Introduction

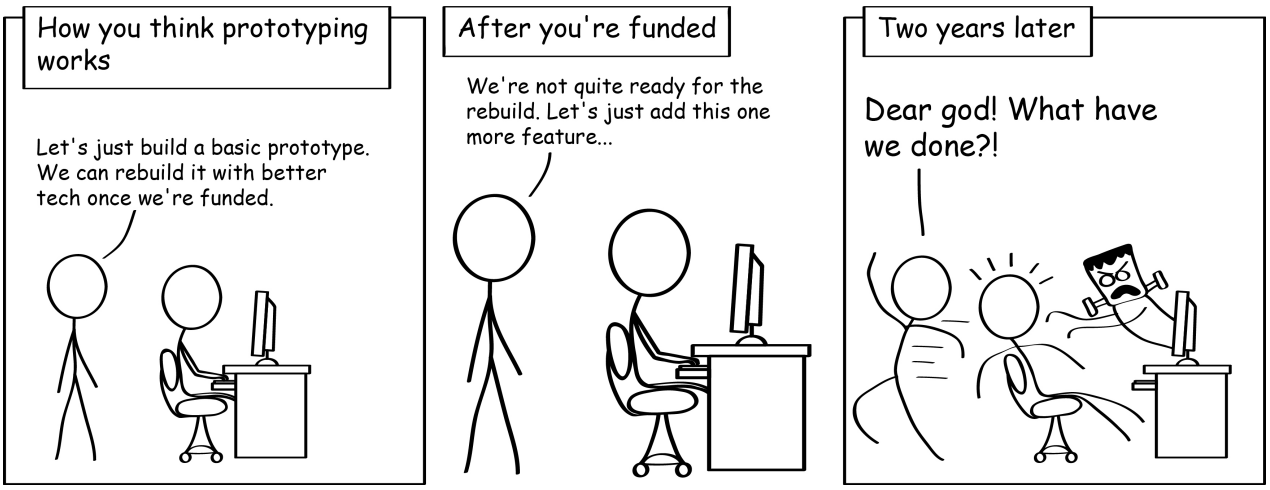
- What is Phoenix?
- What is React?
- What will be covered?
- What are some alternatives?

As any experienced developer knows, scaling is not a trivial problem. In fact, if your product is successful, building the app is often the easiest part of the process.

Scaling is more than just handling concurrent connections or a boat-load of simultaneous requests. It also means dealing with a large team pushing code at the same time, building architecture that makes it easy to change pieces of the app without breaking everything, and knowing about important DevOps concepts that are generally ignored by most tutorials.

Scaling problems are much harder to predict, track down, and prevent if you do not consider them from the outset of your project. You need to be aware of every inefficiency, ensure actions from different customers do not conflict, grow your database, handle errors, tolerate server failures, among a litany of other issues that a smaller app simply does not need to consider.

Developers who have to deal with legacy code on a regular basis know the feeling of staring at a codebase that looks like a Frankenstein monster of outdated technologies and an assortment of hacks that could fall apart at any moment. Refactoring would take too much time, so you just continue to pile onto the monster and pretend that everything is fine.



So why not just build your app in a framework that is designed from the ground up to easily handle your scaling concerns, refactors without server shutdowns, and comes in an easily-understood syntax? Enter Phoenix.

What is Phoenix?

Phoenix is a web framework built with Elixir that is able to leverage the insanely powerful Erlang VM gives you the ability to handle millions of concurrent connections, while also remaining fault-tolerant and easy to implement.

To give you a real comparison, here is how Phoenix performs compared to other popular frameworks (data from mroth):

Framework	Throughput (req/s)	Latency (ms)	Consistency (σ ms)
Gin	51483.20	1.94	0.63
Phoenix	43063.45	2.82	*(1)* 7.46
Express Cluster	27669.46	3.73	2.12
Martini	14798.46	6.81	10.34
Sinatra	9182.86	6.55	3.03
Express	9965.56	10.07	0.95
Rails	3274.81	17.25	6.88
Plug *(1)*	54948.14	3.83	12.40
Play *(2)*	63256.20	1.62	2.96

This is the same underlying technology that WhatsApp uses and one of the reasons Facebook paid $19b for them. Check out this article in Wired appropriately titled Why WhatsApp Only Needs 50 Engineers for Its 900M Users. As you might guess, it has to do with Erlang.

Phoenix uses the programming language Elixir, which is a functional programming language built on Erlang. The syntax is very similar to Ruby and most programmers find it easy to understand. The hardest part is not the language itself, but wrapping one's head around the concepts of functional programming.

The basic syntax for Elixir looks like this:

```
defmodule Fib do
  def fib(0) do 0 end
  def fib(1) do 1 end
  def fib(n) do fib(n-1) + fib(n-2) end
end
```

What is React

React is the "V" in "MVC" (Model View Controller). It is all JavaScript and it has quickly become the new standard for frontends. Rather than using a markup language like HTML to render the DOM, React creates each element in the DOM with a function and creates a "virtual DOM" which it can use to compare against the current version of the DOM and render only the smallest necessary amount of information. If this sounds complicated, don't worry. React does all of this behind the scenes, so all you have to do is write JavaScript.

This roundabout approach might seem like overkill, but it leads to more efficient rendering, better code maintainability, and the advantage of being able to write everything in a syntax that is more coherent than a markup language: JavaScript.

React has become immensely popular, it is supported by Facebook, and it's already used in production in countless major websites and apps.

The basic syntax for JavaScript is:

```
function fibonacci(n) {
   if (n < 2) {
     return n
   } else {
     return fibonacci(n - 2) + fibonacci(n - 1)
   }
}
```

What will be covered?

This set of lessons is intended for people with at least some experience programming. An ideal candidate for this series is someone who just finished a coding bootcamp and wants to start learning how to write production-ready code. If you're a Ruby on Rails developer, or if you have experience with another framework, you should have no problem picking this up.

It would be ideal for someone with experience in functional programming, but this is not necessary; we will go through the fundamental concepts of functional programming.

We will cover some of the basics of Elixir and we will touch on Erlang when necessary. For more detailed tutorials on Elixir, check out ElixirSips.com and ElixirSchool, which will teach you a lot more about the language.

On the frontend, we will cover the basics of React and Redux, but a deeper knowledge of React will be

very useful. The egghead.io tutorials on React, Redux, and React Native are really good if you're looking for additional resources.

We will cover some of the basics of styling and we will use Sass as our preprocessor. We will also be using CSS Modules to make our CSS more maintainable and modular.

This is not an intro-level tutorial. We will not cover the basics of programming. You do not need to be an expert, but you should at least know what a `function` is and generally how to use them in at least one language. You should also know what a `library` (Lodash, Bootstrap, etc) is and have at least some idea of how to use one.

If you are totally new to programming, start with something like CodeSchool. HTML and CSS is probably the easiest place to start, followed by JavaScript and Ruby on Rails. Once you have the basics down, come back and sign up.

What is the ideal type of app for this framework?

Phoenix is ideal for highly concurrent apps, meaning, apps that have a lot of things happening at the same time. For example, a popular messaging app might have a few million people using it at any given time, and each of those users will expect to receive real-time updates when a message is sent or received.

This is not to say that Phoenix does not perform well for apps that do not demand concurrency. Its fault-tolerance alone makes it ideal for most environments. You lose very little in the way of developer efficiency for a final product that is significantly better.

That said, for a small hackathon-type app, Phoenix is probably overkill. A self-contained JavaScript framework such as Meteor.js or Feather.js are hard to beat for this type of app.

There are also languages other than Elixir/Erlang that are better suited for large-scale computation. So if you foresee a lot of number crunching or machine learning in your future, you might want to consider Java, Python, or R.

What are some alternatives?

You can use Phoenix for both the frontend and the backend using Elixir's builtin markup language. If you choose to go that route, check out the book Programming Phoenix.

If you're looking to make a quick prototype and you're proficient in JavaScript Meteor.js and Feathers.js are excellent options.

Ruby on Rails is also a safe bet and is probably the default choice for the majority of startups as of 2016. There are a nearly-endless number of resources Rails. A good place to start is CodeSchool if you're new

to the framework.

There are hundreds, if not thousands of options. If you need something more specific than the options listed above, you probably already know what you need and this section has not been especially useful to you.

General formatting

Each lesson takes you through the process of building an app with varying levels of complexity. We will also generally use different technologies for each app as the landscape evolves. For example, this app will use React and Redux, while a future app may use Cycle.js, React Native, or Elm.

In the event you have a bug to report, please post an issue in the Github repository. The Node ecosystem in particular moves very quickly and it is likely some pieces of this tutorial will become outdated in a matter of weeks. If the question is more generally about Phoenix, React, Elixir, et al, please post the question on StackOverflow.

Code written in codeblocks, such as the code below, shows you the code being added, removed, or changed. Ellipses (`...`) are often used in place of code that is being omitted for reasons of brevity.

```
class LessonShow extends React.Component {

  ...

}
```

Command line inputs will look like the example code below. The command line inputs assume you are using a Mac. If you're on a Linux machine, you probably already know what you're doing and it will be similar.

```
$ cd ~/directory/name
```

Contact

Feel free to contact us at any time with comments or questions at info@learnphoenix.io.

A special thanks to Josh Adams, Sean Callan, and Ryan Swapp for all your help in making this possible.

Installing Phoenix and React

- Install Phoenix
- Node and npm
- Brew
- Postgres

To use Phoenix and React, you need to install a few things on your computer. There are a surprising number of things that can go wrong with the installation, usually involving the installation of PostgreSQL.

Install Homebrew

The first thing you should get is Homebrew. Homebrew allows you to run the command `brew` to install just about anything on your Mac. Run the following command if you do not already have Homebrew installed. It might require you to install xcode or some extra software. If it prompts you to do so, install that too:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You should check to make sure the installation ran properly by running the command `brew help`, which should pop up with some helper text. If you get an error, something went wrong and you should do some Googling to figure out what went wrong.

Install Node.js

Now that you have Homebrew, it's pretty easy to install everything else. You will need Node to run your JavaScript server and to have access to the Node Package Manager (NPM), which you will use to install a variety of assets, including React and Redux.

Run the following command to install Node:

```
$ brew install node
```

Then, for good measure, check to make sure the installation was successful by running `node -v`, which

will return the version of Node you are using. You should also check to make sure that NPM was installed by running `npm -v`.

Install Elixir

We are assuming you're using a Mac, so the instructions below will be for a Mac installation. If you have a different machine, follow this link to find instructions.

If you're on a Mac, run:

```
$ brew install elixir
```

And you're done. To ensure the installation went smoothly, run `iex` to see if the interactive Elixir shell worked.

If it did, and this is your first installation of Elixir, you will need to install the Hex package manager (the Elixir equivalent to NPM). You can install it by running `mix local.hex`

Install Phoenix

Now it's time to install Phoenix. Run the following command to install:

```
$ mix archive.install https://github.com/phoenixframework/archives/raw/master/phoenix_new.e
```

Install PostgreSQL

We will use PostgreSQL (often referred to as "Postgres") as our default database. A lot of things can go wrong here, so you might have to do some debugging to make sure you can get Posgres to work on your machine.

Install Postgres by running the command:

```
$ brew install postgresql
```

Postgres is now installed on your machine, but it is not running. You'll want to start Postgres at login, so run the following two commands to both set Postgres to launch at login and launch now:

```
$ ln -sfv /usr/local/opt/postgresql/*.plist ~/Library/LaunchAgents
$ launchctl load ~/Library/LaunchAgents/homebrew.mxcl.postgresql.plist
```

To make sure everything installed properly, run the command `which psql`, which should return `/usr/local/bin/psql`. If it does not, something went wrong and you're in for some debugging.

Now you need to create a default database, which should be the same as your username. If you do not know your username, you can find it by running the command `whoami`. Create the database with the following command, with your username in place of `<username>`:

```
$ createdb `<username>`
```

Then run `psql` to make sure the database was created properly. If you enter the Postgres shell, you're in business!

Postgres errors

If you already have a previous version of Postgres installed, you will probably run into some problems.

```
$ pg_ctl -D /usr/local/var/postgres -l logfile start
```

Another potential problem that comes up regularly is the lack of a "role". If you get the following error, you will need to configure a role:

```
** (Mix) The database for Firestorm.Repo couldnt be created, reason given: psql: FATAL:  ro
```

You can configure a role by entering the `psql` shell and creating the role. Run the following command:

```
$ psql postgres
```

This will bring up a new shell that gives you the ability to run commands. You should enter the following to create the new role (note that `postgres=#` is the prompt and should not be entered):

```
postgres=# CREATE ROLE postgres LOGIN CREATEDB;
```

You should receive `CREATE ROLE` as the response. Then exit the shell with `\q` and you should be in business!

If you're *still* having issues, try restarting Postgres with the following commands.

```
$ pg_ctl -D /usr/local/var/postgres stop -s -m fast
$ pg_ctl -D /usr/local/var/postgres -l /usr/local/var/postgres/server.log start
```

And if you still can't get it working, try Stackoverflow or a Google search.

Additional Installations

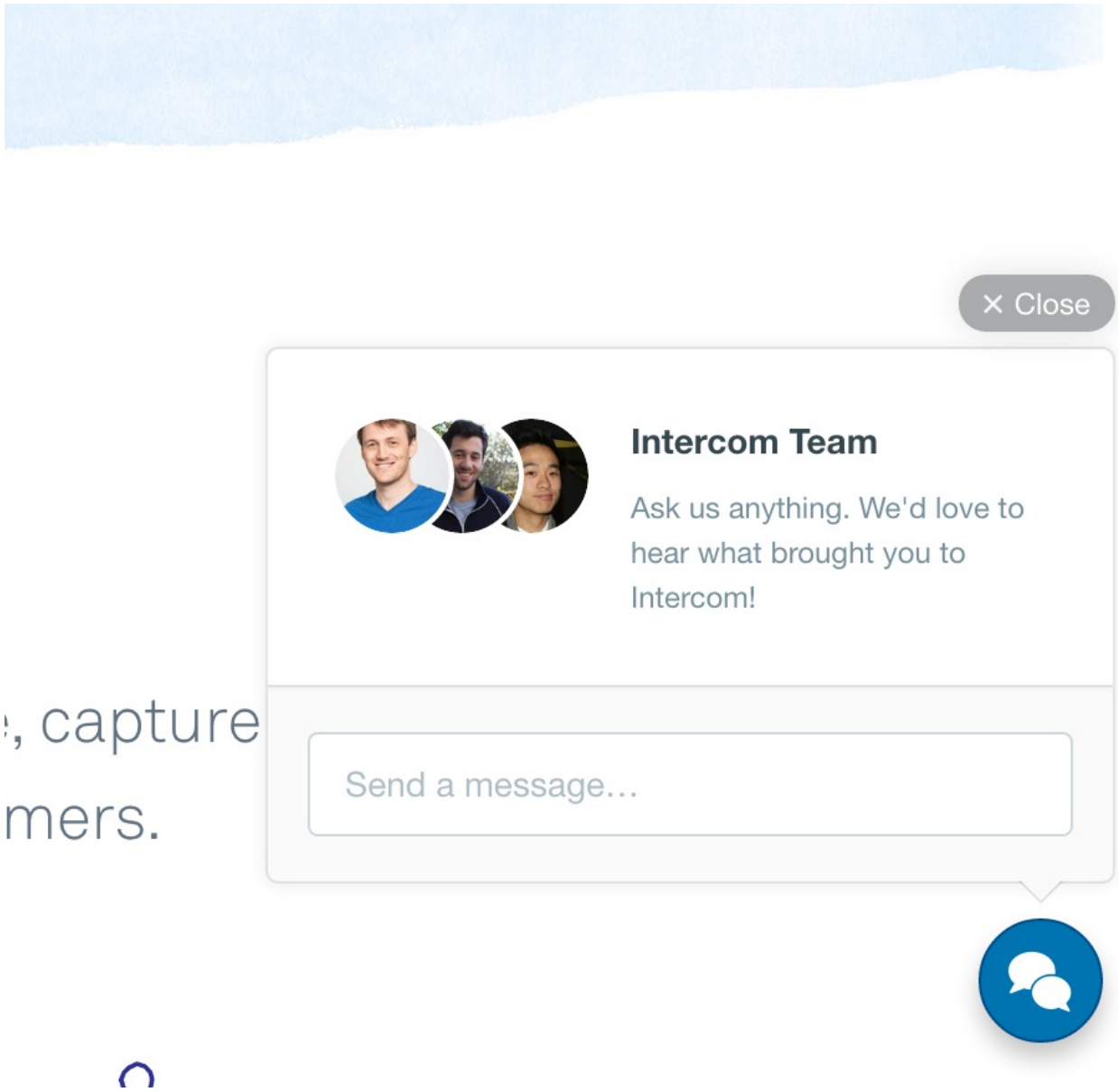
You will also need a text editor of some sort. You can use Sublime Text 3, Atom, or any other text editor you'd like (or if you're a ninja, you can use vim). Some people also prefer an IDE (integrated development environment) such as Webstorm. At the end of the day, this is all a matter of preference and will make little difference.

Directory structure

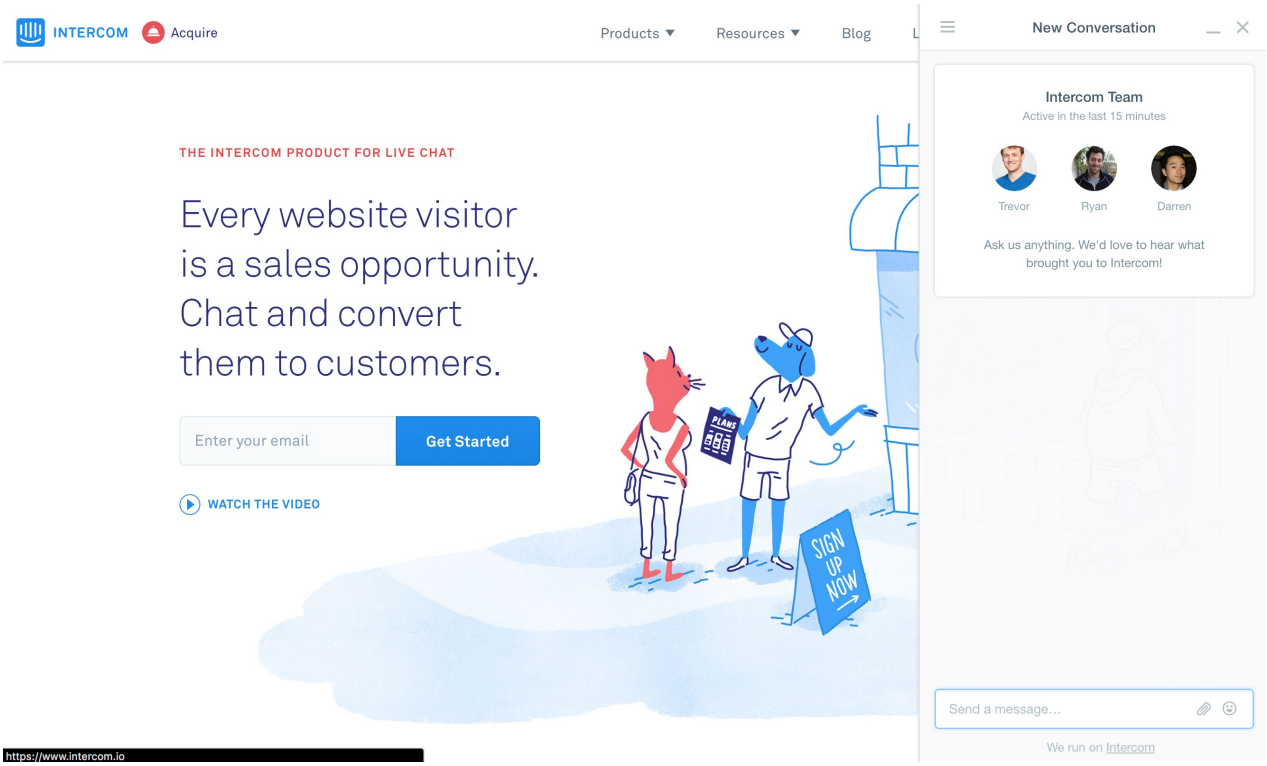
- Create a new Phoenix project
- Create a new React project
- Overview of directory structure

This app is separated into two pieces: the `frontend` and the `backend`. This separation should be familiar to you, as it is the general structure of just about every app. The frontend will be built with React, and the backend with Phoenix.

The first thing we're going to do is create a directory (also known as a "folder") with the name of our app. The app we are building is a clone of the Acquire functionality of Intercom.io. You've probably seen it before--it's the little chat bubble on the bottom right of many websites that allows you to chat with a representative of the company:



When the chat bubble is clicked, a chat window is overlayed on the page:



We will also want to build a web and mobile app that allows the company representative to easily manage several chats at once. We will model this chat feature on Slack and WhatsApp's web app.

Since we are not especially creative when it comes to naming our projects, we'll call this app PhoenixChat and we will keep a live, production-ready version of the completed app at PhoenixChat.io.

```
$ mkdir phoenix-chat-api phoenix-chat-frontend
```

Now that we have directories to contain our project, we need to generate the code for our React frontend and Phoenix backend.

Generating the frontend

Now we need to create our React app. Let's create the folder `frontend`, then enter it and create a Node app with `npm init`, filling out the command line prompts as necessary (you can also use the `-y` flag to accept all defaults if you're feeling lazy):

```
$ cd phoenix-chat-frontend && npm init
```

That's all you need to do for React at the moment. We will go over React in more detail in the next section.

Frontend directory structure

We are not going to set up the directory structure in advance of the app, but the general structure will be (and for those of you who want to know how the tree below was generated, check out tree):

```
phoenix-chat-frontend
  |-- node_modules
  |-- app
      |-- components
      |-- images
      |-- redux
      |-- styles
```

`frontend` will contain everything for the frontend of our application. This includes code for our development environment, as well as our production code. This includes our webpack configuration, our `package.json` file, and everything else we'll need to compile our app into something we can run in production.

`app` contains just the code that will become our app. This includes our components, our styles, and how all those components fit together.

`components` will contain all of our React components. This is what renders on the screen and displays our data.

`images` will contain our static images.

`redux` will contain our `actions`, our `reducers`, and our `store`. These are the core components of Redux and will be covered in a later chapter.

`styles` will contain default styling, such as a css reset or other css libraries we might need import.

Overall, the React structure is pretty straight-forward.

Generating the backend

Fortunately, Phoenix makes it easy to generate boilerplate code. We can use `mix phoenix.new` to create a simple app. We are creating our Phoenix app without Brunch because we are not using Phoenix for our frontend. When you run `mix phoenix.new`, you will be prompted to fetch and install dependencies. You want to enter `y` to install your dependencies.

Run the following commands to go back a directory, and create your new app.

```
$ cd ..
$ mix phoenix.new phoenix-chat-api --app phoenix_chat \
  --module PhoenixChat --no-brunch
```

The first command sends you back a directory (*we may not include directory changes in the future*), then the second command creates your new project within the `phoenix-chat-api` directory we made earlier.

You probably noticed `mix`, which is a command line tool that we will use extensively with Phoenix.

We then choose the `phoenix-chat-api` directory and create a new application called "PhoenixChat". We are passing in the `--no-brunch` flag because we do not need to render HTML or generate all the boilerplate for a frontend since we are building our frontend with React.

Backend directory structure

The Phoenix generator has taken care of most of our boilerplate for us. The general structure is as follows (with some directories omitted for the sake of brevity):

```
phoenix-chat-api
  |-- config
  |-- deps
  |-- lib
  |   |-- phoenix_chat
  |-- priv
  |   |-- repo
  |   |-- static
  |-- test
  |-- web
      |-- channels
      |-- controllers
      |-- models
      |-- static
      |-- templates
      |-- views
```

`config` contains several files related to configuration. Among the most useful are `prod.secret.exs`, which can hold all your secret production keys with less risk.

`deps` contains all the dependencies you installed.

`lib` contains configuration for your endpoints and your Ecto repositories (don't worry about what that means, we will cover it later).

`priv` contains all of your data migrations (i.e. a record of all your database changes), and your static files.

`test` contains all your Elixir tests.

`web` contains everything related to the web API, including `channels`, `controllers`, `models`, and other useful goodies.

If you do not know what all of this means, do not be concerned. We will go into each of these in greater detail in later chapters as they become relevant. For now, just know that Phoenix has generated most of what we need to get started. Since we are creating a web API, most of what we are interested in lives in the `web` directory.

Basics of React

- Basics of React
- Basics of Webpack
- npm and SemVer

In this section, we go over the basics npm and React and some of its syntactical quirks and install some necessary dependencies to be able to handle ES2015, 2016, and 2017 syntax, as well as jsx.

What is React?

React is a JavaScript library built by Facebook for highly efficient DOM rendering and a more functional programming approach to the frontend. From the React docs:

> We built React to solve one problem: building large applications with data that changes over time.

Rather than use a markup language such as HTML, each React component is a function, so you can build the elements within your DOM using a much more powerful language: `JavaScript`.

And because each component is encapsulated in a function, you can reuse code, run tests, and build better abstractions much more easily.

There are a few React competitors that are formidable and arguably better. Among them are Cycle.js, Elm, and Om. Each has its advantages.

The reason we are using React is because it has a much larger community, it is easier to hire developers who know React, and it is supported by Facebook, which makes it more likely to have long-term staying power.

Since `React v0.14.0`, the main `React` function has been split into `React` and `ReactDOM`; `ReactDOM` is what we use to render our components to the DOM and `React` is what we use to create our components.

Most people use `React.createClass` to create components and `ReactDOM.render` to render our components to the DOM. Most people also use the JSX syntax, which mimics HTML's syntax. A very simple React app looks like this:

```
import React from "react"
import ReactDOM from "react-dom"

const App = React.createClass({
  render() {
    return (<div>Hello World!</div>)
  }
})

ReactDOM.render( <App />, document.body )
```

There are those who prefer to use a different syntax for creating React components. That syntax extends the `Component` export from `React`. Both are perfectly valid.

We will be using the `class` syntax, as it is the syntax that Facebook suggests going forward. For a good article that explains the differences between the two, check out this article by Todd Motto. An example of this syntax would be:

```
import React from "react"
import ReactDOM from "react-dom"

class App extends React.Component {
  render() {
    return (<div>Hello World!</div>)
  }
}

ReactDOM.render( <App />, document.body )
```

Another option (though slightly different), and one that we will be using on occasion, is to create your React components as a pure function, also referred to as a "stateless function", or a "presentational component".

```
import React from "react"
import ReactDOM from "react-dom"

const App = () => {
  return (<div>Hello World!</div>)
}

ReactDOM.render( <App />, document.body )
```

Pure functions are highly performant with React, and since we are using Redux to handle state, many of our components will be pure functions. This is the case because our data is all passed down from the

highest-level component, so none of our child components need to worry about their local state. We will cover these concepts in greater detail in the Redux section.

Also worth noting, if you want to pass `props` into one of these functions, you must pass it in through the function: `const App = (props) => { ...`

This should look familiar to people who have used the `PureRenderMixin`. Both rely on pure functions and give a nice performance boost. Just keep in mind that mixins are not supported by ES6 classes and are slowly being phased out.

When using Webpack, you will want to `export` these components so you can `import` them in other files. There are two options when it comes to exporting: default exports and named exports.

Each file can only have one `default` export, and we will primarily use `default` export.

```
import React from "react"
import ReactDOM from "react-dom"

class App extends React.Component {
  render() {
    return (<div>Hello World!</div>)
  }
}

export default App
```

If you then wanted access to this component in a different file, you would simply import it the same way you imported `React` and `ReactDOM`, using the path to the file:

```
import React from "react"
import ReactDOM from "react-dom"
import App from "path/to"
...
```

So if your component lived within `app/components`, you would import it from that directory.

Named exports are slightly different, and you can export several named exports in a single file. The syntax for a named export uses curly braces for the import to pull out a specifically named pieces from within a module: `import { Component } from 'react'`. You can export functions, objects, classes, and other such things. For more information on exporting modules, check out this link.

You will also see examples where we don't name the actual file being imported and simple leave it at the directory name. This is only possible if you name the file `index.js` with a default export. Webpack will assume that the `index.js` file is the file you were looking for.

For example, if the path were `app/Home/index.js`, you could simply import from `app/Home` and Webpack will assume that you meant to import from `index.js`.

What is Webpack?

Webpack is a module bundler. If you've ever used Browserify and/or RequireJS, the concept is very similar but Webpack allows for much greater functionality.

This added functionality comes at the cost of simplicity. Finding oneself stuck in Webpack configuration hell is a not-so-uncommon experience for developers who have used it. That said, the advantages of webpack outweigh the pain of added upfront configuration.

Webpack is the technology that gives us the ability to split our app into modules that we can piece together to make our frontent efficient and well organized.

It also gives us a bunch of loaders which allows Webpack to replace your traditional build tool, such as Grunt or Gulp. We will go over all of this in greater detail as it becomes relevant, so if this is at all confusing, just know that Webpack is awesome, almost everyone uses it for their React projects, and we will be using it for this app.

Webpack is also where even the most advanced developer can find himself stuck in Webpack configuration hell for hours on end. So if you find yourself stuck on Webpack, know that you are not alone.

Setting up our frontend

We're going to change a few things in our newly-generated `package.json` file. We're going to add a `start` command so we can start our server with additional configuration using the command `npm start`. We can do this by changing the `scripts` in our `package.json` file:

```
/package.json
commit: coming soon
```

```
  "scripts": {
    "start": "node server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

What we are doing here is telling Node to use the `server.js` file, which we will create shortly, to start our server. This allows us to keep all of our basic server configuration in one file and is common practice.

We are going to leave `test` alone for now. At the moment, if you run the command `npm test`, you will simply receive a message letting you know that it has not been set up yet.

Something that is worth noting about these scripts is that there are some commands that NPM handles by default. These include `npm start`, `npm test`, and a few others. If you want to run a custom script that is not supported by default (which we will do later on), you must use `npm run <name of script>`, with the name of your script in the appropriate field. You will see this in action later.

The next thing we want to do is set up a few dependencies so we can quickly get this project up and running. These dependencies are split into `devDependencies` and `dependencies`.

Development dependencies (`devDependencies`) will not be run in production, so this is where you put things like your module bundler (Webpack) and transpilers (Babel). Basically, anything that is processed before going into production goes here.

Your production dependencies (`dependencies`) are anything that you need to run your app after compilation. React is an example of a dependency that you will need when your app is live, because your app depends on React to render the frontend.

Add the following dependencies, each of which will be elaborated upon after the codeblock (note that a `\` denotes a line break in your terminal):

```
$ npm install --save-dev babel-core babel-loader \
  babel-preset-es2015 babel-preset-react babel-preset-stage-0
```

When adding packages to your app, you have the option of adding them to the `package.json` file and running `npm install`, which will install the dependencies you added, or running the command in your terminal.

It is also important to use the `--save` or `--save-dev` flag. If you do not, your app will run locally but when someone clones your repository or when you try to deploy, your app will not work because it will not know which dependencies to install.

After running the command above, you should see the following dependencies in your `package.json` file automatically added (though the version numbers will likely be different since these packages are updated constantly). You can also specify version numbers with the `@` syntax: `npm install --save-dev babel-core@6.7.6`.

SemVer

Another thing that sometimes confuses people is the caret (`^`) that comes before the version number. This is for use in packages that abide by (as all packages should) SemVer, which stands for "semantic versioning". SemVer is very important for maintaining project stability and it is definitely worth checking

out the docs to learn more about it.

The short version is that versioning should follow a pattern, and that the pattern actually means something. For example, `X.Y.Z` denotes `X = major version`, `Y = minor version`, `Z = patch`. Any changes in `X` will likely break your app, changes in `Y` will add new features but shouldn't break anything, and `Z` is for small patches and bug fixes that won't break anything.

So coming back to the caret (`^`) and using the example `^6.14.0`, this tells npm that we want to use any package that is major version `6` and any minor version that is at least `14` (i.e. `6.15.2` would be fine). You might also see a tilde (`~`), which denotes a minor version. So for the same version above, it would tell npm to use anything with major version `6`, minor version `14`, and any acceptable patch (i.e. `6.14.3` would be fine). You can also use `x` in place of the version you want updated (i.e. `6.x.x` is equal to `^6.14.0`).

```
...
"devDependencies": {
  "babel-core": "^6.14.0",
  "babel-loader": "^6.2.5",
  "babel-preset-es2015": "^6.14.0",
  "babel-preset-react": "^6.11.1",
  "babel-preset-stage-0": "^6.5.0"
}
```

The first dependency we add is `babel-core`. This is something you have to add to use `babel-loader`, which is really what we want. Why the two are separated, I have no idea, but I'm sure there is a perfectly good reason.

The second dependency is `babel-loader`. This is probably a good time to go over what loaders are and why they are a very useful part of Webpack.

Loaders

Loaders allow you to take code of a particular type and apply transformations to it. Or as it says in the Webpack docs:

> Loaders allow you to preprocess files as you require() or "load" them. Loaders are kind of like "tasks" are in other build tools, and provide a powerful way to handle frontend build steps. Loaders can transform files from a different language like, CoffeeScript to JavaScript, or inline images as data URLs. Loaders even allow you to do things like require() css files right in your JavaScript!

One cool feature of loaders is that you can chain them to apply multiple transformations. For example, if you had a `.less` that had styles you wanted to apply to your component, you could run the `less-loader`,

followed by the `css-loader` (because the `less-loader` transformed it into css), followed by the `style-loader`, which loads the style into your React component:

```
require("style-loader!css-loader!less-loader!./styles.less")
```

We'll cover the specifics of each loader we use as we progress through this course.

Back to setting up the frontend

The next three dependencies are `babel-preset-react`, `babel-preset-es2015`, and `babel-preset-stage-0`. These give you access to all the Babel plugins under that category. So, when you add the `babel-preset-react` plugin, you get access to the plugins:

- `react-constant-elements`
- `react-display-name`
- `react-inline-elements`
- `react-jsx`
- `react-jsx-compat`

The last dependency we add is `babel-preset-stage-0`, which tells Babel to use the most recent experimental version of Babel. The stages range from 0 to 4, with 0 being the least stable and 4 being the most stable. We are going to use `stage-0` because it gives us access to a lot of handy features that are not yet implemented in ES6, such as destructuring assignment and spread operators. We will explain what each feature is in more detail as we use them.

As of Babel 6, you need to be more explicit about what Babel uses. To do this, you have two options: create a `.babelrc` file, or add a `babel` configuration option within `package.json`. This is entirely a matter of preference, but since in this case we don't have many options to configure, let's just add it to `package.json`:

```
/package.json
commit: coming soon
```

```
  ...
  "devDependencies": {
    ...
  },
  "babel": {
    "presets": ["es2015", "react", "stage-0"]
  }
```

Then let's go ahead and add Webpack as a development dependency. We will add both Webpack and the the Webpack development server so we don't have to continually refresh the page to see the changes we made:

```
$ npm install --save-dev webpack webpack-dev-server
```

The last dependency we need before we can render "Hello World!" to our DOM is React. Let's go ahead and add React as a dependency for our app. Recall that this is not a development dependency, but a production dependency. Since we're building this app into static assets, it actually doesn't matter if you install things as `devDependencies` or `dependencies`, but it's generally advisable to keep modules that are exclusively for testing etc separate from things that your app actually depends on in runtime.

```
$ npm install --save react react-dom
```

In version 0.14.0, React split into two main pieces: `React` and `ReactDOM`. ReactDOM handles everything DOM-related, such as rendering to the DOM, while React itself handles the creation of all of your components. If this does not makes sense now, it will become immediately obvious as soon as we start using them.

In the 15.0.0 update, they changed their versioning to `major` versioning in accordance with SemVer mostly because corporate people were afraid to use software that "isn't even 1.0 yet". There are few, if any, breaking changes that will affect most applications.

Basics of Webpack

- Set up Webpack
- Hello World!

In this section, we set up Webpack and get our `Hello World!` app running.

Webpack configuration

Now that we have all of our NPM dependencies set up, we need to configure Webpack. As mentioned before, this is not the simplest task. There are a seemingly limitless number of configuration options, but for now, we will only use the minimum.

Create a file using the `touch` command in your `/phoenix-chat-frontend` directory called `webpack.config.js`.

```
$ touch webpack.config.js
```

In that file, add the following configuration options, which we will go over line-by-line below the code:

```
/webpack.config.js
commit: coming soon
```

```
var path = require('path')
var webpack = require('webpack')

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    './app/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        loaders: ['babel'],
        exclude: /node_modules/,
        include: path.join(__dirname, 'app')
      }
    ]
  },
  resolve: {
    extensions: [ '', '.js' ]
  }
}
```

The first line requires the Node's `path`, which is used to keep track of directory names. This is not strictly necessary, but makes directory paths clearer.

The second line requires `webpack`. We need this because we are going to use some of the plugins from Webpack later on in the configuration.

Next we have our `devtool`, which we set to `eval`. This just sets your debugging tool to one of several options, but almost everyone uses `eval` because it builds faster. If you want better bug tracking, you should use `source-map` or `inline-source-map`.

The next configuration option is `entry`. Entry is an important concept in Webpack because it defines all the modules that are to be loaded when you start the app, with the last one being exported. This is an important piece to understand, so we will go into greater detail.

The first entry point is the `webpack-dev-server`, which we want to run our app through first. Next, we run it through the Webpack hot loader, which hot-loads our code and allows the changes you make to take effect immediately without losing the state of your app. The last thing is `app/index`. This will finally export whatever we eventually put in the `index.js` file, which we will create shortly.

You might be wondering at this point why we left off the `.js` part of `index.js` in our configuration file. That's because of the `resolve` configuration option, which allows us to leave off the file extensions of any named file type--in this case, `.js`.

The next configuration option is `output`. This is another important part of webpack to be aware of. Because React is all JavaScript, Webpack is able to bundle everything into a single file (though not necessarily one file, as we will see much later in this course) and insert it into a script tag in your app.

In our case, we are cleverly naming our bundle, `bundle.js`. We are also specifying the location of all static assets to be compiled into `dist/`. If you want to see webpack in action or look at what would be built in production, you can run `webpack` from the command line and it will compile your app and drop it in `phoenix-chat-frontend/dist`.

The next line is our list of `loaders`. By the time this project is through, we will have several additional loaders for different types of files. This is where you tell Webpack how to handle different file types. You can handle images (`.png`, etc), styling (`.less`, `.scss`, etc), and just about any other type of file imaginable--even file extensions that you set arbitrarily, such as `.potato`. Loaders are among the most useful features of webpack, and they are the easiest to understand.

Also within our `loaders`, we are telling it to ignore any files within our `node_modules` directory, because those files are already compiled, and we telling it to explicitly look for files within our `app` directory.

We now have a functional (if basic) webpack configuration.

Server configuration

The last piece of configuration we need is to configure our server. Recall we changed our `npm start` script to call a `server.js` file so we could add some additional configuration. So the first thing we need to do is create the `server.js` file in our `/phoenix-chat-frontend` directory.

```
$ touch server.js
```

In that file, add the following configuration:

```
/server.js
commit: coming soon
```

```
var webpack = require('webpack')
var WebpackDevServer = require('webpack-dev-server')
var config = require('./webpack.config')

new WebpackDevServer(webpack(config), {
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, res) {
  err ? console.log(err) : console.log("Listening at localhost:3000")
})
```

First we declare our dependencies for this module (`webpack`, `webpack-dev-server`, and `webpack.config`), then we create a new `WebpackDevServer`, with some of the options from our Webpack configuration and tell it to listen on `port 3000`.

The `historyApiFallback` is necessary for handling route changes. We will start out with something called `hashHistory` (explained in detail later), which handles all routing on the client with no server-side configuration necessary. This flag makes routing possible for single page apps. If you want to dig deeper into why this is, check out this resource.

If there is an error, it tells us there was an error. Otherwise, it logs `Listening at localhost:3000`.

HTML, React, and ReactDOM

There are only two remaining additions we need to make to get our "Hello World!" app running. First we need an `index.html` in our root directory, then we need to create our `app/index.js` file to render to the page.

Go ahead and create an `index.html` file in the root directory and add the following.

```
$ touch index.html
```

```
/index.html
commit: coming soon
```

```
<html>
  <head>
    <title>PhoenixChat.io</title>
    <meta name="description" content="Free, easy-to-use React component for real-time chat
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta charset="UTF-8">
  </head>
  <body>
    <div id='root'></div>
    <!-- Webpack Bundle -->
    <script src="bundle.js"></script>
  </body>
</html>
```

Google actually doesn't care about most meta tags, but it does care about the `description` tag.

The next `meta` tag is the `viewport` tag, which makes it possible to resize your app based on the size of the screen that is viewing it--namely, this is what makes viewing your site on a mobile device possible. Pretty much every app should have this tag.

We are also including a `meta` tag that sets the `charset` to `UTF-8`. This is not usually necessary since most browsers will assume this is the case. This allows you to use lots of new characters and symbols. For more on this, check out UTF-8.

Next is the `body`, in which we are creating a `div` with an `id` of `root`. It is within this `div` that we will create our entire app. Though it is possible to render your app directly to `document.body`, it is bad practice and should be avoided. If this point is confusing, it will become clear once we fill out our `index.js` file.

The last thing we add is a `script` tag that links to `bundle.js`. This is where Webpack puts all of our React code to allow us to render our app. You may recall in our Webpack configuration we set our `output` to have a filename of `bundle.js` and a path of `dist`. This is where we reference that configuration option.

Let's create a few basic files that we'll use later while building the app. First, create an `app` directory within `/phoenix-chat-frontend`, and then a file called `index.js` within app:

```
$ mkdir app
$ touch app/index.js
```

Now, within our `app/index.js` file, we want to write the code that will render our app. First, let's create a React component that will render the text "Hello World!" within a `div`.

```
/app/index.js
commit: coming soon
```

```
import React from "react"

class HelloWorld extends React.Component {
  render() {
    return (<div>Hello World!</div>)
  }
}
```

The next thing we need to do is render that component to the DOM. For that, we need `react-dom`. Chance your code to the following to tell ReactDOM to render your component under the `div` we created in our `index.html` file with an `id` of `root`.

```
/app/index.js
commit: coming soon
```

```
import React from "react"
import ReactDOM from "react-dom"

class HelloWorld extends React.Component {
  render() {
    return (<div>Hello World!</div>)
  }
}

ReactDOM.render(
  <HelloWorld />,
  document.getElementById("root")
)
```

Hello World!

We're finally done with configuration! Now it's time to start the app. Within the `phoenix-chat-frontend/` directory, run the following commands to install all your `npm` dependencies and start your app (using the script we created at the beginning of this lesson):

```
$ npm install && npm start
```

You should now have an app that renders "Hello World!" to the screen like this

coming soon

Additional

There are a few optional configurations you can do that will save you some headache down the road. If you're using git for version tracking, you should probably add a `.gitignore` to make sure you don't commit your `node_modules`. Within a `.gitignore` file in the root of your app, simply add the line `node_modules` and git will not track anything in that folder.

If you're on a Mac, you should probably add `.DS_Store` to this list as well, because they're annoying. And while you're at it, add `npm-debug.log` incase you try to run `npm start` from the wrong directory, or cause any other errors with npm. And since everything in `dist` will be created from the contents of `app`, we can safely ignore its contents.

```
$ touch .gitignore
```

```
/.gitignore
commit: coming soon
```

```
node_modules
.DS_Store
npm-debug.log
dist
```

A few additional notes regarding common practices. JavaScript generally uses `camelCase`, Elixir uses `PascalCase`, and erlang uses `under_scores`. You will occasionally use erlang code within Phoenix.

Routes and views

- React components
- React router
- Hash history

In this section, we set up our router using `react-router`, which has become the default router for React. We're also going to create a few basic views so that our router has something to display.

We're also going to go some of the very basics of how routers work and the various routing options. For the time being, we're going to stick to a hash router (explained below).

Components

The first thing we should do is create a few components for our router to render. For now, let's create a `Home` component for our root route (`/`) and a `Settings` component for our user profile and settings (`/settings`).

Because React allows us to reuse components, we should be as explicit as possible with our component names. At some point in the future, we will have a `Message` component that will be used once for every message in a chat between customer and representative, and may even be shared across applications.

For the sake of clarity, we will avoid using plural names whenever possible. `Comment` vs. `Comments` will always lead to confusion down the road. As we add more components, it will become abundantly clear why this is necessary.

We will want to keep all of these organized in a `components/` directory under `app/`. Let's create that now along with directories and files for each of our new components. *Remember, directory changes may not be included in the remainder of this course for the sake of brevity.*

```
$ mkdir -p app/components/{Home,Settings}
$ touch app/components/{Home/index.js,Settings/index.js}
```

Note the `-p` flag in our `mkdir` command. This tells bash that if the directory does not exist, go ahead and make it. If we did not use this, we would have to run `mkdir app/components` first.

And while we're at it, let's create some generic content in each of those `index.js` files we just created.

/app/components/Home/index.js
commit: coming soon

```
import React from "react"

export class Home extends React.Component {
  render() {
    return (
      <div>Home component</div>
    )
  }
}

export default Home
```

/app/components/Settings/index.js
commit: coming soon

```
import React from "react"

export class Settings extends React.Component {
  render() {
    return (
      <div>Settings component</div>
    )
  }
}

export default Settings
```

React Router

We are going to use the popular React Router for our routing. If you are not familiar with what routing is, it's the thing that handles your url path, such as `http://github.com/learnphoenix`, and directs you to the proper view. Without a router, your app will be limited to a single page (this is not strictly true as there are ways around using a router, but this is *generally* true).

Now that we have a few components set up, we should direct our router to access them. In order to use React Router, we need to add a few dependencies.

```
$ npm install --save history react-router
```

React Router requires the `history` package and does not automatically install it for you.

The next thing we need to do is change the `index.js` file within our `app` directory to use our router. Rather than use `ReactDOM` to directly render directly to the DOM, we're going to go through React Router to render the appropriate content based on the current path.

Add the following code the `index.js` file at the root of your project. Each line will be described in detail after the code block.

```
/app/index.js
commit: coming soon
```

```
import React from "react"
import ReactDOM from "react-dom"
import { Router, Route, IndexRoute, hashHistory } from "react-router"

import { default as Home } from "./components/Home"
import { default as Settings } from "./components/Settings"

const App = props => (<div>{props.children}</div>)

ReactDOM.render(
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Home} />
      <Route path="settings" component={Settings} />
    </Route>
  </Router>,
  document.getElementById("root")
)
```

The first thing we have here is a list of our dependencies for this file. `React` and `ReactDOM` should look familiar. From `react-router`, we need `Router`, `Route`, and `IndexRoute`. The `Router` is the outer element that controls our routes, `Route` is each individual route, and `IndexRoute` displays a component when you get to the `/` root route.

Also, since we're exporting both a named export (`export class Home ...`) and a default export (`export default Home`), we are specifying that we are importing the `default` import using the syntax `{ default as ... }`. This is not strictly necessary, and your app will not throw an error, but it's the recommended syntax as of July 2016.

A note on ES2016 arrow function syntax: We've also (temporarily) changed around our `App` component to show of the different ways in which you can write functions in ES2015. When writing arrow functions in ES2015:

- parentheses are optional when there is only one argument (in this case "props" is the only

argument)
- if the function is only one line, it is implicitly returned (you do not need to type out "return")
- curly braces are optional for single-line functions
- you can destructure the arguments.

In other words, all the following are equivalent:

```
const App = function (props) {
  return (
    <div>
      {props.children}
    </div>
  )
}.bind(this)

const App = (props) => {
  return (
    <div>
      {props.children}
    </div>
  )
}

const App = (props) => { return (<div>{props.children}</div>)}

const App = props => (<div>{props.children}</div>)

const App = ({ children }) => (<div>{children}</div>)
```

The `hashHistory` bit is something that React Router is using to create URLs. It does not require server-side configuration, so it is ideal for single page apps like this one. Eventually we might want to change this over to `browserHistory`, but we will hold off on that until we deal with server-side rendering and other options that require us to run our frontend through a server.

Then we import the three components we just made so they can be rendered by the router when the user hits the relevant url.

The next thing we need to do is create a an `App` component that holds all of our other components. Anything in this component will be rendered in every view, so use it sparingly. Within this component, we render all of the child components as we define in our router below.

The last thing we need to do is set up our routes. You should be able to see a pretty obvious hierarchy of routes, with the `App` component holding each of two child components(`Home`, `Settings`) each of which has its own path.

So now, if you go to `localhost:3000`, you should see your Home component. If you go to

`localhost:3000/#/settings`, you should see your settings component.

Hash history

There are a few things to note about `hashHistory`. Your routes will all have a `#` in them. So if you want to view the settings, it will be accessible at `localhost:3000/#/settings` rather than just `localhost:3000/settings`.

What's actually happening under the hood is there is an event listener that listens for a change in the url, and whenever it changes, it triggers a function. When you enter a url, your server ignores anything that comes after a `#` by default. This is because data after a hash (referred to as a `fragment`) is generally used to change things only on the client; it is often used to set scroll position.

You can try this out if you want by typing in `https://github.com/#learnphoenix`. You'll notice that what you actually see is the GitHub homepage. This is because when you make a request to a particular url (e.g. `https://github.com/learnphoenix`), you're sending a message to the server that says, "give me the data I need for `github.com` from the `/learnphoenix` route", but when you use a hash router, the server is only asked for `github.com` and the client handles the routing for everything after the hash (`#`) using `react-router`.

There is also some random stuff that gets put on the end of your url, such as `_k=vm23gw`. This is for people who are using an older browser that does not support the HTML5 push API. This is not strictly necessary and can be disabled if you are not worried about supporting old browsers.

It's actually really easy to create your own basic router. All you have to do is listen for changes in your url path and run a function that renders a new view when a particular route is hit. There's nothing magical about routers; try running `window.location` in your browser console to see what you have access to. That said, there are many additional features and edge cases that make using something like `react-router` more practical.

Set Up Styles in Webpack

- Style loaders
- CSS Modules

Now that we have a few routes set up, we should start building out our frontend components so we can render something more interesting to the user. But before we do that, we need to configure our styling.

There is a seemingly limitless number of options when it comes to styling. You have LESS, Sass, Stylus, plain-old CSS, as well as extra features such as PostCSS and CSSNext, autoprefixers, and some React-specific bits such as the `style-loader`, and `react-css-modules`.

We are going to use the following, and we'll go over why:

- `style-loader` and `css-loader`
- `react-css-modules`
- `postcss-loader`
- `postcss-cssnext`

Preprocessors (LESS, Sass, Coffeescript, etc) save countless hours of development time and allow for additional features. That said, the next CSS specs are pretty similar to what you'll get from a preprocessor and `postcss-cssnext` gives you all of those features now. You can think of it as Babel for styles. Also, CSSNext gives you a built-in `autoprefixer`, so we won't need to install one ourselves.

The `style-loader` and `css-loader` are loaders that every React project needs in order to add styling to a component. After you compile down to CSS, you need to use the `style-loader` to put that CSS into your React component, so you will always see it as the last loader in any styling configuration. For example:

```
{ test: /\.scss$/, loaders: 'style!css!sass' }
```

So, in effect, you are starting with a `.scss` file, converting it to CSS, then using the `style-loader` to add that CSS to your React component.

We are also using an `autoprefixer` so we don't have to worry about those pesky vendor prefixes. Remember, this is built-in to the latest version of `postcss-cssnext`. If you are not familiar with vendor prefixes, they are those annoying extra bits of styling you need to add to make your styles work with different browsers.

```
  -webkit- (Chrome, newer versions of Opera.)
  -moz- (Firefox)
  -o- (Old versions of Opera)
  -ms- (Internet Explorer)
```

These prefixes exist because when a new experimental CSS feature is proposed, browsers will try to implement that feature differently. The prefix tells the browser how to apply that style in a way that is compatible with the browser you are using.

The last piece we're going to add is also the most interesting. The `react-css-modules` configuration allows you create locally-scoped CSS, so you no longer need to keep a massive dictionary of globally-scoped CSS! Or as it says in the docs:

> A CSS Module is a CSS file in which all class names and animation names are scoped locally by default. All URLs (url(...)) and @imports are in module request format (./xxx and ../xxx means relative, xxx and xxx/yyy means in modules folder, i. e. in node_modules).

React CSS Modules is likely to become the new gold-standard of organizing styles, as it is much easier to create and manage styles in JavaScript than it is in CSS.

Webpack configuration

The first thing to do is to add the necessary dependencies to our `package.json` file:

```
$ npm install --save-dev style-loader css-loader \
  postcss-loader postcss-cssnext
$ npm install --save react-css-modules
```

But first, we have to set up the loader to handle CSS files. We're also going to have to do some additional configuration to get our CSS modules working properly.

```
/webpack.config.js
commit: coming soon
```

```
  module: {
    loaders: [
      ...
      {
        test: /\.css$/,
        loader: 'style!css?modules&importLoaders=1&localIdentName=[local]_[hash:base64:5]!p
        include: path.join(__dirname, 'app'),
        exclude: /node_modules/
      }
    ]
  }
```

This probably looks confusing because of the way the modules loader works. In the code above, we are effectively calling in order (*recall, it runs from right to left*) `postcss`, `css` with the `modules` option, and then `style`. The confusing bit is that `modules` has a bunch of options that you pass into it.

The first option is `importLoaders`, which you set to `1`, which simply means "true".

The second option is `localIdentName`, which you set to an expression that looks like this: `[local]_[hash:base64:5]`. This will take the name of the local style (for example `.column`), and add a randomly generated string of 5 characters at the end of it. So, in the example above, you might end up with a class of `column_3nrwE`.

Now we need to configure our `postcss` loader to use the autoprefixer. Fortunately, this is quite simple. Within our `webpack.config.js` file, add the following (**make sure you add cssnext at the top of the file**):

/webpack.config.js
commit: coming soon

```
var cssnext = require('postcss-cssnext')

  ...
  module: {
    ...
  },
  resolve: {
    ...
  },
  postcss: function () {
    return [cssnext]
  },
  ...
```

Resetting default css and globals

We're almost ready to get started making some components. The last thing we need to do is remove the default styling provided by our browser. You can do this with a global reset stylesheet (which is what we're using), or you can use a package like Normalize. It's a matter of preference and it depends on the level of customization you want.

Within our `app` directory, create a new directory called `styles` and create a new file called `reset.css`:

```
$ mkdir app/styles
$ touch app/styles/reset.css
```

Within this directory, we will create all of our shared styles. This will include things like buttons, forms and inputs, and any other styles that are shared across multiple components.

We are not going to use a frontend framework such as Bootstrap or Foundation. We are going to build all of our own styles from scratch. If this sounds intimidating, I assure you it is not as much work as you think. We are going to copy the global reset from meyerweb.com, with a few slight variations. Copy and paste the following into your `reset.css` file:

/app/styles/reset.css
commit: coming soon

```
/* http://meyerweb.com/eric/tools/css/reset/
   v2.0 | 20110126
   License: none (public domain)
*/

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    color: #333;
    padding: 0;
    border: 0;
    font-size: 100%;
    font-family: "Roboto", arial, sans-serif;
    vertical-align: baseline;
```

```
    box-sizing: border-box;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
    display: block;
}
body {
    line-height: 1;
}
ol, ul {
    list-style: none;
}
blockquote, q {
    quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
    content: '';
    content: none;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}

a {
    color: #42A5F5;
    text-decoration: none;
}
```

And to have this reset take effect, import them to the top of your `app/index.js` file:

/app/index.js
commit: coming soon

```
import "./styles/reset.css"

...
```

Remember, when you make changes to your webpack configuration, you need to restart your node server for those changes to take effect.

At this point, you probably understand the sentiment behind JavaScript Fatigue... but at least we're almost ready to start building our app!

Style a React Component

- Creating and styling the Sidebar
- CSS best practices

Creating a Sidebar component

Now that we finally a basic Webpack configuration ready, let's create a new `Sidebar` component and use our `css-modules` to style it.

