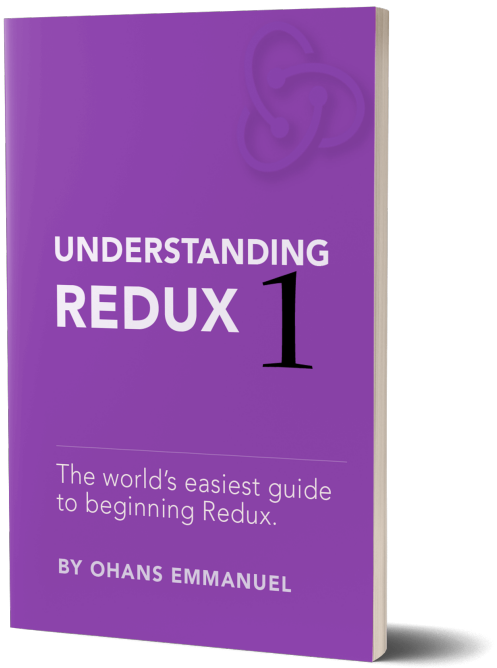

UNDERSTANDING
REDUX 1
The world's easiest guide to beginning Redux.
BY OHANS EMMANUEL

Understanding Redux.

Table of Contents (unexpanded)

Introduction

This book is the missing piece if you've long searched for how to master Redux .

Before getting started, I should tell you that this book is first and foremost about me. Yes, me. My struggles with learning Redux, and seeking a better way to teach it.

A few years ago, I had just learned React. I was excited about it, but again, everyone else seemed to be talking about something else called Redux. *Gosh! Does the learning streak ever end?*

As an Engineer committed to my personal development, I wanted to be in the know. I didn't want to be left out. So, I began to learn Redux.

I checked the Redux documentation. It was pretty good, actually! For some reason, it just didn't entirely click for me. I checked a bunch of youtube videos as well. The ones I found just seemed rushed and not detailed. Poor me.

Honestly, I don't think the video tutorials I watched were bad. There was just something missing. An easy guide that was well-thought-out and written for a sane person like me, and not for some imaginary humanoid.

It appeared I wasn't alone.

A good friend of mine, someone I was mentoring at the time, had just completed a React Developer Certification course where he paid big bucks (over $300) to earn a certificate.

Guess what? When I asked for his honest feedback on the program, his words were along the lines of this:

> *The course was pretty good, but I still don't think Redux was well explained to a beginner like me. It wasn't explained that well.*

You see, there are many more like my friend, all *shitting their Redux pants*. They perhaps use Redux, but they can't say they truly understand how it works.

I decided to find a solution. I was going to understand Redux deeply, and find a clearer way to teach it.

What you are about to read took months of study, and some more time to write and build out the example projects while keeping a daily job and other serious commitments.

But you know what?

I'm super excited to share this with you!

If you've searched for a Redux guide that won't talk over your head, this is it. Don't look further.

I have taken into consideration my struggles and those of many others I know.

I'll make sure to teach you the important stuff - and do so without getting you confused.

Now, that's a promise.

My Approach to Teaching Redux

The real problem with teaching Redux - especially for beginners, isn't the complexity of the Redux library itself.

No. I don't think that is it. It is just a tiny 2kb library (including dependencies.)

Take a look at the Redux community as a beginner, and you're going to lose your mind fast. There's NOT just Redux, but a whole lot of other supposed "associated libraries" to build real world apps.

If you've spent some time doing a bit of research, then you've come across them already. There's Redux, React-Redux, Redux-thunk, Redux-saga, Redux-promise, Reselect, Recompose and many more!

As if that's not enough, there's also some Routing, Authentication, Server side rendering, Testing, and Bundling sprinkled on it - all at once.

Gosh! That is overwhelming.

The "Redux tutorial" isn't so much about Redux, but all the other stuff that come with it.

There's got to be a more sane approach tailored towards beginners. If you're a humanoid developer, you certainly don't have issues with this. Guess what? Most of us are humans.

So, here's my approach to teaching Redux.

Forget about all the extra stuff for a bit and let's just do Redux. Yeah!

I will only introduce the barest minimum you need for now. There will be NO React-router, Redux-form, Reselect, Ajax, Webpack, Authentication, Testing, none of those - for now!

And guess what, that's how you learned to do some of the important life 'skills' you've got.

How did you learn to walk?

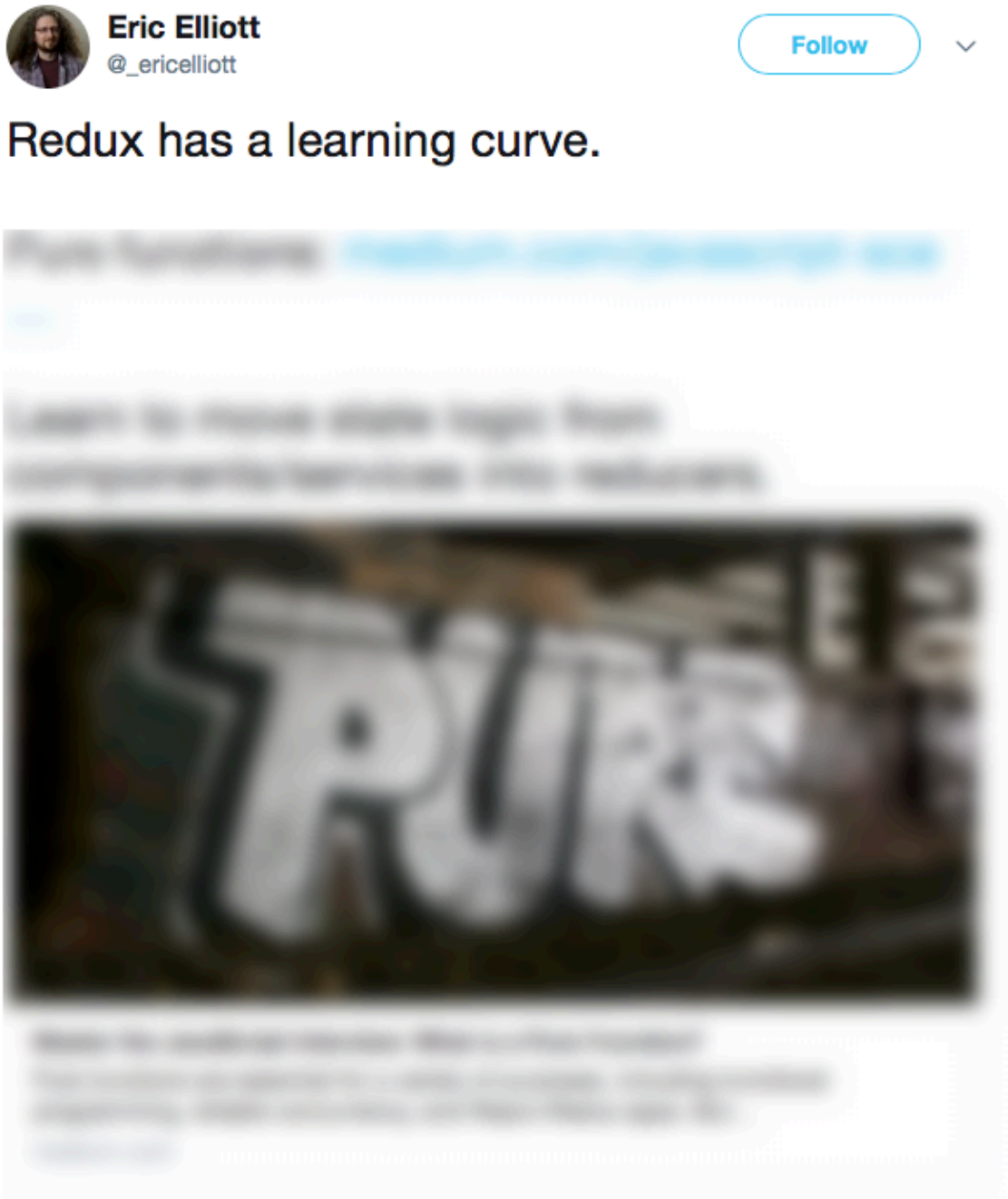
Did you begin to run in one day? Hell, no!

Let me walk you through a sane approach to learning Redux - without the hassles.

Sit tight.

"A rising tide lifts ALL boats"— Once you get the hang of how the basics of Redux works (the rising tide), EVERYTHING else will be easier to reason about (It lifts all boats.)"

A Note on Redux's Learning Curve



Redux does have a learning curve. I am not saying otherwise.

Learning to walk also had a learning curve to it. However, with a systematic approach to learning, you overcame that.

You did fall a few times, but that was okay. Someone was always around to hold you up, and help you get on your feet.

Well, I'm hoping to be that person for you - as you learn Redux with me.

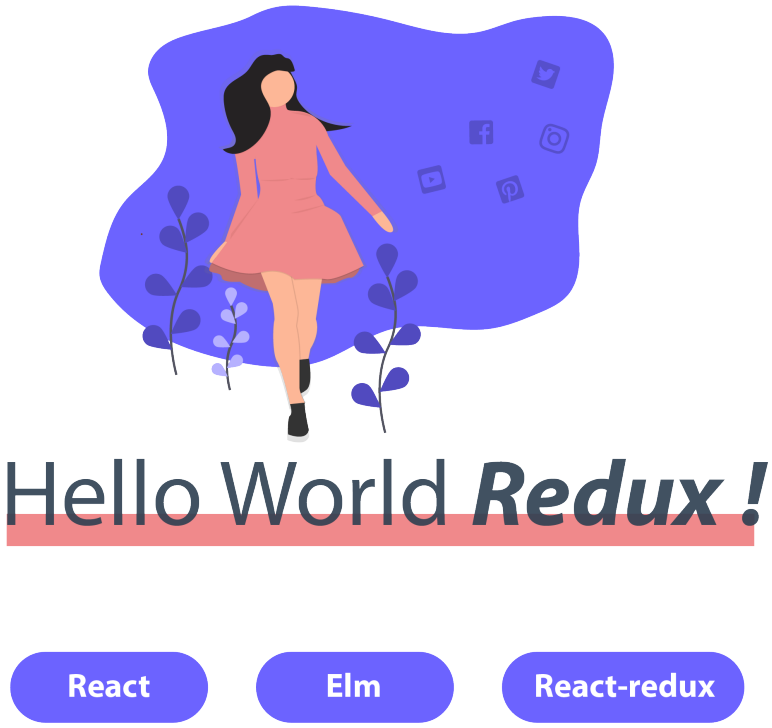
What You will Learn

After all is said and done, you'll come to see that Redux isn't as scary as it seems from the outside.

The underlying principles are so darn easy!

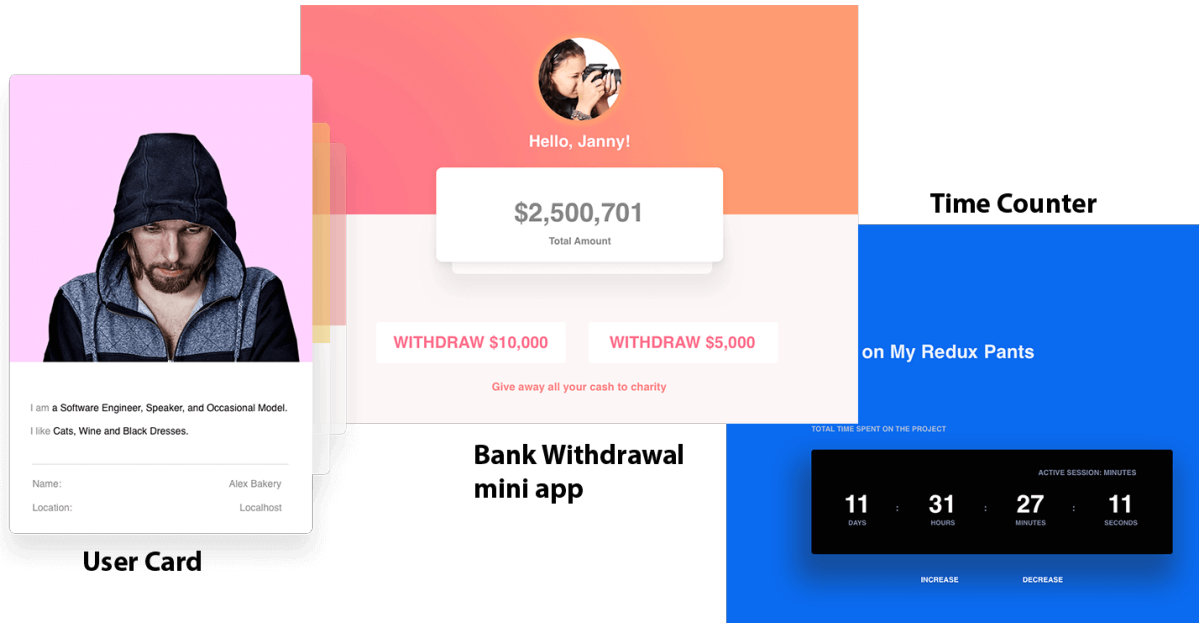
First off, I'll teach you the fundamentals of Redux in plain, easy to approach language.

Then, we'll build a few simple applications. Starting with a basic Hello World app.



But those won't suffice.

I'll include exercises and problems I think you should tackle as well.

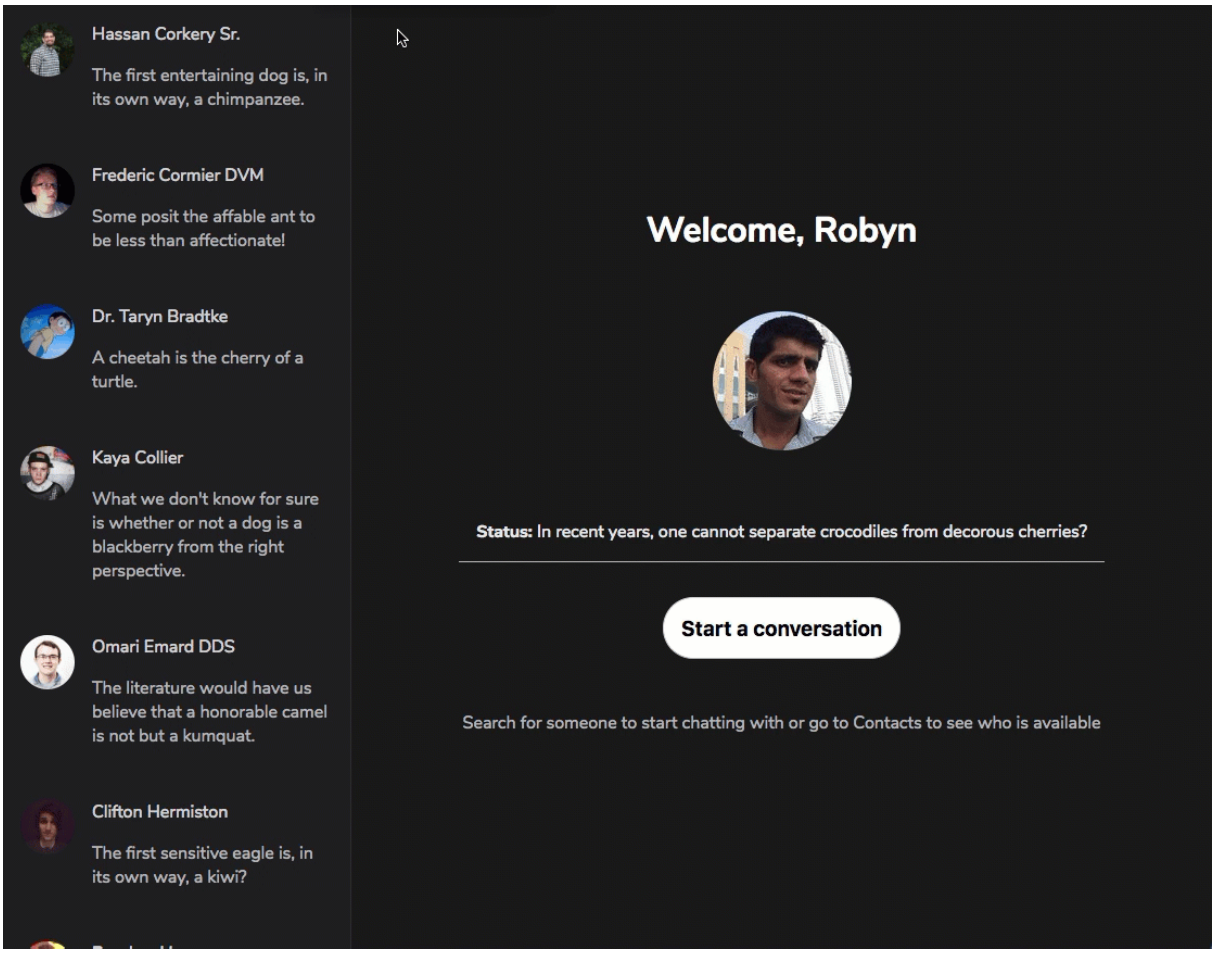


Effective learning isn't about just reading and listening. Effective learning is mostly about practice!

Think of these as homework, but without the angry teacher. While practicing the exercises, tweet at me with the hashtag *#UnderstandingRedux* and I'll definitely have a look!

No angry teachers, eh?

Exercises are good, but you also need to watch me build a bigger application. This is where we wrap things up by building Skypey. A sweet messaging app. Uh, *kinda* like a Skype clone.



Skypey's got features such as editing messages, deleting messages, and sending messages to multiple contacts.

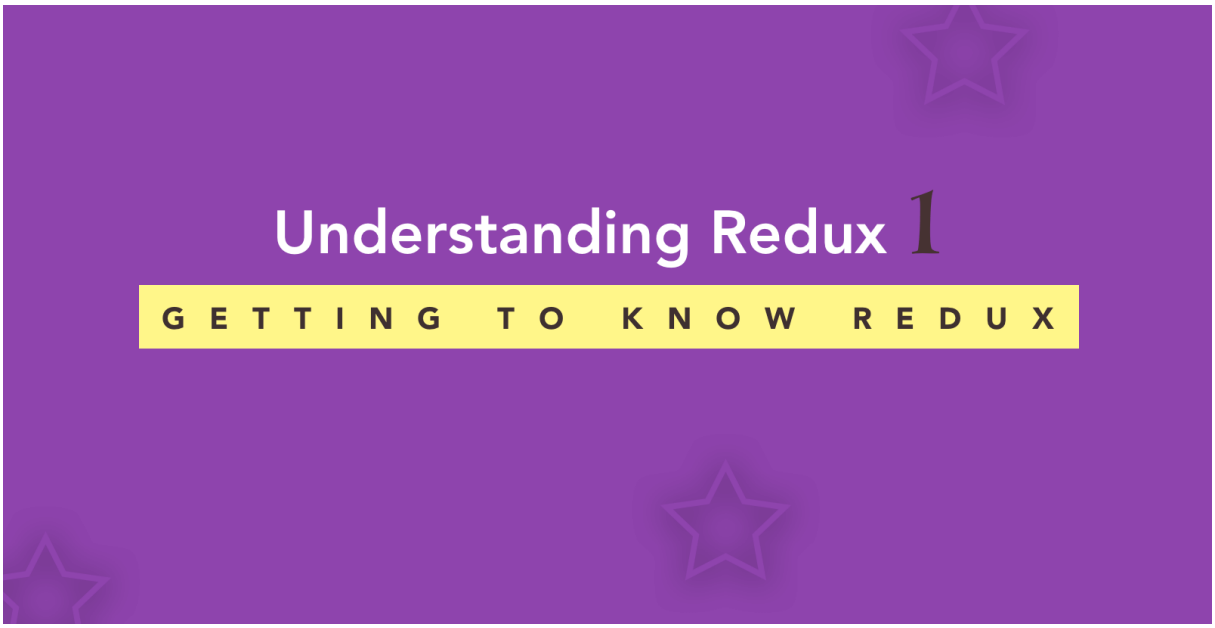
Hurray!

If that didn't get you excited, I don't know what will. I'm super excited to show you these!

Prerequisite

The only prerequisite is that you already know React. If you don't, Daveceddia's Pure React is your best bet if you have some $$ to spare. I'm no affiliate. It's just a good resource.

Chapter 1 : Getting to know Redux.



Some years back, developing frontend applications seemed like a joke to many. These days, the increasing complexity of building decent frontend applications is almost overwhelming.

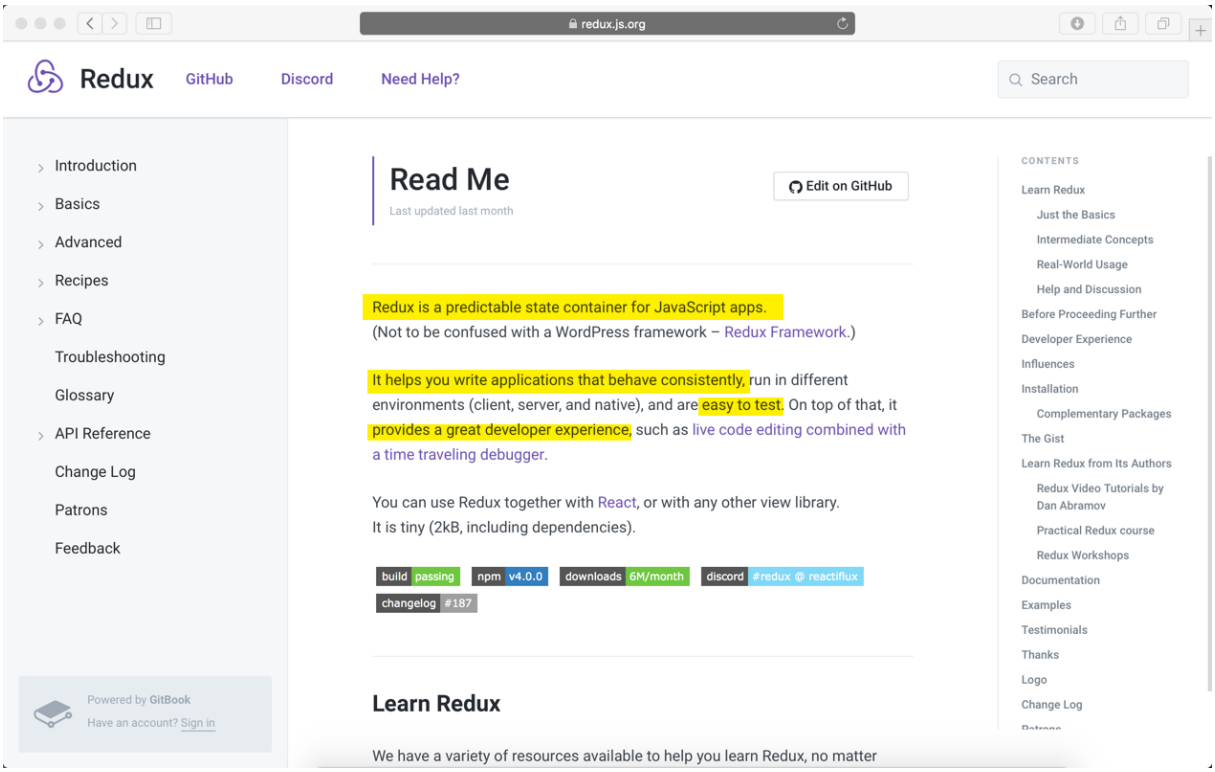
It seems that to meet the pressing requirements of the ever-demanding user, the gentle cute cat has overgrown the confines of a home. It's become a fearless lion with 3-inch claws and a mouth that opens wide enough to fit a human head.

Yeah, that's what modern frontend development feels like these days.

Modern frameworks like Angular, React and Vue have done a great job at taming this "beast". Likewise, modern philosophies such as those enforced by Redux, also exist to give this "beast" a chill pill.

Follow along as we have a look at these philosophies.

What is Redux?



The official documentation for Redux reads:

> *Redux is a predictable state container for JavaScript apps.*

Those 9 words felt like 90 incomplete phrases when I first read them. My bad. I just didn't get it. You most likely don't too.

Don't sweat it. I'll go over that in a bit, and as you use Redux more, that sentence will get clearer.

On the bright side, if you read the documentation a little longer, you'll find the more explanatory stuff somewhere in there.

It reads:

> *It helps you write applications that behave consistently…"*

You see that?

In lay-man's terms, that's saying, *"it helps you tame the beast"*. Metaphorically.

Redux takes away some of the hassles faced with state management in large applications. It provides you with a great developer experience, and makes sure that the testability of your app isn't sacrificed for any of those.

As you develop React applications, you may find that keeping all your state in a top-level component is no longer sufficient for you.

You may also have a lot of data changing in your application over time.

Redux helps solve this kind of problems. Mind you, it isn't the only solution out there.

Why use Redux?

As you already know, questions like, *why should you use A over B?* boils down to your personal preferences.

I have built apps in production that DON'T use Redux. I'm sure that many have done the same too. For me, I was worried about introducing an extra layer of complexity for my team members. Incase you're wondering, I don't regret the decision at all.

Redux's author, Dan Abamov also warns about the danger of introducing Redux too early into your application. You may not like Redux, and that is fair enough. I have friends who don't.

That being said, there are still some very decent reasons to learn Redux.

For example, in larger apps with a lot of moving pieces, state management becomes a huge concern and Redux ticks that off quite well without performance concerns or trading off testability.

One other reason a lot of developers love Redux is the developer experience that comes with it. A lot of other tools have began to do similar things, but big credits to Redux.

Some of the nice things you get with using Redux includes logging, hot reloading, time travel, universal apps, record and replay - all without doing so much on your end as the developer. These things will likely sound fancy except you use them and see for yourself.

Dan's talk called Hot Reloading with Time Travel will give you a good sense of how these work.

Also, Mark Ericsson, one of Redux's maintainers, says that over 60% of React apps in production use Redux. That's a lot!

Consequently, and this is just my thought, a lot of engineers like to show potential employers that they can maintain larger production codebases built in React & Redux, so they learn Redux.

If you want some more reasons to use Redux, Dan, the Redux creator, has a few more reasons highlighted in his article on Medium.

If you don't consider yourself a senior engineer, I advice you learn Redux - largely because of some of the principles it imbibes. You'll learn new ways of doing common things, and this will likely make you a better engineer.

Everyone has different reasons for picking up different technologies. In the end, the call is yours, but it definitely doesn't hurt to add Redux to your skillset.

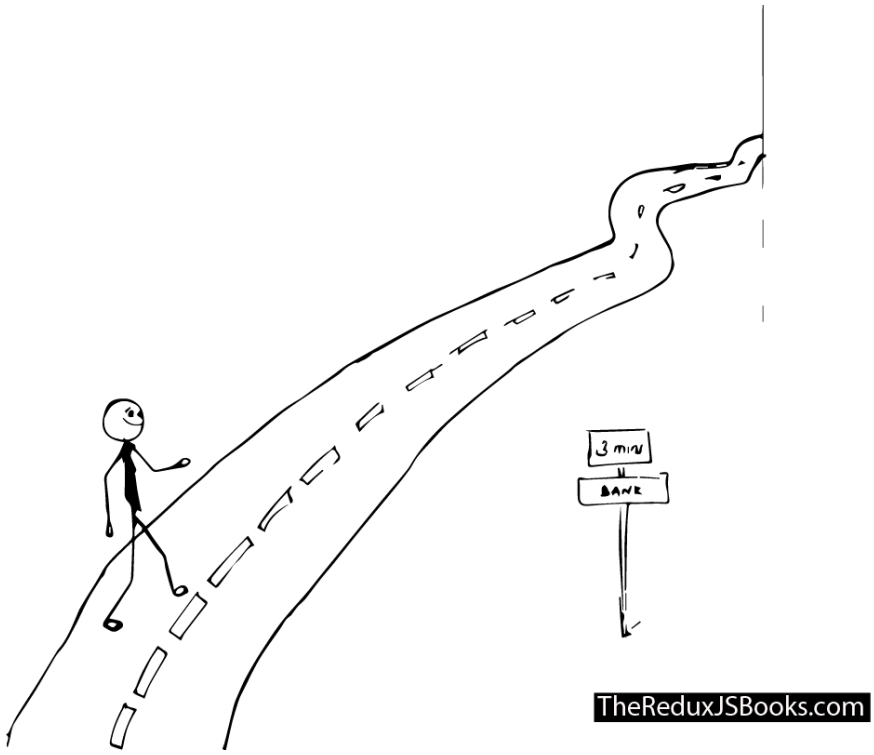
Explaining Redux to a 5 year Old.

This section of the book is really important. The explanation here will be referenced through out the book. So get ready.

Since a 5-year old doesn't have the time for technical jargon, I'll keep this very simple but relevant to our purpose of learning Redux.

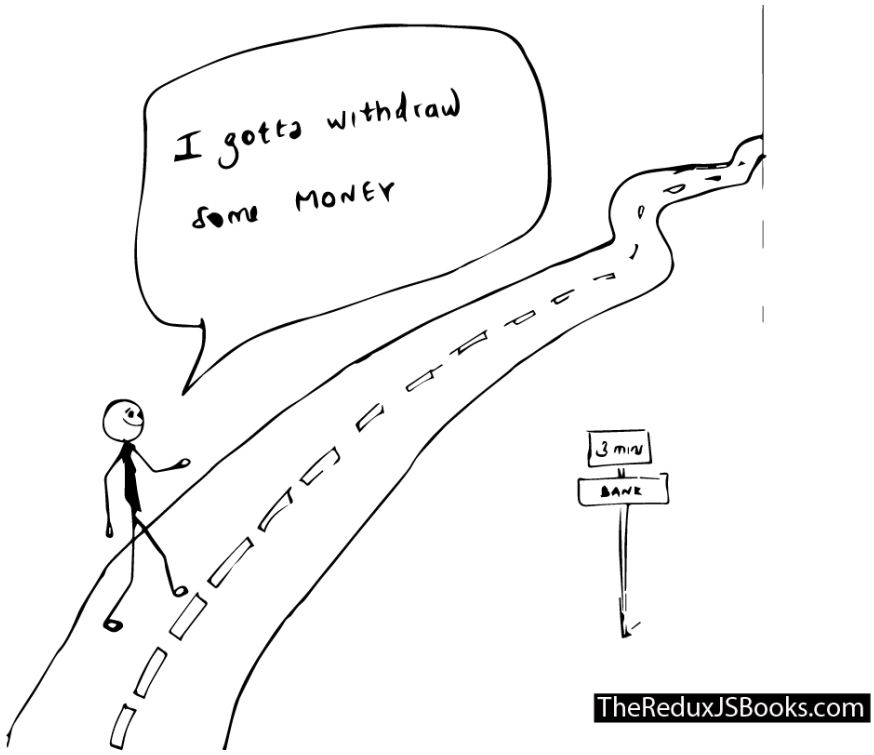
So, here we go!

Let's consider an event you're likely conversant with - going to the bank to withdraw cash. Even if you don't do this often, you're likely aware of what the process looks like.



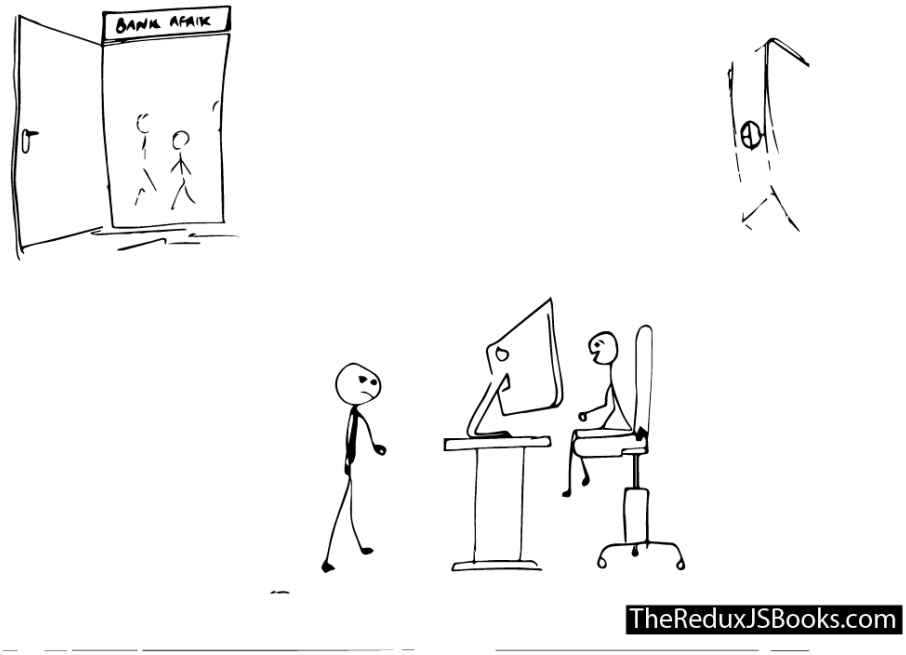
You wake up one morning, and head to the bank as quickly as possible. While going to the bank there's just one **intention / action** you've got in mind i.e `WITHDRAW_MONEY`

You want to withdraw money from the bank.



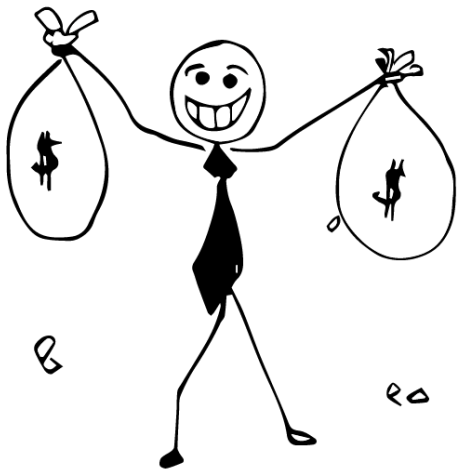
Here's where things get interesting.

When you get into the bank, you then go straight to the Cashier to make your request known.



Wait, you went to the Cashier?

Why didn't you just go into the bank vault to get your money?



After all, it's your hard earned money.

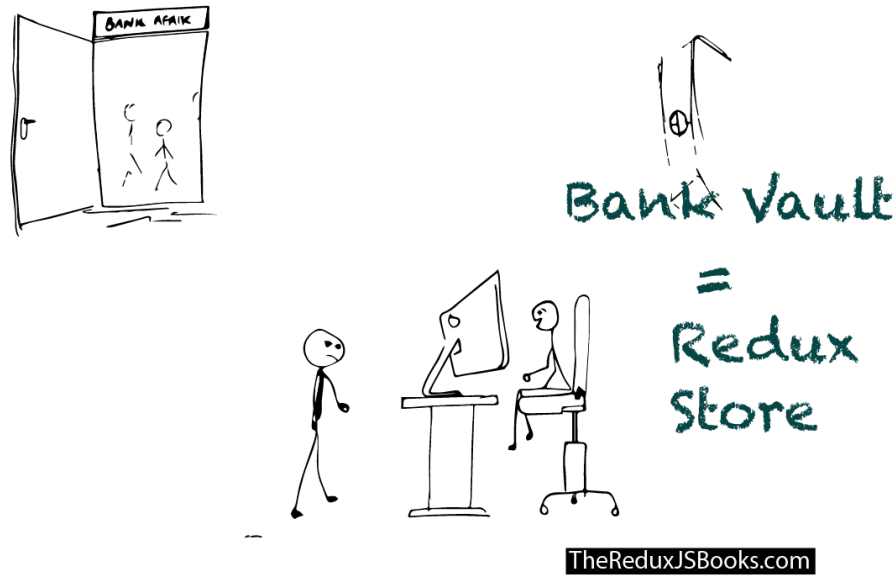
Well, like you already know, things don't work that way. Yes, the bank has money in the vault, but you have to talk to the Cashier to help you follow a due process for withdrawing your own money.

The Cashier, from their computer, then enters some commands and delivers your cash to you. Easy-peasy.

Now, how does Redux fit into this story?

We'll get to more details soon, but first, the terminologies.

1. The BANK VAULT is to the bank what the REDUX STORE is to Redux.



The bank vault keeps the money in the bank, right?

Well, within your application, you don't spend money. Instead, the STATE of your application is like the money you spend. The entire user interface of your application is a function of your state.

Just like the bank vault keeps your money safe in the bank, the state of your application is kept safe by something called a STORE. So, the STORE keeps your "money" i.e state, intact.

Uh, you need to remember this, okay?

The Redux Store can be likened to the Bank Vault. It holds the state of your application - and keeps it safe.

This leads to the first Redux principle:

> *Have a single source of truth: The state of your whole application is stored in an object tree within a single Redux store.*

Don't let the words confuse you.

In simple terms, with Redux, it is is advisable to store your application state in a single object managed by the Redux STORE. It's like having ONE VAULT as opposed to littering money everywhere along the bank hall.

The Redux principle #1.

ONE application STATE OBJECT
managed by ONE STORE

2. Go to the bank with some ACTION in mind.

If you're going to get any money from the bank, you're going to have to go in with some intent or action to withdraw money.

If you just walk into the bank and roam about, no one's going to just give you money. You may even end up been thrown out by the security. Sad stuff.

The same may be said for Redux.

Write as much code as you want, but if you want to update the state of your Redux application (like you do with `setState` in React,) you need to let Redux know about that with an ACTION.

In the same way you follow a due process to withdraw *your own money* from the bank, Redux also accounts for a due process to change/update the state of your application.

Now, this leads to the Redux principle #2.

> *State is read-only:*
>
> *The only way to change the state is to emit an action, an object describing what happened.*

What does that mean in plain language?

When you walk to the bank, you go there with a clear action. In this example, you want to withdraw some money.

If we chose to represent that process in a simple Redux application, your action to the bank may be represented by an object.

One that looks like this:

```
{
  type: "WITHDRAW_MONEY",
  amount: "$10,000"
}
```

In the context of a Redux application, this object is called an **action!** It always has a `type` field that describes the action you want to perform. In this case, `WITHDRAW_MONEY`

Whenever you need to change/update the state of your Redux application, you need to dispatch an action.

The Redux principle #2.

The only way to change the state is to emit an action, an object describing what happened.

https://TheReduxJSBooks.com

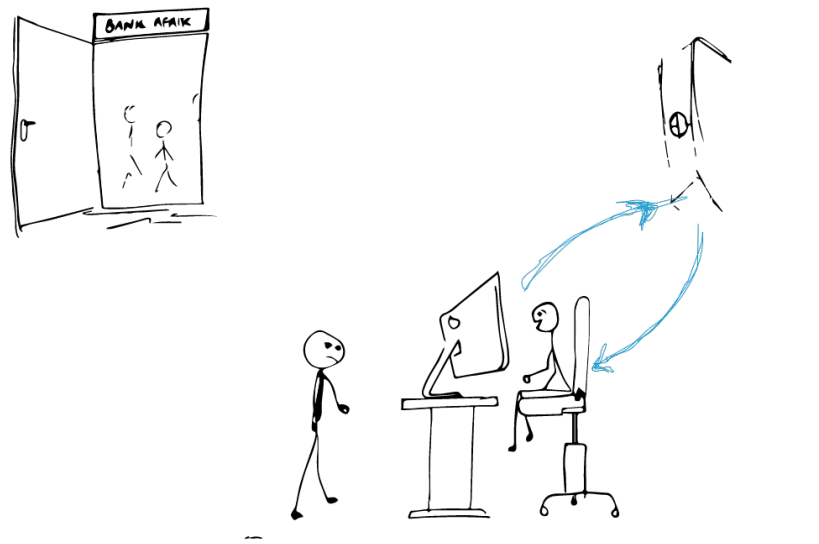
Don't stress over how to do this yet. I'm only laying the foundations here. We'll sure delve into lots of examples soon.

3. The CASHIER is to the bank what the REDUCER is to REDUX

Alright, take a step back.

Remember that in the story above, you couldn't just go straight into the bank vault to retrieve your money from the bank. No. You had to see the Cashier first.

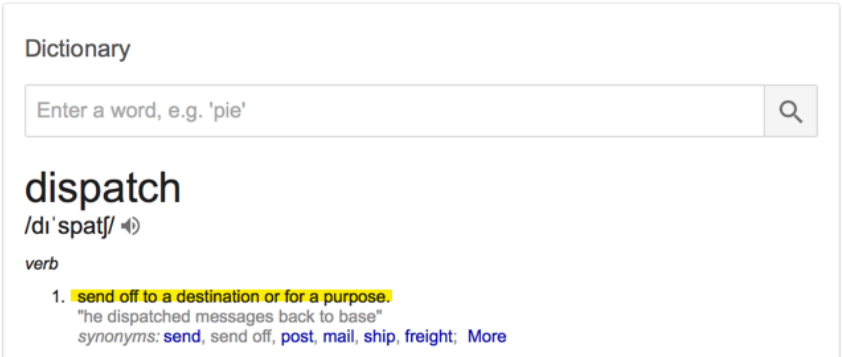
Well, you had an action in mind, but you had to convey that action to someone, the Cashier, which in turn communicated (in whatever way they did) with the vault that holds all of the bank's money.



The same may be said for Redux.

Like you made your action known to the Cashier, you have to do the same in your Redux application. If you want to update the state of your application, you convey your ACTION to the REDUCER - our own Cashier.

This process is mostly called DISPATCHING an ACTION.



Dispatch is just an English word. In this example, and in the Redux world, it is used to mean sending off the action to the reducers.

The REDUCER knows what to do. In this example, It will take your action to `WITHDRAW_MONEY` and ensure you get your MONEY.

In Redux terms, the money you spend is your STATE. So, your reducer knows what to do, and it always returns your NEW STATE.

Hmmm. That wasn't so hard to grasp, right?

And this leads to the last Redux principle:

> *To specify how the state tree is transformed by actions, you write pure reducers.*

As we proceed, I'll explain what a "pure" reducer means. For now, what's important is to understand that, to update the state of your application (like you do with `setState` in React,) your actions must always be sent off (DISPATCHED) to the reducers to get your NEW STATE.

The Redux principle #3.

To specify how the state tree is transformed by actions, you write pure reducers.

https://TheReduxJSBooks.com

With this analogy, you should now have an idea of what the most important Redux actors are: THE STORE, THE REDUCER, and an ACTION.

These three actors are pivotal to any Redux application. Once you understand how they work, the bulk of the deed is done.

Chapter 2: Your First Redux Application

> *"We learn by example and by direct experience because there are real limits to the adequacy of verbal instruction."*
>
> - *Malcom Gladwell*

Even though I have spent ample time explaining the Redux principles in a way you won't forget, verbal instructions have their limits.

To deepen your understanding of the principles, I'll show you an example. Your first Redux application, if you want to call it that.

My approach to teaching is to introduce examples of increasing difficulty. So, for starters, this example is focused on refactoring a simple pure React app to use Redux.

The aim here is to understand how to introduce Redux in a simple React project, and deepen your understanding of the fundamental Redux concepts too.

Ready?

Below is the trivial 'Hello World' React app we will be working with.

Don't laugh it off.

You'll learn to flex your Redux muscles from a "known" concept, *React*, to the "unknown" *Redux*.

The Structure of the React Hello World Application

The React app we'll be working with has been bootstrapped with `create-react-app`. Thus, the structure of the app is one you're already used to.

You may grab the repo from Github if you want to follow along - which I recommend.

There's an `index.js` entry file that renders an `<App />` component to the `DOM`.

The main `App` component comprises of a certain, `<HelloWorld />` component.

This `<HelloWorld />` component that takes in a `tech` prop, and this prop is responsible for the particular technology displayed to the user.

For example, `<HelloWorld tech="React" />` will yield the following:

Also, a `<HelloWorld tech="Redux" />` will yield the following.

Now, you get the gist.

Here's what the `App` component looks like:

src/App.js

```
import React, { Component } from "react";

import HelloWorld from "./HelloWorld";
```

```
class App extends Component {
 state = {
  tech : "React"
}
render() {
  return <HelloWorld tech={this.state.tech}/>
}
}

export default App;
```

Have a good look at the state object.

There's just one field, `tech`, in the state object and it is passed down as `prop` into the `HelloWorld` component as shown below:

```
<HelloWorld tech={this.state.tech}/>
```

Don't worry about the implementation of the `HelloWorld` component - yet. It just takes in a `tech` prop and applies some fancy CSS. That's all.

Since this is focused mainly on Redux, I'll skip the details of the styling.

So, here's the challenge.

How do we refactor our `App` to use `Redux` ?

How do we take away the state object and have it entirely managed by Redux? Remember that Redux is the **state manager** for your app.

Let's begin to answer these questions in the next section.

Revisiting your Knowledge of Redux.

Remember the quote from the official docs ?

> *Redux is a predictable state container for JavaScript apps.*

One key phrase in the above sentence is, **state container**

Technically, you want the STATE of your application to be managed by REDUX.

This is what makes Redux a *state container*.

Your React component state still exists. Redux doesn't take it away.

However, REDUX will efficiently manage your overall APPLICATION STATE. Like a bank vault, it's got a STORE to do that.

For the simple `<App/>` component we've got here, the state object is simple.

Here it is:

```
{
 tech: "React"
}
```

We need to take this out of the `<App />` component state, and have it managed by REDUX.

From an earlier explanation, you should remember the analogy between the BANK VAULT and the REDUX STORE. The Bank Vault keeps money, the Redux store keeps the application state object.

So, what is the first step to refactoring the `<App />` component to use Redux?

Yeah, you got that right.

Remove the component state from within `<App />`.

The Redux STORE will be responsible for managing the App's state. With that being said, we need to remove the current state object from `App/>`

```
import React, { Component } from "react";
import HelloWorld from "./HelloWorld";


class App extends Component {
 // the state object has been removed.
render() {
  return <HelloWorld tech={this.state.tech}/>
}
}


export default App;
```

The solution above is incomplete, but right now, `<App/>` has NO state.

Please install Redux by running `yarn add redux` from the command line interface (CLI). We need the `redux` package to do anything right.

Creating a Redux STORE

If the `<App />` won't manage it's state, then we have to create a Redux STORE to manage our application state.

For a Bank Vault, a couple mechanical engineers were probably hired to create a secure money-keeping facility.

To create a manageable state-keeping facility for our application, we don't need mechanical engineers per se. We'll do so programmatically using some of the APIs Redux avails us.

Here's what the code to create a Redux STORE looks like:

```
import { createStore } from "redux"; //an import from the redux
library

const store = createStore();  // an incomplete solution - for now.
```

First we import the `createStore` factory function from Redux. Then we invoke the function, `createStore()` to create the store.

Now, the `createStore` function takes in a few arguments. The first being a `reducer`

So, a more complete store creation would be represented like this: `createStore(reducer)`

Now, let me explain why we've got a `reducer` in there.

