelines you want to run:

```
conn =
  conn
  |> bypass_through(MyAppWeb.Router, [:browser])
  |> get("/")
  |> MyApp.RequireAuthentication.call([])
assert conn.halted
```

Alternatively, you could only invoke the Endpoint's plugs:

```
conn =
  conn
  |> bypass_through()
  |> get("/")
  |> MyApp.RequireAuthentication.call([])

assert conn.halted
```

bypass_through(conn, router)

```
@spec bypass_through(Plug.Conn.t(), module()) ::
Plug.Conn.t()
```

Calls the Endpoint and Router pipelines for the current route.

See bypass_through/1.

bypass_through(conn, router, pipelines)

```
@spec bypass_through(Plug.Conn.t(), module(), atom() | list()) :: Plug.Conn.t()
```

Calls the Endpoint and the given Router pipelines.

See `bypass_through/1`.

clear_flash(conn)

```
@spec clear_flash(Plug.Conn.t()) :: Plug.Conn.t()
```

Clears up the flash storage.

connect(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

delete(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

delete_req_cookie(conn, key)

```
@spec delete_req_cookie(Plug.Conn.t(), binary()) ::
Plug.Conn.t()
```

Deletes a request cookie.

dispatch(conn, endpoint, method, path_or_action, params_or_body \\ nil)

Dispatches the connection to the given endpoint.

When invoked via `get/3`, `post/3` and friends, the endpoint is automatically retrieved from the `@endpoint` module attribute, otherwise it must be given as an argument.

The connection will be configured with the given `method`, `path_or_action` and `params_or_body`.

If `path_or_action` is a string, it is considered to be the request path and stored as so in the connection. If an atom, it is assumed to be an action and the connection is dispatched to the given action.

Parameters and body

This function, as well as `get/3`, `post/3` and friends, accepts the request body or parameters as last argument:

```
get(build_conn(), "/", some: "param")
get(build_conn(), "/", "some=param&url=encoded")
```

The allowed values are:

- `nil` - meaning there is no body
- a binary - containing a request body. For such cases, `:headers` must be given as option with a content-type
- a map or list - containing the parameters which will automatically set the content-type to multipart. The map or list may contain other lists or maps and all entries will be normalized to string keys
- a struct - unlike other maps, a struct will be passed through as-is without normalizing its entries

ensure_recycled(conn)

```
@spec ensure_recycled(Plug.Conn.t()) :: Plug.Conn.t()
```

Ensures the connection is recycled if it wasn't already.

See `recycle/1` for more information.

fetch_flash(conn)

```
@spec fetch_flash(Plug.Conn.t()) :: Plug.Conn.t()
```

Fetches the flash storage.

get(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

get_flash(conn)

This function is deprecated. get_flash/1 is deprecated. Use conn.assigns.flash instead.

```
@spec get_flash(Plug.Conn.t()) :: map()
```

Gets the whole flash storage.

get_flash(conn, key)

This function is deprecated. get_flash/2 is deprecated. Use Phoenix.Flash.get/2 instead.

```
@spec get_flash(Plug.Conn.t(), term()) :: term()
```

Gets the given key from the flash storage.

head(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

html_response(conn, status)

```
@spec html_response(Plug.Conn.t(), status :: integer() | atom()) :: String.t()
```

Asserts the given status code, that we have an html response and returns the response body if one was set or sent.

Examples

```
assert html_response(conn, 200) =~ "<html>"
```

init_test_session(conn, session)

```
@spec init_test_session(Plug.Conn.t(), map() | keyword()) :: Plug.Conn.t()
```

Inits a session used exclusively for testing.

json_response(conn, status)

```
@spec json_response(Plug.Conn.t(), status :: integer() |
atom()) :: term()
```

Asserts the given status code, that we have a json response and returns the decoded JSON response if one was set or sent.

Examples

```
body = json_response(conn, 200)
assert "can't be blank" in body["errors"]
```

options(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

patch(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

path_params(conn, to)

```
@spec path_params(Plug.Conn.t(), String.t()) :: map()
```

Returns the matched params of the URL for the `%Plug.Conn{}`'s router.

Useful for extracting path params out of returned URLs, such as those returned by `Phoenix.LiveViewTest`'s redirected results.

Examples

```
assert {:error, {:redirect, %{to: "/posts/123" = to}}}
= live(conn, "/path")
assert %{id: "123"} = path_params(conn, to)
```

post(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

put(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See dispatch/5 for more information.

put_flash(conn, key, value)

```
@spec put_flash(Plug.Conn.t(), term(), term()) ::
Plug.Conn.t()
```

Puts the given value under key in the flash storage.

put_req_cookie(conn, key, value)

```
@spec put_req_cookie(Plug.Conn.t(), binary(), binary()) ::
Plug.Conn.t()
```

Puts a request cookie.

recycle(conn, headers \\ ~w(accept accept-language authorization))

```
@spec recycle(Plug.Conn.t(), [String.t()]) ::
Plug.Conn.t()
```

Recycles the connection.

Recycling receives a connection and returns a new connection, containing cookies and relevant information from the given one.

This emulates behaviour performed by browsers where cookies returned in the response are available in following requests.

By default, only the headers "accept", "accept-language", and "authorization" are recycled. However, a custom set of headers can be specified by passing a list of strings representing its names as the second argument of the function.

Note `recycle/1` is automatically invoked when dispatching to the endpoint, unless the connection has already been recycled.

redirected_params(conn, status \\ 302)

```
@spec redirected_params(Plug.Conn.t(), status ::
non_neg_integer()) :: map()
```

Returns the matched params from the URL the connection was redirected to.

Uses the provided `%Plug.Conn{}`s router matched in the previous request. Raises if the response's location header is not set or if the response does not match the redirect status code (defaults to 302).

Examples

```
assert redirected_to(conn) =~ "/posts/123"
assert %{id: "123"} = redirected_params(conn)
assert %{id: "123"} = redirected_params(conn, 303)
```

redirected_to(conn, status \\ 302)

```
@spec redirected_to(Plug.Conn.t(), status ::
non_neg_integer()) :: String.t()
```

Returns the location header from the given redirect response.

Raises if the response does not match the redirect status code (defaults to 302).

Examples

```
assert redirected_to(conn) =~ "/foo/bar"
assert redirected_to(conn, 301) =~ "/foo/bar"
assert redirected_to(conn, :moved_permanently) =~
"/foo/bar"
```

response(conn, given)

```
@spec response(Plug.Conn.t(), status :: integer() |
atom()) :: binary()
```

Asserts the given status code and returns the response body if one was set or sent.

Examples

```
conn = get(build_conn(), "/")
assert response(conn, 200) =~ "hello world"
```

response_content_type(conn, format)

```
@spec response_content_type(Plug.Conn.t(), atom()) ::
String.t()
```

Returns the content type as long as it matches the given format.

Examples

```
# Assert we have an html response with utf-8 charset
assert response_content_type(conn, :html) =~
"charset=utf-8"
```

text_response(conn, status)

```
@spec text_response(Plug.Conn.t(), status :: integer() |
atom()) :: String.t()
```

Asserts the given status code, that we have a text response and returns the response body if one was set or sent.

Examples

```
assert text_response(conn, 200) =~ "hello"
```

trace(conn, path_or_action, params_or_body \\ nil)

(macro)

Dispatches to the current endpoint.

See `dispatch/5` for more information.

Phoenix.CodeReloader

A plug and module to handle automatic code reloading.

To avoid race conditions, all code reloads are funneled through a sequential call operation.