```
$ mkdir app/components/Sidebar
$ touch app/components/Sidebar/{index.js,style.css}
```

Within this new component, we need to define our dependencies, create a placeholder component, then export that component wrapped in our styles.

/app/components/Sidebar/index.js
commit: coming soon

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

export const Sidebar = () => {
  return (
    <div>
      This is the sidebar
    </div>
  )
}

export default cssModules(Sidebar, style)
```

The third and fourth lines import our styles from our `styles.css` file and the `cssModules` function which allows us to export this component with its local styles. You'll see at the bottom of the file that, rather than simply exporting the `Sidebar` component, we are wrapping it in the `cssModules` function and pairing it with the styles that we imported above.

A note on relative vs. absolute paths: For the most part, we want to use absolute paths. Unfortunately, the coverage package we're going to use (Istanbul/nyc) does not sufficiently support the use of `jsdom` in favor of Karma (to be explained in a future lesson), so we will not be able to run webpack when running our tests, so for now, we will have to use relative paths.

Maintaining this consistency makes it much easier to move files around since if everything is absolute, you can merely find and replace. If on the other hand, everything is relative, you have to change the relative path all over your app, which can be a nightmare. Unfortunately, we do not have a choice at the moment, so we will use relative paths so we don't have to run our tests through webpack.

We want our `Sidebar` to look roughly like the image below:

image coming soon

We will use Flexbox for most of our alignment because it is now widely supported and much, much better than hacking away with tables. It's also the standard for React Native. If you break this `Sidebar` component down, you can see that our `Sidebar` is really a single column of fixed width which contains a list of contact cards.

image coming soon

We will eventually split these individual components into smaller pieces, but for now, we can just put them all under the single `Sidebar` component.

Let's start with something small:

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...
export const Sidebar = () => {
  return (
    <div>
      <h3>John Smith</h3>
      <p>Last active: {Math.floor((Math.random() * 10) + 1)} minutes ago.</p>
    </div>
  )
}
...
```

In order to get this to render, we have to import this component into another component. Since we're going to want to render the `Sidebar` on on the root route, we should import it into our `Home` component.

```
/app/components/Home/index.js
commit: coming soon
```

```
...
import { default as Sidebar } from "../Sidebar"

export class Home extends React.Component {
  render() {
    return (
      <div>
        <Sidebar />
        Home component
      </div>
    )
  }
}
...
```

Now when you refresh your page, you'll see your `Sidebar`.

But it hardly looks like a sidebar... So now we need to style it.

Optional formatting for styles

There are several options when it comes to styling with `react-css-modules`. One is to import your styles and reference each style as a property of the style object.

```
.container {
  background: red;
}
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

const HelloWorld = () => {
  return (
    <div className={style.container}>
      Hello World!
    </div>
  )
}
```

Another option is the `styleName` reference that `react-css-modules` gives us. This allows us to use a string rather than an object reference. Because of the way components are wrapped using `cssModules`,

you'll get better errors (compilation instead of runtime) when you mess up or forget to add a style.

```
const HelloWorld = () => {
  return (
    <div styleName="container">
      Hello World!
    </div>
  )
}
```

That said, we're going to use the `className` with the `style` object because it's currently the most widely supported.

A note on using multiple classes for a single element: Don't do it. But if you must, you can set an option within `react-css-modules` to allow you to include multiple modules for a single element. Just be aware, once you start down the road of multiple class names, you start accumulating dead code that is time consuming and tedious to refactor out of your codebase.

Styling the Sidebar component

The first thing we should do is give our `Sidebar` a few `className` attributes so we can reference them in our `style.css` file:

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...
export const Sidebar = () => {
  return (
    <div className={style.sidebar}>
      <h3>John Smith</h3>
      <p>Last active: {Math.floor((Math.random() * 10) + 1)} minutes ago.</p>
    </div>
  )
}
...
```

Then within our `style.css` file, add the classes that we will use to style our `Sidebar`:

```
/app/components/Sidebar/style.css
commit: coming soon
```

```
.sidebar {
  position: absolute;
  left: 0;
  top: 0;
  height: 100vh;
  width: 300px;
  background: #eeeeee;
  overflow-y: scroll;
  border-right: 1px solid #ccc;
}
```

Most of this styling is straightforward. If you are not comfortable with Flexbox, I recommend checking out Flexbox Froggy, which is a game that will get you up to speed in about 10 minutes.

You may have noticed that our "Home component" text disappeared. That's because it's hidden behind the absolutely positioned `Sidebar`. Let's create a wrapper in our `Home` component that will eventually contain all of our chat messages.

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

import { default as Sidebar } from "../Sidebar"

export class Home extends React.Component {
  render() {
    return (
      <div>
        <Sidebar />
        <div className={style.chatWrapper}>
          Home component
        </div>
      </div>
    )
  }
}

export default cssModules(Home, style)
```

Then let's put 300px of margin on the left so that all of our content displays on the page.

Also note that you cannot use dashes (`-`) in the names of your styles when using object notation because JavaScript will interpret them as a mathematical expression. As it turns out, camelCase is already supported by CSS, so this shouldn't cause any problems over the long-run.

```
$ touch app/components/Home/style.css
```

```
.chatWrapper {
  margin-left: 300px;
}
```

Additional

A note on nesting: Nesting your styles is generally considered bad practice. It makes refactoring harder and can lead to a lot of confusion. There are some exceptions.

Some people prefer to nest things like `:hover` or other pseudo-classes, as they could make it easier to understand the full functionality of a particular selector. This is also the case with nested `@media` queries, since it can be useful to quickly see which element is being affected by a particular media query. It's mostly a matter of preference, but the larger style guides tend to avoid nesting, so we will also avoid it.

Snippets and Aliases

- Snippets
- Aliases
- .vimrc

Sometimes there is no way around writing repetitive code. Every React component needs to have some of the same boilerplate to make it function, and there are some bash commands that you have to type over and over again. This is where snippets and aliases come into play. This section is not strictly necessary for the rest of the series, but it might save you a lot of time down the road.

Snippets

One handy feature you can do in most text editors is create a `snippet`. If you are using Atom, you can change the `snippets.cson` file (`.cson` is `.json` but for CoffeeScript) and add a text snippet.

```
$ atom ~/.atom/snippets.cson
```

There should already be a file there with instructions on how to create a snippet. These will save you a lot of time and you can create your own if you find yourself typing out the same boilerplate code, over and over again. The code below will generate boilerplate for our typical React component.

```
# path: ~/.atom/snippets.cson

'.source.js':
  'Standard component':
    'prefix': 'react'
    'body': '''
      import React from "react"
      import cssModules from "react-css-modules"
      import style from "./style.css"

      class $1 extends React.Component {
        constructor(props) {
          super(props)
        }

        render() {
          return (
            <div>
              $1 component
            </div>
          )
        }
      }

      export default cssModules($1, style)
    '''
```

If you are not familiar with CoffeeScript, know that the whitespace matters. Each tab is the same as if you nested something within curly braces. Multiline strings also start and end with three apostrophes `'''`.

The first line tells Atom that you want to target files that end in `.js`. Then we name our snippet, in this case "Standard component", but you can name it whatever you want. The `prefix` is what you type into your text editor before hitting the `tab` key. Again, you can name this whatever you want. Finally, the `body` is the code that is written when you press `tab` after typing the prefix.

A handy thing to note is the `$1` keyword. In Atom, after you initiate a snippet, you can specify where you want the cursor to end up so you can start tying. If you want to get really fancy as we have done above, you can specify multiple cursors so you can type things like the name of the component in multiple places as the same time. If you have a second place you want to jump to, you can use `$2` and press `tab` to jump to it.

The code above will generate your typical React component and will save you a ton of time over the course of this project. As you find yourself writing certain code over and over again, consider writing your own snippet.

Another snippet that will come in handy is a basic test for a React component to make sure it renders properly. The syntax will make sense once we go over testing with `enzyme` in a future lesson.

```
'.source.js':
  'Standard component':
    ...
  'Standard test':
    'prefix': 'test'
    'body': '''
      import React from 'react'
      import expect from 'expect'
      import { shallow } from 'enzyme'

      import { $1 } from './'

      describe('<$1 />', () => {
        it('should render', () => {
          const renderedComponent = shallow(
            <$1 />
          )
          expect(renderedComponent.is('div')).toEqual(true)
        })
      })
    '''
```

Aliases

Aliases let you run complex command line tasks with fewer characters. For example, instead of typing out `git add .`, you could write an alias that lets you type out something as simple as `ga` which will accomplish the same thing.

But this also comes in handy for much more complicated tasks. In this app, we will create a lot of components, and each component will have an `index.js`, `spec.js`, `style.css`, and a `README.md` file. You could type them all out one by one every time, or you could write an alias to make this easier.

If you're on a Mac, your aliases live in `~/.bash_profile`. If you do not have this file, you should go ahead and create it.

```
$ touch ~/.bash_profile
```

Here are a few simple git commands that save a couple minutes every day.

```
alias gs="git status"
alias gd="git diff --stat"
alias ga="git add --all"
```

.vimrc

Coming soon!

Unit Tests with Enzyme

- What is a unit test?
- Enzyme and shallow rendering
- jsdom and DOM rendering
- Istanbul/nyc and test coverage

React makes unit testing extremely simple. Since every component is just a function and since most of our components are pure functions, it's easy to isolate just the pieces of the component you want to test without any side effects.

We are going to use Enzyme, which is a testing utility for React that clears up some of the awkward syntax of React's `TestUtils`.

One other advantage of testing React components is that you can test them without the need to render them to a DOM using something called shallow rendering. Shallow rendering is extremely performant and lets you run through your tests at blazing speed. This is because pulling up a DOM and rendering to it takes significant time, while simply calling a function is really fast.

What is a unit test?

A unit test is the smallest possible testing unit and should only test one thing. Unit tests are different than integration tests because integration tests are in place to make sure that our pieces work together. In short, unit tests make sure our pieces work by themselves, while integration tests make sure our pieces work together.

An example of a unit tests would be taking a single React component and testing to make sure it does everything we want it to do. We might want that component to render a particular element to the DOM, or trigger a specific event. Now, the tricky part about unit tests is that we wouldn't actually care whether or not that event actually did anything--all we care about was that the event was triggered.

Functions (especially pure functions) are really easy to test because they will always give the same output given a certain input.

What to test?

We are going to test our `Home` component, which is just a wrapper. All we really need to tests is that a

`<div>` renders.

For more complex components, you want to test to make sure that everything works the way it's supposed to given certain inputs. You don't want to rely on a database call or on anything outside the specific component that you are testing.

There is also the question of whether testing is worth the time for a particular component. When it comes to unit testing in React, it's really easy to go overboard. As a rule of thumb, if the CEO is going to give a live demo, you want to make sure you have really comprehensive tests for everything he would do in that demo.

It is also worth considering how you might later refactor a component, and how your tests would ensure that the component still operates the same way after the refactor. For example, in a `Button` component, you'd want to make sure that your component rendered its `children` and any other properties that are required for that component to render properly.

Setting up Enzyme

The first thing we need to do is install `enzyme` and all other dependencies we will need.

```
$ npm install --save-dev enzyme mocha expect react-addons-test-utils
```

Because we are using `css-modules` we will need to pass in an empty object as our `style` object and pass in a `noop` (short for "no operation" and pronounced "no op") for images and other non-JavaScript files (thanks to Ole Michelson for solving some of this issue). Create a `.test-setup.js` file and include the following to accomplish this goal:

We also need to add `react-addons-test-utils` because it is a dependency of Enzyme.

```
$ touch .test-setup.js
```

```
/.test-setup.js
commit: coming soon
```

```
const noop = () => {}
const empty = () => ({})

require.extensions['.css'] = empty
require.extensions['.ico'] = noop
require.extensions['.png'] = noop
require.extensions['.jpg'] = noop
require.extensions['.svg'] = noop
```

We are also adding `mocha` so we can run our tests with Mocha using the command line `mocha`.

Writing a unit test

At this point, our `Home` component is pretty simple. All we are going to test is that it renders. Create a file called `spec.js` within our `Home` directory and add the following.

```
$ touch app/components/Home/spec.js
```

```
/app/components/Home/spec.js
commit: coming soon
```

```
import React from "react"
import expect from "expect"
import { shallow } from "enzyme"

import { Home } from "./"

describe("<Home />", () => {
  it("should render", () => {
    const renderedComponent = shallow(
      <Home />
    )
    expect(renderedComponent.is("div")).toEqual(true)
  })
})
```

We're going to use `expect`, which is a simple assertion library that replaces `assert`, `sinon`, and `chai` for our purposes. Almost all of our assertions are going to come down to whether a certain statement is `true` or `false`. Generally speaking, if you're coming up with complex assertions you're probably over-complicating things. It also gives us access to `spy`, which we will go over later.

From `enzyme` we get `shallow`, which is a function that lets us shallow render our component without a DOM. The `shallow` API is really comprehensive and has good documentation. It would be worth your time to look over all the functions you have at your disposal here.

The next part is the `describe` block, where we (not surprisingly) describe the thing that we are testing. Within that `describe` block will be a series of `it` blocks that actually contain the tests we intend to run. `it` blocks usually follow the word "should", and that's a good way to think about what the code in the `it` block should do. In this case, it should render the children of our `Home` component.

The last line is where we make our assertion. If `renderedComponent` has been shallow rendered properly, then it will return a truthy value, which `expect` will pass. If the component is not rendered properly, then our test will not pass.

This is the only test you'll need for most of your components. Basically, we're just testing that the component was able to render without exploding.

If you find yourself testing a lot of logic within your component, you should really consider pulling that logic out into a separate part of your app (likely Redux, which we will go over later). Ideally, you want to keep your components as dumb as possible, meaning they just render content that is passed into them and don't execute any logic.

Running our tests

The last piece is actually running the tests. This is going to require a relatively complicated script in our `package.json` file.

But since we are not running our tests through webpack, we need to make sure we let Babel transpile our code. We can use `babel-register` to transpile our code before passing it on to `mocha`.

```
$ npm install --save-dev babel-register
```

/package.json
commit: coming soon

```
  ...
  "scripts": {
    ...
    "test": "mocha --require babel-register --require .test-setup.js -R spec app/**/spec.js
  },
  ...
```

What we're doing here is telling `mocha` to run our tests, but first passing it through Babel (`--require`

`babel-register`), then changing out our `resolve.extensions` with `.test-setup.js` so we don't have to use webpack, then recursively looking for all `spec.js` files within the `app` directory.

Now you can run it.

```
$ npm test
```

You should get an output like the following.

```
  <Home />
    should render

  1 passing (32ms)
```

It's important to make your test fail in a predictable way so you know that you set your test up properly. Sometimes without knowing it, you might set up a test that passes no matter what you give it. We're not going to do that here for the sake of brevity, but you should make this practice a habit.

Full and static rendering

At a certain point, you'll need to run unit tests that require a DOM. If you want to check to make sure a certain action happened on `componendDidMount`, for example, you'll need to fully render the component so you have access to the lifecycle methods. To accomplish this, you will need Enzyme's Full Render API, which has a syntax almost identical to the shallow render API.

But before we can get this to work, we need a DOM running. The easiest and most efficient way to do this is with jsdom, which is a magical JavaScript implementation of the DOM that lets you run your tests extremely quickly since it's all done in JavaScript.

Setting up `jsdom` is a lot easier than you might expect. Enzyme provides an explanation of how to set it up, and we will walk through it here.

We can copy-paste the code from the documentation to the bottom of our existing `.test-setup.js` file.

```
/.test-setup.js
commit: coming soon
```

```
const noop = () => {}
const empty = () => ({})

require.extensions['.css'] = empty
require.extensions['.ico'] = noop
require.extensions['.png'] = noop
require.extensions['.jpg'] = noop
require.extensions['.svg'] = noop

var jsdom = require('jsdom').jsdom

var exposedProperties = ['window', 'navigator', 'document']

global.document = jsdom('')
global.window = document.defaultView
Object.keys(document.defaultView).forEach((property) => {
  if (typeof global[property] === 'undefined') {
    exposedProperties.push(property)
    global[property] = document.defaultView[property]
  }
})

global.navigator = {
  userAgent: 'node.js'
}
```

What this code is doing is taking our React code and putting it into the global namespace so it can be used by `jsdom`.

Then install `jsdom`.

```
$ npm install --save-dev jsdom
```

And that's all. Now you can run tests with `mount` and `static`, as well as `shallow` as you have before. The app is too simple to require tests of this sort, so we demonstrate the difference between these types of tests when it becomes necessary.

Just be sure you are running a more recent version of node. You must be using at least node 4. If you do not know what version of node you are using, run the following in your terminal.

```
$ node -v
```

If you find that your version of node is not at least 4.0, then update it with `brew`, `nvm`, or whatever method you prefer.

```
$ brew update
$ brew upgrade node
```

Istanbul and test coverage

Another feature that you see in nearly every production app is test coverage. Test coverage is an automated way to let you know how much of your app is covered by tests. If you have no tests, your test coverage is 0%. If you have tests for every function and for every line of code, your test coverage will be 100%. Most companies aim to have tests coverage of 80%-90%, depending on their risk tolerance.

Istanbul is the most popular tool for automated test coverage and it's used by just about every serious open source project. It's also surprisingly easy to set up. Start by installing it. For whatever reason, they decided to call the Babel-compatible command line tool for Istanbul `nyc`.

```
$ npm install --save-dev nyc
```

Then we need to change our `package.json` scripts to run `nyc` over our existing tests so it can automatically detect how much of our code is covered by tests. We don't need to run this every time, so we will make it a separate command. This is going to look a little bit awkward.

/package.json
commit: coming soon

```
  ...
  "scripts": {
    ...
    "cover": "nyc -x '**/*spec.js' -n 'app' -r text -r html -r lcov npm test",
    ...
  }
  ...
```

While this command might look complicated, it's actually pretty reasonable. These commands tell `nyc` to exclude (`-x`) all of our `spec.js` files from our coverage report (because we don't a report on how well our tests are tested), include (`-n`) everything within the `app` directory, use the text, html, and lcov reporters (`-r`) and run `npm test`, defined within our `scripts`, to run the tests.

So now, if you run `npm run cover`, you will see your app with test coverage.

```
-----------|----------|----------|----------|----------|----------------|
File       |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------|----------|----------|----------|----------|----------------|
All files  |       90 |      100 |        0 |       90 |                |
 Home      |      100 |      100 |      100 |      100 |                |
  index.js |      100 |      100 |      100 |      100 |                |
 Sidebar   |       80 |      100 |        0 |       80 |                |
  index.js |       80 |      100 |        0 |       80 |              6 |
-----------|----------|----------|----------|----------|----------------|
```

You should also add the `coverage` directory to `.gitignore` so it doesn't get pushed to your repo.

/.gitignore
commit: coming soon

```
node_modules
.DS_Store
npm-debug.log
dist
coverage
.nyc_output
```

If you're looking for something prettier than the terminal output, you can use your browser to open the `index.html` file within `coverage/lcov-report`. This will also give you hints as to where you should write additional tests to improve your test coverage.

Additional

We will write additional tests as we go along. Testing is an important part of ensuring the long-term stability of your app. Refactoring is inevitable, and if you have good tests, you can ensure that you don't break everything when you change around your components.

Basics of Elixir and Functional Programming

- Functional programming
- Elixir basics
- Types
- Pattern matching

Now that we finally have our React app mostly set up, we can start building out our Phoenix backend. But before we jump into Phoenix, we should go into some of the basics of functional programming and Elixir. This is a long and dense chapter, so you might want to come back to it after a few lessons when you've had a chance to play around with Phoenix.

Elixir is a functional programming language. We could write an entire course simply going over the features of the language, but in this lesson we will just go over some of the specifics of the syntax that you will need to know before moving forward.

Most of what is covered in this lesson are the specifics of Elixir as they relate to Phoenix. The intricacies of Phoenix will be covered in later lessons as they come up.

Before we dive into Elixir and Phoenix, we'll explore the fundamentals of functional programming.

Basics of functional programming

There are two camps in computer programming: Functional programming (often abbreviated FP) and Object Oriented Programming (often abbreviated OOP). There is overlap between these camps, but they differ in their approach to programming.

Most people have experience with OOP, which intentionally mimics objects representing real life things and interactions. Another big part of object oriented programming is `state`. State refers to information stored in an application, object, or variables. Many popular programming languages are object oriented such as Java, JavaScript, C++, and Ruby to name a few.

For example, one can imagine a real-life object, such as a block of clay. You can then mutate that block of clay, apply heat, and end up with a vase.

```
vase = Clay.new
vase.shape
vase.heat
vase.glaze
```

In the above example the vase contains information about itself and whether it has been shaped, heated, or glazed; this is state. Another example of state, using JavaScript, is the `+=` or `-=` operators which modify an existing variable rather than return a new one:

```
var counter = 0;

counter += 5;
counter -= 1;
counter += 2;

console.log(counter); // -> 6
```

Functional programming is a bit more abstract, and functional programming advocates are more hardcore about definitions than object oriented folks. You'll hear a lot about "immutable data" and "pure functions", and a variety of other terms. We will go over the terms that are relevant to working with Phoenix and Elixir.

One of the core concepts of functional programming is the "pure function". A pure function is a function that always yields the same results with the same arguments and has no side effects. You may also hear them referred to as "idempotent" (pronounced īdem'pōt(ə)nt), which is a fancy way of saying "it has no side effects".

A side effect is something like writing to a database, or making a change to something that is outside the scope of a function. They are changes that, if run again, may produce a different outcome. Additionally, pure functions need everything passed to them as arguments; retrieving values elsewhere could lead to a different outcome.

Here is an example of an *impure* function:

```
var counter = 0;

function addOne() {
  counter += 1;
}

addOne()
```

Our example has a state, `counter`, outside the scope of the function that is modified by our function. If

you run it once, the result is `1`. If you run the exact same function twice, the result is `2`.

This side effect wouldn't be possible with a pure function. An example of a pure function would be the following:

```
function addOne(counter) {
  return counter + 1;
}

var counterPlusOne = addOne(2);
```

In this example, we are passing the counter in as an argument to the function, which then adds one and returns the new result. No matter how many times you run this function, it will always give you the same result (`3`).

Pure functions are important to functional programming because they allow us to chain together functions, perform multiple transformations to data, and get a predictable result without side-effects. This lack of side-effects is the foundation of Elixir's fault tolerance and ability to re-try code without fear; this is something we'll discuss more later. As we will see shortly, `Plugs` are an important part of Phoenix and are themselves, by necessity, pure functions.

In functional programming we do not mutate data, instead we create a new copy with our changes. This may seem like a lot of extra work at first but functional programming languages are designed for this. This immutable data furthers our ability to ensure predictable outcomes and allows for the more advanced features of Elixir's concurrency and fault tolerance. Don't worry though, we'll cover all of this in more depth in later lessons.

If you are interested in learning more about functional programming, there is an edX course that covers a lot more than we will cover in this course.

Interactive Elixir

Elixir ships with IEx, an interactive shell. With IEx we can evaluate Elixir expressions and test out functionality before adding it to our code base. The best way to understand IEx is to experience it. In your terminal, run the `iex` command.

```
$ iex
```

You'll get an output similar to the one below.

```
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll

Interactive Elixir (1.2.0) -press Ctrl+C to exit (type h ENTER for help)
iex>
```

We are now in an interactive environment where we can explore Elixir and evaluate expressions. Let's start with trying to run some simple Elixir code. Inside IEx let's add together some numbers. If we type in `2 + 2` and press `return`, the expression is evaluated and the result is displayed.

```
iex> 2 + 2
4
iex> 2.5 + 1
3.5
```

Basic types

As you might expect coming from JavaScript, Ruby, or other major languages Elixir has support for the basic types you'd expect. With IEx open, let's try working with some integers and floats.

```
iex> 1 + 2
3
iex> 2.0 + 2.5
4.5
```

Elixir supports the booleans `true` and `false`, they are also represented by the atoms (explained momentarily) `:true` and `:false`. All values in Elixir, with the exception of `nil` and `false`, are considered truthy.

```
iex> true
true
iex> true === :true
true
iex> false
false
iex> if 6, do: "Truthy"
"Truthy"
```

Atoms

If you're familiar with Ruby then `atoms` won't be foreign to you, they're just like `symbols`. If you're new to both, don't worry.

In Elixir an `atom` is a constant who's name is also its value. They are lowercased and begin with a colon. `:ok` and `:error` are example atoms. What makes Atoms special is how they use memory. When we use the atom `:ok` many times the memory is only allocated once. Atoms are great when we need to reuse the same value many times and wish to conserve memory. Unlike other values though, atoms are not garbage collected so they should be used sparring; if we *always* used atoms in place of other values, our memory would never be freed.

Strings in Elixir

Strings are UTF-8 encoded and use double quotes, let's try the standard `"Hello World"`:

```
iex> "Hello World"
"Hello World"
```

What about single quoted values like `'hello world'`? In Elixir, single quotes are used to denote char lists. A char list is nothing more than a list of individual characters:

```
iex> 'hello world'
'hello world'
iex> is_list('hello world')
true
```

Char lists can be tricky in IEx. If characters are outside the ASCII range then a list of character codes will be displayed instead; these character codes are the numerical value for each character:

```
iex> 'hełło'
[104, 101, 322, 322, 111]
```

Collections

Collections are a big part of functional programming. As we'll see, Elixir has a number of different data

structures available to us. The simplest of these collection is the `list`. Lists may contain different types of values and do not enforce uniqueness, we can store the same value in a list many times.

Let's try creating some simple lists inside IEx:

```
iex> [1, 2, 2, 3, 3, 3]
[1, 2, 2, 3, 3, 3]
iex> [1, 2, 3, "a", "b", "c"]
[1, 2, 3, "a", "b", "c"]
```

If you don't have a background in FP then `tuples` may seem strange to you but tuples are a common sight in Elixir. A tuple is similar to a list but is stored in memory contiguously, making accessing them fast but modification expensive. Tuples are differentiated from lists by using curly braces. Just like `list` our tuples can contain any type of value and do not enforce uniqueness:

```
iex> {:ok, "Success"}
{:ok, "Success"}
iex> {10, "ten", :ten}
{10, "ten", :ten}
```

You can think of tuples as something like a pair of values that are meant to be stored together. They aren't there for efficient modification.

If we combine lists, tuples, and atoms together we get `keyword lists` in Elixir. A keyword list is a collection of two-element tuples who's first value is an atom, very specific. Keyword lists are frequently used for options and configuration. A keyword list can contain multiple tuples with the same key.

```
iex> [age: 30, name: "user"]
[age: 30, name: "user"]
iex> [{:age, 30}, {:name, "user"}]
[age: 30, name: "user"]
```

Maps are the go-to key-value store in Elixir. We can use any value as a key in our maps. To define a `map` we use the `%{}` syntax:

```
iex> map = %{"hello" => :world, :ok => 200}
%{:ok => "okay", "hello" => "world"}
iex> map[:ok]
200
iex> map["hello"]
:world
```

Keys in a map are unique, if we add a new value with the same key it will replace the original.

```
iex> %{"hello" => "world", "hello" => "universe"}
%{"hello" => "universe"}
```

Pattern Matching

If you haven't used functional programming before then `pattern matching` may seem a bit strange at first, but it is arguably one of the best features of Elixir. When we use pattern matching we not only compare values but structure as well.

Through pattern matching we are also able to extract values from a match into a variable, this is known as variable capture. In a moment we'll try some examples which should help clear things up, but first we need to discuss the match operator.

Unlike other languages Elixir does not use the `=` operator for value assignment, instead it is use as our match operator.

Confused yet? When we make a successful match, the matched value will be returned. If the match fails, it will raise an error. In IEx, let's start with a few simple pattern matches to get our feet wet.

```
iex(24)> "hello" = "hello"
"hello"
iex> [1, 2] = [1, 2]
[1, 2]
iex> [1, 2] = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
```

In our next example we'll introduce variable capture. In it's simplest form variable capture does resemble an assignment (like the "=" in JavaScript) but it's important to remember they're different. To better visualize the difference, and to demonstrate the power of variable capture, let's try some examples.

Start with the simple `a = 1` match. In this instance the value of `1` is captured in the variable `a`. Although it may look like an assignment, it's important to remember it is not. To better illustrate the difference, and power of variable captures, let's try matching `[1, b]` to `[1, 2]`. Our match will be successful and if we look at the variable `b`, we should get `2`.

```
iex> a = 2
2
iex> a
2
iex> [1, b] = [1, 2]
[1, 2]
iex> b
2
```

Be aware that variable capture works right to left. Variables included on the right side of the match are evaluated and their values used. With our previous example as a starting point, use `[a, b]` on the right side of a match to demonstrate how the values are used.

```
iex> [1, 2] = [a, b]
** (CompileError) iex:1: undefined function a/0
```

What if we need to use a variable's value in a match but don't want to reassign the value of that variable? To accomplish this, we can use variable pinning, available to us through the pin operator: `^`. Let's try some examples with the pin operator.

To begin let's capture a value into a variable, something like `a = 1` will work. Next we'll use our variable and pin operator, `^a`, as part of a match. To demonstrate a successful match, let's try to match `[^a, 2]` to `[1, 2]`. If we change the right side to `[2, 2]` we should get a match error.

```
iex> a = 1
1
iex> [^a, 2] = [1, 2]
[1, 2]
iex> [^a, 2] = [2, 2]
** (MatchError) no match of right hand side value: [2, 2]
```

You may not find yourself using variable pinning much but it's an important piece to be aware of.

Additional

We're going to go into functions and modules in the next lesson as well as some of the basics of Phoenix and the Model-View-Controller.

More Basics of Elixir and Phoenix

- Functions and modules
- Notes for Ruby developers
- Basics of Phoenix and MVC

Control Structures

Elixir supports `if` and `unless` like other modern languages such as Ruby and CoffeeScript. If you aren't familiar with `unless`, don't worry it's just syntactic sugar for: `if !value`.

```
iex> if true do
...>   "truth"
...> else
...>   "false"
...> end
"truth"
```

If we replace `if` with `unless`, we should get the opposite.

```
iex> unless true do
...>   "true"
...> else
...>   "false"
...> end
"false"
```

Elixir doesn't stop with just `if` and `unless`, it includes two addition control structures: `case` and `cond`. The `case` structure is similar to a `switch` statement in other languages but it relies heavily on pattern matching. The best way to understand the `case` statement is to try it in IEx.

```
iex> case {:ok, "Success"} do
...>   {:ok, value} -> value
...>   _ -> "No match"
...> end
"Success"
```

You may have noticed an underscore (`_`) above. An underscore is a placeholder that accepts any value. So in the example above, if the case does not match `{:ok, value}`, then it will pass to `_` which will match with anything else and return "No match".

Last but not least is `cond`. We can use `cond` when we want to match conditions and not values or patterns. The `cond` structure is similar to `else if` in other languages. An example will help us better understand it's usage.

```
iex> val = 4
4
iex> cond do
...>   val + 2 < 4 -> "val < 2"
...>   val + 2 >= 4 -> "val >= 2"
...> end
"val >= 2"
```

Pipes

One of the nicest syntactic features of Elixir is the pipe operator, `|>`, which allows you to "pipe" a value from one function to another function. This is very similar to the unix pipeline) `|` if you've ever used that. In its simplest form, the `|>` takes the left value and passes it to the right side as the first argument.

This is particularly useful because Elixir is just collections of functions, piping them together makes the code expressive and easy to understand at a glance. For example, if were were to make a cake using functions, we could write.

```
ingredients
|> measure
|> mix
|> bake
|> decorate
```

Each step takes the result of the previous function and applies a transformation to it. This is a surprisingly powerful abstraction that makes it very easy to understand the flow of your app and makes for highly-maintainable code.

Functions

Functions in Elixir, and other functional programming languages, are considered first class citizens. That means that all operations available to other types are available to them, they can be passed as

arguments to other functions, assigned to variables, and returned from other functions.

In Elixir there are a couple types of functions, but for this section we'll focus on anonymous functions. As the name suggests, anonymous functions have no name. If not captured in a variable, an anonymous function would otherwise not exist. To define an anonymous function we'll use the `fn (arg) -> body end` syntax. To call an anonymous function we use the `.()` syntax on our variable, which is similar to what you find in other languages.

Let's start with an example function to add two numbers together and then try using it.

```
iex> add = fn (a, b) -> a + b end
#Function<12.54118792/2 in :erl_eval.expr/5>
iex> add.(1, 2)
3
```

Elixir also includes a helpful shorthand syntax for anonymous functions. This shorthand makes use of the `&` operator. With the shorthand syntax our arguments are referred to as `&1`, `&2`, and so on. We can try our previous example using the shorthand syntax.

```
iex> add = &(&1 + &2)
&:erlang.+/2
iex> add.(1, 2)
3
```

We learned about pattern matching using the match operator (`=`) but we can also use it when specifying function signatures. Let's expand on our prior example. Let's update the function to return the string `"zero"` if someone tries to add `0` and `0` together:

```
iex> add = fn
...>         (0, 0) -> "zero"
...>         (a, b) -> a + b
...>       end
#Function<12.54118792/2 in :erl_eval.expr/5>
iex> add.(1, 2)
2
iex> add.(0, 0)
"zero"
```

That's just the start of functions in Elixir. In the next section, and later lessons, we'll cover other function types and how they all fit together.

Modules

At their most basic, modules are collections of functions. With `defmodule` we can create a new module, add functions to it, and then re-use it elsewhere in our application. We'll learn more about modules in later lessons, but for now we'll focus on the basics.

We've already covered anonymous functions so let's take a look at functions with regards to modules. In our anonymous functions we used the `fn -> ... end` syntax, but named functions (or functions in modules) use `def` and a lowercase name.

Let's create a simple example module that can add two numbers together. We can start by using `defmodule Example` to define our module. Then we'll need to use `def add(a, b)` to define our function, implementing the functionality should be easy. Once we've defined everything we can use it by calling `Example.add(1, 2)`. If we put it all together we should get something like this:

```
iex> defmodule Example do
...>   def add(a, b) do
...>     a + b
...>   end
...> end
{:module, Example,
 <<70, 79, 82, 49, 0, 0, 4, 248, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 157, 131, 104, 2
 {:add, 2}}
iex> Example.add(1, 2)
3
```

You might be confused by all the numbers that Elixir has outputted for us. For now, don't worry about it because we won't be using that information any time soon. But if you insist on jumping ahead, you can check out Elixir binaries.

That's the tip of the iceberg for modules. In the next lessons we'll learn more about composing our modules and connecting the pieces.

Structs

When we need a named key-value store with known keys, we turn to `structs`. Structs share the name of the module they're defined in and use a syntax similar to our maps. We define them with `defstruct` and a keyword list of fields and default values. For our example, let's create a simple module and struct to present a person:

```
iex> defmodule Person do
...>   defstruct name: nil, age: 0, location: nil
...> end
{:module, Person,
 <<70, 79, 82, 49, 0, 0, 5, 32, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 133, 131, 104, 2,
 %Person{age: 0, location: nil, name: nil}}
```

Now that we have our struct created, it's time to use it. As we already learned, structs and maps share a similar syntax. In fact, structs are just maps with a few extra features. Create a new struct containing your information and try to access the fields:

```
iex> sean = %Person{name: "Sean", age: 31, location: "US"}
%Person{age: 31, location: "US", name: "Sean"}
iex> sean.age
31
```

Notes for Ruby Developers

It's no coincidence that Elixir and Ruby share many similarities; Jose, the creator of Elixir, was a contributor to Ruby and Rails.

[TODO] insert handy chart here

Model-View-Controller

Phoenix is the framework we'll be using with Elixir. If you've used an MVC framework such as Rails or SailsJS, a lot these topics will be familiar to you.

Phoenix uses the standard Model-View-Controller (MVC) architecture that we have come to expect while building web apps. In general, anything related to the web application can be found in the `web` directory. In there we will find out `models`, `views`, `templates`, and `controllers`. For those unfamiliar with MVC, we'll do high level overview.

The appeal of the MVC pattern is the separation of concerns. As you probably guessed there are three components to the MVC pattern: models, views, and controllers. Another way of saying that is: data, user interface, decisions. In a MVC application the `model` is responsible for managing data and business rules. Our `view` handles the user interface be it HTML, XML, or JSON. Finally, our `controller` handles most of the work orchestrating inputs to models and finally views.

Most of the logic of how the user interacts with the app will be handled by the controller. You will see this

in practice in short order, so there is no reason to go into detail here.

Additional

We will cover Phoenix and Elixir in more detail as the concepts become relevant in the rest of the tutorial.

In Phoenix, most files are named with `_` underscores. For example, `user_controller.ex`.

When debugging, most people use `IO.puts` to log some particular output to the terminal. For example, `IO.puts "Running!"` will simply return a string if it is called.

User Accounts and Signup: Part 1

- Account creation
- JSON web token (JWT)
- Migrations
- CORS

This is the first lesson in which we will make significant changes to our Phoenix backend. Just about every app has user accounts, often with varying degrees of complexity. For our app, we will use a standard email-password combination.

In this lesson, we create the `User` model and add API endpoints so our frontend can create and update an account. We're also going to create and run a migration to update our database.

Creating our user

The first thing we need to do is create a user model and controller, which we cover in more detail shortly. Creating these components is a standard part of most Phoenix applications so there are helpers available to generate the boilerplate.

To view a list of the generators, and other `mix` tasks available to us, run `mix -h`:

```
$ mix -h

mix                   # Runs the default task (current: "mix run")
...
mix new               # Creates a new Elixir project
mix phoenix.digest    # Digests and compress static files
mix phoenix.gen.channel # Generates a Phoenix channel
mix phoenix.gen.html  # Generates controller, model and views for an HTML based resource
mix phoenix.gen.json  # Generates a controller and model for a JSON based resource
mix phoenix.gen.model # Generates an Ecto model
mix phoenix.gen.secret # Generates a secret
mix phoenix.new       # Create a new Phoenix v1.1.3 application
mix phoenix.routes    # Prints all routes
mix phoenix.server    # Starts applications and their servers
...
```

There are many tasks but for now we'll focus only on the Phoenix ones. Since we're building a JSON API,

let's try the `phoenix.gen.json` generator. This will cause an error but the errors in Phoenix are quite helpful and will tell you what you're missing.

```
$ mix phoenix.gen.json

** (Mix) mix phoenix.gen.json expects both singular and plural names
of the generated resource followed by any number of attributes:

    mix phoenix.gen.json User users name:string
```

Let's try it again but this time for a user with an email, encrypted password, and a username:

```
$ mix phoenix.gen.json User users email:string encrypted_password:string \
  username:string
```

You will get output similar to the following.

```
* creating web/controllers/user_controller.ex
* creating web/views/user_view.ex
* creating test/controllers/user_controller_test.exs
* creating web/views/changeset_view.ex
* creating priv/repo/migrations/20160406190555_create_user.exs
* creating web/models/user.ex
* creating test/models/user_test.exs

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

    resources "/users", UserController, except: [:new, :edit]

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

This generates a number of different files and instructions for setting up our user. We'll take this opportunity to discuss some of the files and their purpose within our project.

The Controller

In the output from our generator the very first file is `web/controllers/user_controller.ex`.

Controllers are responsible for doing most of the work for a given request. When our application receives a request it will use the router to direct the request to a controller and a specific function. For example,

when someone arrives at your website, you need to determine whether or not that person is a logged-in user. The controller is where your app will check with the database to determine how to handle the current connection.

If you come from a Rails background, the concept of a controller should be familiar.

Controllers are a big topic and will be used frequently so if you do not understand them now, you will after we use them a little more.

The View

The second file created is `web/views/user_view.ex`, our view. If you're familiar with other frameworks, you might be expecting the view to be HTML files but that's not necessarily the case. In Phoenix, our view is a module containing functions for rendering data into a consumable format which can be either JSON or HTML. Since we're building a JSON API, it will respond with JSON.

Our layout and other HTML assets are referred to as *templates* in Phoenix. These templates can be used by our views to generate the final HTML or JSON.

By default our generated boilerplate includes our password in the JSON response. Open the user view and update the `render/3` function by removing the `encrypted_password` reference:

web/views/user_view.ex
commit: coming soon

```
...
def render("user.json", %{user: user}) do
  %{id: user.id,
    email: user.email,
    username: user.username}
end
```

The Migration

A migration is a set of changes we want applied to our database schema, such as adding new tables and columns or updating existing ones. When we look at the generated migration we'll see it is creates a new database table for `"users"` and add a number of column for each of the three fields we supplied the generator. The migration that was generated for us can be found at `priv/repo/migrations/20160406190555_create_user.exs`; it's important to note that your file will almost certainly be different since migrations rely on timestamps at the beginning of the file for uniqueness.

Let's open our migration make a few changes. For starters we want to make sure that users enter a valid username and email for an account to be created. We also want to make sure that email addresses are unique.

```
/priv/repo/migrations/2016..._create_user.exs
commit: coming soon
```

```
defmodule PhoenixChat.Repo.Migrations.CreateUser do
  use Ecto.Migration

  def change do
    create table(:users) do
      add :username, :string, null: false
      add :email, :string, null: false
      add :encrypted_password, :string

      timestamps
    end

    create unique_index(:users, [:email])
    create unique_index(:users, [:username])
  end
end
```

The first change we made was to add `null: false` to both `:username` and `:email`, this tells the database that `null` values are not to be accepted. In other words, a value is required. The second change we made was to add `create unique_index/2`. A unique index is to make sure a username and email is only used once.

The Model

Next up is our model, found in `web/models/user.ex`. In Phoenix, models are modules containing functions for working with our data's `schema` and the `struct` that contains its values. We're already familiar with structs and if you've worked in another framework, you're probably already familiar with a schema, but in case you aren't, a schema is describes the different fields of our table and it is defined within the `schema` block. You can think of it as a way to enforce a particular structure.

Let's open our model and add support for the unique index we added into our migration.

```
/web/models/user.ex
commit: coming soon
```

```
defmodule PhoenixChat.User do
  use PhoenixChat.Web, :model

  schema "users" do
    field :email, :string
    field :encrypted_password, :string
    field :username, :string

    timestamps
  end

  @required_fields ~w(email encrypted_password username)
  @optional_fields ~w()

  def changeset(model, params \\ :empty) do
    model
    |> cast(params, @required_fields, @optional_fields)
    |> validate_format(:email, ~r/@/)
    |> validate_length(:username, min: 1, max: 20)
    |> update_change(:email, &String.downcase/1)
    |> unique_constraint(:email)
    |> update_change(:username, &String.downcase/1)
    |> unique_constraint(:username)
  end
end
```

We've made some changes to our `changeset/2` function. A `changeset` is a function that validates and transforms data into a format the database is expecting. For now we'll focus on a single changeset function but it's possible, and common, to use different functions for different situations. You will see some of these later on.

Let's look at the individual changeset changes we made. With `validate_format(:email, ~r/@/)` we check the email field using a regular expression to ensure it contains an `@`. The second change we introduced is `validate_length(:username, min: 1, max: 20)` to ensure a username is at least one character and at most 20.

Finally, the last change ties in with the index we added to our migration: `unique_constraint(:email)`. With the `unique_constraint/3` function we can ensure the email is not already in use, thanks to our `unique_index`. We also do the same for `:username`.

The Router

The last piece to our puzzle is the router, found in `web/router.ex`. The output from our generator directed us to make a change, open the router and update it accordingly.

When our frontend makes a request to our API, the first place it hits is the `router`. This router sends that request to the right place so the request can be handled properly.

/web/router.ex
commit: coming soon

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

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

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

  scope "/", PhoenixChat do
    pipe_through :browser # Use the default browser stack

    get "/", PageController, :index
  end

  scope "/api", PhoenixChat do
    pipe_through :api

    resources "/users", UserController, except: [:show, :index, :new, :edit]
  end
end
```

With this change we are telling Phoenix that any requests to `/users`, except `:new` and `:edit` should be sent to our `UserController`. We're also removing `:index` and `:show` since we don't want random users to be able to see our list of users, but for some apps, you'd want this behavior.

There are a few things to notice in this file. There are two `pipeline` functions and two `scope` functions. Since we are creating an API we won't be using the `:browser` pipeline, but you can see that it accepts HTML and has a series of plugs that apply useful transformations to your connection. For our application we'll focus on the `:api` pipeline.

Within the code we just wrote, you see that any connection that hits the `/users` route is "piped through" the `:api` pipeline. Be wary of adding plugs to the API pipeline, because if you do, that transformation will be applied to every incoming connection. This is a very powerful tool and should be used with discretion.

The `scope` allow us group different routes together under a common base path. In our case, we want to direct anyone hitting the `/api/users` route to the `UserController`.

Finally, we see the `resources/4` function which takes our path, controller, and options. Within the Phoenix router, `resources/4`, is a handy function that takes care of generating all of the standard routes for us. We can see a list of the routes `resources/4` has created by running `mix phoenix.routes` in our command line:

```
$ mix phoenix.routes
```

If you run the above command you should see something like this (if you do not see these, make sure your app has been compiled):

```
page_path  GET     /               PhoenixChat.PageController :index
user_path  POST    /api/users      PhoenixChat.UserController :create
user_path  PATCH   /api/users/:id  PhoenixChat.UserController :update
           PUT     /api/users/:id  PhoenixChat.UserController :update
user_path  DELETE  /api/users/:id  PhoenixChat.UserController :delete
```

In later sections we'll explore creating individual routes without the `resources/4` helper.

Once we've updated our migration we should create the database and run the migration. Running our migration will apply the changes to our database:

```
$ mix ecto.create
$ mix ecto.migrate
```

If this isn't your first time through the tutorial, you might need to run `mix ecto.drop` first to drop the old database.

User Accounts and Signup: Part 2

- Signup
- CORS

In this lesson, we add the ability to create a new user with a secure, hashed password. Then we set up our endpoint to handle incoming messages from our frontend so we can connect and create new users.

Sign-up

Before we can sign-in, we need to sign-up. We have our user boilerplate done so now we can look at how to register new accounts.

In a important part of registering a new account is picking a strong password. An even *more* important part of the process is securely storing that password. For this, we're going to rely on the library [Comeonin][comeonin]. Before we can use it, we need to include it as a dependency in our `mix.exs` file; `mix.exs` is much like to NPM's `package.json` and Bunder's `Gemfile`.

Make sure your version of `phoenix` is at least 1.2 and that your version of `phoenix_ecto` is at least 3.0. Several features of this app will not work if you are not on the current version.

/mix.exs
commit: coming soon

```
...
defp deps do
  [
    {:comeonin, "~> 2.3"},
    {:cowboy, "~> 1.0"},
    {:gettext, "~> 0.11"},
    {:phoenix, "~> 1.2.0"},
    {:phoenix_pubsub, "~> 1.0"},
    {:phoenix_ecto, "~> 3.0"},
    {:phoenix_html, "~> 2.6"},
    {:phoenix_live_reload, "~> 1.0", only: :dev},
    {:postgrex, ">= 0.0.0"}
  ]
end
```

In addition to `deps/0` we need to update `applications/0`. This is a little bit different than you might be used to, but it's still intuitive. In Phoenix/Elixir, some modules can be managed independently and need to be started along with your app. Many dependencies you add will need to be added here.

The primary exceptions are libraries that rely on an existing application that you've already started. For example, some modules will use [HTTPoison][httpoison] and don't need to be started independently. We will cover that in more detail when it becomes relevant. We will cover these applications in a later lesson but for now we only need to add `:comeonin`:

```
/mix.exs
commit: coming soon
```

```
def application do
  [mod: {PhoenixChat, []},
    applications: [
      :comeonin,
      :cowboy,
      :gettext,
      :logger,
      :phoenix,
      :phoenix_ecto,
      :phoenix_html,
      :phoenix_pubsub,
      :postgrex
    ]]
end
```

Now you should run `mix deps.get` to get install our dependencies:

```
$ mix deps.get
```

Next, we'll revisit our model and changesets.

Registration changeset

During the registration process there are some things we need to do then and only then, like validating password length and hashing our password with [Comeonin][comeonin]. For these reasons we'll create a new changeset just for registrations. The first challenge we have to over come is where to store the unhashed password, thankfully we can use a `virtual` field in our schema.

Virtual fields are part of the `schema` and `struct` but their values are not persisted (saved) to our database. Using a virtual field allows us to store both the hashed and unhashed versions of the password

in the same struct, making our work with changesets easier.

Let's begin by updating our model in `web/models/user.ex`. We will start by including the virtual password field:

/web/models/user.ex
commit: coming soon

```
  ...
  schema "users" do
    field :email, :string
    field :encrypted_password, :string
    field :username, :string
    field :password, :string, virtual: true

    timestamps
  end
```

Let's update our original `changeset/2` function to ignore the password:

/web/models/user.ex
commit: coming soon

```
  ...
  @required_fields ~w(email username)
  @optional_fields ~w()
  ...
```

Now we can create our new `registration_changeset/2` function. For this changeset we want to require our virtual password field, validate the password length, and finally hash it using Comeonin. If you're not familiar with hashing, you can think of it as turning a plain-text string into a (pseudo) random string that can only be decoded if you have the right key. Let's make those changes now:

/web/models/user.ex
commit: coming soon

```
  def registration_changeset(model, params) do
    model
    |> changeset(params)
    |> cast(params, ~w(password), [])
    |> validate_length(:password, min: 6, max: 100)
    |> put_encrypted_pw
  end

  defp put_encrypted_pw(changeset) do
    case changeset do
      %Ecto.Changeset{valid?: true, changes: %{password: pass}} ->
        put_change(changeset, :encrypted_password, Comeonin.Bcrypt.hashpwsalt(pass))
      _ ->
        changeset
    end
  end
```

In addition to our `registration_changeset/2` function we've also created `put_encrypted_pw/1`, this private function will hash our password when a changeset is valid.

Before we go too much further, let's look at why we need a separate changeset for registration. The primary reason for separate changeset is so we can validate and hash passwords *only* when necessary. Password hashing is an expensive operation in which a complex algorithm is used to convert our password into a non-reversible random string of characters and numbers. When we hash passwords we are able to safely store them in our database.

Storing passwords unhashed is a *major* no-no. Without hashing, our passwords would be available in clear text to anyone with access to the database, like employees or a malicious attacker. Hashing passwords ensures your user's data is just that much safer.

Controller updates

With our registration changeset complete, we can update our user controller in `controllers/user_controller.ex` to use it. Updating our controller is easy thanks to changesets. All we need to do is update our `create/2` function to use `registration_changeset/2` in place of `changeset/2`.

```
/web/controllers/user_controller.ex
commit: coming soon
```

```
def create(conn, %{"user" => user_params}) do
  changeset = User.registration_changeset(%User{}, user_params)

  case Repo.insert(changeset) do
    {:ok, user} ->
      conn
      |> put_status(:created)
      |> render("show.json", user: user)
    {:error, changeset} ->
      conn
      |> put_status(:unprocessable_entity)
      |> render(PhoenixChat.ChangesetView, "error.json", changeset: changeset)
  end
end
```

We can also delete the `show/2` and `index/2` functions since we aren't going to call them from our frontend.

Our registration API is now ready for use. With a valid email, password, and username our controller will create a new record in the database and return the user as JSON.

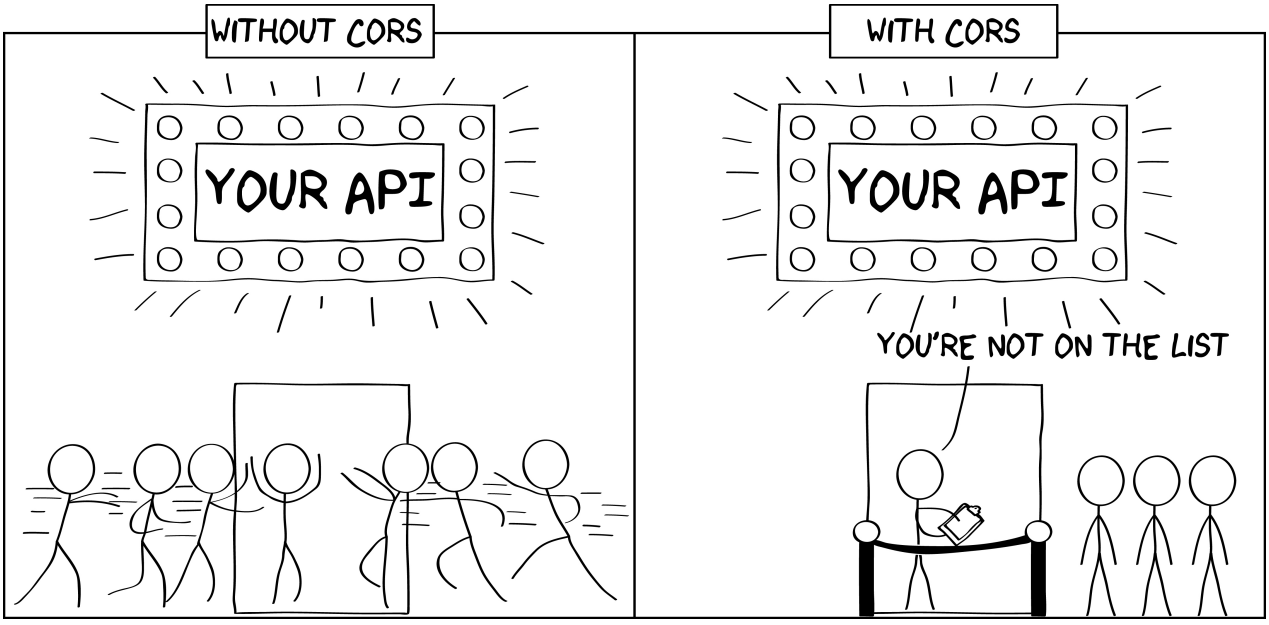
Cross-Origin Resource Sharing (CORS)

If you've heard of CORS before then reading this section title may have cause you to wince; don't worry we'll only be covering CORS at a high level.

Cross-origin resource sharing, or CORS, is a mechanism for limiting which domains and applications can access our API. CORS isn't usually an issue, since we often times serve the front-end application on the same host as the backend (as in, both the backend and the frontend are run from the same server, like with Rails, Meteor, etc).

In our application however, we've separated the front-end from the back-end and hosting them in two totally separate places. While running locally, our API is on `localhost:4000`, while our frontend is running on `localhost:3000`. Because of this, we need to add support for CORS to our project, otherwise, our API would only accept requests originating from `localhost:4000` and would ignore any requests that originate from `localhost:3000`.

You can think of CORS as a doorman that checks to make sure that the person requesting entrance is on the list. If your API is only supposed to be used by your frontend, then you can limit the incoming requests to that domain. You can also set CORS to accept all incoming traffic from any domain.



To faciliate supporting CORS, we're going to include the Corsica library. We need to start by updating our dependencies in `mix.exs`, we won't need to add Corsica as an application though. When we're done, don't forget to run `mix deps.get`:

```
/mix.exs
commit: coming soon
```

```
defp deps do
  [
    {:comeonin, "~> 2.3"},
    {:corsica, "~> 0.4"},
    ...
  ]
end
```

```
$ mix deps.get
```

Now that we have Corsica in our project we need to configure it. Lucky for us there is only one file we need to change: `lib/phoenix_chat/endpoint.ex`. Open the endpoint file and add this changes at the bottom:

```
/lib/phoenix_chat/endpoint.ex
commit: coming soon
```

```
defmodule PhoenixChat.Endpoint do
  ...
  plug Corsica, allow_headers: ~w(Accept Content-Type Authorization)
  plug PhoenixChat.Router
end
```

Don't worry if you don't understand all of this yet, we'll discuss Plugs more shortly. Just know that any request we receive will pass through `Corsica` before it hits our `Router`.

We are instructing Corsica to allow data with headers `Accept`, `Content-Type`, and `Authorization`. This will make more sense when we implement an API call from the frontend.

The last thing to do is to start your Phoenix server so we can start making requests. You do this by running `mix phoenix.server`, which will default to `localhost:4000`.