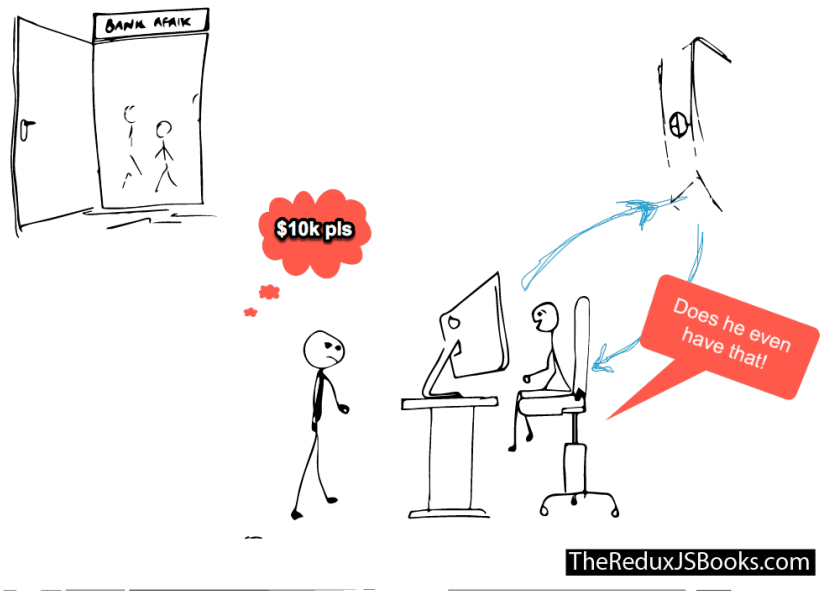
The Store and Reducer Relationship

Back to the bank analogy.

When you go to the bank to make a withdrawal, you meet with the Cashier. After you make your `WITHDRAW_MONEY` intent/action known to the Cashier, they do NOT just hand you the requested money.

No.

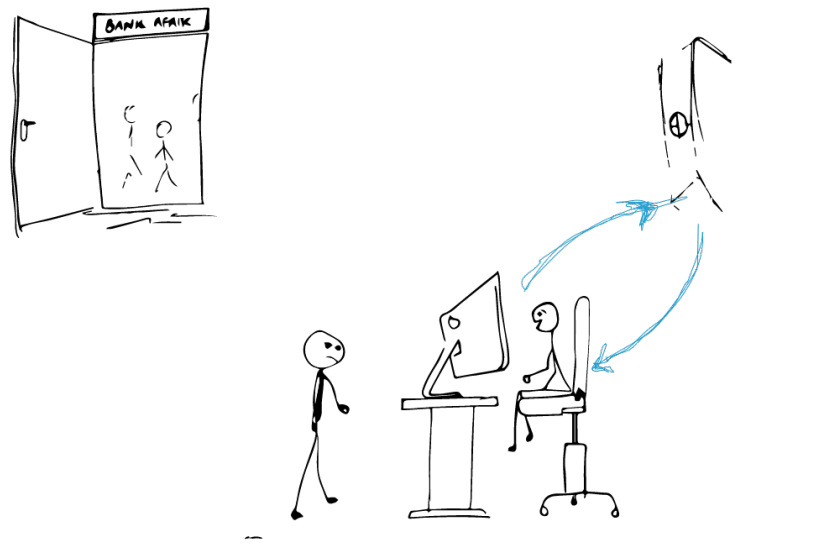
The Cashier first confirms that you have enough money in your account to perform the withdrawal transaction you seek.



The Cashier first makes sure you have the money you say you do.

From the computer, they can see all that - kind of communicating with the Vault , since the Vault keeps all the money in the bank.

In a nutshell, the Cashier and Vault are always in sync. Great buddies!



The same may be said for a Redux `STORE` (our own Vault,) and the Redux `REDUCER` (our own Cashier)

The Store and the Reducer are great buddies. Always in sync.

Why?

The REDUCER always "talks" to the STORE. Just like the Cashier stays in sync with the Vault.

This explains why the creation of the store needs to be invoked with a Reducer, and that is mandatory. The `Reducer` is the only mandatory argument passed into `createStore()`

In the following section we will have a brief look at Reducers and then create a `STORE` by passing the `REDUCER` into the `createStore` factory function.

The Reducer

We will go into greater details pretty soon, but I'll keep this short for now.

When you hear the word, reducer, what comes to your mind?

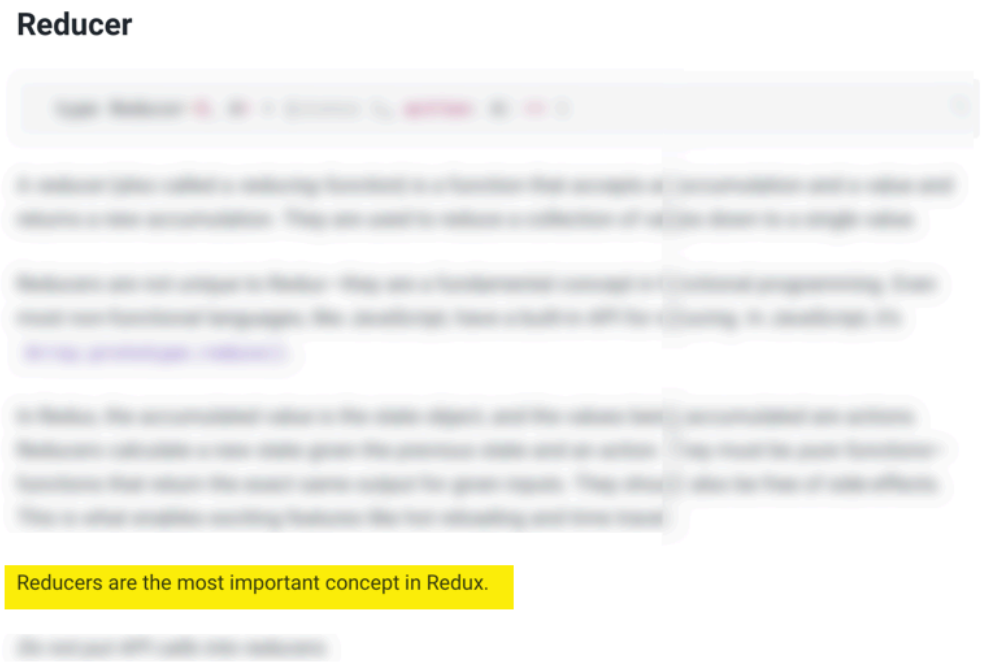
reduce?

Yeah, that's what I thought.

It sounds like *reduce*

Well, according to the Redux official docs:

> *Reducers are the most important concept in Redux.*



Our Cashier is a pretty important person, huh?

So, what's the deal with the Reducer. What does it do?

In more technical terms, a reducer is also called *a reducing function*. You may not have noticed, but you probably already use a reducer - if you're conversant with the Array.reduce() method.

Here's a quick refresher.

Consider the code below.

It is a popular way to get the sum of values in a Javascript Array.

```
let arr = [1,2,3,4,5]

let sum = arr.reduce((x,y) => x + y)

console.log(sum)  //15
```

Under the hood, the function passed into `arr.reduce` is called a `reducer`.

In this example, the reducer takes in two values, an `accumulator` and a `currentValue` where x is the `accumulator` and y, the `currentValue`

In likewise manner, the Redux Reducer is just a function. A function that takes in **two** parameters. The first being the `STATE` of the app, and the other the `ACTION`

Oh my gosh! But where does the `STATE` and `ACTION` passed into the `REDUCER` come from?

When I was learning Redux, I asked myself this question a few times.

Firstly, take a look at the `Array.reduce()` example again:

```
let arr = [1,2,3,4,5]

let sum = arr.reduce((x,y) => x + y)

console.log(sum)  //15
```

The `Array.reduce` method is responsible for passing in the needed arguments, x and y into the function argument, the`reducer` . So, the arguments didn't come out of thing air.

The same may be said for Redux.

The Redux reducer is also passed into a certain method. Guess what is it?

Here you go!

```
createStore(reducer)
```

The `createStore` factory function. There's a little more involved in the process as you'll soon see.

Like, `Array.reduce()`, `createStore()` is responsible for passing the arguments into the reducer.

If you aren't scared of technical stuff, here's the stripped down version of the implementation of `createStore` within the Redux source code.

```
function createStore(reducer) {

    var state;

    var listeners = []
```

```
    function getState() {
        return state
    }

    function subscribe(listener) {
        listeners.push(listener)
        return unsubscribe() {
            var index = listeners.indexOf(listener)
            listeners.splice(index, 1)
        }
    }

    function dispatch(action) {
        state = reducer(state, action)
        listeners.forEach(listener => listener())
    }

    dispatch({})

    return { dispatch, subscribe, getState }
}
```

Don't beat yourself up if you don't get the code above. What I really want to point out is within the `dispatch` function.

Notice how the reducer is called with `state` and `action`

With all that being said, the most minimal code for creating a Redux store is this:

```
import { createStore } from "redux";
```

```
const store = createStore(reducer);   //this has been updated to
include the created reducer.
```

Getting back to the Refactoring Process.

Let's get back to refactoring the *Hello World* React application to use Redux.

If I lost you at any point in the previous section, please read the section just one more time and I'm sure it'll sink. Better still, you can ask me a question.

Okay so here's all the code we have at this point.

```
import React, { Component } from "react";
import HelloWorld from "./HelloWorld";


 import { createStore } from "redux";
 const store = createStore(reducer);


 class App extends Component {
 render() {
   return <HelloWorld tech={this.state.tech}/>
 }
}


export default App;
```

Makes sense?

You may have noticed a problem with this code. See Line 4.

The `reducer` function passed into `createStore` doesn't exist yet.

Now we need to write one. The reducer is just a function, remember?

Create a new directory called, `reducers` and create an `index.js` file in there. Essentially, our reducer function will be in the path, `src/reducers/index.js`

First export a simple function in this file:

```
export default () => {

}
```

Remember, that the reducer takes in two arguments - as established earlier. Right now, we'll concern ourselves with the first argument, `STATE`

Put that into the function, and we have this:

```
export default (state) => {

}
```

Not bad.

A reducer always returns something. In the initial `Array.reduce()`reducer example, we returned the **sum** of the accumulator and current value.

For a Redux Reducer, you always return the NEW STATE of your application.

Let me explain.

After you walk into the bank and make a successful withdrawal, the current amount of money held in the bank's vault for you is no longer the same. Now, if you withdrew $200, you are now short of $200 i.e You account balance minus $200.

Again, the Cashier and Vault remain in sync on how much you now have.

Just like the Cashier, this is exactly how the Reducer works.

Like the Cashier, the Reducer always returns the new state of your application. Just Incase something has changed. We don't want to issue the same bank balance even though a withdrawal action was performed.

We'll get to the internals of to change/update the state later on. For now, blind trust will suffice.

Now, back to the problem at hand.

Since we aren't concerned about changing/updating the state at this point, we will keep NEW STATE being returned as the same `state` passed in.

Here's the representation of this within the reducer:

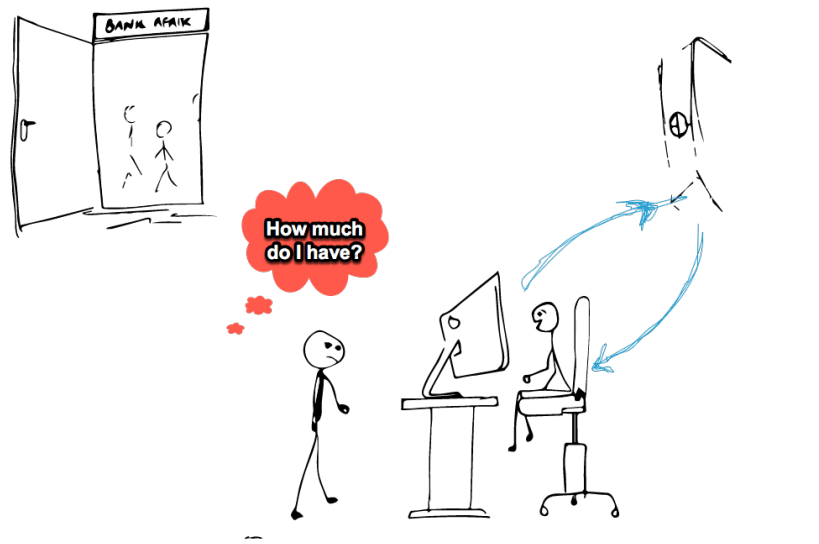
```
export default (state) => {

          return state

}
```

If you go to the bank without performing an action, your bank balance remains the same, right?

Since we aren't performing any `ACTION` or even passing that into the reducer yet, we will just `return` the same `state`

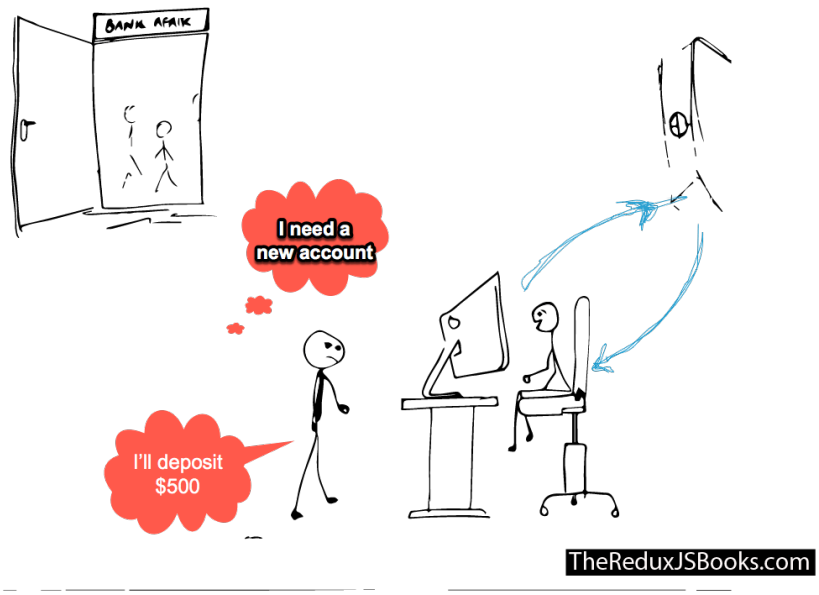
The Second `createStore` Argument.

When you visit the Cashier in the bank, if you asked them for your account balance, they'll look it up and tell it to you.



But how?

When you first created an account with your bank, you either did so with some amount of deposit or not.



Let's call this the initial deposit into your account.

Back to Redux.

In the same way, when you create a redux STORE (our own money keeping VAULT), there's the option of doing so with an initial deposit.

In Redux terms, this is called the `initialState` of the app.

Thinking in code, `initialState` is the second argument passed into the `createStore` function call.

```
const store = createStore(reducer, initialState);
```

Before making any monetary ACTION, If you requested your bank account balance, the INITIAL DEPOSIT will always be returned to you.

Afterwards, anytime you perform any monetary ACTION, this initial deposit will also be updated.

Now, the same goes for Redux.

The object passed in as `initialState` is like the initial deposit to the Vault. This `initialState` will always be returned as the STATE of the application unless you update the state by performing an ACTION.

We will now update the application to pass in an initial state:

```
const initialState = { tech: "React " };

const store = createStore(reducer, initialState);
```

Note how `initialState` is just an object, and it is exactly what we had as the default state in the React App before we began refactoring.

Now, here's all the code we have at this point - with the reducer also imported into `App`

App.js

```
import React, { Component } from "react";

import HelloWorld from "./HelloWorld";

import reducer from "./reducers";

import { createStore } from "redux";


const initialState = { tech: "React " };

const store = createStore(reducer, initialState);


class App extends Component {

 render() {

   return <HelloWorld tech={this.state.tech}/>

 }

 }
```

```
export default App;
```

reducers/index.js

```
export default state  => {
          return state
}
```

If you're coding along and try to run the app now, you'll get an error. Why?

Have a look at the `tech` prop passed into `<HelloWorld />`. It still reads, `this.state.tech`.

There's no longer a state object attached to `<App />`, so that will be `undefined`.

Let's fix that.

The solution is quite simple. Since the STORE now manages the state of our application, this means the application `STATE`object must be retrieved from the STORE. But how?

Whenever you create a store, `createStore()`, the created store has a 3 exposed methods.

One of this is `getState()`.

At any point in time, calling the `getState` method on the created STORE will return the current state of your application.

In our case, `store.getState()` will return the object, `{ tech: "React"}` since this is the `INITIAL STATE` we passed into the `createStore()` method when we created the STORE.

You see how all this come together now?

Hence the `tech` prop will be passed into `<HelloWorld />` as shown below:

App.js

```
import React, { Component } from "react";
```

```
import HelloWorld from "./HelloWorld";
import { createStore } from "redux";

const initialState = { tech: "React " };
const store = createStore(reducer, initialState);

class App extends Component {
 render() {
   return <HelloWorld tech={store.getState().tech}/>
 }
 }
```

Reducers/Reducer.js

```
export default state => {
          return state
}
```

And that is it! You just learned the Redux basics and successfully refactored a simple React app to use Redux.

The React application now has its state managed by Redux. Whatever needs to be gotten from the STATE object will be grabbed from the STORE as shown earlier.

Hopefully, you understood this whole refactoring process.

For a quicker overview, have a look at this Github diff.

With the *Hello World* project, we have taken a good look at some of essential Redux concepts. Even though it's such a tiny project, it provides a decent foundation to build upon!

Possible Gotcha

In the just concluded *Hello World* example, a possible solution you may have come up with for grabbing the state from the STORE may look like this:

```
class App extends Component {
  state = store.getState();
  render() {
    return <HelloWorld tech={this.state.tech} />;
  }
}
```

What do you think? Will this work?

Just as a reminder, the following two ways are correct ways to initialise a React component's state.

(a)

```
class App extends Component {
 constructor(props) {
   super(props);
   this.state = {}
  }
}
```

(b)

```
class App extends Component {
  state = {}
}
```

So, back to answering the question, yes, the solution will work just fine.

`store.getState()` will grab the current state from the Redux STORE.

However, the assignment, `state = store.getState()` will assign the state gotten from Redux to that of the `<App />` component.

By implication, the return statement from `render` i.e `<HelloWorld tech={this.state.tech} />` will be valid.

Note that this reads `this.state.tech` NOT `store.getState().tech`

Even though this works, it is against the ideal philosophy of Redux.

If within the app, you now run, `this.setState()`, the App's state will be updated without the help of REDUX.

This is the default React mechanism, and it isn't what you want. You want the state managed by the Redux STORE to be the single source of truth.

Whether you're retrieving state, as in `store.getState()` or updating/changing state (as we'll cover later) you want that to be entirely managed by Redux, not by `setState()`

Since Redux manages the app's state, all you need to do is feed in state from the Redux STORE as props to any required component.

Another big question you're likely asking yourself is, *why did I have to go through all this stress just to have the state of my App managed by Redux?*

Reducer, Store, createStore blah, blah, blah …

Yeah, I get it.

I felt that way too.

However, consider the fact that you do not just go to the bank and NOT follow a due process for withdrawing your own money. It's your money, but you do have to follow a due process.

The same may be said for Redux.

Redux has it's own "process" of doing things. We've got to learn how that works - and hey, you're not doing badly!

Conclusion and Summary

This chapter has been exciting. We focused mostly on setting a decent foundation for the more interesting things to come.

Here are a few things you learned in this chapter:

- Redux is a predictable **state container** for JavaScript apps.
- The `createStore` factory function from Redux is used to create a Redux STORE
- The `Reducer` is the only mandatory argument passed into `createStore()`
- A REDUCER is just a function. A function that takes in **two** parameters. The first being the `STATE` of the app, and the other being an `ACTION`
- A Reducer always returns the NEW STATE of your application.
- The INITIAL STATE of your application, `initialState` is the second argument passed into the `createStore` function call.
- `Store.getState()` will return the current state of your application. Where `Store` is a valid Redux STORE.

Introducing Exercises

Please, please, please, don't skip the exercises. Especially if you're not confident about your Redux skills and really want to get the best out of this guide.

So, grab your dev hats, and write some code :)

Also, if you want me to pass a feedback on any of your solutions at any point in time, tweet at me with the hashtag *#UnderstandingRedux* and I'll be happy to have a look. I'm not promising to get to every single tweet, but I'll definitely try to.

Once you get the exercises sorted out, I'll see you in the next section.

Remember that a good way to read long content is to break it up into shorter digestible bits. These exercises help you do just that. You take some time off, try to solve the exercises, then you come back to read on. That's an effective way to study.

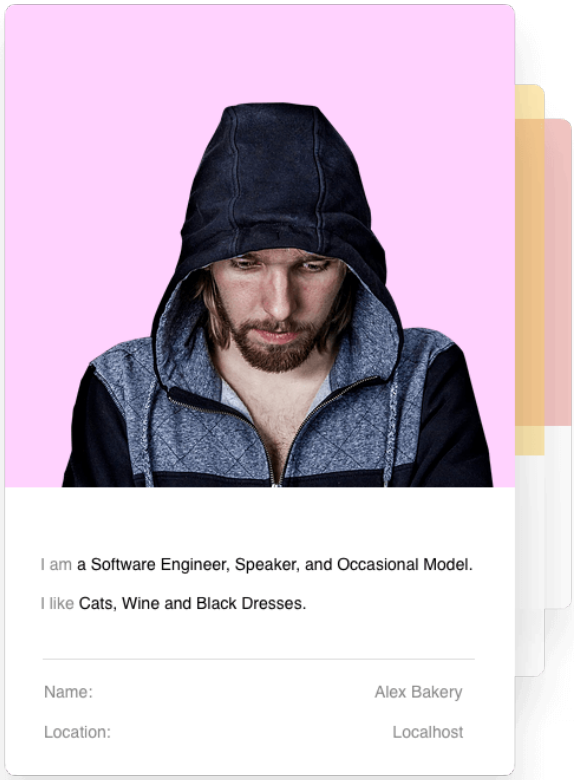
Want to see my solutions to these exercises? I have included the solutions to the exercises in the book package.

So, here's the exercise for this section.

Exercises

(a) Refactor the user card react app to use redux.

In the accompanying code files for the book, you'll find a user card app written solely in React. The state of the App is managed via React. Your task is to move the state to being managed solely by Redux.



I am a Software Engineer, Speaker, and Occasional Model.

I like Cats, Wine and Black Dresses.

Name:	Alex Bakery
Location:	Localhost

Chapter 3 : Understanding State Updates with Actions.

Now that we've got the foundational concepts of Redux discussed, we will begin to do some more interesting things.

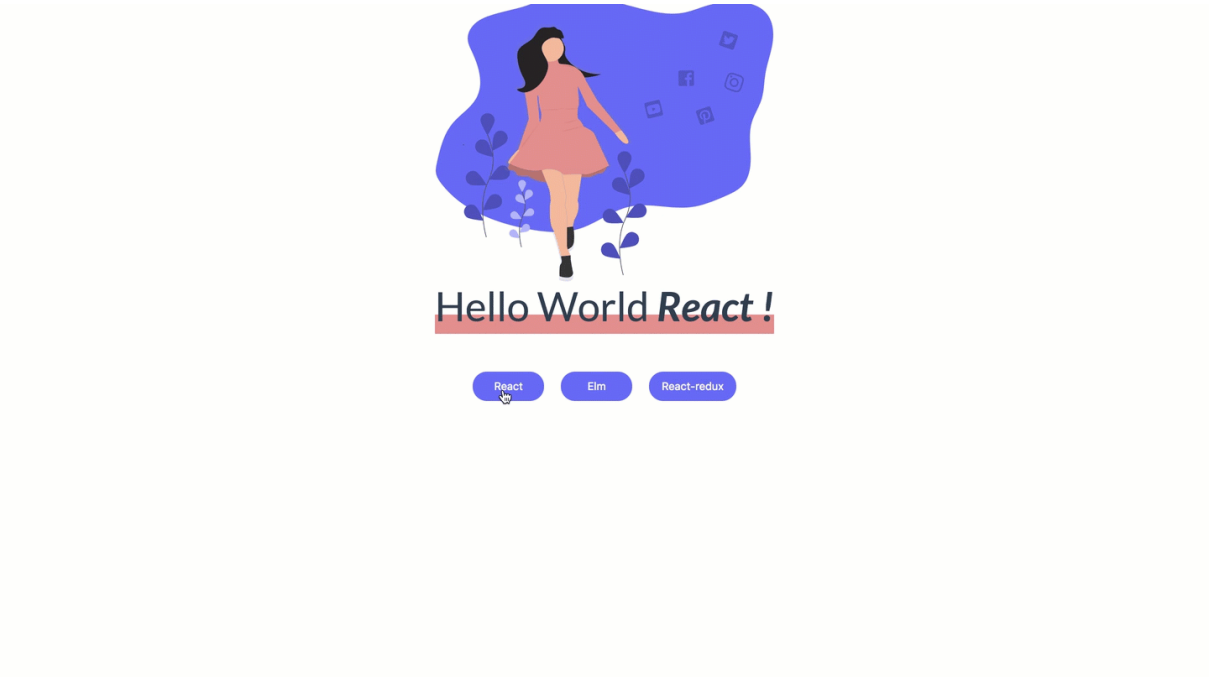
In this chapter, we will continue to learn by doing as I walk you through another project - while explaining every process in detail.

So, what project are going to work on this time?

I've got the perfect one.

Please, consider the mockup below:

Oh, it looks just like the previous example - but with a few changes. This time we will take account of user actions. When we click any of the buttons, we want to update the state of the application as shown in the GIF below:



Here's how this is different from the previous example; in this scenario, the user is performing certain actions that influence the state of the application. In the former example, all we did was display the initial state of the app with no user actions taken into consideration.

What is a Redux Action?

When you walk into a bank, the Cashier receives your action i.e intent for coming into the bank. In our previous example, `WITHDRAWAL_MONEY` . The only way money leaves the bank Vault, is if you make your action i.e intent known to the Cashier.

Now, the same goes for the Redux Reducer.

Unlike `setState()` in pure React, the only way you update the state of a Redux application, is if you make your intent known to the REDUCER.

But how?

By dispatching actions!

In the real world, you know the exact action you want to perform. You could probably write that down in a slip and hand it over to the Cashier.

This works almost the same way with Redux. The only challenge is, how do you describe an action in a Redux app? Definitely not by speaking over the counter or writing it down in a slip.

Well, there's good news.

An action is accurately described with a plain Javascript object. Nothing more.

There's just one thing to be aware of. An action MUST have a `type` field. This field describes the intent of the action.

In the bank story, If we were to describe your action to the bank, it'd look like this:

```
{
  type: "withdraw_money"
}
```

That's all, really.

A Redux action is described as a plain object.

Please have a look at the action above.

Do you think only the `type` field accurately describes your supposed action to make a withdrawal at a bank?

Hmmm. I don't think so. How about the amount of money you want to withdraw?

Many times your action will need some extra data for a complete description. Consider the action below. I argue that this makes for a better described action.

```
{
  type: "withdraw_money",
  amount: "$4000"
}
```

Now, there's sufficient information describing the action. For example sake, ignore every other detail the action may include, such as your bank account number etc.

Other than the `type` field, the structure of your Redux Action is really up to you.

However, a common approach is to have a `type` field and `payload` field as shown below:

```
{
  type: " ",
  payload: {}
}
```

The `type` field describes the action, and every other required data/information that describes the action is put in the `payload` object.

For example:

```
{
  type: "withdraw_money",
  payload: {
     amount: "$4000"
  }
}
```

So, yeah! That's what an action is.

Handling Responses to Actions in the Reducer

Now that you successfully understand what an action is, it is important to see how they become useful i.e in a practical sense.

Earlier, I did say that a reducer takes in *two* arguments. One, `state`, the other, `action`.

Here's a simple Reducer looks like:

```
function reducer(state, action) {

  //return new state

}
```

The `action` is passed in as the second parameter to the Reducer, however, we've done nothing with it within the function itself.

To handle the actions passed into the reducer, you typically write a `switch` statement within your reducer, like this:

```
function reducer (state, action) {

      switch (action.type) {

             case "withdraw_money":

                   //do something

                   break;

            case "deposit-money":

                    //do something

                   break;

            default:

                   return state;

                    }

}
```

Some people seem not to like the `switch` statement, but it's basically an `if/else` for possible values on a single field.

The code above will `switch` over the action `type` and do something based on the type of action passed in. Technically, the *do something* bit is required to return a new state.

Let me explain further.

Assume that you had 2 hypothetical buttons, button #1 and button #2 on a certain webpage, and your state object looked something like this:

```
{
        isOpen: true,
        isClicked: false,
  }
```

When button #1 is clicked, you want to toggle the `isOpen` field. In the context of a React app, the solution is simple. As soon as the button is clicked, you would do this:

```
this.setState({isOpen: !this.state.isOpen})
```

Also, let's assume that when #2 is clicked, you want to update the `isClicked` field. Again, the solution is simple, and along the lines of this:

```
this.setState({isClicked: !this.state.isClicked})
```

Good.

With a Redux app, you can't use `setState()` to update the state object managed by Redux.

You have to dispatch an action first.

Let's assume the actions are as below:

#1 :

```
{
      type: "is_open"
}
```

#2 :

```
 {
      type: "is_clicked"
}
```

In a Redux app, **every action flows through the reducer.** All of them. So, in this example, both action #1 and action #2 will pass through the same reducer.

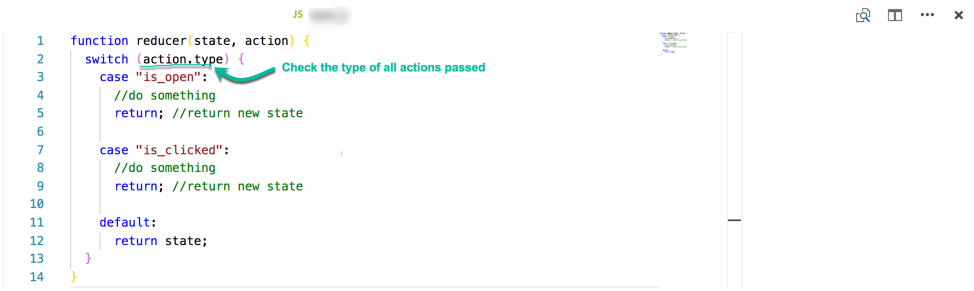
In this case, how does the reducer differentiate each of them?

Yeah, you guessed right.

By switching over the `action.type` we can handle both actions without hassles.

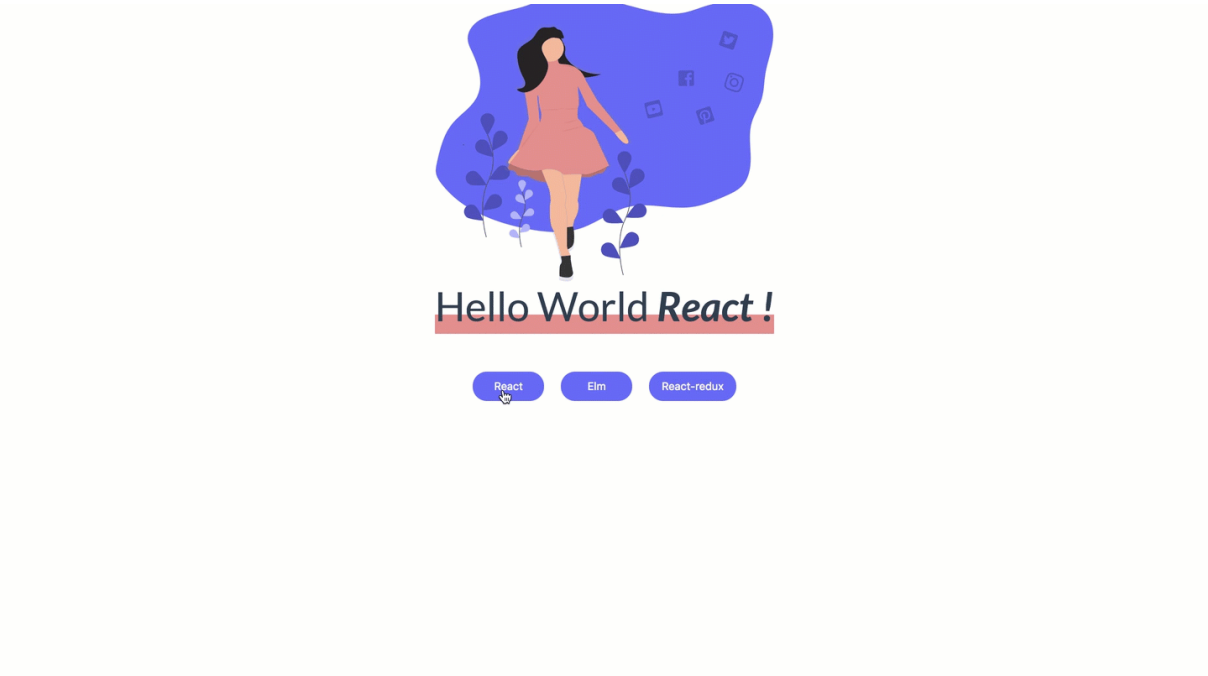
Here is what I mean:

```
function reducer (state, action) {
      switch (action.type) {
            case "is_open":
                  return;  //return new state
            case "is_clicked":
                  return; //return new state
            default:
            return state;
      }
}
```



Now you see why the `switch` statement is useful. All actions will flow through the reducer. Thus, it is important to handle each action type separately.

In the next section, we will continue with the task of building the mini app below:



Examining the Actions in the Application

As earlier explained, whenever there's an intent to update the application state, an `action` must be dispatched.

Whether that intent is initiated by a user click, or a timeout event or even an Ajax request, the rule remains the same. You have to dispatch an action.

The same goes for this application.

Since we intend to update the state of the application, whenever any of the buttons is clicked, we must dispatch an action.

Firstly, let's describe the actions.

Give it a try and see if you get it.

Here's what I came up with:

For the React button:

```
{
    type: "SET_TECHNOLOGY",
    text: "React"
}
```

For the React-Redux button:

```
{
    type: "SET_TECHNOLOGY",
```

```
    text: "React-redux"
  }
```

And finally:

```
{
   type: "SET_TECHNOLOGY",
  text: "Elm"
}
```

Easy, right?

Note that the three actions have the same `type` field. This is because the the three buttons all do the same thing. If they were customers in a bank, then they'd all be depositing money, but different amounts of money. The `type` of action will then be `DEPOSIT_MONEY` but with different `amount` field.

Also, you'll notice that the action type is all written in capital letters. That was intentional. It's not compulsory, but it's a pretty popular style in the Redux community.

Hopefully, you now understand how I came up with the actions.

Introducing Action Creators

Take a look at the actions we created above. You'll notice that we are repeating a few things.

For one, they all have the same `type` field. If we had to dispatch these actions in multiple places, we'll have to duplicate them all over the place. That's not so good. Especially because it's a good idea idea to keep your code *DRY*.

Can we do something about this?

Sure!

Welcome, Action Creators.

Redux has all these fancy names, eh? Reducers, Actions, and now, Action Creators :)

Let me explain what those are.

Action Creators are simply functions that help you create actions. That's all. They are functions that return action objects.

In our particular example, we could create a function that will take in a `text` parameter and return an action, like this:

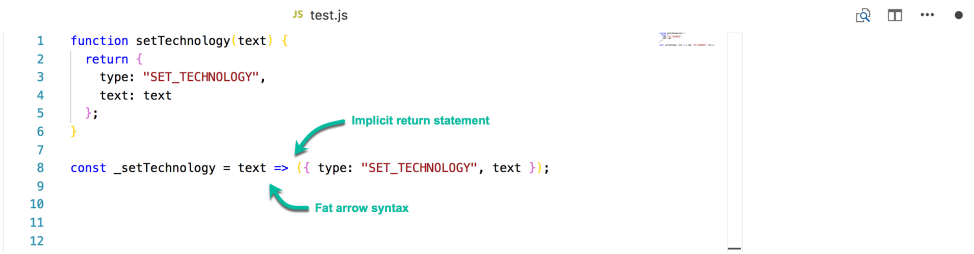
```
export function setTechnology (text) {
  return {
     type: "SET_TECHNOLOGY",
     tech: text
   }
}
```

Now we don't have to bother about duplicating code everywhere. We can just call the `setTechnology` action creator at any time, and we'll get an action back!

What a good use of functions.

Using ES6, the action creator we created above could be simplified to this:

```
const setTechnology = text => ({ type: "SET_TECHNOLOGY", text });
```



Now, that's done.

Bringing Everything Together

All important components required to build the more advanced Hello World app have been discussed in isolation in the earlier sections.

Now, let's put everything together and build the app. Excited?

Firstly, let's talk about folder structure.

When you get to a bank, the Cashier likely sits in their own cubicle/office. The Vault is also kept safe in a secure room. For good reasons, things feel a little more organised that way. Everyone in their own space.

The same may be said for Redux.

It is a common practice to have the major actors of a redux app live within their own folder/directory.

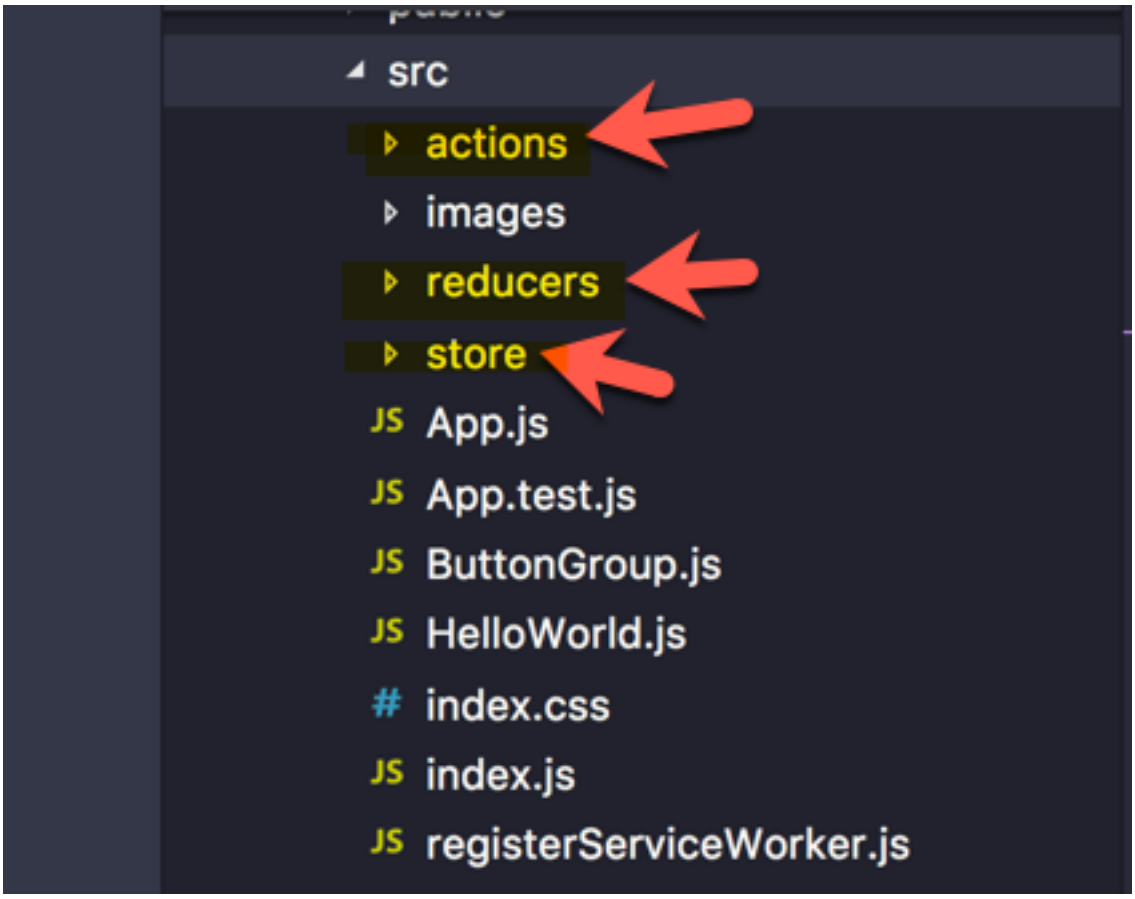
By actors, I mean, the `reducer`, `actions`,and `store`.

It is common to create 3 different folders within your app directory, and name it after these actors.

This isn't a must - and inevitably, you decide how you want to structure your project. For big applications though, this is certainly a pretty decent practice.

We'll now refactor the current app directories we have. Create a few new directories/folders. One called `reducers`, another, `store`, and the last one, `actions`

You should now have a component structure that looks like this:



In each of the folders, create an `index.js` file. This will be the entry point each of the Redux actors (reducers, store and actions). I call them actors - like movie actors. They are the major components of a Redux system.

Now, we'll refactor the previous app from *Chapter 2: Your First Redux Application*, to use this new directory structure.

store/index.js

```
import { createStore } from "redux";
import reducer from "../reducers";


const initialState = { tech: "React " };
export const store = createStore(reducer, initialState);
```

This is just like we had before. The only difference is that the store is now created in its own `index.js` file i.e. having separate cubicles/offices for the different Redux actors.

Now, if we need the store anywhere within our app, we can safely import the store as in, `import store  from "./store";`

With that being said, the `App.js` file for this particular example is slightly different from the former.

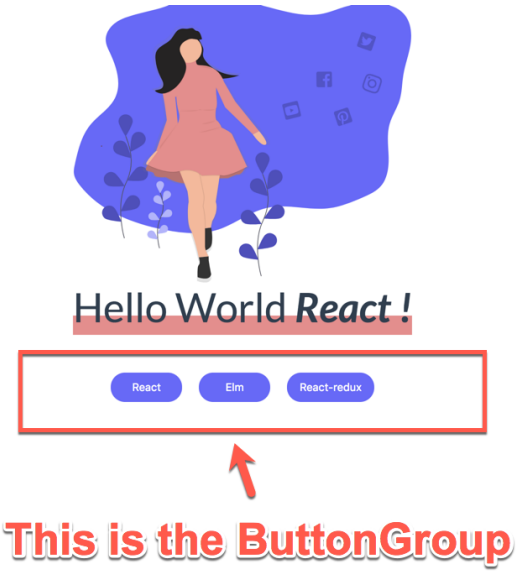
App.js

```
import React, { Component } from "react";

import HelloWorld from "./HelloWorld";

import ButtonGroup from "./ButtonGroup";

import { store } from "./store";


class App extends Component {

  render() {

    return [

      <HelloWorld key={1} tech={store.getState().tech} />,

      <ButtonGroup key={2} technologies={["React", "Elm", "React-
redux"]} />

    ];

  }

}


export default App;
```

What is different?

In line 4, the store is imported from it's own 'cubicle'. Also, there's now a `<ButtonGroup />` component that takes in an array of technologies and spits out

buttons. The `ButtonGroup` component handles the rendering of the three buttons below the "Hello World" text.



Also, you may notice that the `App` component returns an array. That's a `React 16` goodie. With React 16, you don't have to wrap adjacent `JSX` elements in a `div`. You can use an array if you want - but pass in a key prop to each element in the array.

```
class App extends Component {
  render() {
    return [
      <HelloWorld key={1} tech={store.getState().tech} />,
      <ButtonGroup key={2} technologies={["React", "Elm", "React-redux"]} />
    ];
  }
}

export default App;
```

That is it for the `App.js` component.

The implementation of the `ButtonGroup` component is quite simple. Here it is:

ButtonGroup.js

```
import React from "react";


const ButtonGroup = ({ technologies }) => (
  <div>
    {technologies.map((tech, i) => (
      <button
        data-tech={tech}
        key={`btn-${i}`}
        className="hello-btn"
      >
        {tech}
      </button>
    ))}
  </div>
);


export default ButtonGroup;
```

`ButtonGroup` is a stateless component that takes in an array of technologies, denoted by `technologies.`

It loops over this array using `map` and renders a `<button></button` for each of the tech in the array.

In this example, the buttons array passed in is, `["React", "Elm", "React-redux"]`

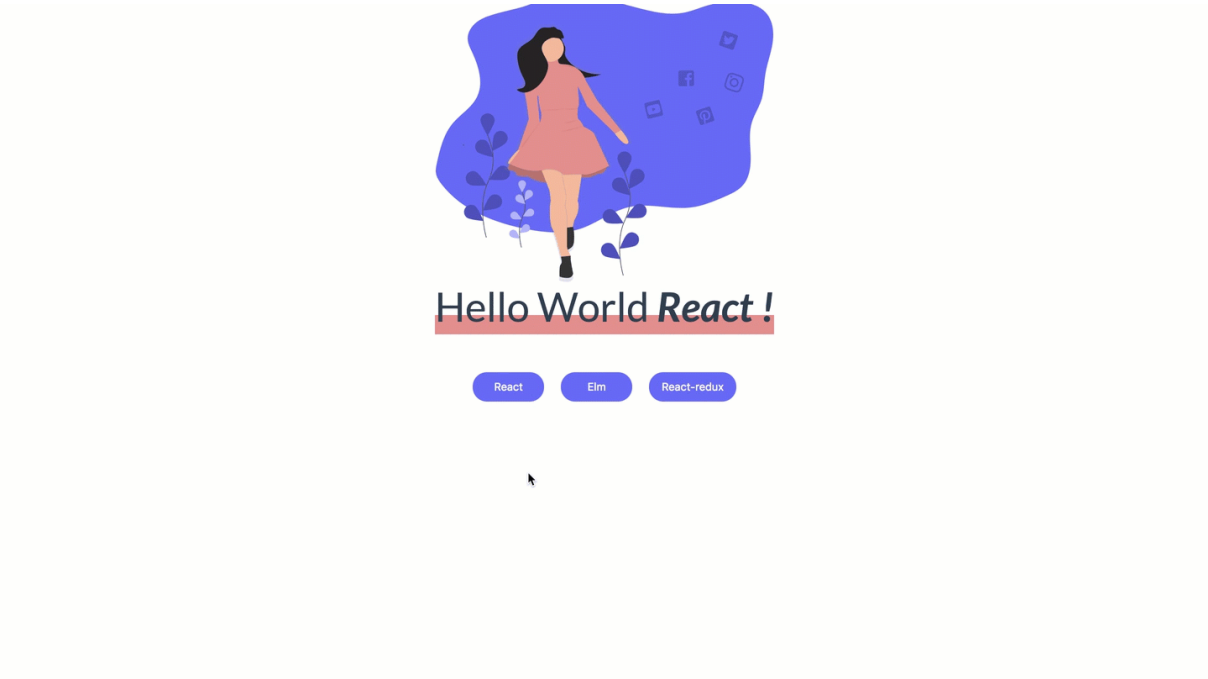
The buttons generated have a few attributes. There's the obvious `className` for styling purposes. There's `key` to prevent the pesky react warning about rendering multiple items without a key prop. Gosh, that error haunts me every time :(

Lastly, there's a `data-tech` attribute on each `button` too. This is called a [data attribute]. It is a way to store some extra information that doesn't have any visual representation. It makes it slightly easier to grab certain values off of an element.

A completely rendered button will look like this:

```
<button
  data-tech="React"
  key="btn-1"
  className="hello-btn"> React </button>
```

Right now, everything renders correctly, but upon clicking the button, nothing happens yet.