Summary

Functions

call(conn, opts)

API used by Plug to invoke the code reloader on every request.

init(opts)

API used by Plug to start the code reloader.

reload(endpoint, opts \\ [])

Reloads code for the current Mix project by invoking the `:reloadable_compilers` on the list of `:reloadable_apps`.

reload!(endpoint, opts)

Same as `reload/1` but it will raise if Mix is not available.

sync()

Synchronizes with the code server if it is alive.

Functions

call(conn, opts)

API used by Plug to invoke the code reloader on every request.

init(opts)

API used by Plug to start the code reloader.

reload(endpoint, opts \\ [])

```
@spec reload(
  module(),
  keyword()
) :: :ok | {:error, binary()}
```

Reloads code for the current Mix project by invoking the `:reloadable_compilers` on the list of `:reloadable_apps`.

This is configured in your application environment like:

```
config :your_app, YourAppWeb.Endpoint,
  reloadable_compilers: [:gettext, :elixir],
  reloadable_apps: [:ui, :backend]
```

Keep in mind `:reloadable_compilers` must be a subset of the `:compilers` specified in `project/0` in your `mix.exs`.

The `:reloadable_apps` defaults to `nil`. In such case default behaviour is to reload the current project if it consists of a single app, or all applications within an umbrella project. You can set `:reloadable_apps` to a subset of default applications to reload only some of them, an empty list - to effectively disable the code reloader, or include external applications from library dependencies.

This function is a no-op and returns `:ok` if Mix is not available.

Options

- `:reloadable_args` - additional CLI args to pass to the compiler tasks. Defaults to `["--no-all-warnings"]` so only warnings related to the files being compiled are printed

reload!(endpoint, opts)

```
@spec reload!(
  module(),
  keyword()
) :: :ok | {:error, binary()}
```

Same as `reload/1` but it will raise if Mix is not available.

sync()

```
@spec sync() :: :ok
```

Synchronizes with the code server if it is alive.

It returns `:ok`. If it is not running, it also returns `:ok`.

Phoenix.Endpoint.Cowboy2Adapter

The Cowboy2 adapter for Phoenix.

Endpoint configuration

This adapter uses the following endpoint configuration:

- `:http` - the configuration for the HTTP server. It accepts all options as defined by `Plug.Cowboy`. Defaults to `false`
- `:https` - the configuration for the HTTPS server. It accepts all options as defined by `Plug.Cowboy`. Defaults to `false`
- `:drainer` - a drainer process that triggers when your application is shutting down to wait for any on-going request to finish. It accepts all options as defined by `Plug.Cowboy.Drainer`. Defaults to `[]`, which will start a drainer process for each configured endpoint, but can be disabled by setting it to `false`.

Custom dispatch options

You can provide custom dispatch options in order to use Phoenix's builtin Cowboy server with custom handlers. For example, to handle raw WebSockets as shown in Cowboy's docs).

The options are passed to both `:http` and `:https` keys in the endpoint configuration. However, once you pass your custom dispatch options, you will need to manually wire the Phoenix endpoint by adding the following rule:

```
{:_, Plug.Cowboy.Handler, {MyAppWeb.Endpoint, []}}
```

For example:

```
config :myapp, MyAppWeb.Endpoint,
  http: [dispatch: [
          {:_, [
              {"/foo", MyAppWeb.CustomHandler, []},
              {:_, Plug.Cowboy.Handler,
{MyAppWeb.Endpoint, []}}
            ]}]]
```

It is also important to specify your handlers first, otherwise Phoenix will intercept the requests before they get to your handler.

Summary

Functions

Functions

server_info(endpoint, scheme)

Phoenix.Endpoint.SyncCodeReloadPlug

Wraps an Endpoint, attempting to sync with Phoenix's code reloader if an exception is raising which indicates that we may be in the middle of a reload.

We detect this by looking at the raised exception and seeing if it indicates that the endpoint is not defined. This indicates that the code reloader may be mid way through a compile, and that we should attempt to retry the request after the compile has completed. This is also why this must be implemented in a separate module (one that is not recompiled in a typical code reload cycle), since otherwise it may be the case that the endpoint itself is not defined.

Summary

Functions

Functions

call(conn, arg)

Callback implementation for `Plug.call/2`.

init(arg)

Callback implementation for `Plug.init/1`.

Phoenix.Digester.Compressor behaviour

Defines the `Phoenix.Digester.Compressor` behaviour for implementing static file compressors.

A custom compressor expects 2 functions to be implemented.

By default, Phoenix uses only `Phoenix.Digester.Gzip` to compress static files, but additional compressors can be defined and added to the digest process.

Example

If you wanted to compress files using an external brotli compression library, you could define a new module implementing the behaviour and add the module to the list of configured Phoenix static compressors.

```
defmodule MyApp.BrotliCompressor do
  @behaviour Phoenix.Digester.Compressor

  def compress_file(file_path, content) do
    valid_extension = Path.extname(file_path) in
Application.fetch_env!(:phoenix, :gzippable_exts)
    {:ok, compressed_content} = :brotli.encode(content)

    if valid_extension && byte_size(compressed_content)
< byte_size(content) do
      {:ok, compressed_content}
    else
      :error
    end
  end

  def file_extensions do
    [".br"]
```

```
  end
end

# config/config.exs
config :phoenix,
  static_compressors: [Phoenix.Digester.Gzip,
MyApp.BrotliCompressor],
  # ...
```

Summary

Callbacks

Callbacks

compress_file(t, binary)

```
@callback compress_file(Path.t(), binary()) :: {:ok,
binary()} | :error
```

file_extensions()

```
@callback file_extensions() :: [String.t(), ...]
```

Phoenix.Digester.Gzip

Gzip compressor for Phoenix.Digester

Summary

Functions

Functions

compress_file(file_path, content)

Callback implementation for `Phoenix.Digester.Compressor.compress_file/2`.

file_extensions()

Callback implementation for Phoenix.Digester.Compressor.file_extensions/0.

Phoenix.Socket.Broadcast

Defines a message sent from pubsub to channels and vice-versa.

The message format requires the following keys:

- `:topic` - The string topic or topic:subtopic pair namespace, for example "messages", "messages:123"
- `:event` - The string event name, for example "phx_join"
- `:payload` - The message payload

Summary

Types

Types

t()

```
@type t() :: %Phoenix.Socket.Broadcast{event: term(),
payload: term(), topic: term()}
```

Phoenix.Socket.Message

Defines a message dispatched over transport to channels and vice-versa.

The message format requires the following keys:

- `:topic` - The string topic or topic:subtopic pair namespace, for example "messages", "messages:123"
- `:event` - The string event name, for example "phx_join"
- `:payload` - The message payload
- `:ref` - The unique string ref
- `:join_ref` - The unique string ref when joining

Summary

Types

t()

Functions

from_map!(map)

Converts a map with string keys into a message struct.

Types

t()

```
@type t() :: %Phoenix.Socket.Message{
  event: term(),
  join_ref: term(),
  payload: term(),
  ref: term(),
  topic: term()
}
```

Functions

from_map!(map)

Converts a map with string keys into a message struct.

Raises `Phoenix.Socket.InvalidMessageError` if not valid.

Phoenix.Socket.Reply

Defines a reply sent from channels to transports.

The message format requires the following keys:

- `:topic` - The string topic or topic:subtopic pair namespace, for example "messages", "messages:123"
- `:status` - The reply status as an atom
- `:payload` - The reply payload
- `:ref` - The unique string ref
- `:join_ref` - The unique string ref when joining

Summary

Types

t()

Types

t()

```
@type t() :: %Phoenix.Socket.Reply{
  join_ref: term(),
  payload: term(),
  ref: term(),
```

```
  status: term(),
  topic: term()
}
```

Phoenix.Socket.Serializer behaviour

A behaviour that serializes incoming and outgoing socket messages.