```
$ mix phoenix.server
```

You will want to keep this open and running so we can make requests to our backend from our React frontend.

Create a Reusable Button Component

- CSS variables
- Buttons
- Unit tests

Before we get too far, we're going to need buttons. Buttons are a simple component that we can easily make reusable, so rather than use a global style and use a class like `button-primary` as you would with a framework like Bootstrap, we're going to create a component that we can reuse. We're going to have at least three types of buttons and we will use `props` to tell the button what to display.

```
$ mkdir app/components/Button
$ touch app/components/Button/{README.md,spec.js,index.js,style.css}
```

Our buttons will take in two properties, `type` and `onClick`. The `type` will determine the style, and `onClick` will take in a function that tell the button what to do when it's clicked. We will also pass in the text of the button as `props.children`. We are going to create our button as an efficient [stateless functional component](#).

```
/app/components/Button/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

export const Button = props => {
  return (
    <button
      style={props.style}
      onClick={props.onClick}
      className={style[props.type]}>
      {props.children}
    </button>
  )
}

export default cssModules(Button, style)
```

The code above should all look familiar. We can style these components with the same technique we

used previously, but since different button types will share a lot of styling, we should look into ways of composing these to keep our CSS as DRY as possible.

Style the buttons

Now back to our `Button` component, within our `style.css` file, add the following, which will compose our buttons without repeating code (explained below).

/app/components/Button/style.css
commit: coming soon

```
.button {
  color: white;
  border-radius: 5px;
  border: 1px;
  padding: 0.8rem 2rem;
  cursor: pointer;
  font-size: 1em;
  align-self: center;
  outline: none;
}

.primary {
  composes: button;
  background: rgb(239, 95, 78);
}
.primary:hover {
  background: color(rgb(239, 95, 78) lightness(+7%));
}

.flat {
  composes: button;
  background: white;
  color: rgb(239, 95, 78);
  border: 1px solid rgb(239, 95, 78);
}
.flat:hover {
  color: color(rgb(239, 95, 78) lightness(+15%));
}

.accentPrimary {
  composes: button;
  background: rgb(58, 155, 207);
  color: white;
}
.accentPrimary:hover {
  background: color(rgb(58, 155, 207) lightness(+10%));
}
```

```
.accentFlat {
  composes: button;
  background: white;
  color: rgb(58, 155, 207);
  border: 1px solid rgb(58, 155, 207);
}
.accentFlat:hover {
  color: color(rgb(58, 155, 207) lightness(+15%));
}

.disabled {
  composes: button;
  cursor: not-allowed;
}
```

First, we're creating the `button` class, which will contain all of the styles that are shared between the other buttons.

Then we compose using `composes`, which we get from CSS Modules, a `flat`, a `primary`, and a `disabled` button from the shared characteristics of the `button`, as well as an alternative color ("accent").

Alternatively, we could use the next-version syntax for applying styles to multiple classes. You can do this by assigning a variable to `:root`, then using `@apply` to add the styles to other classes. For more on this, see the (docs). The downside is that this will repeat code in your stylesheet.

Because we're using CSSNext, we can use the new `color` function (docs), which replace a lot of the useful functions that we get with a preprocessor. In this case, we are taking a color and making it lighter by a certain percentage. You can also make it darker by changing the `+` to a `-`.

We should also define our `PropTypes` since we'll be using this button a lot and it will be good to know if we are accidentally causing an error.

```
/app/components/Button/index.js
commit: coming soon
```

```
...

Button.propTypes = {
  style: React.PropTypes.object,
  onClick: React.PropTypes.func,
  type: React.PropTypes.string.isRequired,
  children: React.PropTypes.node.isRequired
}
```

If you're not familiar with PropTypes, you can think of them as a way to warn you in advance if you are not

sending a necessary piece of data to a particular component. In the case of our `Button`, if we do not send `type` or `children` to our Button, it won't render properly, so we require that both of those props are passed in.

We are setting `onClick` as an optional parameter since we may not necessarily want the button to do anything if we click on it--sometimes the action will be handled by the wrapper outside of the button.

Unit tests

Now we should write our unit tests for our `Button` component. We know that we want our button to 1. Render its children, 2. Take in `primary`, `flat`, and `disabled` as `type`, 3. Handle click events, and 4. Render as a `button`.

Go ahead and add all the following tests. These should all look very familiar as they are almost identical to the tests we wrote for our `Modal` component. The only difference here is that we are passing in the style as `type` and checking to make sure that component rendered with the right style.

```
/app/components/Button/spec.js
commit: coming soon
```

```
import React from "react"
import expect from "expect"
import { shallow } from "enzyme"

import { Button } from "./"

const props = {
  type: "primary",
  onClick: () => {}
}

describe("<Button />", () => {
  it("should render its children", () => {
    const children = (<p>foo</p>)
    const renderedComponent = shallow(
      <Button {...props}>
        { children }
      </Button>
    )
    expect(renderedComponent.contains(children))
  })
  it("should render a <button> element", () => {
    const renderedComponent = shallow(
      <Button {...props}>
        Test
      </Button>
    )
    expect(renderedComponent.is("button")).toEqual(true)
  })
  it("should handle click events", () => {
    const onClickSpy = expect.createSpy()
    const renderedComponent = shallow(
      <Button {...props} onClick={onClickSpy}>
        Test
      </Button>
    )
    renderedComponent.find("button").simulate("click");
    expect(onClickSpy).toHaveBeenCalled();
  })
})
```

And now to use your new `Button` component, all you have to do is import it and pass in the `type` and `children` at a minimum.

Login and Signup Forms

- Form components
- Login and signup views

It is notoriously difficult to create reusable `Form` components. That's because there is so much variation in the inputs and what you end up doing with them. Since we will only have a few forms, we are going to make each one separately. If we end up creating lots of forms with identical characteristics, we can refactor them out later.

Go ahead and create `Signup` and `Login` directories and populate them with the standard files.

```
$ mkdir app/components/{Signup,Login}
$ touch app/components/Signup/{README.md,index.js,style.css,spec.js} \
  app/components/Login/{README.md,index.js,style.css,spec.js}
```

Then we should create the `Signup` component in our `index.js` file.

/app/components/Signup/index.js
commit: coming soon

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

import { default as Button } from "../Button"

export class Signup extends React.Component {
  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          <div className={style.inputGroup}>
            <input
              placeholder="Username"
              className={style.input}
              type="text"
              id="signup-username" />
          </div>
          <div className={style.inputGroup}>
            <input
              placeholder="Email"
```

```
              className={style.input}
              type="text"
              id="signup-email" />
          </div>
          <div className={style.inputGroup}>
            <input
              placeholder="Password"
              className={style.input}
              type="password"
              id="signup-password" />
          </div>
          <div className={style.inputGroup}>
            <input
              placeholder="Verify Password"
              className={style.input}
              type="password"
              id="signup-verify-password" />
          </div>
          <Button
            style={{ width: "100%" }}
            type="primary">
            Submit
          </Button>
        </div>
      </div>
    )
  }
}

export default cssModules(Signup, style)
```

There are many ways to style inputs. If you have a preference for another `input` style, by all means, use that. For now, we're just going to implement some basic styling to make our form more usable. Within `style.css`, add the following:

```
/app/components/Signup/style.css
commit: coming soon
```

```
.wrapper {
  display: flex;
  flex-flow: column nowrap;
  justify-content: center;
  padding-bottom: 2rem;
  width: 400px;
}

.form {
  display: flex;
  flex-direction: column;
  justify-content: center;
  padding: 2rem;
  width: 400px;
}

.input {
  padding: 1rem 1rem;
  border-radius: 3px;
  border: 1px solid #ccc;
  font-size: 1.1em;
  outline: none;
}

.inputGroup {
  display: flex;
  flex-flow: column nowrap;
  padding: 10px 0;
}
```

To get this to render, let's quickly change our `Home` component to the code below.

```
/app/components/Home/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

import { default as Signup } from "../Signup"

export class Home extends React.Component {
  render() {
    return (
      <div>
        <Signup />
      </div>
    )
  }
}

export default cssModules(Home, style)
```

We should also create the `Login`, which will be almost identical but with fewer fields.

```
/app/components/Login/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

import { default as Button } from "../Button"

export class Login extends React.Component {
  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          <div className={style.inputGroup}>
            <input
              placeholder="Email"
              className={style.input}
              type="text"
              id="signup-email" />
          </div>
          <div className={style.inputGroup}>
            <input
              placeholder="Password"
              className={style.input}
              type="password"
              id="signup-password" />
          </div>
          <Button
            style={{ width: "100%" }}
            type="primary">
            Submit
          </Button>
        </div>
      </div>
    )
  }
}

export default cssModules(Login, style)
```

Also add the same styles to the `style.css` file. At some point in the future, we can write code that is more DRY with shared CSS, but for now, let's do it the easy way and just copy-paste.

We will eventually want to submit our form, so let's create a `submit` function that we will attach to our button.

In each of our forms, add a function and update our submit button to handle a click event.

```
/app/components/Signup/index.js
commit: coming soon
```

```
...
export class Signup extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }

  submit(e) {
    console.log("Submit button clicked")
  }

  render() {
    return (
      <div className={style.wrapper}>

        ...

        <Button
          onClick={this.submit}
          style={{ width: "100%" }}
          type="primary">
          Submit
        </Button>
      </div>
    )
  }
}
...
```

Add the same `submit` function and `onClick` event to our `Login` component. Be sure to bind the `submit` function in the constructor as well.

Note: Whenever you write a function that needs access to the `this` context, you need to bind it--usually in the constructor. For example, a component that simply returns some jsx does not need to be bound, while a function that takes in a value from an event triggered by another element does need to be bound.

Now when click on the button, we see a log in our console that says "Submit button clicked". We will eventually use this button to submit the form, but for now, we are just going to use it to write our tests.

Unit tests

There's not a lot that we need to worry about as far as unit tests go at this point. We need to make sure that 1. The form renders, 2. The form has a `submit` function, and 3. The submit button calls the submit function.

Remember, since our components are just classes, we can instantiate them by simply calling `new <Component>` and immediately have access to all their functions to test.

/app/components/Login/spec.js
commit: coming soon

```
import React from "react"
import expect from "expect"
import { shallow } from "enzyme"

import { Signup } from './'

describe('<Signup />', () => {
  it('should render', () => {
    const renderedComponent = shallow(
      <Signup />
    )
    expect(renderedComponent.is('div')).toEqual(true)
  })
  it('should have a submit function', () => {
    const component = new Signup()
    expect(component.submit).toExist()
  })
  // TODO need update
  // it('should call submit function when button is clicked', () => {
  //   const renderedComponent = shallow(
  //     <Signup />
  //   )
  //   const spy = expect.spyOn(renderedComponent.instance(), 'submit')
  //   renderedComponent.find('button').simulate('click')
  //   expect(spy).toHaveBeenCalled()
  // })
})
```

The first new test is really simple. Since our components are just JavaScript classes, we can don't actually have to render them to test certain parts. We're checking to make sure there is a `submit` function, and all we need to do to run this test is instantiate the class using `new`, then checking that the `submit` function exists on that `class`. This is just plain-old JavaScript.

You probably noticed the use of a `spy` in the last test. It would behoove you to learn more about spies for the purpose of writing better tests. A good place to start is the expect documentation to see a few use cases.

Now that we have our form in place, we need to submit it. We will cover that in the next lesson.

Styling the homepage

The last thing we should do is add some basic styling to our `Home` as well as our `Login` component in the event the user already has an account. We're also going to add an image that we can absolutely position behind our input.

/app/components/Home/index.js
commit: coming soon

```
...

import { default as Signup } from "../Signup"
import { default as Login } from "../Login"

export class Home extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      formState: "signup"
    }
  }

  render() {
    return (
      <div className={style.leader}>
        <h1 className={style.title}>Phoenix Chat</h1>
        {this.state.formState === "signup" ? <Signup /> : null}
        {this.state.formState === "login" ? <Login /> : null}
        <img
          role="presentation"
          className={style.circles}
          src="https://s3.amazonaws.com/learnphoenix-static-assets/images/circles-full.png"
      </div>
    )
  }
}

export default cssModules(Home, style)
```

You may have also noticed the `role="presentation"`, which is a replacement for the `alt` text for images where it does not make sense to add `alt` text. In this case, since this image is just going to be a background image, we don't need `alt` text.

And we should center the content to make it look nicer:

/app/components/Home/style.css
commit: coming soon

```
.leader {
  height: 100vh;
  display: flex;
  position: relative;
  align-items: center;
  justify-content: center;
  flex-flow: column nowrap;
  overflow: hidden;
  z-index: 100;
  background: white;
}

.title {
  font-size: 1.8em;
  color: rgb(239, 95, 78);
}

.changeLink {
  cursor: pointer;
  color: #42A5F5;
}

.circles {
  position: absolute;
  right: 0;
  left: 0;
  top: 0;
  margin: auto;
  height: 650px;
  opacity: 0.05;
  z-index: -1;
  transform: scale(3);
}
```

The only tricky piece here is how we're handling the image. We're reducing the opacity and scaling it up by an order of 3. Then we're positioning it in the center and putting the `z-index` at -1 so it stays behind our form.

It's not good practice to have random `z-index` numbers all over the place, but in this instance, it doesn't matter. A common way to do this is to have something like 10 built-in `z-index` values at 100, 200, ..., 1000 and use whichever makes the most sense for the current element.

For now, if you want to access the `Login` component, you can change the hard-coded `formState` to "login". In the future, this will be handled programmatically.

Connect the API

- Making an API call with `fetch`
- Creating users from the frontend

We now have a functional backend that can take `POST` requests and create a user. That user can't sign in yet, but we can at least create one and check the Phoenix logs to make sure it worked. We're going to use the `Signup` component to create a user and send that user to our Phoenix API.

We have not yet implemented Redux, so we are just going to make the API call locally from within our component. This is not how we are going to make these calls in the long-run, but it is the simplest way to make these calls and demonstrate that our frontend and backend are talking to each other.

We're also going to create a `Chat` component that we will use extensively in later lessons as our admin panel.

Making HTTP calls

The first thing we should do is add an HTTP client. There are a lot of options, but we're going to use fetch, which is a polyfill for the soon-to-be native promise-based `fetch` API. Many browsers already support `fetch` without a polyfill.

What `fetch` will allow us to do is make an HTTP call to get a resource from a server--in our case, the Phoenix API.

```
$ npm install --save-dev whatwg-fetch
```

Since we only have signup working on our server, we only need to worry about sending a request to create a user. If you go back to the Phoenix backend and run `mix phoenix.routes`, you'll see a route that looks like the following.

```
$ mix phoenix.routes

...
user_path  POST    /api/users      Firestorm.UserController :create
...
```

This is telling us that we need to send a `POST` request to `localhost:4000/api/users` to create a new user. Let's head over to `Signup` to add `fetch` and get our form ready to make a call to our backend.

New browsers have `fetch` as a native function, but some do not. In order to accommodate those older browsers, we need to add fetch to our webpack configuration. We're going to add it as an entry point, so all our requests get routed through the `whatwg-fetch` polyfill before bundling.

```
/webpack.config.js
commit: coming soon
```

```
...

module.exports = {
  devtool: "eval",
  entry: [
    "whatwg-fetch",
    "webpack-dev-server/client?http://localhost:3000",
    "./app/index"
  ],
  ...
}
```

Remember you must now restart your server. Any time you make a change to your webpack configuration, you must restart your server.

Then we can use `fetch` globally without worrying about old browsers. Let's update our submit function to use `fetch` to make a request to our API endpoint (explanation below the code).

```
/app/components/Signup/index.js
commit: coming soon
```

```
...

  ...
  submit() {
    fetch("http://localhost:4000/api/users", {
      method: "POST",
      headers: {
        Accept: "application/json",
        "Content-Type": "application/json"
      },
      body: JSON.stringify({ test: "not going to work" })
    })
    .then((res) => { return res.json() })
    .then((res) => {
      console.log(res);
    })
    .catch((err) => {
      console.warn(err);
    })
  }
  ...
```

What we're doing in the code above is setting our `Button` to call the `submit` function when it's clicked. Then we call `fetch` with a `POST` method to our API endpoint with an object that won't work.

Because `fetch` is promise-based, we have access to `.then` and `.catch`. Anything within a `.then` function is run if the HTTP call was successful and anything within `.catch` is run if the call was a failure. Since we're sending a bad request, we should expect our `console.log` within the `.catch` to run.

One thing worth noting is that we are *chaining* our promises to include a step that pulls out the `JSON` that we receive from our server (`return res.json()`). Many HTTP clients do this for you, but `fetch` does not.

We are also sending with `headers` that tell our backend that we are sending `json` and that we are expecting `json` in return. But we can't just send raw `json`, so we need to use `JSON.stringify()` to turn our object into a string so we can pass that string to our server.

Go ahead and open the signup modal and click the submit button.

Now check your browser console and you should see an error with the value `POST http://localhost:4000/api/users 400 (Bad Request)`. Great! That's what we wanted.

You should also check your Phoenix terminal, where you should see output along the lines of the following.

```
[debug] Simple CORS request from Origin 'http://localhost:3000' is allowed
[debug] Processing by PhoenixChat.UserController.create/2
  Parameters: %{"test" => "not going to work"}
  Pipelines: [:api]
  ...
```

That's exactly the error we wanted to see. Phoenix was expecting the `user` key, but got `test` instead.

So now let's connect our submission with the form and submit a `user` to Phoenix. Remember, we will eventually use controlled components, but since we just want to get something up and running as quickly as possible, we're just going to pull the values straight from the inputs.

We can do that by changing our `submit` function to the following.

/app/components/Signup/index.js
commit: coming soon

```
  ...
  submit() {
    const user = {
      username: document.getElementById("signup-username").value,
      email: document.getElementById("signup-email").value,
      password: document.getElementById("signup-password").value
    }
    fetch("http://localhost:4000/api/users", {
      method: "POST",
      headers: {
        Accept: "application/json",
        "Content-Type": "application/json"
      },
      body: JSON.stringify({ user })
    })
    .then((res) => { return res.json() })
    .then((res) => {
      console.log(res);
    })
    .catch((err) => {
      console.warn(err);
    })
  }
  ...
```

We're going to find the elements with the `id` that we want and extract the value. We put all those values into a new `user` object, and then pass that object into our `post` call as a parameter.

Recall that in ES6, if the value in an object is the same as the key, you can simply pass a single variable.

So in the example above, we are passing `user` instead of `user: user`.

Go ahead and fill out the form and press the button.

In your browser console, you should see `status: 201` and `statusText: "Created"`. And if you check your Phoenix output, you'll see it was successful there as well.

```
Response {type: "cors", url: "http://localhost:4000/api/users", status: 201, ok: true, stat
```

And if you check your browser console, you should see an object with the key `data` that contains the `email`, `id`, and `username` of the username you just created.

And that's all there is to it. Now we have to go back to Phoenix to give our users the ability to log in. But before we do that, let's create a new `Chat` component that will sit behind our user authentication, as well as a bit more styling.

Chat component

Let's create a new `Chat` component that will function as our admin interface for our app.

```
$ mkdir app/components/Chat
$ touch app/components/Chat/{index.js,spec.js,style.css,README.md}
```

Then within `Chat/index.js`, add the following:

```
/app/components/Chat/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

import { default as Sidebar } from "../Sidebar"

export class Chat extends React.Component {
  render() {
    return (
      <div>
        <Sidebar />
        <div className={style.chatWrapper}>
          chat me
        </div>
        {this.props.children}
      </div>
    )
  }
}

export default cssModules(Chat, style)
```

And add the wrapper to the `style.css` file.

```
/app/components/Chat/style.css
commit: coming soon
```

```
.chatWrapper {
  margin-left: 300px;
}
```

And add a simple test to make sure it renders properly.

```
/app/components/Chat/spec.js
commit: coming soon
```

```
import React from 'react'
import expect from 'expect'
import { shallow } from 'enzyme'

import { Chat } from './'

const props = {}

describe('<Chat />', () => {
  it('should render', () => {
    const renderedComponent = shallow(
      <Chat {...props} />
    )
    expect(renderedComponent.is('div')).toEqual(true)
  })
})
```

So now we have the ability to create users, but they can't sign in. We're also handling our API calls in our local component, which is bad practice, so it's about time we introduce Redux.

Basics of Redux

- Redux and Flux
- `actions`, `reducers`, and `store`

The next step is to log a user in after signup. We're going to handle this with Redux, so now is a good time to go over the basics. If you are not already familiar with Redux, I highly recommend checkout out these 30 free videos sponsored by egghead.io. If you are familiar with Redux, skip ahead to the "Connecting Redux" section.

What is Redux?

To really understand Redux, you must first understand `state`. Think of `state` as the current status of your app. Imagine you're Facebook. You are currently logged in, you have two chat windows open, and you have just launched a modal to upload a new picture.

To generalize, the state of your app could be:

```
{
  currentUser: "username",
  chat: [
    { with: "friend1"},
    { with: "friend2"}
  ],
  modal: true,
  imageUrl: false
}
```

If you wanted to replicate the exact status of this page at this point in time, all you have to do is set your `state` to all of the necessary conditions. This is extremely powerful, especially when used in conjunction with Redux.

Redux consists of `actions`, `reducers`, and a `store`. The concepts around `actions` and `store` are easy to understand--the `reducers` are a little bit more complicated, but still not especially difficult, and it will become much more clear once we start implementing them.

The basic concept is that Redux keeps your `state` at the highest level of your app, and that every component within your app gets its state from that high-level container, whereas a normal React app will have local state.

Basics of Flux

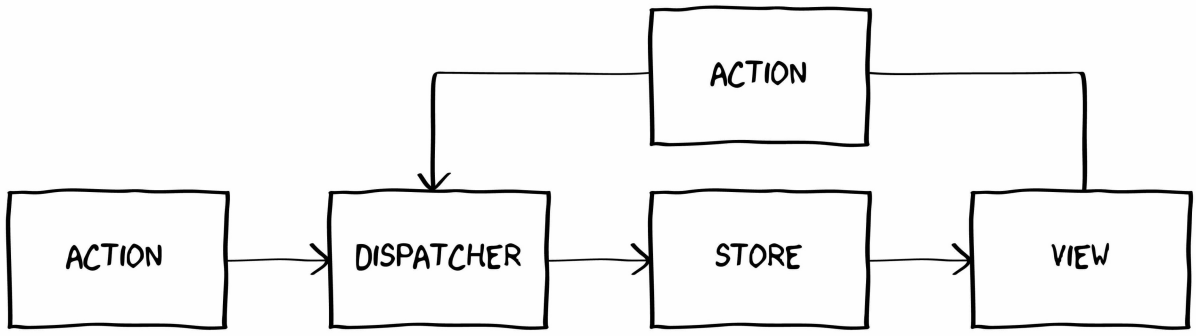
Before we go into Redux, we should really discuss Flux. Flux is one of the major innovations behind React, and it enforces uni-directional data flow.

Rather than go into detail here, check out this video where Facebook explains the origins of Flux and how it works.

Back in the old days of the internet, we used something jQuery to trigger events that would directly manipulate the DOM. For example, we might have a button in one part of our app that appends a new item to a list, or toggle a class on a particular DOM element.

These events were not tied together in any meaningful way, and any data these components received came from sporadic sources. Tracking down the origin of certain events and data was notoriously difficult.

Flux is an attempt at solving that problem. Rather than allowing your models to send data to many views, Flux enforces hierarchy so your views cannot directly make changes to your data.



We will be using Redux, which is a library that is an implementation of Flux.

Working with data

Every interaction our app will have with our server will be through Redux, and all of our data is passed in to our components through Redux, so this is a good time to talk about how to handle data more generally.

A major long-term problem with web APIs, and one not solved by Redux or Flux, is "over-fetching". Twitter is an example of this, because their API returns a deeply-nested object which always contains more data than your app needs.

GraphQL and Falcor are different approaches of solving this problem. They are more efficient, but require more boilerplate to set up.

GraphQL allow you to tell the server specifically what you want and return only that. It is also much more declarative and the server figures out the most efficient way to prepare your data. It also takes more boilerplate code and is only really necessary for apps that have such scale that they need to use it.

At this point in our app, GraphQL is overkill, and we'll just stick to a simple JSON API. Refactoring to use GraphQL and Relay is very doable and we can cross that bridge if it becomes necessary.

With a JSON API, you send a `request` to a server at a specific url with the parameters that the API needs to complete the request. An example request might look like the code below, where we are simply requesting data from `http://github.com/data.json`.

```
fetch("http://github.com/data.json", {
    method: "GET"
  }).then(function(res) {
    // Response here
  }).catch(function(err) {
    // If error, error here
  })
```

You've probably heard a lot about "REST" APIs. People will argue endlessly about the *real* definition of what it means to be "RESTful", and that's because REST does not have an official definition--it's a description of an architecture style, not a protocol.

But the important part is that REST APIs are stateless, so you pass the state to the API as parameters. For example, if you want to create a new user with a REST API, you pass the user details as the parameters to the endpoint. Most of the APIs you've dealt with are REST APIs, so the concept should be at least implicitly familiar.

```
fetch("http://learnphoenix.io/create", {
    method: "POST",
    body: "first=Alan&last=Turing&password=badpassword1"
  }).then(function(res) {
    // Response here
  }).catch(function(err) {
    // If error, error here
  })
```

Our Phoenix backend is a REST API, and we will use Phoenix to handle requests that we send from our frontend, as we have already demonstrated when creating a user.

Basics of Redux

Redux is Flux-ish, with some additional features. The code behind Redux is surprisingly small, and it's really just a well-thought-out design pattern. The main principle of Redux is that you don't keep local state in your components--all of your state is held in one place at the highest-possible level of your app and passed down.

The way you use it is to wrap your entire app in a Redux `Provider` (explained in the next lesson) so that you can later connect each of your React components to this global data source (called a `store`).

The `store` is dumb. It's just an object that holds your state.

In order to process the data and make any changes to your data, you need to pass it through a series of `reducers`. These reducers are actually quite simple and `reduce` is an important function in functional programming.

If you are not familiar with what a `reducer` does, think of it as taking a long list of information and applying some transformation to each item in order. For example, say you have a list of numbers:

```
var list = [3, 1, -3, 5, 1]
```

If you wanted to combine all of these values into a single value by adding them you could write a `for` loop to go over each of them, but that would not be very efficient. Instead, you can write a reducer that takes in the `previous` and `current` values as parameters and combines them within the `reduce` function.

```
var list = [3, 1, -3, 5, 1]

var output = list.reduce(function(previous, current) {
  return previous + current
})
console.log(output) // -> 7
```

Now, think of this in terms of Redux. Let's say we have a list of actions:

```
var list = [
  { modal: open },
  { modal: close },
  { modal: open },
  { chat: {
      name: "Eric",
      isOpen: true
    }
  },
  { chat: {
      name: "Eric",
      isOpen: false
    }
  },
  { chat: {
      name: "John",
      isOpen: true
    }
  }
]
```

What is the current status of this app? Is the modal open or closed? Who has an active chat window? As you will see when we start building our reducers, we can `reduce` a list of actions to the current state of the application.

Reducers listen to `actions` and mutates some piece of state based on the action.

For example, in a "todo" app, the `action` might be something like "create a new list item" or "mark as completed". The action passes any relevant data to the reducer, which changes the current state of the app to reflect the change that the action wanted made. "Mark as completed", for example, would change the specific item in your state and change the `completed` key to `true`.

Whenever the old state does not equal the new state, Redux will fire all of the event listeners and update all of the data in your app that was listening for something that got changed. This means that if you changed something in the todo list, your profile page would not necessarily need to be updated because it does not contain any data related to the todo list.

`action` is a way to trigger those reducers. They are, by default, synchronous. Since much of what you will be doing in a web app is asynchronous, you will quickly need some additional plugins, such as `thunk` and potentially other middleware.

Actions allow you to be more declarative. If you are not familiar with the difference between declarative and imperative, `declarative` means saying what you want, while `imperative` means describing how you will do it. For example, a declarative approach to data fetching would be something like GraphQL, in which you tell your backend what data you want and GraphQL sorts out the details of how to get that data for you.

Setting up Redux

The first thing we need to do is add `redux` as a dependency, as well as several other Redux-related dependencies.

```
$ npm install --save redux react-redux react-router-redux
```

The `react-redux` module binds Redux to React and is necessary to get Redux working with React. The `redux` module contains all of the core functionality that we need. The `react-router-redux` wraps our `react-router` so we can keep track of the state of our routes--without this, our routes are considered a side-effect.

We're going to keep all of our Redux-related files in a new `app/redux` directory. Eventually we will have too many actions and reducers to fit in a single file, but that is easy to refactor and we can do that when it becomes necessary.

```
$ mkdir app/redux
```

Within this directory, we will have three files: `actions.js`, `reducers.js`, and `store.js`.

```
$ touch app/redux/{actions,reducers,store}.js
```

We will go over each of these in more detail in the next section.

Actions, Reducers, and Store

- Proof of concept
- Connect to API

In this section, we create our actions, reducers, and our store. From there, we use our action to make a call to our API and use our reducer to pass that data to our connected components.

The action

The easiest place to start is to create an action. Since we already have the functionality to create and log in users, we will create two actions that we will use to handle our users. We will go over the syntax of the actions below the code.

```
/app/redux/actions.js
commit: coming soon
```

```
const Actions = {}

Actions.userNew = function userNew(user) {
  return {
    type: "USER_NEW",
    payload: {
      user
    }
  }
}

Actions.userLogin = function userLogin(user) {
  return {
    type: "USER_LOGIN",
    payload: {
      user
    }
  }
}

export default Actions
```

First we are declaring an `Actions` object in which we will add all of our actions. This allows us to export

a single object that contains all of our actions, so we can later call these actions with something like `Actions.userNew` after an import statement `import Actions from './redux/actions'`.

Then we define our first action. Every action must have a `type`. That is the only required field. The next field passes in our data. It is common to call this something like `payload`, but you can use whatever term you want. I would avoid using the keyword `data` because our Phoenix server responds with `data`, unless you want to end up accessing your data by calling something like `data.data.data.user`.

Within our `payload`, we are passing in a `user` object that we get from the component that dispatched this action (probably a form of some sort). This value gets passed to our `reducer` which uses it to change the state of our app.

For now, we are just doing a proof of concept, so we aren't going to make an API call to register and log in our user. We will just use our action to set the state of our user in our global Redux `store`.

You will see in the docs for Redux that people declare their actions as string constants, like the code below.

```
const USER_LOGIN = "USER_LOGIN"
```

This is not necessary and we will not be doing this because it's more effort than it's worth at this point in our app. That said, if you're building a huge app, there are perfectly legitimate reasons why you would want to put in this extra work.

If you have a variable called `userNew` but you called `newUser` somewhere in your app, it will fail, but it will fail silently. This is because your app will compile properly and run *up until* you try to dispatch that action, at which point it will cause a run-time error.

If you used a constant, you would get a compile-time error rather than a run-time error because you'd be calling an undeclared variable, so your code wouldn't work unless you used the right variable. This would prevent you from deploying bad code.

The `reducer`

The reducer is what takes in our actions and turns those into the current state of the app. **Reducers should only make calculations and should never have side effects.** When you make an API call, you must do this from your action, not your reducer.

We are going to use the native `Object.assign` to make our immutable data changes, but you can use a library if you prefer. An example would be lodash. What these functions do is make a copy of our state object in order to keep our data immutable. More on this later.

Now we need to create our reducer. The name of the reducer function is the variable name we will use

when we want to access the value in the store. We are going to store all user-related values in a valiable called `user`, so we will name our reducer `user`.

/app/redux/reducers.js
commit: coming soon

```
import { combineReducers } from "redux"

function user(state = {}, action) {
  switch (action.type) {
    case "USER_NEW":
      return state
    case "USER_LOGIN":
      return Object.assign({}, state, {
        email: action.payload.user.email
      })
    default: return state
  }
}

const reducers = combineReducers({
  user
})

export default reducers
```

Our `user` reducer takes in two values: `state` and `action`. `state` is the previous state of the app before going through the reducer, and `action` is the action it receives from the action we dispatched.

We are going to use an ES6 feature that allows us to set default values to our parameters. We are setting our default value of `state` to an empty object (`{}`), so if `user` has not been given any values before, we will start it off with an empty object.

Next we write a `switch` statement that checks the `action.type`. Recall from the actions we created that each one has a `type`. If the type matches the action we passed, we tell the reducer what to do. In the example above, the `USER_NEW` simply passes along state without any changes, while `USER_LOGIN` assigns the key `email` to the value that we passed in our action's `payload`.

It is important to add a `default` case that passes the state unchanged if there are no other matches. If you do not do this, your app will break if nothing matches, and since we will have multiple reducers, it is inevitable that you will pass actions that don't match one of your reducers.

The `combineReducers` function allows us to declare all of our reducers in separate functions and combine them before we export.

The `store`

Now we need to configure our store to take in that reducer. Within `redux/store.js`, add the following.

```
/app/redux/store.js
commit: coming soon
```

```
import { createStore } from "redux"
import reducers from "./reducers"

const store = createStore(reducers)

export default store
```

The last thing we need to do to set up Redux is tie our store into our router. Within `app/index.js`, we need to wrap our entire app in a `Provider` that passes our store down to each child component. It does this somewhat magically, and all we need to do is import our store and pass it into the `Provider` component.

```
/app/index.js
commit: coming soon
```

```
...
import { Provider } from "react-redux"
import store from "./redux/store"

...

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>
      <Route path="/" component={App}>
        <IndexRoute component={Home} />
        <Route path="settings" component={Settings} />
      </Route>
    </Router>
  </Provider>,
  document.getElementById("root")
)
```

A note on capitalization: One thing to note is that our `Provider` is capitalized. Because it is a component that we are using in `jsx`, if you do not capitalize it, your app will break. Capital letters are more than a convention: they are the rule.

Proof of concept

The last thing we will do is prove that this all works. This is not how we will implement Redux in the final product, but it will show how the pieces all fit together.

Within our `Login` component, let's connect our store to our component. We do this with the `connect` function that `react-redux` gives us. At the top of our component, import `connect` and at the bottom we have to create a new function, described below the codeblock.

Recall that, for now, to see the `Login` form, you have to change the `formState` in our `Home` component to "login".

```
/app/components/Login/index.js
commit: coming soon
```

```
import { connect } from "react-redux"

...

const mapStateToProps = state => ({
  user: state.user
})

export default connect(mapStateToProps)(cssModules(Login, style))
```

Recall that with one-line ES2015 arrow functions implicitly `return` the value after the arrow. So these two are equivalent:

```
const mapStateToProps = state => {
  return {
    user: state.user
  }
}

const mapStateToProps = state => ({
  user: state.user
})
```

The `mapStateToProps` function takes the state of the app from your `store` (recall we set the default value of `user` in our reducer to an empty object `{}`) and maps that over to the `props` of your current component. In this case, we are taking `state.user` from our global `store` and assigning `this.props.user` of our current component to that same value. A more accurate name for this might be

`mapStateOfStoreToPropsOfComponent`, but that's perhaps a bit too verbose.

To show that this is working, let's add a `console.log` within our `render` function before we `return` our component.

```
...
  render() {
    console.log(this.props.user)
    return (
      ...
    )
  }
```

Now, if you change the state of `formState` in your `Home` component to "login", you'll see an empty object in your console. This is because we are currently receiving the default, empty object from our reducer.

```
Object {}
```

The next step is to `dispatch` an action with the values from the form that will allow us to change the value of `this.props.user`.

We do this by importing our `Actions` and using the `dispatch` function that we get when we `connect` our `Login` component using `react-redux`.

```
/app/components/Login/index.js
commit: coming soon
```

```
import Actions from "../../redux/actions"
  ...

export class Login extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }

  submit() {
    const user = {
      email: document.getElementById("signup-email").value,
      password: document.getElementById("signup-password").value
    }
    this.props.dispatch(Actions.userLogin(user))
  }

  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          ...
          <Button
            onClick={this.submit}
            style={{ width: "100%" }}
            type="primary">
            Submit
          </Button>
        </div>
      </div>
    )
  }
}

...
```

Now go ahead and fill out the login form and submit it. You'll see a new log to your console that has the email address you submitted.

```
Object {email: "alan@turing.com", password: "badpassword1"}
```

This worked by taking the values from the inputs, dispatching those values as the `userLogin` action, which hits our `user` reducer, which triggers the `USER_LOGIN` type, which assigns `user` to `{ email: action.payload.user.email }`.

You may recall that when `props` are updated, your component automatically updates. In this case, since

`Login` is connected to `this.props.user`, which was mapped from `state.user`, our `console.log` was called again with the updated value.

If that seems like a roundabout way of updating data, you're not alone, but the long-term benefits of keeping your data flow uni-directional far out-weight the upfront cost of setting up some extra code.

Login and Authentication: Part 1

- Sign in
- Ueberauth and Guardian
- Plugs

In this section, we go over Elixir plugs and add user authentication with Ueberauth and Guardian. We also set up an authentication controller (`AuthController`) to handle requests that need authorization.

Account login

Now that we can register for an account, we'll want to sign in. But what does it mean to "log in"? The question is more complex than most people realize. There are many ways of approaching this problem, but we are going to use a JSON web token (JWT) for user authentication.

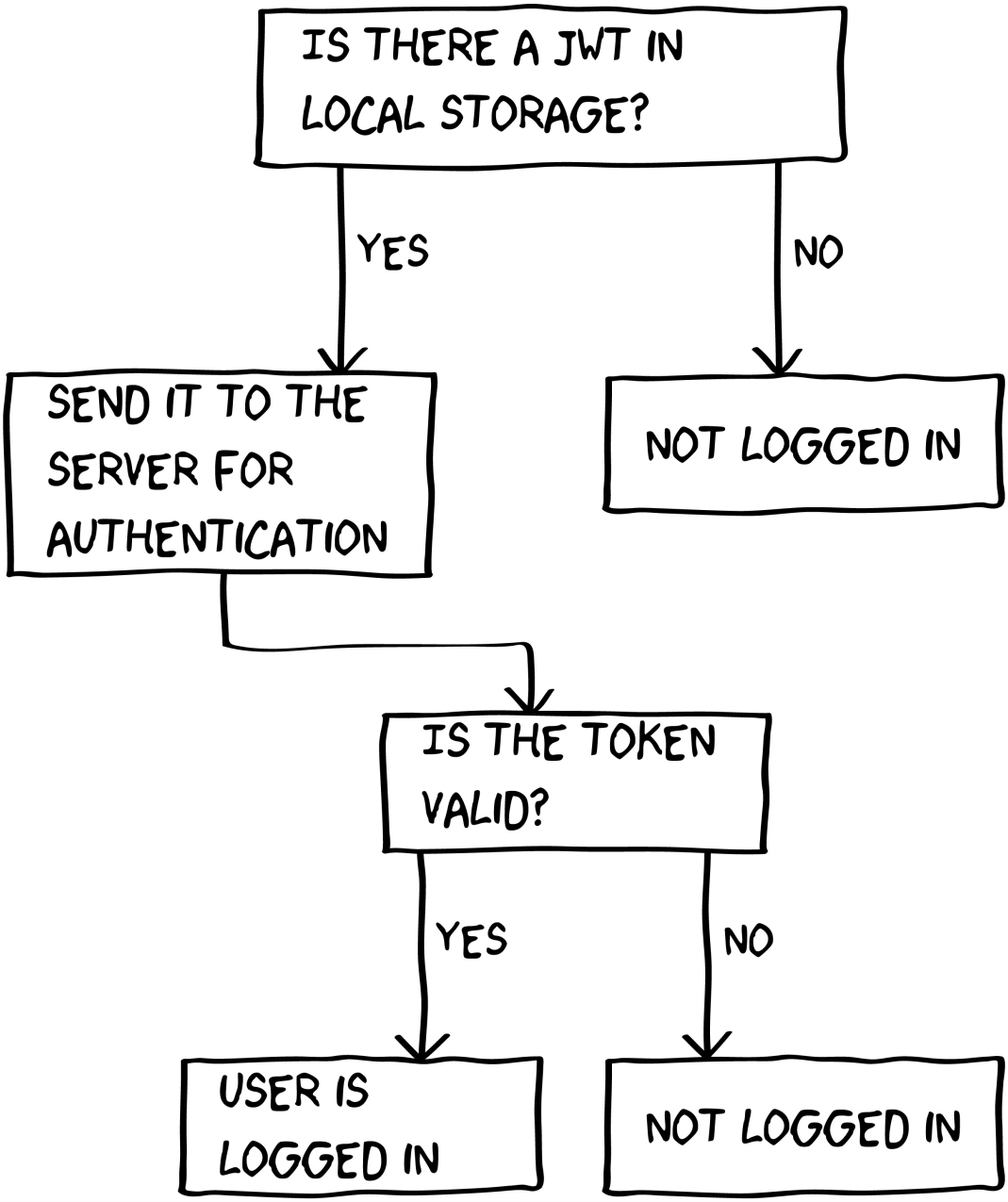
A JSON web token is a random string that our server generates after a user passes in valid credentials (username and password). A JSON web token looks something like this:

```
2b125jQggWD1BFWECdbQyxe4KXVB18Sgi64m9Iu2MgCciljCCsIG
```

When a user passes valid credentials to our server, our server will respond with a JSON web token that we can store in our browser either in `cookies` or `localStorage`. For a user to be considered "logged in", we send that JSON web token in a request to our server asking, "Is this JSON web token still valid?" If the response is in the affirmative, then the user is logged in. If the response is negative, then the user is not logged in.

When a user logs out, we remove the JSON web token from `localStorage` and send a request to our server to invalidate the token to ensure it cannot be used again. This prevents what is called a replay attack where someone uses an old JSON web token to gain access to your account.

The basic strategy we're going to use follows the flow chart below.



For more information on how JSON web tokens work, check out jwt.io.

For authentication we'll work with Überauth (often spelled Ueberauth, without the u-umlaut -- this is how we will refer to it going forward). Ueberauth is an authentication framework that allows us to use many different strategies to authenticate users. For now we'll use Ueberauth Identity to implement our email-password authentication but later we can use Ueberauth Twitter, Ueberauth Facebook, and several others.

In addition to Ueberauth and Ueberauth Identity, we'll use Guardian to generate our JSON web tokens. If you're not familiar with JSON web tokens, we will cover them in greater detail later.

If you're interested to see how Ueberauth works with multiple strategies, take a look at the Ueberauth Example project provided by the Ueberauth team.

Let's go ahead and add `ueberauth`, `ueberauth_identity`, and `guardian` to our project's dependecies in `mix.exs`:

/mix.exs
commit: coming soon

```
defp deps do
  [
    {:comeonin, "~> 2.3"},
    ...
    {:ueberauth, "~> 0.2"},
    {:ueberauth_identity, "~> 0.2"},
    {:guardian, "~> 0.10"}
  ]
end
```

We need to add both `ueberauth` and `ueberauth_identity` to our project's applications:

/mix.exs
commit: coming soon

```
def application do
  [mod: {PhoenixChat, []},
    applications: [
      :comeonin,
      ...
      :ueberauth,
      :ueberauth_identity
    ]]
end
```

Let's run `mix deps.get` and get our new dependencies:

```
$ mix deps.get
```

Configuring Ueberauth

The next thing to do is configure Ueberauth. Within `config/config.exs`, add the following:

/config/config.exs
commit: coming soon

```
config :ueberauth, Ueberauth,
  providers: [
    identity: {
      Ueberauth.Strategy.Identity,
      [callback_methods: ["POST"]]
    }
  ]
```

This is taken straight from the docs, but what it is doing is telling Ueberauth that we are using the `identity` strategy to log in users with a username-password combination. At some point later on, we can add Twitter and Facebook login to the `providers` list.

Configuring Guardian

While we're in `config/config.exs` let's add our Guardian configuration:

```
/config/config.exs
commit: coming soon
```

```
config :guardian, Guardian,
  issuer: "PhoenixChat",
  ttl: {30, :days},
  secret_key: "uw/27wdrIquPn2fktwfJg9tg8qOl5ysTPCFjISw1TCCaLlfWgRUAea1SuWcfERzX",
  serializer: PhoenixChat.GuardianSerializer,
  permissions: %{default: [:read, :write]}
```

Before we go any further, let's generate a new `secret_key` (it's not very secret otherwise). We can either come up with a random string or we can use the handy generator provided by Phoenix:

```
$ mix phoenix.gen.secret
```

Copy and paste the output into the `secret_key` value above and we're done. For now storing the `secret_key` in the file is fine but later we will move this into a system environment variable; relying on our system's environment lets us remove passwords and secret keys from our code and source control, limiting its exposure.

The last step of our Guardian configuration is to create the `PhoenixChat.GuardianSerializer` we specified. We'll put this file in a new directory within the `web` directory:

```
$ mkdir web/auth
$ touch web/auth/guardian_serializer.ex
```

Let's open the serializer at `web/auth/guardian_serializer.ex` and update the file to look like this:

/web/auth/guardian_serializer.ex
commit: coming soon

```
defmodule PhoenixChat.GuardianSerializer do
  @behaviour Guardian.Serializer

  alias PhoenixChat.{Repo, User}

  def for_token(%User{id: id}), do: {:ok, "User:#{id}"}
  def for_token(_), do: {:error, "Unknown resource type"}

  def from_token("User:" <> id), do: {:ok, Repo.get(User, id)}
  def from_token(_), do: {:error, "Unknown resource type"}
end
```

We don't need to dig into this too much as this is verbatim from the Guardian documentation. Once we start using our authentication system, we can come back to this and it will make more sense. The serializer is what ties Guardian in with our user systems. Guardian relies on the `for_token/1` function to convert a given `%User{}` into a token. The `from_token/1` function does the opposite, retrieving our user from a given token.

A quick thing to note is `@behaviour`. Elixir is built on Erlang, which uses the British version of some English words. Just keep that in mind going forward.

The next few steps require a deeper dive into Phoenix.

Plugs

In many ways, every app is just a series of functions. React has popularized this approach on the frontend, each function either transforms data or triggers a side-effect like rendering to the DOM or a change to the database.

Plug is a library that takes a connection and returns a modified version of the connection. With plugs you can authenticate users, validate requests, send HTTP responses, sanitize inputs, and trigger background work. Throughout this course, we'll cover these and many other uses for plugs.

Since plugs are so flexible it's no surprise that our Phoenix app is nothing more than a series of plugs.

Creating our AuthController

For this controller we're not going to use a generator. Instead, we're going to create it manually so we can take a closer look at the different components.

The way the Phoenix directory structure is set up everything web-related lives within the `web` directory. Since we are building a web API, most of what we are doing will be within this directory.

Let's create our auth controller:

```
$ touch web/controllers/auth_controller.ex
```

Now update the new file to look like this:

```
/web/controller/auth_controller.ex
commit: coming soon
```

```
defmodule PhoenixChat.AuthController do
  use PhoenixChat.Web, :controller
  plug Ueberauth

end
```

Here we define our new module and include some helper methods from the `PhoenixChat.Web` module for controllers. We've also included a plug for Ueberauth in our controller. When requests come into our controller, they will be routed through the Ueberauth plug. We'll revisit the Ueberauth plug shortly.

Note on debugging in Elixir: In Elixir, you can log to the terminal with `IO.puts` or `Logger.debug` after adding `require Logger`. This is one of the primary means by which you can debug your apps. It's identical to `console.log()` if you're a JavaScript developer or `puts` if you're a Ruby developer.

In fact, we will keep this controller mostly empty for the time being and just add a log to show that it was hit. This will cause an error, but it will show that the request was hit. Change your `AuthController` to the following:

```
/web/controller/auth_controller.ex
commit: coming soon
```

```
defmodule PhoenixChat.AuthController do
  use PhoenixChat.Web, :controller

  def test(conn, _params) do
    IO.puts "AuthController called!"
    conn
  end
end
```

Open `web/router.ex` and make the following changes:

```
/web/router.ex
commit: coming soon
```

```
...

  scope "/api", PhoenixChat do
    pipe_through :api

    get "/auth", AuthController, :test
    resources "/users", UserController, except: [:new, :edit]
  end

...
```

For now we want to direct anyone who runs a `GET` request to `/api/auth` to be routed to the `AuthController` and the `test/2` function we created.

To try out our new route, start your server:

```
$ mix phoenix.server

Generated PhoenixChat app
[info] Running PhoenixChat.Endpoint with Cowboy using http on port 4000
```

Open up a browser and visit `localhost:4000/api/auth`. We won't see anything in the browser because we aren't telling Phoenix to render any HTML, but if you head over to your terminal (the one in which you ran `mix pheonix.server`), you should see the output "AuthController called!" just before the error.

```
[info] GET /api/auth
AuthController called!
[debug] Processing by PhoenixChat.AuthController.test/2
  Parameters: %{}
  Pipelines: [:api]
  ...
```

This is a contrived example, but you can see the following happen:

```
connection
|> router
|> api_pipeline
|> AuthController.test
```

The connection was sent through a series of functions and transformations before finally being returned to the user.

Login and Authentication: Part 2

- Authentication
- Automatic login on signup

Now that we have the concept, it's time to implement this for our user login. In this section, we add an authentication pipeline for routes that need authentication and we add an endpoint that allows our frontend to login.

Adding authentication

Let's make our `AuthController` a bit more useful and finish setting up Ueberauth and Guardian. Following along with their documentation, let's update our router:

```
/web/router.ex
commit: coming soon
```

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

  ...

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

  pipeline :api_auth do
    plug Guardian.Plug.VerifyHeader, realm: "Bearer"
    plug Guardian.Plug.LoadResource
  end

 ...

  scope "/api", PhoenixChat do
    pipe_through :api

    resources "/users", UserController, except: [:show, :index, :new, :edit]
  end

  scope "/auth", PhoenixChat do
    pipe_through [:api, :api_auth]

    post "/:identity/callback", AuthController, :callback
  end
end
```

In addition to creating an "/auth" scope, we created a new `pipeline`. Our new pipeline takes care to verify our `Authorization` header and load the resource using our serializer. We learned about the serializer when we configured Guardian, so let's discuss the authorization header.

Under-the-hood Guardian relies on an HTTP header, `Authorization`, to pass the JSON web token between the frontend and the backend. If you don't know what an authorization header is, it will become abundantly clear later on when we head back over to the frontend to make a call with an authorization header.

Now we need to implement the callback function we specified. Open up our `AuthController` and change it to include our `callback/2` function. We can also get rid of the `test` function.

```
/web/controller/auth_controller.ex
commit: coming soon
```

```
defmodule PhoenixChat.AuthController do
  use PhoenixChat.Web, :controller

  alias PhoenixChat.{ErrorView, UserView, User, AuthController}

  plug Ueberauth

  def callback(%{assigns: %{ueberauth_auth: auth}} = conn, _params) do
    result = with {:ok, user} <- user_from_auth(auth),
                  :ok <- validate_pass(user.encrypted_password, auth.credentials.other.pass
                  do: signin_user(conn, user)

    case result do
      {:ok, user, token} ->
        conn
        |> put_status(:created)
        |> render(UserView, "show.json", user: user, token: token)
      {:error, reason} ->
        conn
        |> put_status(:bad_request)
        |> render(ErrorView, "error.json", error: reason)
    end
  end
end
```

If we read down the function, what we're doing is: get the user from Ueberauth, validate the password, and sign in the user. If it's successful we'll render the user view along with our JSON web token. In the event of an error, we'll show the error view instead.

Let's implement the individual functions that make up our callback, we'll start with `user_from_auth/1`. For this function we need to lookup a user by the email address provided by Ueberauth. Add the following function to `web/controller/auth_controller.ex`, below the callback.