Well, that's because we haven't provided any click handlers yet. Let's do that now.

Within the render function, let's set up an `onClick` handler:

```
<div>
    {technologies.map((tech, i) => (
      <button
```

```
          data-tech={tech}
          key={`btn-${i}`}
          className="hello-btn"
          onClick={dispatchBtnAction}
        >
          {tech}
        </button>
      ))}
    </div>
```

Good. Let's write the `dispatchBtnAction` now.

Don't forget that the sole aim of this handler is to dispatch an action when a click has happened.

For example, if you click the react button, dispatch the action:

```
{
    type: "SET_TECHNOLOGY",
    tech: "React"
}
```

If you click the React-Redux button, dispatch this action:

```
{
   type: "SET_TECHNOLOGY",
   tech: "React-redux"
}
```

So, here's the `dispatchBtnAction` function.

```
function dispatchBtnAction(e) {
  const tech = e.target.dataset.tech;
```

```
    store.dispatch(setTechnology(tech));
  }
```

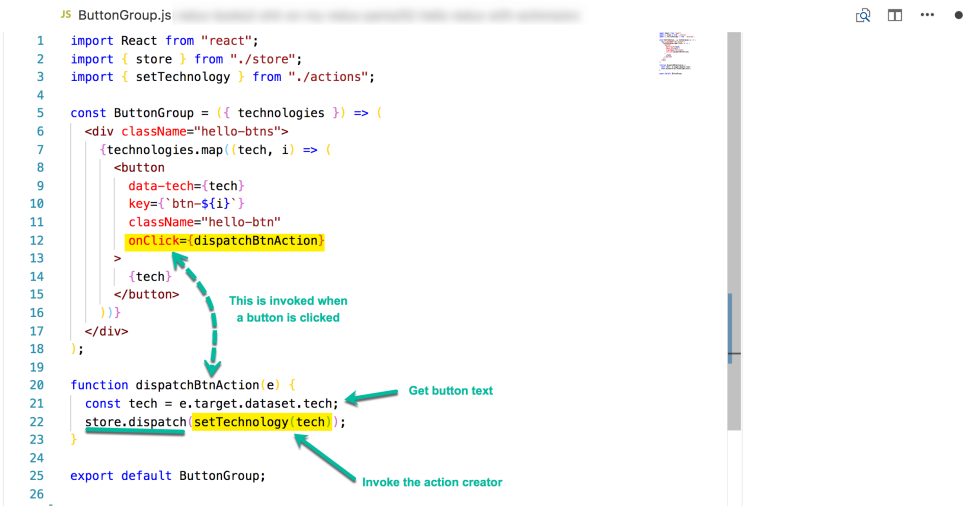
Hmmm. Does the code above make sense to you?

`e.target.dataset.tech` will get the data attribute set on the button, `data-tech` Hence, `tech` will hold the value of the text.

`store.dispatch()` is how you dispatch an action in Redux, and `setTechnology()` is the action creator we wrote earlier!

```
function setTechnology (text) {
  return {
     type: "SET_TECHNOLOGY",
     text: text
   }
}
```

I have gone ahead to add a few comments in the illustration below, just so you understand the code.



Like you already know, `store.dispatch` expects an action object - nothing else. Don't forget the `setTechnology` action creator. It takes in the button text and returns the required action.

Also, the `tech` of the button is grabbed from the dataset of the button. You see, that's exactly why I had a `data-tech` attribute on each button. So we could easily grab the tech off each of the buttons.

Now we're dispatching the right actions. Can we ascertain that this works as expected now?

Actions Dispatched. Does this Thing Work?

Firstly, here's a short quiz question. Upon clicking a `button` and consequently dispatching an action, what happens next within Redux? i.e which of the Redux actors come into play?

Simple. When you hit the bank with a `WITHRAW_MONEY` action, who do you go to? The Cashier, yes.

Same things here. The actions, when dispatched, flow through the reducer.

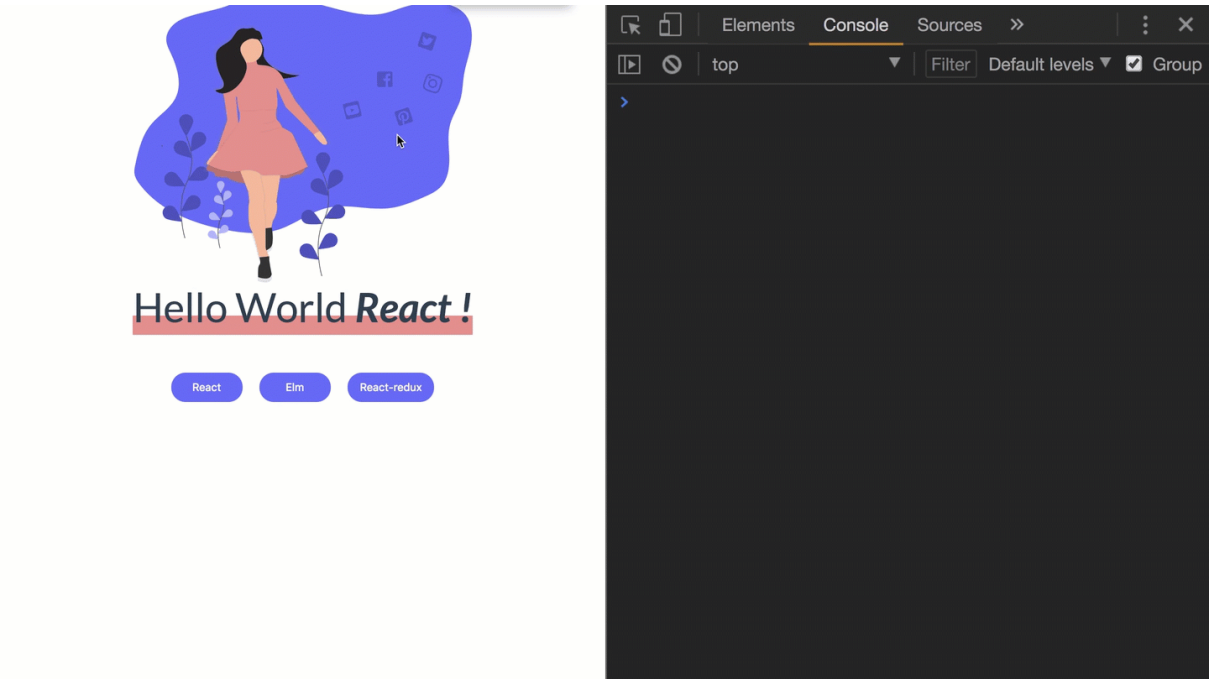
To prove this, I'll log whatever action comes into the reducer.

reducers/index.js

```
export default (state, action) => {
  console.log(action);
  return state;
};
```

The reducer then returns the new sate of the app. In our particular case, we're just returning the same initial `state`

With the `console.log()` in the reducer, let's have a look at what happens when we click.



Oh, yeah!

The actions are logged when the buttons are clicked. Which proves that the actions indeed go through the Reducer. Amazing!

There's one more thing though. As soon as the app starts, there's a weird action being logged as well. It looks like this:

```
{type: "@@redux/INITu.r.5.b.c"}
```

What's that?

Well, do not concern yourself so much about that. It is an action passed by Redux itself when setting up your app. It is usually called the Redux init action, and it is passed into the reducer when Redux initialises your application with the initial state of the app.

Now, we are sure that the actions indeed pass through the Reducer. Great!

While that's exciting, the only reason you go to the Cashier with a withdrawal request is because you want money. If the Reducer isn't taking the action we pass in and doing something with our action, of what value is it?

Making the Reducer Count.

Up until now, the reducer we've worked it hasn't done anything particularly smart. It's like a Cashier who is new to the job and does nothing with our `WITHDRAW_MONEY` intent.

What exactly do we expect the reducer to do?

For now, here's the `initialState` we passed into `createStore` when the STORE was created.

```
const initialState = { tech: "React" };

export const store = createStore(reducer, initialState);
```

When a user clicks any of the buttons, thus passing an action to the reducer, the new state we expect the reducer to return should have the action text in there!

Here's what I mean.

Current state is `{ tech: "React"}`

Given a new action of type `SET_TECHNOLOGY`, and text, `React-Redux` :

```
{
        type: "SET_TECHNOLOGY",
        text: "React-Redux"
}
```

What do you expect the new state to be?

Yeah, `{tech: "React-Redux"}`

The only reason we dispatched an action is because we want a new application state!

Like I mentioned earlier, the common way to handle different action types within a reducer is to use the Javascript `switch` statement as shown below:

```
export default (state, action) => {
  switch (action.type) {
    case "SET_TECHNOLOGY":
      //do something.

    default:
      return state;
  }
};
```

Now we `switch` over the action type. But why?

Well, if you went to see a Cashier, you could have many different actions in mind.

You could want to `WITHDRAW_MONEY`, or `DEPOSIT_MONEY` or maybe just `SAY_HELLO`.

The Cashier is smart, so they take in your action and responds based on your intent.

This is exactly what we're doing with the Reducer.

The `switch` statement checks the `type` of the action. Hey,

What do you want to do? Withdraw, deposit, whatever…

After that, we then handle the known `cases` we expect. For now, there's just one `case` which is `SET_TECHNOLOGY`.

And by default, be sure to just return the `state` of the app.

So far so good.

The Cashier (Reducer) now understands our action. However, they aren't given us any money (state) yet.

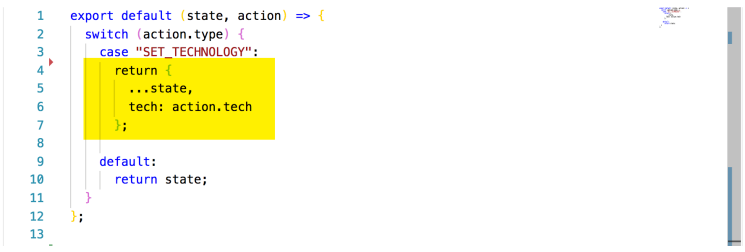
Let's do something within the `case`.

Here's the updated version of the reducer. One that actually gives us *money* :)

```
export default (state, action) => {
  switch (action.type) {
    case "SET_TECHNOLOGY":
      return {
        ...state,
        tech: action.tech
      };

    default:
      return state;
  }
};
```

Aw, yeah!

You see what I'm doing there?



I'll explain what's going on in the next section.

Never Mutate State Within the Reducers.

When returning `state` from reducers, there's something that may put you off at first. However, if you already write good ReactJS code , then you should be familiar with this already.

You should not mutate the `state` received in your Reducer. Instead, you should always return a new copy of the state.

Technically, you should never do this:

```
export default (state, action) => {
  switch (action.type) {
    case "SET_TECHNOLOGY":
      state.tech = action.tech;
      return state;

    default:
      return state;
  }
};
```

This is exactly why the reducer I've written returned this:

```
return {
      ...state,
      tech: action.tech
};
```

Instead of mutating (or changing) the state received from the reducer, I am returning a **new** object. This object has all the properties of the previous state

object. Thanks to the ES6 spread operator, `...state`. However, the `tech` field is updated to what comes in from the action, `action.text`

Also, every Reducer you write should be a pure function with no side-effects I.e No API calls or updating a value outside the scope of the function.

Got that?

Hopefully, yes.

Now, the Cashier isn't ignoring our actions. They're in fact giving us cash now!

After doing this, click the buttons. Does it work now?

Gosh it still this doesn't work. The text doesn't update.

What in the world is wrong this time?

Subscribing to Store Updates

When you visit the bank, let the Cashier know your intended `WITHDRAWAL` action, and successfully receive your money, what next?

Most likely, you will receive an alert via email/text or some other mobile notification saying you have performed a transaction, and your new account balance is so and so.

If you don't receive mobile notifications, you'll definitely receive some sort of 'personal receipt' to show that a successful transaction was carried out on your account.

Okay, note the flow. An action was initiated, you received your money, you got an alert for a successful transaction.

We seem to be having a problem with our Redux code.

An action has been successfully initiated, we've received money (state), but hey, where's the alert for a successful state update?

We've got none.

Well, there's a solution. Where I come from, you subscribe to receive transaction notifications from the bank either by email/text.

The same is true for Redux. If you want the updates, you've got to subscribe to it.

But how?

The Redux store, whatever store you create has a `subscribe` method called like this: `store.subscribe()`

Well named function, if you ask me!

The argument passed into `store.subscribe()` is a function, and it will be invoked whenever there's a state update.

For what it's worth, please remember that the argument passed into `store.subscribe()` should be a **function**. Okay?

Now let's take advantage of this.

Think about it. After the state is updated, what do we want or expect? We expect a re-render, right?

So, state has been updated. *Redux, please, re-render the app with the new state values.*

Let's have a look at where the app is being rendered in `index.js`

Here's what we've got.

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

This is the line that renders the entire application. It takes the `App/>` component and renders it in the DOM. The `root` ID to be specific.

First, let's abstract this into a function.

See this:

```
const render = function() {
```

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

Since this is now within a function, we have to invoke the function to render the app.

```
const render = function() {
   ReactDOM.render(<App />, document.getElementById("root")
}
render()
```

Now, the `<App />` will be rendered just like before.

Using some ES6 goodies, the function can be made simpler.

```
const render = () => ReactDOM.render(<App />,
document.getElementById("root"));


render();
```

Having the rendering of the `<App/>` wrapped within a function means we can now subscribe to updates to the store like this:

```
store.subscribe(render);
```

Where `render` is the entire render logic for the `<App />` - the one we just refactored.

You understand what's happening here, right?

Anytime there's a successful update to the store, the `<App/>` will now be re-rendered with the new state values.

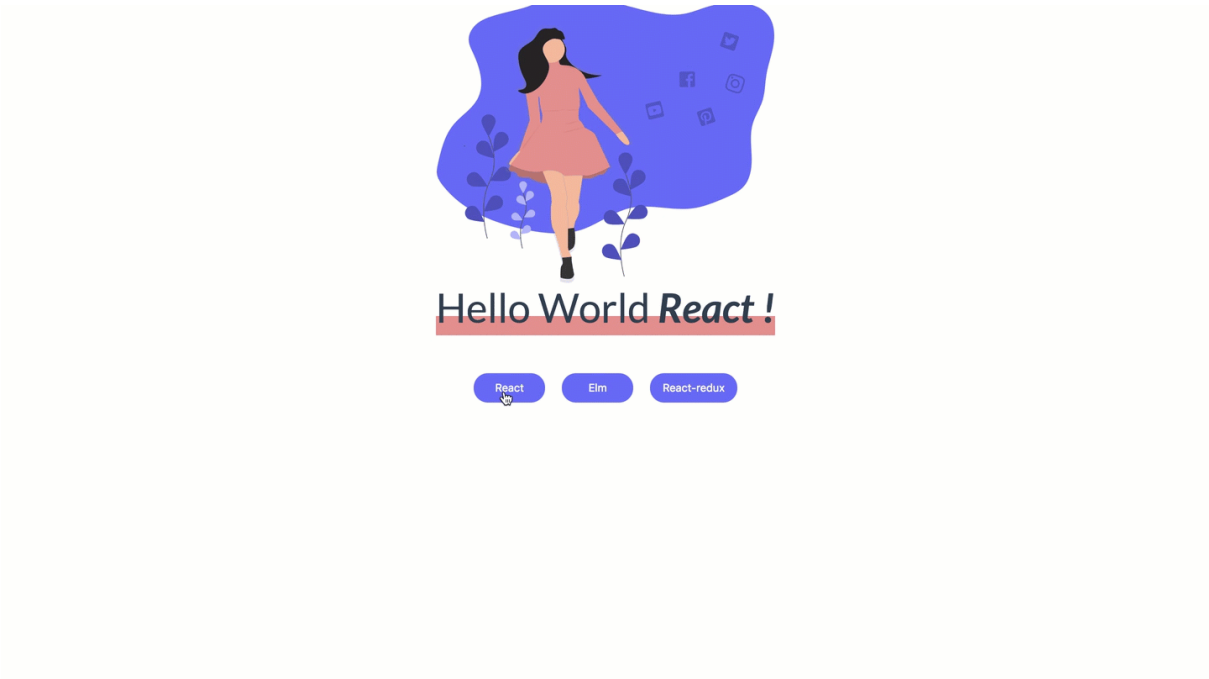
For clarity, here's the <App/> component:

```
class App extends Component {
```

```
  render() {
    return [
      <HelloWorld key={1} tech={store.getState().tech} />,
      <ButtonGroup key={2} technologies={["React", "Elm", "React-
redux"]} />
    ];
  }
}
```

Whenever a re-render occurs, `store.getState()` on line 4 will now fetch the updated state.

Let's see if the app now works as expected.



Hell yeah! This works, and I knew we could do this!

We are successfully dispatching an action, receiving money from the Cashier, and then subscribing to receive notifications. Perfect!

Important Note on Using store.subscribe()

There are a few caveats to using `store.subscribe()` as we've done here. It's a low-level Redux API.

In production, and largely for performance reasons, you'll likely use bindings such as *react-redux* when dealing with larger apps. For now, it is safe to continue using `store.subscribe()` for our learning purposes.

In one of the most beautiful PR comments I've seen in a long time, *Dan Abramov,* in one of the Redux application examples said:

> *The new Counter Vanilla example is aimed to dispel the myth that Redux requires Webpack, React, hot reloading, sagas, action creators, constants, Babel, npm, CSS modules, decorators, fluent Latin, an Egghead subscription, a PhD, or an Exceeds Expectations O.W.L. level.*

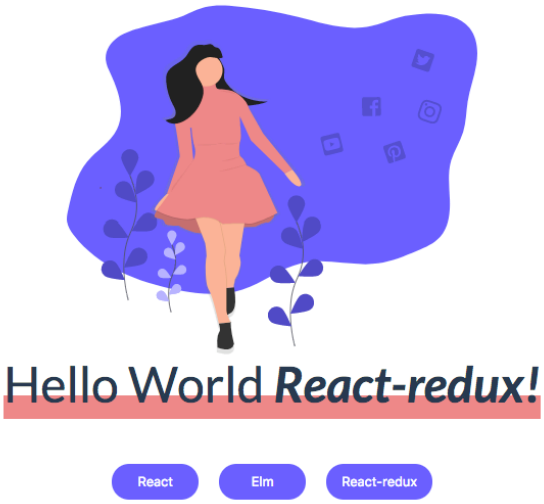
I believe the same.

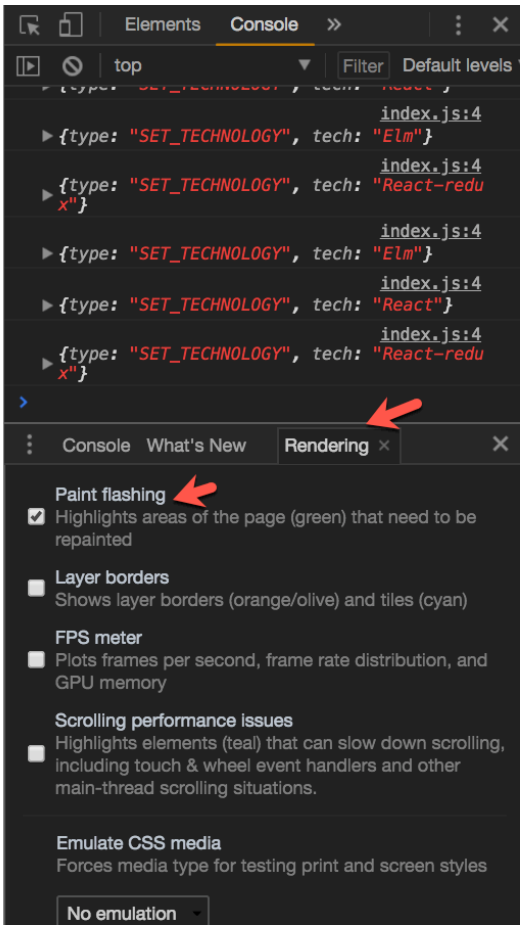
When learning Redux, especially if you're just starting out, you can do away with as many *"extras"* as possible.

Learn to **walk** first, then you can **run** as much as you want.

Okay, Are We Done Yet?

Yeah, we're done, technically. However, there's one more thing I'd love to show you. I'll bring up my browser Devtools and enable paint-flashing.





Now, as we click and update the state of the app, note the green flashes that appear on the screen. The green flashes represent parts of the app being re-painted i.e re-rendered by the Browser engine.

Have a look :

As you can see, even though it appears that the `render` function is invoked every time a state update is made, not the entire app is re-rendered. Just the component with a new state value is re-rendered. In this case, the `<HelloWorld/>` component.

One more thing.

If the current state of the app renders, `Hello World React`, clicking the `React` button again doesn't re-render since the state value is the same.

Good!

This is the React Virtual DOM Diff algorithm at work here. If you know some React, you must have heard this before.

So, yeah. We're done with this! I'm having so much fun explaining this. I hope you are enjoying the read too.

Conclusion and Summary

For a supposedly simple application, this chapter was longer than you probably anticipated. But that's fine. You're now equipped with even greater knowledge on how Redux works.

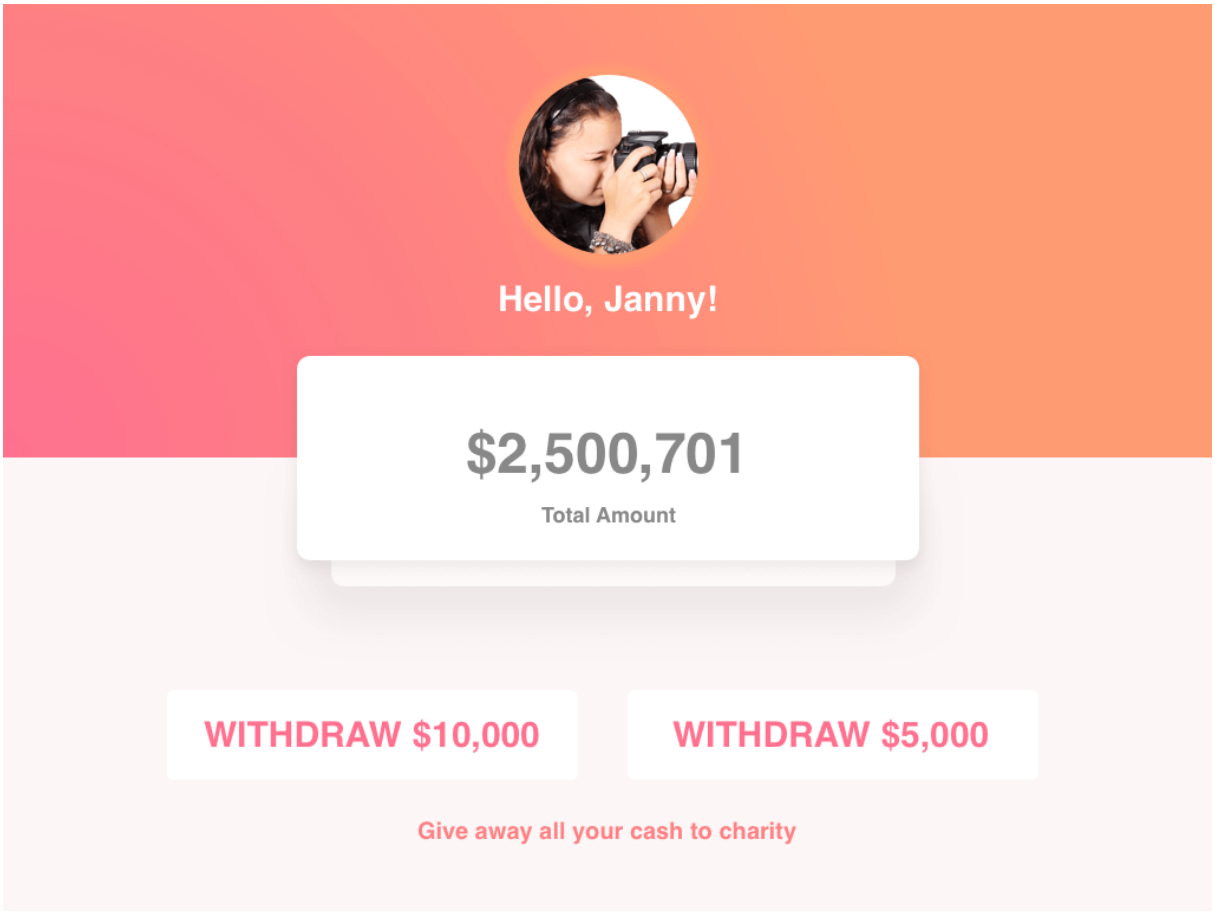
Here are a few things you learned in this chapter:

- Unlike `setState()` in pure React, the only way you update the state of a Redux application is by dispatching an action.
- An action is accurately described with a plain Javascript object, but it must have a `type` field.
- In a Redux app, every action flows through the reducer. All of them.
- By using a `switch` statement, you can handle different action types within your Reducer.
- Action Creators are simply functions that return action objects
- It is a common practice to have the major actors of a redux app live within their own folder/directory.
- You should not mutate the `state` received in your Reducer. Instead, you should always return a new copy of the state.
- To subscribe to store updates, use the `store.subscribe()` method.

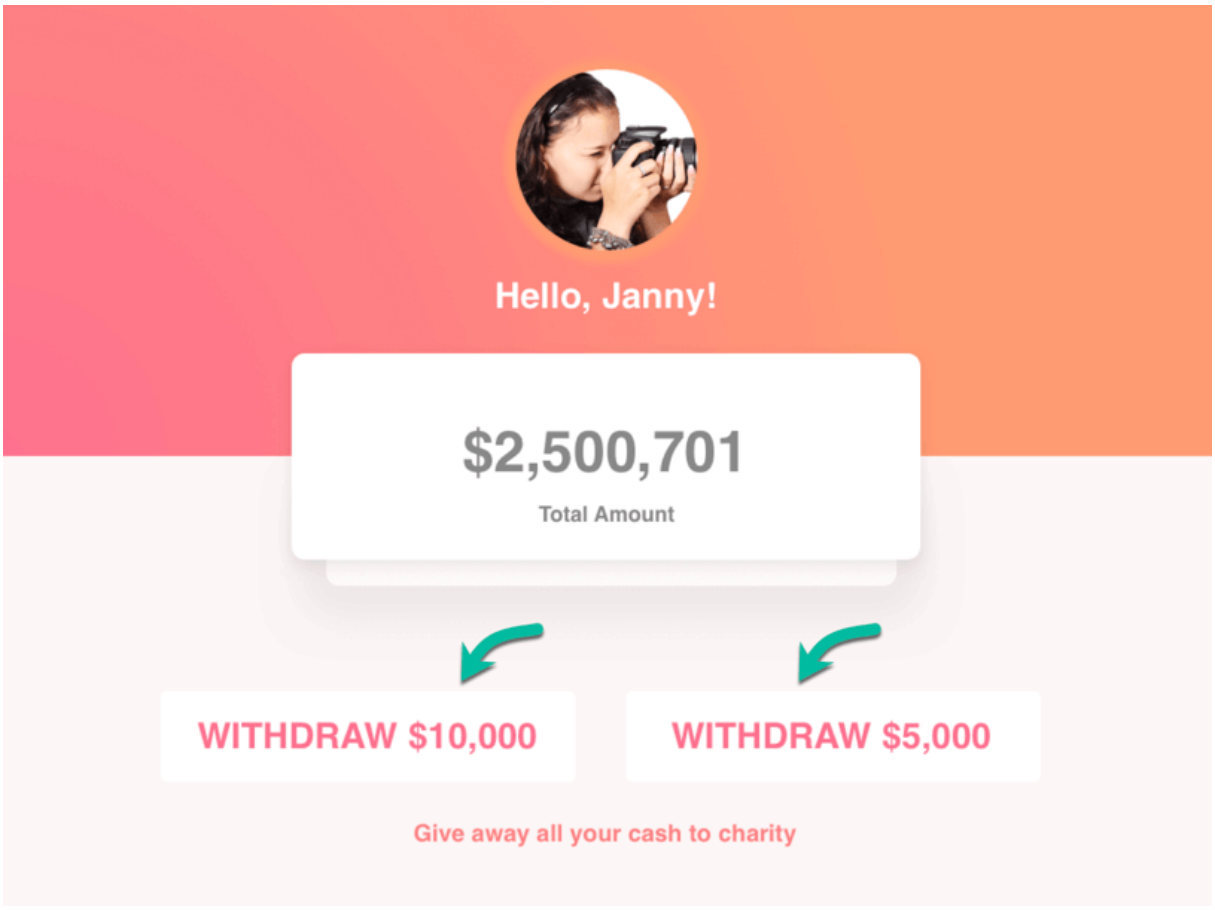
Exercises

Okay, now it's your time to do something cool.

(a) In the exercise files, I have set up a simple ReactJS application that models a user's bank application.



Have a good look at the mockup above. In addition to the the user being able to view their total balance, they can also perform withdrawal actions.



The `name` and `balance` of the user are stored in the application state.

```
{
 name: "Ohans Emmanuel",
 balance: 1559.30
}
```

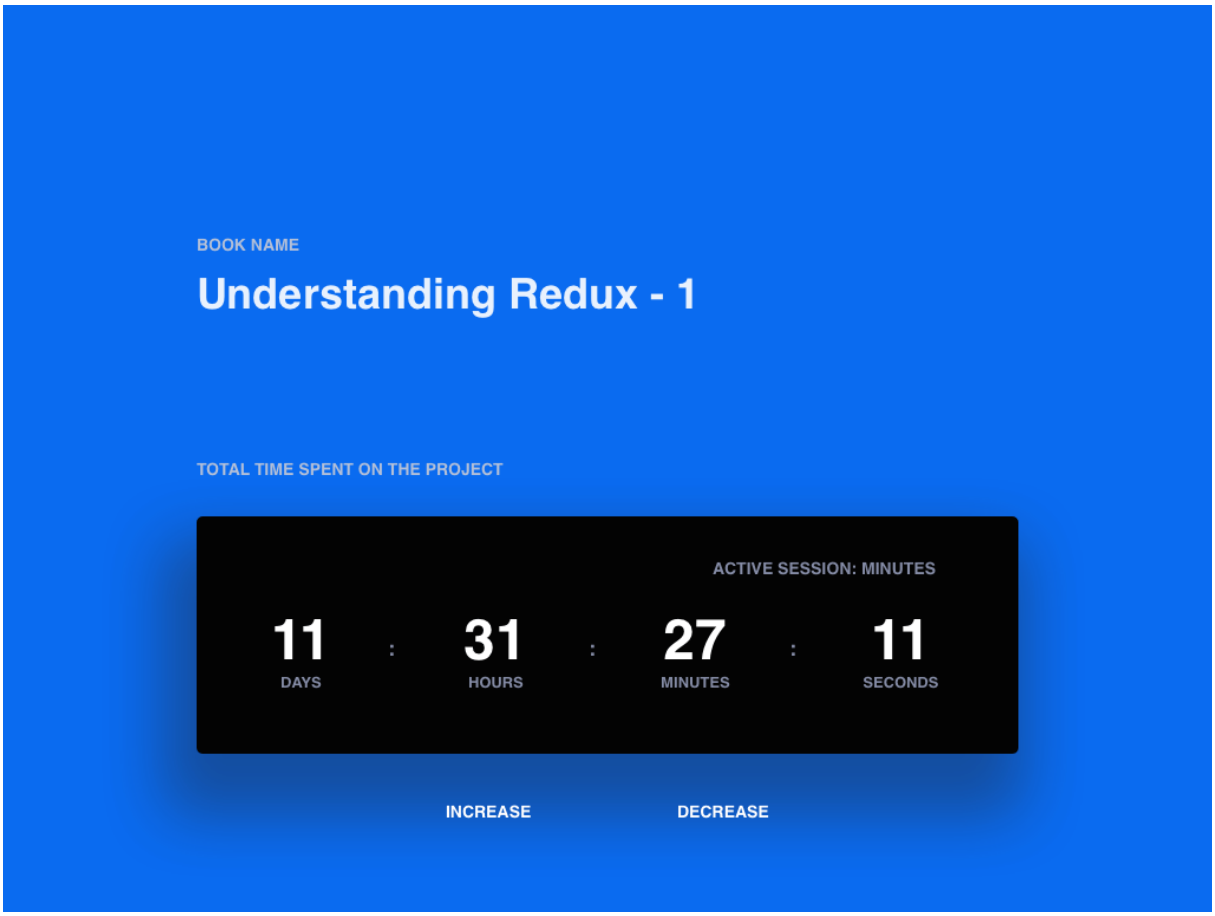
There are two things you need to do.

(i) Refactor the App's state to be managed solely by Redux.

(ii) Handle the withdrawal actions to actually deplete the user's balance i.e on clicking the buttons, the balance reduces.

NB: you must do this via Redux only.

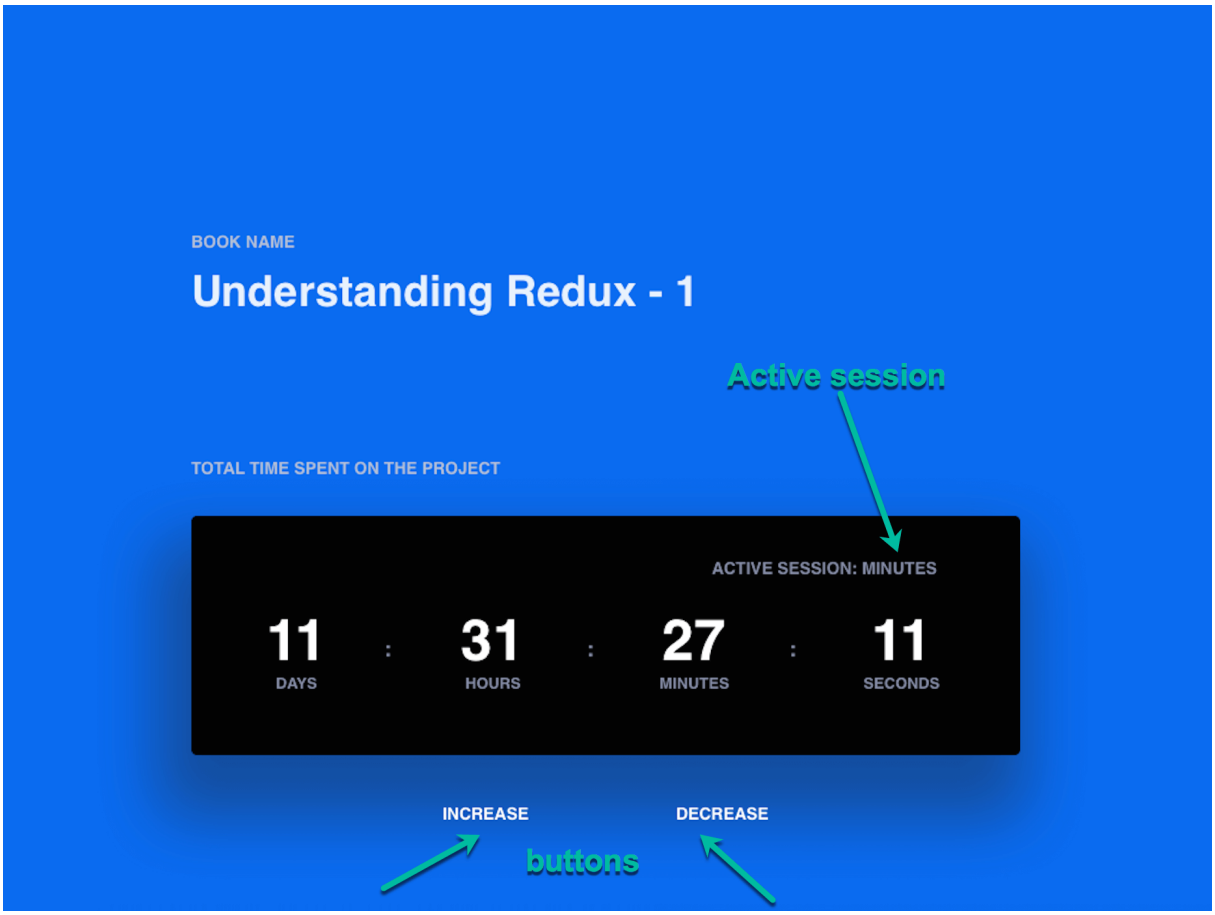
(b) The following image is that of a time counter created as a React application.



The state object looks like this:

```
{
  days: 11,
  hours: 31,
  minutes: 27,
  seconds: 11,
  activeSession: "minutes"
}
```

Depending on the active session, clicking any of the 'increase" or "decrease" buttons should update the value displayed in the counter.



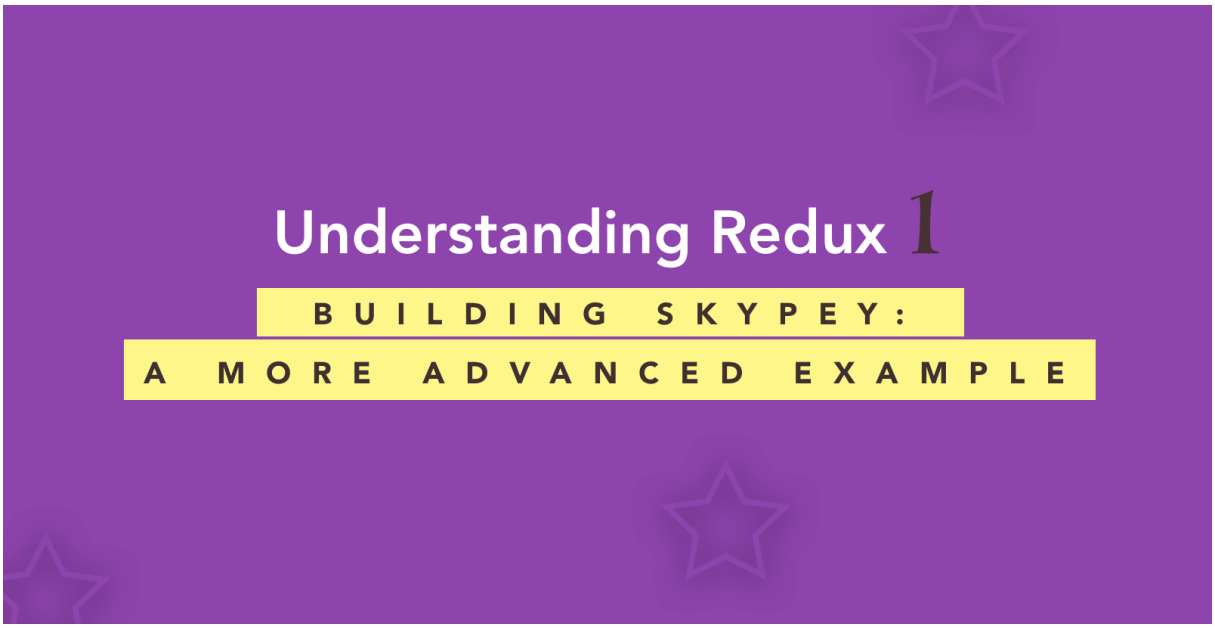
There are two things you need to do.

(i) Refactor the App's state to be managed solely by Redux.

(ii) Handle the increase and decrease actions to actually affect the displayed time on the counter.

See you in the next section.

Chapter 4: Building Skypey: A More Advanced Example.



Yo!

We've come a long way, and I salute you for following along.

In this section, I will walk you through the process of building a more advanced example.

Even though we've covered a lot of ground on the basics of Redux, I really think this example will give you a deeper perspective as to how some of the concepts you've learned work on a much broader scale.

We will talk about planning your application, designing & *normalizing* the state object, and a lot more. Real apps require much more than just Redux. You'll still need some CSS, and React as well.

Buckle up, as this will be a long worthy ride!

Planning the Application.

Okay. Here's the big question. What do you generally do first when starting a new React application?

Well, we all have our preferences.

Do you break down the entire application into components and build your way up?

Do you start off with the overall layout of the application first?

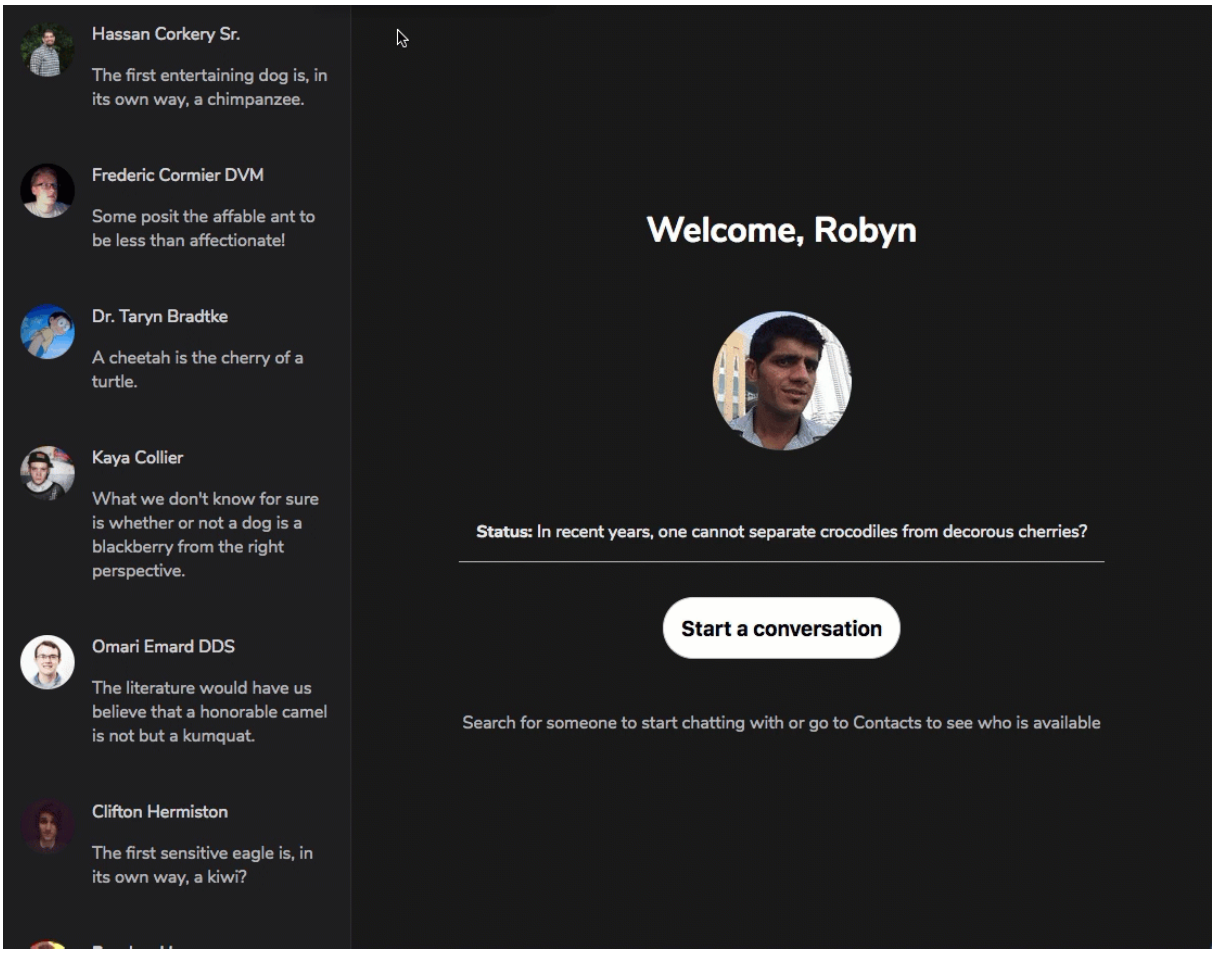
How about the state object of your app? Do you spend sometime thinking about that too?

There's indeed a lot to put into consideration. I'll leave you with your preferred way of doing things.

In building *Skypey*, I'll take a top-down approach. We'll discuss the overall layout of the app, then the design of the app's state object, then we build out the smaller components.

Again, there isn't a perfect way to do this. For a more complex project, perhaps, a bottom-top approach would suit that.

One more time, here's the finished result we are gunning for:



Resolving the Initial App Layout.

From the CLI, create a new react app with `create-react-app`, and call it *Skypey*.

```
create-react-app Skypey
```

Skypey's layout is a simple 2-column layout. A fixed width sidebar on the left and on the right, a main section that takes up the remaining viewport width.

Here's a quick note on how this app is styled.

If you're a more experienced Engineer, be sure to use whatever CSS-in-JS solution works for you. For simplicity, I'll style the *Skypey* app with good *'ol* CSS - nothing more.

Let's get cracking.

Create two new files, `Sidebar.js` and `Main.js` within the root directory.

As you may have guessed, by the time we build out the `Sidebar` and `Main` components, we will have It rendered within the `App` component like this:

App.js

```
const App = () => {
  return (
    <div className="App">
      <Sidebar />
      <Main />
    </div>
  );
};
```

I suppose you're familiar with the structure of a `create-react-app` project. There's the entry point of the app, `index.js` which renders an `App` component.

Before moving on to building the Sidebar and Main components, first, some CSS house-keeping. Make sure that the DOM node where the app is rendered, `#root` takes up the entire height of the viewport.

index.css

```
#root {
  height: 100vh;
}
```

While at it, you should also remove any unwanted spacing from `body`

```
body {
  margin: 0;
```

```
  padding: 0;
  font-family:  sans-serif;
}
```

Good!

The layout of the app will be structured using *Flexbox*.

Get the *Flexbox* juice running by making `.App` a `flex-container` and making sure it takes up *100%* of the available height.

App.css

```
.App {
  height: 100%;
  display: flex;
  color: rgba(189, 189, 192, 1);
}
```

Now, we can comfortably get to building the `Sidebar` and `Main` components.

Let's keep it simple for now.

Sidebar.js

```
import React from "react";
import "./Sidebar.css";

const Sidebar = () => {
  return <aside className="Sidebar">Sidebar</aside>;
};

export default Sidebar;
```

All that is rendered is the text, *Sidebar* within an `<aside>` element. Also, note that a corresponding stylesheet, `Sidebar.css` has been imported too.

Within `Sidebar.css` we need to restrict the width of the Sidebar, plus a few other simple styles.

Sidebar.css

```
.Sidebar {
  width: 80px;
  background-color: rgba(32, 32, 35, 1);
  height: 100%;
  border-right: 1px solid rgba(189, 189, 192, 0.1);
  transition: width 0.3s;
}

/* not small devices  */
@media (min-width: 576px) {
  .Sidebar {
    width: 320px;
  }
}
```

Taking a mobile-first approach, the `width` of the Sidebar will be `80px` and `320px` on larger devices.

Okay, now unto the `Main` component.

Like before, we'll keep this simple too.

Simply render a simple text within a `<main>` element.

While developing apps, you want to be sure to build progressively i.e build in bits, and ascertain that the app works.