By default Phoenix provides a serializer that encodes to JSON and decodes JSON messages.

Custom serializers may be configured in the socket.

Summary

Callbacks

Callbacks

decode!(iodata, options)

```
@callback decode!(iodata(), options :: Keyword.t()) ::
Phoenix.Socket.Message.t()
```

Decodes iodata into `Phoenix.Socket.Message` struct.

encode!(arg1)

```
@callback encode!(Phoenix.Socket.Message.t() |
Phoenix.Socket.Reply.t()) ::
  {:socket_push, :text, iodata()} | {:socket_push,
:binary, iodata()}
```

Encodes `Phoenix.Socket.Message` and `Phoenix.Socket.Reply` structs to push format.

fastlane!(t)

```
@callback fastlane!(Phoenix.Socket.Broadcast.t()) ::
  {:socket_push, :text, iodata()} | {:socket_push,
:binary, iodata()}
```

Encodes a `Phoenix.Socket.Broadcast` struct to fastlane format.

Phoenix.Socket.Transport behaviour

Outlines the Socket <-> Transport communication.

Each transport, such as websockets and longpolling, must interact with a socket. This module defines said behaviour.

`Phoenix.Socket` is just one possible implementation of a socket that multiplexes events over multiple channels. If you implement this behaviour, then a transport can directly invoke your implementation, without passing through channels.

This module also provides convenience functions for implementing transports.

Example

Here is a simple echo socket implementation:

```
defmodule EchoSocket do
  @behaviour Phoenix.Socket.Transport

  def child_spec(opts) do
    # We won't spawn any process, so let's ignore the
child spec
    :ignore
  end

  def connect(state) do
    # Callback to retrieve relevant data from the
connection.
    # The map contains options, params, transport and
endpoint keys.
    {:ok, state}
  end

  def init(state) do
```

```
    # Now we are effectively inside the process that
maintains the socket.
    {:ok, state}
  end

  def handle_in({text, _opts}, state) do
    {:reply, :ok, {:text, text}, state}
  end

  def handle_info(_, state) do
    {:ok, state}
  end

  def terminate(_reason, _state) do
    :ok
  end
end
```

It can be mounted in your endpoint like any other socket:

```
socket "/socket", EchoSocket, websocket: true,
longpoll: true
```

You can now interact with the socket under `/socket/websocket` and `/socket/longpoll`.

Custom transports

Sockets are operated by a transport. When a transport is defined, it usually receives a socket module and the module will be invoked when certain events happen at the transport level.

Whenever the transport receives a new connection, it should invoke the `connect/1` callback with a map of metadata. Different sockets may require different metadata.

If the connection is accepted, the transport can move the connection to another process, if so desires, or keep using the same process. The process responsible for managing the socket should then call `init/1`.

For each message received from the client, the transport must call `handle_in/2` on the socket. For each informational message the transport receives, it should call `handle_info/2` on the socket.

Transports can optionally implement `handle_control/2` for handling control frames such as `:ping` and `:pong`.

On termination, `terminate/2` must be called. A special atom with reason `:closed` can be used to specify that the client terminated the connection.

Booting

Whenever your endpoint starts, it will automatically invoke the `child_spec/1` on each listed socket and start that specification under the endpoint supervisor.

Since the socket supervision tree is started by the endpoint, any custom transport must be started after the endpoint in a supervision tree.

Summary

Types

state()

Callbacks

child_spec(keyword)

Returns a child specification for socket management.

connect(transport_info)

Connects to the socket.

drainer_spec(keyword)

Returns a child specification for terminating the socket.

handle_control({}, state)

Handles incoming control frames.

handle_in({}, state)

Handles incoming socket messages.

handle_info(message, state)

Handles info messages.

init(state)

Initializes the socket state.

terminate(reason, state)

Invoked on termination.

Functions

check_origin(conn, handler, endpoint, opts, sender \\ &Plug.Conn.send_resp/1)

Checks the origin request header against the list of allowed origins.

check_subprotocols(conn, subprotocols)

Checks the Websocket subprotocols request header against the allowed subprotocols.

code_reload(conn, endpoint, opts)

Runs the code reloader if enabled.

connect_info(conn, endpoint, keys)

Extracts connection information from `conn` and returns a map.

transport_log(conn, level)

Logs the transport request.

Types

state()

```
@type state() :: term()
```

Callbacks

child_spec(keyword)

```
@callback child_spec(keyword()) ::
:supervisor.child_spec() | :ignore
```

Returns a child specification for socket management.

This is invoked only once per socket regardless of the number of transports and should be responsible for setting up any process structure

used exclusively by the socket regardless of transports.

Each socket connection is started by the transport and the process that controls the socket likely belongs to the transport. However, some sockets spawn new processes, such as `Phoenix.Socket` which spawns channels, and this gives the ability to start a supervision tree associated to the socket.

It receives the socket options from the endpoint, for example:

```
socket "/my_app", MyApp.Socket, shutdown: 5000
```

means `child_spec([shutdown: 5000])` will be invoked.

`:ignore` means no child spec is necessary for this socket.

connect(transport_info)

```
@callback connect(transport_info :: map()) :: {:ok,
state()} | {:error, term()} | :error
```

Connects to the socket.

The transport passes a map of metadata and the socket returns `{:ok, state}`, `{:error, reason}` or `:error`. The state must be stored by the transport and returned in all future operations. When `{:error, reason}` is returned, some transports - such as WebSockets - allow customizing the response based on `reason` via a custom `:error_handler`.

This function is used for authorization purposes and it may be invoked outside of the process that effectively runs the socket.

In the default `Phoenix.Socket` implementation, the metadata expects the following keys:

- `:endpoint` - the application endpoint
- `:transport` - the transport name
- `:params` - the connection parameters
- `:options` - a keyword list of transport options, often given by developers when configuring the transport. It must include a `:serializer` field with the list of serializers and their requirements

drainer_spec(keyword)

(optional)

```
@callback drainer_spec(keyword()) ::
:supervisor.child_spec() | :ignore
```

Returns a child specification for terminating the socket.

This is a process that is started late in the supervision tree with the specific goal of draining connections on application shutdown.

Similar to `child_spec/1`, it receives the socket options from the endpoint.

handle_control({}, state)

(optional)

```
@callback handle_control(
  {message :: term(), opts :: keyword()},
```

```
  state()
) ::
  {:ok, state()}
  | {:reply, :ok | :error, {opcode :: atom(), message ::
term()}, state()}
  | {:stop, reason :: term(), state()}
```

Handles incoming control frames.

The message is represented as `{payload, options}`. It must return one of:

- `{:ok, state}` - continues the socket with no reply
- `{:reply, status, reply, state}` - continues the socket with reply
- `{:stop, reason, state}` - stops the socket

Control frames only supported when using websockets.

The `options` contains an `opcode` key, this will be either `:ping` or `:pong`.

If a control frame doesn't have a payload, then the payload value will be `nil`.

handle_in({}, state)

```
@callback handle_in(
  {message :: term(), opts :: keyword()},
  state()
) ::
  {:ok, state()}
  | {:reply, :ok | :error, {opcode :: atom(), message ::
```

```
term()}, state()}
  | {:stop, reason :: term(), state()}
```

Handles incoming socket messages.