```
/web/controller/auth_controller.ex
commit: coming soon
```

```
  ...
  defp user_from_auth(auth) do
    result = Repo.get_by(User, email: auth.info.email)
    case result do
      nil -> {:error, %{"email" => ["invalid email"]}}
      user -> {:ok, user}
    end
  end
```

The next step in our authentication process is validating the password. We'll use pattern matching in our

`validate_pass/2` function to handle missing or blank passwords and Comeonin to compare the clear text password and the one stored in the database. To handle the comparison we'll rely on Comeonin and the Bcrypt module.

Bcrypt is the hashing algorithm we used in our `registration_changeset/2` to compute our hashed password. The code below is what we should end up with.

/web/controller/auth_controller.ex
commit: coming soon

```
  ...
  defp validate_pass(_encrypted, password) when password in [nil, ""] do
    {:error, "password required"}
  end

  defp validate_pass(encrypted, password) do
    if Comeonin.Bcrypt.checkpw(password, encrypted) do
      :ok
    else
      {:error, "invalid password"}
    end
  end
```

Lastly, we need to sign in so we can retrieve the user and JSON web token:

/web/controller/auth_controller.ex
commit: coming soon

```
  ...
  defp signin_user(conn, user) do
    token = conn
            |> Guardian.Plug.api_sign_in(user)
            |> Guardian.Plug.current_token
    {:ok, user, token}
  end
```

That's it for the controller but there's still one step left: updating our user view to support the JSON web token. We can start by creating a new function similar to `"user.json"` but including our token. Then we can use pattern matching in `"show.json"` to render the JSON with or without the JWT token.

With all the pieces together, this is how our view should appear:

/web/views/user_view.ex
commit: coming soon

```
defmodule PhoenixChat.UserView do
  use PhoenixChat.Web, :view

  alias PhoenixChat.{UserView}

  def render("index.json", %{users: users}) do
    %{data: render_many(users, UserView, "user.json")}
  end

  def render("show.json", %{user: user, token: token}) do
    %{data: render_one(user, UserView, "user_token.json", token: token)}
  end

  def render("show.json", %{user: user}) do
    %{data: render_one(user, UserView, "user.json")}
  end

  def render("user.json", %{user: user}) do
    %{email: user.email,
      id: user.id,
      username: user.username}
  end

  def render("user_token.json", %{user: user, token: token}) do
    %{email: user.email,
      id: user.id,
      token: token,
      username: user.username}
  end
end
```

One thing our frontend application will need to know, is who current user is. To facilate this, let's build an endpoint to return the user given a JSON web token. This will be our "/me" endpoint, let's go ahead and add the endpoint to the router:

```
/web/router.ex
commit: coming soon
```

```
...

  scope "/auth", PhoenixChat do
    pipe_through [:api, :api_auth]

    get "/me", AuthController, :me
    post "/:identity/callback", AuthController, :callback
    delete "/signout", AuthController, :delete
  end

...
```

Now we can go back to our `AuthController` and implement our `me` function. For now all we want our endpoint to do is retrieve the currently logged in user and return their info to us. If they're not logged in, an HTTP error code will do. To do this, we're going to introduce a new Guardian plug: `EnsureAuthenicated`.

With `EnsureAuthenticated`, Guardian will check to see whether our incoming HTTP request contains our Authentication header. In the event that it does not, Guardian will take care of returning the appropriate error code and won't run any other functions in the module. The first step is to add the plug to the top of our controller:

```
/web/controllers/auth_controller.ex
commit: coming soon
```

```
defmodule PhoenixChat.AuthController do
  use PhoenixChat.Web, :controller

  alias PhoenixChat.{ErrorView, UserView, User, AuthController}

  plug Ueberauth
  plug Guardian.Plug.EnsureAuthenticated, [handler: AuthController] when action in [:delete

  ...
end
```

By now we should be somewhat familiar with what this does but just in case, let's go over it. Here we are adding a new plug to our request pipeline, `EnsureAuthentication`, passing a set of options (namely the error handler), and lastly we use a guard to apply the plug only for requests to our `/me` or `/signout` routes. This is because (obviously) we don't want to ensure a user is authenticated before that user can login.

To get everything working we only need to implement our `me` function now. Since we know `EnsureAuthenticated` will take care of logged out users, our method is pretty simple. We need to retrieve the current user and return them using the `UserView`. Add the following to our `auth_controller`.

/web/controllers/auth_controller.ex
commit: coming soon

```
  ...

  def me(conn, _params) do
    user = Guardian.Plug.current_resource(conn)
    render(conn, UserView, "show.json", user: user)
  end

  ...
```

We're not quite done. According to the Guardian docs for `EnsureAuthenticated`, we will need to add another function in the event a user is not authenticated. From the docs:

> Looks for a previously verified token. If one is found, continues, otherwise it will call the `:unauthenticated` function of your handler.

While we're at it, let's also add an `:unauthorized` function in case we want to use the `EnsurePermissions` plug in the future.

/web/controllers/auth_controller.ex
commit: coming soon

```
  ...
  def unauthenticated(conn, _params) do
    conn
    |> put_status(:unauthorized)
    |> render(ErrorView, "error.json", errors: %{"account" => ["insufficient privilege"]})
  end

  def unauthorized(conn, _params) do
    conn
    |> put_status(:forbidden)
    |> render(ErrorView, "error.json", error: %{"account" => ["unauthorized"]})
  end
```

It's probably worthwhile to at least skim the rest of the Guardian docs so you know about some of the functionality. Knowing what is available and what is required will help a lot when debugging.

Automatic login on signup

Now that we can sign a user up and login independently, let's automatically log a user in upon signup.

This is actually really simple. Since "login" really just means returning a valid token to a user, we can generate a token after an account has successfully been created and pass that along to the user. We only have to change two lines of code.

/web/controllers/user_controller.ex
commit: coming soon

```
def create(conn, %{"user" => user_params}) do
  changeset = User.registration_changeset(%User{}, user_params)

  case Repo.insert(changeset) do
    {:ok, user} ->
      {:ok, token, _claims} = Guardian.encode_and_sign(user, :token)

      conn
      |> put_status(:created)
      |> render("show.json", user: user, token: token)
    {:error, changeset} ->
      conn
      |> put_status(:unprocessable_entity)
      |> render(PhoenixChat.ChangesetView, "error.json", changeset: changeset)
  end
end
```

Now we have everything we need to register for an account and sign-in. Let's head back to the frontend to connect it.

Move Logic to Redux

- Asynchronous actions
- Updating the user

So now that we understand some of the concepts of Redux and have our `store` connected to our app, we can start building our app for real. The first thing we should connect is our user.

We want to be able to:

- create a new user
- log that user in
- authenticate that user
- log that user out.

Since the user is something that we will want to have access to in our entire app, it makes sense to hold this value in the highest-order component we have and pass that value down to all children, which in this case is our `App` component. But since we're using Redux, we can keep that value in our global `store` and pull it out using `connect`.

Asynchronous actions

Before we get much further, configure our app to work properly with Redux. The first thing we should reconfigure is our `Signup` component and the associated action. We want to move our asynchronous request out of our component and into our action, and we want to connect our component to Redux.

But Redux doesn't support asynchronous calls without middleware. That's where we need something like `thunk` comes in. From the docs:

> A thunk is a function that wraps an expression to delay its evaluation.

It's a handy way for us to make asynchronous calls within our actions.

```
$ npm install --save redux-thunk
```

And in order to apply that middleware, we need to add some additional configuration to our `store`. Within `store.js` we need to import a few more functions and add `thunk` to an `applyMiddleware` call. We are also using `compose`, which allows us to create a wrapper function that composes of other

functions in order.

/app/redux/store.js
commit: coming soon

```
import { createStore, compose, applyMiddleware } from "redux"
import thunk from "redux-thunk"
import reducers from "./reducers"

const middlewares = [thunk]

const createStoreWrapper = compose(
  applyMiddleware(...middlewares)
)(createStore)

const store = createStoreWrapper(reducers)

export default store
```

Now that we've applied our `thunk` middleware, we can start running asynchronous actions from our `Actions`. The first one we want to change is `userNew`.

/app/redux/actions.js
commit: coming soon

```
...

Actions.userNew = function userNew(user) {
  return dispatch => fetch("http://localhost:4000/api/users", {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ user })
  })
  .then((res) => {
    return res.json()
  })
  .then((res) => {
    // TODO: More on this later
    console.log(res)
  })
  .catch((err) => {
    console.warn(err)
  })
}
...
```

Here we are returning a function that takes in `dispatch` as a parameter (used by `thunk`) and makes an asynchronous call to our server. If there is an error, we log the error.

We should also change the `userLogin` action to make a call to our backend. If you go to your Phoenix backend and check `mix phoenix.routes` you'll see the route for login (`/auth/identity/callback`).

You'll notice a few things different about this action. For one, we are taking the token we receive from the server and saving it to `localStorage`. If you don't know what localStorage is, you can think of it as the better version of cookies, which are pieces of information you can store on the browser that can only be accessed by the website that sets them. In fact, `localStorage` is supposed to be the replacement for cookies.

After we set the token to `localStorage`, we `dispatch` an action to our `reducer`. Recall that our reducer is already set up to handle a `USER_LOGIN` action.

```
/app/redux/actions.js
commit: coming soon
```

```
Actions.userLogin = function userLogin(user) {
  return dispatch => fetch("http://localhost:4000/auth/identity/callback", {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify({
      email: user.email,
      password: user.password
    })
  })
  .then((res) => { return res.json() })
  .then((res) => {
    /* If success, log the user in */
    localStorage.token = res.data.token
    /* Then send action to reducer */
    dispatch({
      type: "USER_LOGIN",
      payload: {
        user: res.data
      }
    })
  })
  .catch((err) => {
    console.warn(err)
  })
}
```

If you're a security nut, you can be extra careful and store your token in a session cookie with an `httpOnly` flag. This will prevent XSS attacks and side channel (BREACH) attacks. That said, `localStorage` is more common and perfectly fine for our purposes.

Now within our `user` reducer, we want to set the value of our user to the value returned by the server. While we're at it, let's add some default values for our `user` so we don't run into errors later when we assign these values to DOM elements via `props` and also destructure our payload for easier reading.

```
/app/redux/reducers.js
commit: coming soon
```

```
...

function user(state = {
  email: "",
  username: "",
  id: ""
}, action) {
  switch (action.type) {
    case "USER_NEW":
      return Object.assign({}, state, {
        email: action.payload.user.email,
        username: action.payload.user.username,
        id: action.payload.user.id
      })
    case "USER_LOGIN":
      return Object.assign({}, state, {
        email: action.payload.user.email,
        username: action.payload.user.username,
        id: action.payload.user.id
      })
    default: return state
  }
}

...
```

Let's also update our `userNew` function to handle the response from our server.

```
/app/redux/actions.js
commit: coming soon
```

```
...
Actions.userNew = function userNew(user) {
  return dispatch => fetch("http://localhost:4000/api/users", {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ user })
  })
  .then((res) => {
    return res.json()
  })
  .then((res) => {
    /* If success, log the user in */
    localStorage.token = res.data.token
    /* Then send action to reducer */
    dispatch({
      type: "USER_NEW",
      payload: {
        user: res.data
      }
    })
  })
  .catch((err) => {
    console.warn(err)
  })
}
...
```

Connecting the login form

Now that we have everything in place on the backend to handle user login, let's send off the request. Recall that we already set up our `Login` component to send the request via redux--we just previously hadn't sent the http request.

Go ahead and login with valid credentials (the account you created earlier, or a new one) and you'll see the returned username and email from the server.

What's more, if you check `localStorage.token` in your browser console, you'll see your current, valid JSON web token that we will later use for authentication.

Connecting the signup form

Since login is working with Redux, we should change our `Signup` to use the Redux action as well. First import `connect` and `Actions` from Redux, then we need to replace the `fetch` call with a dispatched Redux action and connect our component in our export.

/app/components/Signup/index.js
commit: coming soon

```
import React from "react"
import cssModules from "react-css-modules"
import { connect } from "react-redux"
import style from "./style.css"
import Actions from "../../redux/actions"

...

export class Signup extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }

  submit() {
    const user = {
      username: document.getElementById("signup-username").value,
      email: document.getElementById("signup-email").value,
      password: document.getElementById("signup-password").value
    }
    this.props.dispatch(Actions.userNew(user))
  }

  ...
}

export default connect()(cssModules(Signup, style))
```

Now when you submit your form, you're dispatching the action through Redux instead of through the form component. If you try it with a new username and email, you should see the same successful output in your browser console.

```
> Fetch complete: POST "http://localhost:4000/api/users".
> Object {data: Object}
```

Toggling between signup/login forms

Now is a good time to refactor our `Home` component to allow us to easily toggle between login and signup forms. First, let's create a new function that allows us to toggle the state.

/app/components/Home/index.js
commit: coming soon

```
...
export class Home extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      formState: "login"
    }
    this.setFormState = this.setFormState.bind(this)
  }

  setFormState(formState) {
    this.setState({ formState })
  }
...
}
```

Then let's create a function that determines the content below our form that when clicked will change the state of our form.

/app/components/Home/index.js
commit: coming soon

```
...
export class Home extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      formState: "login"
    }
    this.setFormState = this.setFormState.bind(this)
  }

  ...

  renderToggleContent() {
    switch (this.state.formState) {
      case "login":
        return (
          <div
            className={style.changeLink}
            onClick={() => this.setFormState("signup")}>
            Need an account? Signup.
          </div>
        )
      case "signup":
        return (
          <div
            className={style.changeLink}
            onClick={() => this.setFormState("login")}>
            Have an account? Login.
          </div>
        )
      default: return null
    }
  }

  render() {
    return (
      <div className={style.leader}>
        <h1 className={style.title}>Phoenix Chat</h1>
        { this.state.formState === "signup" ? <Signup /> : null }
        { this.state.formState === "login" ? <Login /> : null }
        { this.renderToggleContent() }
        <img
          role="presentation"
          className={style.circles}
          src="https://s3.amazonaws.com/learnphoenix-static-assets/images/circles-full.png"
      </div>
    )
  }
}
```

One thing that you will often see in React code is the use of `bind` to pass a value to your function using JSX. Now that we have ES2015, we can use arrow functions, which automatically bind to the context `this`. So for example, the two onClick functions below are the same:

```
// ES2015, and what we will use
onClick={() => this.setFormState("login")}>

// ES5
onClick={this.setFormState.bind(this, "login")}
```

Also note that if you need access to the event, you will need to pass it as a parameter if you use ES2015 syntax.

```
// ES2015 syntax, passing event as second parameter
onClick={e => this.setFormState("login", e)}
```

And now, when you click on the link, it toggles the form. Granted, it's not styled, but we can do that later.

Check Login Status

- `auth/me`
- Grant special access

At this point, we have the ability to create an account, log a user in, and get a valid JSON web token. So now we need to check with our server that the token is valid to determine whether or not the user has access to certain views and data.

Setting up `userAuth` action

The way we do this is by sending a request to the `auth/me` route we set up in our Phoenix server. If you recall, the `auth/me` route takes the token from the `Authorization` header and checks to make sure it's valid. If it is, it returns the data we requested (in this case, username and email).

This should look familiar, with the exception of the `Authorization` header line. The standard syntax for sending an `Authorization` header is in the syntax: `Bearer <token>`, as you can see if you search for the word `bearer` in this handy introduction to jwt. If there is no token, it passes an empty string.

```
/app/redux/actions.js
commit: coming soon
```

```
Actions.userAuth = function userAuth() {
  return dispatch => fetch("http://localhost:4000/auth/me", {
    method: "GET",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json",
      Authorization: `Bearer ${localStorage.token}` || ""
    }
  })
  .then((res) => { return res.json() })
  .then((res) => {
    dispatch({
      type: "USER_AUTH",
      payload: {
        user: res.data
      }
    })
  })
  .catch((err) => {
    console.warn(err)
  })
}
```

Then we have to add our `USER_AUTH` action as a `case` to our `user` reducer.

```
/app/redux/reducers.js
commit: coming soon
```

```
  ...
    case "USER_AUTH":
      return Object.assign({}, state, {
        email: action.payload.user.email,
        username: action.payload.user.username,
        id: action.payload.user.id
      })
  ...
```

Automatic authentication

There are a few times in which we would like to automatically check to see if a user is logged in. The first is when the app initially renders, as the user might have come from another session and still has a valid token. The other times we want to automatically log a user in are when we have a change in user status, such as after we run `userNew` or `userLogin`.

Since we want to do this for the whole app, we should run this when the `App` component renders. So the first thing we should do is refactor our App component into it's own directory so we can access its `componentDidMount` function and un-muddle our `app/index.js` file.

```
$ mkdir app/components/App
$ touch app/components/App/{index.js,spec.js,style.css,README.md}
```

Then create our component and `dispatch` our `userAuth` action within `componentDidMount`.

```
/app/components/Apps/index.js
commit: coming soon
```

```
import React from "react"
import { connect } from "react-redux"
import Actions from "../../redux/actions"

export class App extends React.Component {
  componentDidMount() {
    this.props.dispatch(Actions.userAuth())
  }

  render() {
    return (
      <div>
        {this.props.children}
      </div>
    )
  }
}

export default connect()(App)
```

And lets add that functionality to both of our actions as well, since we want to authenticate the user immediately after login and account creation. Remember, when we create an account or login, all we're doing is adding a web token to `localStorage`. We have to use `auth/me` to actually validate that user.

```
/app/redux/actions.js
commit: coming soon
```

```
...
Actions.userNew = function userNew(user) {
  ...
  .then((res) => {
    /* If success, log the user in */
    localStorage.token = res.data.token
    /* Then send action to reducer */
    dispatch({
      type: "USER_NEW",
      payload: {
        user: res.data
      }
    })
    dispatch(Actions.userAuth())
  })
  ...
}

Actions.userLogin = function userLogin(user) {
  ...
  .then((res) => {
    /* If success, log the user in */
    localStorage.token = res.data.token
    /* Then send action to reducer */
    dispatch({
      type: "USER_LOGIN",
      payload: {
        user: res.data
      }
    })
    dispatch(Actions.userAuth())
  })
  ...
}
```

While we're at it, let's write a few tests for this component. We want to make sure that the component:

- renders
- dispatches on `componentDidMount`
- dispatches `Actions.userAuth()` on `componentDidMount`

We will need to import our `Actions` in order to run these tests and make ample use of our `spy`. More details about each test below the codeblock.

```
/app/components/App/spec.js
commit: coming soon
```

```
import React from 'react'
import expect from 'expect'
import { shallow, mount } from 'enzyme'

import Actions from '../../redux/actions'

import { App } from './'

const props = {}

describe('<App />', () => {
  it('should render', () => {
    const renderedComponent = shallow(
      <App {...props} />
    )
    expect(renderedComponent.is('div')).toEqual(true)
  })
  it('calls dispatch on componentDidMount', () => {
    const spy = expect.createSpy()
    const renderedComponent = mount(
      <App dispatch={spy} />
    )
    expect(spy).toHaveBeenCalled()
  })
  it('calls dispatch with Actions.userAuth', () => {
    const spy = expect.createSpy()
    const renderedComponent = mount(
      <App dispatch={spy} />
    )
    expect(spy).toHaveBeenCalledWith(Actions.userAuth())
  })
})
```

The first test is simple and it's something you've seen before.

The second test is a little bit more complicated. The first difference is that we're rendering the component with `mount` rather than `shallow`. This takes longer, but it's necessary since a `shallow` render does not give you access to the lifecycle methods such as `componentDidMount`. Then we're passing in our spy as `this.props.dispatch`, so when `componentDidMount` is triggered, it calls our `spy` rather than what would have ordinarily been passed in as the `dispatch` function from redux.

The third test is similar to the second test, but we are making a different assertion. In this case, we are checking to make sure that `componentDidMount` attempted to call our `dispatch` function with the right parameter. If it called it with `Actions.userAuth()`, then the test will pass.

In theory, we could also test that `componentDidMount` was called, but at that point we'd be testing that React itself is working, and that's outside the scope of our unit tests. It's safe to assume that anything implicit in React will work properly.

So now we need to change our `app/index.js` file to take in the new `App` component. All we have to do is import our new App component and delete the old one.

/app/index.js
commit: coming soon

```
...
import { default as App } from "./components/App"
...
```

And that's that. Your user is now being authenticated automatically when the page renders. You can check this by using `console.log` on your props to a connected component, such as the `Login` component (for now).

Authenticated pages

So now that we can determine whether a user is logged in, we can give that user access to pages that only logged in users would have.

In this case, if a user is logged in, we want them to see the `Chat` component rather than the login/signup screen when they get to the `Home` page.

The first step is to connect our `Home` component to redux so it knows whether or not the user is signed in.

/app/components/Home/index.js
commit: coming soon

```
...
import { connect } from "react-redux"

...

const mapStateToProps = state => ({
  user: state.user
})

export default connect(mapStateToProps)(cssModules(Home, style))
```

Then import our `Chat` component and have it render if the user is currently logged in.

/app/components/Home/index.js
commit: coming soon

```
import { default as Chat } from "../Chat"

...

  render() {
    if (this.props.user.email) {
      return (<Chat />)
    }
    return (
      <div className={style.leader}>
        ...
    )
  }
  ...
```

And since we're already logged in, when you refresh the page, you should see our `Chat` component instead of the login screen.

At this point, we effectively have the outline of what will become our interface for a company representative. This is the primary means by which an administrator can communicate with an anonymous customer visiting the site.

A Note on Frontend Security

You might be thinking, "But is it really safe to determine the authentication status based on `props.user`?" If you were, kudos.

Security does not happen on the client. The best you can do on the client is give the appearance of security by disabling links and guiding the user to the proper view. Someone can always jump into the console and do whatever they want on the client since our entire app is available in our `bundle.js` file.

That said, an app with this architecture is still secure because our *data* is secure and comes from our server. Sure, someone can spend a lot of time figuring out how to get access to the `Chat` component even though they're not supposed to have access, but what can they do with it? They can only get the data to populate that component if they have a valid token and pass that to the server, which only then will respond with the data.

So keep in mind that whatever you put in a view, you cannot hide from your users. Take this tutorial series, for example. You could, in theory, change the links on the course list into something valid and potentially be able to click on it. But since the data does not live on the client, but in markdown files on the server, the page will be empty when you get there and you have no way to get access to the data without a valid token, which you can only get if you are authorized by the server.

In short, that's a long-winded way of saying that security is on the server not the client.

NPM Package for a React Component: Part 1

- Create the chat component
- Publish to NPM

So now that we have the ability to sign up and login, as well as a basic interface for chat, we need to create the interface for the anonymous customer.

The primary use-case of this app is as an NPM installation on an existing project to add chat functionality. Fortunately, this is really easy for us to do because of React's modular nature.

This piece of the app is gives us the chat functionality that you can easily install on any client and it's the medium through which our potential customers will contact the company representatives. When it's finished, it will look like the images below.

Installing dependencies

The first thing we need to is create a new directory and initialize a new NPM package. This is the same as the last time we initialized NPM. *Do not create this directory within the existing phoenix-chat-frontend directory; this is a separate project*. So the directory structure will look roughly like the following, with three separate directories:

```
phoenix-chat
  |-- node_modules
  |-- dist
  |-- src

phoenix-chat-frontend

phoenix-chat-api
```

```
$ mkdir phoenix-chat
$ cd phoenix-chat && npm init
```

Then we're going install `babel` which will allow us to use React with `jsx` and `ES2015` syntax.

```
$ npm install --save-dev babel-cli babel-preset-react \
  babel-preset-es2015 babel-preset-stage-0
```

Then, just as we did with our frontend app, we need to add Babel presets to package.json.

/package.json
commit: coming soon

```
...
  "devDependencies": {
    ...
  },
  "babel": {
    "presets": [
      "es2015",
      "react",
      "stage-0"
    ]
  }
...
```

Creating the PhoenixChat component

Now we need to create our `PhoenixChat` component. We will start with the fixed-position chat bubble in the bottom right, then build out the chat interface.

To start, we need a `jsx` file and a `style.js` file that will contain out `style` object. You may recall from an earlier section that there are many ways to import styles in React. For this project, we are going to use inline-styling.

```
$ mkdir src dist
$ touch src/{PhoenixChat.jsx,style.js,README.md} README.md
```

And since placeholder content is boring, lets add a chat image. At some point, you should upload your own image to S3 and use that URL for static asset hosting, but for now, we'll just use Github and a filter to turn the image white.

Then let's create the chat bubble component.

/src/PhoenixChat.jsx
commit: coming soon

```
import React from 'react'
import style from './style.js'

export class PhoenixChat extends React.Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <div
        style={style.chatButton}>
        <img
          src="https://github.com/LearnPhoenix/graphics/blob/master/phoenix-chat-icon.png?r
          style={style.chatImage} />
      </div>
    )
  }
}

export default PhoenixChat
```

Then we need to add styling to this component within our `style.js` file. We're just going to use pure JavaScript styling, both because it's worth knowing how to do and because it's quick and easy.

```
/src/style.js
commit: coming soon
```

```
export const style = {
  chatButton: {
    position: "fixed",
    right: "20px",
    bottom: "20px",
    background: "rgb(58, 155, 207)",
    color: "white",
    zIndex: "1000",
    height: "60px",
    width: "60px",
    border: "1px solid #ccc",
    borderRadius: "50%",
    display: "flex",
    justifyContent: "center",
    alignItems: "center",
    cursor: "pointer",
    boxShadow: "2px 4px 2px -2px rgba(0,0,0,0.3)"
  },
  chatImage: {
    display: "block",
    height: "40px",
    width: "40px",
    filter: "invert(100%)",
    WebkitFilter: "invert(100%)",
    MozFilter: "invert(100%)"
  }
}

export default style
```

Connecting in development

Before we can check to make sure this works properly, we have to change around a few things in our `package.json` file and link this package to our existing frontend (at least temporarily).

We're going to add two scripts to `package.json`, and change the main import file. The first, `watch`, will watch for changes and automatically compile your code as you go. This is what you'll have running while in development. `build` will prepare the app for production, and `version` prepares the next version for deployment.

We are also changing the import file to `dist/phoenix-chat.js`, which is where our `babel` command line tool will output the resulting file.

```
/package.json
commit: coming soon
```

```
...
  "main": "./dist/PhoenixChat.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "watch": "babel src --watch --out-dir dist",
    "build": "babel src --out-dir dist"
  },
```

Now that we have our package ready, it's time to link it to our `phoenix-chat-frontend`. We can use `npm link` to create a globally-installed symbolic link that you can use to locally install the package in a different app. First let's build the package.

```
$ npm run build
```

Then we're going to create an `npm link` between our package and our `node_modules` for our `phoenix-chat-frontend`. Do this by running `npm link` within the `phoenix-chat` directory to create the global module, then running `npm link phoenix-chat` from within `phoenix-chat-frontend` to link the module.

```
$ npm link
$ cd ../phoenix-chat-frontend
$ npm link phoenix-chat
```

Unfortunately with the current version of npm this will not work, because two copies of React will end up being installed and you won't be able to use `refs` (see this specific issue if you are interested). According to the docs, this is intentional, but it's an easy fix with webpack. Within `phoenix-chat-frontend`, add the following to your webpack configuration:

```
...
  resolve: {
    extensions: [ '', '.js' ],
    fallback: path.join(__dirname, "node_modules")
  },
  resolveLoader: {
    fallback: path.join(__dirname, "node_modules")
  },
...
```

Now if you look in your `node_modules` directory within your frontend app, you should see a `phoenix-chat` module installed. Keep in mind that if you `npm prune` or re-download this repo, you will have to re-link this package since it wont be downloaded from an `npm install`.

Now we're going to run the `phoenix-chat-frontend` and add this component to our `Home` component.

/app/components/Home/index.js
commit: coming soon

```
import PhoenixChat from "phoenix-chat"

export class Home extends React.Component {
  ...

  render() {
    if (this.props.user.email) {
      return (
        <Chat>
          <PhoenixChat />
        </Chat>
      )
    }
    return (
      <div className={style.leader}>
        <h1 className={style.title}>Phoenix Chat</h1>
        { this.state.formState === "signup" ? <Signup /> : null }
        { this.state.formState === "login" ? <Login /> : null }
        { this.renderToggleContent() }
        <img
          role="presentation"
          className={style.circles}
          src="https://s3.amazonaws.com/learnphoenix-static-assets/images/circles-full.png"
        <PhoenixChat />
      </div>
    )
  }
  ...
}
```

Obviously, in the long-run, there is no reason to have an anonymous chat window on the same screen as the administrator, since he would be anonymously messaging himself, but this makes testing a lot easier.

Now you should see the chat bubble appear in the bottom right.

From here, we can continue working on our npm module while watching our changes take effect in realtime. Within the `phoenix-chat` directory, run the `watch` command to keep your project updated as you make changes.

```
$ npm run watch
```

But this component isn't really all that useful, since it's just a button that doesn't do anything. In the next lesson, we'll add some more functionality.

NPM Package for a React Component: Part 2

- Build the chat interface

At this point, our npm package is not especially functional--it's just a button that hovers over the bottom-right of your app. So in order to make it useful, we need to add a chat interface where users can input a message and (eventually) send it to our administrator.

Chat interface

The first thing we need to do is break the button out into its own component called `PhoenixChatButton`, then create the `PhoenixChatSidebar` component, and finally tie them together based on user actions.

So first, let's refactor the `PhoenixChat` component so it knows it's state (whether the chat is open or closed) and create a function that toggles the state of the component.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
export class PhoenixChat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      isOpen: false
    }
    this.toggleChat = this.toggleChat.bind(this)
  }

  toggleChat() {
    this.setState({ isOpen: !this.state.isOpen })
  }

  render() {
    return (
      <div>
        { this.state.isOpen
          ? <PhoenixChatSidebar toggleChat={this.toggleChat} />
          : <PhoenixChatButton toggleChat={this.toggleChat} /> }
      </div>
    )
  }
}
```

Now we should create the `PhoenixChatButton` based on the component we made previously, but with an `onClick` that toggles the chat sidebar when a user clicks on it. We're going to add this to the same file to keep things simple, but you can create a new file and import this into `src/PhoenixChat.jsx` if you prefer.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
export class PhoenixChatButton extends React.Component {
  render() {
    return (
      <div
        onClick={this.props.toggleChat}
        style={style.chatButton}>
        <img
          src="https://github.com/LearnPhoenix/graphics/blob/master/phoenix-chat-icon.png?r
          style={style.chatImage} />
      </div>
    )
  }
}
```

And finally, we should create the `PhoenixChatSidebar` component. We're going to include some hard-

coded data in the state of this component that we will eventually swap out.

The only thing that might look new is the `map` under the render function that maps over the dummy data in `this.state.chat` and creates an array of components. What we're doing is checking which `name` is present and assigning a particular style based on that name (either right or left aligned). Other than that, everything is fairly straightforward with a column of elements.

We're also adding an index as a key. If you get to a point where you have a lot of messages and this causes performance issues, you'll need to assign a unique id to each message and assign the key to that id. But in this case, since we can only add messages and can't remove them, an index is fine.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
export class PhoenixChatSidebar extends React.Component {
  constructor(props) {
    super(props)
    this.closeChat = this.closeChat.bind(this)
    this.state = {
      messages: [
        {from: "Client", body: "Test"},
        {from: "John", body: "Foo"},
        {from: "Client", body: "Bar"}
      ]
    }
  }

  closeChat() {
    this.props.toggleChat()
  }

  render() {
    const list = !this.state.messages ? null : this.state.messages.map((bubble, i) => {
      const right = bubble.from === "Client"
      return (
        <div style={{...style.messageWrapper, justifyContent: right ? "flex-end" : "flex-st
          <div
            key={i}
            style={ right ? style.chatRight : style.chatLeft }>
            { bubble.body }
          </div>
        </div>
      )
    })
    return (
      ...
    )
  }
}
```

And then in the final return, under our list, add the following jsx:

```
...
export class PhoenixChatSidebar extends React.Component {
  ...

  render() {
    ...

    return (
      <div style={style.client}>
        <div style={style.header}>
          <div style={style.logo}>
            <img
              style={{ height: "40px", paddingRight: "5px" }}
              alt="learnphoenix logo"
              src="https://s3.amazonaws.com/learnphoenix-static-assets/favicons/favicon-96x
            <span>PhoenixChat.io</span>
          </div>
          <div
            style={style.close}
            onClick={this.closeChat}>
            Close
          </div>
        </div>
        <div style={style.chatContainer}>
          { list }
        </div>
        <div style={style.inputContainer}>
          <input
            type="text"
            style={style.inputBox} />
          <div>
            100% free by <a href="https://learnphoenix.io">PhoenixChat</a>
          </div>
        </div>
      </div>
    )
  }

  ...
}
```

The last thing we need is styling for this new component. Within `style.js`, add the following (explained below the code):

```
/src/style.js
commit: coming soon
```

```
export const style = {
```

```
  ...
  messageWrapper: {
    display: "flex",
    flexFlow: "row nowrap"
  },
  chatRight: {
    color: "#666666",
    margin: "0.5rem 0",
    padding: "1rem",
    borderRadius: "5px",
    background: "rgb(230, 230, 234)"
  },
  chatLeft: {
    color: "white",
    margin: "0.5rem 0",
    padding: "1rem",
    borderRadius: "5px",
    background: "rgb(58, 155, 207)"
  },
  client: {
    width: "350px",
    position: "fixed",
    zIndex: "1000",
    right: "0",
    top: "0",
    bottom: "0",
    background: "rgb(247, 247, 248)",
    borderLeft: "1px solid #ccc"
  },
  header: {
    height: "50px",
    background: "white",
    borderBottom: "1px solid #ccc",
    width: "100%",
    boxShadow: "0 2px 2px -1px rgba(0,0,0,0.1)",
    display: "flex",
    justifyContent: "space-around",
    alignItems: "center"
  },
  logo: {
    display: "flex",
    flexFlow: "row nowrap",
    alignItems: "center"
  },
  close: {
    cursor: "pointer"
  },
  chatContainer: {
    padding: "1rem",
    overflowY: "auto",
    height: "calc(100vh - 130px)"
  },
```

```
  inputContainer: {
    position: "absolute",
    bottom: "0",
    left: "0",
    width: "100%",
    display: "flex",
    flexFlow: "column nowrap",
    alignItems: "center",
    justifyContent: "space-around",
    height: "80px"
  },
  inputBox: {
    width: "90%",
    height: "40px",
    borderRadius: "5px",
    fontSize: "14px",
    border: "1px solid #ccc",
    paddingLeft: "10px",
    outline: "none"
  }
}
```

Most of these styles serve to fix the position of the chat element on the right of the screen with everything either fixed or absolutely positioned.

Auto-scroll

At this point, you have an interface that performs almost as you would expect. You can add messages and they show up on a list. Unfortunately, this list does not know that the bottom is what we want to keep track of. In order to fix this, we need to auto-scroll to the bottom.

Although this seems like something you should be able to do with CSS, you have to use JavaScript to add this functionality. We're going to create a new function and introduce another React feature called `refs` (docs). We are going to use `refs` to create a unique identifier for each of our messages so we can auto-scroll to the end of the list.

Think of `refs` as a way of creating a reference to a particular React element. This is generally what you would use in place of something like `document.getElementById()` to find elements.

The ref that we are adding to each message takes in the index (`i`) from the map over our `this.state.messages` list and assigns that to the name of the reference. That way, when we need to later reference the last item, we can find its relative position in our current messages list (the oldest message will have an index of `0`, the newest will have index of `this.state.length - 1`).

We're also going to add a `ref` to our chat container (the div that contains all of our messages) so we can select it and change the scroll position.

Also, make sure you bind the `map` function to the current `this` context or you will get an error because `map` has its own context. An alternate approach to using `bind` is to use the `self = this` approach.

Note: You will often see `refs` used as strings, but this is now deprecated in favor of functions.

```
...
export class PhoenixChatSidebar extends React.Component {

...

  render() {
    const list = !this.state.messages ? null : this.state.messages.map(({ body, id, from },
      const right = from === "Client"

      return (
        <div
          ref={ ref => { this[`chatMessage:${i}`] = ref }}
          key={i}
          style={{...style.messageWrapper, justifyContent: right ? "flex-end" : "flex-start
          <div
            style={right ? style.chatRight : style.chatLeft}>
            { body }
          </div>
        </div>
      )
    })
    return (
      <div style={style.client}>
        <div style={style.header}>
          <div style={style.logo}>
            <img
              style={{ height: "40px", paddingRight: "5px" }}
              alt="learnphoenix logo"
              src="https://s3.amazonaws.com/learnphoenix-static-assets/favicons/favicon-96x
            <span style={style.title}>PhoenixChat.io</span>
          </div>
          <div
            style={style.close}
            onClick={this.closeChat}>
            Close
          </div>
        </div>
        <div
          ref={ref => this.chatContainer = ref}
          style={style.chatContainer}>
          { list }
        </div>
      ...
    )
    ...
  }
}
```

So now we have two refs set up. The first will create a new ref for each message and the second will

keep a reference to our message list.

From here, we can reference those nodes and set the scroll position. To do that, we need to hook in to our lifecycle method after render and after the DOM is updated because we need the actual DOM values, which we can do with `componentDidUpdate`.

We will start by selecting the last message by finding the `ref` to message with the index of `this.state.messages.length - 1`, then we set the chat container's `scrollTop` to the last element in the list's `offsetTop`, which sets the scroll position of our list container to the bottom of our last message. Theoretically, this will push our scroll position past the bottom of the container, but it will actually just stop at the bottom of the list container.

```
...
export class PhoenixChatSidebar extends React.Component {
  ...
  componentDidUpdate() {
    if (this.props.messages.length > 0) {
      const lastMessage = this[`chatMessage:${this.props.messages.length - 1}`]
      this.chatContainer.scrollTop = lastMessage.offsetTop
    }
  }
  ...
}
```

Please note: since our `input` is not currently connected, this will not do anything. But when we connect this component to our channel in a couple lessons and send messages, we will have the auto-scrolling chat component that we expect.

And since you're likely tracking all of this with git, you should create a `.gitignore`.

```
$ touch .gitignore
```

Include your `dist` directory and your `node_modules`, because both of these will be generated dynamically.

```
dist
node_modules
```

peerDependencies

For the sake of good housekeeping, let's add React as a peer dependency, so when the package is

installed it will note that it requires React in order to work.

```
  "dependencies": {
    ...
  },
  "peerDependencies": {
    "react": "^15.0.0"
  }
```

And finally, if you'd like to publish this package to npm, it's as simple as following these instructions.

If you choose to go this route, be aware of a few things that might trip you up. Npm will automatically generate a `.npmignore` file if you do not include one, and by default, this copies your `.gitignore`. So, if you include `dist` in your `.gitignore`, your `dist` will be ignored by npm and your package won't work. The solution to this is to create an `.npmignore` file with just the things that npm should ignore:

```
$ touch .npmignore
```

This will be basically the same as our `.gitignore` except we want to make sure that our `dist` is not ignored.

```
node_modules
```

And that's a wrap. We won't touch this again until we connect it to our Phoenix channel.

Set up Phoenix Channels

- Setting up a `room` channel
- Extracting authorization

Now that we have a user interface in place to send and receive messages, we should start putting together our backend to actually handle these messages. This is where Phoenix really shines.

If you're familiar with a `socket`, you'll pick up `channels` fairly quickly. If you're not familiar with the concept of websockets, it's best to think of the internet as split between two types of connections: 1) request-response, and 2) sockets.

Request-response is how we're handling our JSON web token. Each time you send a request, the connection (`conn`) is essentially a new `struct` that goes through your pipeline and then disappears. A real-world analogy would be sending and receiving letters in the mail; once you write the letter and drop it off (request), you have to wait until you get a response.

Using sockets/channels is more like starting an ongoing conversation with someone standing next to you. The line of communication is always open and feedback is immediate.

What's more, a `channel` can be stateful, meaning when we connect to a `channel` we don't have to worry about storing data and keeping track of our conversations. When you start a connection, the socket stays alive and is transformed until the connection is broken.

If this sounds complicated or magical, it should make more sense when we start to implement it.

Setting up a `room` channel

Based on the way our app works, we're going to create a new `room` for each of our anonymous customers and list them in the `Sidebar`. When we click on one of the active chats, we want access to the chat history between the company representative and the anonymous customer.

We're going to use the built-in generator to create our `room` channel. Remember, you can run `mix --help` to see all the built-in functions you can run with `mix`.

```
$ mix phoenix.gen.channel Room
```

Then (as it says in the console output), we want to add that `channel` to our `user_socket.ex` file. As it

turns out, this is already a default option, so you can just un-comment the code that already exists.

We're also going to use the `assign` function to store data about the user in the socket so we can more easily identify the user (the reason for this will become more clearer when we connect to the frontend). You can think of `assign` as a way to pass values to other parts of the app on the socket struct/object.

```
/web/channels/user_socket.ex
commit: coming soon
```

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

  ## Channels
  channel "room:*", PhoenixChat.RoomChannel
  ...

  def connect(params, socket) do
    socket = socket
      |> assign(:user_id, params["id"])
      |> assign(:username, params["username"])
      |> assign(:email, params["email"])
      |> assign(:uuid, params["uuid"])
    {:ok, socket}
  end

  ...
```

Then we need to handle incoming messages, so within `room_channel.ex`, add the following (with explanation below the code).

```
/web/channels/room_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.RoomChannel do
  use PhoenixChat.Web, :channel
  require Logger

  def join("room:" <> _uid, payload, socket) do
    if authorized?(payload) do
      {:ok, socket}
    else
      {:error, %{reason: "unauthorized"}}
    end
  end

  def handle_in("message", payload, socket) do
    Logger.debug "#{inspect payload}"
    broadcast socket, "message", payload
    {:noreply, socket}
  end

  ...
```

First, we're adding a `join` function that tells Phoenix what to do when a frontend attempts to join the socket. In this case, we are using pattern matching to determine what to join. So if an incoming `join` request starts with the pattern `room:` with anything following (other than `room:lobby` from the boilerplate code), the `join` that we defined will get called.

If we get a pattern that matches, we check to see if the user is authorized. The current `authorized?` function simply returns true, but this is where we can add authorization later on. If the user is authorized, it returns the socket allowing us to connect to it.

Note: in Elixir, question marks (`?`) are valid characters in function names.

The next function we're adding is `handle_in`, which is one of the built-in functions that Phoenix looks for when the socket receives an incoming request. In our case, we want to listen for a request that matches the pattern `message` (which is an arbitrary title; you could name it whatever you want) then `broadcast` that message to everything on the frontend that is currently listening to this socket. Or, directly from the docs:

> 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. Typically you'll either forward a message to all listeners with `broadcast!/3`, or push a message directly down the socket with `push/3`. Incoming callbacks must return the `socket` to maintain ephemeral state.

We're also going to temporarily include a `Logger.debug` to log our output. Note that we're using `inspect`, which allows us to log data that is more complex than a string (tuples, etc). For more on built-in protocols, check the docs.

And believe it or not, that's all we need for a functioning channel. Phoenix makes it really easy to set up `channels`. If this has you confused, it will make more sense when we implement the frontend code to connect to the socket.

Extracting authorization

We're going to create a `ChannelHelpers` module and separate out the authorization logic because we're going to need to use the same logic in multiple channels. We will likely implement additional utility functions in the future, so in order to keep our code DRY and consistent we're going to keep all these functions in one place.

We're also going to introduce some inline documentation using `@moduledoc` and `@doc`. Both accept markdown and are used to automatically generate documentation. If you've ever wondered how Elixir, Phoenix, Ecto, and every other hex package has such awesome documentation, it's because people use the inline documentation to describe their functions.

The two terms are pretty self-explanatory: `@moduledoc` is for describing the module, while `@doc` is for describing a function. You should definitely look over some existing packages to see how functions are documented.

```
$ touch web/channels/channel_helpers.ex
```

```
/web/channels/channel_helpers.ex
commit: coming soon
```

```
defmodule PhoenixChat.ChannelHelpers do
  @moduledoc """
  Convenience functions imported in all Channels
  """

  @doc """
  Convenience function for authorization
  """
  def authorize(payload, fun, custom_authorize \\ nil) do
    check_authorization = custom_authorize || &authorized?/1
    if check_authorization.(payload) do
      fun.()
    else
      {:error, %{reason: "unauthorized"}}
    end
  end

  @doc """
  Function that determines authorization logic. If `true`, all users will be authorized.
  """
  def authorized?(_payload) do
    true
  end
end
```

The `authorize/2` function takes in the payload and an anonymous function. If the user passes the `authorized?/1` function, run the function. If it's not clear what this function does, it will become clear when we implement it in our `RoomChannel`. If it fails the `authorized?` tests, then we return an `:error` with `unauthorized`.

We've also set up `authorize/3`, which takes in three arguments. If we so choose, we can pass in a `custom_authorize` override that will override the authorization we set up in `authorized?`. This is useful if you have a request that has an unusual authorization method.

Recall that anonymous functions are called with an additional `.`, such as `function.()` rather than simply `function()`. We are noting this again because it regularly causes confusion for people new to Elixir. For a more detailed explanation as to why Elixir has different syntax for named functions and anonymous functions, check out this StackOverflow answer by Elixir creator Jose Valim.

Then, since we're going to want to access these helpers in all of our channels, we're going to import them into every channel. Within `web/web.ex`, you can add modules to every function of a particular type (such as `channel`, `model`, `controller`, etc). This should be done sparingly because every function you add here is imported into each struct and can significantly slow down your app if you're not careful.

```
/web/web.ex
commit: coming soon
```

```
  ...

  def channel do
    quote do
      use Phoenix.Channel

      alias PhoenixChat.Repo
      import Ecto
      import Ecto.Query, only: [from: 1, from: 2]
      import PhoenixChat.Gettext
      import PhoenixChat.ChannelHelpers
    end
  end

  ...
```

Then we should update our `RoomChannel` to use these helper functions, so we'll remove the `authorized?/1` function and replace the authorization conditional within our `join/3` function.

/web/channels/room_channel.ex
commit: coming soon

```
defmodule PhoenixChat.RoomChannel do
  use PhoenixChat.Web, :channel
  require Logger

  def join("room:" <> _uid, payload, socket) do
    authorize(payload, fn ->
      {:ok, socket}
    end)
  end

  def handle_in("message", payload, socket) do
    Logger.debug "#{inspect payload}"
    broadcast socket, "message", payload
    {:noreply, socket}
  end
end
```

We've changed the join function to take in both a `payload` and an anonymous function that returns the same thing our previous `authorized?/1` function returned, which is `{:ok, socket}`.

Connect React to Channels

- Channels with local state
- Anonymous user channels

As it turns out, it's also not especially difficult to connect React to a Phoenix channel. We're going to connect to the `channel`, then pass our data in through Redux and render our messages to the page.

We're going to start with our `phoenix-chat` npm component and handle state locally. Then we'll switch to our `phoenix-chat-frontend` and handle our state with Redux to show how the approaches differ.

Adding Phoenix.js

Phoenix has recently been added to npm, which makes things a lot easier. Previously, this required copying a file and keeping it in a `/vendor` directory. We don't have to worry about that anymore.

```
$ npm install --save phoenix
```

Configuring your `channel`

We want access to our socket from the entire `PhoenixChat` component so we can push messages when a user has the `PhoenixChatSidebar` closed, so we'll connect to the socket on `componentDidMount`. We'll also want to create a `configureChannels` function to pass in all the configuration options we need. Keep in mind that sockets are lightweight and connecting and disconnecting is not an expensive operation.

You'll notice we're hard-coding `localhost:4000` into our component. When we actually deploy this, we're going to change this out for our actual server.

First we need to get rid of our hard-coded messages from our `PhoenixChatSidebar`.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
import { Socket } from "phoenix"

...

export class PhoenixChatSidebar extends React.Component {
  constructor(props) {
    super(props)
    this.closeChat = this.closeChat.bind(this)
  }

  ...
}
```

Then we need to configure our channel, which we'll do in a `configureChannels` function (explanation below the code).

```
...

export class PhoenixChat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      isOpen: false,
      input: "",
      messages: []
    }
    this.toggleChat = this.toggleChat.bind(this)
    this.configureChannels = this.configureChannels.bind(this)
  }

  componentDidMount() {
    this.socket = new Socket("ws://localhost:4000/socket")
    this.socket.connect()
    this.configureChannels("foo")
  }

  configureChannels(room) {
    this.channel = this.socket.channel(`room:${room}`)
    this.channel.join()
      .receive("ok", ({ messages }) => {
        console.log(`Succesfully joined the ${room} chat room.`)
        this.setState({
          messages: messages || []
        })
      })
      .receive("error", () => {
        console.log(`Unable to join the ${room} chat room.`)
      })
    this.channel.on("message", payload => {
```

```
      this.setState({
        messages: this.state.messages.concat([payload])
      })
    })
  }

  toggleChat() {
    this.setState({ isOpen: !this.state.isOpen })
  }

  render() {
    return (
      <div>
        { this.state.isOpen
          ? <PhoenixChatSidebar
              input={this.state.input}
              messages={this.state.messages}
              toggleChat={this.toggleChat} />
          : <PhoenixChatButton toggleChat={this.toggleChat} /> }
      </div>
    )
  }
}
```

When the component mounts, we're going to initialize the socket and assign it to `this.socket`, then call `configureChannels` and pass in a string `foo` to connect us to `room:foo`. Obviously, this is jsut a temporary value; soon, we will pass a unique ID.

Within `configureChannels`, we take the `room` that we got from `componentDidMount` and we initiated the channel with `socket.channel`. Then we joined the channel with `channel.join`. If we receive an `ok` response, we log success and update our state to include any previous messages, otherwise log the failure.

The second part is a listener for whenever our channel receives a `message`, which is an arbitrary name that we are using to label this kind of action. When a `message` is received (as you will see later in a `handleMessageSubmit` function), we want to add the object to `this.state.messages`. The last thing we need to do is to add the current `channel` to `this.channel` so we can manipulate it later.

We also changed our render function to handle an empty `messages` array; if you try to `map` over an empty array, you get an error.

Now when you check your browser console, you should see the following.

```
Succesfully joined the foo chat room.
```

Our channel is now configured.

Anonymous users

Now that we have the ability to connect to the channel, we're going to need to set this up to handle anonymous users. We're going to use the `uuid` package to create a universally unique identifier for each of our anonymous users.

```
$ npm install --save uuid
```

Then we just need to change our `componentDidMount` function to assign a universally unique id (`uuid`) or find one that already exists. We're going to store this `uuid` in `localStorage`. If one already exists, then we're going to connect to the room with that id. Otherwise, we're going to generate a new id, set that to `localStorage` and connect to a new room.

/src/PhoenixChat.jsx
commit: coming soon

```
import uuid from 'uuid'

  ...
export class PhoenixChat extends React.Component {

  ...

  componentDidMount() {
    if (!localStorage.phoenix_chat_uuid) {
      localStorage.phoenix_chat_uuid = uuid.v4()
    }

    this.uuid = localStorage.phoenix_chat_uuid
    const params = { uuid: this.uuid }
    this.socket = new Socket("ws://localhost:4000/socket", { params })
    this.socket.connect()

    this.configureChannels(this.uuid)
  }

  ...
}
```

And that's all we need to do to handle anonymous users. Refresh and check your browser console and you'll see that you're now connected to a different `room`.

Submitting messages

Now we need to change our `messages.map` to handle our current anonymous user and handle messages sent to our `input`.

You might notice we've added a `from` field, which we do not currently have in our messages. We're going to add this and other fields in the lesson on message persistence. For now, let's just use them as placeholders. **So keep in mind, this app will not work until we update our model.**

We're also changing the values from `this.state` to `this.props`, since we're going to pass in all the data from the parent (`PhoenixChat`) component, which is connected to the socket.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
...

export class PhoenixChatSidebar extends React.Component {

  ...

  render() {
    const list = !this.props.messages ? null : this.props.messages.map(({ body, id, from },
      const right = from === localStorage.phoenix_chat_uuid

      return (
        <div
          ref={ref => this[`chatMessage:${i}`] = ref}
          key={i}
          style={{ ...style.messageWrapper, justifyContent: right ? "flex-end" : "flex-star
          <div
            style={right ? style.chatRight : style.chatLeft}>
            { body }
          </div>
        </div>
      )
    })

    return (
      <div style={style.client}>
        ...
        <div style={style.inputContainer}>
          <input
            onKeyDown={this.props.handleMessageSubmit}
            onChange={this.props.handleChange}
            value={this.props.input}
            type="text"
            style={style.inputBox} />
          ...
        </div>
      </div>
    )
  }
}
```

In order to handle our inputs, we're going to need two additional functions: `handleMessageSubmit` and `handleChange`.

`handleMessageSubmit` will listen for `onKeyDown` events (if you don't know all the React events you have at your disposal, you should peruse the docs) and when the key is the "return" key (which has `keyCode` number 13), it will send our input to our channel and clear the input. We're also checking to make sure the input value isn't empty.

`handleChange` listens to changes in our input and changes the value of that input accordingly. This is useful if we decide to implement any sort of validation or text formatting (such as markdown) later on.

/src/PhoenixChat.jsx
commit: coming soon

```
  ...

export class PhoenixChat extends React.Component {
  constructor(props) {
    super(props)
    this.handleMessageSubmit = this.handleMessageSubmit.bind(this)
    this.handleChange = this.handleChange.bind(this)
    ...
  }
  ...

  handleMessageSubmit(e) {
    if (e.keyCode === 13 && this.state.input !== "") {
      this.channel.push('message', {
        room: localStorage.phoenix_chat_uuid,
        body: this.state.input,
        timestamp: new Date().getTime()
      })
      this.setState({ input: "" })
    }
  }

  handleChange(e) {
    this.setState({ input: e.target.value })
  }

  ...

  render() {
    return (
      <div>
        { this.state.isOpen
          ? <PhoenixChatSidebar
              handleChange={this.handleChange}
              handleMessageSubmit={this.handleMessageSubmit}
              input={this.state.input}
              messages={this.state.messages}
              toggleChat={this.toggleChat} />
          : <PhoenixChatButton toggleChat={this.toggleChat} /> }
      </div>
    )
  }
}
```

Let's also add a `componentWillUnmount` function to our `PhoenixChat` component to disconnect from our socket when we close the window.

/src/PhoenixChat.jsx
commit: coming soon

```
  ...
  componentWillUnmount() {
    this.channel.leave()
  }
  ...
```

And now you have (half) a functional anonymous chat client set up. You can send messages and see them log in your Phoenix backend.

The next thing we need to do is give our admin the ability to join this channel so she can respond. This will be done in `phoenix-chat-frontend`. In order to do this, we will have to create a new channel just for our administrator.

Use Presence to List Active Users

- Creating an `AdminChannel`
- Using `Presence`

Now that we have the ability to send messages to our backend, we need to give our administrator the ability to respond to these messages. We're going to accomplish this by listing all of our active chats in the sidebar on the left within our `phoenix-chat-frontend` app.

This sidebar will connect to a new channel called `admin`, which will have the topic `active_users`, which we will use to keep track of the presence of all active sockets.

Basics of Phoenix Presence

Presence is an awesome feature in Phoenix. It gives you the ability to easily discover which users are currently connected to a particular socket. For example, if you wanted to list all users that are currently active in a chatroom, you could use `Presence` to easily list them all rather than manipulating a list of users when people join or leave the chatroom. In other frameworks, this is a difficult problem to solve, but in Phoenix, we get it for free.

Make sure you are running at least Phoenix 1.2.0 `{:phoenix, "~> 1.2.0"}` within your `mix.exs` file. If you are not, you will get a compilation error.

Then create a `Presence` module in `lib/phoenix_chat/presence.ex`.

```
$ touch lib/phoenix_chat/presence.ex
```

```
/lib/phoenix_chat/presence.ex
commit: coming soon
```

```
defmodule PhoenixChat.Presence do
  use Phoenix.Presence, otp_app: :phoenix_chat,
                        pubsub_server: PhoenixChat.PubSub
end
```

And finally add `Presence` to your supervision tree within `lib/phoenix_chat.ex`:

/lib/phoenix_chat.ex
commit: coming soon

```
  ...

  def start(_type, _args) do
    import Supervisor.Spec

    children = [
      # Start the endpoint when the application starts
      supervisor(PhoenixChat.Endpoint, []),
      # Start the Ecto repository
      supervisor(PhoenixChat.Repo, []),
      supervisor(PhoenixChat.Presence, [])
    ]

    ...
  end
```

Presence is now set up and ready for action.

Creating the AdminChannel

The first thing we need is the `AdminChannel`, which we will use to give the administrator access to the list of active users. An explanation of each function is below the code block.

```
$ touch web/channels/admin_channel.ex
```

/web/channels/admin_channel.ex
commit: coming soon

```
defmodule PhoenixChat.AdminChannel do
  @moduledoc """
  The channel used to give the administrator access to all users.
  """

  use PhoenixChat.Web, :channel
  require Logger

  alias PhoenixChat.{Presence}

  @doc """
  The `admin:active_users` topic is how we identify all users currently using the app.
  """
  def join("admin:active_users", payload, socket) do
    authorize(payload, fn ->
      send(self, :after_join)
      {:ok, socket}
    end)
  end

  @doc """
  This handles the `:after_join` event and tracks the presence of the socket that has subsc
  """
  def handle_info(:after_join, socket) do
    push socket, "presence_state", Presence.list(socket)
    Logger.debug "Presence for socket: #{inspect socket}"
    id = socket.assigns.user_id || socket.assigns.uuid
    {:ok, _} = Presence.track(socket, id, %{
      online_at: inspect(System.system_time(:seconds))
    })
    {:noreply, socket}
  end
end
```

Our `join/3` function should look familiar, as it's mostly the same as our `join/3` function in our `room` channel. The difference here is that we've included a new function: `send(self, :after_join)`.

This `send/2` function (docs) sends a message to a process, in our case back to the `AdminChannel` (`self`). We want this to happen when a user joins for the first time. It calls the `send/2` function with an arbitrary atom used for pattern matching. Since we want to run an action after user joins for the first time, we'll call our message `:after_join`.

Then we use the `handle_info/2` function (docs) to pattern match against messages sent by `send/2`. If the message matches the pattern `:after_join`, we run the `handle_info` function above. This function is a little bit complicated, so we'll go over it line-by-line.

The first line is `push socket, "presence_state", Presence.list(socket)`, which triggers a `presence_state` event in the client. `presence_state` is an arbitrary name that we are giving this event

and we will use it later when we connect this to our `phoenix-chat-frontend` app.

The next line simply logs the socket for debugging purposes.

The third line sets `id` to either the `user_id`, if the user is an admin (and therefore has a `user_id` in the database), or sets it to the `uuid` of the anonymous user (which we are not storing in the database).

Then we use pattern matching to make sure that `Presence.track` is working properly. If the response is `:ok`, we tell `Presence` to `track` (docs) the socket, using the `id` as the identifier and a map that contains the `online_at` metadata letting `Presence` know when the user joined the socket.

Otherwise, we send `:noreply` and return the socket unchanged.

Let's also add this channel to our `UserSocket`.