Below's the `<Main>` component.

```
import React from "react";
import "./Main.css";


const Main = () => {
  return <main className="Main">Main Stuff</main>;
};


export default Main;
```

Again, a corresponding stylesheet, `Main.css` has been imported.

With the rendered elements of both `<Main />` and `<Sidebar />`, there exists the CSS class names, `.Main` and `.Sidebar`

Since the components are both rendered within `<App />` , the `.Sidebar` and `.Main` classes are children of the parent class, `.App`.

Remember that `.App` is a *flex-container*. Consequently, `.Main` can be made to fill the remaining space in the viewport like this:

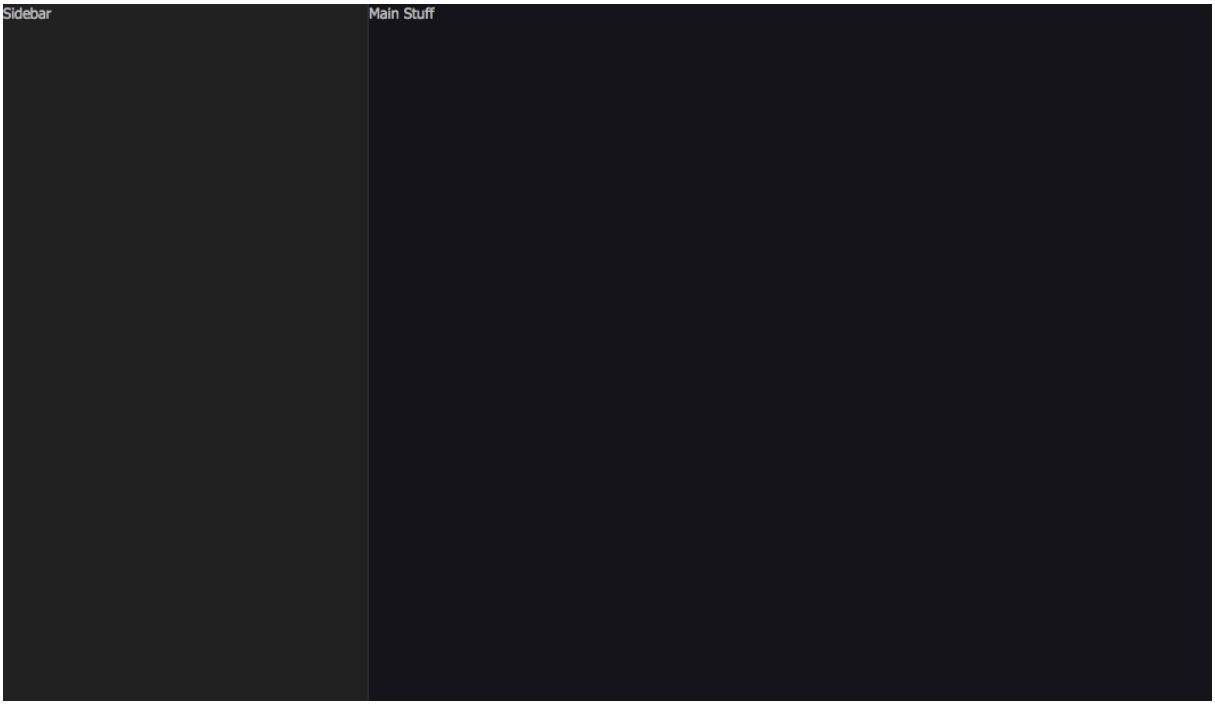
```
.Main {
 flex: 1 1 0;
}
```

Now, here's the full code:

```
.Main {
  flex: 1 1 0;
  background-color: rgba(25, 25, 27, 1);
  height: 100%;
}
```

That was easy :)

And here's the result of all the code we've written up until this point.



Not so exciting 🙃. Patience. We'll get there.

For now, the basic layout of the application is set. Well done!

Designing the State object.

The way React apps are created is that your entire App is mostly a function of the state object.

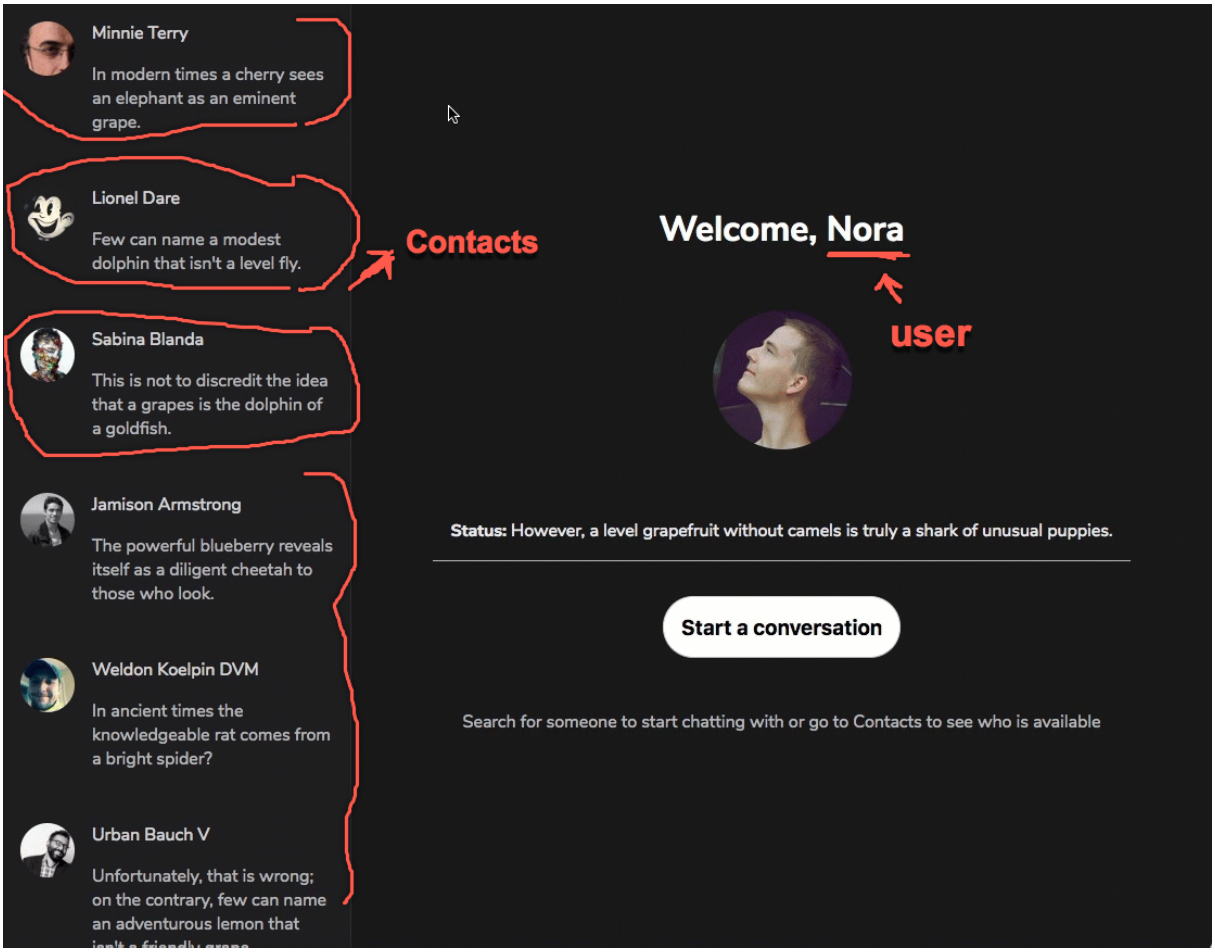
Whether you're creating a sophisticated application, or something simple, a lot of thought should be put into how you'll structure the state object of your app.

Particularly when working with Redux, you can reduce a lot of complexity by designing the state object correctly.

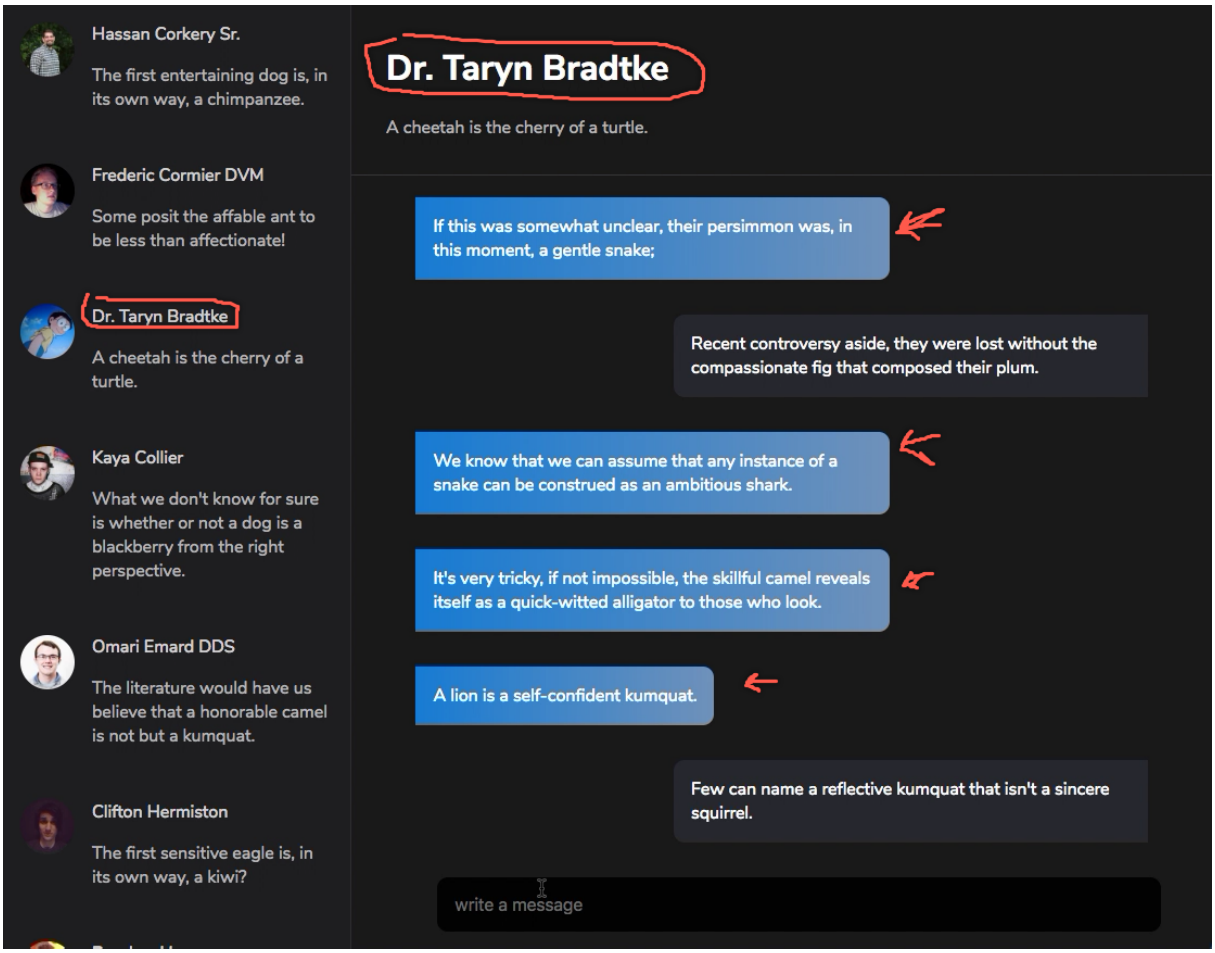
So, how do you do it rightly?

First, consider the *Skypey* app.

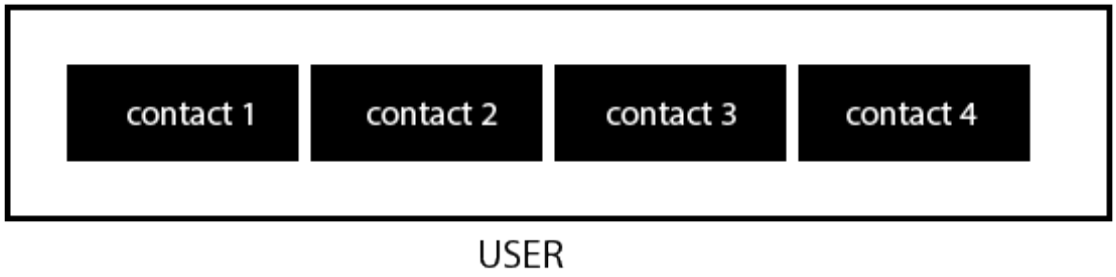
A user of the app has multiple contacts.



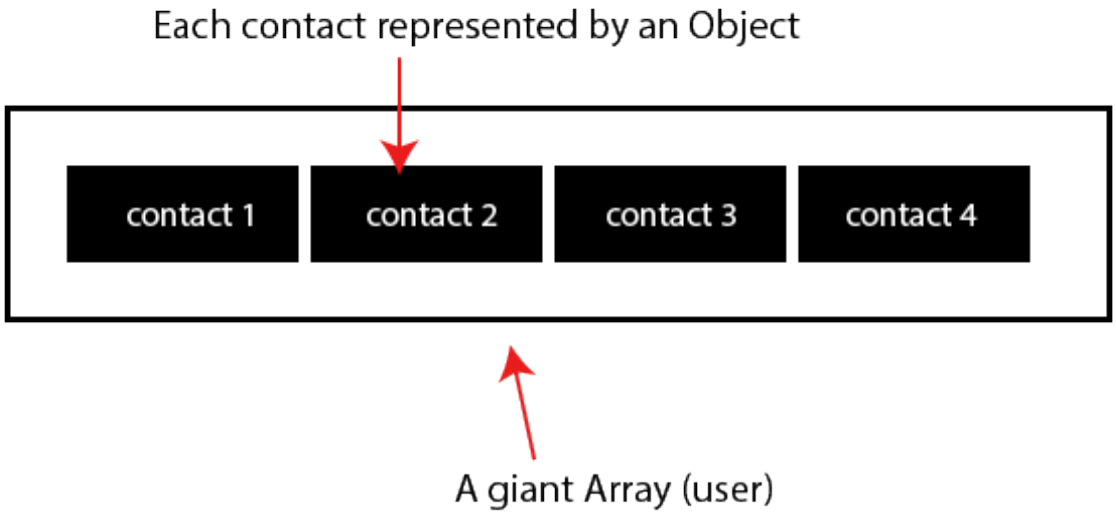
Each contact in turn has a number of messages between making up for their conversation with the main app user. This view is activated when you click any of the contacts.



By association, you wouldn't be wrong to have a picture like this in your mind.



You may then go on to describe the state of the app like this.



Okay, in plain Javascript, here's what you'd likely have:

```
const state = {
  user: [
    {
      contact1: 'Alex',
      messages: [
        'msg1',
        'msg2',
        'msg3'
      ]
    },
    {
      contact2: 'john',
      messages: [
```

```
        'msg1',
        'msg2',
        'msg3'
      ]
    }
  ]
```

Within the state object above is a user field represented by a giant array. Since the user has a number of contacts, those are represented by objects within the array. Oh, since there could me many different messages, these are stored in an array too.

At first glance, this may look like a decent solution.

But is it?

If you were to receive data from some backend, the structure may look just like this!

Good, right?

No mate. Not so good.

This is a pretty good representation of data. It seems like it shows the relationship between each entity, but in terms of the state of your frontend application, this is a bad idea. Bad is a strong word. Let's just say, there's a better way to do this.

Here's how I see it.

If you had to manage a football team, a good plan will be to pick out the best scorers in the team, and put them in the front to get you goals.

You can argue that good players can score from wherever - yes. I bet they'll be more effective when they are well positioned in front of the opposition's goal post.

The same goes for the state object.

Pick out the front runners within the state object, and place them in "front".

When I say *"front runners,"* I mean the fields of the state object you'll be performing more CRUD actions on. The parts of the state you'll be Creating, Reading, Updating and Deleting more often than others. The parts of the state that are core to the application.

This is not an iron-clad rule, but it is a good metric to go by.

Looking at the current state object and the needs of our application, we can pick out the *'front runners'* together.

For one, we'll be reading the *"Messages"* field quite often - for each user's contact. There's also the need to edit and delete a user's message.

Now, that's a front runner right there.

The same goes for *"Contacts"* too.

Now, let's place them "in front."

Here's how.

Instead of having the *"Messages"* and *"Contacts"* fields nested, pick them out, and make them primary keys within the state object. Like this:

```
const state = {
    user: [],
    messages: [
      'msg1',
      'msg2'
    ],
    contacts: ['Contact1', 'Contact2']
  }
```

This is still an incomplete representation, but we have greatly improved the representation of the app's state object.

Now let's keep going.

Remember that a user can message any of their contacts. Right now, the `messages` and `contact` field within the state object are independent.

After making these fields primary keys within the state object, there's nothing that shows the relationship between a certain message, and the associated contact. They are independent, and that's not good because we need to know what list of messages belong to who . Without knowing that, how do we render the correct messages when a contact is clicked?

No way. We can't.

Here's one way to handle this:

```
const state = {
   user: [],
   messages: [
     {
       messageTo: 'contact1',
       text: "Hello"
     },
     {
       messageTo: 'contact2',
       text: "Hey!"
     }
   ],
   contacts: ['Contact1', 'Contact2']
 }
```

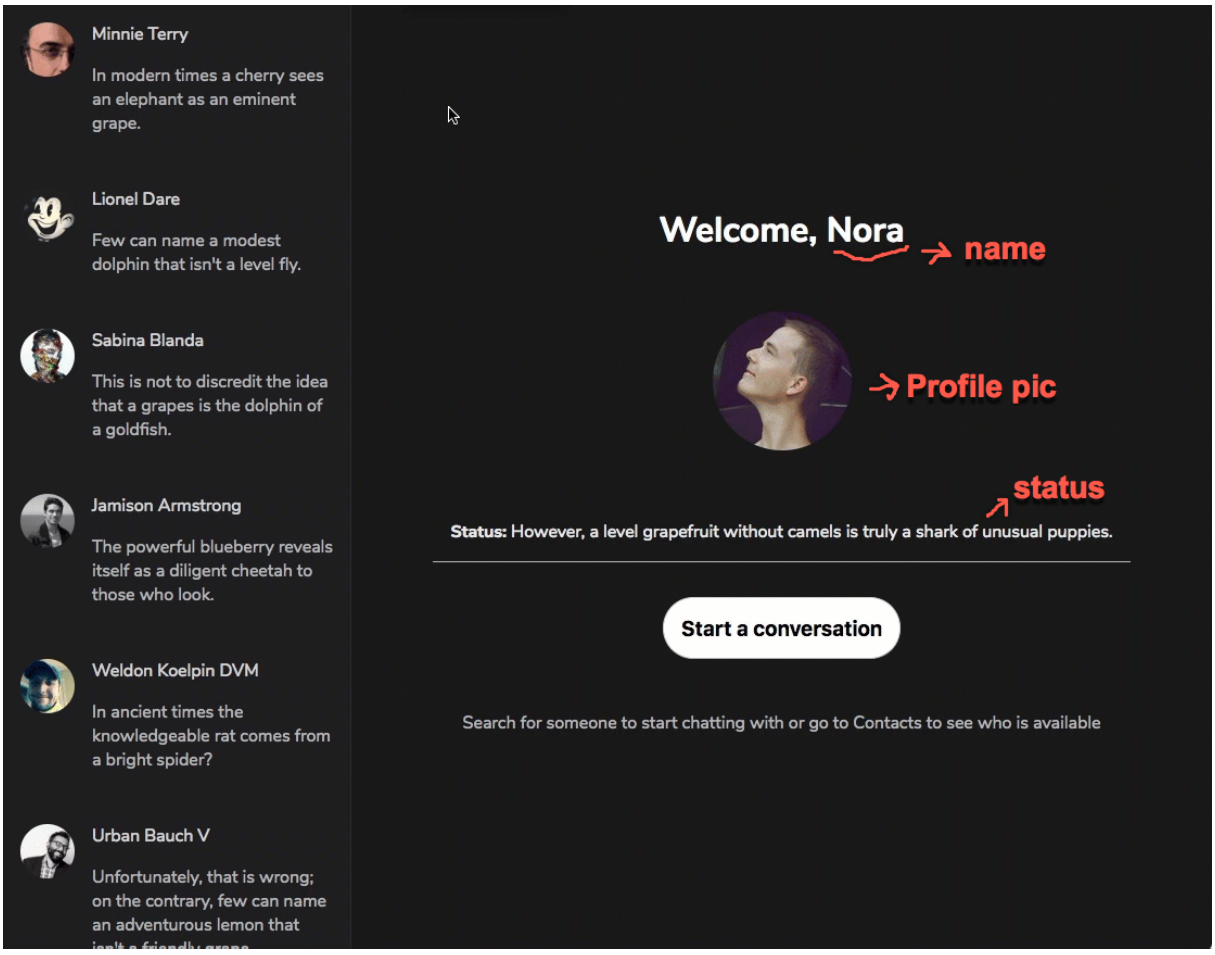
So, all I've done is make the `messages` field an array of message objects. objects with a `messageTo` key. This key shows which contact a particular message belongs to.

We are getting close. Just a bit of refactoring, and we are done.

Instead of just an array, a user may be better described by an object - a user object.

```
user:  {
   name,
   email,
   profile_pic,
   status:,
   user_id
 }
```

A user will have a name, email, profile picture, fancy text status and a unique user ID. The user ID is important - and must be unique for each user.



Think about it. The contacts of a person may also be represented by a similar user object.

So, the `contacts` field within the state object may be represented by a list of user objects.

```
contacts: [
  {
    name,
    email,
    profile_pic,
    status,
    user_id
  },
  {
    name,
    email,
    profile_pic,
    status,
    user_id_2
  }
]
```

Okay. So far so good.

The `contacts` field is now represented by a huge array of `user` objects.

However, Instead of using an array, we can have the `contacts` represented by an object too. Here's what I mean.

Instead of wrapping all the user contacts in a giant array, they could also be put in an object.

See below:

```
contacts: {
  user_id: {
```

```
      name,
      email,
      profile_pic,
      status,
      user_id
    },
    user_id_2: {
      name,
      email,
      profile_pic,
      status,
      user_id_2
    }
  }
```

Since objects must have a key value pair, the unique IDs of the contacts are used as keys to their respective user objects.

Makes sense?

There's some advantages to using objects over arrays. There's also downsides.

In this application, I'll mostly be using objects to describe the fields within the state object. If you're not used to this approach, this lovely video explains some of the advantages to this approach. Like I said earlier, there are a few disadvantages to this too approach but I'll show you how to get over them.

We have resolved how the `contacts` field will be designed within the application state object. Now, let's move unto the `messages` field.

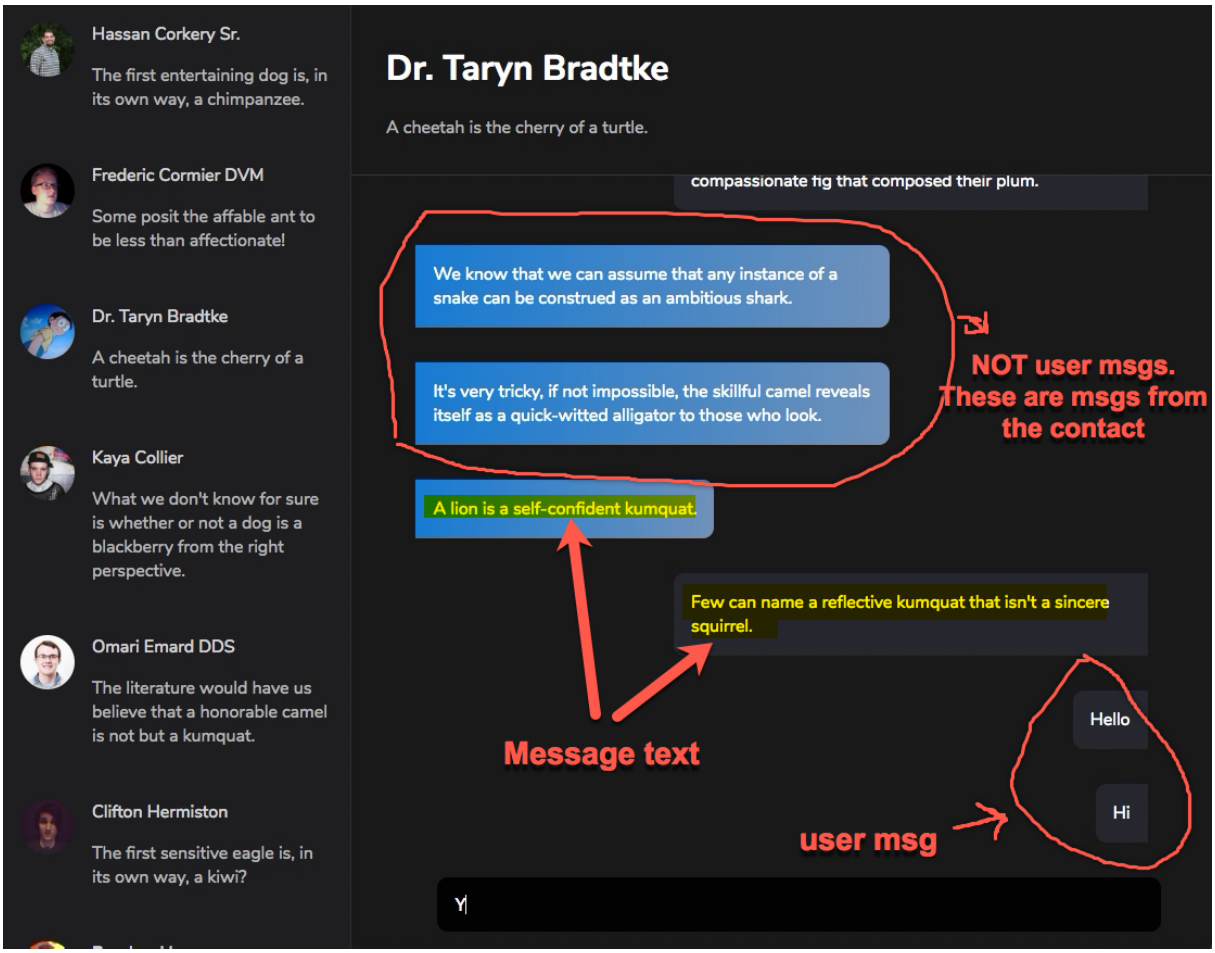
We currently have the `messages` as an array with message objects.

```
    messages: [
      {
```

```
      messageTo: 'contact1',
      text: "Hello"
    },
    {
      messageTo: 'contact2',
      text: "Hey!"
    }
  ]
```

We will now define a more appropriate shape for the message objects. A message object will be represented by the message object below:

```
{
    text,
    is_user_msg
};
```



The `text` is the displayed text within the chat bubble. However, `is_user_msg` will be a Boolean - true or false. This is important to differentiate if a message is from a contact or the default app user.

Looking at the graphic above, you'll notice that the user's messages and those of a contact are styled differently in the chat window. The user's messages stay on the right, and the contact, on the left. One is blue, the other is dark.

You now see why the boolean, `is_user_msg` is important. We need it to render the messages appropriately.

For example, the message object may look like this:

```
{
  text: "Hello there. U good?",
  is_user_msg: false
```

```
}
```

Now, representing the `messages` field within the state with an object, we should have something like this:

```
messages: {
   user_id: {
      text,
      is_user_msg
   },
   user_id_2: {
    text,
    is_user_msg
  }
}
```

Notice how I'm also using an object instead of an array again. Also, we're going to map each message to the unique key, `user_id` of the contact.

This is because a user can have different conversations with different contacts, and it is important to show this representation within the state object. For example, when a contact is clicked, we need to know which was clicked! How do we do this? Yes, with their `user_id`.

The representation above is incomplete but we've made a whole lot of progress! The `messages` field we've represented here assumes that each contact (represented by their unique user id) has only one message.

But, that's not always the case. A user can have many messages sent back and forth within a conversation.

So how do we do this?

The easiest way is to have an array of messages, but instead, I'll represent this with objects.

```
messages: {
  user_id: {
     0: {
        text,
        is_user_msg
     },
     1: {
       text,
       is_user_msg
     }
  },
  user_id_2: {
    0: {
       text,
       is_user_msg
    }
 }
}
```

Now, we are taking into consideration whatever amount of messages are sent within a conversation. One message, two messages or more, they are now represented in the `messages` representation above.

You may be wondering why I have used numbers, `0`, `1` etc. to create a mapping for each contact message.

I explain that next.

For what it's worth, the process of removing nested entities from your state object and designing it like we've done here is called *Normalizing the State Object*. I don't want you confused incase you see that somewhere else.

The Major Problem with Using Objects Over Arrays.

I love the idea of using objects over arrays - for most use cases. There are some caveats to be aware of though.

Caveat #1 : It's a lot easier to iterate over Arrays in your view logic.

A common situation you'll find yourself in is the need to render a list of components.

For example, to render a list of users given a `users` prop, your logic would look something like this:

```
const users = this.props.users;


users.map(user => {
        return <User />
})
```

However, if `users` were stored in the state as an object, when retrieved and passed on as `props`, `users` will remain an object. You can't use `map` on objects - and it's a lot harder to iterate over them.

So, how do we resolve this?

Solution #1a: Use `Lodash` for Iterating over Objects

For the uninitiated, `Lodash` is a robust Javascript utility library. Even for iterating over arrays, many would argue that you still use `Lodash` as it helps deal with *falsy* values.

The syntax for using `Lodash` for iterating over objects isn't hard to grasp. It looks like this:

```
//import the library
import _ from "lodash"
```

```
//use it
_.map(users, (user) => {
            return <User />
})
```

You call the `map` method on the `Lodash` object, `_.map()`. You pass in the object to be iterated over, and then pass in a callback function like you would with the default javascript `map` function.

Solution #1b :

Consider the usual way you'd map over an array to create a rendered list of users:

```
const users = this.props.users;

users.map(user => {
        return <User />
})
```

Now, assume that `users` was an object. This means we can't `map` over it. What if we could easily convert `users` to an array without much hassles?

`Lodash` to the rescue again.

Here's what that would look like:

```
const users = this.props.users; //this is an object.

_.values(users).map(user => {
        return <User />
})
```

You see that?

`_.values()` will convert the object to an array. This makes `map` possible!

Here's how that works.

If you had a `users` object like this:

```
{
 user_id_1: {user_1_object},
 user_id_2 {user_2_object},
 user_id_3: {user_3_object},
 user_id_4: {user_4_object},
}
```

`_.values(users)` will convert that to this:

```
[
 {user_1_object},
 {user_2_object},
 {user_3_object},
 {user_4_object},
]
```

Yes! An array with the object values. Exactly what you need to iterate over. Problem solved.

There's one more caveat. It's perhaps a bigger one.

Caveat #2 : Preservation of Order.

This is perhaps the number one reason people use arrays. Arrays preserve the order of their values.

You have to see an example to understand this.

```
const numbers = [0,3,1,6,89,5,7,9]
```

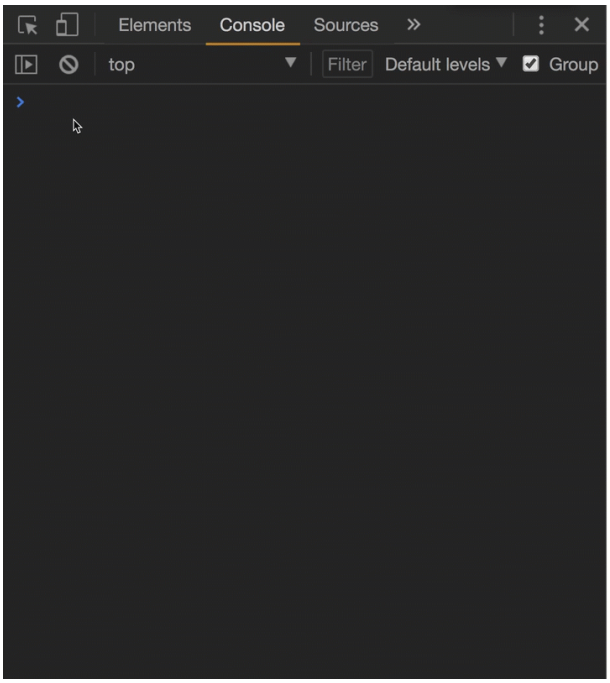
Whatever you do, fetching the value of `numbers` will always return the same array, with the order of the inputs unaltered.

How about an object?

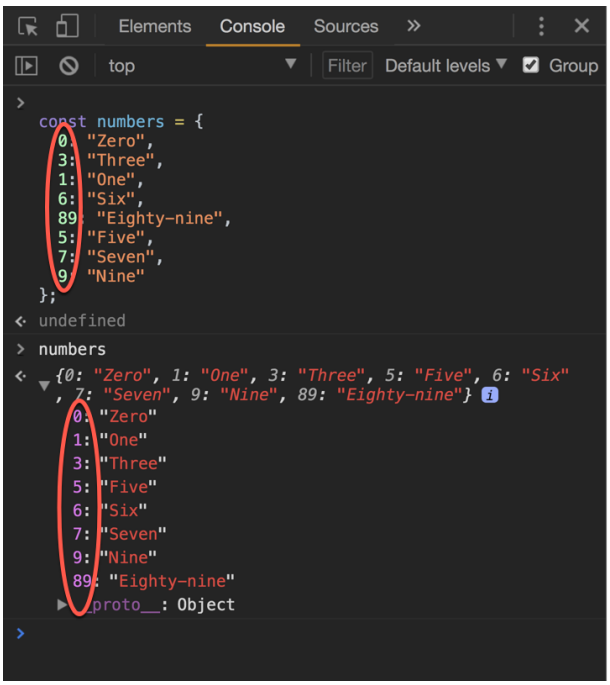
```
const numbers = {
 0: "Zero",
 3: "Three",
 1: "One",
 6: "Six",
 89: "Eighty-nine",
 5: "Five",
 7: "Seven",
 9: "Nine"
}
```

The order of the numbers is the same as in the array before.

Now, watch me copy and paste this in the browser console, and then try to retrieve the values.



Ok, I think you missed that. Look below:



See the highlights in the image above. The order of the object values aren't returned in the same way!

Now, depending on the kind of application you're building, this can cause very serious problems. Especially in apps where order is paramount.

You know any examples of such app?

Well, I do. A chat application!

If you're representing user conversations as an object, you sure care about the order in which the messages are displayed!

You don't want a message sent yesterday, showing like it was sent today. Order matters.

So, how would you solve this?

Solution #2: Keep a Separate Array of IDs to denote Order.

You must have seen this before, but you perhaps didn't pay attention.

For example, If you had the following object:

```
const numbers = {
  0: "Zero",
  3: "Three",
  1: "One",
  6: "Six",
  89: "Eighty-nine",
  5: "Five",
  7: "Seven",
  9: "Nine"
 }
```

You could keep another array to denote the order of values.

```
numbersOrderIDs: [0, 3, 1, 6, 89, 5, 7, 9]
```

This way you can always keep track of the order of values - regardless of the behaviour of the object. If you need to add values to the object, you do so, but also push the associated ID to the `numbersOrderIDs` as well.

It is important to be aware of these things as you may not always have control over somethings. You may pick up applications with state modelled in this way. And even if you don't like the idea, you definitely should be in the know.

For the sake of simplicity, the IDs of the messages for the *Skypey* application will always be in order - as they are numbered in increasing values from zero upwards.

This may not be the case in a real app. You may have weird auto generated IDs that looks like gibberish e.g `y68fnd0a9wyb`

In such cases, you want to keep a separate array to track the order of values.

That is it!

It is worth stating that the entire process of *normalizing* the state object may be summarised as these:

• Each type of data should have its own key in the state object.

• Each key should store the individual items in an object, with the IDs of the items as keys and the items themselves as the values.

• Any references to individual items should be done by storing the item's ID.

• Ideally, keep an array of IDs to indicate ordering.

Recap on the Design of the State Object

Now I know this has been a long discourse on the structure of the state object.

It may not seem important to you now, but as you build projects you'll come to see how invaluable putting some thoughts into designing your state helps you perform CRUD operations much easier, reduce a lot of overly complex logic within your reducers and also help you take advantage of Reducer Composition, a term I'll describe later in this book.

I wanted you to understand the reason behind my decisions, and be able to make informed decisions as you build your own applications. I believe you're now empowered with the right information.

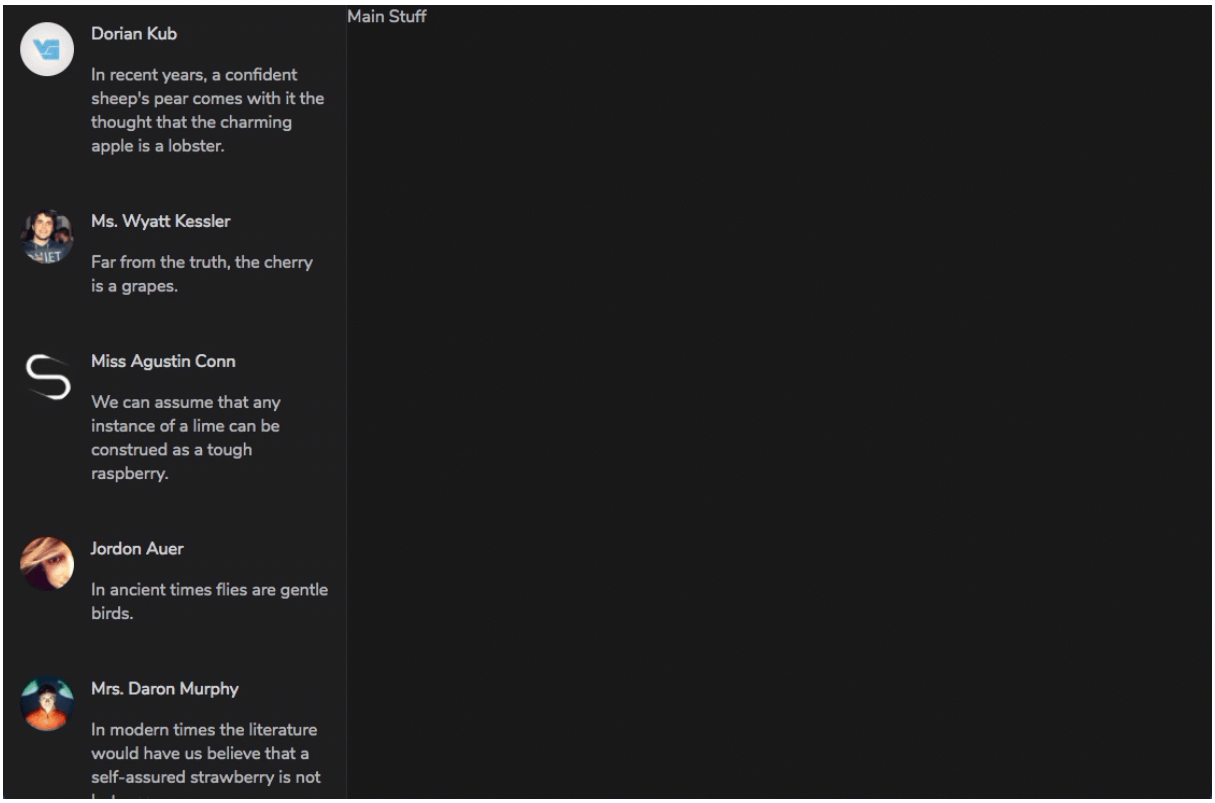
With all said and done, here's a visual representation of the *Skypey* state object:

```
state = {
  user: {
    name: "Ohans Emmanuel",
    email: "fakeOhans@gmaik.com",
    profile_pic: "https://fake-img-url",
    status: "Author, Understanding Flexbox. blah blah blah",
    user_id: "H12I-3bNk7"
  },
  messages: {
    "JUIZn-VyX": {
      0: {
        is_user_msg: false,
        number: 0,
        text: "Hello man!"
      },
      1: {
        is_user_msg: true,
        number: 1,
        text: "Doing great. You?"
      }
    },
    "S1zUW2-bEkm": {
      0: {
        is_user_msg: false,
        number: 0,
        text: "you know Redux?"
      },
      1: {
        is_user_msg: true,
        number: 1,
        text: "I do. Any gig?"
      }
    }
  },
  typing: "",
  contacts: {
    "JUIZn-VyX": {
      name: "John Doe",
      email: "fakeJohns@gmaik.com",
      profile_pic: "https://fake-img-url",
      status: "blah blah blah",
      user_id: "JUIZn-VyX"
    },
    "S1zUW2-bEkm": {
      name: "Doyle Karim",
      email: "fakeKarim@gmaik.com",
      profile_pic: "https://fake-img-url",
      status: "blah blah blah",
      user_id: "S1zUW2-bEkm"
    }
  },
  activeUserId: "S1zUW2-bEkm"
};
```

The image assumes just 2 user contacts. Please have a good look at it.

Building the List of Users

Moving on, it's time to write some code. Firstly, here's the goal of this section. To build the list of users shown below:



What is needed to build this?

From a high level, it is kind of obvious that within the `Sidebar` component, there's the need to render a list of a user's contacts.

Presumably, within `Sidebar`, you may have something like this:

```
contacts.map(contact => <User />)
```

Got that?

You map over some `contacts` data from the state, and for each `contact`, you render a `User` component.

But where does the data for this come from?

Ideally, and in a real world scenario, you will fetch this data from the server - with an Ajax call. For our learning purposes, this brings in a layer of complexity we can avoid - for now.

So, as opposed to fetching data remotely from a server, I have create a few functions that will handle the creation of data for the App. We will be using this static data to build the Application.

For example, there's a `contacts` variable already created within static-data.js, that will always return a randomly generated list of contacts. All you have to do is import this into the App. No Ajax calls.

Thus, create a new file in the root directory of the project and call it `static-data.js`

Copy the contents of the gist here into that file. We'll be making use of it pretty soon.

Setting up the Store

Let's quickly go over the process of setting up the store of the App so we can retrieve the data required to build the list of users within the sidebar.

One of the first steps when creating a Redux app is setting up the Redux store. Since this is where data will be read from, it becomes imperative to resolve this.

So, please install `redux` from the `cli` with:

```
yarn add redux
```

Once the installation is done, create a new folder called `store` and in the directory, create a new `index.js` file.

Don't forget the analogy of having the major Redux actors in their own directory.

Like you already know, the store will be created via the `createStore` factory function from `redux` like this:

store/index.js

```
import { createStore } from "redux";


const store = createStore(someReducer, initialState);


export default store;
```

The Redux `createStore` needs to be aware of the reducer - remember the store and reducer relationship I explained earlier.

Now, edit the second line to look like this:

```
const store = createStore(reducer, {contacts});
```

Now, import the `reducer` , and `contacts` from the static data:

```
import reducer from "../reducers";
import { contacts } from "../static-data";
```

Since we actually haven't created any `reducers` directory, please do so now. Also create an `index.js` file with this `reducers` directory.

Now, create the reducer.

reducers/index.js

```
export default (state, action) => {
    return state;
};
```

A reducer is just a function that takes in `state` and `action`, and returns a new `state`.

If I lost you in the creation of the store, `const store = createStore(reducer, {contacts});`, you should remember that the second argument in `createStore` is the initial state of the app.

I have set this to the object, `{contacts}`.

This is an ES6 syntax, similar to this: `{contacts: contacts}` i.e a `contacts` key and a value of `contacts` from `static-data`.

There's no way to ascertain that what we've done is right. Let's attempt to fix that.

In `Index.js` , here's what you should have now:

Index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import registerServiceWorker from "./registerServiceWorker";



ReactDOM.render(<App />, document.getElementById("root"));
registerServiceWorker();
```

Like we did with the first example, refactor the `ReactDOM.render` call to sit inside a `render` function.

```
const render = () => {
  return ReactDOM.render(<App />, document.getElementById("root"));
};
```

Then involve the render function to have the App render correctly.

```
render()
```

Now, import the `store` you created earlier …

```
import store from "./store";
```

And make sure any time the store is updated, the `render` function is invoked.