The message is represented as `{payload, options}`. It must return one of:

- `{:ok, state}` - continues the socket with no reply
- `{:reply, status, reply, state}` - continues the socket with reply
- `{:stop, reason, state}` - stops the socket

The `reply` is a tuple contain an `opcode` atom and a message that can be any term. The built-in websocket transport supports both `:text` and `:binary` opcode and the message must be always iodata. Long polling only supports text opcode.

handle_info(message, state)

```
@callback handle_info(message :: term(), state()) ::
  {:ok, state()}
  | {:push, {opcode :: atom(), message :: term()},
state()}
  | {:stop, reason :: term(), state()}
```

Handles info messages.

The message is a term. It must return one of:

- `{:ok, state}` - continues the socket with no reply
- `{:push, reply, state}` - continues the socket with reply
- `{:stop, reason, state}` - stops the socket

The `reply` is a tuple contain an `opcode` atom and a message that can be any term. The built-in websocket transport supports both `:text` and `:binary` opcode and the message must be always iodata. Long polling only supports text opcode.

init(state)

```
@callback init(state()) :: {:ok, state()}
```

Initializes the socket state.

This must be executed from the process that will effectively operate the socket.

terminate(reason, state)

```
@callback terminate(reason :: term(), state()) :: :ok
```

Invoked on termination.

If `reason` is `:closed`, it means the client closed the socket. This is considered a `:normal` exit signal, so linked process will not automatically exit. See `Process.exit/2` for more details on exit signals.

Functions

check_origin(conn, handler, endpoint, opts, sender \\ &Plug.Conn.send_resp/1)

Checks the origin request header against the list of allowed origins.

Should be called by transports before connecting when appropriate. If the origin header matches the allowed origins, no origin header was sent or no origin was configured, it will return the given connection.

Otherwise a 403 Forbidden response will be sent and the connection halted. It is a noop if the connection has been halted.

check_subprotocols(conn, subprotocols)

Checks the Websocket subprotocols request header against the allowed subprotocols.

Should be called by transports before connecting when appropriate. If the sec-websocket-protocol header matches the allowed subprotocols, it will put sec-websocket-protocol response header and return the given connection. If no sec-websocket-protocol header was sent it will return the given connection.

Otherwise a 403 Forbidden response will be sent and the connection halted. It is a noop if the connection has been halted.

code_reload(conn, endpoint, opts)

Runs the code reloader if enabled.

connect_info(conn, endpoint, keys)

Extracts connection information from `conn` and returns a map.

Keys are retrieved from the optional transport option `:connect_info`. This functionality is transport specific. Please refer to your transports' documentation for more information.

The supported keys are:

- `:peer_data` - the result of `Plug.Conn.get_peer_data/1`
- `:trace_context_headers` - a list of all trace context headers
- `:x_headers` - a list of all request headers that have an "x-" prefix
- `:uri` - a `%URI{}` derived from the conn
- `:user_agent` - the value of the "user-agent" request header

transport_log(conn, level)

Logs the transport request.

Available for transports that generate a connection.

Phoenix.ActionClauseError exception

Phoenix.MissingParamError exception

Raised when a key is expected to be present in the request parameters, but is not.

This exception is raised by `Phoenix.Controller.scrub_params/2` which:

- Checks to see if the required_key is present (can be empty)
- Changes all empty parameters to nils ("" -> nil)

If you are seeing this error, you should handle the error and surface it to the end user. It means that there is a parameter missing from the request.

Phoenix.NotAcceptableError exception

Raised when one of the `accept*` headers is not accepted by the server.

This exception is commonly raised by `Phoenix.Controller.accepts/2` which negotiates the media types the server is able to serve with the contents the client is able to render.

If you are seeing this error, you should check if you are listing the desired formats in your `:accepts` plug or if you are setting the proper accept header in the client. The exception contains the acceptable mime types in the `accepts` field.

Phoenix.Router.MalformedURIError exception

Exception raised when the URI is malformed on matching.

Phoenix.Router.NoRouteError exception

Exception raised when no route is found.

Phoenix.Socket.InvalidMessageError exception

Raised when the socket message is invalid.

mix local.phx

Updates the Phoenix project generator locally.

```
$ mix local.phx
```

Accepts the same command line options as `archive.install hex phx_new`.

mix phx

Prints Phoenix tasks and their information.

```
$ mix phx
```

To print the Phoenix version, pass `-v` or `--version`, for example:

```
$ mix phx --version
```

mix phx.digest

Digests and compresses static files.

```
$ mix phx.digest
$ mix phx.digest priv/static -o /www/public
```

The first argument is the path where the static files are located. The `-o` option indicates the path that will be used to save the digested and compressed files.

If no path is given, it will use `priv/static` as the input and output path.

The output folder will contain:

- the original file
- the file compressed with gzip
- a file containing the original file name and its digest
- a compressed file containing the file name and its digest
- a cache manifest file

Example of generated files:

- app.js
- app.js.gz
- app-eb0a5b9302e8d32828d8a73f137cc8f0.js
- app-eb0a5b9302e8d32828d8a73f137cc8f0.js.gz
- cache_manifest.json

You can use `mix phx.digest.clean` to prune stale versions of the assets. If you want to remove all produced files, run `mix phx.digest.clean --all`.

vsn

It is possible to digest the stylesheet asset references without the query string "?vsn=d" with the option `--no-vsn`.

mix phx.digest.clean

Removes old versions of compiled assets.

By default, it will keep the latest version and 2 previous versions as well as any digest created in the last hour.

```
$ mix phx.digest.clean
$ mix phx.digest.clean -o /www/public
$ mix phx.digest.clean --age 600 --keep 3
$ mix phx.digest.clean --all
```

Options

- `-o, --output` - indicates the path to your compiled assets directory. Defaults to `priv/static`
- `--age` - specifies a maximum age (in seconds) for assets. Files older than age that are not in the last `--keep` versions will be removed. Defaults to 3600 (1 hour)
- `--keep` - specifies how many previous versions of assets to keep. Defaults to 2 previous versions
- `--all` - specifies that all compiled assets (including the manifest) will be removed. Note this overrides the age and keep switches.

mix phx.gen

Lists all available Phoenix generators.

CRUD related generators

The table below shows a summary of the contents created by the CRUD generators:

Task	Schema	Migration	Context	Controller	View	LiveView
`phx.gen.embedded`	x					
`phx.gen.schema`	x	x				
`phx.gen.context`	x	x	x			
`phx.gen.live`	x	x	x			x
`phx.gen.json`	x	x	x	x	x	
`phx.gen.html`	x	x	x	x	x	

Summary

Functions

run(args)

Callback implementation for `Mix.Task.run/1`.

Functions

run(args)

Callback implementation for `Mix.Task.run/1`.

mix phx.gen.auth

Generates authentication logic and related views for a resource.

```
$ mix phx.gen.auth Accounts User users
```

The first argument is the context module followed by the schema module and its plural name (used as the schema table name).

Additional information and security considerations are detailed in the `mix phx.gen.auth` guide.

LiveView vs conventional Controllers & Views

Authentication views can either be generated to use LiveView by passing the `--live` option, or they can use conventional Phoenix Controllers & Views by passing `--no-live`.

If neither of these options are provided, a prompt will be displayed.

Using the `--live` option is advised if you plan on using LiveView elsewhere in your application. The user experience when navigating between LiveViews can be tightly controlled, allowing you to let your users navigate to authentication views without necessarily triggering a new HTTP request each time (which would result in a full page load).

Password hashing

The password hashing mechanism defaults to `bcrypt` for Unix systems and `pbkdf2` for Windows systems. Both systems use the Comeonin interface.