```
/web/channels/user_socket.ex
commit: coming soon
```

```
...

## Channels
channel "room:*", PhoenixChat.RoomChannel
channel "admin:*", PhoenixChat.AdminChannel

...
```

And that's all we need to do to set up `Presence`. The next step is to handle incoming messages and persist them to the database.

Persist Messages to the Database

- Creating a Message model
- Querying database
- Persist messages to database
- Unix timestamp to DateTime

To keep our app simple with the functionality most people expect, we're going to persist all messages to our Postgres database. In order to do this, we need to create a `Message` model so our database knows what kind of data it's receiving.

Later on, we may decide that we don't need to store these messages in a database or we may need to cache our messages in memory for better performance. If we later decide to go that route, we can use ETS. No need for premature optimization.

Creating the `Message` model

We're going to use the built-in `mix` generator to create the boilerplate for a new migration.

```
$ mix phoenix.gen.model Message messages body from room \
  timestamp:datetime user_id:references:users
```

This will create three new files, the names of which should have logged in your terminal.

The first thing we should check out is our migration file to update our database with the new data type. This file should already have all the relevant fields we are interested in and we don't have to change anything.

The last thing we're going to do before we persist our messages to the database is add a function that gives us the ability to limit the number of messages we receive from the database (with a default of 10).

`Ecto` allows us to define our queries as functions using `from/2`, which is a really nice feature, so we're going to create a new function called `latest_room_messages` that finds all messages from a particular room, then orders them by the most recent, and limits the number of messages.

```
/web/models/message.ex
commit: coming soon
```

```
defmodule PhoenixChat.Message do
  use PhoenixChat.Web, :model

  schema "messages" do
    field :body, :string
    field :timestamp, Ecto.DateTime
    field :room, :string
    field :from, :string
    belongs_to :user, PhoenixChat.User

    timestamps
  end

  @required_fields ~w(body timestamp room)
  @optional_fields ~w(user_id from)

  @doc """
  Creates a changeset based on the `model` and `params`.
  If no params are provided, an invalid changeset is returned
  with no validation performed.
  """
  def changeset(model, params \\ :empty) do
    model
    |> cast(params, @required_fields, @optional_fields)
  end

  @doc """
  An `Ecto.Query` that returns the last 10 message records for a given room.
  """
  def latest_room_messages(room, number \\ 10) do
    from m in __MODULE__,
         where: m.room ==  ^room,
         order_by: [desc: :timestamp],
         limit: ^number
  end
end
```

At this point, you should run your migration.

```
$ mix ecto.migrate
```

Updating our `RoomChannel`

Now we need to update our `RoomChannel` to start using our new model.

The first thing to change is our `join/3` function. Rather than simply checking the authorization status and returning the socket, we're going to reply with a payload of message records. This way, when a client joins the channel, she automatically receives the message history.

We start this function with the `room_id` and pass it into the `Message.latest_room_messages/2` function we created earlier. This creates an Ecto query based on the `room_id`, which we can them pass to `Repo.all`, which finds all of our data that matches the query.

Then we use `Enum.map` to format our messages in the way our client expects to receive them (explained in greater detail below the code block). Then we reverse the order to show the newest results first.

Finally, we return the messages that we just created along with our socket. `join/3` expects one of three things as its return value: `{:ok, Phoenix.Socket.t}`, `{:ok, map, Phoenix.Socket.t}`, or `{:error, map}` We were previously sending the first value, but now that we have a map to send along, we're sending the second value.

We also created a private function called `message_payload`, which takes the message and formats it before returning it.

```
/web/channels/room_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.RoomChannel do
  use PhoenixChat.Web, :channel
  require Logger

  alias PhoenixChat.{Message, Repo}

  def join("room:" <> room_id, payload, socket) do
    authorize(payload, fn ->
      messages = room_id
        |> Message.latest_room_messages
        |> Repo.all
        |> Enum.map(&message_payload/1)
        |> Enum.reverse
      {:ok, %{messages: messages}, socket}
    end)
  end

  defp message_payload(message) do
    from = message.user_id || message.from
    %{body: message.body,
      timestamp: message.timestamp,
      room: message.room,
      from: from,
      id: message.id}
  end

  ...
end
```

Note the syntax used to call `message_payload`. Using an ampersand (`&`) along with the arity (number of arguments) is a shorthand way of writing anonymous functions. The two functions below are equivalent.

```
|> Enum.map(&message_payload/1)
```

```
|> Enum.map(fn x -> message_payload(x) end)
```

You will see this syntax a lot in Elixir. For more, see the docs.

The last thing we should do before actually writing these to our database is check if the `id` passed to `UserSocket` is an administrator id or an anonymous user id.

We first pull the `id` out of the params passed to `connect/2`. Then, if that value exists *and* we can find that user in the database, then we assign that to the value `user`.

Then we assign values to `socket` depending on whether or not `user` passed the two criteria above (exists and is in the database). If the user does exist, we assign the `user_id`, `username`, and `email` to the `socket`. If `user` does not have a value, we assign the `user_id` to `nil` and give it the `uuid` for the anonymous user.

/web/channels/user_socket.ex
commit: coming soon

```
  alias PhoenixChat.{Repo, User}

  ...

  def connect(params, socket) do
    user_id = params["id"]
    user = user_id && Repo.get(User, user_id)

    socket = if user do
      socket
        |> assign(:user_id, user_id)
        |> assign(:username, user.username)
        |> assign(:email, user.email)
      else
        socket
          |> assign(:user_id, nil)
          |> assign(:uuid, params["uuid"])
      end

    {:ok, socket}
  end
```

Now we can persist our messages to the database. The most logical place to do this is within our `handle_in` function in the `RoomChannel`, since this is where we handle incoming messages. Rather than simply `broadcast` the message as we did previously, we're going to add the message to the database, then broadcast it.

Updating handle_in

Once we receive an incoming connection, we add two parameters to our `payload` from `socket.assigns` using `Map.put`, which adds the named key-value pair to a map: `user_id` and `from`.

Then we create a `changeset` with the `%Message{}` model and the payload we just updated in preparation for our database update.

Then we use a `case` statement to check if `Repo.insert` accepts our `changeset`. If it does, it will return `{:ok, message}` and we should format our message with `message_payload` and broadcast as we we did

before. In the event of an error, it returns an error.

The message payload sent to all client subscribers to the `"message"` event and includes a `from` field. This `from` is used to identify who sent the message and is either a `user_id` or a `uuid`. If a `uuid`, then user is anonymous; if a `user_id`, the user is an admin.

```
/web/channels/room_channel.ex
commit: coming soon
```

```
def handle_in("message", payload, socket) do
  payload = payload
    |> Map.put("user_id", socket.assigns.user_id)
    |> Map.put("from", socket.assigns[:uuid])
  changeset = Message.changeset(%Message{}, payload)

  case Repo.insert(changeset) do
    {:ok, message} ->
      payload = message_payload(message)
      broadcast! socket, "message", payload
      {:reply, :ok, socket}
    {:error, changeset} ->
      {:reply, {:error, %{errors: changeset}}, socket}
  end
end
```

Unix timestamp to DateTime

Since we're receiving all of our timestamps from JavaScript, the time is going to be in milliseconds since January 1, 1970, which is a different format than Ecto.DateTime expects. Because of this disparity, we're going to create a small module to handle this for us.

```
$ touch lib/phoenix_chat/date_time.ex
```

```
/lib/phoenix_chat/date_time.ex
commit: coming soon
```

```
defmodule PhoenixChat.DateTime do
  @behaviour Ecto.Type

  def type(), do: :datetime

  def cast(milliseconds) when is_integer(milliseconds) do
    with {:ok, datetime} <- DateTime.from_unix(milliseconds, :milliseconds),
         {:ok, ecto_datetime} <- Ecto.DateTime.cast(datetime),
         do: {:ok, ecto_datetime}
  end

  def cast(value) do
    Ecto.DateTime.cast(value)
  end

  def load(value) do
    Ecto.DateTime.load(value)
  end

  def dump(value) do
    Ecto.DateTime.dump(value)
  end
end
```

Then we need to change our `Message` model to use our `PhoenixChat.DateTime` module in `message.ex`.

```
/web/models/message.ex
commit: coming soon
```

```
...
  schema "messages" do
    field :body, :string
    field :timestamp, PhoenixChat.DateTime
    field :room, :string
    field :from, :string
    belongs_to :user, PhoenixChat.User

    timestamps
  end
...
```

And now we have message persistence to our Postgres database. You'll also notice that when write messages in your `phoenix-chat` component, the messages will appear on the correct side. That's because we're now passing in the `from` field, which our component is using to determine on which side the messages displays.

Create Anonymous User Model

- Create anonymous users with fake data
- Create and update models

At this point in our app, we have a functioning chat component and a backend that can handle messages. The next step is to add our anonymous users to a list and add them to our admin dashboard so our admin can start interacting with them.

But before we do that, we should give our anonymous users some fake information. Since it's hard to remember users based on uuids, we're going to give our users fake names and avatars for easier reference (if it's not clear to you why were are doing this, you'll see in a few lessons).

Add Faker

We're going to generate fake user data using the ubiquitous Faker package, which has an Elixir version. It's worth glancing over the docs to see how much fake data we have access to. Let's add that now.

```
/mix.exs
commit: coming soon
```

```
...
  def application do
    [mod: {PhoenixChat, []},
      applications: [
        :comeonin,
        :cowboy,
        :faker,
        :gettext,
        ...
      ]]
  end
...
  defp deps do
    [
      {:comeonin, "~> 2.3"},
      {:corsica, "~> 0.4"},
      {:cowboy, "~> 1.0"},
      {:faker, "~> 0.7"},
      {:gettext, "~> 0.11"},
      ...
    ]
  end
...
```

Then be sure to run `mix deps.get` and restart your server.

```
$ mix deps.get
```

The next step is to generate two migrations: one to create an `AnonymousUsers` model and the other to associate that anonymous user with our existing `Messages` model.

```
$ mix ecto.gen.migration create_anonymous_user
$ mix ecto.gen.migration message_belongs_to_anonymous_user
```

We'll start with the anonymous user migration. We want to use the `uuid` as the primary key, which is a way of telling Ecto that the `uuid` is the unique identifier for each anonymous user. Ecto will, by default, use `:id` and a `:integer` as the primary key. Each user will also have a name and an avatar (both of which will be fake).

```
/priv/repo/migrations/2...9_create_anonymous_user.exs
commit: coming soon
```

```
defmodule PhoenixChat.Repo.Migrations.CreateAnonymousUsers do
  use Ecto.Migration

  def change do
    create table(:anonymous_users, primary_key: false) do
      add :id, :uuid, primary_key: true
      add :name, :string
      add :avatar, :string

      timestamps
    end
  end
end
```

This should look familiar to you since it's basically the same as the other migrations we created. The next migration is a little bit more interesting and introduces a new concept that we haven't touched on before.

We need to `alter` the existing `:messages` table to add a reference to the new `:anonymous_users` we just created and remove the temporary `:from` field we added previously since we are now storing references from a separate table.

To do this, we need to create both `up` and `down` blocks. The code in the up block runs a migration as you would expect when you run `mix ecto.migrate`. The down block runs what is called a "down" migration, which you can think of as an "undo". In the event you screw up a migration, you can "rollback" your migrations and re-run them (hopefully) with the right data the next time.

In our `up` migration, we're adding the `:anonymous_user_id` to the `:messages` table and declaring that it references the `:anonymous_user` table we just created. We're also saying that when the referring anonymous user is deleted (`on_delete`), change all messages references to that user (which no longer exists) to `nil`.

In our `down` migration, we're saying, "if we have to undo this migration, remove the `:anonymous_user_id` from the messages table and add `:from` back to the table".

```
/priv/repo/migrations/2...9_message_belongs_to_anonymous_user.exs
commit: coming soon
```

```
defmodule PhoenixChat.Repo.Migrations.MessageBelongsToAnonymousUser do
  use Ecto.Migration

  def up do
    alter table(:messages) do
      add :anonymous_user_id, references(:anonymous_users, on_delete: :nilify_all, type: :u
      remove :from
    end
  end

  def down do
    alter table(:messages) do
      remove :anonymous_user_id
      add :from, :string
    end
  end
end
```

Now that we have our migrations, we need to set up our `AnonymousUser` model and update our `Message` model to handle the new association.

Create and update models

Let's start by creating our `AnonymousUser` model.

```
$ touch web/models/anonymous_user.ex
```

Much of this should look familiar. We're creating `has_many` (docs) reference from our anonymous user to her messages, saying that each anonymous user can have many messages associated with her.

The only tricky part is that we're telling our model explicitly not to auto-generate a primary key since we're providing one with the uuid. The uuid is binary (a text string), so we need to specify `@foreign_key_type` to `:binary_id` because it defaults to `:integer`. In sum, we're telling `@primary_key` that the `:id` should not `autogenerate` and that it's a `:binary_id`.

In our `changeset/2`, we're referencing two functions that don't exist yet. We will create those below.

```
/web/models/anonymous_user.ex
commit: coming soon
```

```
defmodule PhoenixChat.AnonymousUser do
  use PhoenixChat.Web, :model

  alias PhoenixChat.Message

  @primary_key {:id, :binary_id, autogenerate: false}
  @foreign_key_type :binary_id

  schema "anonymous_users" do
    field :name
    field :avatar
    has_many :messages, Message

    timestamps
  end

  def changeset(model, params \\ :empty) do
    model
    |> cast(params, ~w(id), ~w())
    |> put_avatar
    |> put_name
  end
end
```

Now we need to create two functions to add a name and an avatar to our new anonymous user: `put_name/1` and `put_avatar/1`.

In `put_name/1`, we start by generating a fake name for our user. We do this by using Faker to generate a random color and a buzzword_suffix, which gives us fake names like:

- Orange Alliance
- Blue Framework
- Purple Website

We don't care if they're unique since these are just for visual reference. From here, we use `put_change/2` to add the `:name` to our `changeset/2`.

Then we use `put_avatar/1` to do the same thing but for a 25 by 25 pixel image using Faker's image_url.

```
/web/models/anonymous_user.ex
commit: coming soon
```

```
defmodule PhoenixChat.AnonymousUser do

  ...

  defp put_name(changeset) do
    adjective = Faker.Color.name |> String.capitalize
    noun = Faker.Company.buzzword_suffix |> String.capitalize
    name = adjective <> " " <> noun
    changeset
    |> put_change(:name, name)
  end

  defp put_avatar(changeset) do
    changeset
    |> put_change(:avatar, Faker.Avatar.image_url(25, 25))
  end
end
```

We're also going to add an additional function that will come into play later. Rather than keep a separate table called something like `:lobby`, we can figure out who our recently active users are by finding the most recent messages and returning the users associated with those messages, with a default value of the 20 most recent users (detailed explanation below the code).

```
/web/models/anonymous_user.ex
commit: coming soon
```

```
defmodule PhoenixChat.AnonymousUser do

  ...

  @doc """
  This query returns users with the most recent messages up to a given limit.
  """
  def recently_active_users(limit \\ 20) do
    from u in __MODULE__,
      left_join: m in Message, on: m.anonymous_user_id == u.id,
      distinct: u.id,
      order_by: [desc: u.inserted_at, desc: m.inserted_at],
      limit: ^limit
  end
end
```

If you've worked a lot with SQL, this will look familiar to you. If this looks complicated, we'll go through it line-by-line.

In the first line, we start with `from u in __MODULE__`, which is our way of telling Ecto that we want to

search through anything that matches our `AnonymousUser` struct. In other words, it will look through all of our anonymous users that we have stored in the database.

The next line uses a `left_join`, which is a SQL term that compares two tables and returns the first table along with anything that matches from the second table. For an excellent example of a left join, check out the docs from W3 Schools. In our case, we want to take our anonymous user's id and find all the messages associated with that user so we can determine which anonymous users have most recently sent a message.

Then we use `distinct` to make sure that all of the anonymous users are unique based on their ids. It's worth noting that that not all databases support `distinct`. We're using Postgres, which does support it.

Then we pass in `order_by` along with two elements in a list, which tell our anonymous users that we want to order them by most recent message, then by most recent addition to the database in descending order based on timestamp (`inserted_at`). This way, we prioritize users that have active conversations but still show users even if they haven't sent a message.

Finally, we `limit` the results to whatever was passed in, or default to 20 users. The caret (`^`) is required when passing a value into an Ecto query. Without the caret, Ecto will think the variable is part of the query syntax and not a value that you passed in.

Another thing to note is that Ecto does not care in which order you pass these arguments because it will prepare a SQL statement for you with the proper ordering. So, for example, you can have you `limit` statement before your `order_by` and it won't cause your results to be limited to 20 before they're ordered.

The last thing we need to do is update our `Message` model to handle the new association with anonymous users. The only difference here from previous associations is that we once again need to set the primary key using `:type` to a `:binary_id` since our uuid is a string.

```
/web/models/message.ex
commit: coming soon
```

```
defmodule PhoenixChat.Message do
  use PhoenixChat.Web, :model

  alias PhoenixChat.{DateTime, User, AnonymousUser}

  schema "messages" do
    field :body, :string
    field :timestamp, DateTime
    field :room, :string

    belongs_to :user, User
    belongs_to :anonymous_user, AnonymousUser, type: :binary_id

    timestamps
  end

  @required_fields ~w(body timestamp room)
  @optional_fields ~w(anonymous_user_id user_id)

  ...
end
```

Before you move on to the next lesson, be sure to run your migration.

```
$ mix ecto.migrate
```

Now we have a way to add anonymous users to our database, associate them with incoming messages, and return a list of our most recently active users. Keep in mind that the app is currently broken since we haven't updated our controllers.

Persist Anonymous Users

- Add anonymous users to admin channel
- Validate params

At this point, we have an `AnonymousUser` model and a `changeset/2` to add them to the database, but we aren't triggering this addition anywhere. In this section, we trigger a database addition when an anonymous user joins our `admin:active_users` topic created previously.

We're also going to remove our admin presence from our `active_users` to make our frontend code simpler (so we don't need to filter out admins any longer) and so we aren't passing along any extra data. We're still keeping track of the admin presence on the backend and we will use that later on.

We also add some simple validations on the params we pass on our socket connection. For example, we want to raise an error if both the id and uuid params are empty. This helps to simplify our code and lets the frontend know if something is missing.

Add anonymous users to admin channel

We'll start by updating our `join/3` function to pass along the socket with the id of the user who joined the channel (either admin or anonymous user).

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  @moduledoc """
  The channel used to give the administrator access to all users.
  """

  use PhoenixChat.Web, :channel
  require Logger

  alias PhoenixChat.{Presence, Repo, AnonymousUser}

  @doc """
  The `admin:active_users` topic is how we identify all users currently using the app.
  """
  def join("admin:active_users", payload, socket) do
    authorize(payload, fn ->
      send(self, :after_join)
      id = socket.assigns[:uuid] || socket.assigns[:user_id]
      {:ok, %{id: id}, socket}
    end)
  end

  ...
end
```

Then we use pattern matching to match users that have a `:user_id` (which are our admins) and return the socket without adding any data. This is effectively filtering out our admins on the backend so we don't need to run the filter on the frontend and it ensures we aren't sending extra, unnecessary data to our frontend. Recall that an underscore denotes a variable that will not be used in the body of the function (`_user_id`). If you do not include this underscore, you will get a warning that there is an unused variable.

Then we create a new `handle_info/2` function that matches a `:uuid` for an anonymous user. As soon as this anonymous user joins, we check to make sure that the user is saved to the database using `ensure_user_saved`, which we define below. Then, just as before, we track the presence state of this user and push it to our socket, with the only difference that we are now sending a `uuid` rather than a generic `id` since we are now only sending anonymous users.

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  ...

  @doc """
  This handles the `:after_join` event and tracks the presence of the socket that
  has subscribed to the `admin:active_users` topic.
  """
  def handle_info(:after_join, %{assigns: %{user_id: _user_id}} = socket) do
    push socket, "presence_state", Presence.list(socket)
    {:noreply, socket}
  end

  def handle_info(:after_join, %{assigns: %{uuid: uuid}} = socket) do
    ensure_user_saved!(uuid)

    push socket, "presence_state", Presence.list(socket)
    Logger.debug "Presence for socket: #{inspect socket}"
    {:ok, _} = Presence.track(socket, uuid, %{
      online_at: inspect(System.system_time(:seconds))
    })
    {:noreply, socket}
  end
end
```

In our `ensure_user_saved/1` function, we need to check if the user already exists. If the user does exist, then we do nothing. If the user does not exist, then we create a new user with the `uuid` passed in as a parameter. Recall that to add something to the database, you first need to create a changeset, then pass that changeset to `Repo` to make the change.

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  ...

  defp ensure_user_saved!(uuid) do
    user_exists = Repo.get(AnonymousUser, uuid)
    unless user_exists do
      changeset = AnonymousUser.changeset(%AnonymousUser{}, %{id: uuid})
      Repo.insert!(changeset)
    end
  end
end
```

So now, when an anonymous user joins a channel for the first time we add them to the database. You should be careful where you add validations like this because they can quickly get out of control. For

example, imagine if we added this validation on every message that was received rather than on join. This would accomplish the same thing but would require dozens/hundreds of additional database reads per user.

Validate params

It's generally good practice to validate your parameters when you receive data from a user. In our `user_socket`, go ahead and add a `validate_params/1` function (defined later) and remove the `:user_id` from our socket since we are no longer passing along admin presence data.

```
/web/channels/user_socket.ex
commit: coming soon
```

```
defmodule PhoenixChat.UserSocket do
  ...

  def connect(params, socket) do
    user_id = params["id"]
    user = user_id && Repo.get(User, user_id)
    validate_params!(params)

    socket = if user do
        socket
        |> assign(:user_id, user_id)
        |> assign(:username, user.username)
        |> assign(:email, user.email)
      else
        socket
        |> assign(:uuid, params["uuid"])
      end

    {:ok, socket}
  end

  ...
end
```

Now we need to define our `validate_params/1` function. Ideally, we want to raise an error as early as possible so we can track it down easier. In this case, we're pattern matching our `id` and `uuid` and checking to make sure that they aren't empty. If they are, we `raise` an error that explicitly tells us what the problem is.

If the parameters don't match anything, then we don't do anything (`do: nil` is a "no operation" or "noop").

/web/channels/user_socket.ex
commit: coming soon

```
defmodule PhoeniChat.UserSocket do

  ...

  @empty ["", nil]
  defp validate_params!(%{"id" => id, "uuid" => uuid})
  when id in @empty or uuid in @empty do
    raise "id or uuid must not be empty"
  end

  defp validate_params!(_), do: nil
end
```

At this point, our app is storing anonymous users when they join for the first time and we are sending only the anonymous user's data to our frontend. The next step is to update our `room_channel` to make sure we're sending along the new `name` and `avatar` to our frontend.

Broadcast Active and Inactive Users

- Poison.Encoder and \@derive
- Broadcast lobby_list

Now that we can receive messages, add anonymous users to the database, and keep track of recent users based on the date of the last message they sent, we need to broadcast to our admins every time a new user joins. Our frontend will then be able to listen to this event and handle updates accordingly.

This event will get triggered every time a user joins the `admin:active_users` topic so it will be up to the frontend to check whether a user is already on the list.

We're also going to do a little bit of refactoring to use the `@derive` module attribute, which is a convenient way to customize/instruct how Protocols treat our struct (explained in detail later). This is not strictly necessary, but makes for cleaner code.

Reactor Message and RoomChannel

The first thing we're going to do is add `@derive` to our `Message` model. `@derive` (docs) is a module attribute that allows you to customize how the struct is treated when it interacts with a Protocol (in our case, our JSON API). We're going to instruct `Poison.Encode` to only encode certain fields and ignore the rest.

This will save us from writing a bunch of code for specifying which fields to include in our JSON version of our structs (which is what we're currently doing in our `message_payload/1` function).

```
/web/models/message.ex
commit: coming soon
```

```
defmodule PhoenixChat.Message do
  use PhoenixChat.Web, :model

  alias PhoenixChat.{DateTime, User, AnonymousUser}

  @derive {Poison.Encoder, only: ~w(id body timestamp room user_id anonymous_user_id)a}

  ...
end
```

So now, we send message data using our API, we're only going to send `id`, `body`, `timestamp`, `room`, `user_id`, and `anonymous_user_id`.

We should also update our `AnonymousUser` to use `@derive` as well to send `id`, `name`, and `avatar`.

```
/web/models/anonymous_user.ex
commit: coming soon
```

```
defmodule PhoenixChat.AnonymousUser do
  use PhoenixChat.Web, :model

  alias PhoenixChat.Message

  @primary_key {:id, :binary_id, autogenerate: false}
  @foreign_key_type :binary_id
  @derive {Poison.Encoder, only: ~w(id name avatar)a}

  ...

end
```

Now that we've defined what data we're sending from within our model, we no longer need a `message_payload/1` function to define our message payload. Let's go ahead and delete that from our `RoomChannel`.

```
/web/channels/room_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.RoomChannel do
  use PhoenixChat.Web, :channel
  require Logger

  alias PhoenixChat.{Message, Repo}

  def join("room:" <> room_id, payload, socket) do
    authorize(payload, fn ->
      messages = room_id
        |> Message.latest_room_messages
        |> Repo.all
        |> Enum.reverse
      {:ok, %{messages: messages}, socket}
    end)
  end

  ...
end
```

Then let's pass `anonymous_user_id` as the `uuid` instead of the `from` field and remove the extra code that called our no-longer-existing `message_payload/1` function.

/web/models/room_channel.ex
commit: coming soon

```
defmodule PhoenixChat.RoomChannel do
  ...

  def handle_in("message", payload, socket) do
    payload = payload
      |> Map.put("user_id", socket.assigns[:user_id])
      |> Map.put("anonymous_user_id", socket.assigns[:uuid])
    changeset = Message.changeset(%Message{}, payload)

    case Repo.insert(changeset) do
      {:ok, message} ->
        broadcast! socket, "message", message
        {:reply, :ok, socket}
      {:error, changeset} ->
        {:reply, {:error, %{errors: changeset}}, socket}
    end
  end
end
```

And now if you refresh your frontend, you should have a functioning app again.

Broadcast lobby_list

This where we use the `recently_active_users/0` function we defined earlier to get the 20 most recently active anonymous users. Recall that this works by looking through our `Messages`, finds the most recent message from each anonymous user, and returns the 20 users that have sent or received messages most recently.

After we make that query, we pass the list of 20 users along with our socket.

/web/channels/admin_channel.ex
commit: coming soon

```
defmodule PhoenixChat.AdminChannel do
  ...

  @doc """
  The `admin:active_users` topic is how we identify all users currently using the app.
  """
  def join("admin:active_users", payload, socket) do
    authorize(payload, fn ->
      send(self, :after_join)
      id = socket.assigns[:uuid] || socket.assigns[:user_id]
      lobby_list = AnonymousUser.recently_active_users |> Repo.all
      {:ok, %{id: id, lobby_list: lobby_list}, socket}
    end)
  end

 ...

end
```

Next, we need to broadcast these changes to our frontend. The best time to do this is after a new user joins the channel, so we can hook into our `:after_join` pattern match in `handle_info` to add the broadcast.

We should also update our `ensure_user_saved/1` to return the user after the query so we can pass that user along to our `broadcast`.

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  ...

  def handle_info(:after_join, %{assigns: %{uuid: uuid}} = socket) do
    user = ensure_user_saved!(uuid)

    broadcast! socket, "lobby_list", user

    push socket, "presence_state", Presence.list(socket)
    Logger.debug "Presence for socket: #{inspect socket}"
    {:ok, _} = Presence.track(socket, uuid, %{
      online_at: inspect(System.system_time(:seconds))
    })
    {:noreply, socket}
  end

  defp ensure_user_saved!(uuid) do
    user_exists = Repo.get(AnonymousUser, uuid)
    if user_exists do
      user_exists
    else
      changeset = AnonymousUser.changeset(%AnonymousUser{}, %{id: uuid})
      Repo.insert!(changeset)
    end
  end
end
```

But at the moment, we're going to broadcast these events to everyone who is subscribed to the channel, but we actually only want to send this information to our admins. We're going to do this using `intercept/1` (docs), which allows us to hook into outgoing responses and manipulates them. In this case, we want to hook into any broadcast to `lobby_list` and only broadcast it to administrators (not anonymous users).

When the intercept happens, it passes to the `handle_out` function, in which we check if the user is an admin (has a `user_id`). If so, we proceed. Otherwise, we return a `{:noreply, socket}`.

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  ...

  intercept ~w(lobby_list)

  ...

  @doc """
  Sends the lobby_list only to admins
  """
  def handle_out("lobby_list", payload, socket) do
    assigns = socket.assigns
    if assigns[:user_id] do
      push socket, "lobby_list", payload
    end
    {:noreply, socket}
  end

  ...
end
```

And that's all we have to change in our `admin_channel`. We're now broadcasting the latest 20 active users on join and sending an updated list any time a new user joins. Now we should head over to the frontend to implement these changes.

Connect the Frontend to AdminChannel

- Connect frontend
- Display active users

Now that our backend is keeping track of active users, we need to connect our frontend to list them. From there, our admin can select a conversation and respond to incoming chats from anonymous users.

First we'll update our `phoenix-chat` component to add the `AdminChannel` to the `configureChannels` function. That way, when a user opens the chat window, they are subscribed to the `active_users` topic, which adds them to the list of active users stored in ETS and keeps track of their status with Presence.

Then we need to update our `phoenix-chat-frontend` to list all users, make them selectable, retrieve messages, and give our admin the ability to respond.

Updating phoenix-chat

Our update to the `phoenix-chat` component is pretty simple. Within the `configureChannels` function, all we have to do is add a new channel called `adminChannel` and join it. Then on `componentWillUnmount`, we'll leave the channel.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
  ...
export class PhoenixChat extends React.Component {

  componentWillUnmount() {
    this.channel.leave()
    this.adminChannel.leave()
  }

  configureChannels() {
    this.channel = this.socket.channel(`room:${room}`)
    ...

    this.adminChannel = this.socket.channel(`admin:active_users`)
    this.adminChannel.join()
      .receive("ok", () => {
        console.log(`Succesfully joined the active_users topic.`)
      })
  }

  ...
}
...
```

And since we got rid of the `from` field, we need to update the `from` constant to determine on which side our messages should appear.

/src/PhoenixChat.jsx
commit: coming soon

```
...

export class PhoenixChatSidebar extends React.Component {
  ...

  render() {
    const list = !this.props.messages
      ? null
      : this.props.messages.map(({ body, id, anonymous_user_id }, i) => {
      const right = anonymous_user_id === localStorage.phoenix_chat_uuid

      ...
    })
  }

}
...
```

And that's all we have to change on our `phoenix-chat` component to connect to the admin channel.

Adding AdminChannel to Chat

Now we need to add `Presence` to our `Chat` component, which can then pass it down to the `Sidebar` component, which renders the list of active users. We are going to handle all of this in local state for the time being, but we may eventually move this into Redux if it makes sense.

Whent the component mounts, we want to initialize the socket and set up the `adminChannel`, which we do within the `configureAdminChannel` function.

Within `configureAdminChannel`, we set the topic to `active_users` and listen for a `presence_state` or `presence_diff` action, which lets us know a user joins or leaves the socket. Whenever a change occurs, we update the `this.state.presences` object, which triggers a re-render.

Then we pass `this.state.presences` as props to our `Sidebar` component. Finally we connect our `Chat` component to Redux and map the state of the user to the Redux `state`. It's a lot of code, but you've seen most of it before.

Note: we're eventually going to move all of this logic to Redux. Generally speaking, when you find yourself working with components that contain their own logic, you probably want to consider moving that logic to Redux.

```
$ npm install --save phoenix
```

```
/app/components/Chat/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import { Socket, Presence } from "phoenix"
import { connect } from "react-redux"
import style from "./style.css"

import { default as Sidebar } from "../Sidebar"

export class Chat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      presences: {}
    }
  }

  componentDidMount() {
    const params = this.props.user
    this.socket = new Socket("ws://localhost:4000/socket", { params })
    this.socket.connect()
    this.configureAdminChannel()
  }

  ...

}

const mapStateToProps = state => ({
  user: state.user
})

export default connect(mapStateToProps)(cssModules(Chat, style))
```

Now configure the admin channel.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
  ...

  configureAdminChannel() {
    this.adminChannel = this.socket.channel("admin:active_users")
    this.adminChannel.on("presence_state", state => {
      const presences = Presence.syncState(this.state.presences, state)
      console.log('Presences after sync: ', presences)
      this.setState({ presences })
    })
    this.adminChannel.on("presence_diff", state => {
      const presences = Presence.syncDiff(this.state.presences, state)
      console.log('Presences after diff: ', presences)
      this.setState({ presences })
    })
    this.adminChannel.join()
      .receive("ok", ({ id }) => {
        console.log(`${id} succesfully joined the active_users topic.`)
      })
  }

  ...
}
...
```

And finally render the `Chat` component.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
...

  render() {
    return (
      <div>
        <Sidebar
          presences={this.state.presences} />
        <div className={style.chatWrapper}>
          chat me
        </div>
        { this.props.children }
      </div>
    )
  }
}
...
```

The next thing we need to do is list our presences and users in our `Sidebar`. We're going to create two functions within this component: `listBy` and `renderList`. The first function orders our data and the second returns the jsx that we will render. We're going to use a lot from the new ES205 syntax, so a more detailed explanation of each is provided below the codeblock.

Keep in mind that functions should sometimes live outside of the component. This is because React will always redraw anything that is in a nested function because each time `Sidebar` renders, it creates a completely new constant called `listBy`. This optimization doesn't usually make a difference, but it's worth keeping in mind when you have components that are rendered a lot.

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import { Presence } from "phoenix"
import style from "./style.css"

const listBy = (id, { metas: [first, ...rest] }) => {
  first.count = rest.length + 1
  first.id = id
  return first
}

const renderList = props => {
  return Presence.list(props.presences, listBy)
    .map(({ id }) => {
      return (
        <div key={id}>
          { id }
        </div>
      )
    })
}

export const Sidebar = props => {
  return (
    <div className={style.sidebar}>
      { renderList(props) }
    </div>
  )
}

export default cssModules(Sidebar, style)
```

The `renderList` function is the return value from `Phoenix.list`. Note that `Phoenix.list` here is different on the frontend than the backend. As in, it's not this list, it's this one. There currently isn't much in the way of docs, but that's the price you pay for working with cutting-edge frameworks.

From the docs:

> By default, all presence metadata is returned, but a `listBy` function can be supplied to allow the client to select which metadata to use for a given presence.

So within our `listBy` function, we take in `props.presence`. Each `presence` object has the keys `id` and `metas`. `metas` contains a list of all the devices with which the user has been present. We are going to use our `listBy` function to prioritize the first device/tab registered for each user. This is not especially important for us since each anonymous user will generate a new `uuid` from each device, but we will do it in case we need to track the presence of our admin.

We use object destructuring assignment and a spread operator to pull out the `first` value from our list

of `metas`. From there we are going to add the `count` field, which lets us know how many devices are connected and the `id` of the user's presence.

Now, back to our `renderList`. After passing through `Phoenix.list` and going through the `listBy` function, we have a list of presences that have been filtered down to one instance. We then `map` over the list and return jsx objects for each presence.

Next we're going to make these users click-able so an admin can connect to a channel to receive and respond to message from anonymous users.

Join Room and Receive Messages

- Connect to chatroom
- Receive messages

Now that we have a list of all active presences (we will eventually expand this to include all users, not just currently active presences), we need to give our admin the ability to select one and respond. The UI for this will be clicking on a `uuid` on the left that represents a user.

This will subscribe the admin to the topic and allow her to respond to the user. It will also grab the most recent 10 messages from the chatroom and use that to populate our `Chat` component with `messages`.

Create ChatRoom component

Our `Chat` component is already starting to get complicated, so we're going to create a new component called `ChatRoom` to handle the actual rendering of messages and we'll leave the socket logic in `Chat` for now.

```
$ mkdir app/components/ChatRoom
$ touch app/components/ChatRoom/{index.js,style.css,README.md,spec.js}
```

Our `ChatRoom` component is going to receive a list of messages as `props`, which it will then render to the page. Our `renderMessages` function maps over each of these messages and returns a jsx component for each.

```
/app/components/ChatRoom/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

export class ChatRoom extends React.Component {
  componentDidUpdate() {
    if (this.props.messages.length > 0) {
      const lastMessage = this[`chatMessage:${this.props.messages.length - 1}`]
      this.chatContainer.scrollTop = lastMessage.offsetTop
    }
  }

  render() {
    return (
      <div
        ref={ref => { this.chatContainer = ref }}
        className={style.chatWrapper}>
        { this.renderMessages(this.props) }
      </div>
    )
  }
}

export default cssModules(ChatRoom, style)
```

Then we should add a function to map over each of our messages and render them.

/app/components/ChatRoom/index.js
commit: coming soon

```
...
export class ChatRoom extends React.Component {
  ...
  renderMessages() {
    return this.props.messages.map(({ body, id }, i) => {
      return (
        <div
          ref={ref => { this[`chatMessage:${i}`] = ref }}
          key={id}>
          { body }
        </div>
      )
    })
  }
  ...
}
...
```

Then we should add a wrapper to keep this component visible next to the `Sidebar`. Go ahead and remove the `.chatWrapper` from `../Chat/style.css` as well.

/app/components/ChatRoom/style.css
commit: coming soon

```
.chatWrapper {
  margin-left: 300px;
}
```

The next step is to connect our `Chat` component to our room channel and render our new `ChatRoom` component.

Connecting to channel

Now we need to connect to our room channel, create a `changeRoom` function that we can pass to our `Sidebar` that will allow us to change rooms, and render our `ChatRoom` component with the list of messages we get from our socket.

/app/components/Chat/index.js
commit: coming soon

```
...

import { default as ChatRoom } from "../ChatRoom"

export class Chat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      presences: {},
      messages: [],
      currentRoom: ""
    }
    this.changeChatroom = this.changeChatroom.bind(this)
  }

  ...

  changeChatroom(room) {
    this.channel = this.socket.channel(`room:${room}`)
    this.setState({
      messages: []
    })
    this.configureRoomChannel(room)
  }
  ...
}
```

Now we need to configure our room channel to take in this new value.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...
export class Chat extends React.Component {
  ...

  configureRoomChannel(room) {
    this.channel.join()
      .receive("ok", ({ messages }) => {
        console.log(`Succesfully joined the ${room} chat room.`, messages)
        this.setState({
          messages,
          currentRoom: room
        })
      })
      .receive("error", () => { console.log(`Unable to join the ${room} chat room.`) })

    this.channel.on("message", payload => {
      this.setState({
        messages: this.state.messages.concat([payload])
      })
    })
  }

  ...
}
...
```

And finally we need to pass our click handler to our `Sidebar` and pass our messages to our `ChatRoom` component.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...
export class Chat extends React.Component {
  ...
  render() {
    return (
      <div>
        <Sidebar
          presences={this.state.presences}
          onRoomClick={this.changeChatroom} />
        <ChatRoom messages={this.state.messages} />
        { this.props.children }
      </div>
    )
  }
  ...
}
...
```

The last change we need to make is to add an `onClick` to each of our presence components in the `Sidebar` list.

/app/components/Sidebar/index.js
commit: coming soon

```
...
const renderList = props => {
  return Presence.list(props.presences, listBy)
    .map(({ id }) => {
      return (
        <div
          onClick={() => { props.onRoomClick(id) }}
          key={id}>
          { id }
        </div>
      )
    })
}
...
```

At this point, our admin can receive messages. You should see an anonymous user's id appear in the `Sidebar` and you should be able to click on that and see incoming messages. You can send messages to this admin from your `phoenix-chat` component.

The next step is to allow our admin to respond.

Change Room and Respond to Messages

- Respond to messages

Now that we have the ability to receive messages from our anonymous users, we need to be able to respond to them.

Responding to messages

In order to respond to messages, we're going to need an input and a way to submit that message. We're going to use a controlled input to handle our message data. We will cover controlled inputs in greater detail in a future lesson, but for now you can think of them as a way to store values in state rather than in the DOM element itself.

The first function we're adding is `handleMessageSubmit`, which we will attach to an event (`e`) on `onKeyDown` on our input. If the key that was pressed happens to have the `keyCode` of 13, then it's the `return` key. If the `return` key is pressed, we are currently in a room, and there is currently a value in the input, we push a `message` to the room channel with the current room, the current value of the input, and a timestamp. Then we reset the input's value.

The second function is `handleChange`, which listens for a change to the input field and updates `this.state.input` with the new value.

The last function we're adding creates the input element.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      presences: {},
      messages: [],
      input: "",
      currentRoom: ""
    }
    this.changeChatroom = this.changeChatroom.bind(this)
    this.handleMessageSubmit = this.handleMessageSubmit.bind(this)
    this.handleChange = this.handleChange.bind(this)
  }

  ...

  handleChange(e) {
    this.setState({ input: e.target.value })
  }

  handleMessageSubmit(e) {
    if (e.keyCode === 13 && this.state.currentRoom && this.state.input) {
      this.channel.push("message", {
        room: this.state.currentRoom,
        body: this.state.input,
        timestamp: (new Date()).getTime()
      })
      this.setState({ input: "" })
    }
  }
  ...

}
```

Then we need to pass along the handler functions to our `ChatRoom` component so we can connect them to the `onKeyDown` and `onChange` events of our input.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
  ...

  render() {
    return (
      <div>
        <Sidebar
          presences={this.state.presences}
          onRoomClick={this.changeChatroom} />
        <ChatRoom
          input={this.state.input}
          currentRoom={this.state.currentRoom}
          handleChange={this.handleChange}
          handleMessageSubmit={this.handleMessageSubmit}
          messages={this.state.messages} />
        { this.props.children }
      </div>
    )
  }
}

...
```

The next step is to add a basic label to each of our messages so we can tell difference between messages sent from our anonymous users and our admin.

```
/app/components/ChatRoom/index.js
commit: coming soon
```

```
import React from "react"
import cssModules from "react-css-modules"
import style from "./style.css"

export class ChatRoom extends React.Component {
  ...

  renderMessages() {
    return this.props.messages.map(({
      body,
      id,
      user_id,
      anonymous_user_id
    }, i) => {
      const from = user_id ? 'Me' : anonymous_user_id.substring(0,10)
      const msg = `${from}: ${body}`
      return (
        <div
          ref={ref => { this[`chatMessage:${i}`] = ref }}
          key={id}>
          { msg }
        </div>
      )
    })
  }
  ...

}
...
```

Now we should add an input box so our admin can respond to messages sent by the anonymous users. We're going to take the event handlers we passed in from our `Chat` component and attach them to events in the input box. Recall that when an `onKeyDown` event is triggered with the return key, the message is submitted.

```
/app/components/ChatRoom/index.js
commit: coming soon
```

```
...

export class ChatRoom extends React.Component {
  ...

  renderInput() {
    if (!this.props.currentRoom) return null
    return (
      <div className={style.inputWrapper}>
        <input
          value={this.props.input}
          onKeyDown={this.props.handleMessageSubmit}
          onChange={this.props.handleChange}
          className={style.input} />
      </div>
    )
  }

  render() {
    return (
      <div className={style.container}>
        <div
          ref={ref => { this.chatContainer = ref }}
          className={style.chatWrapper}>
          { this.renderMessages(this.props) }
        </div>
        { this.renderInput(this.props) }
      </div>
    )
  }
}

...
```

Let's also add some basic styling to make our input easier to use.

```
/app/components/ChatRoom/style.css
commit: coming soon
```

```
.container {
  position: relative;
  height: 100vh;
  margin-left: 300px;
}

.chatWrapper {
  position: relative;
  height: calc(100vh - 60px);
  padding-left: 20px;
  padding-top: 10px;
  padding-bottom: 10px;
  overflow-y: scroll;
}

.input {
  position: absolute;
  bottom: 20px;
  left: 20px;
  width: calc(100% - 40px);
  line-height: 40px;
  font-size: 20px;
  outline: none;
  border: 1px solid #ccc;
  border-radius: 3px;
  padding-left: 10px;
  color: #333;
}
```

Now when you enter information into the input and press `return`, you can send messages back to the anonymous user.

List Active and Inactive Users

- Listing inactive users
- Merging with active users from Presence
- Distinguish admin messages

Currently, our lobby only lists active users that we are tracking with presence. You may recall that we are also storing all of our users (not just the active users) in Postgres. What we would like to do is track presence and the `lobbyList` separately and use presence to highlight users in the list that are currently active and move them to the top of the list.

Fortunately, this is not difficult.

List inactive users

All we have to do is update our `configureAdminChannel` to handle `lobby_list` changes and update a `state` variable called `lobbyList` when things change.

We're also going to leave our channels when the component unmounts so we don't accidentally keep the connection alive for longer than necessary.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      presences: {},
      messages: [],
      input: "",
      currentRoom: "",
      lobbyList: []
    }
    ...
  }

  componentWillUnmount() {
    if (this.channel) this.channel.leave()
    if (this.adminChannel) this.adminChannel.leave()
  }

  configureAdminChannel() {
    this.adminChannel = this.socket.channel("admin:active_users")

    ...

    this.adminChannel.on("lobby_list", (user) => {
      if (!this.state.lobbyList.includes(user)) {
        this.setState({ lobbyList: this.state.lobbyList.concat([user]) })
      }
    })
    this.adminChannel.join()
      .receive("ok", ({ id, lobby_list }) => {
        console.log(`${id} succesfully joined the active_users topic.`)
        this.setState({ lobbyList: lobby_list })
      })
  }

  ...

}
```

Then we need to pass our `lobbyList` that we get from our backend and store in local `state` to our `Sidebar` component. We will eventually refactor this and move this data into Redux, but are handling it locally now to make it easier to see where the data is coming from.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...

export class Chat extends React.Component {
  ...

  render() {
    return (
      <div>
        <Sidebar
          currentRoom={this.state.currentRoom}
          presences={this.state.presences}
          lobbyList={this.state.lobbyList}
          onRoomClick={this.changeChatroom} />
        <ChatRoom
          input={this.state.input}
          handleChange={this.handleChange}
          handleMessageSubmit={this.handleMessageSubmit}
          currentRoom={this.state.currentRoom}
          messages={this.state.messages} />
        { this.props.children }
      </div>
    )
  }
}
```

So at this point, we're getting a list of all anonymous users in the lobby and passing it along to our `Sidebar`. The next step is to take those users and render them in a list.

Merge with Presence

We're going to add a bunch of new functions that will allow us to order our users in a meaningful way, then find users who are active and give them some special indicator.

The first function we're changing is our `listBy` function. Since we don't care about the number of devices, we're just going to pass in an anonymous function that returns the `id`.

Next we're creating an `orderByActivity` function, which we're going to use to order our list of users. The way this works is it takes in two values. If both users have the same activity, it doesn't change the position. If the users are not the same (one is active, one is not), then if the current user is active (`a.active === true`), we push it up one on the list. Otherwise, the user is not active and we push it down one on the list.

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...

const orderByActivity = (a, b) => {
  if (a.active === b.active) return 0
  if (b.active === true) return 1
  return -1
}

...
```

Now we need to update our `renderList` function to render our users. The `activeList` const is a list of all our active users. We're eventually going to use this list of users to compare against our list of all users. When there's a match, that means that the user is active and we will format that user differently. If no match, then the user belongs in the list but is not currently present on the socket.

The `lobbyList` function is where we merge the two lists together to determine which users are active and which are not. We start by mapping over each user in `props.lobbyList` (which contains all users that have joined the lobby), then we check to see if `activeList` (which contains all our present users) contains the current user with Array.includes. If the user ids match, then the user is deemed "active", so we set `active` to `true` (`Array.includes` returns `true` or `false`).

The last function is `renderList`, which now uses Array.sort and our `orderByActivity` function to put the most recently active users at the top of the list. Then we `map` over each and assign an inset `box-shadow` to all active users. Finally, we return the jsx object to render in the list.

Also, now that we have access to the name and avatar of each user, we can use that instead of the random series of number in the uuid to identify them.

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...
const renderList = (props) => {
  const activeList = Presence.list(props.presences, (id, _metas) => id)

  const lobbyList = props.lobbyList.map(({ id, name, avatar }) => {
    const active = activeList.includes(id)
    return {
      name,
      avatar,
      id,
      active
    }
  })

  return lobbyList
    .sort(orderByActivity)
    .map(({ id, active, name, avatar }) => {
      const newStyle = {}
      if (active) newStyle.boxShadow = "inset 0px 0px 6px 4px rgba(58, 155, 207, 0.6)"
      if (props.currentRoom === id) newStyle.background = "#ddd"

      return (
        <div
          style={newStyle}
          className={style.user}
          key={id}
          onClick={() => { props.onRoomClick(id) }}>
          <div>
            <img alt="user avatar" src={avatar} />
          </div>
          <div>
            { name }
          </div>
        </div>
      )
    })
}
...
```

Let's also add some styling to make our users easier to distinguish.

```
/app/components/Sidebar/style.css
commit: coming soon
```

```
...

.user {
  display: flex;
  align-items: center;
  padding: 0.5rem;
  border-bottom: 1px solid rgba(0,0,0,0.1);
  border-bottom-width: 80%;
  min-height: 50px;
  cursor: pointer;
  background: white;
  transition: background 0.1s ease;
  box-shadow: 0px 4px 4px -2px rgba(0,0,0,0.2);
}
.user:hover {
  background: #ddd;
}
```

We now have a functional list of users, prioritized by activity level. The admin can also click on any of them and respond.

Empty room indicator

We should also give some indicator for when a no room is selected. Otherwise, people might just assume that the app is broken. We'll do this in a new `renderEmpty` function.

```
/app/components/ChatRoom/index.js
commit: coming soon
```

```
...

  renderEmpty() {
    if (this.props.currentRoom) return null
    return (
      <div className={style.empty}>
        No chat selected
      </div>
    )
  }

  ...
```

And some styling for our placeholder.

/app/components/ChatRoom/style.css
commit: coming soon

```
...

.empty {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 80%;
  color: #ccc;
  font-size: 2em;
}
```

At this point, we can now see active and inactive users and respond to them properly. We're also sorting the users by giving priority to those who are currently present.

Styling the Chat Component

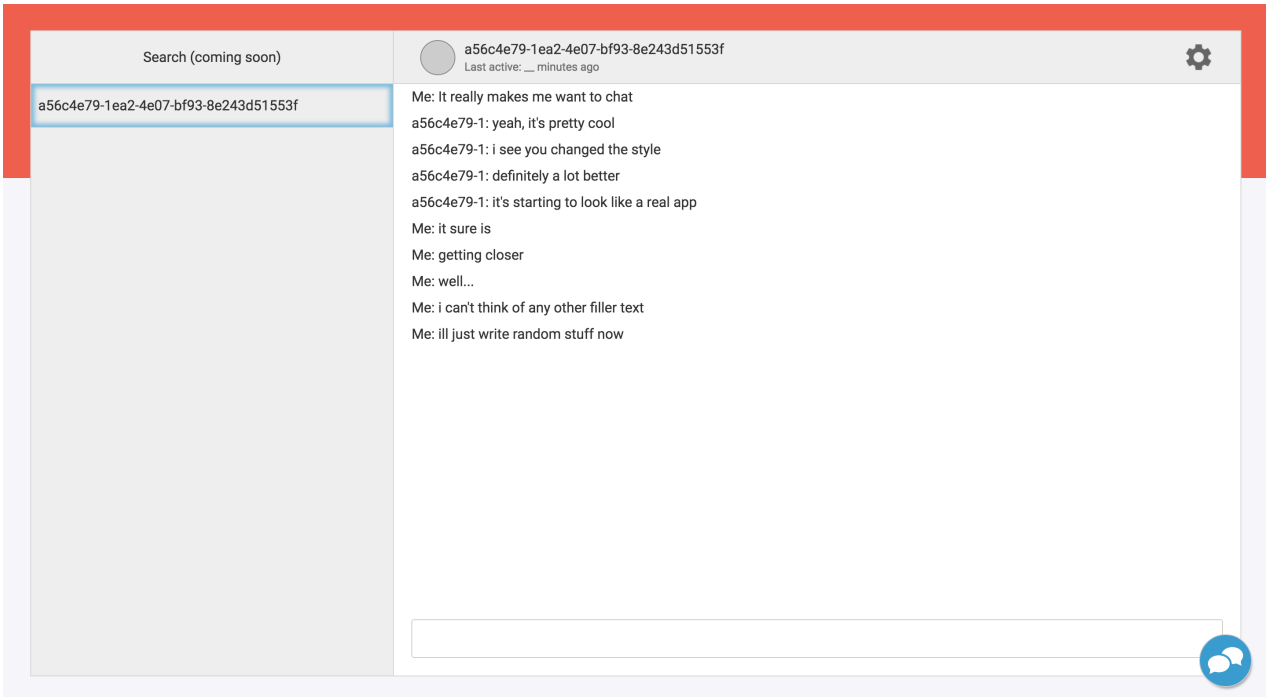
- Refactor with flexbox
- Make it look fancy

In this lesson, we just change our styling to use more of flexbox. This will set us up for our next lesson where we display the recent activity of our users.

Update styles

Before we get much further, we should update the style of our `Chat` component and our `Sidebar` so we can display things in a way that looks normal to a user. For starters, we're going to change around our `Chat` component so that our elements are no longer absolutely positioned to demonstrate different ways of aligning items and we're going to take this opportunity to show a few more features of flexbox.

After updating our styles, we're going to want a chatroom that looks something like the image below.



Although you generally don't want to mess with your `App` component, we're going to add some background styling to it since we want our entire app to live within this background to give us that fancy, subtle background color.

```
/app/components/App/index.js
```

commit: coming soon

```
import React from "react"
import cssModules from "react-css-modules"
import { connect } from "react-redux"
import style from "./style.css"
import Actions from "../../redux/actions"

export class App extends React.Component {
  componentDidMount() {
    this.props.dispatch(Actions.userAuth())
  }

  render() {
    return (
      <div className={style.background}>
        <div className={style.backgroundHeader} />
        <div className={style.backgroundFooter} />
        {this.props.children}
      </div>
    )
  }
}

export default connect()(cssModules(App, style))
```

So now we have three additional elements around our `Chat` component. The first one we'll just set to `fixed` and cover the entire app. Then we have a header, which we will color our theme red and a footer which we will color an off-white.

```
/app/components/App/style.css
commit: coming soon
```

```
.background {
  position: fixed;
  top: 0;
  left: 0;
  height: 100vh;
  width: 100vw;
}

.backgroundHeader {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  height: 200px;
  background-color: rgb(239, 95, 78);
}

.backgroundFooter {
  position: absolute;
  top: 200px;
  left: 0;
  right: 0;
  height: calc(100vh - 200px);
  background-color: rgb(245, 245, 250);
}
```

This will make our app look somewhat broken since our chat component does not have a background color.

Refactor Sidebar component

We're going to add a header to this component, which will eventually be used to filter/search through messages and we're going to change the positioning from `absolute` to flexbox. On the React-side, all we're doing is adding a header. On the CSS-side, changing the layout to a flex column.

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...

export const Sidebar = props => {
  return (
    <div className={style.sidebar}>
      <div className={style.header}>
        Search (coming soon)
      </div>
      { renderList(props) }
    </div>
  )
}
...
```

And now let's change the sidebar to a flex column with a width of 30%. We're also going to use a pattern from the awesome site Subtle Patterns to give our sidebar a little flare.

/app/components/Sidebar/style.css
commit: coming soon

```
...