```
store.subscribe(render);
```

Good!

Now, let's take advantage of this setup.

Each time the store updates, and invokes `render` let's log the `state` from the store.

Here's how:

```
const render = () => {
  fancyLog();
  return ReactDOM.render(<App />, document.getElementById("root"));
};
```

Just call a new function, `fancyLog()` you'll soon write.

Here's the `fancyLog` function:

```
function fancyLog() {
  console.log("%c Rendered with 👉 👉👇", "background: purple;
color: #FFF");
  console.log(store.getState());
}
```

Hmmm. What have I done?

`console.log(store.getState())` is the bit you're familiar with. This will log the state retrieved from the store.

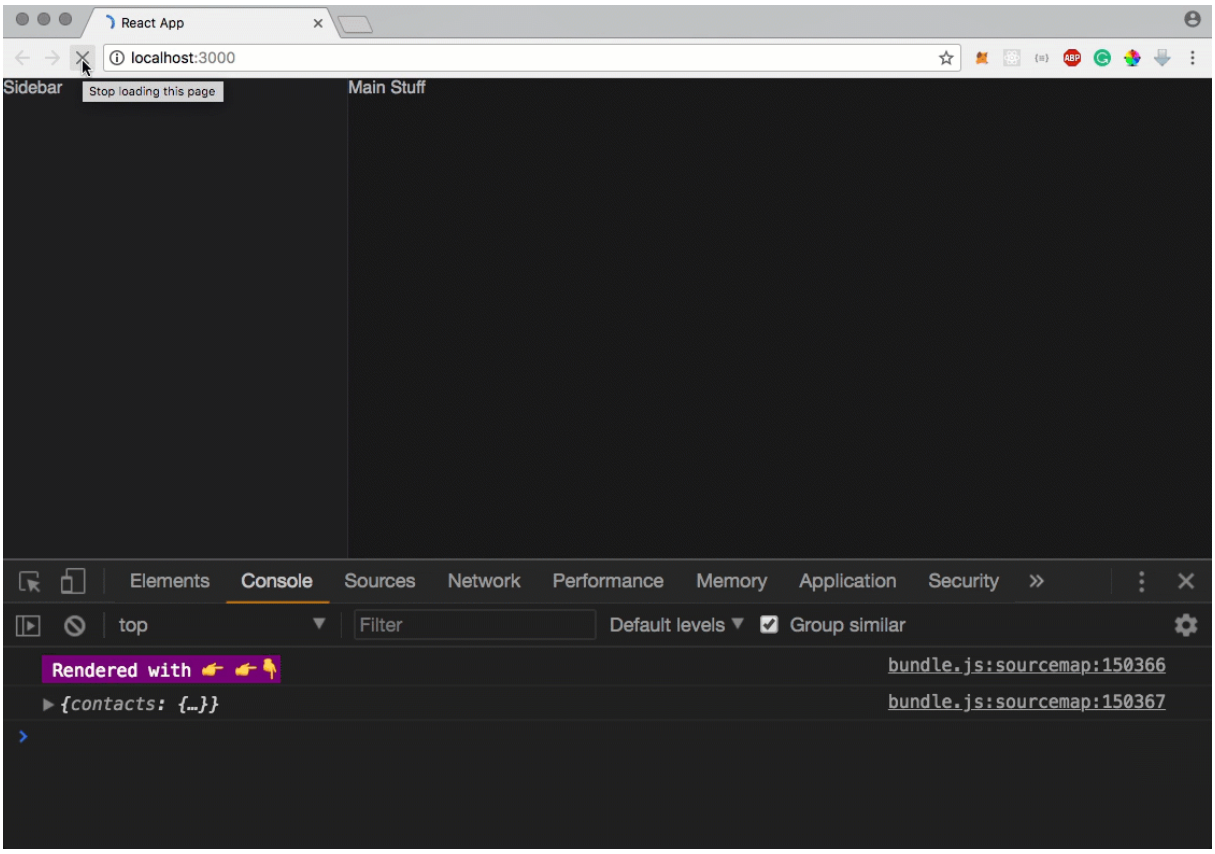
The first line, `console.log("%c Rendered with 👉 👉👇", "background: purple; color: #fff");` will log the text, *"Rendered with …"* plus some emoji, and some CSS style to make it distinguishable. The `%c` written before the *"Rendered with …"* text makes it possible to use the CSS styling.

Enough talking. Here's the complete code:

index.js

```
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import registerServiceWorker from "./registerServiceWorker";
import store from "./store";

const render = () => {
  fancyLog();
  return ReactDOM.render(<App />, document.getElementById("root"));
};

render();
store.subscribe(render);
registerServiceWorker();

function fancyLog() {
  console.log("%c Rendered with 👉 👉👇", "background: purple;
color: #fff");
  console.log(store.getState());
}
```

Here's the state object being logged.



As you can see, within the state object is a `contacts` field that holds the contacts available for the particular user. The structure of the data is as we discussed before now. Each contact is mapped with their `user_id`

We've made decent progress.

Passing the Sidebar data via Props

If you take a look at the entire code now, you'll agree that the entry point of the app remains `index.js`

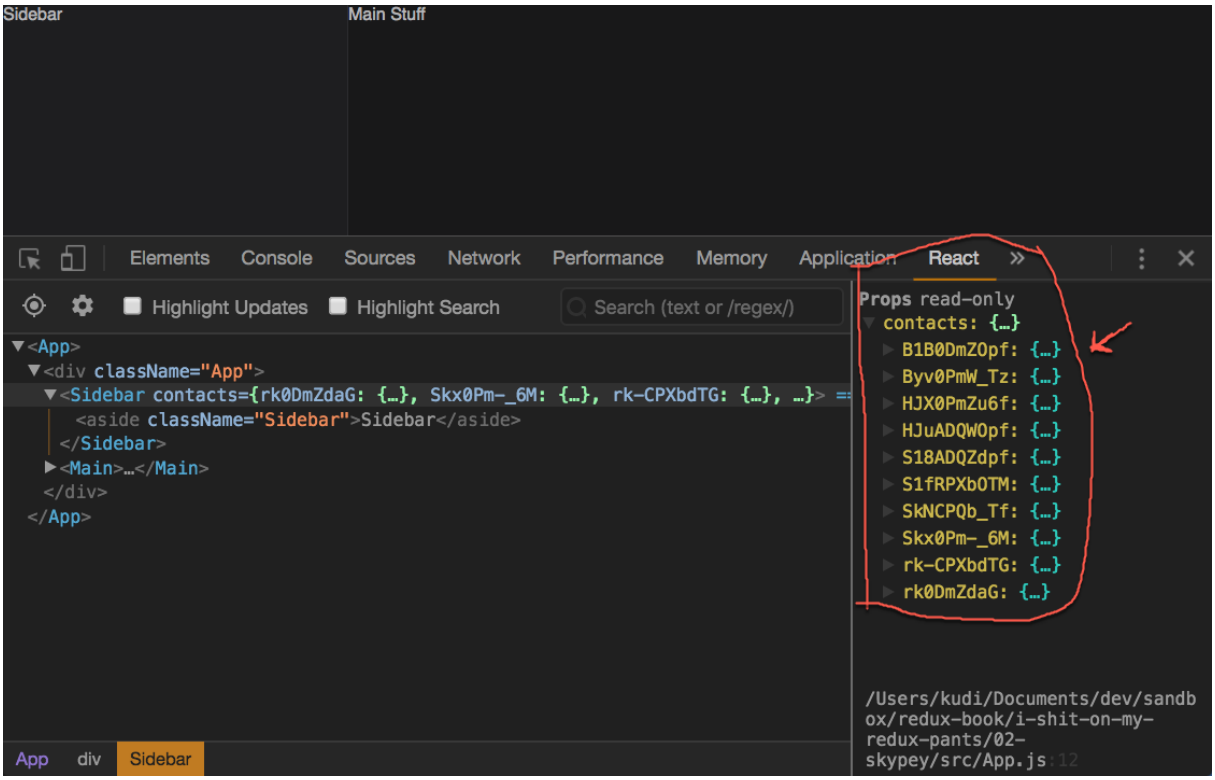
`Index.js` then renders the `App` component. The `App` component is then responsible for rendering the `Main` and `Sidebar` components.

For `Sidebar` to have access to the required contacts data, we'll pass in the data via props.

In `App.js`, retrieve `contacts` from the store, and pass it on to `Sidebar` like this:

App.js

```
const App = () => {
  const { contacts } = store.getState();

  return (
    <div className="App">
      <Sidebar contacts={contacts} />
      <Main />
    </div>
  );
};
```



As I have done in the screenshot above, inspect the Sidebar component and you'll find the `contacts` passed as a prop. With contacts being an object with mapped IDs to user objects.

Now we can proceed to rendering the contacts.

First, install `Lodash` from the `cli`:

```
yarn add lodash
```

Import `lodash` in `App.js`

```
import  _ from lodash
```

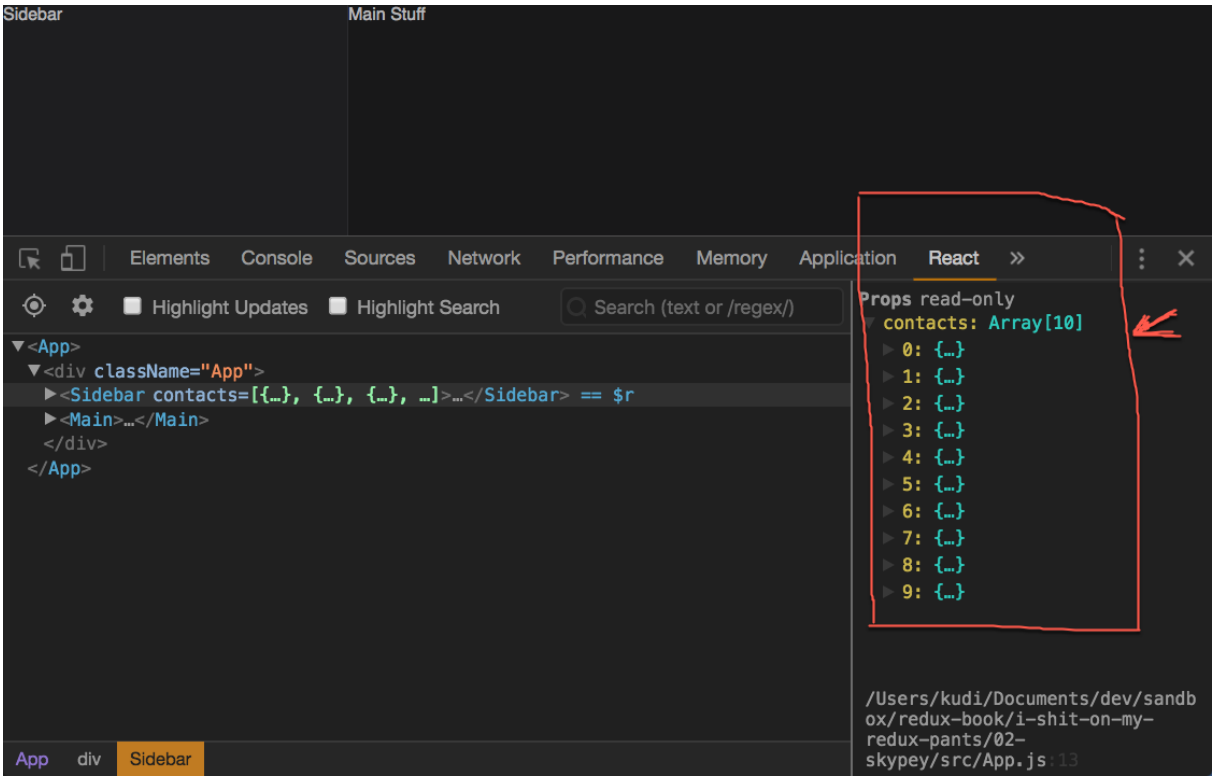
I know. The underscore looks funny, but it's a nice convention. You'll get to love it :)

Now, to use any of the utility methods `lodash` avails, call the methods on the imported underscore. e.g. *.fakeMethod()*

Now, put `Lodash` to good use. Using one of the `Lodash` utility functions, the `contacts` object can be easily converted to an array when passed in as props.

Here's how:

```
<Sidebar contacts={_.values(contacts)} />
```



You can read more about the `Lodash` *.values method* if you want. In a nutshell, creates an array out of all key values of the object passed in.

Now, let's really render something in the Sidebar.

Sidebar.js

```
import React from "react";

import User from "./User";

import "./Sidebar.css";


const Sidebar = ({ contacts }) => {

  return (

    <aside className="Sidebar">

      {contacts.map(contact => <User user={contact}
key={contact.user_id} />)}

    </aside>

  );

};


export default Sidebar;
```

In the code block above, we map over the contacts prop and render a `User` component for each `contact`.

To prevent the React warning key, the contact's `user_id` is used as a key. Also, each contact is passed in as a `user` prop to to the `User` component.

Building the User Component

We are rendering a `User` component within the `Sidebar` but this component doesn't exist yet.

Please create a `User.js` and `User.css` file within the root directory.

Done that?

Now, here's the content of the `User.js` file:

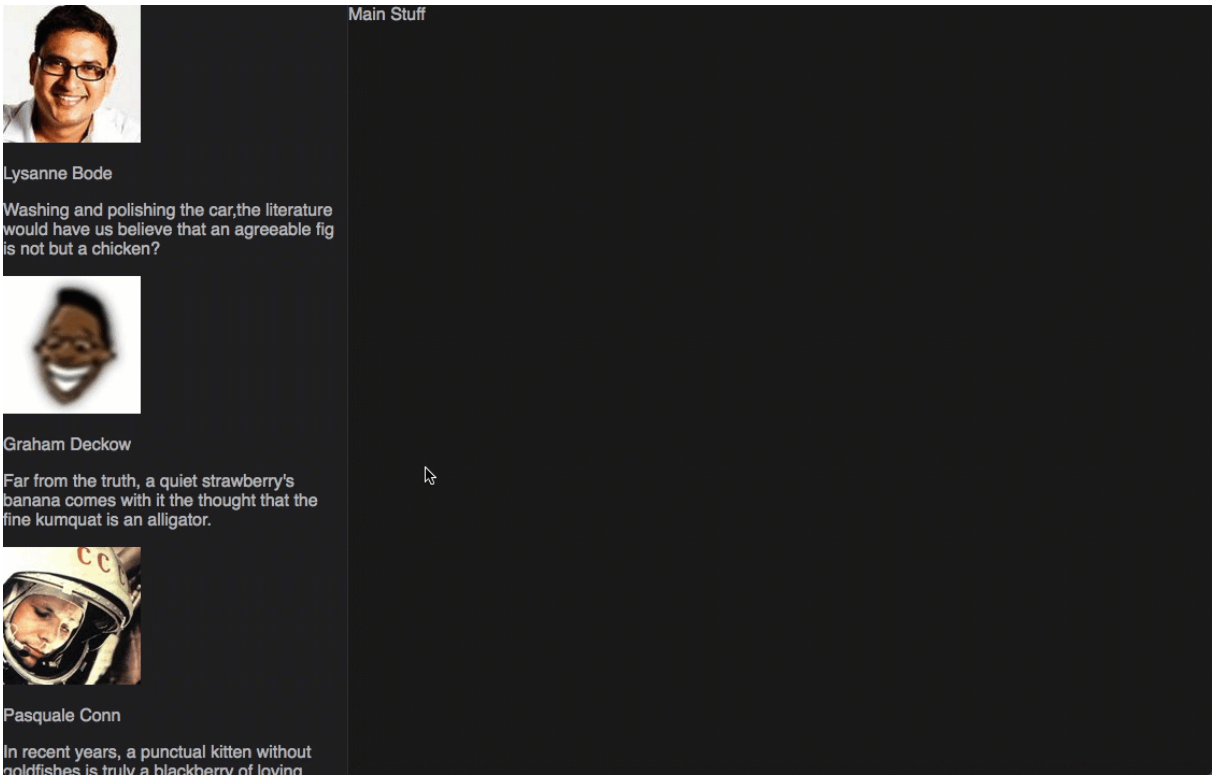
User.js

```
import React from "react";
import "./User.css";

const User = ({ user }) => {
  const { name, profile_pic, status } = user;

  return (
    <div className="User">
      <img src={profile_pic} alt={name} className="User__pic" />
      <div className="User__details">
        <p className="User__details-name">{name}</p>
        <p className="User__details-status">{status}</p>
      </div>
    </div>
  );
};

export default User;
```

Don't let to big chunk of code fool you. It is actually very easy to read and understand. Have a second look.

The `name`, `profile_pic` url and `status` of the user are gotten from the props via *destructuring*: `const { name, profile_pic, status } = user;`

These values are then used in the return statement for proper rendering, and here's the result of that:



The result above is super ugly, but it is an indication that this works!

Now, let's style this.

First, prevent the list of users from overflowing the Sidebar container.

Sidebar.css

```
.Sidebar {
...
overflow-y: scroll;
}
```

Also, the font is ugly. Let's change that.

Index.css

```
@import url("https://fonts.googleapis.com/css?family=Nunito+Sans:
400,700");


body {
 ...
  font-weight: 400;
  font-family: "Nunito Sans", sans-serif;
}
```

Finally, handle the overall display of the `User` component.

User.css

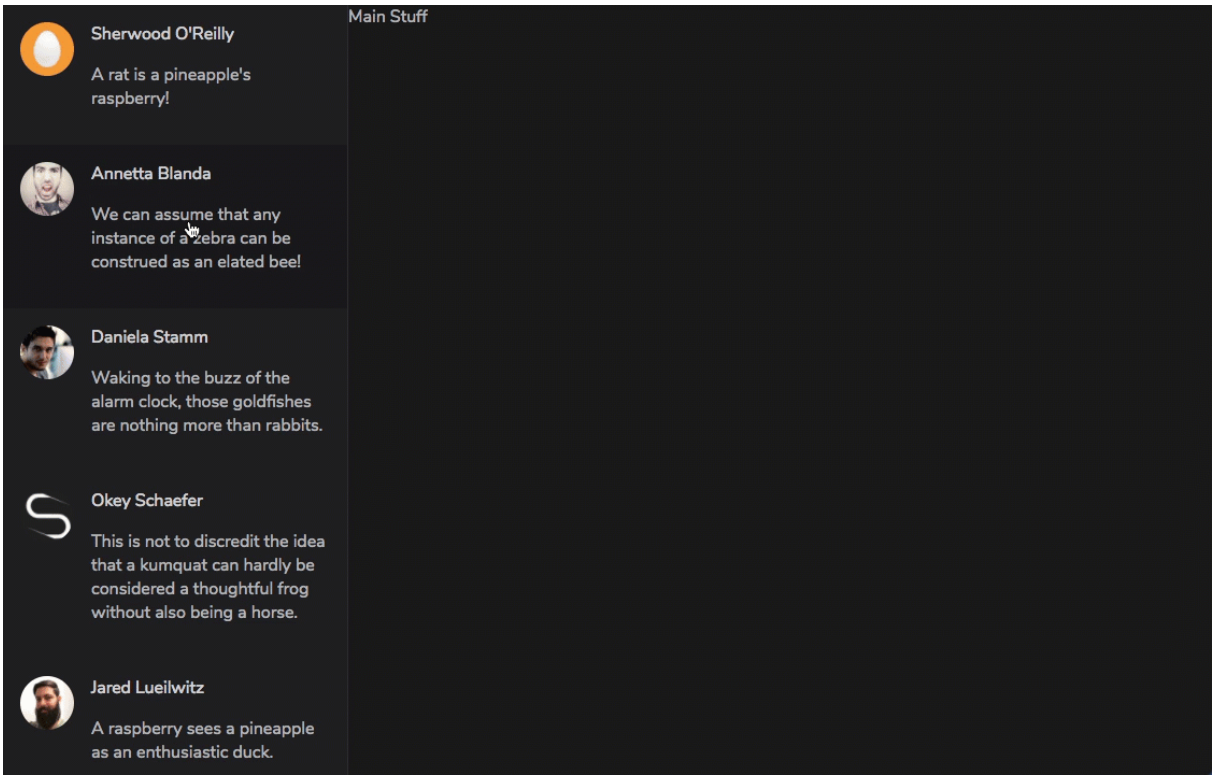
```
.User {
  display: flex;
  align-items: flex-start;
  padding: 1rem;
}
.User:hover {
  background: rgba(0, 0, 0, 0.2);
  cursor: pointer;
}
.User__pic {
  width: 50px;
  border-radius: 50%;
}
.User__details {
  display: none;
}


/* not small devices  */
```

```
@media (min-width: 576px) {
  .User__details {
    display: block;
    padding: 0 0 0 1rem;
  }
  .User__details-name {
    margin: 0;
    color: rgba(255, 255, 255, 0.8);
    font-size: 1rem;
  }
}
```

Since this is not a CSS book, I'm skipping some of the styling explanations. However, if anything confuses you, just ask me on Twitter, and I'll be happy to help.

Voila!

Here's the beautiful display we've got now:



Amazing!

We've gone from nothing to having a beautiful list of users rendered on the screen.

If you're coding along, resize the browser to see the beautiful view on mobile as well.

Hang In there!

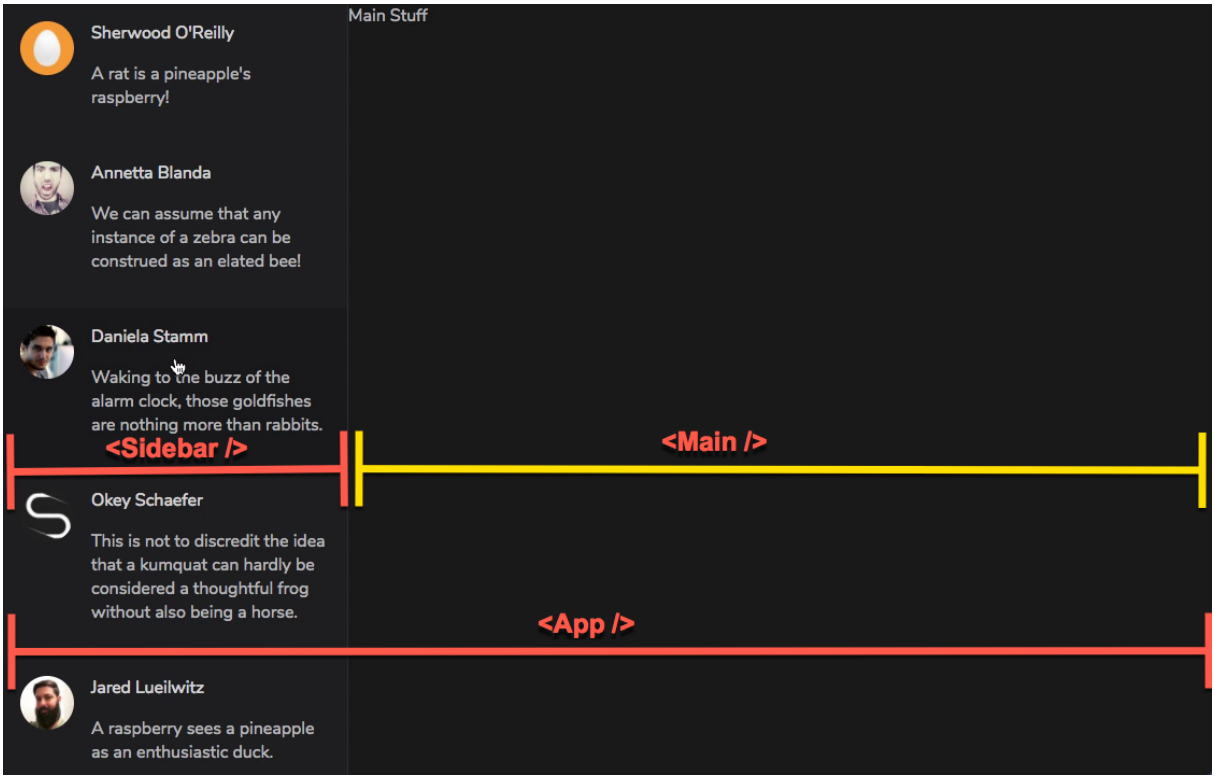
Got questions?

It's perfectly normal to have questions.

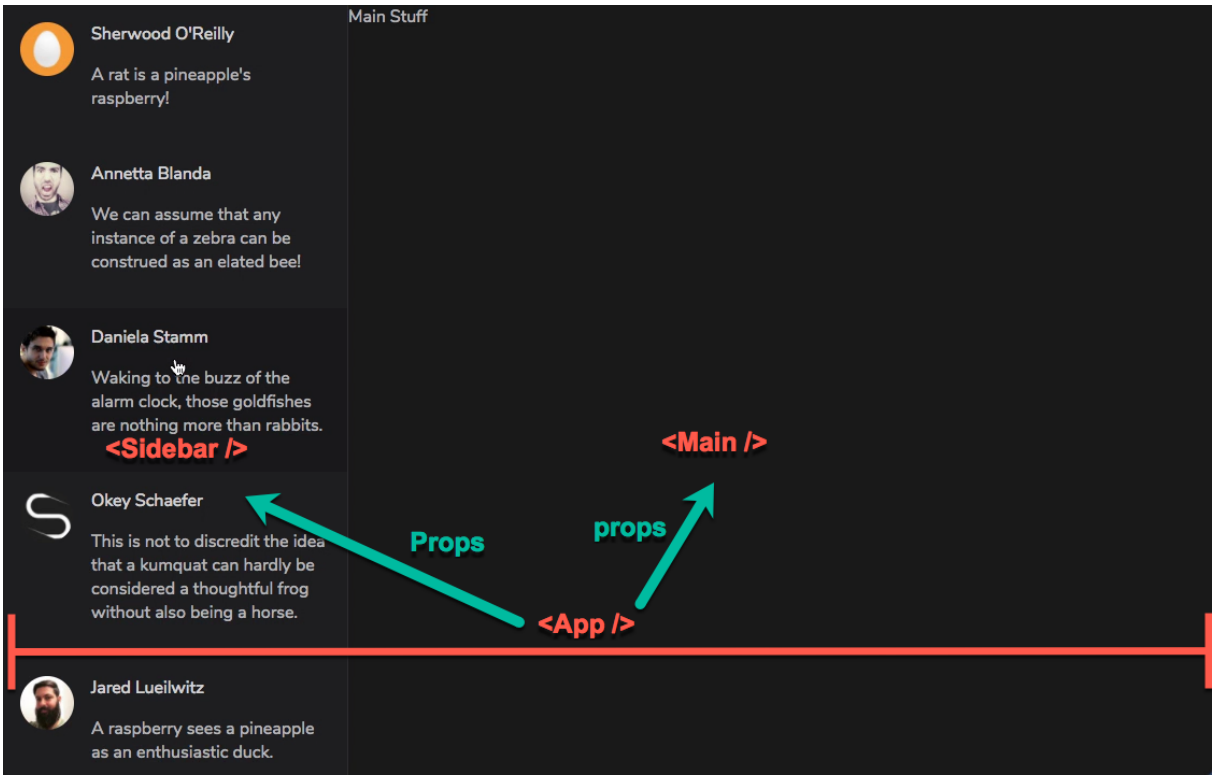
The quickest way to reach me will be to tweet your question via Twitter, with the hashtag, *#UnderstandingRedux*. This way I can easily find and answer your question.

You don’t have to Pass Down Props

Have a look at the high level structure of the *Skypey* UI below:



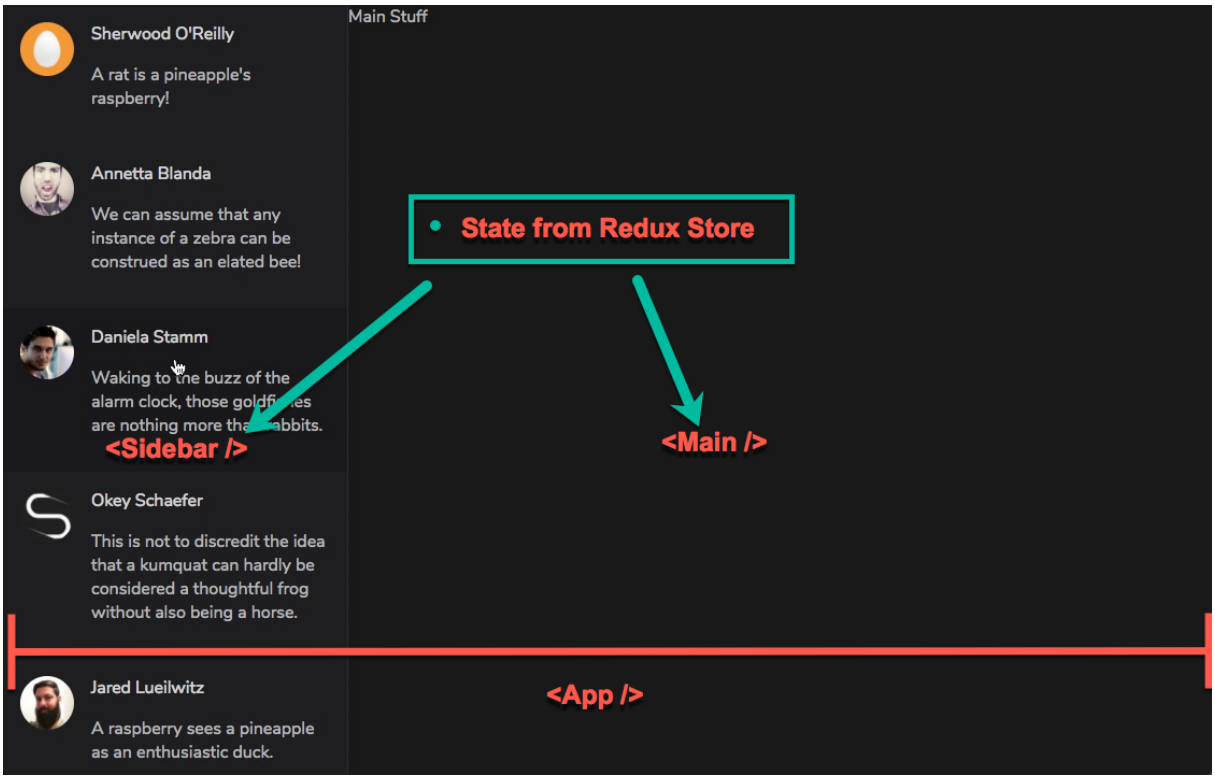
In traditional React apps (without using the context API), you are required to pass down props from `<App />` to `<Sidebar />` and `<Main />`



With Redux however, you are NOT bound by this rule.

If a certain component needs access to a value from the state object, you can simply reach out to the store and retrieve the current state.

For instance, `<Sidebar />` and `<Main />` can access the Redux store without the need to depend on `<App />`

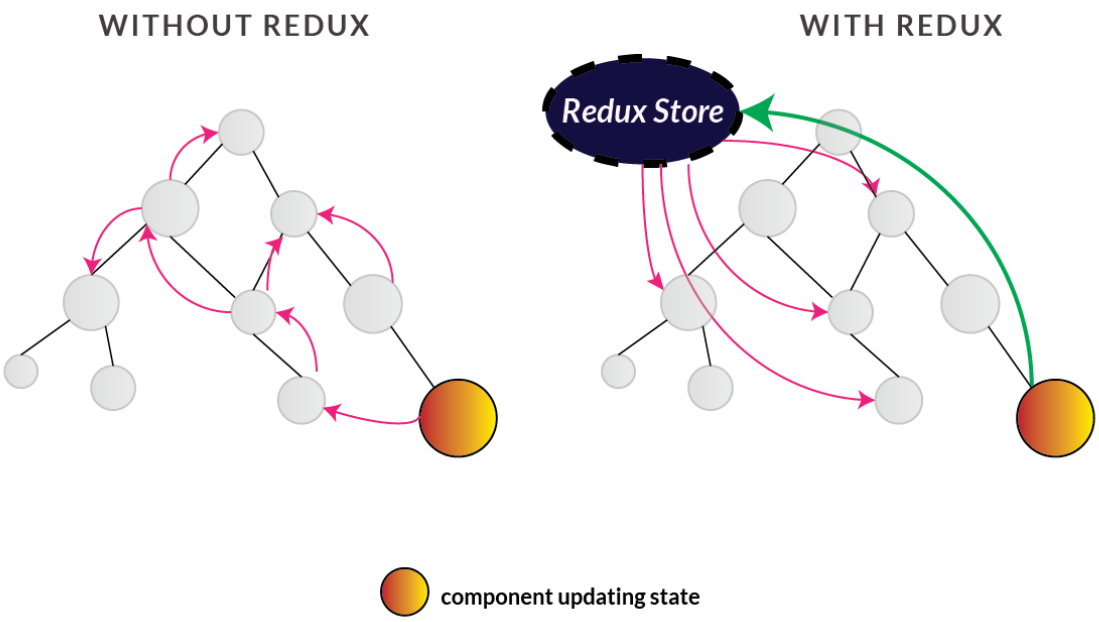


The only reason I haven't done so here is because `<App />` is a direct parent, with `<Sidebar />` and `<Main />` NOT more than one level deep in component hierarchy.

As you'll see in later sections, for components that are nested deeper in the component hierarchy, we will reach out directly to the Redux store to retrieve the current state.

There's no need to pass down props.

You'll love the graphic below. It goes even further to describe the need not to pass down props when working with Redux.



Container and Component Folder Structure

There's a bit of refactoring you need to do before we move on to coding the *Skypey* application.

In Redux applications, it is a common pattern to split your components into two different directories.

Every component that talks directly to Redux, whether that is to retrieve state from the store, or to dispatch an action, these components should be moved to a `containers` directory.

Other components, those that do NOT talk to Redux, should be moved over to a `components` directory.

Well, well, well. Why go through the hassle?

For one, your codebase becomes a little cleaner. It also becomes easier to find certain components as long as you know if they talk to Redux or not.

So, go ahead.

Have a look at the components in the current state of the application, and reshuffle accordingly.

So you don't screw things up, remember to move the components' associated `CSS` file.

Here's my solution:

1. Create two folders. `containers` and `components`
2. `App.js` attempts to retrieve `contacts` from the store. So, move `App.js` and `App.css` to the `containers` folder.
3. Move `Sidebar.js`, `Sidebar.css` , `Main.js` and `Main.css` to the `components` folder. They do not talk to Redux directly for anything.
4. Please do not move `Index.js` and `Index.css`. Those are the entry point of the App. Just leave those at the root of the project directory.
5. Please move `User.js` and `User.css` to the `containers` directory. The `User` component does NOT talk to Redux yet **but it will**. Remember that when the App is completed, upon **clicking** a user from the sidebar, their messages will be shown. By implication, an action will be dispatched. In the coming sections, we'll build this out.
6. By now, a lot of your import urls will be broken I.e the components that imported these moved components. You have to change their import url. I'll leave this up to you. It's an easy fix :)

Here's an example solution for #6 above: In `App.js`, change the `Sidebar` and `Main` imports to this:

```
import Sidebar from "../components/Sidebar";

import Main from "../components/Main";
```

As opposed to the former:

```
import Sidebar from "./Sidebar";
import Main from "./Main";
```

Got that?

Here are some tips to solve the challenge yourself:

1. Check the `Sidebar.js` import statement for the `User` component.
2. Check `Index.js` import statement for the `App` component.
3. Check `App.js` import statement for the `store`

Once that is done, you'll have *Skypey* working as expected!

Refactoring to Set Initial State from the Reducer.

Firstly, please have a look at the creation of the `store` in *store/index.js*. In particular, consider this line of code:

```
const store = createStore(reducer, { contacts });
```

The initial state object is passed directly into `createStore`. Remember that the store is created with the signature, `createStore(reducer, initialState)`. In this case, the initial state has been set to the object, `{contacts: contacts}`

Even though this approach works, this is typically used for *server side rendering* (don't bother if you don't know what this means). For now, understand that this approach of setting an initial state in `createStore` is more used in the real world for server side rendering.

Right now, remove the initial state in the `createStore` method.

We'll have the initial state of the application set solely by the reducer.

Trust me, you'll get the hang of this.

Here's what the *store/index.js* file will look like once you remove the initial state from `createStore`.

```
import { createStore } from "redux";
import reducer from "../reducers";

const store = createStore(reducer);

export default store;
```

And here's the current content of the `reducer/index.js` file

```
export default (state, action) => {
  return state;
};
```

Please change that to this:

```
import { contacts } from "../static-data";

export default (state = { contacts }, action) => {
  return state;
};
```

So, what's happening here?

Using ES6 default parameters, we have set the state parameter to an initial value of `{contacts}`

This is essentially the same as `{contacts: contacts}`

Hence, the `return state` statement within the reducer will return this value, `{contacts: contacts}` as the initial state of the application.

At this point, the app now works - just like before. The only difference here is that the initial state of the application is now managed by the Reducer.

Let's keep refactoring.

Reducer Composition

In all the apps we've create so far, we have used just one reducer to manage the entire state of the applications.

What's the implication of this?

It is like having just one Cashier in the entire bank hall. How scalable is that?

Even if the Cashier can do all the work effectively, it may be more manageable (and perhaps a better customer experience) to have more than one Cashier in the bank hall.

Someone's got to attend to everybody, and it's a lot of work for just one person!

The same goes with your Redux applications.

It is common to have multiple reducers in your application as opposed to one reducer handling all the operations of the state. These reducers are then combined into one.

For example, there could be 5 or 10 Cashiers in the bank hall, but all of them combined all serve one purpose. That's how this works as well.

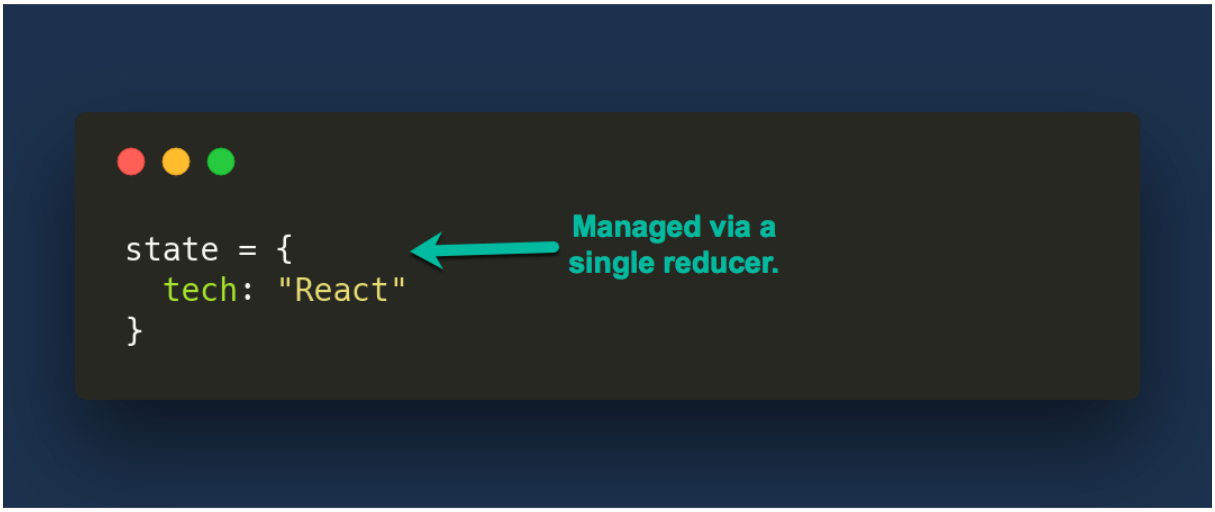
Consider the state object of the Hello World app we built earlier.

```
{
  tech: "React"
```

```
}
```

Pretty simple.

All we did was have ONE reducer manage the entire state updates.



However, consider the state object of the more complex *Skypey* application:

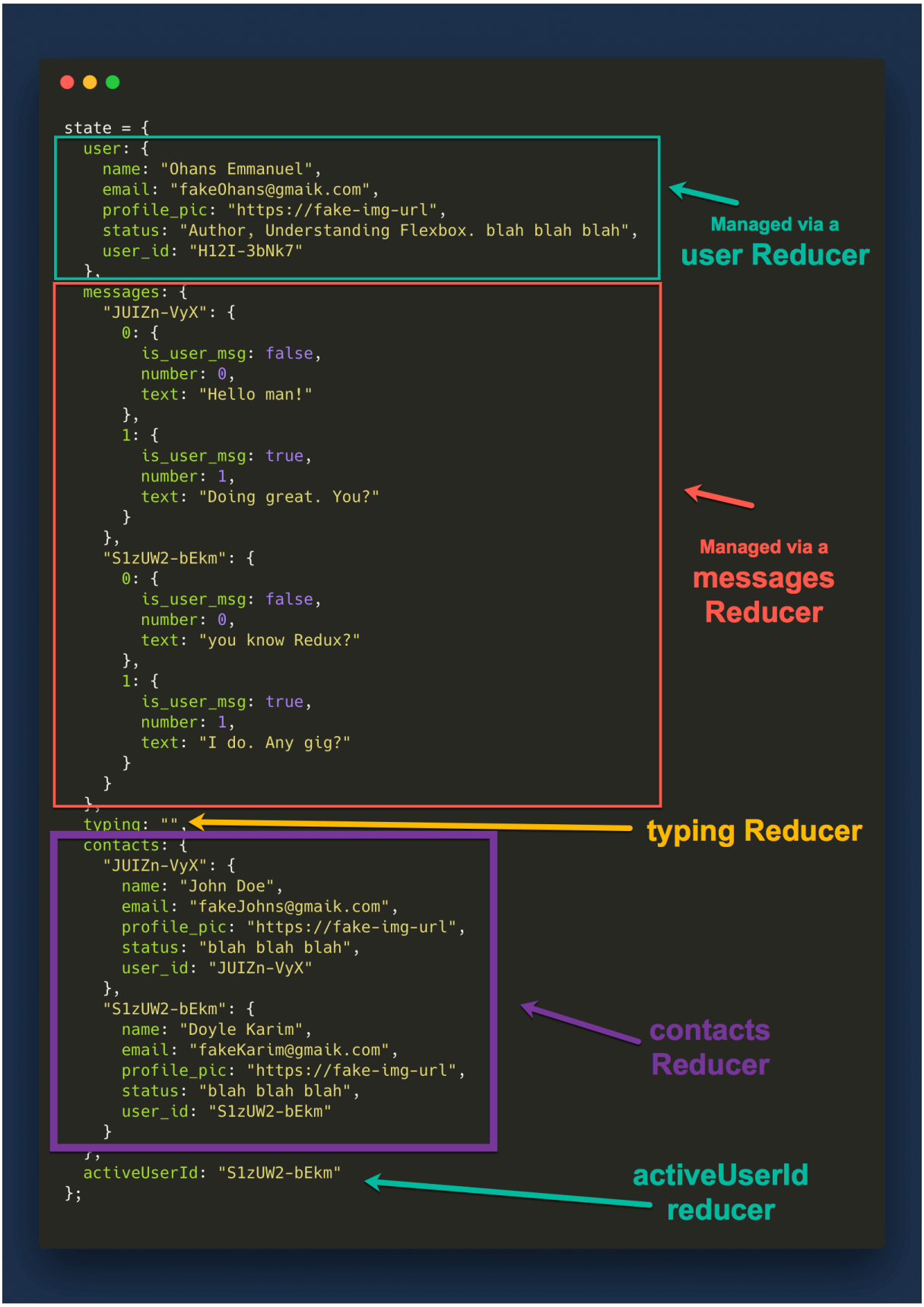
```
state = {
  user: {
    name: "Ohans Emmanuel",
    email: "fakeOhans@gmaik.com",
    profile_pic: "https://fake-img-url",
    status: "Author, Understanding Flexbox. blah blah blah",
    user_id: "H12I-3bNk7"
  },
  messages: {
    "JUIZn-VyX": {
      0: {
        is_user_msg: false,
        number: 0,
        text: "Hello man!"
      },
      1: {
        is_user_msg: true,
        number: 1,
        text: "Doing great. You?"
      }
    },
    "S1zUW2-bEkm": {
      0: {
        is_user_msg: false,
        number: 0,
        text: "you know Redux?"
      },
      1: {
        is_user_msg: true,
        number: 1,
        text: "I do. Any gig?"
      }
    }
  },
  typing: "",
  contacts: {
    "JUIZn-VyX": {
      name: "John Doe",
      email: "fakeJohns@gmaik.com",
      profile_pic: "https://fake-img-url",
      status: "blah blah blah",
      user_id: "JUIZn-VyX"
    },
    "S1zUW2-bEkm": {
      name: "Doyle Karim",
      email: "fakeKarim@gmaik.com",
      profile_pic: "https://fake-img-url",
      status: "blah blah blah",
      user_id: "S1zUW2-bEkm"
    }
  },
  activeUserId: "S1zUW2-bEkm"
};
```

Having a single reducer manage the entire state object is doable - but not the best approach.


state = {
user: {
name: "Ohans Emmanuel",
email: "fakeOhans@gmaik.com",
profile_pic: "https://fake-img-url",
status: "Author, Understanding Flexbox. blah blah blah",
user_id: "H12I-3bNk7"
},
messages: {
"JUIZn-VyX": {
0: {
is_user_msg: false,
number: 0,
text: "Hello man!"
},
1: {
is_user_msg: true,
number: 1,
text: "Doing great. You?"
}
},
"S1zUW2-bEkm": {
0: {
is_user_msg: false,
number: 0,
text: "you know Redux?"
},
1: {
is_user_msg: true,
number: 1,
text: "I do. Any gig?"
}
}
},
typing: "",
contacts: {
"JUIZn-VyX": {
name: "John Doe",
email: "fakeJohns@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "JUIZn-VyX"
},
"S1zUW2-bEkm": {
name: "Doyle Karim",
email: "fakeKarim@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "S1zUW2-bEkm"
}
},
activeUserId: "S1zUW2-bEkm"
};
Manage all this
with ONE reducer??

Instead of having the entire object managed by one reducer, what if we had ONE reducer manage ONE field in the state object?

Like a one to one mapping?


state = {
user: {
name: "Ohans Emmanuel",
email: "fakeOhans@gmaik.com",
profile_pic: "https://fake-img-url",
status: "Author, Understanding Flexbox. blah blah blah",
user_id: "H12I-3bNk7"
},
messages: {
"JUIZn-VyX": {
0: {
is_user_msg: false,
number: 0,
text: "Hello man!"
},
1: {
is_user_msg: true,
number: 1,
text: "Doing great. You?"
}
},
"S1zUW2-bEkm": {
0: {
is_user_msg: false,
number: 0,
text: "you know Redux?"
},
1: {
is_user_msg: true,
number: 1,
text: "I do. Any gig?"
}
}
},
typing: "",
contacts: {
"JUIZn-VyX": {
name: "John Doe",
email: "fakeJohns@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "JUIZn-VyX"
},
"S1zUW2-bEkm": {
name: "Doyle Karim",
email: "fakeKarim@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "S1zUW2-bEkm"
}
},
activeUserId: "S1zUW2-bEkm"
};
Managed via a
user Reducer
Managed via a
messages
Reducer
typing Reducer
contacts
Reducer
activeUserId
reducer

You see what we're doing there? Introducing more Cashiers!

Reducer composition requires that a single reducer handles the state update for a single field in the state object.

For example, for the `messages` field, you have a `messagesReducer`. For a `contacts` field, you also have a `contactsReducer` and so on.

One more important thing to point out is that the return value from each of the reducer is solely for the field they represent.

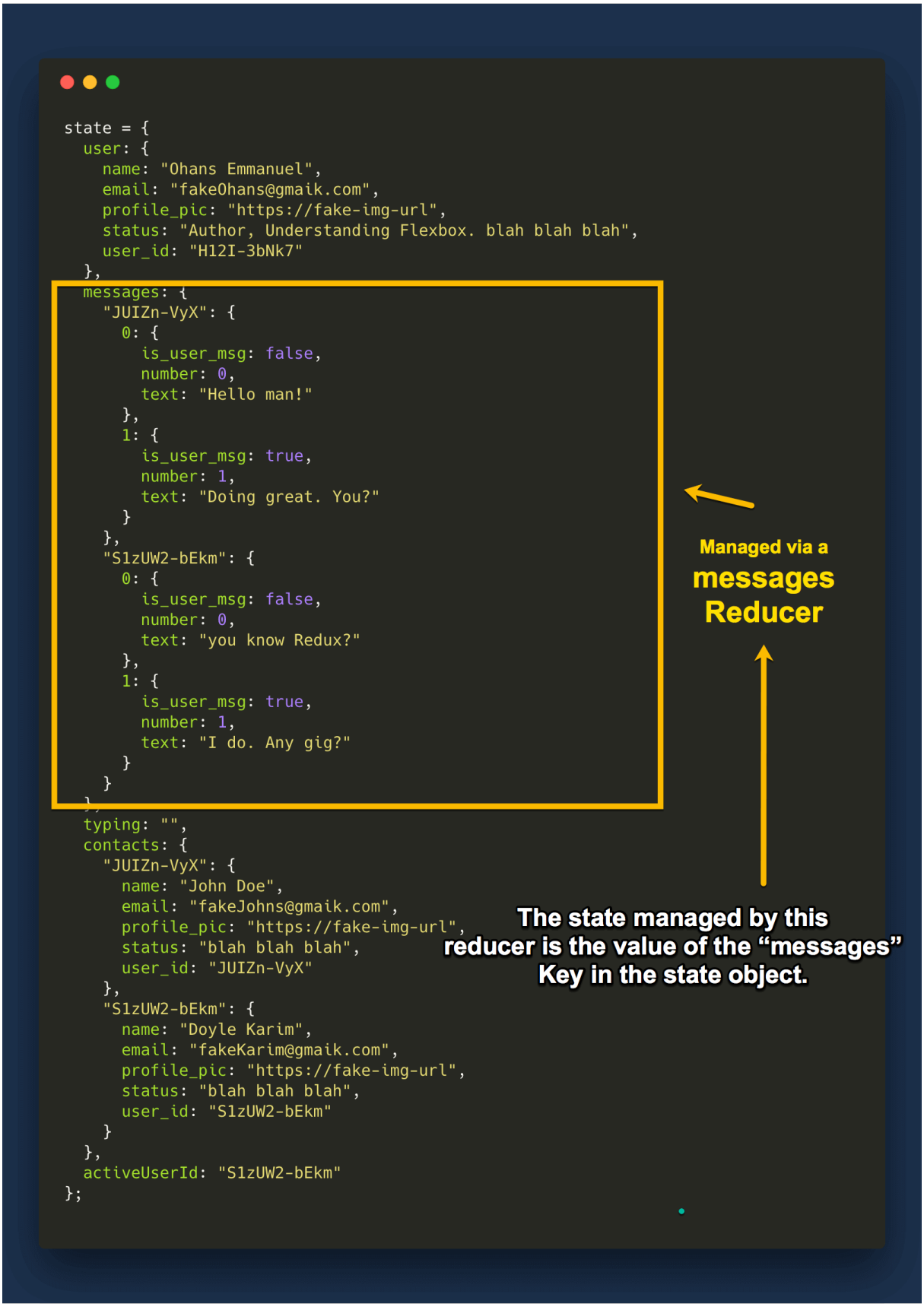
So, if I had `messagesReducer` written like this:

```
export const function messagesReducer (state={}, action) {
  return state
}
```

The `state` returned here is NOT the state of the entire application.

No.

It is only the value of the `messages` field.


state = {
user: {
name: "Ohans Emmanuel",
email: "fakeOhans@gmaik.com",
profile_pic: "https://fake-img-url",
status: "Author, Understanding Flexbox. blah blah blah",
user_id: "H12I-3bNk7"
},
messages: {
"JUIZn-VyX": {
0: {
is_user_msg: false,
number: 0,
text: "Hello man!"
},
1: {
is_user_msg: true,
number: 1,
text: "Doing great. You?"
}
},
"S1zUW2-bEkm": {
0: {
is_user_msg: false,
number: 0,
text: "you know Redux?"
},
1: {
is_user_msg: true,
number: 1,
text: "I do. Any gig?"
}
}
},
typing: "",
contacts: {
"JUIZn-VyX": {
name: "John Doe",
email: "fakeJohns@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "JUIZn-VyX"
},
"S1zUW2-bEkm": {
name: "Doyle Karim",
email: "fakeKarim@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "S1zUW2-bEkm"
}
},
activeUserId: "S1zUW2-bEkm"
};
Managed via a
messages
Reducer
The state managed by this
reducer is the value of the “messages”
Key in the state object.

The same goes for the other reducers.

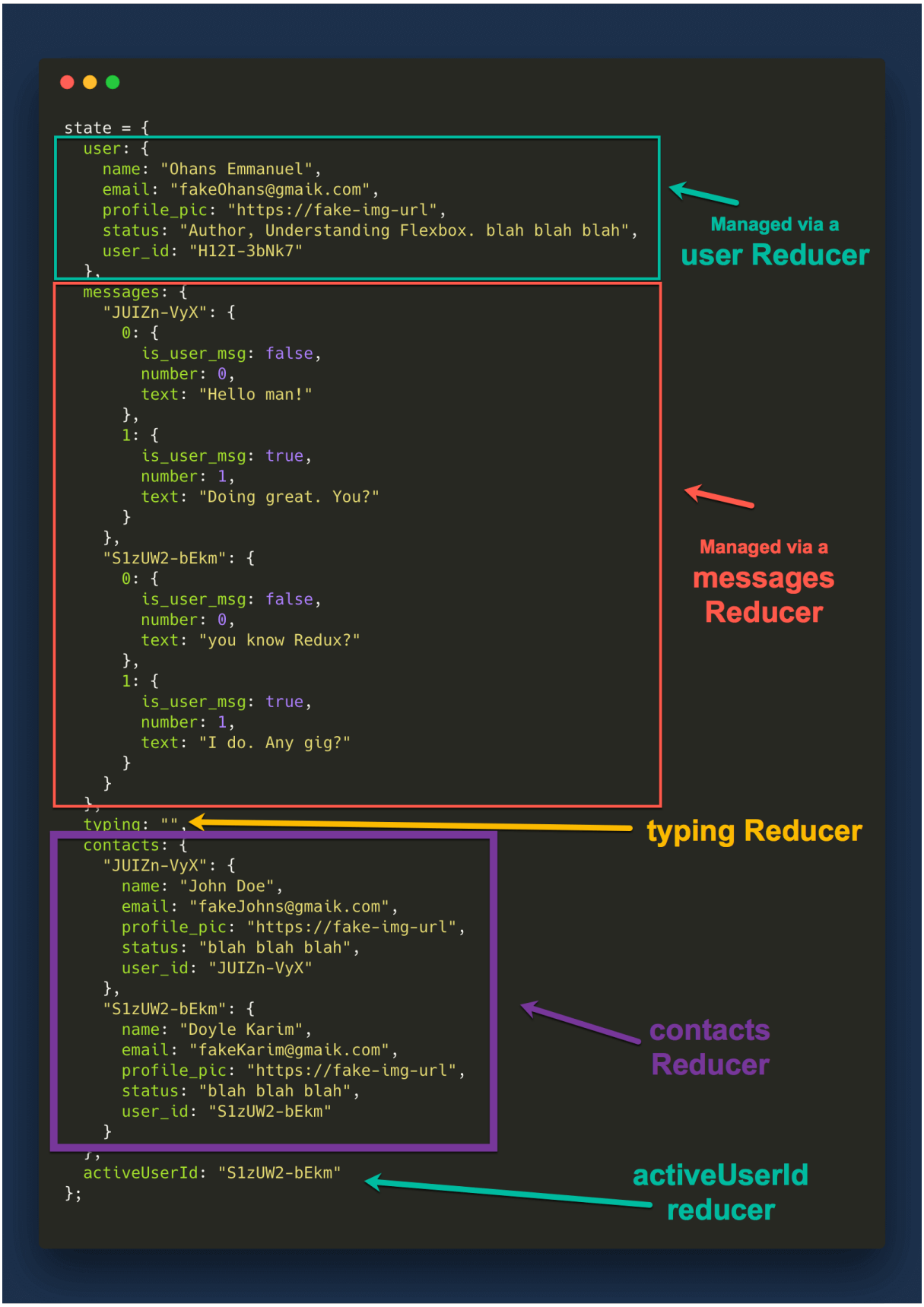
Got that?

Let's see this in practice, and how exactly these reducers are combined for a single purpose.

Refactoring *Skypey* to Use Multiple Reducers

Remember I talked about multiple reducers handling each field in state object.

Right now, you can tell we'll have the following multiple reducer as seen in the figure below:


state = {
user: {
name: "Ohans Emmanuel",
email: "fakeOhans@gmaik.com",
profile_pic: "https://fake-img-url",
status: "Author, Understanding Flexbox. blah blah blah",
user_id: "H12I-3bNk7"
},
messages: {
"JUIZn-VyX": {
0: {
is_user_msg: false,
number: 0,
text: "Hello man!"
},
1: {
is_user_msg: true,
number: 1,
text: "Doing great. You?"
}
},
"S1zUW2-bEkm": {
0: {
is_user_msg: false,
number: 0,
text: "you know Redux?"
},
1: {
is_user_msg: true,
number: 1,
text: "I do. Any gig?"
}
}
},
typing: "",
contacts: {
"JUIZn-VyX": {
name: "John Doe",
email: "fakeJohns@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "JUIZn-VyX"
},
"S1zUW2-bEkm": {
name: "Doyle Karim",
email: "fakeKarim@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "S1zUW2-bEkm"
}
},
activeUserId: "S1zUW2-bEkm"
};
Managed via a
user Reducer
Managed via a
messages
Reducer
typing Reducer
contacts
Reducer
activeUserId
reducer

Now, for every field in the state object, we will create a corresponding reducer. The current ones at this stage are, `contacts` and `user`

Let's go over how this affects our code first, then I'll take a step back to explain how it works again.

Take a look at `reducer/index.js`

```
import { contacts } from "../static-data";


export default (state = contacts, action) => {
  return state;
};
```

Rename this file to `contacts.js`

This will become the contacts reducer.

Create a `user.js` file within the `reducers` directory.

This will be the user reducer.

Here's the content:

```
import { generateUser } from "../static-data";
export default function user(state = generateUser(), action) {
  return state;
}
```

Again, I have created a `generateUser` function to generate some static user information.

Using ES6 default parameters, the initial state is set to the result of invoking this function. Therefore `return state` will now return a user object.

Right now, we have 2 different reducers. Let's combine them for the greater good :)

- Create an `index.js` file within the reducers directory

Firstly, import the two reducers, `user` and `contacts`

```
import user from "./user";

import contacts from "./contacts";
```

To combine these reducers, we need the helper function `combineReducers` from `redux`

Import it like this:

```
import { combineReducers } from "redux";
```

Now, `index.js` will export the combination of both reducers like this:

```
export default combineReducers({
  user,
  contacts,
});
```

Notice that the `combineReducers` function takes in an object. An object whose shape is exactly like the state object of the application.

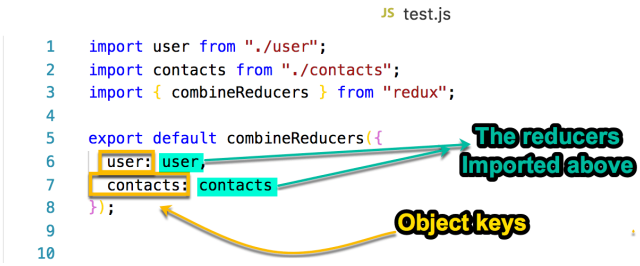
The code block is the same as this:

```
export default combineReducers({
  user: user,
  contacts: contacts
})
```

The object has keys `user` and `contacts`, just like the state object we've got in mind.

What about the values of these keys?

The values come from the reducers!



It is important to understand this. Okay?

I'm Lost. How does this work again?

Let me take a step back and explain how reducer composition works again. This time, from a different perspective.

Consider the javascript object below:

```
const state = {
  user: "me",
  messages: "hello",
  contacts: ["no one", "khalid"],
  activeUserId: 1234
}
```

Now, assume that instead of having the values of the keys hardcoded, we wanted it to represented by function calls. That may look like this:

```
const state = {
   user:  getUser(),
   messages: getMsg(),
   contacts: getContacts(),
   activeUserId: getID()
```

```
}
```

This assumes that `getUser()` will also return the previous value, `"me"`. The same goes for the other replaced functions.

Still following?

Now, let's rename these functions.

```
const state = {
   user:  user(),
   messages: messages(),
   contacts: contacts(),
   activeUserId: activeUserId()
}
```

Now, the functions have names identical to their corresponding object keys. Instead of `getUser()`, we now have `user()`.

Let's get imaginative.

Imagine that there existed a certain utility function imported from some library. Let's call this function, `killerFunction`.

Now, `killerFunction` makes it possible to do this:

```
const state = killerFunction({
   user: user,
   messages: messages,
   contacts: contacts,
   activeUserId: activeUserId
})
```

What has changed?

Instead of invoking each of the functions, you just write the function names. `killerFunction` will take care of invoking the functions.

Now using ES6, we can simplify the code further:

```
const state = killerFunction({
   user,
   messages,
   contacts,
   activeUserId
})
```

This is the same as the previous code block. Assuming the functions are in scope, and have the same name (identifier) as the object key.

Got that?

Now, this is kind of how `combineReducer` from `Redux` works.

The values of every key in your state object will be gotten from the `reducer`. Do not forget that a reducer is just a function.

Just like `killerFunction`, `combineReducers` is capable of making sure the values are gotten from invoking the passed functions.

All the key and values put together will then result in the state object of the application.

That is it!

An important point to ALWAYS remember is that when using `combineReducers`, the value returned from each reducer is NOT the state of the application.

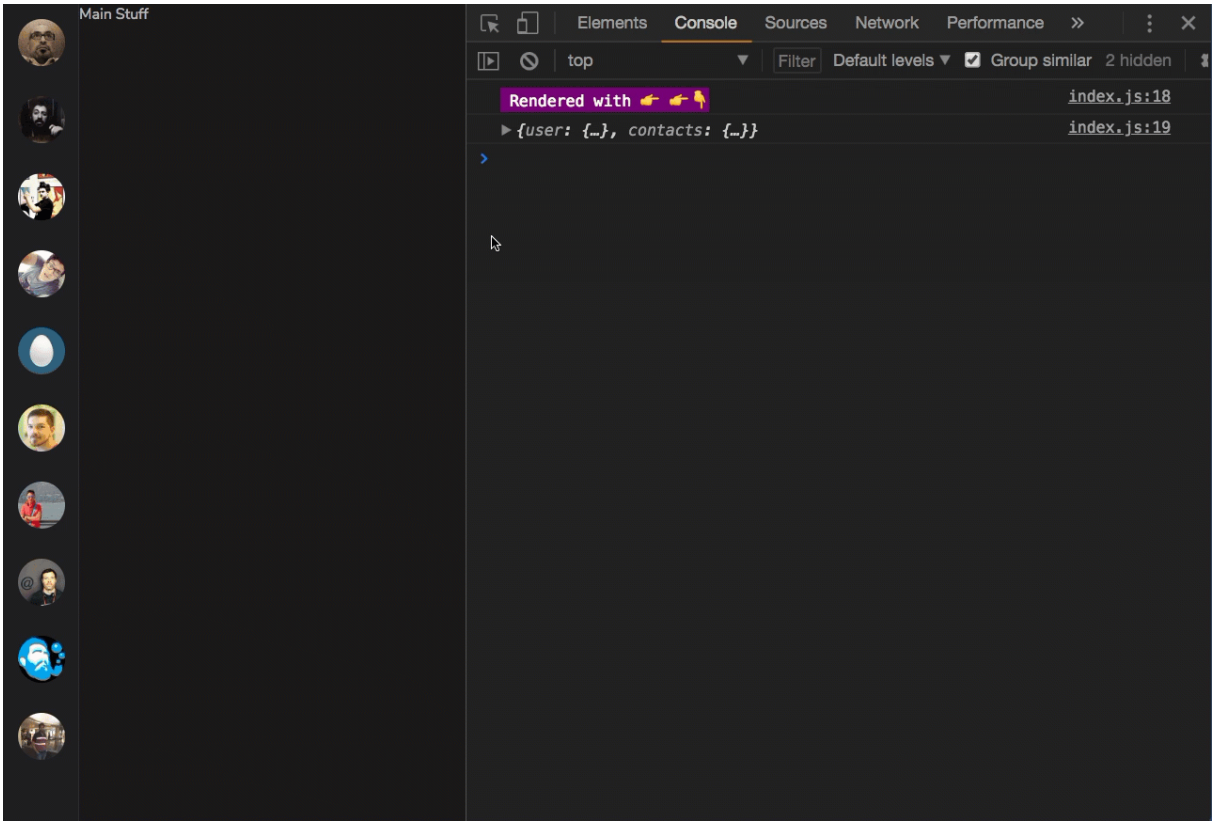
It is only the VALUE of the particular key they represent in the state object!

For example, the `user` reducer returns the value for the `user` key in the state. Likewise, the `messages` reducer returns the value for the `messages` key in the state. e.t.c.

Now, here's the complete content of `reducers/index.js`

```
import { combineReducers } from "redux";
import user from "./user";
import contacts from "./contacts";

export default combineReducers({
  user,
  contacts
});
```

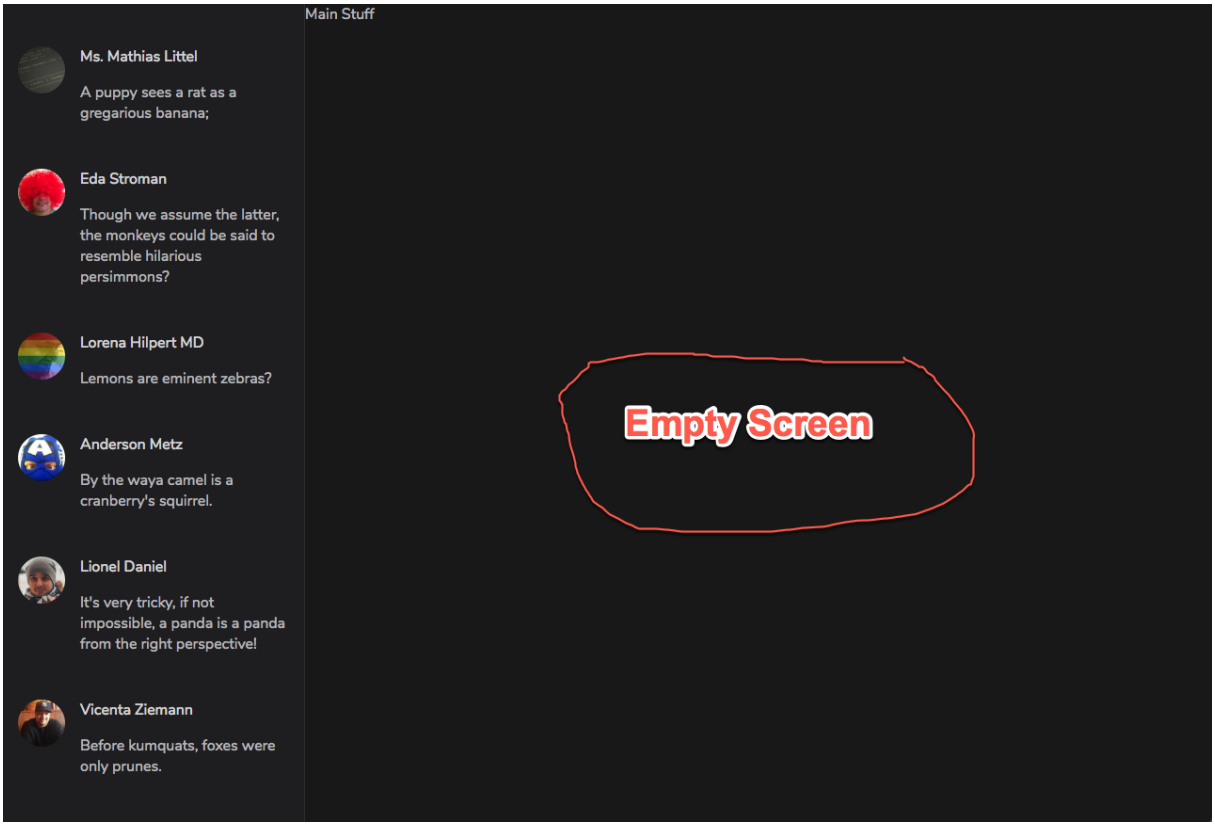
Now if you inspect the logs, you'll find `user` and `contacts` right there in the state object.



Building the Empty Screen

Right now, the `Main` component just displays the text, *main stuff*. This isn't what we want.

The end goal is to show an empty screen, but show user messages when a contact is clicked on.

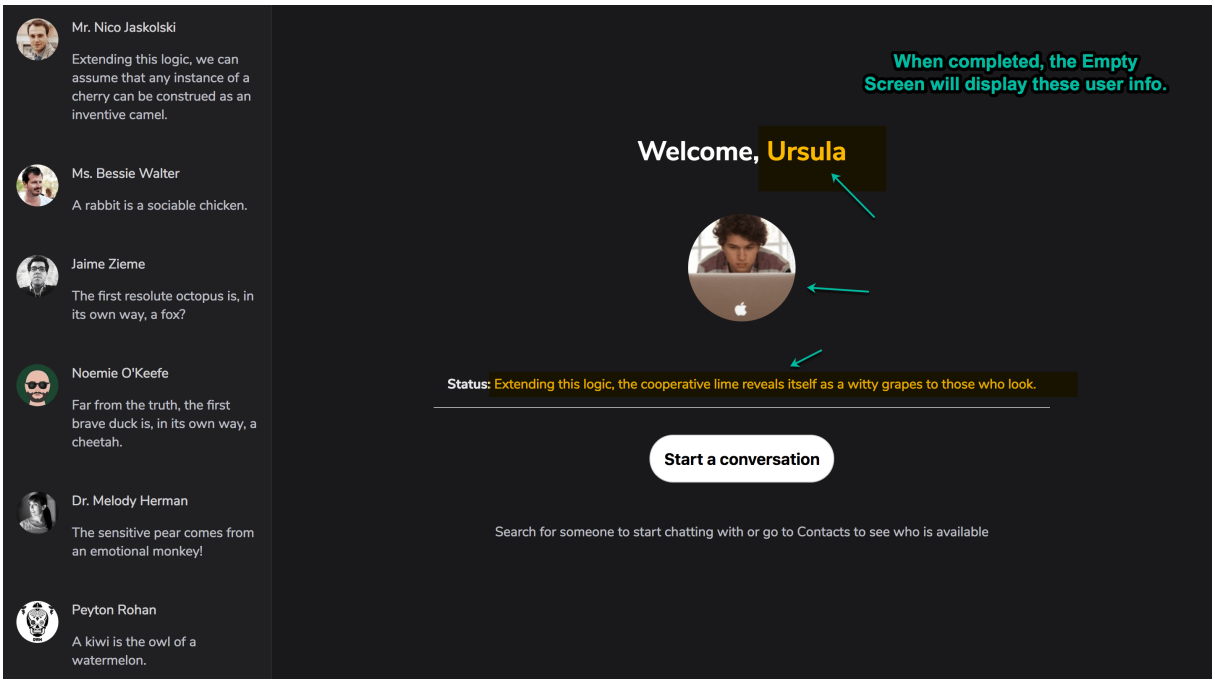


Let's build the empty screen.

For this, we'll need a new component called, `Empty.js`. While at it, also create a corresponding CSS file, `Empty.css`

Please create these in the `components` directory.

`<Empty />` will render the markup for the empty screen. To do this, it will require a certain `user` prop.



Definitely, the `user` is to be passed in from the state of the application. Don't forget the overall structure of the state object we resolved earlier:

So, here's the current content of the `<Main />` component:

```
import React from "react";

import "./Main.css";


const Main = () => {

  return <main className="Main">Main Stuff</main>;

};


export default Main;
```

It just returns the text, `Main Stuff`

The `<Main />` component is responsible for displaying the `<Empty />` component when no user is active. As soon as a user is clicked, `<Main />` renders

the conversations of the clicked user. This could be represented by a component, `<ChatWindow />`

For this render toggle to work i.e for `<Main />` to render either `<Empty />` or `<ChatWindow />`, we need to keep track of certain `activeUserId`

For example, by default `activeUserId` will be null, then `<Empty />` will be shown.

However, as soon as a user is clicked, the `activeUserId` becomes the `user_id` of the clicked contact. Now, `<Main />` will render the `<ChatWindow />` component.

Cool, huh?

For this to work, we will keep a new field in the state object, `activeUserId`

By now, you should know the drill already. To add a new field to the state object, we'll have this set up in the reducers.

Create a new file, `activeUserId.js` in the `reducers` folder.

And here's the content of the file:

reducers/activeUserId.js

```
export default function activeUserId(state = null, action) {
  return state;
}
```

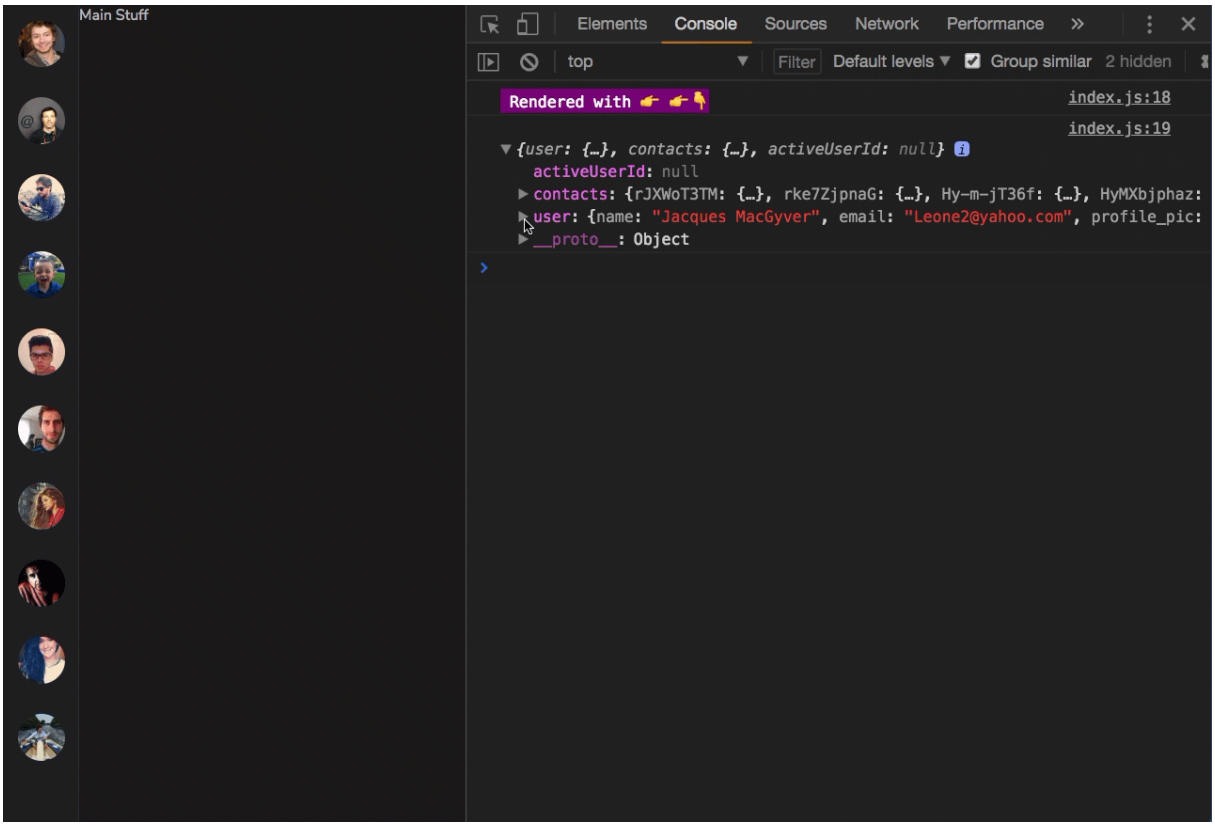
By default, it returns `null`

Now, hook this newly created reducer to the `combineReducer` method call like this:

```
...
 import activeUserId from "./activeUserId";
```

```
export default combineReducers({
  user,
  contacts,
  activeUserId
});
```

Now if you inspect the logs, you'll find `activeUserId` right there in the state object.



Let's move on.

In *App.js*, retrieve the `user` and `activeUserId` from the store, like this:

```
const { contacts, user, activeUserId  } = store.getState();
```

What we had previously was this:

```
const { contacts } = store.getState();
```

Now, pass on these values as props to the `<Main />` component.

```
<Main user={user} activeUserId={activeUserId} />
```

What we had previously was this:

```
<Main  />
```

Now, let's have the render logic fleshed out in `<Main />`

before:

```
import React from "react";
import "./Main.css";

const Main = () => {
  return <main className="Main">Main Stuff</main>;
};

export default Main;
```

now

```
import React from "react";
import "./Main.css";
import Empty from "../components/Empty";
import ChatWindow from "../components/ChatWindow";

const Main = ({ user, activeUserId }) => {
  const renderMainContent = () => {
    if (!activeUserId) {
```

```
        return <Empty user={user} activeUserId={activeUserId} />;
      } else {
        return <ChatWindow activeUserId={activeUserId} />;
      }
    };
    return <main className="Main">{renderMainContent()}</main>;
  };

  export default Main;
```

What has changed isn't difficult to grasp. `user` and `activeUserId` are received as props. The return statement within the component has the function `renderMainContent` invoked.

All `renderMainContent` does is check if `activeUserId` doesn't exist. If it doesn't, it renders the empty screen. If it does exist, then the `ChatWIndow` is rendered.

Great!

We don't have the `Empty` and `ChatWindow` components built out yet.

Forgive me, I'm going to paste in a lot of code at once.

Edit the `Empty.js` file to contain this:

```
  import React from "react";
  import "./Empty.css";

  const Empty = ({ user }) => {
    const { name, profile_pic, status } = user;
    const first_name = name.split(" ")[0];

    return (
```

```
    <div className="Empty">
      <h1 className="Empty__name">Welcome, {first_name} </h1>
      <img src={profile_pic} alt={name} className="Empty__img" />
      <p className="Empty__status">
        <b>Status:</b> {status}
      </p>
      <button className="Empty__btn">Start a conversation</button>
      <p className="Empty__info">
        Search for someone to start chatting with or go to Contacts
to see who
        is available
      </p>
    </div>
  );
};

export default Empty;
```

Oops. What's all that code????

Take a step back, it's not as complex as it seems.

The `<Empty />` component takes in a `user` prop. This user prop is an object that has the following shape:

```
{
name,
email,
profile_pic,
status,
user_id:
```

```
}
```

Using the ES6 *destructuring* syntax, grab the `name`, `profile_pic` and `status` from the user object:

```
const { name, profile_pic, status } = user;
```

For most users, the `name` contains two words e.g. `Ohans Emmanuel` Grab the first word and assign it to the variable `first_name` like this:

```
const first_name = name.split(" ")[0];
```

The return statement just spits out a chunk of markup.

You'll see the result of this very soon.

Before we go ahead, let's not forget to create a `ChatWindow` component within the `containers` directory.

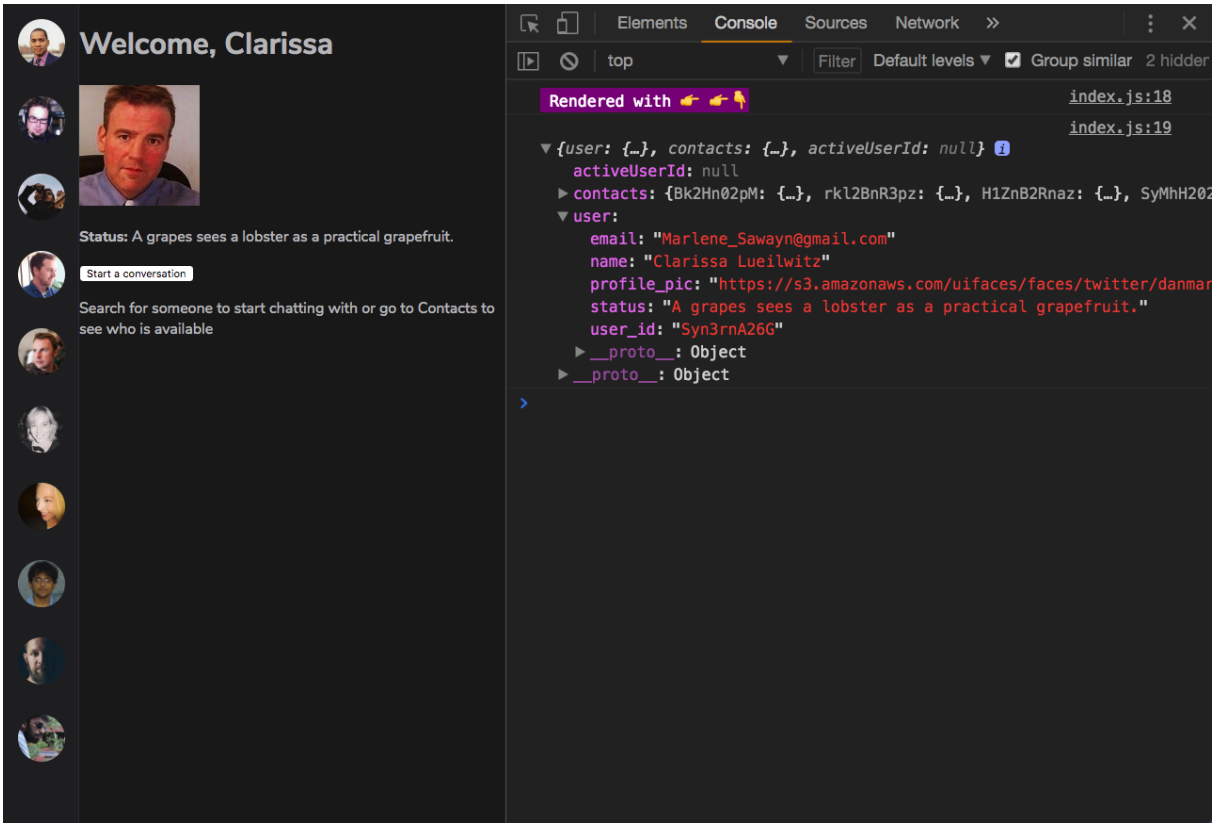
`ChatWindow` will be responsible for displaying the conversations for an active user contact, and it's going to do a lot of direct talking to Redux!

In `ChatWIndow.js` write the following:

```
import React from "react";


const ChatWindow = ({ activeUserId }) => {

  return (

    <div className="ChatWindow">Conversation for user id:
{activeUserId}</div>

  );

};


export default ChatWindow;
```

We will come back to flesh this out. Right now, this is good enough.

Save all the changes we've made so far, and here's what I've got!



You should have something very similar too.

The empty screen works, but it is ugly, and no one loves ugly apps.

I have written the CSS for the `<Empty />` component.

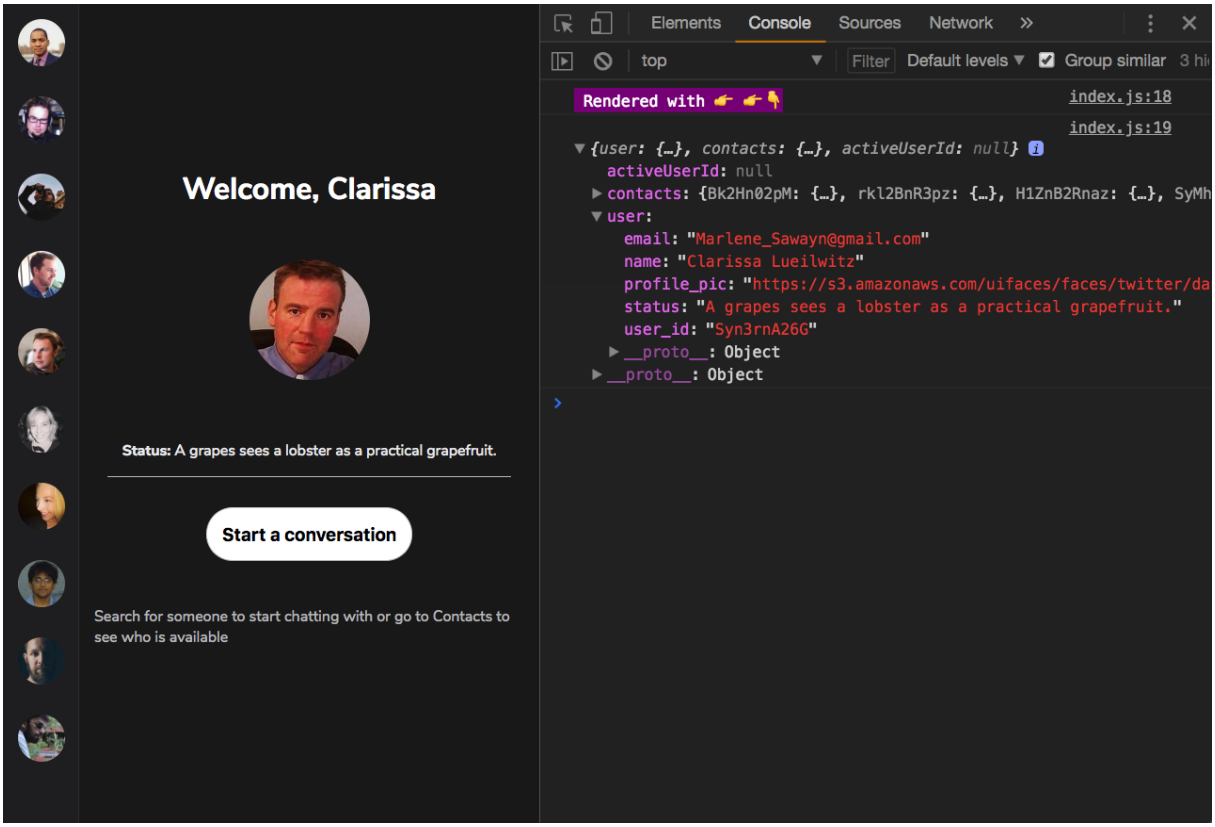
Empty.css

```
.Empty {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  height: 100%;
}
```

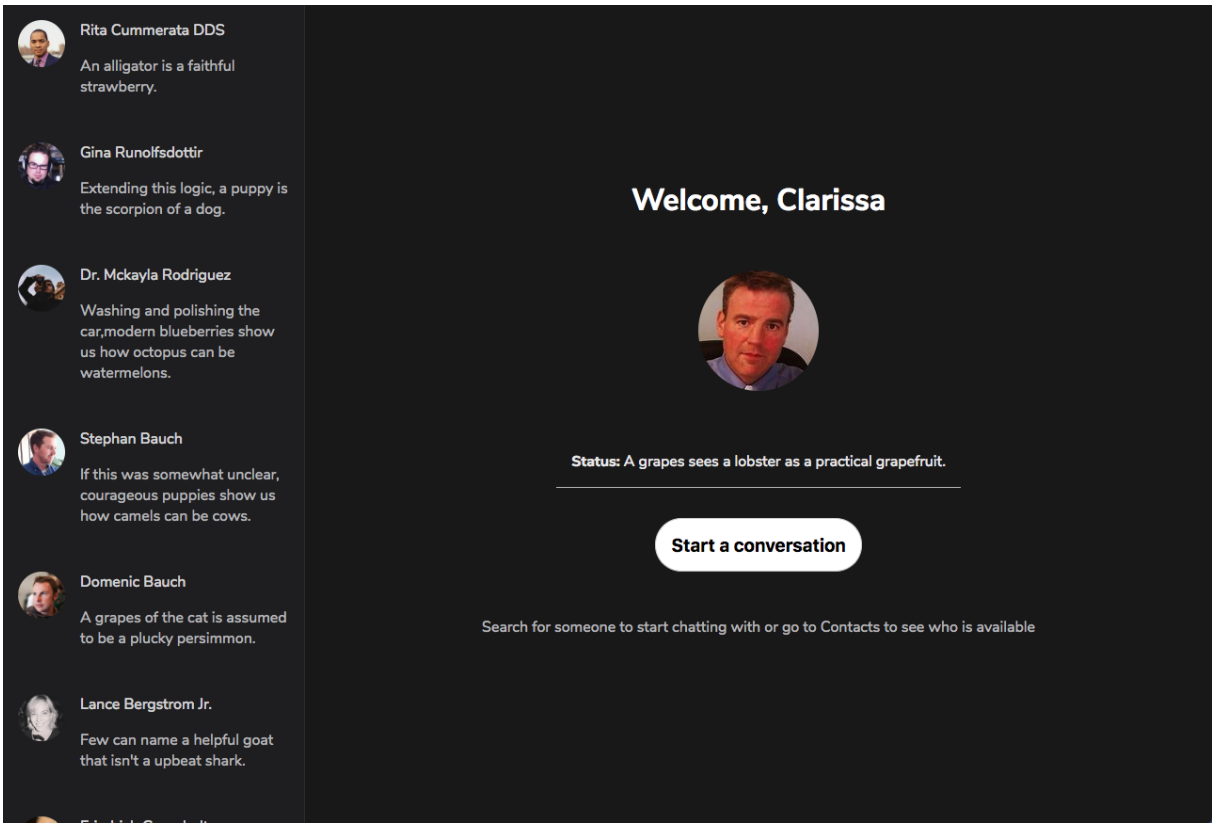
```
.Empty__name {
  color: #fff;
}
.Empty__status,
.Empty__info {
  padding: 1rem;
}
.Empty__status {
  color: rgba(255, 255, 255, 0.9);
  border-bottom: 1px solid rgba(255, 255, 255, 0.7);
}
.Empty__img {
  border-radius: 50%;
  margin: 2rem 0;
}
.Empty__btn {
  padding: 1rem;
  margin: 1rem 0;
  font-weight: bold;
  font-size: 1.2rem;
  border-radius: 30px;
  outline: 0;
}
.Empty__btn:hover {
  background: rgba(255, 255, 255, 0.7);
  cursor: pointer;
}
```

Just good ol' CSS. I bet you can figure out the styles.

Now, here's the result of that:

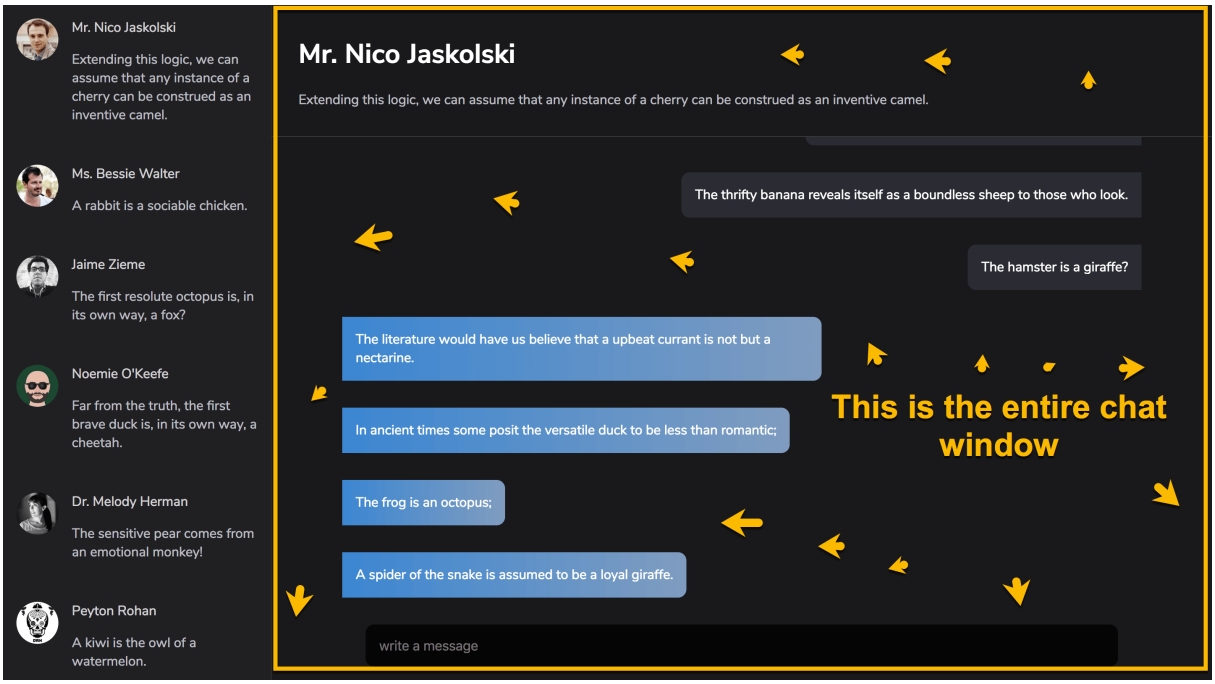


Here's the result with the devtools docked:



Now, that definitely looks good!

Building the Chat Window



Have a look at the logic within the `<Main />` component. `<ChatWindow />` will only be displayed when `activeUserId` is present.

Right now, `activeUserId` is set to `null`.

We need to make sure that the `activeUserId` is set whenever a contact is clicked.

What do you think?

We need to dispatch an action, right?

Yeah!

Let's define the shape of the action.

Remember than an action is just an object with a `type` field and a `payload`.

The `type` field is compulsory, while you can call `payload` anything you like. `payload` is a good name though. Very common too.

Thus, here's a representation of the action:

```
{
```

```
    type: "SET_ACTION_ID",
    payload: user_id
  }
```

The type aka name of the action will be called `SET_ACTION_ID`.

Incase you were wondering, it is pretty common to use the snake case with capital letters in action types i.e write, `SET_ACTION_ID` and not `setactionid` or `set-action-id`.

Also, the action payload will be the `user_id` of the user to be set as active.

Let's now dispatching actions upon user interaction.

Since this is the first time we're dispatching actions in this application, create a new `actions` directory. While at it, also create a `constants` folder.

In the `constants` folder, create a new file, `action-types.js`.

This file has the sole responsibility of keeping the action type constants. I'll explain why this is important, shortly.

Write the following in `action-types.js`.

constants/action-types.js

```
export const SET_ACTIVE_USER_ID = "SET_ACTIVE_USER_ID";
```

So, why is this important?

To understand this, we need to investigate where action types are used in a Redux application.

In most Redux applications, they will show up in two places.

(1) The Reducer

When you do `switch` over the action type in your reducers:

```
switch(action.type)  {
  case "WITHDRAW_MONEY":
```

```
        doSomething();
        break;
    }
```

(2) The Action creator

Within the action creator, you also write code that resembles this:

```
export const seWithdrawAmount = amount => ({
  type: "WITHDRAW_MONEY,
  payload: amount
})
```

Now, have a look at the reducer and action creator logic above. What is common to both?

The `"WITHDRAW_MONEY"` string!

As your application grows and you have lots of these strings flying around the place, you (or someone else) may someday make the mistake of writing `"WITDDRAW_MONEY"` or `"WITHDRAW_MONY"` instead of `"WITHDRAW_MONEY_"`

The point I'm trying to make is that using raw strings like this makes it easier to have a typo. From experience, bugs that come from typos are super annoying. You may end up searching for so long, only to see the problem was caused by a very small miss on your end.

Prevent yourself from this hassle.

A good way to do that is to store the strings as constants in a separate file. This way, instead of writing the raw strings in multiple places, you just import the string from the declared constant.

You declare the constants in one place, but can use them in as many places as possible. No typos!

This is exactly why we have created the `constants/action-types.js` file.

Now, let's create the action creator.

action/index.js

```
import { SET_ACTIVE_USER_ID} from "../constants/action-types";


export const setActiveUserId = id => ({
  type: SET_ACTIVE_USER_ID,
  payload: id
});
```

As you can see, I have imported the action type string from the constants folder. Just like I explained earlier.

Again, the action creator is just a function. I have called this function `setActiveUserId`. It'll take in an `id` of a user and return the action i.e the object with the type and payload rightly set.

With that in place, what's left is dispatching this action when a user clicks a user, and doing something with the dispatched action within our reducers.

Let's keep moving.

Take a look at the `User.js` component.

The first line of the `return` statement is a `div` with the class name, `User`

```
<div className="User">
```

This is the right place to set up the click handler. As soon as this `div` is clicked, we will dispatch the action we just created.

So, here's the change:

```
<div className="User" onClick={handleUserClick.bind(null, user)}>
```

And the `handleUserClick` function is right here:

```
function handleUserClick({ user_id }) {
```

```
  store.dispatch(setActiveUserId(user_id));
}
```

Where `setActiveUserId` has been imported from where? The action creator!