The password hashing mechanism can be overridden with the `--hashing-lib` option. The following values are supported:

- `bcrypt` - bcrypt_elixir
- `pbkdf2` - pbkdf2_elixir
- `argon2` - argon2_elixir

We recommend developers to consider using `argon2`, which is the most robust of all 3. The downside is that `argon2` is quite CPU and memory intensive, and you will need more powerful instances to run your applications on.

For more information about choosing these libraries, see the Comeonin project.

Web namespace

By default, the controllers and HTML view will be namespaced by the schema name. You can customize the web module namespace by passing the `--web` flag with a module name, for example:

```
$ mix phx.gen.auth Accounts User users --web Warehouse
```

Which would generate the controllers, views, templates and associated tests nested in the `MyAppWeb.Warehouse` namespace:

- `lib/my_app_web/controllers/warehouse/user_auth.ex`
- `lib/my_app_web/controllers/warehouse/user_confirmation_controller.ex`
- `lib/my_app_web/controllers/warehouse/user_confirmation_html.ex`
- `lib/my_app_web/controllers/warehouse/user_confirmation_html/new.html.heex`
- `test/my_app_web/controllers/warehouse/user_auth_test.exs`
- `test/my_app_web/controllers/warehouse/user_confirmation_controller_test.exs`
- and so on...

Multiple invocations

You can invoke this generator multiple times. This is typically useful if you have distinct resources that go through distinct authentication workflows:

```
$ mix phx.gen.auth Store User users
$ mix phx.gen.auth Backoffice Admin admins
```

Binary ids

The `--binary-id` option causes the generated migration to use `binary_id` for its primary key and foreign keys.

Default options

This generator uses default options provided in the `:generators` configuration of your application. These are the defaults:

```
config :your_app, :generators,
  binary_id: false,
  sample_binary_id: "11111111-1111-1111-1111-
111111111111"
```

You can override those options per invocation by providing corresponding switches, e.g. `--no-binary-id` to use normal ids despite the default configuration.

Custom table names

By default, the table name for the migration and schema will be the plural name provided for the resource. To customize this value, a `--table` option may be provided. For example:

```
$ mix phx.gen.auth Accounts User users --table
accounts_users
```

This will cause the generated tables to be named `"accounts_users"` and `"accounts_users_tokens"`.

mix phx.gen.cert

Generates a self-signed certificate for HTTPS testing.

```
$ mix phx.gen.cert
$ mix phx.gen.cert my-app my-app.local my-
app.internal.example.com
```

Creates a private key and a self-signed certificate in PEM format. These files can be referenced in the `certfile` and `keyfile` parameters of an HTTPS Endpoint.

WARNING: only use the generated certificate for testing in a closed network environment, such as running a development server on `localhost`. For production, staging, or testing servers on the public internet, obtain a proper certificate, for example from Let's Encrypt.

NOTE: when using Google Chrome, open chrome://flags/#allow-insecure-localhost to enable the use of self-signed certificates on `localhost`.

Arguments

The list of hostnames, if none are specified, defaults to:

- localhost

Other (optional) arguments:

- `--output` (`-o`): the path and base filename for the certificate and key (default: priv/cert/selfsigned)
- `--name` (`-n`): the Common Name value in certificate's subject (default: "Self-signed test certificate")

Requires OTP 21.3 or later.

mix phx.gen.channel

Generates a Phoenix channel.

```
$ mix phx.gen.channel Room
```

Accepts the module name for the channel

The generated files will contain:

For a regular application:

- a channel in `lib/my_app_web/channels`
- a channel test in `test/my_app_web/channels`

For an umbrella application:

- a channel in `apps/my_app_web/lib/app_name_web/channels`
- a channel test in `apps/my_app_web/test/my_app_web/channels`

mix phx.gen.context

Generates a context with functions around an Ecto schema.

```
$ mix phx.gen.context Accounts User users name:string
age:integer
```

The first argument is the context module followed by the schema module and its plural name (used as the schema table name).

The context is an Elixir module that serves as an API boundary for the given resource. A context often holds many related resources. Therefore, if the context already exists, it will be augmented with functions for the given resource.

> Note: A resource may also be split over distinct contexts (such as Accounts.User and Payments.User).

The schema is responsible for mapping the database fields into an Elixir struct.

Overall, this generator will add the following files to `lib/your_app`:

- a context module in `accounts.ex`, serving as the API boundary
- a schema in `accounts/user.ex`, with a `users` table

A migration file for the repository and test files for the context will also be generated.

Generating without a schema

In some cases, you may wish to bootstrap the context module and tests, but leave internal implementation of the context and schema to yourself. Use the `--no-schema` flags to accomplish this.

table

By default, the table name for the migration and schema will be the plural name provided for the resource. To customize this value, a `--table` option may be provided. For example:

```
$ mix phx.gen.context Accounts User users --table
cms_users
```

binary_id

Generated migration can use `binary_id` for schema's primary key and its references with option `--binary-id`.

Default options

This generator uses default options provided in the `:generators` configuration of your application. These are the defaults:

```
config :your_app, :generators,
  migration: true,
  binary_id: false,
  timestamp_type: :naive_datetime,
  sample_binary_id: "11111111-1111-1111-1111-
111111111111"
```

You can override those options per invocation by providing corresponding switches, e.g. `--no-binary-id` to use normal ids despite the default configuration or `--migration` to force generation of the migration.

Read the documentation for `phx.gen.schema` for more information on attributes.

Skipping prompts

This generator will prompt you if there is an existing context with the same name, in order to provide more instructions on how to correctly use phoenix contexts. You can skip this prompt and automatically merge the new schema access functions and tests into the existing context using `--merge-with-existing-context`. To prevent changes to the existing context and exit the generator, use `--no-merge-with-existing-context`.

mix phx.gen.embedded

Generates an embedded Ecto schema for casting/validating data outside the DB.

```
mix phx.gen.embedded Blog.Post title:string
views:integer
```

The first argument is the schema module followed by the schema attributes.

The generated schema above will contain:

- an embedded schema file in `lib/my_app/blog/post.ex`

Attributes

The resource fields are given using `name:type` syntax where type are the types supported by Ecto. Omitting the type makes it default to `:string`:

```
mix phx.gen.embedded Blog.Post title views:integer
```

The following types are supported:

- `:integer`
- `:float`
- `:decimal`
- `:boolean`
- `:map`

- `:string`
- `:array`
- `:references`
- `:text`
- `:date`
- `:time`
- `:time_usec`
- `:naive_datetime`
- `:naive_datetime_usec`
- `:utc_datetime`
- `:utc_datetime_usec`
- `:uuid`
- `:binary`
- `:enum`
- `:datetime` - An alias for `:naive_datetime`

mix phx.gen.html

Generates controller with view, templates, schema and context for an HTML resource.

```
mix phx.gen.html Accounts User users name:string
age:integer
```

The first argument, `Accounts`, is the resource's context. A context is an Elixir module that serves as an API boundary for closely related resources.

The second argument, `User`, is the resource's schema. A schema is an Elixir module responsible for mapping database fields into an Elixir struct. The `User` schema above specifies two fields with their respective colon-delimited data types: `name:string` and `age:integer`. See `mix phx.gen.schema` for more information on attributes.

> Note: A resource may also be split over distinct contexts (such as `Accounts.User` and `Payments.User`).