.sidebar {
  display: flex;
  flex-flow: column nowrap;
  background: url("https://s3.amazonaws.com/learnphoenix-static-assets/images/swirl_pattern
  overflow-y: scroll;
  border-right: 1px solid rgb(213, 213, 213);
  width: 30%;
  position: relative;
}

.header {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 60px;
  background: rgb(238, 238, 239);
  border-bottom: 1px solid rgb(213, 213, 213);
}
```

And now our app looks even more broken! Next we need to wrap our `ChatRoom` component and change the positioning of the items to use flexbox as well.

Refactor ChatRoom component

Since the elements in our `ChatRoom` component are mostly absolutely positioned, this is going to be a significant change. The first thing we need to do is wrap our `input` so we can position it properly, then we're going to wrap our chat component and set up flex rows and columns.

This app is basically just one row with two columns (one 30% width, and one 70% width). The first column is our `Sidebar`, which we already handled, and the second contains the header, our messages, and the input box.

First, let's wrap our whole `Chat` component with some padding to show a little bit of the background, and then wrap our other components in a row.

/app/components/Chat/index.js
commit: coming soon

```
...

export class Chat extends React.Component {
  ...

  render() {
    return (
      <div className={style.container}>
        <div className={style.row}>
        <Sidebar
          presences={this.state.presences}
          lobbyList={this.state.lobbyList}
          onRoomClick={this.changeChatroom} />
        <ChatRoom
          input={this.state.input}
          handleChange={this.handleChange}
          handleMessageSubmit={this.handleMessageSubmit}
          currentRoom={this.state.currentRoom}
          messages={this.state.messages} />
        </div>
        { this.props.children }
      </div>
    )
  }
}

...
```

Now we're going to introduce our styles. These are pretty simple and bring to what is almost a functional

app. The first wraps our chat component and pads it with `2rem` so we show a little bit of the background behind our chat interface. It also centers it.

The `.row` is basically our entire app, so we're going to give it a white background and a border. The `.column` contains our header, our messages, and our input box.

/app/components/Chat/style.css
commit: coming soon

```
.container {
  display: flex;
  justify-content: center;
  align-items: center;
  position: relative;
  height: 100vh;
  width: 100vw;
  z-index: 100;
  padding: 2rem;
}

.row {
  display: flex;
  flex-flow: row nowrap;
  height: 100%;
  width: 100%;
  border: 1px solid rgb(213, 213, 213);
  background: white;
}
```

The next step is to style our `ChatRoom` component. We're also going to add a header which we can fill in with content in a later lesson.

/app/components/ChatRoom/index.js
commit: coming soon

```
...
export class ChatRoom extends React.Component {
  ...

  renderHeader() {
    return (
      <div className={style.header}>
        Header (coming soon)
      </div>
    )
  }

  ...

  render() {
    return (
      <div className={style.container}>
        { this.renderHeader() }
        ...

      </div>
    )
  }
}
...
```

Then we should style the components in our chatroom.

```
.container {
  position: relative;
  display: flex;
  flex-flow: column nowrap;
  position: relative;
  width: 70%;
}

.header {
  display: flex;
  flex-flow: row nowrap;
  width: 100%;
  background: rgb(238, 238, 239);
  border-bottom: 1px solid rgb(213, 213, 213);
  height: 60px;
  display: flex;
  flex-flow: row nowrap;
  justify-content: space-between;
  align-items: center;
  padding-left: 10px;
}

.inputWrapper {
  padding: 0 20px 20px 20px;
}

.input {
  width: 100%;
  line-height: 40px;
  font-size: 16px;
  outline: none;
  border: 1px solid #ccc;
  border-radius: 3px;
  padding-left: 10px;
  color: #333;
}
```

At this point, the app is starting to come together quite nicely! But if you start typing messages, you'll see that the messages flow over behind our input box and our scrolling is all messed up. So let's add some more styles to fix that.

This is also where we introduce `flex-grow`. You'll often see `flex: 1` used, which is shorthand for the following:

```
.style {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: auto;
}
```

Spend a couple minutes looking through this and this from CSS-Tricks for a great intro to how the `flex` property is used. Since our header and input are fixed height, we want to be able to tell our `messageWrapper` to fill all the remaining vertical space within our `row`. Thank to flexbox, all we have to do is set `flex-grow: 1`.

/app/components/ChatRoom/style.css
commit: coming soon

```
...

.chatWrapper {
  padding-top: 5px;
  flex-grow: 1;
  position: relative;
  overflow-y: scroll;
}
```

So now that our app is styled and works properly, let's add some more detail to our `Sidebar`.

Transactional Email with Mailgun and Bamboo

- Set up Mailgun
- Set up Bamboo
- Send welcome email

The Elixir ecosystem is rich with libraries offering us the ability to send emails such as Swoosh and Bamboo. For our application we'll be using Bamboo by Thoughtbot which has support for composable emails and many different email providers. Bamboo also has really good documentation that's worth looking over.

Sending emails is one of the ways we see Elixir sets itself apart from other languages like Ruby. With great support for concurrency we won't need to include other dependencies to enable our applications to send emails in the background.

Install Bamboo

The first thing we need to do is add our newest dependency, `:bamboo` to our `mix.exs` file:

/mix.exs
commit: coming soon

```
...
  defp deps do
    [
      {:bamboo, "~> 0.7"},
     ...

    ]
  end
...
```

Next we need to add `:bamboo` to our application function in `mix.exs`:

/mix.exs
commit: coming soon

```
def application do
  [mod: {PhoenixChat, []},
    applications: [
      :bamboo,
      ...

  ]]
end
```

Then be sure to get your dependencies.

```
$ mix deps.get
```

Configuration

One of the best features of Bamboo is its support for different providers. For our project we'll be using Mailgun but if you have a strong preference for another provider, the configuration should be similar.

Let's open up `config/dev.exs` and add a section for Bamboo at the bottom. This configuration will be used only in development mode and informs Bamboo as to which adapter or provider to use:

/config/dev.exs
commit: coming soon

```
config :phoenix_chat, PhoenixChat.Mailer,
  adapter: Bamboo.LocalAdapter
```

The `Bamboo.LocalAdapter` is great for development. Rather than sending emails over the network and through our providers, the emails are captured locally and available for previewing.

The other configurations need to be updated to include an appropriate Bamboo config. For testing we'll want to use the special `Bamboo.TestAdapter`, which handles email in memory and does not send emails, and in production we'll use the `Bamboo.MailgunAdapter`.

For a complete list of adapters see the Bamboo docs: Adapters.

Open `config/test.exs` and add the following:

/config/test.exs
commit: coming soon

```
config :phoenix_chat, PhoenixChat.Mailer,
  adapter: Bamboo.TestAdapter
```

For the production configuration we'll use Mailgun and pull the API key and Mailgun domain from our system environment variables, which we will define later:

```
/config/prod.exs
commit: coming soon
```

```
config :phoenix_chat, PhoenixChat.Mailer,
  adapter: Bamboo.MailgunAdapter,
  api_key: System.get_env("MAILGUN_API_KEY"),
  domain: System.get_env("MAILGUN_DOMAIN")
```

Defining Emails

Now that we've installed Bamboo and configured it, we need to create our Mailer and define our emails.

Create a new file, `lib/phoenix_chat/mailer.ex`, so we can define our Mailer module:

```
$ touch lib/phoenix_chat/mailer.ex
```

```
/lib/phoenix_chat/mailer.ex
commit: coming soon
```

```
defmodule PhoenixChat.Mailer do
  use Bamboo.Mailer, otp_app: :phoenix_chat
end
```

The `Mailer` is used for dispatching the emails to the adapter but construction of the Emails is done elsewhere. Let's create an `Email` module `lib/phoenix_chat/email.ex` to construct our emails.

Phoenix is not necessary to use Bamboo, but we can use the special adapter to make email formatting easier down the road. It also comes with this handy tool for previewing formatted emails. If you don't need formatted emails, you can simply use `import Bamboo.Email` rather than `Bamboo.Phoenix`.

```
$ touch lib/phoenix_chat/email.ex
```

```
/lib/phoenix_chat/email.ex
commit: coming soon
```

```
defmodule PhoenixChat.Email do
  use Bamboo.Phoenix, view: PhoenixChat.EmailView

end
```

Next we'll need to create our first email. In Bamboo, emails are defined as functions and are built up using a helpful domain-specific language (DSL). If you're not familiar with DSLs, you can think of them as tiny, special-purpose languages or libraries that are really good at doing one thing--in this case, email.

To get started, we'll define a basic welcome email:

```
/lib/phoenix_chat/email.ex
commit: coming soon
```

```
defmodule PhoenixChat.Email do
  use Bamboo.Phoenix, view: PhoenixChat.EmailView

  alias PhoenixChat.{User}

  def welcome_email(%User{email: email}) do
    new_email
    |> to(email)
    |> from("no-reply@phoenixchat.io")
    |> subject("Welcome")
    |> text_body("Welcome to PhoenixChat!")
  end
end
```

Bamboo's API is pretty easy. As you can see in the above code we specify the email to send to, the from address, subject, and then the text body.

That's it for creating an email, now we need to send it.

Sending Emails

With Bamboo there are two ways to send emails within the current request and in the background. To

send an email within the current request, we use the `deliver_now/1` function on our mailer. This function will dispatch the email request immediately and will need a response before the HTTP request can continue.

```
alias PhoenixChat.{Email, Mailer}

user |> Email.welcome_email |> Mailer.deliver_now
```

To send our emails in the background we can use the `deliver_later/1` function in Bamboo:

```
user |> Email.welcome_email |> Mailer.deliver_later
```

The advantage of sending in the background is we won't block the current request, this has the least impact on the user's experience and it's what we will use almost exclusively.

With our emails created and configured, we can proceed with integrating the welcome email into Phoenix. Let's start by opening `web/controller/user_controller.ex` and creating a new method, `send_welcome_email/`, to handle sending the email.

```
/web/controllers/user_controller.ex
commit: coming soon
```

```
alias PhoenixChat.{Email, Mailer, User}

  ...

  defp send_welcome_email(user) do
    user
    |> Email.welcome_email
    |> Mailer.deliver_later
  end
...
```

Once we've created our function we can incorporate it into `create/2`, which is the function that gets called when we create a new user.

```
/web/controllers/user_controller.ex
commit: coming soon
```

```
  ...

  def create(conn, %{"user" => user_params}) do
    changeset = User.registration_changeset(%User{}, user_params)

    case Repo.insert(changeset) do
      {:ok, user} ->
        {:ok, token, _claims} = Guardian.encode_and_sign(user, :token)

        send_welcome_email(user)

        ...
    end
  end
```

That's it! Now our users will receive a simple welcome email when they register for a new account. If you sign up, you should see something along the lines of the following in your server logs.

```
%Bamboo.Email{assigns: %{}, bcc: [], cc: [],
from: {nil, "no-reply@phoenixchat.io"}, headers: %{},
html_body: nil, private: %{}, subject: "Welcome",
text_body: "Welcome to PhoenixChat!",
to: [nil: "info@learnphoenix.io"]}
```

Testing with Elixir and Phoenix: Controllers

- UserController tests
- Metaprogramming
- AuthController tests

We've already gone over the basics of testing in a previous lesson with React, so in this lesson we'll jump right into testing your Phoenix app. We're going to start by testing our `UserController`.

To do this we're going to simulate requests by calling `get/3`, `put/3`, `delete/3`, `post/3`, etc. Those functions simulate a dispatch to an endpoint in your app without the need to go through HTTP using Phoenix.ConnTest.

UserController tests

The first thing you'll notice is the `use PhoenixChat.ConnCase`. This module comes by default in `test/support/conn_case.ex` and it's worth looking it over to get a sense of what it's providing to you on every test.

Now in order to save us some keystrokes, we're going to create two variables at the top of the file. One is for valid user attributes and one for invalid. Generally speaking, when we pass valid attributes, we expect things to pass, while invalid values should give us an error.

```
/test/controllers/user_controller_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.UserControllerTest do
  use PhoenixChat.ConnCase

  alias PhoenixChat.User
  @valid_attrs %{email: "me@test.com", password: "some content", username: "some content"}
  @invalid_attrs %{}

  ...

end
```

`PhoenixChat.ConnCase` provides us with a default `setup` block that passes a `conn` struct, just like we would have in our actual Controller. From there, we can add things to it, such as headers. As it happens,

this is already set up for us from our generator, but it should look like the code below.

```
  setup %{conn: conn} do
    {:ok, conn: put_req_header(conn, "accept", "application/json")}
  end
```

The next block is our first test. Note that `%{conn: conn}` is the second argument to `test/3` (docs), which is a pattern-match on the map passed by the `setup` block (the first argument is the string that describes the test and the third argument is `do` block). A more detailed explanation below the code.

```
test "lists all entries on index", %{conn: conn} do
  conn = get conn, user_path(conn, :index)
  assert json_response(conn, 200)["data"] == []
end
```

In the first line of our test, we simulate the `get` request (docs) for `users.index` and update our `conn` with that information.

Then we `assert` that the json response (docs) returned by `users.index` was successful (status code 200) and is an empty list, which is the result we should expect since the database has no user records.

Metaprogramming

In the next test, we create a user, then check to make sure that the user was added. So in order to make our lives easier, we're going to create a utility function to create a user. This will allow us to call `create_user` instead of a full `Repo.insert/1` command each time. And since we're going to use this function across multiple controllers, we're going to dig a little deeper into Phoenix to make this function reusable.

We're going to add `create_user/1` and `create_user/2` to our `test/support/conn_case.ex` file. If you open that file up, you'll see a block called `using`. This `using` block is what gets added every time you add the `use` keyword to a module. So in the example of our `UserControllerTest` above, by adding `use PhoenixChat.ConnCase`, we have effectively imported everything within this block. This is what is known as metaprogramming in Elixir. It's abstraction layer upon abstraction layer.

```
...
using do
  quote do
    # Import conveniences for testing with connections
    use Phoenix.ConnTest

    alias PhoenixChat.Repo
    import Ecto
    import Ecto.Changeset
    import Ecto.Query, only: [from: 1, from: 2]

    import PhoenixChat.Router.Helpers

    # The default endpoint for testing
    @endpoint PhoenixChat.Endpoint
  end
end
...
```

So now let's add the `create_user` functions and the necessary aliases to the file, and import `PhoenixChat.ConnCase` into our `using` block. You might be thinking that we've created an infinite loop by importing a module into itself, which imports itself into itself... but when you use `import`, it ignores the `using` block, so by importing `PhoenixChat.ConnCase` into this block, we've just added the two `create_user` functions into any module that uses `use PhoenixChat.ConnCase`.

```
/test/support/conn_case.ex
commit: coming soon
```

```
defmodule PhoenixChat.ConnCase do
  use ExUnit.CaseTemplate

  alias PhoenixChat.{Repo, User}

  using do
    quote do

      ...

      import PhoenixChat.ConnCase

      # The default endpoint for testing
      @endpoint PhoenixChat.Endpoint
    end
  end

  ...

  def create_user!() do
    Repo.insert! %User{username: "foo", email: "foo@bar.com"}
  end

  def create_user!(attrs) do
    map = Map.merge(%{username: "foo", email: "foo@bar.com"}, attrs)
    struct = struct(User, map)
    Repo.insert! struct
  end
end
```

In our test, we're going to simulate a post request with valid parameters to create a new user, then assert that we get the appropriate `201` response for account creation and that we can find the user in the database. For our query to work, we're going to drop (docs) the password field since we do not store plain text passwords in our database.

To clarify, we're running two assertions to test this functionality. First we're making sure that our post request worked, then we're making sure that the database was properly updated with the value.

```
/test/controllers/user_controller_test.exs
commit: coming soon
```

```
test "creates and renders resource when data is valid", %{conn: conn} do
  conn = post conn, user_path(conn, :create), user: @valid_attrs
  assert json_response(conn, 201)["data"]["id"]
  assert Repo.get_by(User, Map.drop(@valid_attrs, [:password]))
end
```

The second line runs a `post/2` (docs) request with the `conn` and `user_path`, which is an automatically generated helper for calling our `user` route (docs). We're asking that route to render `:create`, which you can see in your `UserView`.

If you want to see all your helpers, run the following and you will see all routes, with their helpers on the left.

```
$ mix phoenix.routes

page_path  GET   /           PhoenixChat.PageController :index
user_path  GET   /api/users  PhoenixChat.UserController :index
...
```

Then we need to test to make sure that invalid attributes fails the way we want. This should already be in place from our generator.

/test/controllers/user_controller_test.exs
commit: coming soon

```
test "does not create resource and renders errors when data is invalid", %{conn: conn} do
  conn = post conn, user_path(conn, :create), user: @invalid_attrs
  assert json_response(conn, 422)["errors"] != %{}
end
```

We're also asserting that we get the proper error when an `id` is not present. Asserting errors is simple. We use `assert_error_sent/2`, along with the error code we expect and a function that should generate the error. In the case below, since there is no `id` of `-1`, it should give us an error. Your generator should have created this for you as well.

Now we're going to test our `:update` function. Everything here is the same the previous test in which we tested `:create`.

/test/controllers/user_controller_test.exs
commit: coming soon

```
test "updates and renders chosen resource when data is valid", %{conn: conn} do
  user = create_user!
  conn = put conn, user_path(conn, :update, user), user: @valid_attrs
  assert json_response(conn, 200)["data"]["id"]
  assert Repo.get_by(User, Map.drop(@valid_attrs, [:password]))
end
```

Now we need to test the opposite and make sure it doesn't work with invalid parameters.

/test/controllers/user_controller_test.exs
commit: coming soon

```
test "does not update chosen resource and renders errors when data is invalid", %{conn: con
  user = create_user!
  conn = put conn, user_path(conn, :update, user), user: %{email: "foo"}
  assert json_response(conn, 422)["errors"] != %{}
end
```

And for our last test, we're going to make sure that we can delete a user bu calling `:delete` and making sure that we get a negative result from a `Repo.get` query by using `refute`.

/test/controllers/user_controller_test.exs
commit: coming soon

```
test "deletes chosen resource", %{conn: conn} do
  user = create_user!
  conn = delete conn, user_path(conn, :delete, user)
  assert response(conn, 204)
  refute Repo.get(User, user.id)
end
```

AuthController tests

We have a lot to test in our `AuthController`. We aren't going to test the internals of things like `Comeonin` or `Guardian` because it's safe to assume those will work.

```
$ touch test/controllers/auth_controller_test.exs
```

Just like we did with our previous tests, we need to set up our valid attributes and a `setup` block. This is almost identical to our other controller.

/test/controllers/auth_controller_test.exs
commit: coming soon

```
defmodule PhoenixChat.AuthControllerTest do
  use PhoenixChat.ConnCase

  alias PhoenixChat.User
  @valid_attrs %{email: "me@test.com", password: "password", username: "test"}
  @invalid_attrs %{}

  setup do
    {:ok, conn: put_req_header(build_conn, "accept", "application/json")}
  end

end
```

Now it's time to test our authentication routes. We're going to nest some of these tests within a `describe` block so we can group similar tests.

For our first tests, we want to make sure that a successful authentication returns a JSON web token, since that's the whole point of our authorization route.

In the first line, we create a user with valid attributes, then make a `post` request to `/auth/identity/callback` with the valid parameters to successfully create an account.

Then we assert that the response is `201`, which means it was created successfully.

Then we're asserting that the response contained the parameters we want (username and email), and at the same time we are assigning the web token to the value `token` so we can use it later.

In our next assertion, we take that token we received in our last assertion and use `Guardian.decode_and_verify/1` to decode the token, which should work, and at the same time we are storing the response in `claims`.

Then we take that `claims` value and find the `user.id` that it's associated with. Check `PhoenixChat.GuardianSerializer.for_token/1` to see what's stored in `sub`.

Then we just make sure that if we give bad values an account does not validate.

```
/test/controllers/auth_controller_test.exs
commit: coming soon
```

```
...
  describe "post '/auth/identity/callback'" do
    test "successful authentication returns JWT token", %{conn: conn} do
      user = User.registration_changeset(%User{}, @valid_attrs) |> Repo.insert!
      params = %{email: user.email, password: "password"}
      conn = post conn, "/auth/identity/callback", params

      response = json_response(conn, 201)["data"]
      assert response

      assert %{"username" => "test", "token" => token, "email" => "me@test.com"} = response
      assert {:ok, claims} = Guardian.decode_and_verify(token)
      assert claims["sub"] == "User:#{user.id}"
    end

    test "unsuccessful authentication", %{conn: conn} do
      params = %{email: "non@existent.com", password: "password"}
      conn = post conn, "/auth/identity/callback", params

      assert json_response(conn, 400) == "Internal server error"
    end
  end
...
```

Our next `describe` block will handle our `/auth/me` route. This route takes in a JWT via `authorization` header and checks to see if the token is valid.

First we need to create a user with valid parameters. Then we create a token by passing the connection through `Guardian`.

Now that we have a valid token, we can simulate passing it to our `/auth/me` route in the auth header. We then assert that the response we should receive is 200, which means it worked.

We're also going to assert that the response body returns the email, id, and the username of the user because our frontend expects those values. If you forgot what the `^` does, it references a value above rather than use the value for assignment.

In the second test, we simply pass an invalid token and assert that the response is 401 unauthorized.

```
/test/controllers/auth_controller_test.exs
commit: coming soon
```

```
...
  describe "get '/auth/me'" do
    test "authorized user gets json response", %{conn: conn} do
      user = User.registration_changeset(%User{}, @valid_attrs) |> Repo.insert!
      token = conn
              |> Guardian.Plug.api_sign_in(user)
              |> Guardian.Plug.current_token

      conn = conn
             |> put_req_header("authorization", "Bearer #{token}")
             |> get("/auth/me")

      response = json_response(conn, 200)["data"]
      assert response

      %{email: email, id: id, username: username} = user
      assert %{"email" => ^email, "id" => ^id, "username" => ^username} = response
    end

    test "unauthorized user", %{conn: conn} do
      conn = conn
             |> put_req_header("authorization", "Bearer fake_token")
             |> get("/auth/me")

      assert json_response(conn, 401) == "Internal server error"
    end
  end
...
```

And for the sake of optimization, we're going to give our test environment really bad keys so they generate faster. This will overwrite our normal configuration when in the test environment.

/config/test.exs
commit: coming soon

```
config :comeonin, :bcrypt_log_rounds, 4
config :comeonin, :pbkdf2_rounds, 1
```

Debugging

If this is not your first time through the course, you might run into an error when you run `mix test`.

```
** (Postgrex.Error) ERROR (duplicate_table): relation "users" already exists
...
```

If you run into this error, run `MIX_ENV=test mix ecto.reset` to reset your test database. Ecto creates a separate database for your test environment, so just like on your actual app, when you start a new project and your migrations change, you need to drop and create a fresh database.

Testing with Elixir and Phoenix: Channels

- RoomChannel tests
- Channel helpers tests
- AdminChannel tests
- UserSocket tests

Channel tests, like Controller tests, simulate the broadcasting and pushing of events to the client without making an actual socket connection. By simulating the connection, your tests will run much faster and with less configuration so you can focus on testing the logic in your Channel handlers.

We're going to start with our `RoomChannel`.

RoomChannel tests

Just like with our controller tests, we get a lot out of the box with Phoenix. For Channels it's called `ChannelCase` rather than `ConnCase`. It's worth taking a quick look at `test/support/channel_case.ex` to get a sense of what you get for free with `ChannelCase`.

The `setup` block allows us to setup each test case with a socket that's subscribed to something that we want to test, which in this case is `room:lobby`. We're also going assign a `uuid` to a number and `user_id` to nil to test for anonymous users who are not admins.

To connect to the channel, we use `subscribe_and_join/2` (docs).

While you're at it, go ahead and delete the existing tests.

```
/test/controllers/room_channel_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.RoomChannelTest do
  use PhoenixChat.ChannelCase

  alias PhoenixChat.{RoomChannel, Message}

  setup do
    {:ok, %{messages: []}, socket} =
      socket("user_id", %{uuid: "1144", user_id: nil})
      |> subscribe_and_join(RoomChannel, "room:lobby")
    {:ok, socket: socket}
  end

  ...
end
```

Now that we have our channel set up for each of our tests, we can start interacting with it. In our first test, we're going to join a room, add messages to the database, then check to make sure the messages are returned as the payload.

The first few lines add three messages to the database: two in `room:1` and one in `room:2`. The next line calls the socket with `user_id` and `%{some: :assign}` which is there to communicate that it expects a map for `socket.assigns`, then subscribes to the `room:1` channel. We're using pattern matching in the `{:ok, %{messages: messages}, _}` expression to match the result of our connection (which will be a list of the latest 10 messages) to the term `messages`.

We then assert that we received two messages, since two of the messages we added were added to the room we connected to: `room:1`.

```
/test/controllers/room_channel_test.exs
commit: coming soon
```

```
test "joining a room returns messages from the DB as payload" do
  timestamp = Ecto.DateTime.utc()
  Repo.insert!(%Message{body: "Foo", timestamp: timestamp, room: "1"})
  Repo.insert!(%Message{body: "Bar", timestamp: timestamp, room: "1"})
  Repo.insert!(%Message{body: "Bar", timestamp: timestamp, room: "2"})

  {:ok, %{messages: messages}, _} =
    socket("user_id", %{some: :assign})
    |> subscribe_and_join(RoomChannel, "room:1")

  assert length(messages) == 2
end
```

The last thing we need to test for our `RoomChannel` is that we can push messages using our socket. To

do this we're creating a valid `payload`, then using the `push` function (docs) to send a `message` event to the socket subscribed to `room:lobby` with the payload we just defined. Recall that whenever a `message` event is sent to our room channel, it triggers the `def handle_in("message", payload, socket) do` function defined in `web/channels/room_channel.ex`.

Then we use another assertion type called `assert_reply` (docs), which, as you might have guessed, asserts that we received a reply that contains a reference to the event that was pushed, an `:ok` atom, and the payload.

Then for good measure we're going to check our database to make sure that the message we sent was in fact added to the database.

```
/test/controllers/room_channel_test.exs
commit: coming soon
```

```
test "message replies with status ok and saves message to DB", %{socket: socket} do
  payload = %{
    body: "hello",
    timestamp: 1470637865914,
    room: "lobby",
    from: "1144"
  }

  ref = push socket, "message", payload
  assert_reply ref, :ok, payload
  assert Repo.get_by(Message, payload)
end
```

ChannelHelpers tests

We created a few helpers to make authorization easier, so we need to test them too.

```
$ touch test/channels/channel_helpers_test.exs
```

Since this is normal Elixir, we are going to import `ExUnit.Case`, which is what you would import for any other Elixir module you want to test. The functions in our `ChannelHelpers` are pure functions so they're really easy to test. This is because pure functions have no side effects and always return the same output given the same inputs.

```
/test/channels/channel_helpers_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.ChannelHelpersTest do
  use ExUnit.Case

  import PhoenixChat.ChannelHelpers

end
```

To test our authorization function, need to test both a passing and a failing test. Recall that `authorize/2` takes in a payload and a function. The payload is used to determine if the user is authorized and the function is what we run if the user is authorized. In this case, the payload is "test" since we currently set `authorized?/1` to return true no matter what it receives, and we are passing a function that returns "foo". If we get that value, then the `authorize/2` method worked properly.

Next we test a failed authorization. We do this by passing in a `custom_authorize` function that returns false. Then we check to make sure that we received the error we expected: "unauthorized".

The last test is just a placeholder at this point since we will always return true no matter what value we send to `authorized?/1`.

```
/test/channels/channel_helpers_test.ex
commit: coming soon
```

```
test "authorize/2 and authorize/3" do
  assert authorize("test", fn -> "foo" end) == "foo"

  failed_authorization = authorize("test", fn -> "foo" end, fn _ -> false end)
  assert {:error, %{reason: "unauthorized"}} == failed_authorization
end

test "authorized?/1" do
  assert authorized?(false) == true
end
```

AdminChannel tests

Now we're going to write tests for our `AdminChannel`, which is going to be very similar to our `RoomChannel` tests.

```
$ touch test/channels/admin_channel_test.exs
```

First we add `ChannelCase` since we're testing a channel, then we add our `setup` block. We're going to

use `on_exit`, which runs after every test, to call `delete_all_objects/1` in our list from `:ets` so we start every test with a fresh ETS table.

/test/channels/admin_channel_test.exs
commit: coming soon

```
defmodule PhoenixChat.AdminChannelTest do
  use PhoenixChat.ChannelCase

  alias PhoenixChat.{AdminChannel, LobbyList}

  setup do
    on_exit fn ->
      :ets.delete_all_objects(LobbyList)
    end
  end

end
```

Our first test makes sure we can join the `admin:active_users` channel as an admin. This is the channel from which we gain access to the list of all currently active users and populate the sidebar on our frontend.

First we add some fake data to our `LobbyList`, then we simulate joining the socket with a `user_id` of `1`, since in our application we assume that any user with a `user_id` is an admin (anonymous users have a `uuid`). This is mostly the same as our other channel test.

In our first assertion, we're testing that the `lobby_list` that we get back from our socket contains two items by checking its `length`.

Then we use `assert_push` to assert that we can push a new event to `lobby_list`. In this case, we're adding a new anonymous user.

Next we `assert_push` our `presence_state`, which starts out with an empty payload since no presences have been tracked at this point (we're adding it now).

Finally, we `assert_push` that `presence_diff` is pushed to the client with a payload that signifies that a presence has been tracked.

```
test "joining admin:active_users as admin" do
  LobbyList.insert("foo")
  LobbyList.insert("bar")

  {:ok, %{lobby_list: lobby_list}, _socket} =
    socket("user_id", %{user_id: 1})
    |> subscribe_and_join(AdminChannel, "admin:active_users")

  assert length(lobby_list) == 2
  assert_push "lobby_list", %{uuid: 1}
  assert_push "presence_state", %{}
  assert_push "presence_diff", %{joins: %{"1" => %{}}}
end
```

The last thing we want to test is to make sure that non-admins aren't receiving the list of all active users.

We start by connecting to the socket as an anonymous user by setting `user_id` to `nil` and `uuid` to something valid.

Then we `refute_push` to refute that the `lobby_list` event is triggered with a payload. This is our way of testing `handle_out/3` for `lobby_list`, which filters non-admins from receiving the event.

```
/test/channels/admin_channel_test.exs
commit: coming soon
```

```
test "non-admin users do not receive the 'lobby_list' event on join" do
  {:ok, %{lobby_list: _}, _} =
    socket("user_id", %{user_id: nil, uuid: 5})
    |> subscribe_and_join(AdminChannel, "admin:active_users")

  refute_push "lobby_list", %{}
end
```

UserSocket tests

The last thing we need to test for our channels is our `UserSocket`. All we need to do to test this is insert a user into our database and make sure that the user is being assigned to our socket.

```
$ touch test/channels/user_socket.exs
```

The first line adds an admin, then we connect to our UserSocket and pass the newly created admin's `id`

along to it. Then we `subscribe_and_join/3` the socket and then assert that we find each of the parameters we expect.

In the second test, we do basically the same thing but as an anonymous user. If the user is anonymous, she won't have a `user_id`, so we `refute` that there is one, and then assert that the `uuid` is the same as the one we passed in.

/test/channels/user_socket.exs
commit: coming soon

```
defmodule PhoenixChat.UserSocketTest do
  use PhoenixChat.ChannelCase

  alias PhoenixChat.{Repo, User, UserSocket}

  test "connecting to user socket as logged-in user" do
    admin = Repo.insert!(%User{email: "admin@bar.com", username: "admin"})

    {:ok, socket} = connect(UserSocket, %{"id" => admin.id})
    {:ok, _, socket} = subscribe_and_join(socket, "room:1", %{})

    assert socket.assigns.user_id == admin.id
    assert socket.assigns.email == admin.email
    assert socket.assigns.username == admin.username
  end

  test "connecting to user socket as anonymous user" do
    {:ok, socket} = connect(UserSocket, %{"uuid" => 25})
    {:ok, _, socket} = subscribe_and_join(socket, "room:25", %{})

    refute socket.assigns.user_id
    assert socket.assigns.uuid == 25
  end
end
```

Testing with Elixir and Phoenix: Models

- Write tests for User model
- Write tests for Message model

The tests we're going to write for our models are a little bit more specific than the ones we wrote for our controllers and channels. This is because we may call Model behaviors in our Controllers or Channels but not the other way around.

So we need to test our input data validations and our custom queries.

Channel/Controller tests often indirectly test the behavior of our model, like when we get initial messages on `channel.join` and expect `latest_room_messages/2` to have been called with the most recent 10 messages. But we also want to make sure that our models work independently of our controllers and channels. Fortunately these tests are simpler than the Controller/Channel tests since we can call the functions directly (e.g. `User.changeset/2`) rather than simulating an HTTP request like we had to do with our Controller.

User model

We start by importing `ModelCase`, which contains useful functions for `Model` testing. Check them out at `test/support/model_case.ex`. Then, just as we did in our other tests, we create a set of valid and invalid attributes to make our tests easier to read.

```
/test/models/user_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.UserTest do
  use PhoenixChat.ModelCase

  alias PhoenixChat.User

  @valid_attrs %{email: "foo@bar.com", encrypted_password: "some content", username: "some
  @invalid_attrs %{}

  ...
end
```

Since we don't have any custom queries for our `User` model yet, all we're going to test is our validations

on input data. The first and simplest is testing our `changeset` with valid inputs. To do this we use `changeset.valid?`, which is a field in the `Ecto.Changeset` struct (docs). If you log a valid changeset, you'll get something like the result below (note the `valid?: true` field).

```
#Ecto.Changeset<action: nil, changes: %{email: "foo@bar.com", username: "some content"}, er
```

This should be the same as the output from the generator.

```
/test/models/user_test.exs
commit: coming soon
```

```
test "changeset with valid attributes" do
  changeset = User.changeset(%User{}, @valid_attrs)
  assert changeset.valid?
end
```

Then we're going to test all the ways in which we can make an invalid `changeset`. Cure

If you look at our `changeset` in `web/models/user.ex`, you'll see that we have a few constraints: namely `cast`, `validate_format`, `validate_length`, and `unique_constraint`.

```
def changeset(model, params \\ :empty) do
  model
  |> cast(params, ~w(email username), [])
  |> validate_format(:email, ~r/@/)
  |> validate_length(:username, min: 1, max: 20)
  |> update_change(:email, &String.downcase/1)
  |> unique_constraint(:email)
  |> update_change(:username, &String.downcase/1)
  |> unique_constraint(:username)
end
```

In each of the following tests, we're going to create a changeset with invalid attributes of a particular type and check the error message to make sure it's the error we expect.

`changeset` stores errors in its `:errors` field, which is populated by a list of two-element tuples in the format `{field_name, error_msg}`. So in our assertion, we're pulling out the value we want using `in` to pull the value out of the `errors` list. We encourage you to put in the occasional `IO.puts` to make sure you understand which values are coming from where.

```
/test/models/user_test.exs
commit: coming soon
```

```
test "changeset with blank email and username" do
  %{errors: errors} = User.changeset(%User{}, %{})
  assert {:email, {"can't be blank", []}} in errors
  assert {:username, {"can't be blank", []}} in errors
end

test "changeset with invalid email format" do
  %{errors: errors} = User.changeset(%User{}, %{email: "foo"})
  assert {:email, {"has invalid format", []}} in errors
end

test "changeset with username invalid length" do
  long_username = String.duplicate "f", 21
  %{errors: errors} = User.changeset(%User{}, %{username: long_username})
  assert {:username, {"should be at most %{count} character(s)", [count: 20]}} in errors
end

test "changeset must have a unique email" do
  changeset = User.changeset(%User{}, @valid_attrs)
  Repo.insert!(changeset)

  changeset = User.changeset(%User{}, @valid_attrs)
  {:error, changeset} = Repo.insert(changeset)
  assert {:email, {"has already been taken", []}} in changeset.errors
end

test "changeset must have a unique username" do
  changeset = User.changeset(%User{}, @valid_attrs)
  Repo.insert!(changeset)

  attrs = Map.put(@valid_attrs, :email, "test@bar.com")
  changeset = User.changeset(%User{}, attrs)
  {:error, changeset} = Repo.insert(changeset)
  assert {:username, {"has already been taken", []}} in changeset.errors
end
```

Then we need to test our `registration_changeset` in the same way. First we pass valid attributes and make sure it works. We're also testing to make sure that an `encrypted_password` field was created by using the `get_change/3` method (docs). We don't need to test `put_encrypted_pw/1` directly because it is safe to assume that `Comeonin` and `Bcrypt` work properly.

Once we've tested a passing changeset, we send a password that is too short and ensure that our error corresponds to the "password is too short" error.

```
/test/models/user_test.exs
commit: coming soon
```

```
test "registration changeset with valid attributes" do
  valid_attrs = Map.put(@valid_attrs, :password, "password")
  changeset = User.registration_changeset(%User{}, valid_attrs)
  assert changeset.valid?
  assert get_change(changeset, :encrypted_password)
end

test "registration changeset with invalid password length" do
  long_password = String.duplicate "p", 101
  %{errors: errors} = User.registration_changeset(%User{}, %{password: long_password})
  assert {:password, {"should be at most %{count} character(s)", [count: 100]}} in errors
end
```

Message model

Our message model is a little bit more complicated since we need to test our custom query that fetches the latest 10 posts.

First we set up the valid and invalid attributes.

```
/test/models/message_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.MessageTest do
  use PhoenixChat.ModelCase

  alias PhoenixChat.Message

  @valid_attrs %{body: "some content", room: "some content", timestamp: "2010-04-17 14:00:0
  @invalid_attrs %{}

  ...
end
```

Then we test our changeset with those attributes, just as we did with the `User` model.

```
/test/models/message_test.exs
commit: coming soon
```

```
test "changeset with valid attributes" do
  changeset = Message.changeset(%Message{}, @valid_attrs)
  assert changeset.valid?
end

test "changeset with invalid attributes" do
  %{errors: errors} = Message.changeset(%Message{}, @invalid_attrs)
  assert {:body, {"can't be blank", []}} in errors
  assert {:timestamp, {"can't be blank", []}} in errors
  assert {:room, {"can't be blank", []}} in errors
end
```

And now we test our query. We can test queries by running them and then checking the records returned by the query and matching them with the results that we expect.

In some circumstances, we may choose to run an assertion on the query struct returned by `Ecto`. We're going to do this in the last assertion of this test because all we're testing is that the default `limit` is set to 10. You could also test this by creating 11 messages, running the query, then assert that we only got 10 back.

We prepare the test by inserting multiple messages to the database with different timestamps, since we need to know which are the most recent to ensure they are properly ordered.

Then we create our query by calling `latest_room_messages/2` with `1` as the room (note that we added three messages to room 1) and `2` as the override for the default value of `10`, so we will only get the two most recent messages back. Now that we've created the query we want, we use `Repo.all/1` to get the result of the query.

Now we run our first assertion, which makes sure that we only received 2 messages. If we hadn't specified `2` in our `latest_room_messages/2`, we should have received all 3 messages we added to the database because the default limit is 10 messages.

From there we use the head-tail pattern matching to separate out the first message from the list (the head) from the rest of the message (the tail) and assign the head to `msg1` and the tail to `msg2`.

Then in our assertion, we are checking to make sure that the second message (`msg2`) has the same timestamp value as the second message we added to our database (`second`). We do this with Ecto.DateTime.compare/2. Keep in mind that comparing Ecto.DateTime structs with things like `>` or other comparison operators won't work as expected because they're treated as structs rather than normal values.

Our next assertion checks to make sure that the values we received are ordered properly. We expect that our most recent (by timestamp) message should be first, so we use `compare/2` again to assert that `msg1.timestamp` is greater than (`:gt`) `msg2.timestamp`.

The last assertion, as mentioned earlier, tests to make sure that the default value for our `limit` is 10 if

we do not specify anything.

/test/models/message_test.exs
commit: coming soon

```
test "query that returns latest messages of a given room" do
  first  = Ecto.DateTime.from_erl({{2016, 5, 23}, {12, 30, 12}})
  second = Ecto.DateTime.from_erl({{2016, 5, 24}, {12, 30, 12}})
  third  = Ecto.DateTime.from_erl({{2016, 5, 25}, {12, 30, 12}})
  Repo.insert!(%Message{room: "1", body: "test", timestamp: first})
  Repo.insert!(%Message{room: "1", body: "test", timestamp: second})
  Repo.insert!(%Message{room: "1", body: "test", timestamp: third})
  Repo.insert!(%Message{room: "2", body: "test", timestamp: Ecto.DateTime.utc()})

  messages = Message.latest_room_messages("1", 2) |> Repo.all
  assert length(messages) == 2

  [msg1 | [msg2]] = messages
  assert Ecto.DateTime.compare(msg2.timestamp, second) == :eq
  assert Ecto.DateTime.compare(msg1.timestamp, msg2.timestamp) == :gt
  assert {10, :integer} in Message.latest_room_messages("1").limit.params
end
```

If you run your tests and you're getting an error along the lines of the code below, you are most likely running the wrong version of Ecto. You need to upgrade to Ecto 2.0.

```
3) test changeset with invalid attributes (PhoenixChat.MessageTest)
   test/models/message_test.exs:14
   Assertion with in failed
   code: {:body, "can't be blank"} in errors
   lhs:  {:body, "can't be blank"}
   rhs:  [body: {"can't be blank", []}, timestamp: {"can't be blank", []},
          room: {"can't be blank", []}]
   stacktrace:
     test/models/message_test.exs:16: (test)
```

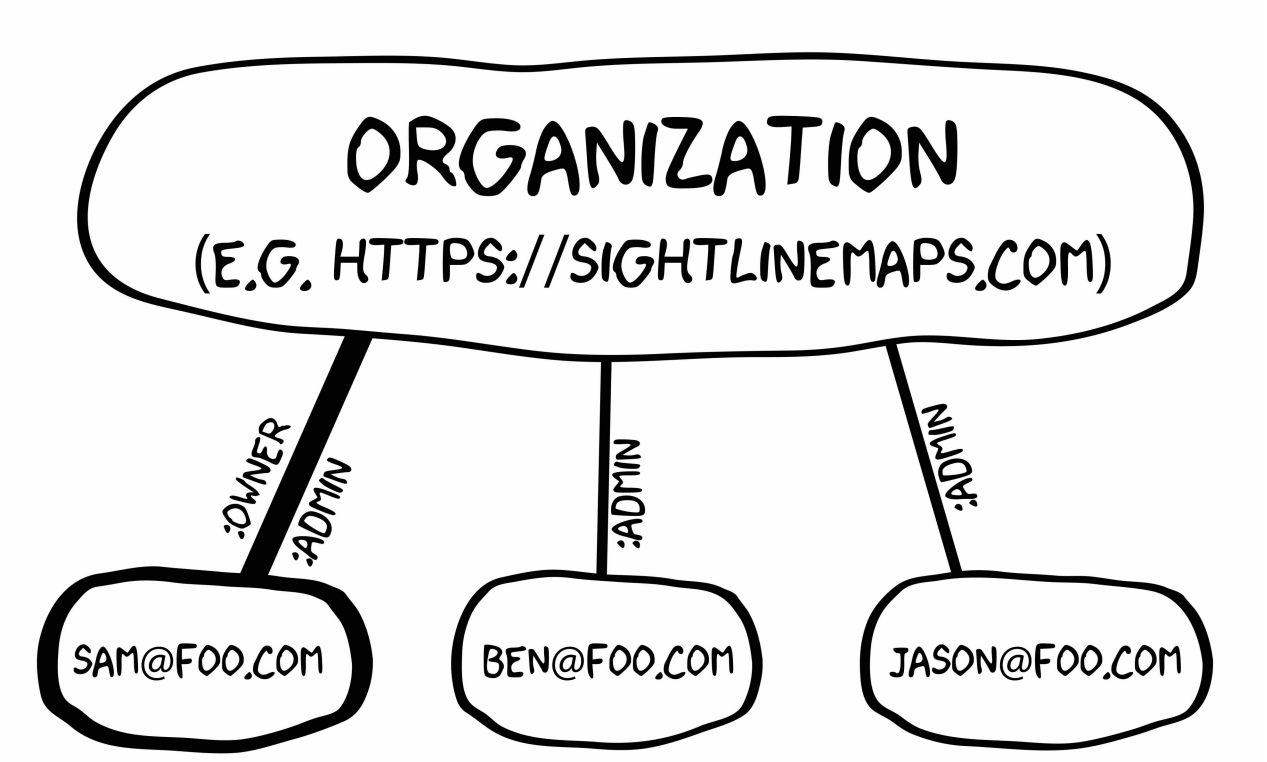
Creating an Organization Model

- Creating the model
- Writing tests

At this point in our app, all users who sign up will have access to the same chatroom. This is fine if you're self-hosting, but we want to build this to handle multiple accounts, which we will call `Organizations`. Then at some point, we will add the ability to include multiple administrators within each organization.

For example, if you have two websites, say sightlinemaps.com and learnphoenix.io, each website would represent an organization. We want to send chat messages from visitors to sightlinemaps.com only to the admins of the Sightline Maps organization, and not to LearnPhoenix.

The structure of our organizations will look like the image below, with the following characteristics: 1) Every user (admin/owner) may only be associated with one organization, 2) Each organization may only have one owner, 3) Every organization may have multiple administrators who can see and respond to messages.



Creating an Organization model

We're going to start by adding an `Organization` model with only three fields:

`public key` - randomly generated unique key that will be used for routing messages from users to admins. `website` - a unique website. `owner` - an association with a `User`. This is the creator/owner of the organization.

The public key will be automatically generated when an organization is created. This public key will be used for routing messages from users to admins. All users of website A will use the same public key as all admins of website A. Every user and admin socket connection will store the public key of the relevant organization.

We also take steps to ensure our data is valid since we want `website` and `public_key` to be unique. Also, we want the website to have a valid URL format. So to validate the URL format, we use Regular Expressions, which we will cover only briefly. If you are not familiar with Regular Expressions, there are ample resources online to learn how they work and what they are used for. One of them is regexone.com.

So now let's create an `Organization` model with a `website`, a `public_key`, and an `owner_id`. We're also going to generate the controller and a lot of other boilerplate that comes with it by using `mix phoenix.gen.json`.

```
$ mix phoenix.gen.json Organization organizations \
  public_key website owner_id:references:users
```

Then we need to change a few things. The first notable change is that we're adding a `:nilify_all` option to the `:on_delete` for our user (docs). The reason we're doing this is because every organization must have an owner, and if a user is deleted, we have to decide what to do with the organization. For now, we're just going to set the value of the owner to `nil` and we can later decide what we want to do with un-owned organizations. If we didn't do this, we would only know if an organization is un-owned by filtering out all organization that do not have a user that is currently registered.

Then we change around our index since we want our organizations to be unique based on their `public_key` and their `website`. We're also adding an index on `owner_id` because we might have to search based on that parameter at some point.

```
/priv/repo/migrations/...create_organization.exs
commit: coming soon
```

```
defmodule PhoenixChat.Repo.Migrations.CreateOrganization do
  use Ecto.Migration

  def change do
    create table(:organizations) do
      add :public_key, :string, null: false
      add :website, :string, null: false
      add :owner_id, references(:users, on_delete: :nilify_all)

      timestamps()
    end

    create index(:organizations, [:owner_id])
    create unique_index(:organizations, [:public_key])
    create unique_index(:organizations, [:website])
  end
end
```

And now we have to build out our `Organization` model. For the sake of clarity, we're going to go through each function individually. The first is our `changeset`, in which we define what is a valid `Organization` model with a variety of constraints. These constraints and functions are defined below the `changeset` and are explained in greater detail below. The format of the changeset should look familiar.

```
/web/models/organization.ex
commit: coming soon
```

```
defmodule PhoenixChat.Organization do
  use PhoenixChat.Web, :model

  schema "organizations" do
    field :public_key, :string
    field :website, :string
    belongs_to :owner, PhoenixChat.Owner

    timestamps()
  end

  @doc """
  Builds a changeset based on the `struct` and `params`.
  """
  def changeset(struct, params \\ %{}) do
    struct
    |> cast(params, [:website])
    |> validate_required([:website])
    |> update_change(:website, &set_uri_scheme/1)
    |> validate_change(:website, &validate_website/2)
    |> unique_constraint(:website)
    |> put_public_key()
    |> unique_constraint(:public_key)
  end

  ...
end
```

Within our `changeset` pipeline, we have a `put_public_key/1` function. This is the function that generates the `:public_key` for the organization. We're going to allow for the future possibility that someone might need to change their information without changing the `public_key`, so we're going to use this function only if the organization is new, which we determine based on whether the incoming data has an `id` field (if it's been added to the database already, it will have an `id`, otherwise it won't).

Assuming the organization is new and the `changeset` is valid, we pass it along to `put_change/3` (docs), which changes a key (in this case `:public_key`) with a value. On our case, that value is the output of another function we need to create called `random_key/0`. If the organization already exists, then we pass it along to the rest of the changeset without changing the `public_key` field.

Within the `random_key` function, we're generating a random key using the `crypto` module in Erlang, which gives us random bytes which encode to Base64 text, then we trim that to a certain number of characters. We're setting a default length of 10 characters. If you're interested in learning more about has collision probabilities to determine how long you should make your API key, check our blog post on the subject.

```
/web/models/organization.ex
commit: coming soon
```

```
defmodule PhoenixChat.Organization do
  use PhoenixChat.Web, :model

  ...

  defp put_public_key(%{data: data} = changeset) do
    if changeset.valid? && !data.id do
      changeset
      |> put_change(:public_key, random_key())
    else
      changeset
    end
  end

  defp random_key(length \\ 10) do
    :crypto.strong_rand_bytes(length) |> Base.encode64 |> binary_part(0, length)
  end

  ...

end
```

The next couple functions deal with the website input. If the user provides an input for a website without a URI prefix, such as `http://`, `https://`, or `ftp://`, we are going to prepend it with a default URI scheme of `https://`, so that a website of `foo.com` becomes `https://foo.com`.

Within this function, we're using Elixir's Regex module to match on any string that starts with (`^`) any set of numbers or letters (`w+`) followed by `://`. If it matches, that means the website provided already has a URI scheme. If it doesn't match, then we're prepending `https://` to it. And regardless of whether it matches or not, we're turning the whole string into lowercase before returning. If you want to test out if your particular regex is valid with Elixir, check out Elixre (you might even want to bookmark it).

```
/web/models/organization.ex
commit: coming soon
```

```
defmodule PhoenixChat.Organization do
  use PhoenixChat.Web, :model

  ...

  defp set_uri_scheme(nil), do: nil
  defp set_uri_scheme(website) do
    if Regex.match?(~r/^\w+:\/\//, website) do
      website
    else
      "https://" <> website
    end |> String.downcase()
  end
end
```

In the `validate_website` function, we're validating that the website address we received is valid. This a complicated thing to match and we aren't going to even attempt to handle the edge cases, but this validation should handle the overwhelming majority of websites (if you want to get a sense of how many random things are actually valid URLs, check out formvalidation.io). This function starts by using the Elixir URI module to split the website into `scheme` (e.g. `https://`) and `host` (e.g. `learnphoenix.io`).

We want to return an error if the URI scheme is not `http` or `https`, or if it does not pass our `valid_host_format/1` function, which has its own regular expression. The first grouping of the regular expression looks for some series of numbers and letters that end in a `.` (`^([a-zA-z]+\.)`). The asterisk (`*`) tells us that

Within the `valid_host_format/1` function, we're running a new regular expression that checks to make sure that the url starts with some series of numbers and/or letters, followed by a `.`, followed by anything (`*`), and must end with a series of letters.

```
/web/models/organization.ex
commit: coming soon
```

```
defmodule PhoenixChat.Organization do
  use PhoenixChat.Web, :model

  ...

  defp validate_website(:website, website) do
    %URI{scheme: scheme, host: host} = URI.parse(website)

    if is_nil(scheme) ||
       is_nil(host) ||
       not scheme in ~w(http https) ||
       !valid_host_format?(host) do
      [website: "invalid url format"]
    else
      []
    end
  end

  defp valid_host_format?(host) do
    Regex.match? ~r/^([a-zA-z]+\.)*[a-zA-Z]+$/, host
  end
end
```

Organization model tests

To ensure the stability of our app, we're going to write some tests for our Organization model. Much of the code below will have already been written by the generator and all of the content of these tests has already been covered in the previous chapters on testing. (If there is anything in this codeblock that you do not believe was covered previously, please email info@learnphoenix.io and we will add clarification)

```
/test/models/organization_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.OrganizationTest do
  use PhoenixChat.ModelCase

  alias PhoenixChat.Organization

  @valid_attrs %{website: "foo.com"}
  @invalid_attrs %{}

  test "changeset with valid attributes" do
    changeset = Organization.changeset(%Organization{}, @valid_attrs)

    assert changeset.valid?
  end
```

```
  end

  test "changeset must have unique website" do
    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    Repo.insert! changeset

    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    {:error, changeset} = Repo.insert(changeset)

    assert {:website, {"has already been taken", []}} in changeset.errors
  end

  test "changeset must have a unique public key generated on create" do
    changeset = Organization.changeset(%Organization{}, %{website: "http://foo.com"})
    org1 = Repo.insert! changeset

    changeset = Organization.changeset(%Organization{}, %{website: "http://bar.com"})
    org2 = Repo.insert! changeset

    assert org1.public_key != org2.public_key
  end

  test "changeset's website must be a valid url" do
    some_invalid_urls = ["test this", "???", "...", ".www.foo.bar.", "foo.", "ftp://foo.com

    for invalid_url <- some_invalid_urls do
      changeset = Organization.changeset(%Organization{}, %{website: invalid_url})

      assert {:website, {"invalid url format", []}} in changeset.errors
    end

    some_valid_urls = ~w(foo.com www.foo.com http://foo.com https://foo.com?test=foo foo.co

    for valid_url <- some_valid_urls do
      changeset = Organization.changeset(%Organization{}, %{website: valid_url})

      assert changeset.valid?
    end
  end
end
```

Also, just for the sake of making sure our app compiles at this point, let's add the endpoint that the generator suggested to `router.ex`.

```
/web/router.ex
commit: coming soon
```

```
...
  scope "/api", PhoenixChat do
    pipe_through :api

    resources "/users", UserController, except: [:new, :edit]
    resources "/organizations", OrganizationController, except: [:new, :edit]
  end
...
```

Next we need to associate our users with an organization and set up endpoints to change our organizations.

Associate Organizations and Users

- Alter :users table
- Update models
- Update model tests

Now that we have organizations set up, we need to associate our `User` model with an organization. A user may create an organization and a user will be associated with an organization. The owner of the organization will have special priveleges that we will add later on (such as inviting others, deleting the organization, etc).

Alter :users table

The first and most straightforward thing to do is to generate a new migration and alter the existing `:users` table to include a new field for `:organization_id` so we know to which organization this user belongs.

```
$ mix ecto.gen.migration add_organization_reference_in_user
```

Then we alter the `:users` table and add an `:organization_id` which references our `:organizations` table. We're also setting an option to `nilify_all` so when a user is deleted, all of the user's owned organizations will have their `owner_id` set to `nil`. We do this since we want users whose organizations have been deleted to have a `nil` value rather than continue to reference a non-existent user.