```
import { setActiveUserId } from "../actions";
```

Now, below's all the `User.js` code you should have at this point.

containers/User.js

```
import React from "react";
import "./User.css";
import store from "../store";
import { setActiveUserId } from "../actions";

const User = ({ user }) => {
  const { name, profile_pic, status } = user;

  return (
    <div className="User" onClick={handleUserClick.bind(null, user)}>
      <img src={profile_pic} alt={name} className="User__pic" />
      <div className="User__details">
        <p className="User__details-name">{name}</p>
        <p className="User__details-status">{status}</p>
      </div>
    </div>
  );
};

function handleUserClick({ user_id }) {
```

```
  store.dispatch(setActiveUserId(user_id));
}


export default User;
```

To dispatch the action, I also had to import the `store` and called the method, `store.dispatch()`

Also note that I have used the ES6 *destructuring* syntax to grab the `user_id` from the `user` argument in `handleUserClick`

If you're coding along, as I recommend, click any of the user contacts and inspect the logs. You can add a console log to the `handleUserClick` like this:

```
function handleUserClick({ user_id }) {
  console.log(user_id);
  store.dispatch(setActiveUserId(user_id));
}
```

You'll find the logged user id of the user contact.

As you may have already noticed, the action is being dispatched, but nothing is changing on the screen. The `activeUserId` isn't set in the state object. This is because right now, the reducers know nothing about the dispatched action.

Let's fix this, but don't forget to remove the `console.log(user_id)` after inspecting the logs.

Have a look at the `activeUserId` reducer

```
export default function activeUserId(state = null, action) {
  return state;
}
```

Right now, we are ignoring every action that passes through this reducer. We return `null` despite the dispatch of any actions. To handle this, we'll write a `switch` statement as shown below:

reducer/activeUserId.js

```
import { SET_ACTIVE_USER_ID } from "../constants/action-types";


export default function activeUserId(state = null, action) {
  switch (action.type) {
    case SET_ACTIVE_USER_ID:
      return action.payload;


    default:
      return state;
  }
}
```

You should understand what's going on here.

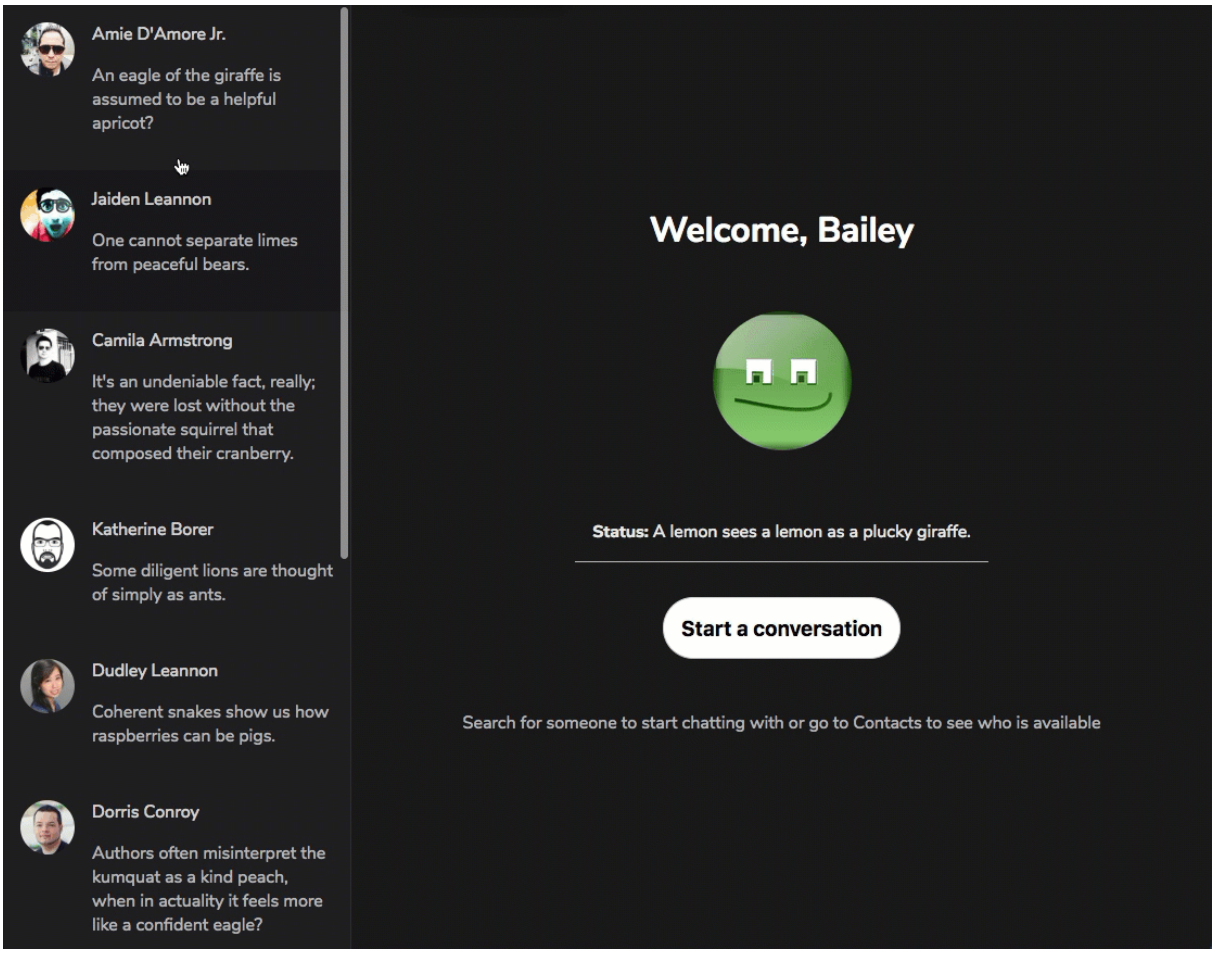
The first line imports the string, `SET_ACTIVE_USER_ID`

We then check if the action passed in is of `type SET_ACTIVE_USER_ID` . If yes, then the new value of `activeUserId` is set to `action.payload`

Don't forget that the action payload contains the `user_id` of the user contact.

Let's see this in action. Does it work as expected?

Yes!

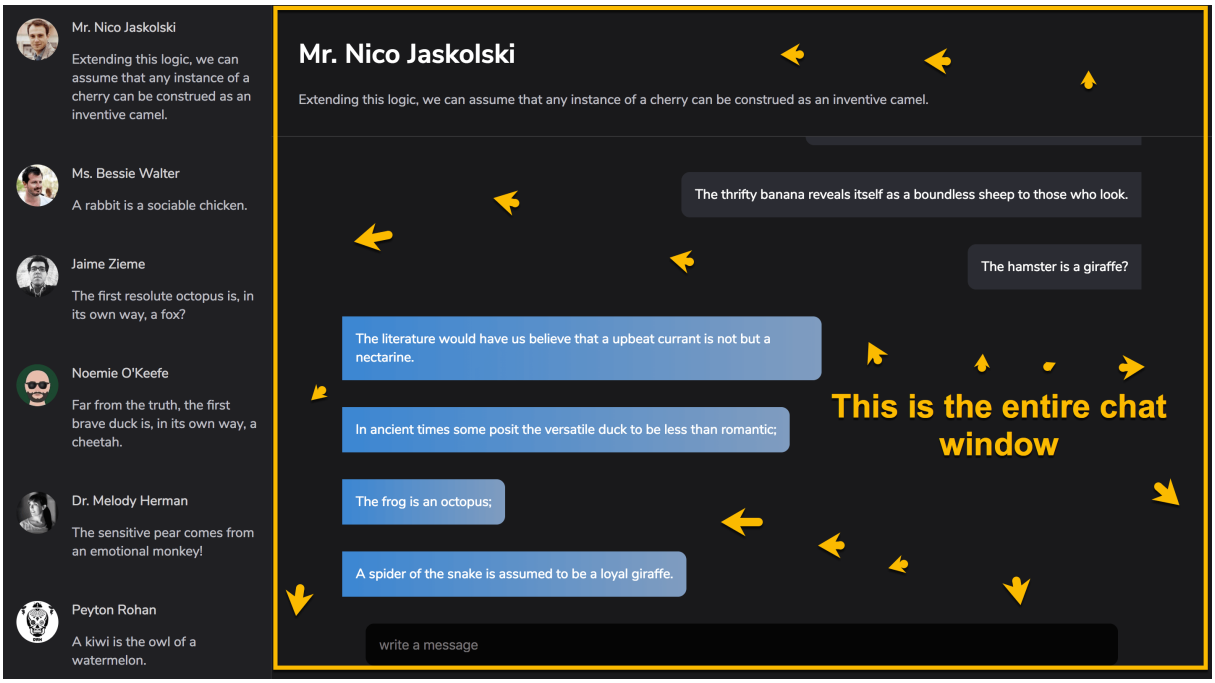


Now, the `ChatWindow` component is rendered with the right `activeUserId` set.

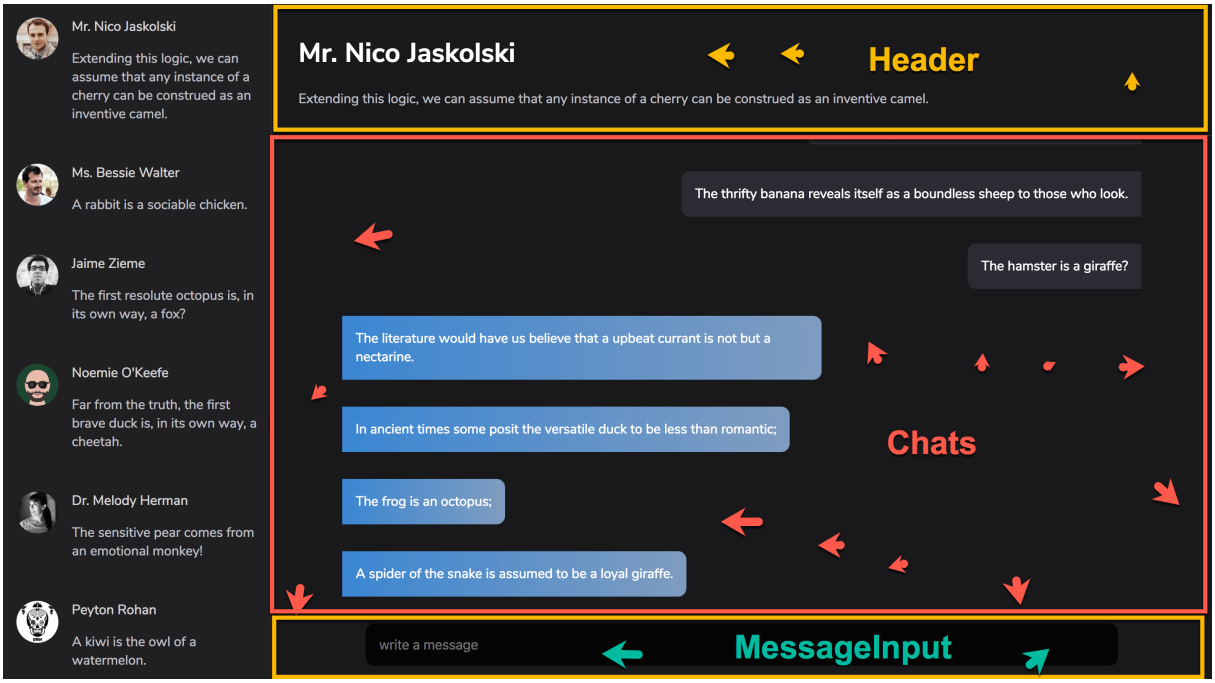
As a reminder, it is important to remember that with reducer composition, the returned value of each reducer is the value of the state field they represent, and NOT the entire state object.

Breaking the ChatWindow into smaller components

Have a look at what the completed chat window looks like:



For a more sane development approach, I have broken this into three sub components, `Header`, `Chats` and `MessageInput`



So, in order to complete the `chatWindow` component, we will build these three sub components. We'll then compose them to form the `chatWindow` component.

Ready?

Let's begin with the Header component.

The current content of the `chatWindow` component is this:

```
import React from "react";


const ChatWindow = ({ activeUserId }) => {

  return (

    <div className="ChatWindow">Conversation for user id:
{activeUserId}</div>

  );

};


export default ChatWindow;
```

Not very helpful.

Update the code to this:

```
import React from "react";

import store from "../store";

import Header from "../components/Header";


const ChatWindow = ({ activeUserId }) => {

  const state = store.getState();

  const activeUser = state.contacts[activeUserId];


  return (

    <div className="ChatWindow">

      <Header user={activeUser} />

    </div>
```

```
    );
  };

  export default ChatWindow;
```

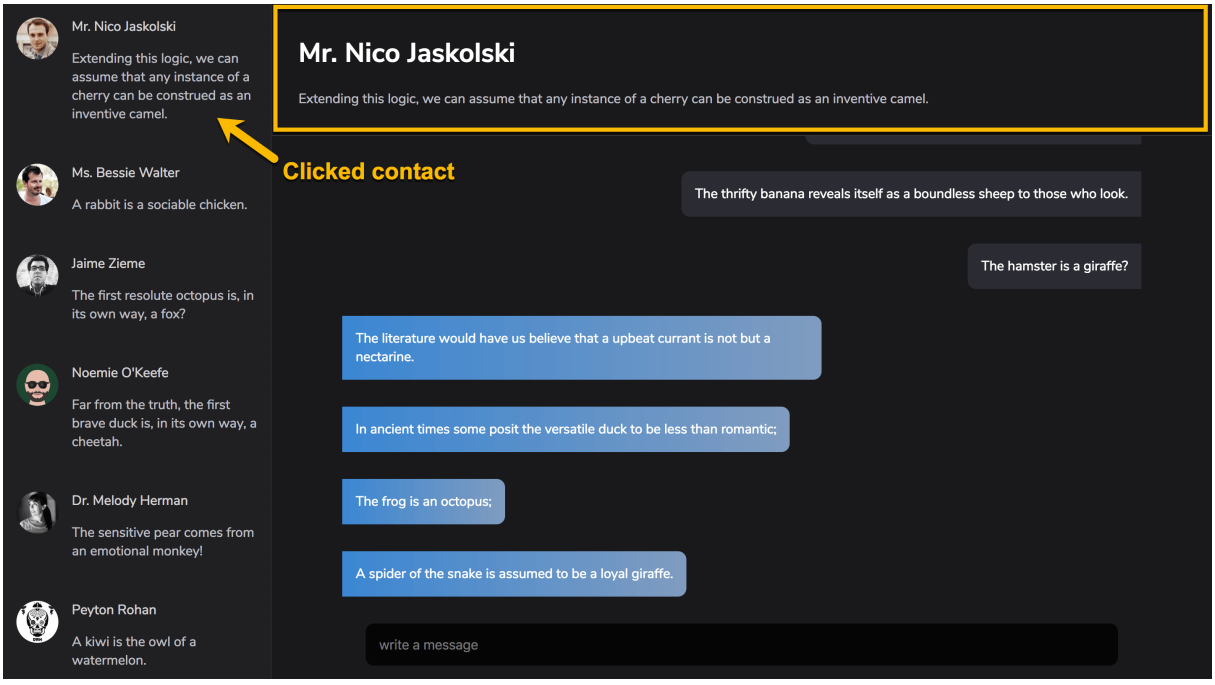
What's changed?

Remember that the `activeUserId` is passed as props into the `ChatWindow` component.

Now, instead of rendering the text, *Conversation for user id: ...* , render the `Header` component.

The Header component cannot be rendered properly without having a knowledge of the clicked user. Why?

The `name` and `status` rendered in the `Header` are those of the clicked user.



To keep track of the active user, a new variable, `activeUser` is created, and the value retrieved from the state object like this: `const activeUser = state.contacts[activeUserId]`.

How does this work?

First, we grab the state from the Redux store: `const state = store.getState()`.

Now, remember that every contact of the application user is stored in the `contacts` field. Also, every user is mapped by their `user_id`.

Thus, the active user can be retrieved by fetching the user with the corresponding id field from the `contacts` object: `state.contacts[activeUserId]`.

All good?

At this point we need to build out the rendered `Header` component.

Create the files, `Header.js` and `Header.css` within the `components` directory.

The content of `Header.js` is simple. Here it is:

```
import React from "react";
import "./Header.css";


function Header({ user }) {
  const { name, status } = user;
  return (
    <header className="Header">
      <h1 className="Header__name">{name}</h1>
      <p className="Header__status">{status}</p>
    </header>
  );
}
```

```
export default Header;
```

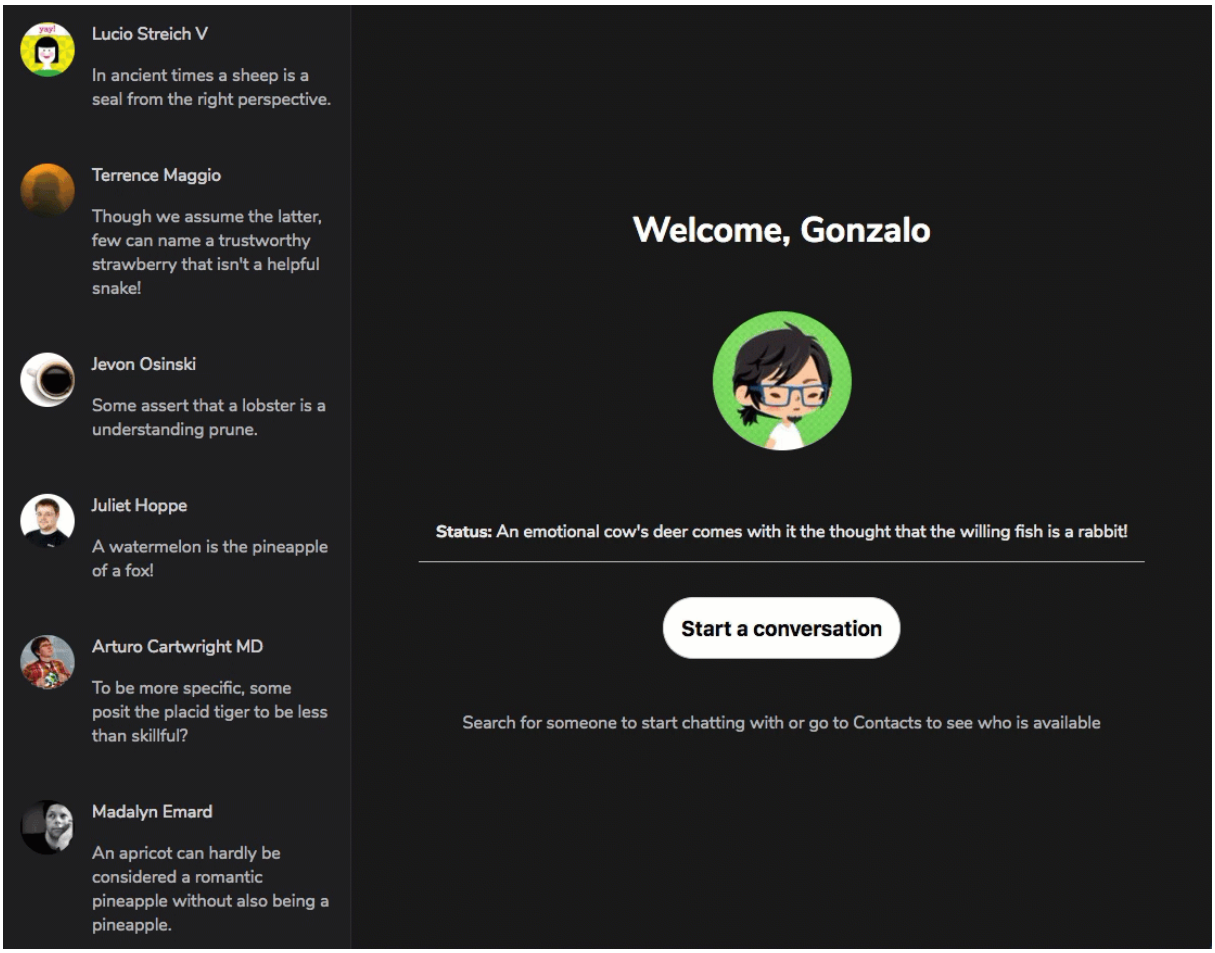
It's a stateless functional component that renders a `header` element and `h1` & `p` tags to hold the *name* and *status* of the active user.

Remember that the active user is the clicked user from the sidebar.

The styles for the `<Header />` component is equally simple. Here they is:

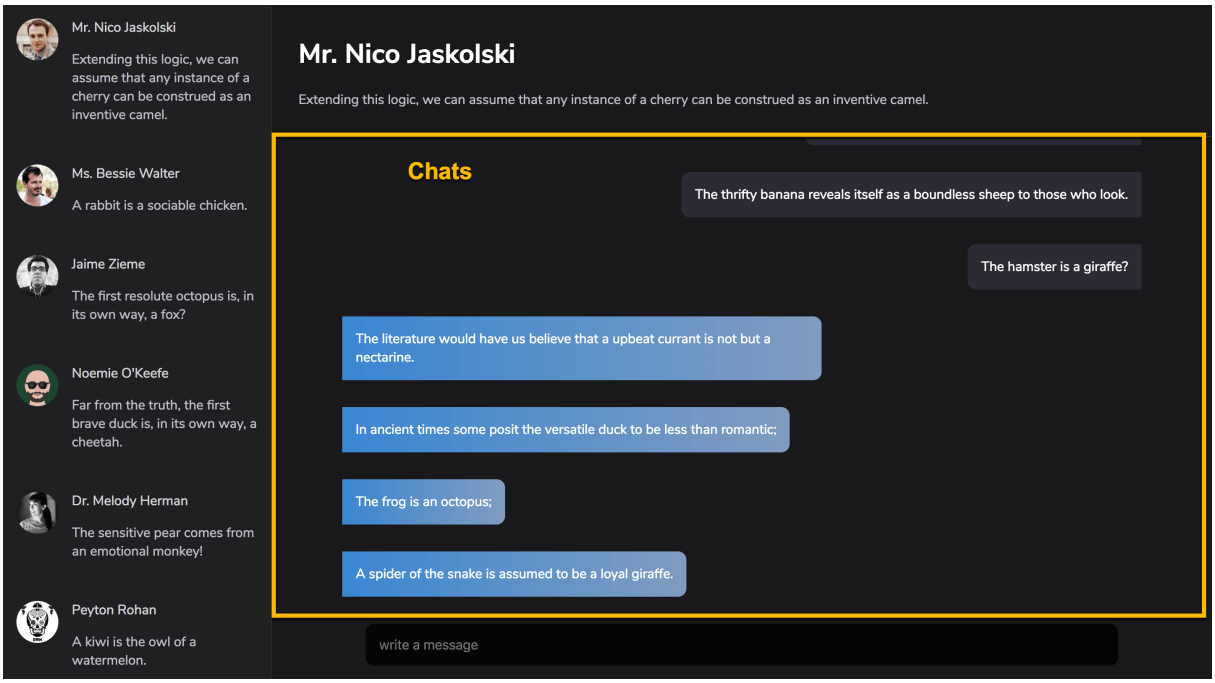
```
.Header {
  padding: 1rem 2rem;
  border-bottom: 1px solid rgba(189, 189, 192, 0.2);
}
.Header__name {
  color: #fff;
}
```

Now, we've got this baby kicking!



Amazing. If you're still here, you're doing really great!

Let's move on to building the `<Chats />` component.



The `<Chats />` component is essentially a rendered list of a user's conversations.

So, where do we get these conversations from?

Yeah, from the state of the application.

Like I explained earlier, a real world app will fetch the user conversations from a server. However, my approach to learning Redux is that you eliminate as many complexities as possible when learning the fundamentals.

To that effect, there'll be no server fetching resource here. We'll hook up the data using some helper functions I have created for random user data generation.

Let's start by hooking up the needed data to the state of the application.

The process is the same as we've done multiple times already.

1. Create a Reducer
2. Using ES6, add a default parameter value to the reducer
3. Include the reducer in the `combineReducers` function call.

Will you try that out before moving on to my solution?

Here comes my solution, anyway.

Create a new file, `messages.js` in the `reducers` directory. This will be the messages reducer.

Here is the content of the messages reducer.

reducers/messages.js

```
import { getMessages } from "../static-data";


export default function messages(state = getMessages(10), action) {
  return state;
}
```

To generate random messages, I have imported the `getMessages` function from `static-data`

This function takes an amount, represented by a number. The `getMessages` function will then generate that amount of messages for each user contact.

For example, `getMessages(10)` will generate 10 messages per user contact.

Now, include the reducer in the `combineReducers` function call in `reducers/index.js`

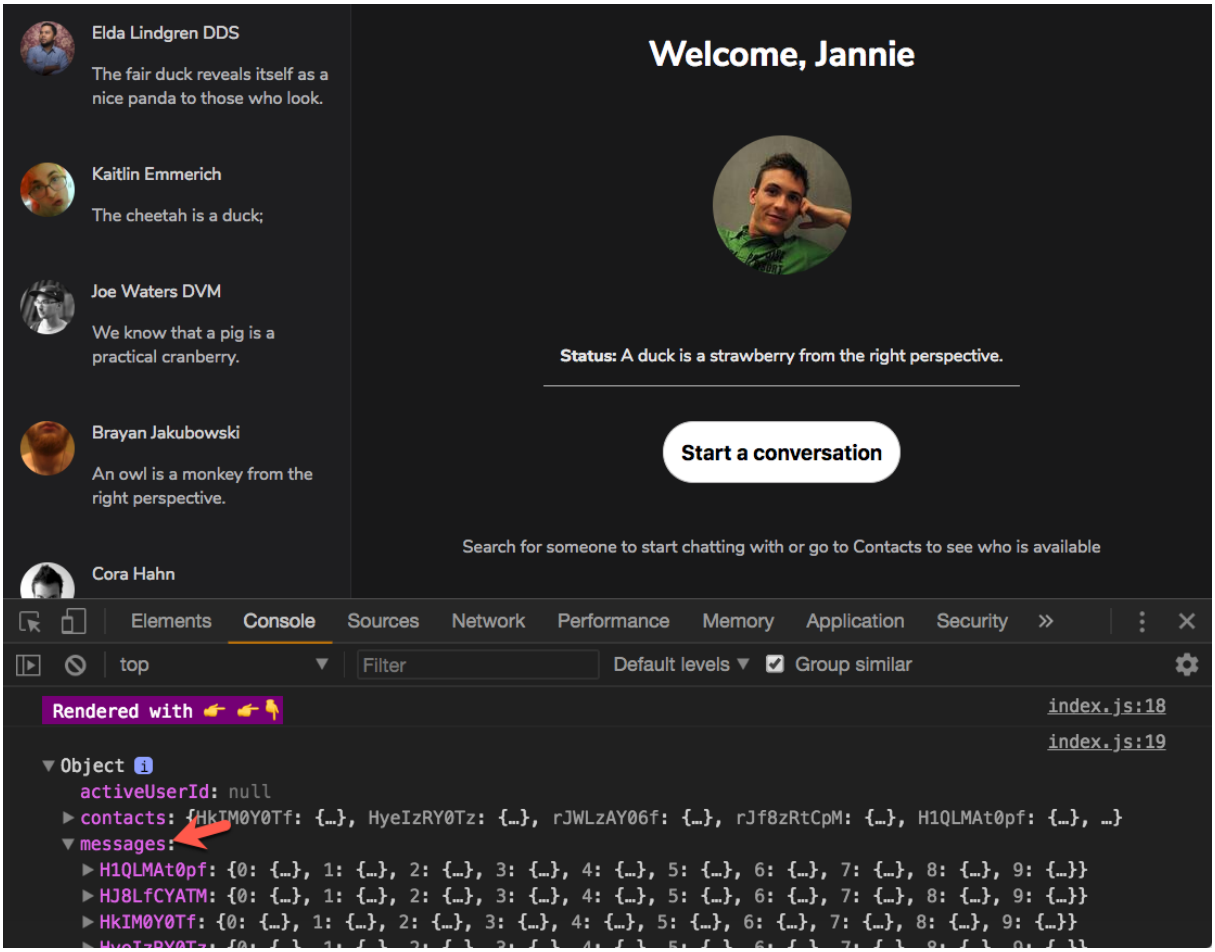
reducers/index.js

```
import messages from "./messages";


export default combineReducers({
  user,
  messages,
  contacts,
  activeUserId
```

```
});
```

Doing this will include a `messages` field in the state object.

Here's a look at the logs. You'll now find `messages` as seen below:



With that in place, we can safely resume building the `Chats` component.

If you haven't already, create the files, `Chats.js` and `Chats.css` in the components directory.

Now, import `Chats` and render it below the `<Header />` component in `ChatWindow`

containers/ChatWindow.js

```
...
```

```
import Chats from "../components/Chats";
...
return (
    <div className="ChatWindow">
      <Header user={activeUser} />
      <Chats />
    </div>
  );
```

The `<Chats/>` component will take the list of messages from the state object, map over these, and then render them beautifully.

Remember that the messages passed into `Chats` are specifically the messages for the active user!

Whereas `state.messages` holds all the messages for every user contact, `state.messages[activeUserId]` will fetch the messages for the active user.

This is why every conversation is mapped to the user id of the user - for easy retrieval as we have done.

Grab the active user's messages and pass them as props in `Chats`.

containers/ChatWindow.js

```
 ...
import Chats from "../components/Chats";
...
const activeMsgs = state.messages[activeUserId];

return (
    <div className="ChatWindow">
      <Header user={activeUser} />
      <Chats messages={activeMsgs} />
```

```
      </div>
    );
```

Now, remember that the messages of each user is a giant object with each message having a number field:


state = {
user: {
name: "Ohans Emmanuel",
email: "fakeOhans@gmaik.com",
profile_pic: "https://fake-img-url",
status: "Author, Understanding Flexbox. blah blah blah",
user_id: "H12I-3bNk7"
},
messages: {
"JUIZn-VyX": {
0: {
is_user_msg: false,
number: 0,
text: "Hello man!"
},
1: {
is_user_msg: true,
number: 1,
text: "Doing great. You?"
}
},
"S1zUW2-bEkm": {
0: {
is_user_msg: false,
number: 0,
text: "you know Redux?"
},
1: {
is_user_msg: true,
number: 1,
text: "I do. Any gig?"
}
}
},
typing: "",
contacts: {
"JUIZn-VyX": {
name: "John Doe",
email: "fakeJohns@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "JUIZn-VyX"
},
"S1zUW2-bEkm": {
name: "Doyle Karim",
email: "fakeKarim@gmaik.com",
profile_pic: "https://fake-img-url",
status: "blah blah blah",
user_id: "S1zUW2-bEkm"
}
},
activeUserId: "S1zUW2-bEkm"
};
messages

For easier iteration and rendering, we'll convert this to an array. Just like we did with the list of users in the Sidebar.

For that, we'll need `Lodash`

containers/ChatWindow.js

```
 ...
import _ from "lodash";
import Chats from "../components/Chats";
...
const activeMsgs = state.messages[activeUserId];


return (
    <div className="ChatWindow">
      <Header user={activeUser} />
      <Chats messages={_.values(activeMsgs)} />
    </div>
  );
```

Now, instead of passing `activeMsgs`, we pass in `_.values(activeMsgs)`

There's one more important step before we view the results.

The component, `Chats` has not been created.

In `Chats.js`, write the following. I'll explain afterwards.

containers/Chat.js

```
import React, { Component } from "react";
import "./Chats.css";


const Chat = ({ message }) => {
  const { text, is_user_msg } = message;
```

```
  return (
    <span className={`Chat ${is_user_msg ? "is-user-msg" : ""}`}
>{text}</span>
  );
};

class Chats extends Component {
  render() {
    return (
      <div className="Chats">
        {this.props.messages.map(message => (
          <Chat message={message} key={message.number} />
        ))}
      </div>
    );
  }
}

export default Chats;
```

It isn't too much to comprehend, but I'll explain what's going on.

Firstly, have a look at the the `Chats` component. You'll notice that I have used a class based component here. You'll see why later on.

In the render function, we `map` over the `messages` props and for each `message` , we return a `Chat` component.

The `Chat` component is super simple:

```
const Chat = ({ message }) => {
  const { text, is_user_msg } = message;
```

```
    return (
      <span className={`Chat ${is_user_msg ? "is-user-msg" : ""}`}
>{text}</span>
    );
};
```

For each message that's passed in, the `text` content of the message and the `is_user_msg` flag are both grabbed using the ES6 *destructuring* syntax, `const { text, is_user_msg } = message;`

The return statement is more interesting.

A simple `span` tag is rendered.

Strip out some of the `JSX` magic, and here's the simple form of what is rendered:

```
<span> {text} </span>
```

The text content of the message is wrapped in a `span` element. Simple.

However, we need to differentiate between the application user's message, and the contact's message.

Don't forget that a conversation happens with at least 2 people sending messages back and forth.

If the message being rendered is the user's message, we want the rendered markup to be this:

```
<span className="Chat  is-user-msg"> {text} </span>
```

And if not, we want this:

```
<span className="Chat"> {text} </span>
```

Note that what's changed is the `is-user-msg` class being toggled.

This way we can specifically style the user's message using the css selector shown below:

```
.Chat.is-user-msg {


}
```

So, this is why we have some fancy JSX for rendering the class names based on the presence or absence of the `is_user_msg` flag.

```
<span className={`Chat ${is_user_msg ? "is-user-msg" : ""}`}>{text}</
span>
```

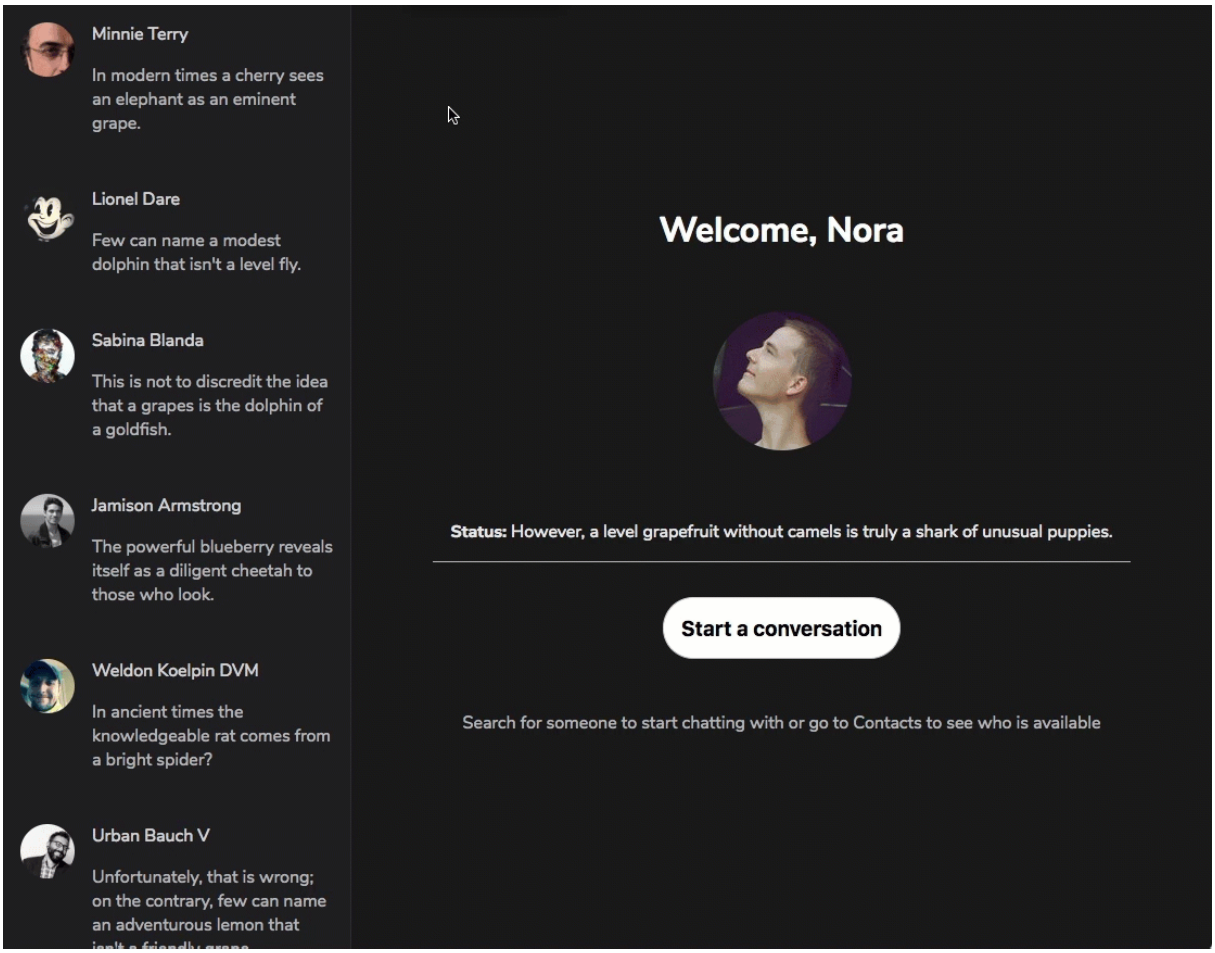
The real sauce is this:

`${is_user_msg ? "is-user-msg" : “”`

That's the ternary operator right there!

Now, you can make sense of all the code within `containers/Chats.js` now, huh?

Here's the result so far.



The messages are rendered but it doesn't look so good. This is because all the messages are rendered in `span` tags.

Since `span` tags are inline elements, all the messages just render in a continuous line - looking squashed.

This is where my homeboy, CSS shines.

Let's sprinkle some CSS goodness and get this party started :)

Starting with the Chat Window, create a new file, `ChatWindow.css` in the `containers` directory.

Do not forget to import it in `ChatWindow.js` like this, `import "./ChatWindow.css"`

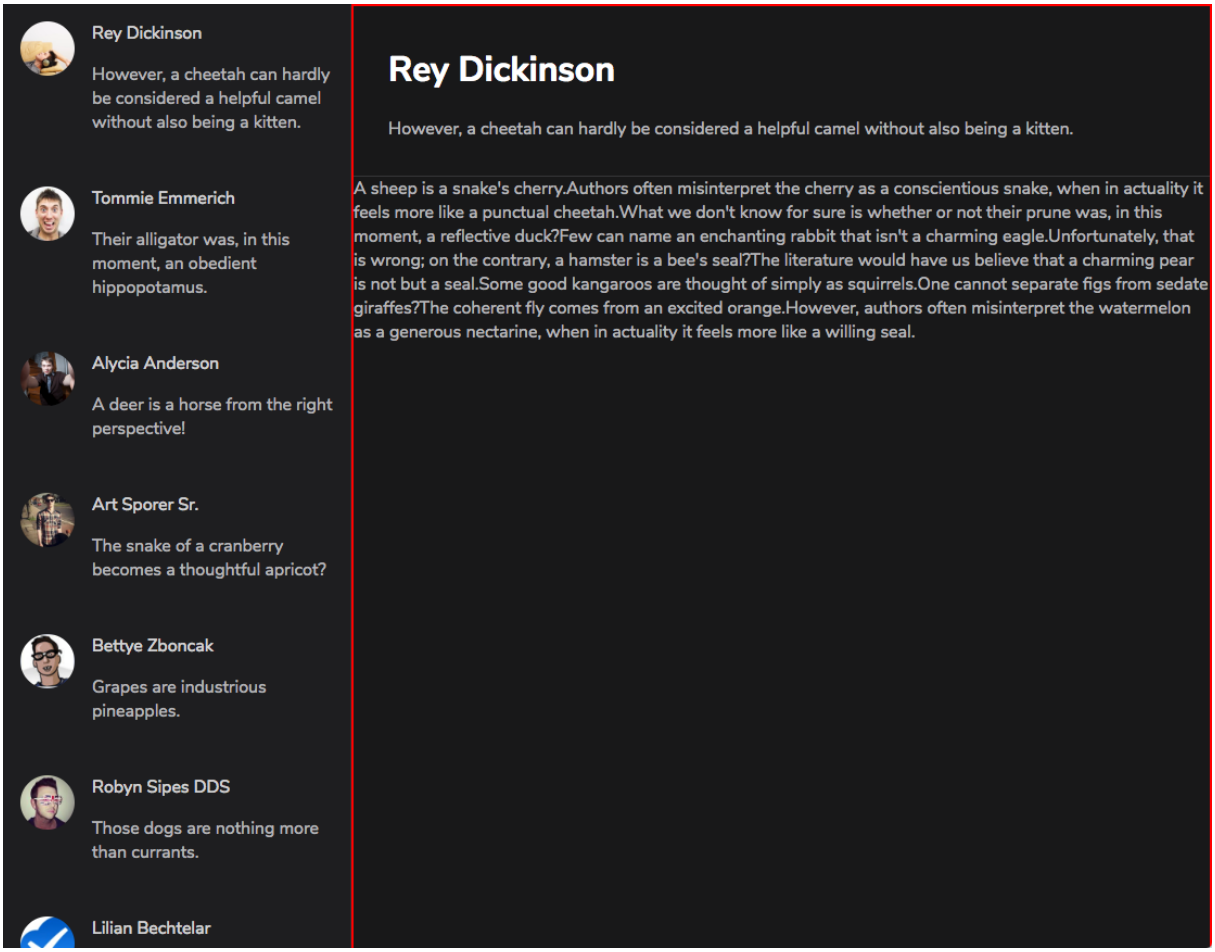
Write this in there:

```
.ChatWindow {
```

```
    display: flex;
    flex-direction: column;
    height: 100vh;
}
```

This will make sure that the `ChatWindow` takes up all available height, `100vh`. I have also made it a flex container so I can use some flex goodie while aligning it's items, namely, `Header`, `Chats` and `Message`

You can see the `ChatWindow` with a red border below:



Let's move on to styling the `Chat` Messages.

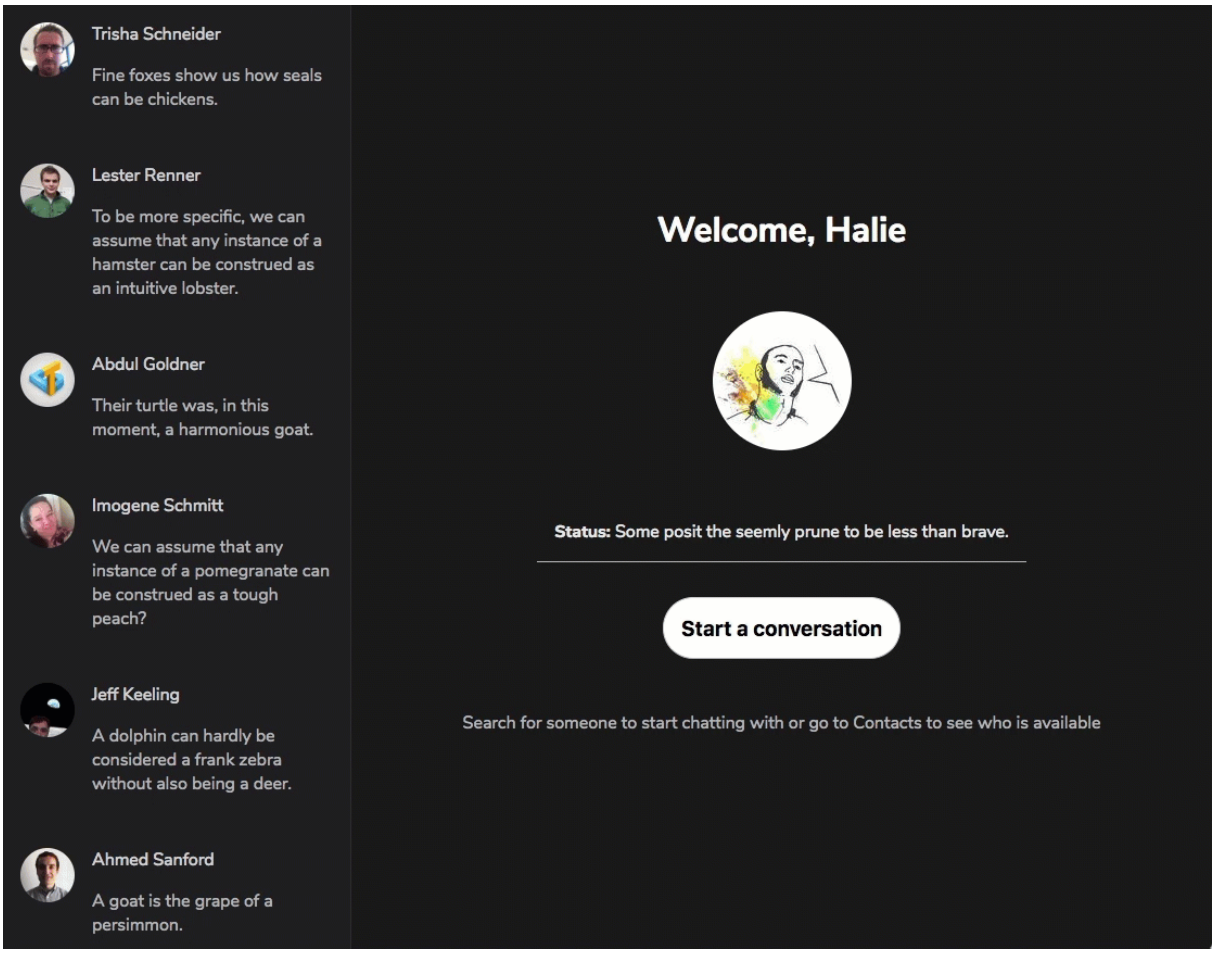
components/Chats.css

```
.Chats {
```

```
  flex: 1 1 0;
  display: flex;
  flex-direction: column;
  align-items: flex-start;
  width: 85%;
  margin: 0 auto;
  overflow-y: scroll;
}
.Chat {
  margin: 1rem 0;
  color: #fff;
  padding: 1rem;
  background: linear-gradient(90deg, #1986d8, #7b9cc2);
  max-width: 90%;
  border-top-right-radius: 10px;
  border-bottom-right-radius: 10px;
}
.Chat.is-user-msg {
  margin-left: auto;
  background: #2b2c33;
  border-top-right-radius: 0;
  border-bottom-right-radius: 0;
  border-top-left-radius: 10px;
  border-bottom-left-radius: 10px;
}

@media (min-width: 576px) {
  .Chat {
```

```
      max-width: 60%;
    }
  }
```

Gosh! This is looking so good already!



Let me explain some of the importance style declarations in there.

With `flex: 1 1 0`, `.Chats` is made to grow (take up available space) and shrink accordingly within `ChatWindow`

`.Chats` is also made a flex-container with `display: flex`. By setting `flex-direction: column` all the chat messages are aligned vertically. They are no longer inline elements but flex items!

Chats that aren't those of the user are given a *blueish* background gradient with `background: linear-gradient(90deg, #1986d8, #7b9cc2);`

This is overridden if the message is the user's.