This generator adds the following files to `lib/`:

- a controller in `lib/my_app_web/controllers/user_controller.ex`
- default CRUD HTML templates in `lib/my_app_web/controllers/user_html`
- an HTML view collocated with the controller in `lib/my_app_web/controllers/user_html.ex`
- a schema in `lib/my_app/accounts/user.ex`, with an `users` table
- a context module in `lib/my_app/accounts.ex` for the accounts API

Additionally, this generator creates the following files:

- a migration for the schema in `priv/repo/migrations`
- a controller test module in `test/my_app/controllers/user_controller_test.exs`
- a context test module in `test/my_app/accounts_test.exs`
- a context test helper module in `test/support/fixtures/accounts_fixtures.ex`

If the context already exists, this generator injects functions for the given resource into the context, context test, and context test helper modules.

Umbrella app configuration

By default, Phoenix injects both web and domain specific functionality into the same application. When using umbrella applications, those concerns are typically broken into two separate apps, your context application - let's call it `my_app` - and its web layer, which Phoenix assumes to be `my_app_web`.

You can teach Phoenix to use this style via the `:context_app` configuration option in your `my_app_umbrella/config/config.exs`:

```
config :my_app_web,
  ecto_repos: [Stuff.Repo],
  generators: [context_app: :my_app]
```

Alternatively, the `--context-app` option may be supplied to the generator:

```
mix phx.gen.html Sales User users --context-app my_app
```

If you delete the `:context_app` configuration option, Phoenix will automatically put generated web files in `my_app_umbrella/apps/my_app_web_web`.

If you change the value of `:context_app` to `:new_value`, `my_app_umbrella/apps/new_value_web` must already exist or you will get the following error:

** (Mix) no directory for context_app :new_value found in my_app_web's deps.

Web namespace

By default, the controller and HTML view will be namespaced by the schema name. You can customize the web module namespace by passing the `--web` flag with a module name, for example:

```
mix phx.gen.html Sales User users --web Sales
```

Which would generate a `lib/app_web/controllers/sales/user_controller.ex` and `lib/app_web/controllers/sales/user_html.ex`.

Customizing the context, schema, tables and migrations

In some cases, you may wish to bootstrap HTML templates, controllers, and controller tests, but leave internal implementation of the context or schema to yourself. You can use the `--no-context` and `--no-schema` flags for file generation control.

You can also change the table name or configure the migrations to use binary ids for primary keys, see `mix phx.gen.schema` for more information.

mix phx.gen.json

Generates controller, JSON view, and context for a JSON resource.

```
mix phx.gen.json Accounts User users name:string
age:integer
```

The first argument is the context module followed by the schema module and its plural name (used as the schema table name).

The context is an Elixir module that serves as an API boundary for the given resource. A context often holds many related resources. Therefore, if the context already exists, it will be augmented with functions for the given resource.

> Note: A resource may also be split over distinct contexts (such as `Accounts.User` and `Payments.User`).

The schema is responsible for mapping the database fields into an Elixir struct. It is followed by an optional list of attributes, with their respective names and types. See `mix phx.gen.schema` for more information on attributes.

Overall, this generator will add the following files to `lib/`:

- a context module in `lib/app/accounts.ex` for the accounts API
- a schema in `lib/app/accounts/user.ex`, with an `users` table
- a controller in `lib/app_web/controllers/user_controller.ex`
- a JSON view collocated with the controller in `lib/app_web/controllers/user_json.ex`

A migration file for the repository and test files for the context and controller features will also be generated.

API Prefix

By default, the prefix "/api" will be generated for API route paths. This can be customized via the `:api_prefix` generators configuration:

```
config :your_app, :generators,
  api_prefix: "/api/v1"
```

The context app

The location of the web files (controllers, json views, etc) in an umbrella application will vary based on the `:context_app` config located in your applications `:generators` configuration. When set, the Phoenix generators will generate web files directly in your lib and test folders since the application is assumed to be isolated to web specific functionality. If `:context_app` is not set, the generators will place web related lib and test files in a `web/` directory since the application is assumed to be handling both web and domain specific functionality. Example configuration:

```
config :my_app_web, :generators, context_app: :my_app
```

Alternatively, the `--context-app` option may be supplied to the generator:

```
mix phx.gen.json Sales User users --context-app
warehouse
```

Web namespace

By default, the controller and json view will be namespaced by the schema name. You can customize the web module namespace by passing the `--web` flag with a module name, for example:

```
mix phx.gen.json Sales User users --web Sales
```

Which would generate a `lib/app_web/controllers/sales/user_controller.ex` and `lib/app_web/controller/sales/user_json.ex`.

Customizing the context, schema, tables and migrations

In some cases, you may wish to bootstrap JSON views, controllers, and controller tests, but leave internal implementation of the context or schema to yourself. You can use the `--no-context` and `--no-schema` flags for file generation control.

You can also change the table name or configure the migrations to use binary ids for primary keys, see `mix phx.gen.schema` for more information.

mix phx.gen.live

Generates LiveView, templates, and context for a resource.

```
mix phx.gen.live Accounts User users name:string
age:integer
```

The first argument is the context module. The context is an Elixir module that serves as an API boundary for the given resource. A context often holds many related resources. Therefore, if the context already exists, it will be augmented with functions for the given resource.

The second argument is the schema module. The schema is responsible for mapping the database fields into an Elixir struct.

The remaining arguments are the schema module plural name (used as the schema table name), and an optional list of attributes as their respective names and types. See `mix help phx.gen.schema` for more information on attributes.

When this command is run for the first time, a `Components` module will be created if it does not exist, along with the resource level LiveViews and components, including `UserLive.Index`, `UserLive.Show`, and `UserLive.FormComponent` modules for the new resource.

> Note: A resource may also be split over distinct contexts (such as `Accounts.User` and `Payments.User`).

Overall, this generator will add the following files:

- a context module in `lib/app/accounts.ex` for the accounts API
- a schema in `lib/app/accounts/user.ex`, with a `users` table
- a LiveView in `lib/app_web/live/user_live/show.ex`
- a LiveView in `lib/app_web/live/user_live/index.ex`

- a LiveComponent in `lib/app_web/live/user_live/form_component.ex`
- a helpers module in `lib/app_web/live/live_helpers.ex` with a modal

After file generation is complete, there will be output regarding required updates to the `lib/app_web/router.ex` file.

```
Add the live routes to your browser scope in
lib/app_web/router.ex:

  live "/users", UserLive.Index, :index
  live "/users/new", UserLive.Index, :new
  live "/users/:id/edit", UserLive.Index, :edit

  live "/users/:id", UserLive.Show, :show
  live "/users/:id/show/edit", UserLive.Show, :edit
```

The context app

A migration file for the repository and test files for the context and controller features will also be generated.

The location of the web files (LiveView's, views, templates, etc.) in an umbrella application will vary based on the `:context_app` config located in your applications `:generators` configuration. When set, the Phoenix generators will generate web files directly in your lib and test folders since the application is assumed to be isolated to web specific functionality. If `:context_app` is not set, the generators will place web related lib and test files in a `web/` directory since the application is assumed to be handling both web and domain specific functionality. Example configuration:

```
config :my_app_web, :generators, context_app: :my_app
```

Alternatively, the `--context-app` option may be supplied to the generator:

```
mix phx.gen.live Accounts User users --context-app
warehouse
```

Web namespace

By default, the LiveView modules will be namespaced by the web module. You can customize the web module namespace by passing the `--web` flag with a module name, for example:

```
mix phx.gen.live Accounts User users --web Sales
```

Which would generate the LiveViews in `lib/app_web/live/sales/user_live/`, namespaced `AppWeb.Sales.UserLive` instead of `AppWeb.UserLive`.