```
/priv/repo/migrations/...reference_in_user.exs
commit: coming soon
```

```
defmodule PhoenixChat.Repo.Migrations.AddOrganizationReferenceInUser do
  use Ecto.Migration

  def change do
    alter table(:users) do
      add :organization_id, references(:organizations, on_delete: :nilify_all)
    end
  end
end
```

Now we need to update our models to reflect this change.

Update User and Organization models

The first change we'll make is to the `User` model, in which we will add an association between our `User` model and our `Organization` model. In this case, we're adding a `has_one` association between the `Organization` and the `User`, signifying that each user may only own one organization. Then we're adding a `belongs_to` association to link our users to a particular organization.

For a more detailed explanation of the `belongs_to/3` function, here's one from the Ecto docs.

> You should use belongs_to in the table that contains the foreign key. Imagine a company <-> manager relationship. If the company contains the manager_id in the underlying database table, we say the company belongs to manager.

We're also going to associate our organizations with our users.

`/web/models/user.ex`
commit: coming soon

```
defmodule PhoenixChat.User do
  use PhoenixChat.Web, :model
  alias PhoenixChat.Organization

  schema "users" do
    field :email, :string
    field :encrypted_password, :string
    field :username, :string
    field :password, :string, virtual: true

    has_one :owned_organization, Organization, foreign_key: :owner_id
    belongs_to :organization, Organization

    timestamps
  end

  ...
end
```

Then within our `Organization` model, we add a `has_many` association with `:admins` with the `:organization_id` as the foreign key, telling our `Organization` schema that it should expect to have (potentially) many admins. We also say that an organization `belongs_to` whichever `User` is the `:owner_id` foreign key that we set above in our `User` model.

We're also going to adjust our `changeset` to require an `:owner_id` when an organization is created

because we don't want organizations without an owner.

/web/models/organization.ex
commit: coming soon

```
defmodule PhoenixChat.Organization do
  use PhoenixChat.Web, :model
  alias PhoenixChat.{User, Repo}

  schema "organizations" do
    field :public_key, :string
    field :website, :string

    has_many  :admins, User, foreign_key: :organization_id
    belongs_to :owner, User, foreign_key: :owner_id

    timestamps()
  end

  @doc """
  Builds a changeset based on the `struct` and `params`.
  """
  def changeset(struct, params \\ %{}) do
    struct
    |> cast(params, [:website, :owner_id])
    |> validate_required([:website, :owner_id])
    |> update_change(:website, &set_uri_scheme/1)
    |> validate_change(:website, &validate_website/2)
    |> unique_constraint(:website)
    |> put_public_key()
    |> unique_constraint(:public_key)
  end

  ...
end
```

The last thing we should do before proceeding is update our model tests with these new associations.

Update model tests

We'll start with the `User` tests because they're simpler.

Our first test checks to make sure that we can create an organization with an `:owner_id` that references a user. In order to do this, we need to use `Repo.preload/3` (docs) to load the `:owned_organization` association with our user.

If you're not familiar with how relational databases handle associations, you can think of the association as a way of pointing to another struct (in our case, `User` pointing to `Organization`), but without the need to copy the entire organization into the user. But, if you want to actually check to make sure that the organization is the one you're intending to reference (`user.owned_organization`), you need to load it into the user, otherwise the `:owned_organization` of the user would not contain any data.

/test/models/user_test.exs
commit: coming soon

```
defmodule PhoenixChat.UserTest do
  use PhoenixChat.ModelCase

  alias PhoenixChat.{User, Organization, ConnCase}

  @valid_attrs %{email: "foo@bar.com", encrypted_password: "some content", username: "some
  @invalid_attrs %{}

  test "user can own an organization" do
    user = ConnCase.create_user!
    org  = Repo.insert! %Organization{website: "foo.com", owner_id: user.id, public_key: "t
    user = Repo.preload(user, :owned_organization)

    assert user.owned_organization == org
  end

  test "user belongs to organization" do
    org  = Repo.insert! %Organization{website: "foo.com", public_key: "test"}
    user = ConnCase.create_user!(%{organization_id: org.id})
           |> Repo.preload(:organization)

    assert user.organization == org
  end

  ...
end
```

Then we need to set up and alter a few of our `Organization` tests. The first change is to our `setup` block, in which we want to create a user for every test since every organization will need an owner. And since the database clears before every test, each new user will have an `id` of `1`. Because of that, we're adding `owner_id: 1` to our `@valid_attrs`, which should reference the user created at the start of each test.

/test/models/organization_test.exs
commit: coming soon

```
defmodule PhoenixChat.OrganizationTest do
  use PhoenixChat.ModelCase

  alias PhoenixChat.{Organization, ConnCase}

  @valid_attrs %{website: "foo.com", owner_id: 1}
  @invalid_attrs %{}

  setup do
    user = ConnCase.create_user!(%{id: 1})
    {:ok, %{user: user}}
  end

  test "organization belongs to owner", %{user: user} do
    org = Repo.insert! %Organization{website: "foo.com", owner_id: user.id, public_key: "te
          |> Repo.preload(:owner)

    assert org.owner == user
  end

  test "organization can have one admin" do
    org  = Repo.insert! %Organization{website: "foo.com", public_key: "test"}
    user = ConnCase.create_user!(%{username: "bar", email: "bar@foo.com", organization_id:

    org = Repo.preload(org, :admins)

    assert org.admins == [user]
  end

  test "organization has many admins" do
    org = Repo.insert! %Organization{website: "foo.com", public_key: "test"}
    user = ConnCase.create_user!(%{username: "bar", email: "bar@foo.com", organization_id:
    user2 = ConnCase.create_user!(%{username: "baz", email: "baz@qux.com", organization_id:

    org = Repo.preload(org, :admins)

    assert org.admins == [user, user2]
  end

  ...
end
```

Then we need to update a few of our other tests with the new `:owner_id` attribute so that their changeset becomes valid.

```
/test/models/organization_test.exs
commit: coming soon
```

```
...
test "changeset must have a unique public key generated on create" do
  changeset = Organization.changeset(%Organization{}, @valid_attrs)
  org1 = Repo.insert! changeset

  changeset = Organization.changeset(%Organization{}, %{website: "http://bar.com", owner_id
  org2 = Repo.insert! changeset

  assert org1.public_key != org2.public_key
end

test "changeset's website must be a valid url" do
  ...

  for valid_url <- some_valid_urls do
    changeset = Organization.changeset(%Organization{}, %{website: valid_url, owner_id: 1})

    assert changeset.valid?
  end
end
...
```

Now if you run your tests, all of your model tests will pass (though you should still have 5 errors since we haven't updated our `OrganizationControllerTest`).

Also, make sure you run your migration!

```
$ mix ecto.migrate
```

The next step is to make our CRUD endpoints to allow our users to create a new organization. From there, our users will have access to their public key that they can use to populate their `phoenix-chat` component and start sending and receiving messages routed to the proper organization.

CRUD Endpoints for Organization: Part 1

- Update OrganizationControllerTest
- Update OrganizationController
- Create organization on signup

Now that we have our `Organization` ready, we need to give our frontend the ability to add and update them. We're going to do this by adding some simple CRUD (Create Read Update Delete) operations.

Update OrganizationControllerTest

Most of the work for creating an `Organization` was in the model. In fact, there's almost nothing we have to change from the generator output in our `OrganizationController`, but we will need to make some significant changes to our tests.

This is a big block of code, but none of this is new--we covered each of these tests in the previous section on testing controllers. One thing to note is that we can call `create_user!/1` and `create_user!/2` since we're adding `use PhoenixChat.ConnCase` to our module. You should go through each test to make sure you understand what it's testing. If you run into issues, refer back to the chapter on testing controllers.

```
/test/controllers/organization_controller_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.OrganizationControllerTest do
  use PhoenixChat.ConnCase

  alias PhoenixChat.{Organization, Repo}
  @valid_attrs %{website: "http://www.foo.com", owner_id: 1}
  @invalid_attrs %{}

  test "lists all entries on index", %{conn: conn} do
    conn = get conn, organization_path(conn, :index)
    assert json_response(conn, 200)["data"] == []
  end

  test "shows chosen resource", %{conn: conn} do
    create_user!(%{id: 1})
    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    org = Repo.insert!(changeset)

    conn = get conn, organization_path(conn, :show, org)
```

```
    response = json_response(conn, 200)["data"]
    assert %{"public_key" => _, "owner_id" => 1, "website" => "http://www.foo.com",
      "id" => _} = response
  end

  test "does not show resource instead throws error when id is nonexistent", %{conn: conn}
    assert_error_sent 404, fn ->
      get conn, organization_path(conn, :show, -1)
    end
  end

  test "creates and renders resource when data is valid", %{conn: conn} do
    create_user!(%{id: 1})
    conn = post conn, organization_path(conn, :create), organization: @valid_attrs

    assert json_response(conn, 201)["data"]["id"]
    assert Repo.get_by(Organization, @valid_attrs)
  end

  test "does not create resource and renders errors when data is invalid", %{conn: conn} do
    conn = post conn, organization_path(conn, :create), organization: @invalid_attrs
    assert json_response(conn, 422)["errors"] != %{}
  end

  test "updates and renders chosen resource when data is valid", %{conn: conn} do
    create_user!(%{id: 1})
    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    org = Repo.insert!(changeset)

    new_attrs = %{website: "http://www.bar.com"}
    conn = put conn, organization_path(conn, :update, org), organization: new_attrs

    assert json_response(conn, 200)["data"]["id"]
    assert Repo.get_by(Organization, new_attrs)
  end

  test "does not update chosen resource and renders errors when data is invalid", %{conn: c
    create_user!(%{id: 1})
    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    org  = Repo.insert!(changeset)
    conn = put conn, organization_path(conn, :update, org), organization: %{website: nil}

    assert json_response(conn, 422)["errors"] != %{}
  end

  test "deletes chosen resource", %{conn: conn} do
    create_user!(%{id: 1})
    changeset = Organization.changeset(%Organization{}, @valid_attrs)
    org  = Repo.insert!(changeset)
    conn = delete conn, organization_path(conn, :delete, org)

    assert response(conn, 204)
```

```
    refute Repo.get(Organization, org.id)
  end
end
```

Now when you run your tests, they should all pass.

Update OrganizationController

The only thing we need to change in our controller is to add `scrub_params/2` (docs), which simply checks to make sure the required parameters are there and turns all empty strings into `nil`. Everything else was created by the generator.

```
/web/controllers/organization_controller.ex
commit: coming soon
```

```
defmodule PhoenixChat.OrganizationController do
  use PhoenixChat.Web, :controller

  alias PhoenixChat.{Organization, Repo}

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

  ...

end
```

Create organization on signup

We'll also want to give our user the ability to create an organization at the same time they create their account. To accomplish this, we're going to use `Ecto.Changeset.cast_assoc/2` in our `User.registration_changeset/2`, which will allow us to create and update the organization association at the same time we create our organization.

The first thing do is change our required fields in our `changeset`. Since we are allowing an organization to be created at the same time as a user, we may want to pass a newly-created `User` as the `association` parameter. If we do, then we don't need to pass in an `:owner_id` since we will handle that in our `cast_assoc/2` in our `registration_changeset/2` and not in our `changeset/2`.

```
/web/models/organization.ex
commit: coming soon
```

```
defmodule PhoenixChat.Organization do
  ...

  def changeset(struct, params \\ %{}, association \\ false) do
    required_fields = if association, do: [:website], else: [:website, :owner_id]

    struct
    |> cast(params, [:website, :owner_id])
    |> validate_required(required_fields)
    |> update_change(:website, &set_uri_scheme/1)
    |> validate_change(:website, &validate_website/2)
    |> unique_constraint(:website)
    |> put_public_key()
    |> unique_constraint(:public_key)
  end

  @doc """
  User for `cast_assoc/2` in `User.registration_changeset/2`. It's only difference
  from `changeset/3` is that it does not require an `owner_id`.
  """
  def owner_changeset(struct, params \\ %{}) do
    changeset(struct, params, true)
  end

  ...
end
```

So now within our `User` model, we need to update our `registration_changeset/2` to handle the association mentioned above. This `cast_assoc/2` will look for an `:owned_organization` field in the params and use that to assign an `:owned_organization` association between the user currently being created and a new organization.

```
/web/models/user.ex
commit: coming soon
```

```
...
  def registration_changeset(model, params) do
    model
    |> changeset(params)
    |> cast(params, ~w(password), [])
    |> cast_assoc(:owned_organization, with: &Organization.owner_changeset/2)
    |> validate_length(:password, min: 6, max: 100)
    |> put_encrypted_pw
  end
...
```

Note that our `cast_assoc` is looking for an `:owned_organization` key, so we will need to change the data we're sending on account creation to the following format:

```
{
  username,
  email,
  password,
  owned_organization: {
    website
  }
}
```

Then for good housekeeping, let's write a couple tests for this change. For our `Organization` test, we are making sure that the new `owner_changeset/2` works as expected.

```
/test/models/organization_test.ex
commit: coming soon
```

```
...
  test "owner changeset does not require an owner id" do
    changeset = Organization.owner_changeset(%Organization{}, %{website: "foo.com"})
    assert changeset.valid?
  end
...
```

Then within our `User` test, we are updating our `registration_changeset/2` test to make sure that a new `:owned_organization` is created if it's provided.

```
/test/models/user_test.ex
commit: coming soon
```

```
...
  test "registration changeset with valid attributes" do
    valid_attrs = Map.put(@valid_attrs, :password, "password")
    changeset = User.registration_changeset(%User{}, valid_attrs)
    assert changeset.valid?
    assert get_change(changeset, :encrypted_password)

    # Test that a new owned_organization is created properly if it is provided
    valid_attrs = %{owned_organization: %{website: "http://www.foo.com"}}
    changeset = User.registration_changeset(changeset, valid_attrs)
    assert changeset.valid?
    assert get_change(changeset, :owned_organization).changes.website == "http://www.foo.co
    assert get_change(changeset, :owned_organization).changes.public_key
  end
...
```

And that's it. Most of the boilerplate code for our CRUD endpoints was built by our generator. We barely had to change `OrganizationController` and we didn't have to touch our `OrganizationView`.

CRUD Endpoints for Organization: Part 2

- Additional validation for organization creation
- Return organization data on authorization

Additional validations

Since we currently only want each user to be associated with one organization (either as an admin or an owner), we need to add a validation to make sure that a user cannot create or be associated with more than one account. It's easy to disable this endpoint on the frontend, but we also want to make sure that a feisty customer can't break his own account by using a `curl` command.

We're going to create another validation just like we did with our website validation above, using `validate_change/3` (docs). In this case, we're creating a function called `validate_new_owner_admin/2` to make sure the user trying to create an organization is not already associated with an existing account. Just like we did in our `auth/me` route, we're finding a user, preloading the associations, then checking to see if we get an organization. If we do, we return an error.

```
/web/models/organization.ex
commit: coming soon
```

```
...
  def changeset(struct, params \\ %{}, association \\ false) do
    required_fields = if association, do: [:website], else: [:website, :owner_id]

    struct
    |> cast(params, [:website, :owner_id])
    |> validate_required(required_fields)
    |> validate_change(:owner_id, &validate_new_owner_admin/2)
    |> update_change(:website, &set_uri_scheme/1)
    |> validate_change(:website, &validate_website/2)
    |> unique_constraint(:website)
    |> put_public_key()
    |> unique_constraint(:public_key)
  end

  ...

  defp validate_new_owner_admin(:owner_id, owner_id) do
    user = Repo.get! User, owner_id
    org = Repo.preload(user, :organization).organization || Repo.preload(user, :owned_organ

    if org do
      [owner_id: "user is owner or admin of existing organiation"]
    else
      []
    end
  end
  ...
```

Update auth/me

Now that we can create an organization, we need to be able to return the `public_key` to the user in a consistent manner. Currently, when a user creates an organization we return the key in the HTTP response, but after that response the user has no way to access her key again. To solve this, we're going to update the `auth/me` route to return the `public_key` of the organization (if there is one) that the user owns or is an admin.

Although we currently only have `owners` and no `admins`, we want to check if the current user is either an owner or an admin of an existing organization. We check this by using `Repo.preload/3` (docs) to load the associations of both `:organization` and `:owned_organization`. We then check to see if there is a match in either one. If there is no match, it returns `nil` so we want to render just the user data without the organization data.

```
/web/controllers/auth_controller.ex
commit: coming soon
```

```
...

  def me(conn, _params) do
    user = Guardian.Plug.current_resource(conn)
    org = Repo.preload(user, :organization).organization || Repo.preload(user, :owned_organ
    case org do
      nil -> render(conn, UserView, "show.json", user: user)
      org -> render(conn, UserView, "show.json", user: user, org: org)
    end
  end
...
```

Then we need to update our `UserView` to handle this new data.

```
/web/views/user_view.ex
commit: coming soon
```

```
...

def render("show.json", %{user: user, org: org}) do
  %{data: render_one(user, UserView, "user_organization.json", org: org)}
end

def render("show.json", %{user: user}) do
  %{data: render_one(user, UserView, "user.json")}
end

...

def render("user_organization.json", %{user: user, org: org}) do
  %{email: user.email,
    id: user.id,
    username: user.username,
    website: org.website,
    public_key: org.public_key}
end
```

And now when we authorize a user, we check to see if that user is associated with an organization and return the organization's data along with the authorization request. The next step is to connect our frontend and allow our existing user to create an organization.

Controlling and Validating Forms

- Controlled forms
- Validating forms
- Giving UI feedback

Controlled form

Now that our components are starting to look closer to what we would like, we should tie our form in with the state of our component. When people first start using Redux, they have a habit of tying everything into `state`. But just because you have the global `store` doesn't mean that everything belongs there. Or as Redux founder Dan Abramov said:

> You don't need a single source of truth for *everything*. Just make sure you have a single source of truth for any particular thing.

A form like a signup or login form is a good example of transitory data that does not need to be saved in your Redux `store`. If a user navigates away and comes back, does he expect that Redux would remember the status of that form? Probably not. In fact, she would probably think it's weird.

It is for this reason that our `Signup` and `Login` components will have their values tied to local `state`. First we should set the initial values of our form state.

```
/app/components/Signup/index.js
commit: coming soon
```

```
...
export class Signup extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      username: "",
      email: "",
      password: "",
      passwordVerify: ""
    }
    this.submit = this.submit.bind(this)
  }
  ...
}
```

/app/components/Login/index.js
commit: coming soon

```
...
export class Login extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      email: "",
      password: ""
    }
  }
  ...
}
```

Update Login component

Then we need three more things. We need a `handleChange` function (or whatever you prefer to name it) that will update our state based on the input, we need to set the `value` of each input to the value of the corresponding state value, and we need to change the values that we submit to our new state values.

/app/components/Login/index.js
commit: coming soon

```
export class Login extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
```

```
      email: "",
      password: ""
    }
    this.submit = this.submit.bind(this)
  }

  submit() {
    const user = {
      email: this.state.email,
      password: this.state.password
    }
    this.props.dispatch(Actions.userLogin(user))
  }

  handleChange(input, e) {
    this.setState({ [input]: e.target.value })
  }

  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          <div className={style.inputGroup}>
            <input
              value={this.state.email}
              onChange={ e => { this.handleChange("email", e) }}
              placeholder="Email"
              className={style.input}
              type="text" />
          </div>
          <div className={style.inputGroup}>
            <input
              value={this.state.password}
              onChange={ e => { this.handleChange("password", e) }}
              placeholder="Password"
              className={style.input}
              type="password" />
          </div>
          <Button
            onClick={this.submit}
            _style={{ width: "100%" }}
            type="primary">
            Submit
          </Button>
        </div>
      </div>
    )
  }
}
...
```

So now, when you type something into the `input` field, it triggers the `onChange` function, which in turn triggers the `this.handleChange` function that we defined. Within that function, we set the local `state` of the relevant input to the value of the input, which is also tied to the relevant `this.state[value]`. We've provided a handy chart showing the general flow of controlled forms below.

Handy chart coming soon

This might sound like a roundabout way of filling out forms, and that's because it is. But there are advantages of doing it this way (as you will see shortly).

Update Signup component

One things users have come to expect from forms is immediate feedback as to the validity of their form inputs because it's annoying to fill out an entire form only to submit the form and have an error telling you that it's invalid (especially when it then erases all existing form data, which many websites do for some reason).

Let's start by updating our `Signup` component to use a controlled form just like our `Login` component above.

```
/app/components/Signup/index.js
commit: coming soon
```

```
export class Signup extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      username: "",
      email: "",
      password: "",
      passwordVerify: ""
    }
    this.submit = this.submit.bind(this)
  }

  submit() {
    const user = {
      username: this.state.username,
      email: this.state.email,
      password: this.state.password
    }
    this.props.dispatch(Actions.userNew(user))
  }

  handleChange(input, e) {
    this.setState({ [input]: e.target.value })
  }
```

```
  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("username", e) }}
              value={this.state.username}
              placeholder="Username"
              className={style.input}
              type="text" />
          </div>
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("email", e) }}
              value={this.state.email}
              placeholder="Email"
              className={style.input}
              type="text" />
          </div>
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("password", e) }}
              value={this.state.password}
              placeholder="Password"
              className={style.input}
              type="password" />
          </div>
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("passwordVerify", e) }}
              value={this.state.passwordVerify}
              placeholder="Verify Password"
              className={style.input}
              type="password" />
          </div>
          <Button
            onClick={this.submit}
            _style={{ width: "100%" }}
            type="primary">
            Submit
          </Button>
        </div>
      </div>
    )
  }
}
```

And now that we have a controlled form, we can use the values we received to validate our inputs before

a user submits them. We're going to create two validations to make it easy for a user to know in advance whether the email address they entered is valid and whether the password match.

We'll start by creating valid and invalid constants that we can use as out inline style objects. If valid, the input will receive a green underline; if invalid, it will receive a red underline.

The first function is `validatePassword`, which compares the current state of our passwords and returns invalid if less than 6 characters or when the passwords don't match. It returns valid when the password are at least 6 characters and match.

The second is `validateEmail`, which uses a regular expression to make sure the email address contains an `@` symbol, which is a bit oversimplified, but it servers our purposes.

```
/app/components/Signup/index.js
commit: coming soon
```

```
...

const validInput = { borderBottom: "3px solid #4CAF50" }
const invalidInput = { borderBottom: "3px solid #F44336" }

export class Signup extends React.Component {
  ...

  validateEmail() {
    if (this.state.email.length < 1) return {}
    return /@/.test(this.state.email) ? validInput : invalidInput
  }

  validatePassword() {
    if (this.state.password.length < 1) return {}
    if (this.state.password.length < 6) return invalidInput
    return this.state.password === this.state.passwordVerify ? validInput : invalidInput
  }

  ...
}
```

Then we need to connect these functions to our inputs. We're going to add a `style` tag to our `email` and `passwordVerify` inputs, and in that `style` tag we call the relevant function to validate. Note that in the event of no match (in both cases, if length < 1), we're returning an empty object.

```
/app/components/Signup/index.js
commit: coming soon
```

```
...

export class Signup extends React.Component {
  ...

  render() {

    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          ...
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("email", e) }}
              style={this.validateEmail()}
              value={this.state.email}
              placeholder="Email"
              className={style.input}
              type="text" />
          </div>
          ...
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("passwordVerify", e) }}
              style={this.validatePassword()}
              value={this.state.passwordVerify}
              placeholder="Verify Password"
              className={style.input}
              type="password" />
          </div>
        </div>
      </div>
    )
  }
}
```

And now you have form validation. As you can see, React's inline styling makes this really easy.

Create and Join an Organization on Signup

- Update Signup component
- Connect to endpoint
- Create new action
- Add header to Chat component

Now that we can create an organization on our backend, let's add a new view on our frontend that will access our new endpoints. We already have a `/settings` route, so let's use that for creating an organization from an existing account and let's update our `Signup` component to handle an optional organization input.

Since creating an organization from initial signup is easier, we will start with that the move on to creating a user from our `/settings` route.

Update Signup component

As it turns out, this requires very little effort. All we have to do is add a `website` field to our form and send it to our backend in the format that it expects.

`/app/components/Signup/index.js`
commit: coming soon

```
...

export class Signup extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      username: "",
      email: "",
      password: "",
      passwordVerify: "",
      website: ""
    }
    this.submit = this.submit.bind(this)
  }

  submit() {
    if (this.state.website) {
      const userWithOrg = {
```

```
        username: this.state.username,
        email: this.state.email,
        password: this.state.password,
        owned_organization: {
          website: this.state.website
        }
      }
      this.props.dispatch(Actions.userNew(userWithOrg))
    } else {
      const user = {
        username: this.state.username,
        email: this.state.email,
        password: this.state.password,
      }
      this.props.dispatch(Actions.userNew(user))
    }
  }

  render() {
    return (
      <div className={style.wrapper}>
        <div className={style.form}>
          ...
          <div className={style.inputGroup}>
            <input
              onChange={e => { this.handleChange("website", e) }}
              value={this.state.website}
              placeholder="Website (optional)"
              className={style.input}
              type="text" />
          </div>
          ...
        </div>
      </div>
    )
  }
}
...
```

Now go ahead and logout by deleting your token from localStorage (`delete localStorage.token`), then refresh the page. You are now logged out and unauthenticated. Now, when you create a valid account with a valid website, you'll create both a user and an organization at the same time.

To check to make sure this is working, try putting a `console.log` in your `userAuth` action. If you do, you should now see `public_key` and `website` along with the data you previously received from the server.

The next step is to allow a user who already has an account but no organization create an organization from the `/settings` route.

Create an action

Go ahead and sign out again by deleting your token and refreshing, then create a new user *without* including a website. From here, we'll create a new Redux action that connects to our endpoint so that the existing user can create a new organization.

Everything in this action should look familiar. In the event of an error, we're passing our errors to our reducers but we aren't going to bother with proper error handling at the moment.

The `organization` parameter we're sending to our server expects a `website` and an `owner_id`, which is the ID of the currently logged-in user. At some point in the future, we will refactor this to handle authentication on the backend using `Guardian`, but for now, let's just pass the user's ID we're storing in `props` as the `owner_id`.

If the organization is created successfully, we will re-authorize the user so we can get the credentials for this organization, which come from our recently-changed `auth/me` route.

```
/app/redux/actions.js
commit: coming soon
```

```
Actions.organizationNew = function organizationNew(organization) {
  return dispatch => fetch(`${API_HOST}/api/organizations`, {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ organization })
  })
  .then((res) => { return res.json() })
  .then((res) => {
    if (res.errors) {
      return dispatch({
        type: "ORGANIZATION_NEW_ERROR",
        payload: {
          errors: res.errors
        }
      })
    }
    dispatch({
      type: "ORGANIZATION_NEW"
    })
    return dispatch(Actions.userAuth())
  })
  .catch((err) => {
    console.warn(err)
  })
}
```

We should also update our `user` reducer to take in the new information that we get from the `auth/me` endpoint. Recall that we now receive `public_key` and `website` from that endpoint.

```
function user(state = {
  email: "",
  username: "",
  id: "",
  public_key: "",
  website: ""
}, action) {
  switch (action.type) {

    ...

    case "USER_AUTH":
      return Object.assign({}, state, {
        email: action.payload.user.email,
        username: action.payload.user.username,
        id: action.payload.user.id,
        public_key: action.payload.user.public_key,
        website: action.payload.user.website
      })
    default: return state
  }
})
```

The next step is to update our `Settings` component and connect it to this action. But before we do that, let's update our `Chat` component with a header that can easily link us to the `/settings` route.

Add header to Chat component

Now that we have an endpoint, we should add a header to the top of our app that contains a link to our `/settings` route. From there, can add the functionality to create a new organization.

We're going to make a fairly simple header that shows the name of the currently-selected anonymous user (i.e. room), displays a (currently placeholder) avatar, and shows when they were last active (currently a placeholder as well).

If there is no current room selected (`!this.state.currentRoom`), then we render an empty header with just the settings cog.

```
/app/components/Chat/index.js
commit: coming soon
```

```
...
import { Link } from "react-router"
...
```

```
export class Chat extends React.Component {
  ...

  renderHeader() {
    if (!this.state.currentRoom) {
      return (
        <div className={style.header}>
          <div />
          <Link to="settings" className={style.settings}>
            <img
              alt="link to settings"
              className={style.cog}
              src="https://s3.amazonaws.com/learnphoenix-static-assets/icons/cog.png" />
          </Link>
        </div>
      )
    }

    const avatar = {
      height: "40px",
      width: "40px",
      background: "#ccc",
      border: "1px solid #888",
      borderRadius: "50%"
    }

    return (
      <div className={style.header}>
        <div className={style.identity}>
          <div style={avatar} />
          <div className={style.titleGroup}>
            <h3 className={style.title}>
              { this.state.currentRoom }
            </h3>
            <div className={style.lastActive}>
              Last active: __ minutes ago
            </div>
          </div>
        </div>
        <Link to="settings" className={style.settings}>
          <img
            alt="link to settings"
            className={style.cog}
            src="https://s3.amazonaws.com/learnphoenix-static-assets/icons/cog.png" />
        </Link>
      </div>
    )
  }

  render() {
    return (
      <div>
```

```
        <Sidebar
          presences={this.state.presences}
          onRoomClick={this.changeChatroom}
          lobbyList={this.state.lobbyList} />
        <div className={style.chatWrapper}>
          { this.renderHeader() }
          <div
            className={style.chatContainer}
            ref={ref => { this.chatContainer = ref }}>
            { this.renderEmpty() }
            { this.renderMessages() }
          </div>
          { this.renderInput() }
        </div>
        { this.props.children }
      </div>
    )
  }
}
...
```

From here we can add some styling to make our header look more like we would expect.

```
/app/components/Home/style.css
commit: coming soon
```

```
...

.header {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  background: rgb(238, 238, 239);
  border: 1px solid rgb(213, 213, 213);
  height: 60px;
  display: flex;
  flex-flow: row nowrap;
  justify-content: space-between;
  align-items: center;
  z-index: 100;
}

.identity {
  display: flex;
  flex-flow: row nowrap;
  align-items: center;
  padding-left: 30px;
}

.titleGroup {
  padding-left: 10px;
}

.title {
  font-size: 1em;
  font-weight: normal;
  color: #333;
}

.lastActive {
  padding-top: 6px;
  color: #727272;
  font-size: 0.8em;
}

.settings {
  padding-right: 30px;
}

.cog {
  filter: invert(40%);
  height: 35px;
}
```

Now we have an easy way to navigate to our `/settings` route by clicking on the cog in the header. From here, we need to add the ability to create an organization from the settings page or display the affiliated organization's API key.

Create Organization from Settings Route

- Update Settings component

So at this point, we can create an organization on signup, but we cannot create an organization after we've signed up. What we need is a settings page that will allow unaffiliated administrators to create a new organization and will allow affiliated administrators to see their API key.

Update Settings component

Now that we can easily link to our `/settings` route, we should add the ability to create a new organization from within that component.

This is a lot of code, but none of it is new. We're adding a form just like we did in all our other forms and we're selectively rendering either a form or an API key depending on whether or not the current user has an associated organization.

/app/components/Settings/index.js
commit: coming soon

```
import React from "react"
import cssModules from "react-css-modules"
import { connect } from "react-redux"
import { Link } from "react-router"
import style from "./style.css"
import Actions from "../../redux/actions"

import Button from "../Button"

export class Settings extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      website: ""
    }
    this.submit = this.submit.bind(this)
  }

  submit() {
    const organization = {
      website: this.state.website,
      owner_id: this.props.user.id
```

```
    }
    this.props.dispatch(Actions.organizationNew(organization))
  }

  handleChange(input, e) {
    this.setState({ [input]: e.target.value })
  }

  renderOrganization() {
    if (!this.props.user.public_key) {
      return (
        <div className={style.inputGroup}>
          <input
            onChange={e => { this.handleChange("website", e) }}
            placeholder="Website (e.g. https://phoenixchat.io)"
            type="text"
            className={style.input} />
          <Button
            onClick={this.submit}
            style={{ marginTop: "15px" }}
            type="primary">
            Create Organization
          </Button>
        </div>
      )
    }
    return (
      <div className={style.apiKey}>
        Your API key: {this.props.user.public_key}
      </div>
    )
  }

  render() {
    return (
      <div>
        <div className={style.nav}>
          <Link
            className={style.logo}
            to="/">
            <img
              alt="learnphoenix logo"
              className={style.image}
              src="https://s3.amazonaws.com/learnphoenix-static-assets/favicons/favicon-96x
            <div className={style.title}>
              PhoenixChat.io
            </div>
          </Link>
          <div />
        </div>
        <div className={style.settings}>
          {this.renderOrganization()}
```

```
          {this.renderOrganization()}
        </div>
      </div>
    )
  }
}

Settings.propTypes = {
  user: React.PropTypes.object,
  organization: React.PropTypes.object,
  dispatch: React.PropTypes.func
}

const mapStateToProps = state => ({
  user: state.user
})

export default connect(mapStateToProps)(cssModules(Settings, style))
```

That's a lot of code, but it's nothing you haven't seen before. And now to make this page usable, we need to add some styling.

```
$ touch app/components/Settings/{style.css,spec.js}
```

/app/components/Settings/index.js
commit: coming soon

```
.settings {
  display: flex;
  flex-flow: column nowrap;
  align-items: center;
  height: calc(100vh - 60px);
  margin-top: 50px;
  position: relative;
  z-index: 100;
}

.nav {
  height: 60px;
  padding: 0 30px;
  display: flex;
  flex-flow: row nowrap;
  justify-content: space-between;
  align-items: center;
}

.logo {
  display: flex;
```

```
  display: flex;
  flex-flow: row nowrap;
  align-items: center;
}

.title {
  font-size: 1.5em;
  cursor: pointer;
  color: #4c4c4c;
}

.image {
  height: 50px;
  padding-right: 5px;
}

.input {
  padding: 1rem 1rem;
  border-radius: 3px;
  border: 1px solid #ccc;
  font-size: 1.1em;
  outline: none;
  width: 400px;
}

.inputGroup {
  display: flex;
  flex-flow: column nowrap;
}

.apiKey {
  font-size: 1.5em;
}
```

Now, if you're signed in and not affiliated with an organization, you can create one. Otherwise, it shows your API key. Our settings page is a little bit sparse, but to be fair, we don't have a lot of settings at the moment.

Settings tests

We should also write a few tests. This component really doesn't do all that much, so all we should really test is that it renders, it has a submit function, and that the proper organization option is rendered (form with button or the key).

```
import React from 'react'
import expect from 'expect'
import { shallow } from 'enzyme'

import { Settings } from './'

const props = {
  user: {
    email: "foo",
    id: 1234,
    public_key: "asdf"
  }
}

describe('<Settings />', () => {
  it('should render', () => {
    const renderedComponent = shallow(
      <Settings {...props} />
    )
    expect(renderedComponent.is('div')).toEqual(true)
  })
  it('should have a submit function', () => {
    const component = new Settings()
    expect(component.submit).toExist()
  })
  // TODO does not work
  // it('should render a button if no key is present', () => {
  //   const renderedComponent = shallow(
  //     <Settings {...props} public_key="" />
  //   )
  //   expect(renderedComponent.find('button').length).toEqual(1)
  // })
  // it('should render no button if key is present', () => {
  //   const renderedComponent = shallow(
  //     <Settings {...props} />
  //   )
  //   expect(renderedComponent.find('button').length).toEqual(0)
  // })
})
```

Now that we have access to our public key, let's add that to our npm component, which will pass it along to our API in order to properly route messages.

Pass API Key from Frontend

- Add API key to phoenix-chat
- Pass key via socket

In this section, we're just going to add the public key that we generated to our `PhoenixChat` component, then update our component to handle that key.

Add token to PhoenixChat

In `phoenix-chat-frontend`, all we need to do is add the `token` to our `PhoenixChat` component. Obviously, add the proper token instead of `NlkfVrAM9/`.

```
/app/components/Home/index.js
commit: coming soon
```

```
...

export class Home extends React.Component {
  ...

  render() {
    if (this.props.user.email) {
      return (
        <Chat>
          <PhoenixChat token="NlkfVrAM9/" />
        </Chat>
      )
    }
    return (
      <div className={style.leader}>
        <h1 className={style.title}>Phoenix Chat</h1>
        { this.state.formState === "signup" ? <Signup /> : null }
        { this.state.formState === "login" ? <Login /> : null }
        { this.renderToggleContent() }
        <PhoenixChat token="NlkfVrAM9/" />
        <img
          role="presentation"
          className={style.circles}
          src="https://s3.amazonaws.com/learnphoenix-static-assets/images/circles-full.png"
      </div>
    )
  }
}
...
```

Since we're already passing `this.props.user` as the `params` in our `Chat` component when we connect to the socket, we don't need to change anything else.

And that's it for our frontend. Now your `phoeinx-chat` component has access to `token` via `this.props.token`. Make sure you have `npm run watch` going in your `phoenix-chat` directory so your app will update when you make changes.

Update phoenix-chat

Now all we need to do is add the `token` to our parameters before passing it along to the socket. Alternatively, we could pass the public key along with every message, but passing it along in the socket will save us some bandwidth.

```
/src/PhoenixChat.jsx
commit: coming soon
```

```
...

export class PhoenixChat extends React.Component {
  ...

  componentDidMount() {
    if (!localStorage.phoenix_chat_uuid) {
      localStorage.phoenix_chat_uuid = uuid.v4()
    }

    this.uuid = localStorage.phoenix_chat_uuid
    const params = { uuid: this.uuid, public_key: this.props.token }
    this.socket = new Socket("ws://localhost:4000/socket", { params })
    this.socket.connect()

    this.configureChannels(this.uuid)
  }
  ...

}
...
```

And believe it or not, that's it! We're done configuring the frontend with API keys. The rest of the work will be in our API to route messages to the appropriate organization.

Routing Messages Using API Keys

- Update UserSocket
- Broadcast by public_key
- Populate lobby_list by public_key

At this point in our app we can create a new organization from our frontend, but the organizations don't really mean anything since our messages aren't associated with an organization. So in order to separate the messages based on organization, we need to update our channels so that admins can only see messages of other admins and so that admins can only see messages associated with the API keys of their organizations.

Rather than send the `public_key` as data along with every message, we've required the client to connect to the socket with the key in the params, so we now have access to that key via `assign/3` (docs), which we've used a number of times already.

Update UserSocket

We will start by updating our user socket. In our `connect/2` function, we're assigning `:public_key` from our params whether the user is an anonymous user or an administrator. Note that in our anonymous client the token comes directly in the params while in our admin dashboard, it comes from our user information.

```
/web/channels/user_socket.ex
commit: coming soon
```

```
...
  def connect(params, socket) do
    user_id = params["id"]
    user = user_id && Repo.get(User, user_id)

    socket = if user do
        socket
        |> assign(:user_id, user_id)
        |> assign(:username, user.username)
        |> assign(:email, user.email)
      else
        socket
        |> assign(:user_id, nil)
        |> assign(:uuid, params["uuid"])
      end
        |> assign(:public_key, params["public_key"])

    {:ok, socket}
  end
...
```

Then we should update our tests to take in this new parameter.

```
/test/channels/user_socket_test.exs
commit: coming soon
```

```
defmodule PhoenixChat.UserSocketTest do
  use PhoenixChat.ChannelCase

  alias PhoenixChat.{Repo, User, UserSocket}

  test "connecting to user socket as logged-in user" do
    admin = Repo.insert!(%User{email: "admin@bar.com", username: "admin"})

    {:ok, socket} = connect(UserSocket, %{"id" => admin.id, "public_key" => "pub_key"})
    {:ok, _, socket} = subscribe_and_join(socket, "room:1", %{})

    assert socket.assigns.user_id == admin.id
    assert socket.assigns.email == admin.email
    assert socket.assigns.username == admin.username
  end

  test "connecting to user socket as anonymous user" do
    {:ok, socket} = connect(UserSocket, %{"uuid" => 25, "public_key" => "pub_key"})
    {:ok, _, socket} = subscribe_and_join(socket, "room:25", %{})

    refute socket.assigns.user_id
    assert socket.assigns.uuid == 25
    assert socket.assigns.public_key == "pub_key"
  end
end
```

Broadcast by public_key

Now it's time to match the chats from anonymous users to the admins in the organization associated with the `public_key` they're passing along.

Note that we are using the public key we stored in `socket.assigns` in the previous lesson. We use this public key to filter out the admins that do not have the same public keys from receiving the `"lobby_list"` event for a particular user. This means that a user public key `"foo"` will only be visible to admins with public key `"foo"`.

```
/web/channels/admin_channel.ex
commit: coming soon
```

```
defmodule PhoenixChat.AdminChannel do
  ...

  def handle_info(:after_join, socket) do
    push socket, "presence_state", Presence.list(socket)
    %{assigns: assigns} = socket
    id = assigns.user_id || assigns.uuid
    LobbyList.insert(id)
    broadcast! socket, "lobby_list", %{uuid: id, public_key: assigns.public_key}
    {:ok, _} = Presence.track(socket, id, %{
        online_at: inspect(System.system_time(:seconds))
      })
    {:noreply, socket}
  end

  def handle_out("lobby_list", payload, socket) do
    %{assigns: assigns} = socket
    if assigns.user_id && assigns.public_key == payload.public_key do
      push socket, "lobby_list", payload
    end
    {:noreply, socket}
  end
end
```

Then we need to update our admin channel tests to handle this new information.

```
/test/channels/admin_channel_test.exs
commit: coming soon
```

```
...
  test "joining admin:active_users as admin" do
    LobbyList.insert("foo")
    LobbyList.insert("bar")

    {:ok, %{lobby_list: lobby_list}, _socket} =
      socket("user_id", %{user_id: 1, public_key: "pub_key"})
      |> subscribe_and_join(AdminChannel, "admin:active_users")

    assert length(lobby_list) == 2
    assert_push "lobby_list", %{uuid: 1}
    assert_push "presence_state", %{}
    assert_push "presence_diff", %{joins: %{"1" => %{}}}
  end

  test "non-admin users do not receive the 'lobby_list' event on join" do
    {:ok, %{lobby_list: _}, _} =
      socket("user_id", %{user_id: nil, uuid: 5, public_key: "pub_key"})
      |> subscribe_and_join(AdminChannel, "admin:active_users")

    refute_push "lobby_list", %{}
  end
...
```

So now we're properly sorting which users will receive the lobby_list associated with a particular public_key. The next step is to ensure that each lobby_list is populated with the right anonymous users.

Populate lobby_list by public_key

We want to populate the list of chatrooms in the admin's sidebar only with chatrooms/users that have the same public key. To do this, we store `uuids` in ETS using their `public_key` as the filter. Then we'll create a new function, `LobbyList.lookup/1`, that will allow us to retrieve all uuids in the LobbyList table with the same public_key.

Note that we set the `:bag` option for our :ets table so that it can store multiple values with the same key. You can read more about ETS tables and the difference between a `:bag` and a `:set` in the (docs).

```
/lib/phoenix_chat/lobby_list.ex
commit: coming soon
```

```
defmodule PhoenixChat.LobbyList do

  @table __MODULE__

  @doc """
  Create an :ets table for this module. We set the `:bag` option so that we can
  store multiple values with the same keys.
  """
  def init do
    opts = [:public, :named_table, {:write_concurrency, true}, {:read_concurrency, false},
    :ets.new(@table, opts)
  end

  def insert(public_key, uuid) do
    :ets.insert(@table, {public_key, uuid})
  end

  def delete(public_key) do
    :ets.delete(@table, public_key)
  end

  def lookup(public_key) do
    @table
    |> :ets.lookup(public_key)
    |> Enum.map(fn {_, uuid} -> uuid end)
  end
end
```

And now that we've significantly altered our ETS table, we need to change our `admin_channel` to use our handy new `lookup/2` function.

The first change is in our `join/3` function, in which we get a list of uuids of users with the same public key as the current admin and send it back to the frontend to be displayed in the sidebar.

Then, in our `handle_info/2` function, we use `LobbyList.insert/2`, which now takes in a `public_key`, to keep track of rooms to be displayed to the proper admins. And finally we push the presence state of these users to track which users are online.

```
defmodule PhoenixChat.AdminChannel do
  ...

  def join("admin:active_users", payload, socket) do
    authorize(payload, fn ->
      send(self, :after_join)

      public_key = socket.assigns.public_key
      lobby_list = LobbyList.lookup(public_key)
      {:ok, %{lobby_list: lobby_list}, socket}
    end)
  end

  def handle_info(:after_join, socket) do
    %{assigns: assigns} = socket
    id = assigns.user_id || assigns.uuid

    # Keep track of rooms to be displayed to admins
    LobbyList.insert(assigns.public_key, id)
    broadcast! socket, "lobby_list", %{uuid: id, public_key: assigns.public_key}

    # Keep track of users that are online
    push socket, "presence_state", Presence.list(socket)
    {:ok, _} = Presence.track(socket, id, %{
        online_at: inspect(System.system_time(:seconds))
      })
    {:noreply, socket}
  end
  ...

end
```

Finally, we need to make some minor alterations to our tests to handle the changes in our `LobbyList`.

```
/test/channels/admin_channel_test.exs
commit: coming soon
```

```
...
test "joining admin:active_users as admin" do
  LobbyList.insert("pub_key", "id1")
  LobbyList.insert("pub_key", "id2")
  ...

end
...
```

Granted, we aren't doing much in the way of authentication, but we now have a functional app! Try

opening an incognito window or a different browser, go to `localhost:3000`, and sign up. You'll see that you no longer have access to the chats from other organizations.

Store and Track Recent Activity

- Generate fake name
- Generate fake avatar
- Store and send last active
- Pass along last message

coming soon

Display Recent Activity

- Show when last online
- Use MomentJs

Coming soon

In this section, we're going to use Presence to display when a user was last active. To handle time, we're going to use Moment.js, which gives us easy access to things like relative time.

Currently, there is no modular way to import Moment.js and it's frustratingly large (about 50kb), but at some point they will make it modular and we can replace the import with just the relative time function. (If this change has happened and we are not aware of it, please send us an email at info@learnphoenix.io and we'll update)

Update users when present

TODO

Update user list

On our frontend, the first thing we should do is update our user list to take in some additional information about our user. Currently, we're just displaying the user's uuid. What we want to be able to see an avatar for easy reference, a generated text name (again, for easy reference), the last messages that was sent between the two users, and the time they were last active.

It makes more sense to handle random avatar generation and names on the server, and we have no way to access the last message sent from the frontend, so we will just use placeholders for now.

Our user element is basically one row with three columns. The first column contains the avatar, the second contains the user id (and later the generated words) followed by the latest message, and the third contains the last time the user was active. The avatar and last active will have fixed width, while the name and text will grow to fill the remaining space.

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
...

export const Sidebar = (props) => {
  ...

  const renderList = lobbyList
  .filter(({ id }) => { return id.length === 36 })
  .sort(orderByActivity)
  .map(({ id, active }) => {
    const newStyle = active ? { boxShadow: "inset 0px 0px 6px 4px rgba(58, 155, 207, 0.6)"

    const avatar = {
      height: "40px",
      width: "40px",
      background: "#ccc",
      border: "1px solid #888",
      borderRadius: "50%"
    }

    return (
      <div
        style={newStyle}
        className={style.user}
        key={id}
        onClick={() => { props.onRoomClick(id) }}>
        <div className={style.avatar}>
          <div style={avatar} />
        </div>
        <div className={style.content}>
          <div className={style.name}>
            { id }
          </div>
          <div className={style.lastMessage}>
            This was our last message
          </div>
        </div>
        <div className={style.activity}>
          test
        </div>
      </div>
    )
  })

  ...
}

...
```

Then we need to add styles to reflect those changes. Nothing is new here, except that we are composing

a few columns from `.column` and using `white-space: nowrap` to stop our uuid from wrapping around. If the message or the uuid is too long, we simply hide it. At some point, we'll introduce something more sophisticated that can handle ellipses (`...`), but this will be fine for now.

/app/components/Sidebar/style.css
commit: coming soon

```
...

.user {
  display: flex;
  flex-flow: row nowrap;
  align-items: center;
  padding: 0.5rem;
  border-bottom: 1px solid #ccc;
  height: 70px;
  background: white;
  cursor: pointer;
  transition: background 0.1s ease;
  box-shadow: 0 3px 3px -2px rgba(0,0,0,0.25);
}
.user:hover {
  background: #eeeeee;
}

.column {
  display: flex;
  flex-flow: column nowrap;
}

.avatar {
  composes: column;
  padding-right: 10px;
}

.content {
  composes: column;
  flex: 1;
  overflow: hidden;
}

.name {
  white-space: nowrap;
  padding-bottom: 6px;
}

.lastMessage {
  color: #565656;
}

.activity {
```

```
.activity {
  composes: column;
  width: 60px;
  color: #aaa;
  font-weight: 300;
  font-size: 0.8em;
  align-items: flex-end;
}
```

Now that our user is styled with placeholders, it's time to add Moment.js and add the last activity to the profile.

Latest activity

Now we need to install moment. Then we just need to use time from now to display the proper value in our sidebar.

```
$ npm install --save moment
```

```
/app/components/Sidebar/index.js
commit: coming soon
```

```
import moment from "moment"
...
```

Deploying the API to Heroku

- Phoenix Buildpacks
- Configuring the App
- Environment Variables
- Deploying

The performance of Phoenix/Elixir is most apparent when using Heroku. Popular sites like Hex.pm run on a single dyno resulting in big cost savings.

Before we proceed, you'll need to register for a Heroku account and install the Heroku toolbelt. The toolbelt is a command line interface for Heroku that allows us to easily deploy our code, read logs, or make environment changes.

Phoenix Buildpacks

Heroku relies on buildpacks to configure a dyno for a specific language and/or framework. For our Phoenix application we'll add a buildpack that will install Erlang, Elixir, and our application dependencies.

In one fell swoop we can create our new Heroku application and add our buildpack, let's run the Heroku create command in our project directory:

```
$ heroku create --buildpack "https://github.com/HashNuke/heroku-buildpack-elixir.git"
Creating enigmatic-ravine-4201... done, stack is cedar-14
Buildpack set. Next release on enigmatic-ravine-4201 will use https://github.com/HashNuke/h
https://enigmatic-ravine-4201.herokuapp.com/ | https://git.heroku.com/enigmatic-ravine-4201
Git remote heroku added
```

Or if you've already signed in and created a Heroku project, you can add it as a git remote and assign a buildpack.

```
$ heroku buildpacks:add "https://github.com/HashNuke/heroku-buildpack-elixir.git"
```

It's probably worthwhile to look through the brief documentation for this particular buildpack in case you need to do some debugging.

Sometimes Heroku will not automatically add the `git remote`. You can test this by running the

command `git remote -v` and you should see two for `origin` that go to Github and two for `heroku`, which go to Heroku. In the event that it does not, log in to Heroku.com, select the project, click `Settings`, and copy the `Git URL` from the `Info` section and run the command (with the proper URL):

```
$ git remote add heroku <git_url>
```

The create command tells us it created `enigmatic-ravine-4201`, the name of our application within Heroku, set our buildpack, and finally added a remote to git. The outputted URL is the URL for our application, right now it only shows the Heroku welcome page.

> *Note*: Our Phoenix application doesn't rely on static assets but if it did, we'd need to add > another buildpack to handle asset compliation:

```
$ heroku buildpacks:add https://github.com/gjaldon/heroku-buildpack-phoenix-static.git
```

Configuring the App

With the application created and buildpack set, we need to make some final changes to Phoenix before we can deploy to Heroku.

For the purposes of security, we don't want to store our secret keys in a file on Heroku. Instead, we'll use environment variables. Be sure to use the `host:` that relates to your Heroku project. Let's open `config/prod.exs` and add the `secret_key_base` key and retrieve the value from the system.

We're also setting `check_origin` to false because we want to allow all of our anonymous chat clients to be able to connect no matter what their domain.

```
/config/prod.exs
commit: coming soon
```

```
config :phoenix_chat, PhoenixChat.Endpoint,
  http: [port: {:system, "PORT"}],
  check_origin: false,
  force_ssl: [rewrite_on: [:x_forwarded_proto]],
  secret_key_base: System.get_env("SECRET_KEY_BASE")
```

Since we won't be storing our secret keys in `config/prod.secret.exs`, remove the following line from our `config/prod.exs`:

```
import_config "prod.secret.exs"
```

Lastly, we need to update our Repo configure to rely on the `"DATABASE_URL"` environment variable Heroku sets:

```
/config/prod.exs
commit: coming soon
```

```
config :phoenix_chat, PhoenixChat.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: System.get_env("DATABASE_URL"),
  pool_size: 10,
  ssl: true
```

Environment Variables

To populate our `"DATABASE_URL"` environment variable, we need to use a Heroku add-on database. For our application the Heroku Postgres hobby tier will be sufficient (and free). Using the Heroku toolbelt, let's add it to our project:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

We will also need to create an `elixir_buildpack.config` file that will tell Elixir about our environment variables in Heroku.

```
$ touch elixir_buildpack.config
```

Within that file, we need to add our `DATABASE_URL` that you can find when you log into Heroku. This one will actually be added for you automatically, but we will need to do this in the future with Mailgun and other environment variables, so it's best to get used to doing it now.

```
/elixir_buildpack.config
commit: coming soon
```

```
config_vars_to_export=(
  DATABASE_URL
  SECRET_KEY_BASE
)
```

Finally, we need to generate and set a new secret key. With mix, let's generate a new secret:

```
$ mix phoenix.gen.secret
NrRwQpjBoUeKz1JnvWT+WrmF+hSnRPvkeDkHKwIgv5blKVeS0PUp9GA69S6VGXVf
```

Now we can add our new secret key to Heroku:

```
$ heroku config:set SECRET_KEY_BASE="NrRwQpjBoUeKz1JnvWT+WrmF+hSnRPvkeDkHKwIgv5blKVeS0PUp9G
```

If you intend to send emails, you'll also need to set up a Mailgun account and add those keys to your configuration as well. Otherwise, you should comment out the mail portion of your app otherwise it will cause an error on deployment.

It might also be a good idea to set `always_rebuild` to true since this will save you from some hard-to-debug errors later on.

```
/elixir_buildpack.config
commit: coming soon
```

```
config_vars_to_export=(
  DATABASE_URL
  SECRET_KEY_BASE
  MAILGUN_API_KEY
  MAILGUN_DOMAIN
)
always_rebuild=true
elixir_version=1.3.2
```

```
$ heroku config:set MAILGUN_API_KEY=key-cec83902409402940958398a2cf3
$ heroku config:set MAILGUN_DOMAIN=https://api.mailgun.net/v3/mg.domainname.io
```

You're also going to need a `Procfile` that tells your server how to run.

```
$ echo "web: MIX_ENV=prod mix phoenix.server" > Procfile
```

Deploying

We've created a Heroku project, added our buildpack, configured our application, and setup the environment, all that's left is for us to commit our changes and deploy them to Heroku:

```
$ git add .
$ git commit -am "chore: adjust configuration for deploy"
```

Deploying to Heroku is as easy as pushing your code. Note that Heroku ignores all branches other than master, so if you want to push a different branch, you need to specify. So if you wanted to push the `develop` branch, you would have to run: `git push heroku develop:master`

```
$ git push heroku master

Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 288 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Compressing source files... done.
...
...
...
remote: Verifying deploy.... done.
To https://git.heroku.com/enigmatic-ravine-4201.git
   f7ac6d3..e08a63e  test -> master
```

Before we visit our site, let's run our migrations on Heroku. We need to set the mix environment to `prod`, as it defaults to `dev`.

```
$ heroku run MIX_ENV=prod mix ecto.migrate
```

Now we're ready to see our app in action, typing `heroku open` in the console will open our browser to our app:

```
$ heroku open
```

And now we've successfully deployed our Phoenix application to Heroku. Take note of the URL of your API because we will need it later.

Deploying the Frontend to S3

- Webpack production optimizations
- AWS S3

Now that we have an app that sort-of functions on our local environment, we should start looking into deployment options. You may have noticed that our app doesn't have much in the way of server configuration in our `/server.js` file. That's no coincidence; our app doesn't run on a server.

In fact, if you compile your app using webpack with the `webpack` command and then open the `index.html` file within `/dist`, you'll see that you have a fully functional app just from static assets. That `index.html` file has all the assets it needs to run an app without a server because everything is static and all of your data comes from an API call to our backend.

Given that everything is static, it's super easy to host our site. We don't even have to spin up a server to do it; we can simply drop in the files and serve them when someone hits your url. We're going to host these files on Amazon Web Service's (AWS) Simple Storage Service (S3). We will go into detail on these later.

Change API_HOST

Now we have a functional app. The last thing we need to do in preparation for deployment is change out our hardcoded `localhost:4000` values.

To do this, we need to change our `webpack.config.js` to include a `webpack.DefinePlugin`.

```
...
module: {
  ...
},
plugins: [
  new webpack.DefinePlugin({
    "process.env": {
      API_HOST: JSON.stringify('http://localhost:4000'),
      SOCKET_HOST: JSON.stringify('ws://localhost:4000/socket')
    }
  })
],
...
```

Now all we have to do is change out every instance of `http://localhost:4000` for `process.env.API_HOST`. We can use ES6 string interpolation (which uses a `${}`) to insert a string into an existing string, so your new paths would look something like this: `${API_HOST}/api/users`. Also change out the `ws://localhost:4000/socket` for `process.env.SOCKET_HOST`.

You'll probably have to run this manually with a find and replace. Keep in mind that string interpolation only works with back ticks (``), not quotes (""). As always, when you change your webpack configuration, you need to restart your server.

Starting with AWS

One of the reasons we're using S3 rather than a Node server is that S3 is *insanely cheap*. It's something like 5 cents per gigabyte per month for storage and another 5 cents per gigabyte for data transfer. In other words, you would have to have an insane amount of traffic for the cost to even become noticeable.

Not only that, but this type of architecture can instantly scale to a seemingly-infinite amount of traffic (if you're at a point at which AWS cannot handle your level of traffic, then you shouldn't be reading this tutorial series).

Go to aws.amazon.com and create an account. You'll need to add your credit card as well, but this is going to cost little-to-nothing, so relax.

Then go to your console where you can see all of the tools that AWS has to offer. There are lots and lots of them, and they all have vague and confusing names. All we're going to use for the time being is S3, so go ahead and click on `S3`.

On the left you'll see a list of your `buckets`. Think of a bucket as a figurative bucket that holds your files. You can dump all kinds of data in a bucket. We're going to create a new bucket that will hold all of our static files for our frontend app.

Click on the "Create Bucket" button and create a bucket with a name that is unique across all AWS. A good strategy for naming your domain is to give it the name of your domain, since by definition your domain will be unique. Since we're going to host this app at `www.phoenixchat.io` we can give it that bucket name. Since yours will have to be unique, you'll have to come up with something different, but you get the idea. Go ahead and click `Create`.

Now you'll see your bucket on the left. On the right you'll see an option for `Static Website Hosting`. Go ahead and click on that dropdown and select `Enable website hosting`. Where it says `Index Document` go ahead and type in `index.html` and click `Save`.

The next part is a little bit less obvious. We need to set up `Permissions` that determine who can access the bucket and what can be done with it. Under `Permissions`, click on `Add bucket policy`, after which point you'll see a blank textarea. This text area expects your configuration options in JSON and Amazon gives us a tool to automatically generate this code. On the bottom left of this modal, you should see `AWS`

`Policy Generator`. Click on that.

Change `Select Type of Policy` to `S3 Bucket Policy`.

Change `Principal` to `*`, which means "everyone".

For `Actions` go ahead and check the box for `GetObject`.

The `Amazon Resource Name (ARN)` follows a pattern that is shown below the text input. It's something like `arn:aws:s3:::<bucket_name>/<key_name>`. If your bucket is named `www.phoenixchat.io` and you want it to apply to everyone your configuration will look like: `arn:aws:s3:::www.phoenixchat.io/*`

Then click `Add Statement`, which will show your configuration options in a table. Check them over and make sure they're right and then click `Generate Policy`.

Copy the JSON that was generated and paste that into the other tab with the empty text area and click `Save`.

AWS Identity and Access Management (IAM)

Now we want to send our files to S3. But before we can do that, we need to give ourselves access to S3 through the command line tool (you could upload your files through the website, but it's clunky and can't be done programmatically).

Go back to the AWS console and click on `Identity and Access Management`.

From here, click on `Groups` on the left and create a new group. `Administrators` is a perfectly valid group name, but you can enter any name you like and click `Next Step`.

Then add the policy `AmazonS3FullAccess` which you can find by filtering down the options. Then click `Next Step` to review, and if everything looks right click `Create Group`.

Now that we have a group, we need to create a user to add to that group. On the left side, click `Users`, then `Create New User`. Enter a username for that user. In our case, we're going to enter the username `phoenixchat`, but it's common to use your first and last name as the username.

Make sure the `Generate an access key for each user` is checked, then create the user. You'll see an option to `Download credentials`. Go ahead and do that now because we will need those credentials later.

Now click on the username from the list (not the checkbox), go to the `Groups` tab and `Add User to Groups`. Select the `Administrators` group and click `Add to groups`.

Finally, under the `Security Credentials` tab under `Sign-In Credentials`, choose `Manage Password`. Go ahead and fill out the `Assign a custom password` option and click `Apply`.

We're now ready to send our app to S3... but not quite. We need to change around our webpack configuration a little bit to prepare our app for production.

Configuring webpack for production

We need webpack to build our app in production mode. This means a few things need to change.

1. We need to uglify and minify our code to make it smaller
2. We need to dedupe our code to remove duplicate information
3. Export our environment variables
4. (Potentially) load React and other libraries as externals

The first takes our code and removes whitespace and any extraneous data to make our bundle as small as possible. This simple step often cuts your bundle size by more than 1/3.

The second is deduplication (dedupe), which looks for files that are duplicates and removes the copy. This doesn't often noticeably affect the bundle size, but it doesn't cost anything to add.

The third is adding our environment variables. In this case, we only have one, which as you will see later tells webpack to use our Heroku app rather than `localhost:4000`. In the future, this is where you would inject things like Stripe tokens. But keep in mind, your environment variables are not hidden from the user. You shouldn't have anything secret living in your frontend.

The last piece of optimization is to load some of our libraries as externals. It's possible that our user will already have common libraries, such as React, cached in their browser. If that's the case, then we don't need to fetch that data again. This is not strictly necessary, and we aren't going to bother with it at this point.

In the future, we will look into more advanced optimizations, such as code splitting and the use of an `entry-chunk` to reduce our initial bundle size as much as possible. For more on webpack optimization, check out this page.

Before we can proceed, with our build, we're going to need to set up some environment variables. You can use the code below (with the proper information instead of the placeholders within `< >`) to export your variables to the environment. You can also keep them in a `.sh` file such as `.env.sh` and run run it with `source .env.sh` to export all of the variables at the same time.

If you choose to keep these keys in a file, be sure to add that file to your `.gitignore` so you don't accidentally expose your keys to the world.

```
export PROFILE=<profile name>
export AWS_ACCESS_KEY_ID=<access key>
export AWS_SECRET_ACCESS_KEY=<secret key>
export BUCKET=<bucket name>
```

Go ahead and run this now so we have access to these variables.

Now let's create a special webpack configuration just for production. We'll call it `webpack.prod.config.js`.

There are many ways to go about setting up your webpack configuration for production. Since the configuration is just an object, you can run it through a series of functions or conditionals to determine what you're exporting (often based on the currently chosen environment).

It's also common to make a "shared" configuration that all apps will use and then build on that as you add more environments and platforms. Keep in mind that this configuration is just JavaScript, so you can manipulate the configuration object just like you could any other JavaScript object.

That said, we're just going to keep it simple and put this in a new file for now and optimize later if it becomes necessary.

```
$ touch webpack.prod.config.js
```

We're going to need the `extract-text-webpack-plugin`, which combines all of our CSS into a separate CSS file, which prevents React from inlining your styles into the JavaScript and bundles them all into a single file that gets loaded in parallel with your JavaScript bundle. This is something you want to do with `react-css-modules` in production.

```
$ npm install --save-dev extract-text-webpack-plugin
```

Within your `webpack.prod.config.js` file, add the following configuration. It will be almost the same as our existing code with a few differences, explained below the codeblock.

```
var path = require('path')
var webpack = require('webpack')
var ExtractTextPlugin = require('extract-text-webpack-plugin')
var cssnext = require('postcss-cssnext')
var HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  devtool: 'cheap-module-source-map',
  entry: [
    "whatwg-fetch",
    './app/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  plugins: [
```

```
    new ExtractTextPlugin('style.css', {
      allChunks: true
    }),
    new webpack.optimize.DedupePlugin(),
    new webpack.optimize.UglifyJsPlugin({
      minimize: true,
      compress: {
        warnings: false
      }
    }),
    new webpack.DefinePlugin({
      API_HOST: JSON.stringify("https://phoenix-chat-api.herokuapp.com"),
      SOCKET_HOST: JSON.stringify("wss://phoenix-chat-api.herokuapp.com"),
      "process.env": {
        NODE_ENV: JSON.stringify('production')
      }
    }),
    new HtmlWebpackPlugin({
      template: "index.html",
      hash: true,
      filename: "index.html"
    })
  ],
  module: {
    loaders: [
      {
        test: /\.js$/,
        loaders: ['babel'],
        exclude: /node_modules/,
        include: path.join(__dirname, 'app')
      },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract('style', 'css?modules&importLoaders=1&localIdentN
        include: path.join(__dirname, 'app')
      }
    ]
  },
  postcss: function () {
    return [cssnext]
  },
  resolve: {
    extensions: [ '', '.js' ]
  }
}
```

The first thing we changed is our `devtool`, which we changed from `eval` to `cheap-module-source-map`. This change along will save you about 80% on your bundle size. If you aren't familiar with what source maps are, you can think of it as how your computer lets you know how errors that occur in compiled code

correspond to your un-compiled, human-readable code. So with a good source map, rather than getting an error like "error bundle.js line 1", you would get, "error app/components/Chat/index.js line 42".

You'll notice that our `entry` point no longer includes webpack dev server, because we don't need it. If you're worried about cross-browser compatibility, you can also npm install `babel-polyfill` and add that as entry point as well. That will add some weight to your code, but it'll make sure your app works with older browsers.

We've also added 5 plugins. The first is the extract text plugin, which pulls out your inline styles and adds them to a single CSS file at the top of app. The next two are simply optimizations. They are not necessary for a successful build, but they make your bundle smaller and more efficient.

The last plugin is the `HtmlWebpackPlugin`, which is not an optimization, but it will inject our `index.html` file into our production distribution. You could simply copy this over or use the `file-loader` if you wanted, but we want to use this plugin to hash our bundle names to solve any cache invalidation issues with S3. When you compile a production build, check your `index.html` file and you'll see that your bundle and style files have a `?` and a random series of numbers and letters after it. We will explain why this is important in a later lesson when we go over CloudFront.

And now that we're injecting our HTML file into our app, we should remove the tags we have hardcoded in our `index.html` file:

```
<!-- Delete this -->
<link rel="stylesheet" href="style.css">

<!-- Delete this too -->
<script src="bundle.js"></script>
```

But in doing this, we've broken our dev build, so you need to add the `HtmlWebpackPlugin` to your `webpack.config.js` file as well.

```
...
var HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  ...

  new HtmlWebpackPlugin({
    template: "index.html",
    hash: true,
    filename: "index.html"
  })
  ...
}
```

We will also need to install that plugin.

```
$ npm install html-webpack-plugin --save-dev
```

Now go ahead and run webpack with the `webpack.prod.config.js` to build our app using the production configuration.

```
$ webpack --config webpack.prod.config.js
```

Check the `/dist` directory to make sure everything is in order. If you open the `dist/index.html` file in a browser, you should have a fully functional app. If this works as you would expect, then we're ready to deploy.

For now, we'll hack this together. In the next lesson, we'll go over continuous integration and automated deployments.

Hosting on S3

Go back to AWS and click on that bucket we made earlier (in our case, `www.phoenixchat.io`). Click `Upload` in the top left and add the `index.html`, `bundle.js`, and `style.css` files currently in `/dist`.

Once those are uploaded, click `Properties` on the top right and select `Static Website Hosting`.

Change the setting to `Enable website hosting` and set the `Index document` to `index.html`. When you save, you should see an endpoint just above the tabs with a long domain name ending in `...amazonaws.com`.

Click on the link. Your site is now live.

Continuous Integration and Deployment

- Automated deployment
- Continuous integration with CircleCI
- Continuous deployment

We now have a (mostly) functional app that has been uploaded to S3 and is live for the world to see. But opening up AWS and dragging files after a manual build is tedious and prone to error. To remedy this situation, we're going to write an automated deployment script that will run a series of commands to send our newly created site to S3.

Once we have our automated deployment set up, we'll create an account with CircleCI for continuous integration. If you're not familiar with continuous integration, it's a service that will automatically build your app, run your tests, and if everything seems to work, it will automatically deploy the current version.

Continuous integration is extremely useful for large organizations that have many developers pushing code to the same codebase at the same time while in production. That said, it requires rigorous testing or you might find yourself in a position where you deployed an update that broke something.

Automated deployment

We're first going to set up automated deployment on our local environment, then we'll worry about continuous integration. We're going to need to install the AWS command line tool.

```
$ brew install awscli
```

Then we need to configure our AWS credentials so we can deploy. You can do this one piece at a time, but I recommend writing a shell script so you can execute these the same way every time. Let's go ahead and create a new file called `deploy.sh`.

```
$ touch deploy.sh
```

Within this file, we should add all of the necessary commands to deploy to S3. Remember from a previous lesson that we need to `export` all of our environment variables so we can access them later. This is what the `source .env.sh` command was for.

```
echo "Version number..."
aws --version

echo "Configuring access id..."
aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID

echo "Configuring secret key..."
aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY

echo "Syncing with $BUCKET"
aws s3 sync ./dist s3://$BUCKET
```

If you do not know your bucket location, you can run the following to find out your default region.

```
$ aws s3api get-bucket-location --bucket $BUCKET
```

We're also including `echo` commands so we can see in our terminal what step in the process our configuration has reached. This is basically just a `console.log` to let us know if and where we run into issues.

Now we should be ready to deploy. Make sure your terminal is in the root of `phoenix-chat-frontend` and simply run the commands in our `deploy.sh` script.

```
$ sh deploy.sh
```

And like magic, all of your files are uploaded. If you want to run a test to make sure your `deploy.sh` script is working, try changing the background to `red` or something obvious and re-deploy. You should see the changes immediately propagate when you refresh the page.

Continuous integration

The next step is to set up continuous integration. Go ahead and sign up with CircleCI and link your Github account. If you have a preference for another continuous integration company, feel free to use them--they're mostly all the same, but the files will be different.

Go ahead and click on `Add project` from the menu on the left, then **add those environment variables we have stored in our `.env.sh` file** and `API_HOST` and `SOCKET_HOST` as environment variables to CircleCI.

Create a `circle.yml` file at the root of your project. This is required for CircleCI and it gives a series of instructions for CircleCI to execute in order to properly run your code.

```
$ touch circle.yml
```

Then add the following code to configure CircleCI. We will go over each line, but there are many more configuration options available. It might be worth your time to skim through the configuration docs to get a sense of how much you can do with CircleCI and the various continuous integration environments.

```
machine:
  node:
    version: 6.3.0 # React CSS Loader does not work with 0.X versions of node

dependencies:
  override:
    - npm install
    - npm install -g webpack # Need access to webpack cli to build the project
    - sudo pip install awscli # Need access to AWS CLI to deploy the project

deployment:
  aws:
    branch: master
    commands:
      - chmod +x deploy.sh
      - webpack --config webpack.prod.config.js
      - ./deploy.sh
```

The first line sets the version of Node to something we want to use. This is not strictly necessary, but it's good to know what version you're using rather than just accepting the default.

Next we set our dependencies. In this case, we are running `npm install` (which it will run by default anyway, but we're being safe), and two additional installations for the `webpack` command line tool that will allow CircleCI to compile our code and `awscli` which is the AWS command line tool we need to run our `deploy.sh` script.

Finally, we have our deployment to AWS. CircleCI has a special configuration for AWS since so many people use it. We are telling CircleCI to only use these commands for our `master` branch. If you wanted, you could set up special deploy configurations for other branches. It's not uncommon to deploy to `beta` or `develop` to make sure the site works on other branches as you go along.

The only tricky thing here is the `chmod +x deploy.sh`, which changes the permissions of our container to allow us to execute the deploy script. `chmod` is one of those things that is rarely covered and used often. If you ever run into errors related to permissions, it's probably an issue with `chmod`.

Save this file, git add, git commit, and git push, and you should see CircleCI create a container, build the project, run the tests, then deploy to AWS. Pretty cool!

SSL and CloudFront

- Getting an SSL certificate
- Setting up CloudFront
- Cache Invalidation
- Domain names with Route 53

While Heroku automatically gives us an encrypted connection to our API via `HTTPS` (which is just a normal `HTTP` connection plus an `SSL` security layer--people will use the two terms interchangeably), S3 does not. That's because there's nothing sitting between the user and the files.

The only way to solve this problem is to 1) set up a Node server to handle requests and run this on EC2, or 2) run our requests through a proxy like CloudFront. Since we're going to need to communicate to our backend by a secure route, we're going to need to set up CloudFront and route our connections through it.

We will also need an SSL certificate. This used to be a long, tedious, and painful process that often cost hundreds of dollars a year (Godaddy still charges $299/yr for it), but thanks to AWS, you get it for free and it's super easy to set up!

Getting an SSL certificate

As mentioned above, this used to be complicated and expensive, but now it's really easy. Log in to `aws.amazon.com` and open up the `Certificate Manager`. Click the `Request a certificate` button in the top left.

Then you want to request what is called a `wildcard` certificate and a naked domain certificate. That gives you the ability to protect all the single-level sub-domains of a particular site, so it would cover `www.phoenixchat.io`, `api.phoenixchat.io`, etc. Assuming the domain is `phoenixchat.io`, the requested domains would be `*.phoenixchat.io` and `phoenixchat.io`. Click `Review and request` and `Confirm and request`.

It often takes a few minutes to validate, but that's it. You're done.

Setting up CloudFront

CloudFront is something you can put in front of your app to make load times faster and enable HTTPS. It does aggressive caching in certain locations around the world so your app loads faster since your app

won't have to request assets every time the page loads.

Within `CloudFront`, click on `Create Distribution` followed by `Create` under the `Web` section. From `Origin Domain Name` you can select the S3 bucket you just made from the dropdown. Make sure you add domain you would like to reference in the `Alternate Domain Names` section (e.g. phoenixchat.io). Change `Viewer Protocol Policy` to `Redirect HTTP to HTTPS` and add `index.html` as the `Default Root Object`. Then under `SSL Certificate`, choose `Custom SSL Certificate` and select the certificate you just created. Then create the distribution.

This will take a while to provision (often minutes to hours). While we're waiting, we can explain how cache invalidation works.

Cache Invalidation

Cache invalidation is a big subject and an especially complicated one. Fortunately for us, our use case is pretty simple. Because CloudFront caches files based on their names, if we make a change to `bundle.js`, CloudFront doesn't know that it needs to clear the cache and bring in the new version of the app. The act of clearing this cache is called an `invalidation`.

Since we're injecting our HTML file with hashed bundle and style files, all we have to do is tell CloudFront not to cache our `index.html` file, since every time we update the app, our bundle and style files will have a new hashed suffix.

The way to tell CloudFront to never cache the `index.html` file is by setting a new behavior. Select the distribution by clicking on the ID. Then go to the `Behaviors` and create a new behavior. Add `index.html` to the `Path Pattern`. Then change `Object Caching` to `Customize`, and set the `Maximum` and `Default` TTL (Time To Live--which is how long the object is cached) to `0`.

Domain names with Route 53

If you've purchased your domain from another provider, your setup is going to be different. Assuming you used Amazon's `Route 53`, this is how you would point your domain to the CloudFront distribution you just created.

After purchasing a domain, go to `Hosted zones` in the left panel and select `Create hosted zone` for the domain you would like to change.

Now choose `Create record set`. Then in the panel on the right, toggle the `Alias` to `Yes` and select the CloudFront distribution you just created. Keep in mind that the distribution might not be provisioned yet, in which case you'll have to wait (if you go to CloudFront and it says "in progress", it isn't done). Then click `Create` and you're done. Now if you go to the domain you purchased (e.g. https://phoenixchat.io) it should look like you expect with HTTPS enabled.

Keep in mind, these changes can often take hours to propagate. If your changes don't immediately appear, don't automatically assume there is an error. Unfortunately there is no way around this and it makes your code really difficult to debug.

ESLint and Airbnb style guide

- What is ESlint
- Install ESLint
- Examples of rule overrides

When you're working with a big team or on open source projects, there will come a point where the various styles of each person will begin to collide. Some people insist on semicolons, some despise them; some like 2 spaces for indentation, some insist on 4. At a certain point, your codebase begins to look rather inconsistent. Wouldn't it be nice if you could set rules for your codebase that everyone needed to follow so your format was totally uniform throughout your app? Enter ESLint.

ESLint allows you to set formatting rules and will throw errors/warnings if someone attempts to violate those rules. And if you're using an IDE like Atom, you can use a linter in realtime, which will let you know if the code you just wrote is out of format every time you save. If you're using Atom, you can install ESLint by using the Atom package manager, apm (below).

```
$ apm install linter-eslint
```

Install ESLint

The most commonly used starting point for ESLint is the Airbnb style guide. Almost everyone uses this as the default options when building a React application.

```
$ npm install --save-dev eslint-config-airbnb eslint \
  eslint-plugin-jsx-a11y eslint-plugin-import \
  eslint-plugin-react babel-eslint
```

Once those are installed, you'll need a `.eslintrc.js` file, in which you specify options for ESLint (you will often see this as simply `.eslintrc`, but this is deprecated in favor of explicit file extensions like `.js`). These options include plugins, globals, and rule overrides.

```
$ touch .eslintrc.js
```

Within this file, we will extend the existing `airbnb` style guide and add the plugins that enable us to use React and jsx. We also define our `globals` here. ESLint will throw an error if you try to use a function or variable that is not declared in the file, so global namespaces like `document` or `window` will not work. In order to get around this, you have to declare all your global variables within your configuration so ESLint knows what to expect.

From there, we define a few rules. These rules are necessarily a matter of opinion. We have included some pretty common overrides and we will go over each of them (and the reason for using them) below the codeblock.

```
module.exports = {
  "extends": "airbnb",
  "parser": "babel-eslint",
  "plugins": [
    "react",
    "jsx-a11y",
    "import"
  ],
  "globals": {
    "window": true,
    "document": true,
    "fetch": true,
    "localStorage": true
  },
  "rules": {
    "comma-dangle": [1, "never"],
    "semi": [2, "never"],
    "arrow-body-style": 0,
    "quotes": ["error", "double"],
    'react/jsx-closing-bracket-location': [1, 'after-props'],
    'no-param-reassign': ["error", { "props": false }],
    "react/jsx-filename-extension": [1, { "extensions": [".js"] }]
  }
}
```

Rules

Every rule in ESLint has documentation--pretty darn good documentation, actually. If you want to learn more about a particular rule, you can always find it by using the search bar at the top of the ESLint docs.

Within the configuration of your rules, you can set a rule to `0`, which means "off", `1`, which means "warning", or `2`, which means error and will throw a exit code of 1. Many of these rules will have additional configurations, which are specified in the documentation associated with each rule.

The first rule we're going to override is the `comma-dangle` (docs). Some people like comma dangles

because it can save you from errors caused by a lack of a comma when adding or removing items from lists or objects. We're not going to use them because comma dangles don't work with JSON. But we're just going to turn this into a warning rather than an error, so our code will still pass even if we have a few comma dangles here and there.

The second rule is `semi`, which requires semicolons. We're setting the configuration to require that semicolons are not used or it will throw an error. This is because semicolons provide nothing and merely serve to clutter your JavaScript code.

The third rule is `arrow-body-style`, which controls how you use curly braces around your ES2015 arrow functions. We're just going to turn this off because there is too much variation in how these functions can be written in React.

The fourth rule is requiring that all quotes are double quotes. For some reason, this is always controversial, but we're using double quotes because you're more likely to include a single `'` within a string than you are a double quote and because Elixir and JSON use double quotes for strings.

The fifth rule is specific to React, where we determine how the closing bracket should be formed. Some people like the extra `>` on a new line, others do not. We're going to add this extra character at the end of our element properties, like we've been doing so far in the React lessons. For more information on this rule, check the (docs).

The sixth rule, `no-param-reassign`, is somewhat obscure, but it makes it so that you cannot reassign variables that are passed in as parameters (docs). We're overriding part of the rule so that we still cannot reassign the variable, but we can assign its properties. For example, we cannot reassign `foo` if it's passed in from `myFunction(foo)`, but we can assign `foo.bar`.

The last rule we're overriding is the filename extension. We're just going to name all of our files `.js` rather than specifying `.jsx` or `.es6` or anything like that. There aren't any real advantages to including all the extra filetypes, so we're just going to make everything `.js`.

Additional

You'll also want to create a `.eslintignore` file so you can ignore files that you don't need to lint.

```
$ touch .eslintignore
```

We're going to add all the generated files, as well as our webpack configuration, our `server.js` file, and all of our tests.

```
.nyc_output
dist
coverage
node_modules
webpack.*
server.js
**/spec.js
```

The reason we're ignoring these files is because they'll each require a litany of ESLint overrides. For example, you can include specific ESLint overrides in a comment on the top of any file. Just to get our `server.js` file to pass, we'll need to add the following 3 overrides.

```
/* eslint-disable import/no-extraneous-dependencies */
/* eslint-disable no-console */
/* eslint-disable no-unused-expressions */
```

And to get each of our `spec.js` files to pass, we'd have to add even more, and we'de have to add them to every spec file. So unless you want to do that (which you're more than welcome to do--writing an alias would probably speed up the process), it's just easier to skip them.

Refactoring Redux

- Separating concerns in Redux
- Best practices

In this section, we're going to change all of our Redux actions into actions that are more performant, testable, and easier to read. You probably noticed that our Actions were starting to get a little out of control. Now it's time we refactor them.

Incremental refactoring

As far as refactors go, this will be pretty easy so we're going to do this a little differently than previous lessons. Since all of our actions are more or less the same, we're just going to show you how to refactor one of the actions and you should try refactoring the others using the same strategy. If you get stuck, send us a message on Slack and we'll walk you through it.

Let's take a look at our `userAuth` action:

```
Actions.userAuth = function userAuth() {
  return dispatch => fetch(`${process.env.API_HOST}/auth/me`, {
    method: "GET",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json",
      Authorization: `Bearer ${localStorage.token}` || ""
    }
  })
  .then((res) => {
    return res.json()
  })
  .then((res) => {
    dispatch({
      type: "USER_AUTH",
      payload: {
        user: res.data
      }
    })
  })
  .catch((err) => {
    console.warn(err)
  })
}
```

We should also move our actions into new directory and into an `index.js` file. It will make sense why we are doing this later on.

```
$ mkdir app/redux/actions
$ touch app/redux/actions/{index,async}.js
```

Then copy over the existing content from `actions.js` into `async.js` and delete `actions.js`. So now your Redux directory/file structure should look like this, with all of your actions living in `async.js`:

```
redux
|-- reducers.js
|-- store.js
|-- actions
    |-- async.js
    |-- index.js
```

The next thing we should do is change our exports so that we're exporting each action individually. So let's get rid of the `Actions` object and simply use `export function`. We'll also need to get rid of that `default export` at the end. So now, each of your functions should look like this (with the body of the

actions removed for brevity):

/app/redux/actions/async.js
commit: coming soon

```
export function userAuth() {
  ...
}

export function userNew(user) {
  ...
}

export function userLogin(user) {
  ...
}

export function organizationNew(organization) {
  ...
}
```

But now we aren't exporting our actions, so nothing will work. In order to export all of our exported actions, let's add an `export` in our `index.js` file that takes in all the actions from `async.js`.

/app/redux/actions/index.js
commit: coming soon

```
export * from "./async"
```

And now we're ready for major refactoring.

Remove Try-Catch

The first thing we should do is remove the `catch` statement at the end and handle errors within `then`. This is debatable, but using `try-catch` for logic can end up displaying errors that aren't what you would expect. For example, if you had a spelling error or something like that within your `then`, it would raise in your `catch` statement, making bugs harder to track down. Check out this comment from Dan Abramov for another example.

/app/redux/actions/async.js
commit: coming soon

```
Actions.userAuth = function userAuth() {
  return dispatch => fetch(`${process.env.API_HOST}/auth/me`, {
    method: "GET",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json",
      Authorization: `Bearer ${localStorage.token}` || ""
    }
  })
  .then((res) => {
    return res.json()
  })
  .then((res) => {
    dispatch({
      type: "USER_AUTH",
      payload: {
        user: res.data
      }
    })
  })
}
```

Pulling out HTTP calls

The next part of our refactor involves pulling our HTTP calls into a separate file. Let's create an `http.js` file within our `actions` directory and add some logic there.

```
$ touch app/redux/actions/http.js
```

Again, we're only going to change `userAuth` in this lesson, but you should refactor the other actions as well.

```
/app/redux/actions/http.js
commit: coming soon
```

```
export function userAuth() {
  return fetch(`${process.env.API_HOST}/auth/me`, {
    method: "GET",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json",
      Authorization: `Bearer ${localStorage.token}` || ""
    }
  })
}
```

So now that we have our HTTP call pulled out, we can use it in our `async.js` file.

/app/redux/actions/async.js
commit: coming soon

```
import * as http from "./http"

export function userAuth() {
  return dispatch => {
    http.userAuth()
    .then((res) => {
      return res.json()
    })
    .then((res) => {
      dispatch({
        type: "USER_AUTH",
        payload: {
          user: res.data
        }
      })
    })
  }
}
```

This will allow us to write tests specific to our HTTP calls and different tests specific to our asynchronous actions.

Async-Await

The next thing to do is to get rid of the `then` block by using the ES2016 Async-Await which was added in July of 2016. If you're not familiar, it's similar to a generator.

We're also going to add a failure action (`USER_AUTH_FAILURE`) in case we want to do something in the

event of auth failure.

/app/redux/actions/async.js
commit: coming soon

```
import * as http from "./http"

export function userAuth() {
  return async dispatch => {
    const response = await http.userAuth()
    if (response.status !== 200) {
      dispatch({ type: "USER_AUTH_FAILURE" })
    } else {
      const result = await response.json()
      dispatch({
        type: "USER_AUTH",
        payload: {
          user: result.data
        }
      })
    }
  }
}
```

If you try to run this, you will get an error. That's because we need `babel-polyfill` to handle `async-await` until it's widely supported.

```
$ npm install --save-dev babel-polyfill
```

Then add it as an entry point in our webpack configuration just like we did with `whatwg-fetch`. You'll have to do this in both `webpack.config.js` and `webpack.prod.config.js`.

/webpack.config.js
commit: coming soon

```
...
module.exports = {
  devtool: 'eval',
  entry: [
    'whatwg-fetch',
    'babel-polyfill',
    'webpack-dev-server/client?http://localhost:3000',
    './app/index'
  ],
  ...
}
```

Although it doesn't affect us here, something to keep in mind that is that every time you dispatch and change state, you're causing a re-render. So if you look at our `userNew` function, you'll see that we're dispatching the user data to our reducer, then calling `userAuth`. This will cause two renders, which you may or may not want to do depending on your situation.

Separating synchronous actions

The last major change we'll make is to pull out our synchronous actions. These are all of our actions that don't require a request to our server. So that includes changing modal states, loading states, and dispatching the results of an asynchronous request.

```
$ touch app/redux/actions/sync.js
```

/app/redux/actions/sync.js
commit: coming soon

```
export const userAuthSuccess = user => ({
  type: "USER_AUTH",
  payload: {
    user
  }
})

export const userAuthFailure = () => ({
  type: "USER_AUTH_FAILURE"
})
```

/app/redux/actions/async.js
commit: coming soon

```
import * as http from "./http"
import * as sync from "./sync"

export function userAuth() {
  return async dispatch => {
    const response = await http.userAuth()
    if (response.status !== 200) {
      dispatch(sync.userAuthFailure())
    } else {
      const result = await response.json()
      dispatch(sync.userAuthSuccess(result.data))
    }
  }
}
```

We're also going to want to use our synchronous actions in some of our components, so we should add an export in our `index.js` file that gives us access to them.

/app/redux/actions/index.js
commit: coming soon

```
export * from "./sync"
export * from "./async"
```

The only other immediate change you should make for this refactor is in you `App` component. Instead of importing `Actions`, we need to import the individual action, `userAuth`. And then we have to change the dispatch.

```
import { userAuth } from "../../redux/actions"

export class App extends React.Component {
  componentDidMount() {
    this.props.dispatch(userAuth())
  }

  ...
}
```

And that should do it. Our Redux actions are now split into functions that are task-specific, which will make them much easier to test and reason about. When you refresh, your app should still function.

You will need to refactor out all of the `Actions` imports, since we are only exporting named exports now. So, for example, the `Login` component should now import `userLogin` like this:

```
// Old import
import Actions from "../../redux/actions"

// New import
import { userLogin } from "../../redux/actions"
```

Testing Redux

- Action tests
- Reducer tests

coming soon

Refactoring React

- mapDispatchToProps

coming soon

UI Component Development with React Storybook

- Installation
- Component isloation

What is it?

```
$ npm install --save-dev @kadira/storybook
```

```
"scripts": {
  "storybook": "start-storybook -p 9001"
}
```

https://github.com/kadirahq/react-storybook

Redux Optimization with ImmutableJS

Basics of analytics

- The goal of analytics
- What to track
- All of the options

If you've built a website any time since about 2005, you've probably used Google Analytics. You add a snippet to your page and you magically have access to pageviews, bounce rate, customer location, browser, device, and many other pieces of information. This is a good starting point, but there is so much more that can be done with analytics.

Two common tools are Mixpanel and Kissmetrics, which give you deeper insights into customer behavior and conversion rates. If you're a lazy programmer and you want to pawn even tracking off on non-technical people, Heap Analytics is a great option since it tracks all events and lets you define them on a graphical user interface after the fact.

Any tool you choose will be fine. They've all converged on some of the same basic functionality that we'll need.

The goal of analytics

Developers who are new to analytics often take the "analyze everything" approach, where they want to track every event and user interaction down to the last excruciating detail. While this approach has its advantages, it's usually better to take a step back and think about what you hope to accomplish with your analytics.

If you have an e-commerce site, your goal is to get as much revenue as possible. You will want to track the series of actions that users take that lead to a purchase.

If users that come from a particular source (Google ads, Facebook ads, organic traffic) have a particular affinity to certain products, then you need to know that. If users often get frustrated and leave because they have too many menus to click through, then you need to know that.

Everything you do will be focused on how to get customers to make purchases and turn the average visitor into a customer.

We are building a business-facing chat app, so our goals are different than the goals of an e-commerce site. There are really two things we want to track:

1. Conversion rate

2. Feature usage

We will go over each of these independently at a high level.

Conversion tracking

Conversion tracking lets you know how effective certain things are in your site at turning a potential customer into a paying customer. For example, it's worthwhile for a marketing team to know which of their ads are the most successful both in terms of allocating their budget and adjusting their messaging.

A typical way to track conversions is to keep track of a variety of `UTM` codes. `UTM` stands for "Urchin Tracking Module", which is from a company that Google bought and they decided to keep the name.

There are 5 universal UTM codes. They are:

`utm_source`: identifies the source of the traffic (for example, Google)

`utm_medium`: identifies a medium, such as email, newsletter, etc

`utm_term`: identifies the search term used to get the user to click on the ad.

`utm_content`: used for A/B testing to differentiate between two pages that link to the same url. This is not used very often.

`utm_campaign`: identifies a broad campaign that your ad is a part of. For example, "spring_sale" could be the overall campaign.

Within Mixpanel, you can use these UTM codes to track which ads are the most successful.

To achieve reasonable conversion tracking, we're going to need to track the original source of the traffic and follow that user through a series of events that lead us to the desired action.

So, an example funnel for conversion tracking would be:

1. A user gets to our site through a Google ad for the term "phoenix tutorial", which we know because of the tag `utm_source=google?utm_term=phoenix+tutorial` that was appended to the link and tracked by Mixpanel.
2. The user viewed four of the free lessons, which we know because we tracked that user across multiple page routes.
3. That user went back to the homepage.
4. The the user clicked the purchase button.
5. The user went through with the purchase.

At every level of this process there is something that could go wrong and therefore something to learn.

At step 1, we know that the keyword was able to convince someone to come to the site. But perhaps the keyword was not as specific as it should have been and the user was actually looking for something else. If the user quickly leaves, it's safe to assume we did not target this properly.

Alternatively, if we chose a keyword such as `phoenix tutorial free`, we might be attracting the wrong customers, which is to say, only people who do not want to pay for tutorials. There's something to be said for the marketing effect of free lessons, but it's something to keep in mind.

At step 2, we know that the user viewed several free lessons. This tells us that the free lessons are useful for keeping the user's attention. If this user eventually converts, it means that giving her a sample of what the lessons are all about helped turn her into a paying customer.

At step 3, we see that the user went back to the homepage. And why would she do that? Well, it might be because there's no purchase button in the view she was in before. If there is significant drop-off between step 3 and 4, then maybe you should make it easier for someone to purchase your product, reducing the friction to make the purchase.

At step 4, the user has committed to making the purchase. But that doesn't always guarantee that the user will actually pay. Sometimes if you force them to go through a long signup process, you'll successfully convince them that it's not worth the time to fill everything out and they'll just leave.

And finally, step 5 means that you got a new customer. It's always worth your time to find customers who actually converted and backtrack their actions to see what converting customers so you might be able to garner some insights into how to turn other potential customers into paying customers.

Customer engagement

Are new users more engaged than existing users? Are people in Australia more engaged than people in Texas? If so, why? Are users that receive direct communication from an admin more likely to be engaged? There are hundreds of useful questions you could ask.

For PhoenixChat, we also want to know if people are actually using the app. But not only that, we want to know how they're using the app. If we find that all of our customers are using a certain feature, but we are hiding that feature behind several layers of dropdown menus, then maybe we should move that item to somewhere more visible.

If you build out a feature (say, gamification or statistics), and you later discover that nobody is using it, that's helpful information. It tells you that you need to 1) figure out a way to make these features more interesting to your users, 2) easier to find, or 3) abandon them altogether and stop spending resources on something that nobody uses.

Feature usage also tells you what features are useful to your customers and what you should be spending your time on. If your customers seem to spend lots of time looking at their statistics, perhaps you should add fancier charts and give them more feedback about the performance of their posts. If users seem to send direct messages to each other on a regular basis, maybe you should expand chat into a standalone app with additional functionality.

Tracking feature usage is one of the keys to knowing how and where to pivot if necessary. You might find

that you set out to build a social network, but your users are actually more interested in some small subset of your app, such as image sharing. You can either try to force everyone into the existing feature set (which is almost always wrong), or you can embrace the behavior of your users and focus your app on what they find most useful.

The many options in analytics

There are a seemingly endless number of options when it comes to analytics. Google Analytics, Mixpanel, Kissmetrics, Heap, Segment.io, Customer.io, Intercom.io, and dozens (hundreds?) of others.

At the end of the day, most of these are the 90% the same. They all track events and they all track user behavior and pageviews.

Almost every website uses Google Analytics. There is much more to Google Analytics than just the snippet of code that tracks pageviews etc.,

In the software space, the two big players are Mixpanel and Kissmetrics. The two are more or less identical with a slightly different user interface. They give you great reporting, deep user analytics, and a lot of other useful features--some of which we will go over later.

For this app, we are going to use Segment.io and Mixpanel.

Segment.io is an intermediary that separates your data from your analytics. You send all of your events to Segment.io and then have Segment.io pass that data to the analytics service you want to use. We are going to use Segment.io to pass data into both Mixpanel and Customer.io.

If you have a preference for Kissmetrics, feel free to use it. We are going with Mixpanel over Kissmetrics solely because it has a free tier. Also, since we're using Segment.io as a data-intermediary, there is nothing stopping you from using both. All you need to do is toggle the switch to turn on any number of analytics tools.

The next step is to actually integrate Segment.io and start tracking the usage statistics of our app.

Connecting Segment.io and Mixpanel

- Using `redux-segment`
- Activating Mixpanel

Since we're using Redux, we're lucky. We don't have to worry about

React and SEO

structured data

document.title

sitemap.xml

data highlighter

Password Reset Link and Email

- Password reset flow
- Generate reset token
- Validate token and reset password

In this lesson, we're going to cover how to send a password reset link via email. If you've skipped any lessons so far, you'll want to make sure that you've at least implemented transactional email.

Password reset flow

Resetting a password is more complicated than you might expect. The way password reset works is

First, create a link that allows a user to request a new password

Second, create the view that for the user to submit the reset link

Third, generate and send the password reset link via email

Fourth, create another view that accepts a token as a query parameter and sends the new password along with the token

Fifth, match the token with an existing user account

Sixth, automatically log the user in if the token and new password is valid

Generate reset token

The first thing we're going to do is change our backend to accept a password reset. We're going to need two additional endpoints to add this functionality.

The first endpoint will be `reset-request` which will take the initial request from the frontend and will include the email address for which we want to reset the password. The second endpoint will be `reset` which will pass along the new password and a (presumably) valid token in order to reset the password of the account that made the request. Let's add those endpoints now.

```
...

scope "/api", PhoenixChat do
  pipe_through [:api, :api_auth]

  resources "/users", UserController, except: [:index, :show, :new, :edit]
  post "/users/reset-request", UserController, :reset_request, as: :reset
  post "/users/reset", UserController, :reset, as: :reset

  ...
end

...
```

The next thing we should do is update our `UserController`. You can see in the code above that we created two new functions, `:reset_request` and `:reset`, so we should define those now. We will start with `reset_request`, which is the function we'll call along with an email to send the reset link to the email address specified. We'll go over each line below the code block.

```
defmodule PhoenixChat.UserController do
  ...

  def reset_request(conn, %{"email" => email}) do
    user = Repo.get_by(User, email: String.downcase(email))

    if user do
      user
      |> User.password_reset_changeset
      |> Repo.update!
      |> send_password_reset_email

      send_resp(conn, :no_content, "")
    else
      send_resp(conn, :bad_request, "user does not exist")
    end
  end

  ...
end
```

First, we check to see if the user exists by using `Repo.get_by`. If the user exists, we pass the user along to the `password_reset_changeset` (which we will create later) that will generate the token and store it along with our user so we can match against it later.

Then we send it along to the `send_password_reset_email` function, which we will define later. If it does not exist, we send along an error.

In order to store this token with our user, we need to create a new migration to update our `User` model.

```
$ mix ecto.gen.migration add_user_password_reset_fields
```

Within our new migration, we need to add two fields. One is the token that we will match against and the other is a timestamp for the last time a user changed her password.

```
defmodule PhoenixChat.Repo.Migrations.AddUserPasswordResetFields do
  use Ecto.Migration

  def change do
    alter table(:users) do
      add :password_reset_token, :string, default: nil
      add :password_reset_timestamp, :datetime
    end
  end
end
```

Go ahead and run `mix ecto.migrate`. Now that we have our model updated, we need to create our `password_reset_changeset`. This looks similar to our other changesets except for the `put_token` function, in which we will generate a token and add it to our user.

```
...

def password_reset_changeset(model, params \\ %{}) do
  model
  |> changeset(params)
  |> put_change(:password_reset_timestamp, Ecto.DateTime.utc)
  |> put_token(:password_reset_token)
end

...
```

Now we need to define `put_token`. In the function below, we're checking to make sure the changeset is valid (recall that changesets have the `valid?` parameter), and if it is, we call `generate_token` (defined later) and use `put_change` to add the token to our user.

```
...

def put_token(changeset, field) when field in ~w(password_reset_token)a do
  case changeset do
    %Ecto.Changeset{valid?: true} ->
      token = generate_token()
      put_change(changeset, field, token)
    _ ->
      changeset
  end
end

...
```

So now we need to define `generate_token`. All we're doing is creating a 50-character random string.

```
...

defp generate_token do
  50
  |> :crypto.strong_rand_bytes
  |> Base.url_encode64
  |> binary_part(0, 50)
end

...
```

This confuses a lot of people because it's so simple. Contrary to what you might think, there is no magic that comes with keys or tokens—they're just random numbers and letters that you match against another key or token.

In our case, we are creating a token and saving it with our user. When that user requests a new password, we send along that token as a query parameter in a link via email. When that link is clicked, we search our database for matching tokens, and if we find a match, we allow the user to reset the password.

Send reset email

Now that we have our token, we need to define the password reset email function, which we're calling in our `reset_request` function. This simply takes in the user and passes it along to the `Email.password_reset_email` function, which we will also need to define.

```
defmodule PhoenixChat.UserController do
  ...

  defp send_password_reset_email(user) do
    user
    |> Email.password_reset_email
    |> Mailer.deliver_later
  end

  ...
end
```

Now we need to define the email that is sent to the user who forgot her password. Recall that we're taking in the `user`, which now has a `password_reset_token` field. For now, we're going to hard-code the reset link, but eventually we would want to use the `email_reset_link` function to set the link based on the environment (`test`, `dev`, `prod`, etc).

The first thing we do is `URI` encode the query so that it makes for a valid link, then we concatenate it with the `email_reset_link`. From there, we pass along the `reset_link` to the email address associated with the current user.

```
defmodule PhoenixChat.Email do
  ...

  def password_reset_email(user) do
    query = URI.encode_query(token: user.password_reset_token)
    reset_link = "#{email_reset_link}?#{query}"

    user
    |> base_email
    |> subject("Password Reset")
    |> text_body("""
      A password reset was requested, follow #{reset_link} to reset your password.
      If this was a mistake, please ignore this email.
      """)
  end

  defp email_reset_link do
    # This will eventually handle for environment using
    # Application.get_env(PhoenixChat.Mailer)
    "http://localhost:3000/#/reset-password"
  end

  ...
end
```

Now, when a user clicks on the link, she will be directed a page that will allow her to set a new password.

The last thing we need to do before moving to the frontend is to set up our `reset` endpoint that will actually allow our user to change her password.

In this function, we're going to find the user that has the matching token (`user_for_password_token`), ensure that the token is less than 48 hours old, update the password, send the user an email, and log the user in.

Adding an expiration is a good idea from a security perspective because it limits the time that an account could be potentially vulnerable. That said, it's still next to impossible that someone can guess a 50-character token in their lifetime no matter how many guesses per second (see our blog post for more information on collision probabilities), so leaving an account sort-of-vulnerable for 48 hours is not a big deal.

```
defmodule PhoenixChat.UserController do
  ...

  def reset(conn, %{"token" => token, "password" => password}) do
    case user_for_password_token(token) do
      user = %User{} ->
        user
        |> User.registration_changeset(%{password: password})
        |> Repo.update!

        {:ok, jwt, _claims} = Guardian.encode_and_sign(user, :token)

        send_password_change_email(user)

        conn
        |> put_status(:ok)
        |> render("show.json", user: user, web_token: jwt)
      nil ->
        conn
        |> put_status(:bad_request)
        |> render(ErrorView, "error.json", errors: ["invalid or expired token"])
    end
  end

  ...
end
```

This should all look familiar. We are using `user_for_password_token` (defined below) to match the token and sure validity, then using the existing `registration_changeset` to update the password, then `Guardian.encode_and_sign` to log the user in, just as we did before in our automatic login on signup, and finally sending the email.

The `user_for_password_token` function is a little bit tricky. We're creating a query that searches through our `User` models and finds a `password_reset_token` that matches the token that was passed into the function. Then, it checks to make sure that the timestamp is less than 48 hours old using fragment, which allows us to send database queries directly to Postgres.

```
...

  defp user_for_password_token(token) do
    query = from u in User,
            where: u.password_reset_token == ^token
              and u.password_reset_timestamp > fragment("now() - interval '48hours'"),
            select: u
    Repo.one(query)
  end

...
```

Now we need to define the function in which we send the confirmation email.

```
...

defp send_password_change_email(user) do
  user
  |> Email.password_change_email
  |> Mailer.deliver_later
end

...
```

And finally, define the email.

```
...

def password_change_email(user) do
  user
  |> base_email
  |> subject("Password Change")
  |> text_body("Your password on Learn Phoenix has been updated recently.")
end

...
```

And that's it for our backend. Now we need to create the forms on our frontend and connect it to our backend.

Connect Password Reset

Send New Message Notifications

Browser Notifications

Web Workers and Page Visibility

- Basics of Web Workers
- Page Visibility API
- Send messages in the background

coming soon

Persisting ETS Data

Add Profile Image

Basics of React Native

- Background
- Differences from React
- Installation
- Hello World!
- Ignite

One of the biggest advantages of this stack is the ease with which you can create native mobile applications that share code and allow for significantly faster iteration cycles.

In this lesson, we're going to go over some of the basics of React Native, walk through a "Hello World!" app, then use Ignite to create a boilerplate app.

Background

TODO(image: cordova, ionic)

In the olden days we built hybrid apps with technologies like Cordova, Phonegap, Ionic, Sencha Touch, et al. They gave us easy access to native API calls, but the majority of the app still ran as HTML and JavaScript within a WebView, but it was still better than the alternatives (which were limited). The motto of the hybrid app:

> Write once, run everywhere

But while these hybrid apps made the development process easier, it was not as performant as truly native app written in Objective C (iOS) or Java (Android). Things like scrolling, keyboard behavior, and other small things were a dead giveaway that the app wasn't really native.

TODO(image: react native)

Then in 2015, Facebook released React Native, which changed everything. While everything is still written in JavaScript (similar to a web-based React project), the components are rendered as native platform widgets that run on a separate thread, giving you native look, feel, and performance (for most tasks) that was not possible with your typical hybrid app. The new motto:

> Learn once, write everywhere

The reason for this change in mindset is that while you can share logic between platforms, it does not make sense to deploy the same frontend app to Android as iOS, since user experience will be noticeably

different. But, by sharing a similar syntax, we can at least save ourselves the time of learning a new programming language and syntax and we can reuse a lot of our core logic between the platforms.

To further drill down on the question, "But is it really native?", for all practical purposes the answer is "yes". Your JavaScript code is run on its own thread that is separate from the main UI thread, so even when your JavaScript code is running complicated logic, it won't mess with your UI, which is the reason that hybrid apps can feel janky.

Differences from React

Since we've been working with React, you'll feel right at home. Instead of using `<div>` and `<p>` tags, you use `<View>` and `<Text>`. These are mapped to the iOS and Android equivalents of `UIView` and `android.view` respectively and are rendered as native components.

React Native also does not support CSS, so you'll have to do all your styling in JavaScript, much like how we handled styles in our NPM component. An example stylesheet would look like the code below.

```
import { StyleSheet } from 'react-native'

export default StyleSheet.create({
  container: {
    justifyContent: 'center',
    marginVertical: '25'
  }
})
```

You probably noticed the `StyleSheet.create` function. This is not strictly necessary, but it's something you get for free with React Native which handles style caching and it's something you'll want to use.

React Native also uses a subset of Flexbox to handle alignment and justification. Since we're already using Flexbox, this should be easy to understand once we start implementing it.

Routing is done through `Navigator` and there are several libraries out there that make it as easy to use as `react-router`, which we have already used.

And that's about it. With the exception of directory structure changes, you'll start to feel comfortable writing React Native code in no time.

Installation

There's a lot to install, and this can take some time. Follow the steps in the docs to install everything you need. The basics are:

```
$ brew install node
$ brew install watchman
$ npm install -g react-native-cli
```

Then you need to make sure you have Xcode installed as well as Android Studio. Once that's done, we can move to the next step.

Hello World!

The React Native Docs handle the "Hello World!" app about as well as it can be handled, but we'll go over it here for the sake of simplicity. The simplest possible app looks like the code below.

```
import React, { Component } from 'react'
import { AppRegistry, Text } from 'react-native'

class HelloWorldApp extends Component {
  render() {
    return (
      <Text>Hello world!</Text>
    )
  }
}

AppRegistry.registerComponent('HelloWorldApp', () => HelloWorldApp)
```

With the exception of `AppRegistry` (which is effectively `ReactDOM`), this should look very familiar. And that's it! You can run this on iOS with the following command:

```
$ react-native run-ios
```

Ignite

Unlike previous lessons where we built the app from an empty text file, we're going to use a generator. One of the best generators out there is Ignite put together by the fine people at Infinite Red, a dev shop based out of Portland and San Francisco. Once we have that set up, we'll spend the rest of the lesson walking through the different pieces of the app.

The first step is to install Ignite.

```
$ npm install -g react-native-ignite
```

Then we should create our app.

```
$ ignite new phoenix-chat-mobile
```

Now to run it, `cd` into the directory and build it. We'll stick with iOS for now.

```
$ cd phoenix-chat-mobile
$ react-native run-ios
```

This will build your app and launch the simulator. This can take some time, so be patient. Also, for some reason this can error out on your first build, so go ahead and refresh with **Command + R** if you run into a glaring red error screen.

Now let's take a look at the directory structure. Most of the code you write will be within the `App` directory. We're only going to touch on things that are directly relevant to our app for now, then branch out as other pieces become necessary.

The `Components` directory is where things like buttons will live. You'll see that all of these components have an associated `.js` file in the `./Styles` sub-directory. This is where the styles for those components live.

The next directory to look at is `Containers`, which contains your views. The first view you'll see when you run the simulator is `PresentationScreen`, so let's check out that file:

```
/App/Containers
commit: coming soon
```

```
import React from 'react'
import { ScrollView, Text, Image, View } from 'react-native'
import { Images } from '../Themes'
import RoundedButton from '../Components/RoundedButton'
import { Actions as NavigationActions } from 'react-native-router-flux'

// Styles
import styles from './Styles/PresentationScreenStyle'

export default class PresentationScreen extends React.Component {
  render () {
    return (
      <View style={styles.mainContainer}>
        <ScrollView style={styles.container}>

          <View style={styles.section} >
            <Text style={styles.sectionText} >
              Default screens for development, debugging, and alpha testing
              are available below.
            </Text>
          </View>

          <RoundedButton onPress={NavigationActions.componentExamples}>
            Component Examples Screen
          </RoundedButton>

          ...

          <View style={styles.centered}>
            <Text style={styles.subtitle}>Made with ❤ by Infinite Red</Text>
          </View>

        </ScrollView>
      </View>
    )
  }
}
```

With the exception of the different tag names, this should look very familiar. `<View>` is the replacement for `<div>`, `<ScrollView>` docs lets React Native know that this should scroll and `<Text>` contains text. The `RoundedButton` component is simply an imported component, just like we've done several times already. Also, `onPress` replaces `onClick`.

It's also worth checking out the list of generators that Ignite gives us, which will save you some time down the road.

Now that we have the app template, it's time to dig into the meat of Ignite and build our mobile app with React Native.

Server-Side Rendering with React

- Set up Express server

Coming Soon

Stripe Subscriptions

- Add Stripe
- Connect customer_id to user

While one-off payments with Stripe are so easy as to be almost trivial, setting up subscriptions is surprisingly complicated. This is primarily due to the sizable number of edge cases that need to be handled. For example, you need to handle all of the following:

- Does subscription end immediately when user cancels?
- Does subscription refund pro-rated amount?
- User upgrades/downgrades?
- User cancels, then re-subscribes?
- Plan expiration date?
- Plan is active, but does not auto-renew?

Patch Security Vulnerabilities

- Index and show user endpoints
- Delete other users

coming soon