Customizing the context, schema, tables and migrations

In some cases, you may wish to bootstrap HTML templates, LiveViews, and tests, but leave internal implementation of the context or schema to yourself. You can use the `--no-context` and `--no-schema` flags for file generation control.

```
mix phx.gen.live Accounts User users --no-context --no-
schema
```

In the cases above, tests are still generated, but they will all fail.

You can also change the table name or configure the migrations to use binary ids for primary keys, see `mix help phx.gen.schema` for more information.

mix phx.gen.notifier

Generates a notifier that delivers emails by default.

```
$ mix phx.gen.notifier Accounts User welcome_user
reset_password confirmation_instructions
```

This task expects a context module name, followed by a notifier name and one or more message names. Messages are the functions that will be created prefixed by "deliver", so the message name should be "snake_case" without punctuation.

Additionally a context app can be specified with the flag `--context-app`, which is useful if the notifier is being generated in a different app under an umbrella.

```
$ mix phx.gen.notifier Accounts User welcome_user --
context-app marketing
```

The app "marketing" must exist before the command is executed.

mix phx.gen.presence

Generates a Presence tracker.

```
$ mix phx.gen.presence
$ mix phx.gen.presence MyPresence
```

The argument, which defaults to `Presence`, defines the module name of the Presence tracker.

Generates a new file, `lib/my_app_web/channels/my_presence.ex`, where `my_presence` is the snake-cased version of the provided module name.

mix phx.gen.release

Generates release files and optional Dockerfile for release-based deployments.

The following release files are created:

- `lib/app_name/release.ex` - A release module containing tasks for running migrations inside a release
- `rel/overlays/bin/migrate` - A migrate script for conveniently invoking the release system migrations
- `rel/overlays/bin/server` - A server script for conveniently invoking the release system with environment variables to start the phoenix web server

Note, the `rel/overlays` directory is copied into the release build by default when running `mix release`.

To skip generating the migration-related files, use the `--no-ecto` flag. To force these migration-related files to be generated, the use `--ecto` flag.

Docker

When the `--docker` flag is passed, the following docker files are generated:

- `Dockerfile` - The Dockerfile for use in any standard docker deployment
- `.dockerignore` - A docker ignore file with standard elixir defaults

For extended release configuration, the `mix release.init` task can be used in addition to this task. See the `Mix.Release` docs for more details.

Summary

Functions

Functions

otp_vsn()

mix phx.gen.schema

Generates an Ecto schema and migration.

```
$ mix phx.gen.schema Blog.Post blog_posts title:string
views:integer
```

The first argument is the schema module followed by its plural name (used as the table name).

The generated schema above will contain:

- a schema file in `lib/my_app/blog/post.ex`, with a `blog_posts` table
- a migration file for the repository

The generated migration can be skipped with `--no-migration`.

Contexts

Your schemas can be generated and added to a separate OTP app. Make sure your configuration is properly setup or manually specify the context app with the `--context-app` option with the CLI.

Via config:

```
config :marketing_web, :generators, context_app:
:marketing
```

Via CLI:

```
$ mix phx.gen.schema Blog.Post blog_posts title:string
views:integer --context-app marketing
```

Attributes

The resource fields are given using `name:type` syntax where type are the types supported by Ecto. Omitting the type makes it default to `:string`:

```
$ mix phx.gen.schema Blog.Post blog_posts title
views:integer
```

The following types are supported:

- `:integer`
- `:float`
- `:decimal`
- `:boolean`
- `:map`
- `:string`
- `:array`
- `:references`
- `:text`
- `:date`
- `:time`
- `:time_usec`
- `:naive_datetime`

- `:naive_datetime_usec`
- `:utc_datetime`
- `:utc_datetime_usec`
- `:uuid`
- `:binary`
- `:enum`
- `:datetime` - An alias for `:naive_datetime`

The generator also supports references, which we will properly associate the given column to the primary key column of the referenced table:

```
$ mix phx.gen.schema Blog.Post blog_posts title
user_id:references:users
```

This will result in a migration with an `:integer` column of `:user_id` and create an index.

Furthermore an array type can also be given if it is supported by your database, although it requires the type of the underlying array element to be given too:

```
$ mix phx.gen.schema Blog.Post blog_posts
tags:array:string
```

Unique columns can be automatically generated by using:

```
$ mix phx.gen.schema Blog.Post blog_posts title:unique
unique_int:integer:unique
```

Redact columns can be automatically generated by using:

```
$ mix phx.gen.schema Accounts.Superhero superheroes
secret_identity:redact password:string:redact
```

Ecto.Enum fields can be generated by using:

```
$ mix phx.gen.schema Blog.Post blog_posts title
status:enum:unpublished:published:deleted
```

If no data type is given, it defaults to a string.

table

By default, the table name for the migration and schema will be the plural name provided for the resource. To customize this value, a `--table` option may be provided. For example:

```
$ mix phx.gen.schema Blog.Post posts --table cms_posts
```

binary_id

Generated migration can use `binary_id` for schema's primary key and its references with option `--binary-id`.

repo

Generated migration can use `repo` to set the migration repository folder with option `--repo`:

```
$ mix phx.gen.schema Blog.Post posts --repo
MyApp.Repo.Auth
```

migration_dir

Generated migrations can be added to a specific `--migration-dir` which sets the migration folder path:

```
$ mix phx.gen.schema Blog.Post posts --migration-dir
/path/to/directory
```

prefix

By default migrations and schemas are generated without a prefix.

For PostgreSQL this sets the "SCHEMA" (typically set via `search_path`) and for MySQL it sets the database for the generated migration and schema. The prefix can be used to thematically organize your tables on the database level.

A prefix can be specified with the `--prefix` flags. For example:

```
$ mix phx.gen.schema Blog.Post posts --prefix blog
```

> **Warning**
>
> The flag does not generate migrations to create the schema / database. This needs to be done manually or in a separate migration.

Default options

This generator uses default options provided in the `:generators` configuration of your application. These are the defaults:

```
config :your_app, :generators,
  migration: true,
  binary_id: false,
  timestamp_type: :naive_datetime,
```

```
  sample_binary_id: "11111111-1111-1111-1111-
111111111111"
```

You can override those options per invocation by providing corresponding switches, e.g. `--no-binary-id` to use normal ids despite the default configuration or `--migration` to force generation of the migration.

UTC timestamps

By setting the `:timestamp_type` to `:utc_datetime`, the timestamps will be created using the UTC timezone. This results in a `DateTime` struct instead of a `NaiveDateTime`. This can also be set to `:utc_datetime_usec` for microsecond precision.

mix phx.gen.secret

Generates a secret and prints it to the terminal.

```
$ mix phx.gen.secret [length]
```

By default, mix phx.gen.secret generates a key 64 characters long.

The minimum value for `length` is 32.

mix phx.gen.socket

Generates a Phoenix socket handler.

```
$ mix phx.gen.socket User
```

Accepts the module name for the socket.

The generated files will contain:

For a regular application:

- a client in `assets/js`
- a socket in `lib/my_app_web/channels`

For an umbrella application:

- a client in `apps/my_app_web/assets/js`
- a socket in `apps/my_app_web/lib/my_app_web/channels`

You can then generate channels with `mix phx.gen.channel`.

mix phx.new

Creates a new Phoenix project.

It expects the path of the project as an argument.

```
$ mix phx.new PATH [--module MODULE] [--app APP]
```

A project at the given PATH will be created. The application name and module name will be retrieved from the path, unless `--module` or `--app` is given.

Options

- `--umbrella` - generate an umbrella project, with one application for your domain, and a second application for the web interface.
- `--app` - the name of the OTP application
- `--module` - the name of the base module in the generated skeleton
- `--database` - specify the database adapter for Ecto. One of:
  - `postgres` - via https://github.com/elixir-ecto/postgrex
  - `mysql` - via https://github.com/elixir-ecto/myxql
  - `mssql` - via https://github.com/livehelpnow/tds
  - `sqlite3` - via https://github.com/elixir-sqlite/ecto_sqlite3

  Please check the driver docs for more information and requirements. Defaults to "postgres".
- `--adapter` - specify the http adapter. One of:
  - `cowboy` - via https://github.com/elixir-plug/plug_cowboy

    - `bandit` - via https://github.com/mtrudel/bandit

  Please check the adapter docs for more information and requirements. Defaults to "bandit".

- `--no-assets` - equivalent to `--no-esbuild` and `--no-tailwind`
- `--no-dashboard` - do not include Phoenix.LiveDashboard
- `--no-ecto` - do not generate Ecto files
- `--no-esbuild` - do not include esbuild dependencies and assets. We do not recommend setting this option, unless for API only applications, as doing so requires you to manually add and track JavaScript dependencies
- `--no-gettext` - do not generate gettext files
- `--no-html` - do not generate HTML views
- `--no-live` - comment out LiveView socket setup in your Endpoint and assets/js/app.js. Automatically disabled if --no-html is given
- `--no-mailer` - do not generate Swoosh mailer files
- `--no-tailwind` - do not include tailwind dependencies and assets. The generated markup will still include Tailwind CSS classes, those are left-in as reference for the subsequent styling of your layout and components
- `--binary-id` - use `binary_id` as primary key type in Ecto schemas
- `--verbose` - use verbose output
- `-v`, `--version` - prints the Phoenix installer version

When passing the `--no-ecto` flag, Phoenix generators such as `phx.gen.html`, `phx.gen.json`, `phx.gen.live`, and `phx.gen.context` may no longer work as expected as they generate context files that rely on Ecto for the database access. In those cases, you can pass the `--no-context` flag to generate most of the HTML and JSON files but skip the context, allowing you to fill in the blanks as desired.

Similarly, if `--no-html` is given, the files generated by `phx.gen.html` will no longer work, as important HTML components will be missing.

Installation

`mix phx.new` by default prompts you to fetch and install your dependencies. You can enable this behaviour by passing the `--install` flag or disable it with the `--no-install` flag.

Examples

```
$ mix phx.new hello_world
```

Is equivalent to:

```
$ mix phx.new hello_world --module HelloWorld
```

Or without the HTML and JS bits (useful for APIs):

```
$ mix phx.new ~/Workspace/hello_world --no-html --no-
assets
```

As an umbrella:

```
$ mix phx.new hello --umbrella
```

Would generate the following directory structure and modules:

```
hello_umbrella/    Hello.Umbrella
  apps/
    hello/         Hello
    hello_web/     HelloWeb
```

You can read more about umbrella projects using the official Elixir guide

PHX_NEW_CACHE_DIR

In rare cases, it may be useful to copy the build from a previously cached build. To do this, set the `PHX_NEW_CACHE_DIR` environment variable before running `mix phx.new`. For example, you could generate a cache by running:

```
mix phx.new mycache --no-install && cd mycache   && mix
deps.get && mix deps.compile && mix assets.setup   &&
rm -rf assets config lib priv test mix.exs README.md
```

Your cached build directory should contain:

```
_build
deps
mix.lock
```

Then you could run:

```
PHX_NEW_CACHE_DIR=/path/to/mycache mix phx.new myapp
```

The entire cache directory will be copied to the new project, replacing any existing files where conflicts exist.

mix phx.new.ecto

Creates a new Ecto project within an umbrella project.

This task is intended to create a bare Ecto project without web integration, which serves as a core application of your domain for web applications and your greater umbrella platform to integrate with.

It expects the name of the project as an argument.

```
$ cd my_umbrella/apps
$ mix phx.new.ecto APP [--module MODULE] [--app APP]
```

A project at the given APP directory will be created. The application name and module name will be retrieved from the application name, unless `--module` or `--app` is given.

Options

- `--app` - the name of the OTP application
- `--module` - the name of the base module in the generated skeleton
- `--database` - specify the database adapter for Ecto. One of:
  - `postgres` - via https://github.com/elixir-ecto/postgrex
  - `mysql` - via https://github.com/elixir-ecto/myxql
  - `mssql` - via https://github.com/livehelpnow/tds
  - `sqlite3` - via https://github.com/elixir-sqlite/ecto_sqlite3

  Please check the driver docs for more information and requirements. Defaults to "postgres".

- `--binary-id` - use `binary_id` as primary key type in Ecto schemas

Examples

```
$ mix phx.new.ecto hello_ecto
```

Is equivalent to:

```
$ mix phx.new.ecto hello_ecto --module HelloEcto
```

mix phx.new.web

Creates a new Phoenix web project within an umbrella project.

It expects the name of the OTP app as the first argument and for the command to be run inside your umbrella application's apps directory:

```
$ cd my_umbrella/apps
$ mix phx.new.web APP [--module MODULE] [--app APP]
```

This task is intended to create a bare Phoenix project without database integration, which interfaces with your greater umbrella application(s).

Examples

```
$ mix phx.new.web hello_web
```

Is equivalent to:

```
$ mix phx.new.web hello_web --module HelloWeb
```

Supports the same options as the `phx.new` task. See `Mix.Tasks.Phx.New` for details.

mix phx.routes

Prints all routes for the default or a given router. Can also locate the controller function behind a specified url.

```
$ mix phx.routes [ROUTER] [--info URL]
```

The default router is inflected from the application name unless a configuration named `:namespace` is set inside your application configuration. For example, the configuration:

```
config :my_app,
  namespace: My.App
```

will exhibit the routes for `My.App.Router` when this task is invoked without arguments.

Umbrella projects do not have a default router and therefore always expect a router to be given. An alias can be added to mix.exs to automate this:

```
defp aliases do
  [
    "phx.routes": "phx.routes MyAppWeb.Router",
    # aliases...
  ]
```

Options

- `--info` - locate the controller function definition called by the given url

- `--method` - what HTTP method to use with the given url, only works when used with `--info` and defaults to `get`

Examples

Print all routes for the default router:

```
$ mix phx.routes
```

Print all routes for the given router:

```
$ mix phx.routes MyApp.AnotherRouter
```

Print information about the controller function called by a specified url:

```
$ mix phx.routes --info http://0.0.0.0:4000/home
  Module: RouteInfoTestWeb.PageController
  Function: :index
  /home/my_app/controllers/page_controller.ex:4
```

Print information about the controller function called by a specified url and HTTP method:

```
$ mix phx.routes --info http://0.0.0.0:4000/users --
method post
  Module: RouteInfoTestWeb.UserController
  Function: :create
  /home/my_app/controllers/user_controller.ex:24
```

Summary

Functions

Functions

get_url_info(url, arg)

mix phx.server

Starts the application by configuring all endpoints servers to run.

Note: to start the endpoint without using this mix task you must set `server: true` in your `Phoenix.Endpoint` configuration.

Command line options

- `--open` - open browser window for each started endpoint

Furthermore, this task accepts the same command-line options as `mix run`.

For example, to run `phx.server` without recompiling:

```
$ mix phx.server --no-compile
```

The `--no-halt` flag is automatically added.

Note that the `--no-deps-check` flag cannot be used this way, because Mix needs to check dependencies to find `phx.server`.

To run `phx.server` without checking dependencies, you can run:

```
$ mix do deps.loadpaths --no-deps-check, phx.server
```