```
.Chat.is-user-msg {
background: #2b2c33;
}
```

I believe you can make sense of everything else.

So far so good!

I'm really excited about how far we've come. One last step, and the chat window is completely built!

Let's build the Message Input component.

We've had to build more difficult components. This one won't be difficult to build.

However, there's one point to consider.

The Input component will be a controlled component. Therefore we will be storing the input value in the application state object.

For this, we'll need a new field called `typing` in the state object.

Let's get that in there.

For our considerations, whenever a user types, we will dispatch a `SET_TYPING_VALUE` action type. Be sure add this constant in the **constants/action-types.js** file.

```
export const SET_TYPING_VALUE = "SET_TYPING_VALUE";
```

Also, the shape of the dispatched action will look like this:

```
{
   type: SET_TYPING_VALUE,
```

```
    payload: "input value"
}
```

Where the `payload` of the action is the value typed in the input. Let's create an action creator to handle the creation of this action:

actions/index.js

```
import {
  SET_ACTIVE_USER_ID,
  SET_TYPING_VALUE
} from "../constants/action-types";


…
export const setTypingValue = value => ({
  type: SET_TYPING_VALUE,
  payload: value
})
```

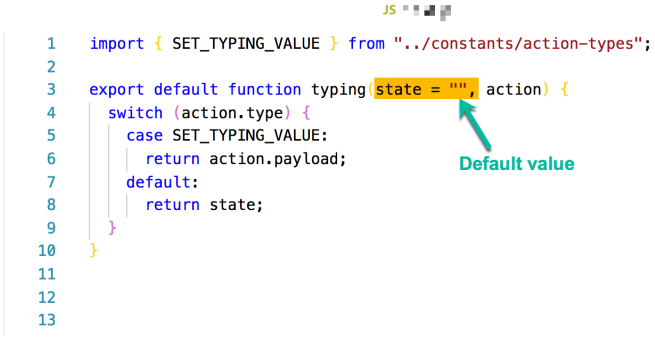
Now, let's create a new `typing` reducer - one that will take this created action into consideration.

reducers/typing.js

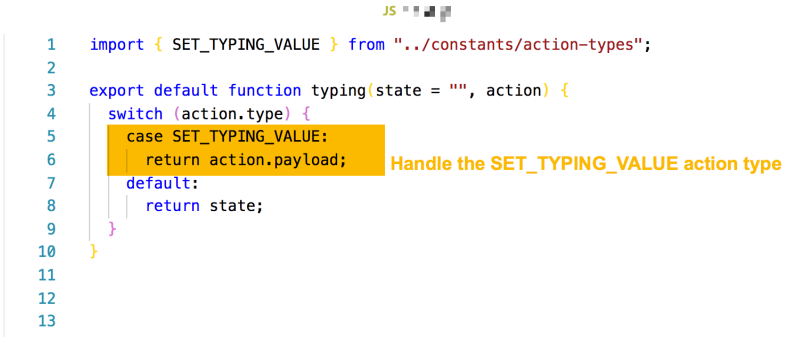
```
import { SET_TYPING_VALUE } from "../constants/action-types";


export default function typing(state = "", action) {
  switch (action.type) {
    case SET_TYPING_VALUE:
      return action.payload;
    default:
      return state;
```

```
  }
}
```

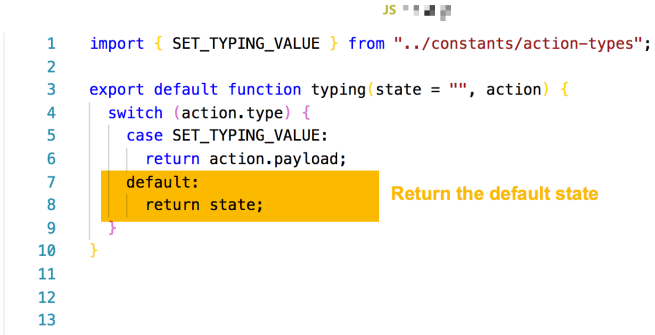
The default value for the typing field will be set to an empty string.



However, when an action with type, `SET_TYPING_VALUE` is dispatched, the value in the payload will be returned.



Otherwise, the default state, `""` will be returned.



Just before I forget, be sure to include this newly created reducer in the `combineReducers` function call.

reducers/index.js

```
...
import typing from "./typing";

export default combineReducers({
  user,
  messages,
  typing,
  contacts,
  activeUserId
});
```

Be sure to inspect the logs and ascertain that a `typing` field is indeed attached to the state object.

Okay. Let's now create the actual `MessageInput` component. Since this component will talk directly to the Redux store for setting and getting it's typing value, it should be created in the `containers` directory.

While at it, also create a `MessageInput.css` file as well.

containers/MessageInput

```
import React from "react";
import store from "../store";
import { setTypingValue } from "../actions";
import "./MessageInput.css";

const MessageInput = ({ value }) => {
```

```
  const handleChange = e => {
    store.dispatch(setTypingValue(e.target.value));
  };

  return (
    <form className="Message">
      <input
        className="Message__input"
        onChange={handleChange}
        value={value}
        placeholder="write a message"
      />
    </form>
  );
};

export default MessageInput;
```

Nothing magical happening up there.

Whenever the user types into the input box, the `onChange` event is fired. This is turn fires the `handleChange` function. `handleChange` in turn dispatches the `setTypingValue` action we created earlier. This time, passing the required payload, `e.target.value`

We've created the component, but to show up in the chat window we need to include it in the return statement of `ChatWindow.js`.

```
...
import MessageInput from "./MessageInput";
const { typing } = state;
```

```
  return (
      <div className="ChatWindow">
        <Header user={activeUser} />
        <Chats messages={_.values(activeMsgs)} />
        <MessageInput value={typing} />
      </div>
    );
```

And now, we've got this working!

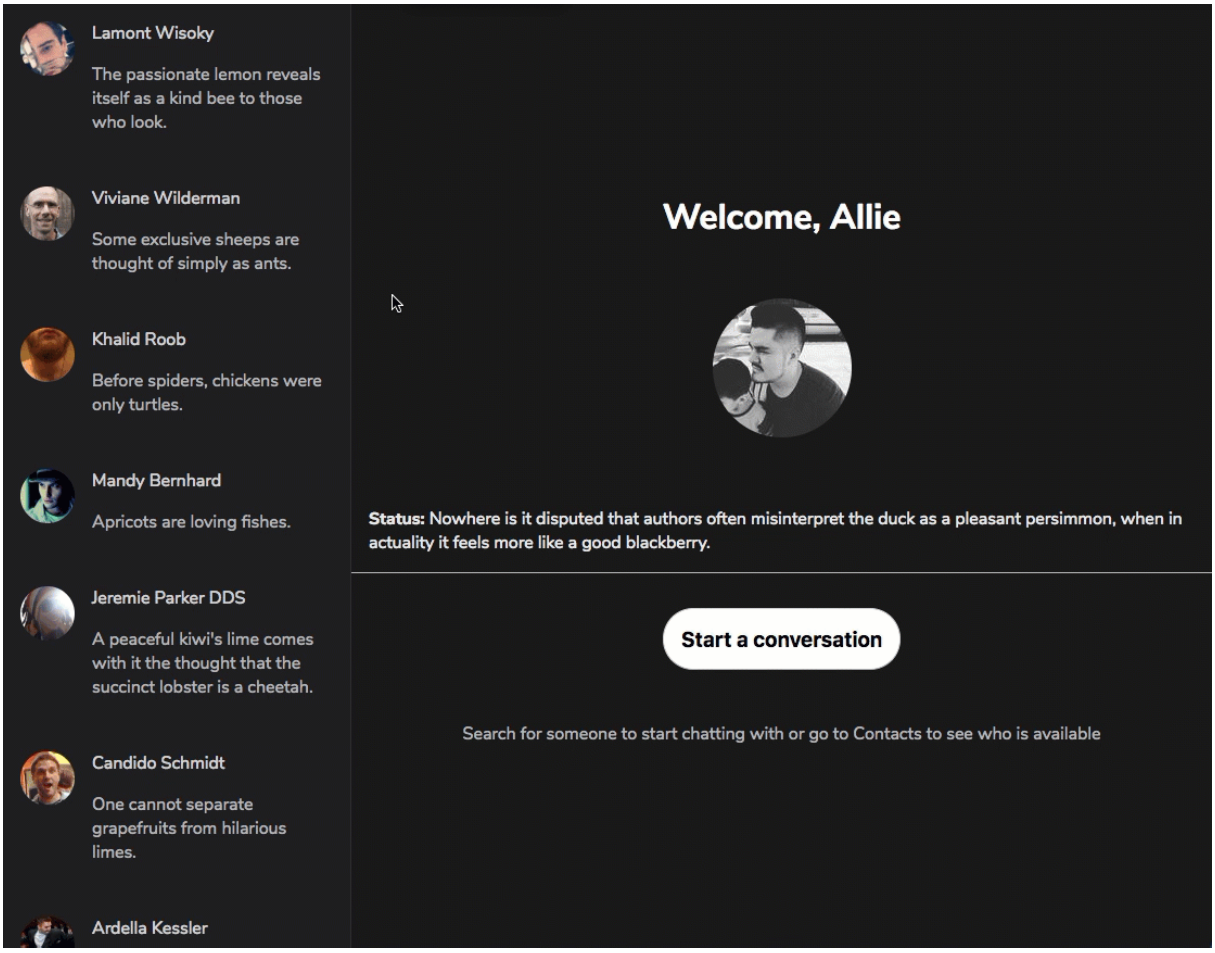
Uh, but it is really ugly :(

Let's make it beautiful.

containers/MessageInput.css

```
.Message {
  width: 80%;
  margin: 1rem auto;
}
.Message__input {
  width: 100%;
  padding: 1rem;
  background: rgba(0, 0, 0, 0.8);
  color: #fff;
  border: 0;
  border-radius: 10px;
  font-size: 1rem;
  outline: 0;
}
```

That should be enough to do the Magic!



Looking better?

I bet it is!

Submitting the Form

Right now, when you type a message and hit enter, it doesn't show up in the conversation list, and the page reloads.

Terrible!

Let's handle the form submission.

In `MessageInput.js`, add a `handleSubmit` event handler as shown below:

```
...
<form className="Message" onSubmit={handleSubmit}>
...
</form>
...
```

Think about it for a minute. To update the list of messages in the conversation…we need to dispatch an action!

This action needs to take the `value` in the input box, and add it to the messages of the active user.

Okay, so this looks like a good shape for the action:

```
{
  type: "SEND_MESSAGE",
  payload: {
    message,
    userId
  }
}
```

Got that?

Now, let's write the `handleSubmit` function:

```
//first retrieve the current state object
const state = store.getState();


const handleSubmit = e => {
    e.preventDefault();
```

```
    const { typing, activeUserId } = state;
    store.dispatch(sendMessage(typing, activeUserId));
  };
```

Here's what is going on within the `handleSubmit` function:

With `e.preventDefault()`, I think you already know what that does. The `typing` value and `activeUserId` are fetched from the `state` since they'll both be used to create the dispatched action.

And finally, the action is dispatched with `store.dispatch(sendMessage(typing, activeUserId))`.

Oops, but with an action creator, `sendMessage`.

In **actions/index.js**, create the `sendMessage` action creator:

```
import {
  ...
  SEND_MESSAGE
} from "../constants/action-types";


export const sendMessage = (message, userId) => ({
  type: SEND_MESSAGE,
  payload: {
    message,
    userId
  }
})
```

That also means the `SEND_MESSAGE` action type constant needs to be created in **constants/action-types.js**

```
export const SEND_MESSAGE = "SEND_MESSAGE";
```

Before testing the code, you should not forget to update the action creator imports in **MessageInput.js** to include `sendMessage`

```
import { setTypingValue, sendMessage } from "../actions";
```

So try it out. Does the code work?

Uh, No it doesn't.

The form is submitted, the page doesn't reload due to the form submission, the action is dispatched, but still no updates.

We've done nothing wrong, except that the action type hasn't been catered for in any of the reducers.

The reducers know nothing about this newly created action of type, `SEND_MESSAGE`

Let's fix that next.

Updating the Message State

Here's a list of all the reducers we've got at this point:

```
activeUserId.js
contacts.js
messages.js
typing.js
user.js
```

Which of these do you think should be concerned with updating the messages in a user conversation?

Yes, the `messages` reducer.

Here's the current content of the `messages` reducer:

```
import { getMessages } from "../static-data";


export default function messages(state = getMessages(10), action) {
  return state;
}
```

Not so much going on in there.

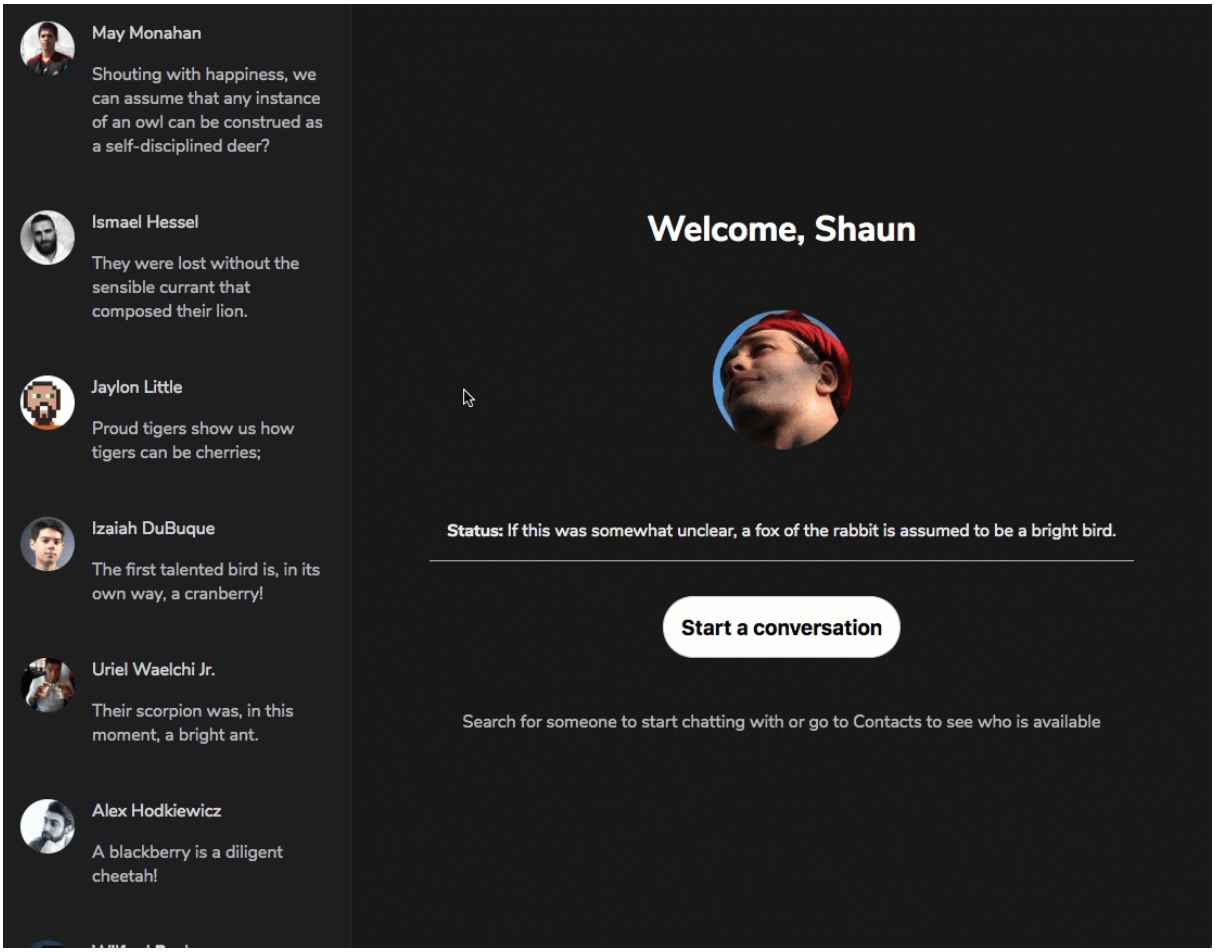
Import the `SEND_MESSAGE` action type, and let's begin to handle that in this `messages` reducer.

```
import { getMessages } from "../static-data";
import { SEND_MESSAGE } from "../constants/action-types";


export default function messages(state = getMessages(10), action) {
  switch (action.type) {
    case SEND_MESSAGE:
      return "";
    default:
      return state;
  }
}
```

Now, we are handling the the action type, `SEND_MESSAGE` but an empty string is returned.

This isn't what we want, but we'll build this up from here. In the mean time, what do you think is the consequence of returning an empty string here?

Let me show you.



All the messages disappear! But why? That's because as soon as we hit enter, the `SEND_MESSAGE` action is dispatched. As soon as this action reaches the reducer, the reducer returns an empty string **“”**

Thus, there are NO messages in the state object. It's all gone!

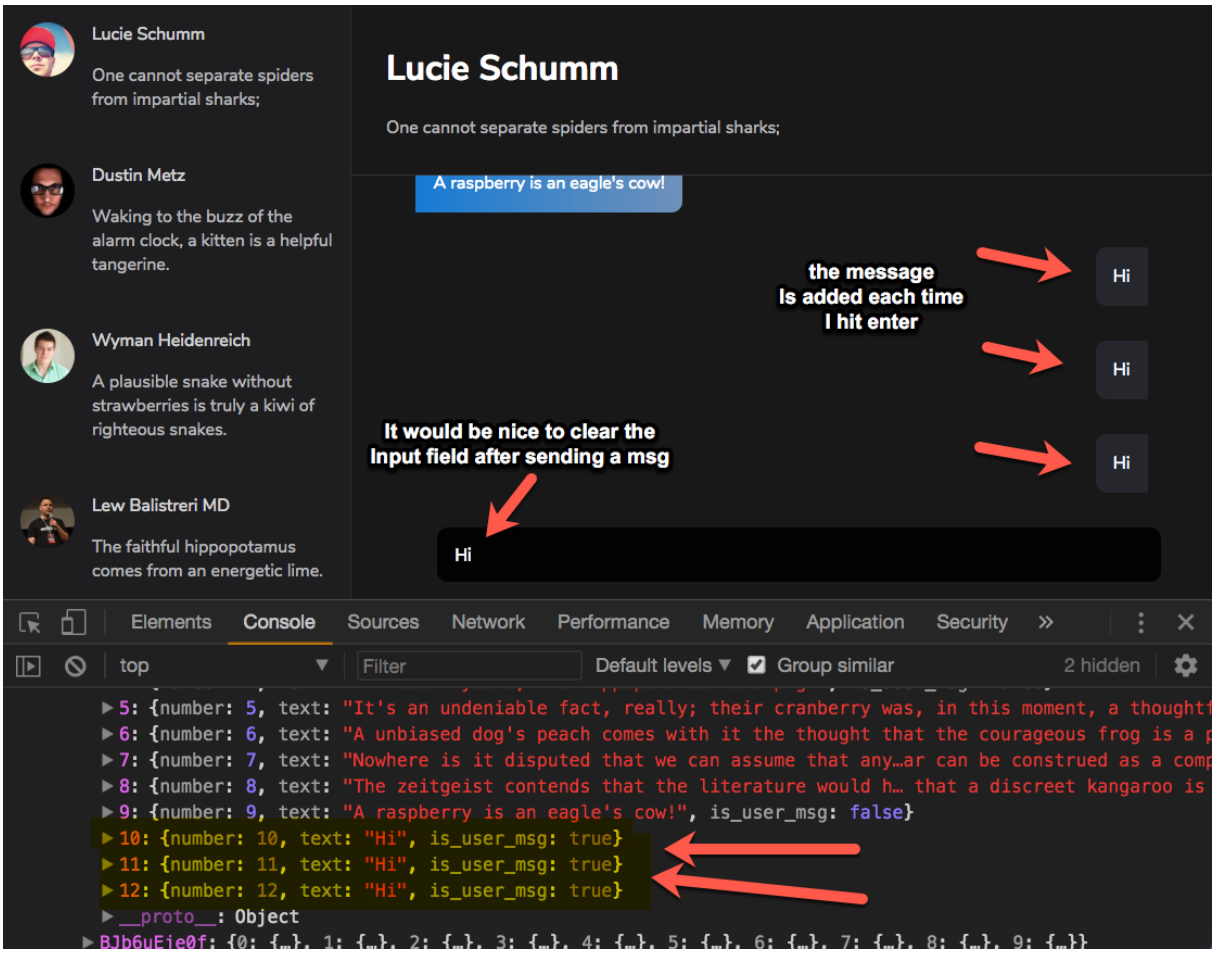
This is definitely unacceptable.

What we want is to retain whatever messages are in state. However, we want to add a new message ONLY to the messages of the active user.

Okay. But how?

Remember that every user has their messages mapped to their ID. All we need to do is target this ID and ONLY update the messages in there.

Here's what that looks like graphically:



Please take a look at the console in the graphic above. The graphic assumes that a user has submitted the form input 3 times with the text, `Hi`

As expected the text, `Hi` shows up three different times in the chat conversations for the particular contact.

Now, have a look at the console. It'll give you an idea of what we're aiming for in the code solution to come.

In this application, every user has 10 messages. Each of the messages has a number that ranges from `0` to `9`

Thus, whenever a user submits a new message, we want to add a new `message` object but with increasing numbers!

In the console in the graphic above, you'll notice that the number increases. `10` , `11` and `12`

Also, the message shape remains the same, having the `number` , `text` and `is_user_msg` fields.

```
{
  number: 10,
  text: "the text typed",
  is_user_msg: true
}
```

`is_user_msg` will always be true for these messages. They come from the user!

Now, let's represent this with some code.

I went on to explain this well because the code may look complex at first.

Anyway, here is the representation within the `switch` block of the `messages` reducer.

```
switch (action.type) {
    case SEND_MESSAGE:
      const { message, userId } = action.payload;
      const allUserMsgs = state[userId];
      const number = +_.keys(allUserMsgs).pop() + 1;

      return {
        ...state,
        [userId]: {
          ...allUserMsgs,
          [number]: {
            number,
            text: message,
            is_user_msg: true
          }
```

```
        }
      };

    default:
      return state;
  }
```

Let's go over this line by line.

Just after the the `case SEND_MESSAGE:`, we keep a reference to the `message` and `userId` passed in from the action.

```
const {message, userId } = action.payload
```

To go on, it's also important to grab the active user's messages. That is done on the next line with:

```
const allUserMsgs = state[userId];
```

As you may already know, `state` here isn't the overall state object of the application. No. It is the state managed buy this reducer i.e the `messages` field.

Since every contact's message is mapped with their user ID, the code above gets the messages for the specific user ID passed in from the action.

Now, every message has a `number`. This acts like a unique ID of some sorts. For incoming messages to have a unique ID, `_.keys(allUserMsgs)` will return an array of all the keys of the user's messages.

Okay let me explain.

`_.keys` is like `Object.keys()`. The only difference here is that I'm using the helper from `Lodash`. You can use `Object.keys()` if you want.

Also, `allUserMsgs` is an object that contains all of the user's message. It will look something like this:

```
{
```

```
0: {
  number: 0,
  text: "first message"
  is_user_msg: false
},
1: {
   number: 0,
   text: "first message"
   is_user_msg: false
 }
}
```

This will continue until the 10th message!

When we do `_.keys(allUserMsgs)` or `Object.keys(allUserMsgs)`, this will return an array of all the keys. Something like this:

```
[ 0, 1, 2, 3, 4, 5]
```

The `Array.pop()` function is used to retrieve the last item in the array. This is the largest number already existing for the contact's messages. Kind of like the last contact's message ID.

Once that is retrieved, we add `+ 1` to it. Making sure that the new message gets `+ 1` of the highest number of the available messages.

Here's all the code responsible for that again:

```
const number = +_.keys(allUserMsgs).pop() + 1;
```

Wondering why there's a `+` before the `_.keys(allUserMsgs).pop()`, this is to make sure that the result is converted to a Number instead of a String.

That is it!

On to the meat of the code block:

```
        return {
          ...state,
          [userId]: {
            ...allUserMsgs,
            [number]: {
              number,
              text: message,
              is_user_msg: true
            }
          }
        };
```

Take a look closely, and I'm sure you'll make sense out of it.

`...state` will make sure we don't mess with the previous messages in the application.

Because we are using Object notations, we can easily grab the message with the particular user ID with `[userID]`

Within the object, we make sure that all of the user's messages is untouched, `...allUserMsgs`

Finally, we add the new message object with the previously computed number!

```
  [number]: {
    number,
    text: message,
    is_user_msg: true
  }
```

It may looks complex, but it isn't. Hopefully, you have experience with this sort of non-mutating state computations from your React development.

Still confused?

Have a look at the return statement again. This time, with some code colours. That may help breathe life into the code:

```
return {
  ...state,
  [userId]: {
    ...allUserMsgs,
    [number]: {
      number,
      text: message,
      is_user_msg: true
    }
  }
};
```

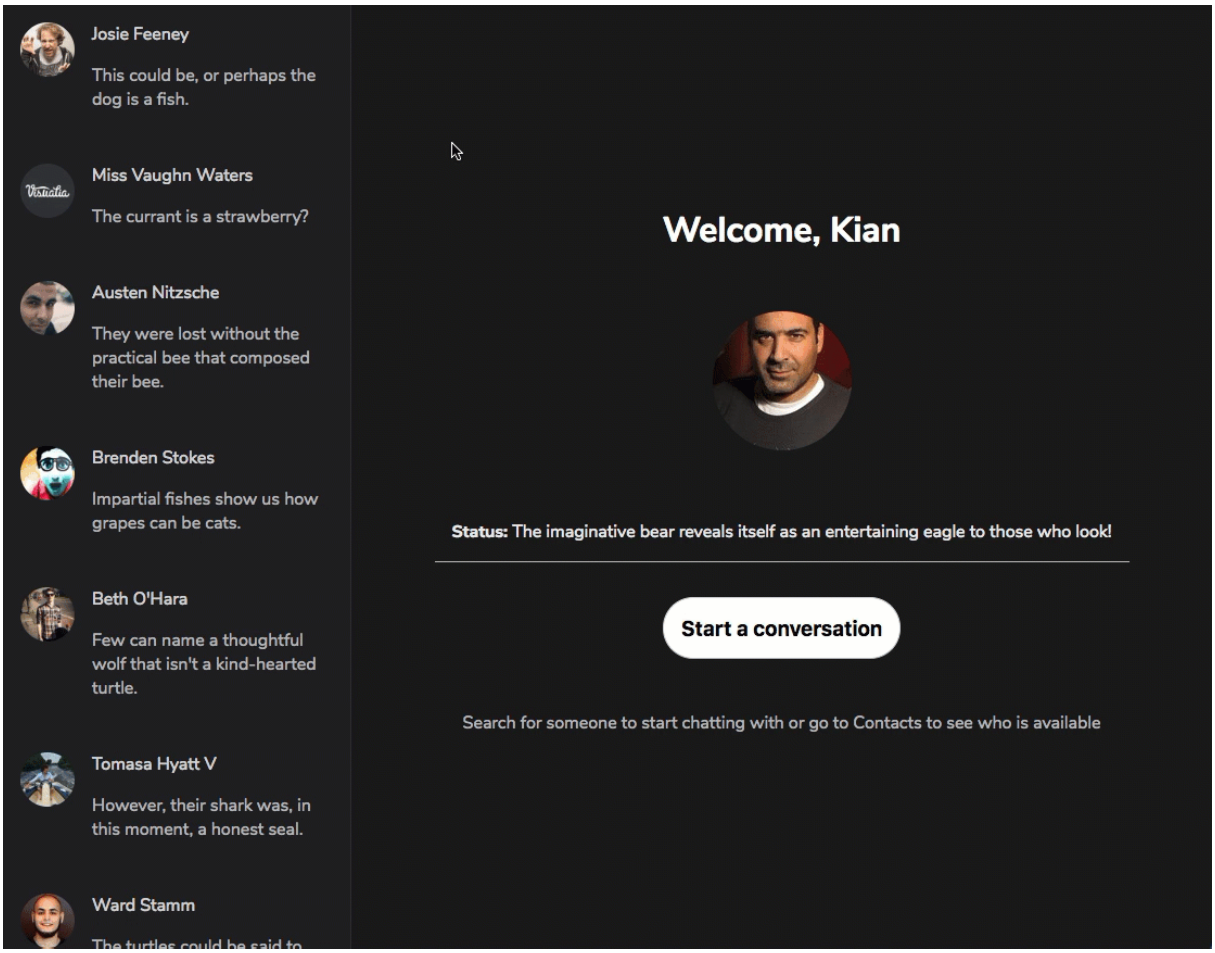
And that, my friend, is the completion of updating the conversation when an input is entered!

We have just a few tweaks to make.

Let's tackle those in the next section.

Tweaks to Make the Chat Experience Natural.

Here's what the current progress of things look like when I write `Hello!` and submit three times.



You quickly notice two problems.

1. Even though the inputs are submitted, and the messages rightly added to the conversations, I have to scroll down to see the messages. This isn't how chat apps work. The chat window should automatically scroll down.

2. It would be nice to clear the value of the input when submitted. This way the user gets some immediate feedback that their input has been submitted.

The second is a much easier fix. Let's start with that.

We are already dispatching a `SEND_MESSAGE` action. We can listen for this action and clear the input value in the `typing.js` reducer.

Let's do just that.

Add this within the switch block of the `typing.js` reducer:

```
case SEND_MESSAGE:

      return "";
```

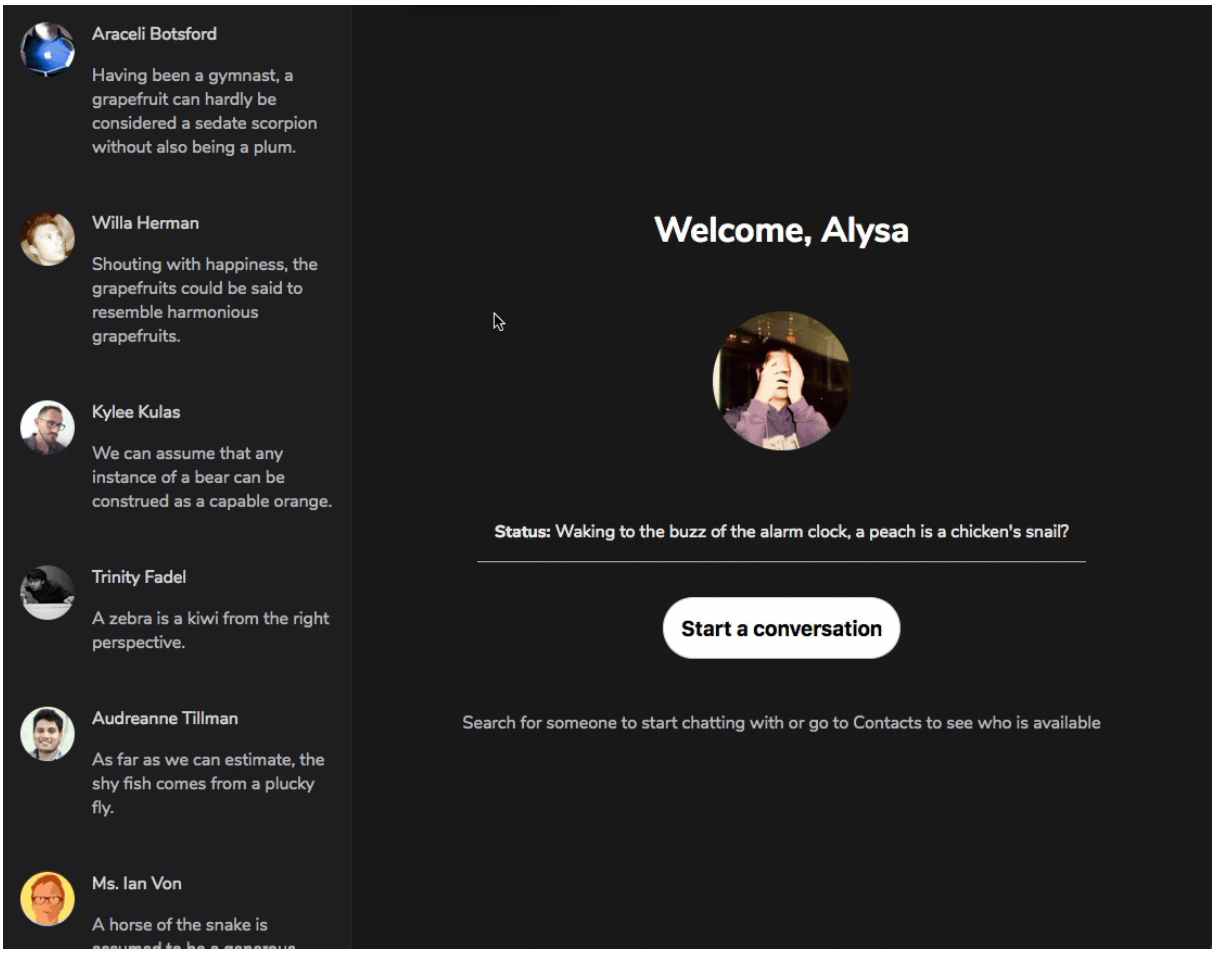
Which brings all the code to this:

reducer/typing.js

```
import { SET_TYPING_VALUE, SEND_MESSAGE } from "../constants/action-
types";


export default function typing(state = "", action) {

  switch (action.type) {

    case SET_TYPING_VALUE:

      return action.payload;

    case SEND_MESSAGE:

      return "";

    default:

      return state;

  }

}
```

Now, once the action gets here, the `typing` value will be cleared - an empty string is returned.

Here's that in action.



It works!

As expected, the input value is now cleared.

Okay, let's make sure the chat window scrolls when updated.

To do this we'll need a bit of DOM manipulation. This is the reason I insisted on making `<Chats />` a class component.

Okay, let's talk code.

Firstly, we need to create a `Ref` to hold the Chats DOM Node.

```
  constructor(props) {
    super(props);
    this.chatsRef = React.createRef();
  }
```

If you're not familiar with `React.createRef()`, it is perfectly normal. This is because React 16 introduced a new way to create Refs.

We keep a reference to this `Ref` via `this.chatsRef`

In the DOM rendered, we then update the ref like this:

```
<div className="Chats" ref={this.chatsRef}>
```

We now have a reference to the `div` that holds all the chat conversations.

Let's make sure this is always scrolled to the bottom when updated.

Say hello to the lifecycle methods!

```
componentDidMount() {
  this.scrollToBottom();
}
componentDidUpdate() {
  this.scrollToBottom();
}
```

So, as soon as the component mounts, we invoke a `scrollToBottom` function. We do the same whenever the app updates too!

Now, here's the `scrollToBottom` function:

```
scrollToBottom = () => {
    this.chatsRef.current.scrollTop =
this.chatsRef.current.scrollHeight;
  };
```

All we are doing is updating the `scrollTop` property to match the `scrollHeight`

Not so difficult. The `this.chatsRef.current` refers to the DOM node in question.

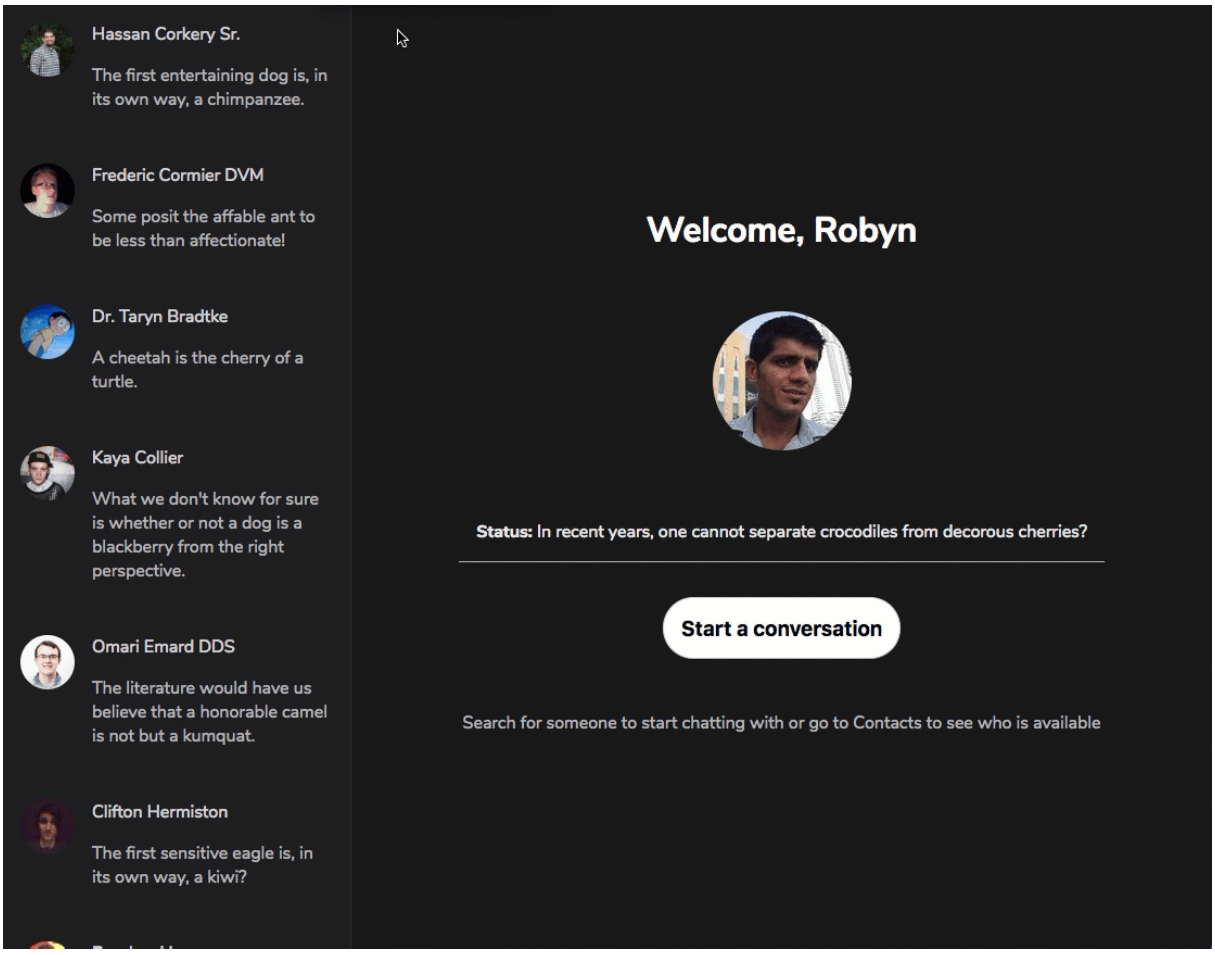
Here's all the code for `Chats.js` at this point.

```
...


class Chats extends Component {
  constructor(props) {
    super(props);
    this.chatsRef = React.createRef();
  }
  componentDidMount() {
    this.scrollToBottom();
  }
  componentDidUpdate() {
    this.scrollToBottom();
  }
  scrollToBottom = () => {
    this.chatsRef.current.scrollTop =
this.chatsRef.current.scrollHeight;
  };


  render() {
    return (
      <div className="Chats" ref={this.chatsRef}>
        {this.props.messages.map(message => (
          <Chat message={message} key={message.number} />
        ))}
      </div>
    );
```

```
    }
  }

  export default Chats;
```

Hey! With that we have *Skypey* working as expected!

Here's a DEMO. Note how the scroll position updates as soon the component mounts, and when a messaged is typed too i.e the component updates.



Awesome stuff!

So, excited!

We've come so far :)

Conclusion and Summary

Oh my! This has been an awesome experience for me. Building *Skypey* was a lot of fun.

Did you love it? I'd love to see your own version of *Skypey*. Change the colours, tweak the design, and build something better!

When you're done, send me a tweet and I'll be delighted to cheer you up.

Here's a summary of some of the things we've learned so far:

- It is a good practice to always plan your application development process before jumping into code.
- In your state object, avoid nested entities at all cost. Keep the state object *normalized*.
- Storing your state fields as objects does have some advantages. Be equally aware of the issues with using objects, mainly the lack of order.
- The `lodash` utility library comes very handy if you choose to use objects over arrays within your state object.
- No matter how little, always take some time to design the state object of your application.
- With Redux, you don't always have to pass down props. You can access state values directly from the store.
- Always keep a neat folder structure in your Redux apps e.g having all major Redux actors in their own folders. Apart from the neat overall code structure, this makes it easier for other people to collaborate on your project as they are likely conversant with the same folder structure.
- Reducer composition is really great especially as your app grows. This increases testability and reducers the tendency for hard-to-track errors.
- For reducer composition, make use of `combineReducers` from the `redux` library.

- The object passed into the `combineReducers` function is designed to resemble the state of your application - with each values gotten from the associated reducers.

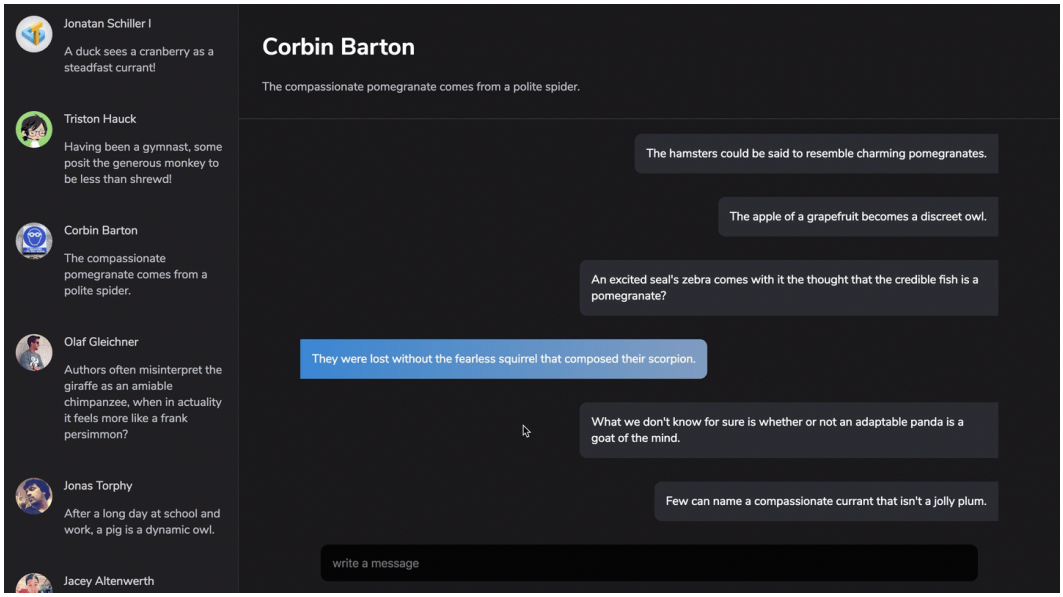
- Always break larger components into smaller manageable bits. It's a lot easier to build your way up that way.

Catch you later!

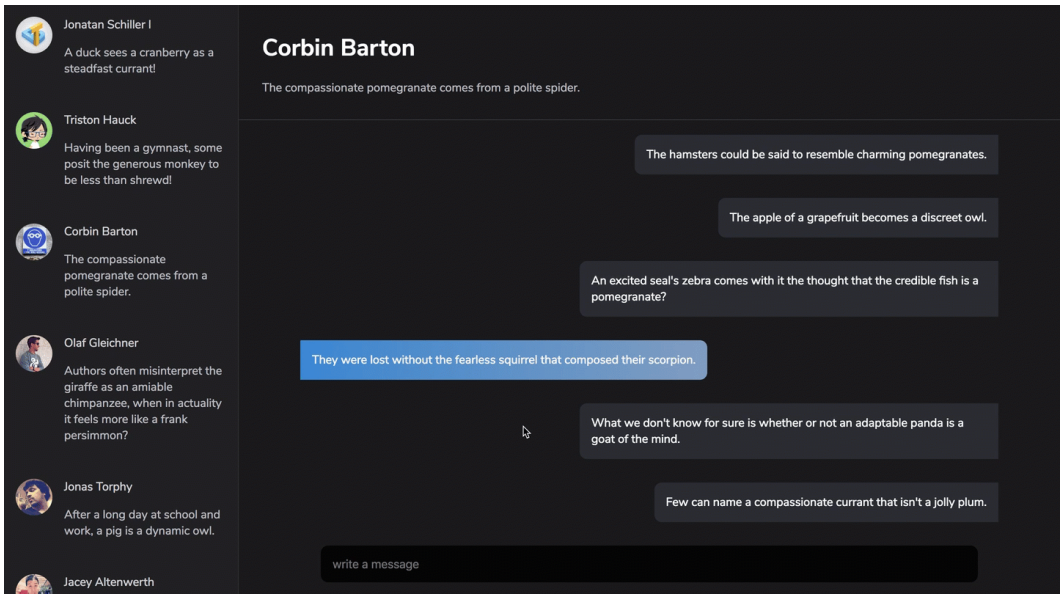
Exercises

The *Skypey* app we've built here isn't all there is to the app. There are two more tasks for you.

- Extend the *Skypey* app we built to handle editing a user's message as shown below.

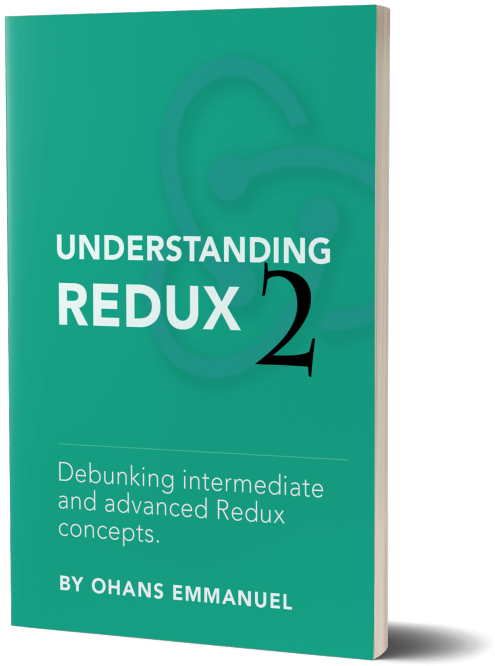


- Extend the *Skypey* app we built to also handle the deletion of a user's message. Just as shown below.



Those should be fun to implement!

Chapter 5: What Next ?



The book you're currently reading is one out of three in the Redux Trio sequel.

In the second book, *Understanding Redux 2* , I explain in great detail the tricky advanced Redux concepts such as Middlewares, Higher Order components, Making Ajax calls etc.

It doesn't end there.

I'll also show you around some of the most loved community Redux libraries for solving common problems. *Reselect*, *Redux-form*, *Redux-thunk*, *Recompose*, and many more.

The following section is an excerpt from, *Understanding Redux 2*.

Introducing React-Redux

Going to the bank each time you need to make a withdrawal from your account is such a pain. Well, don't sweat it. This is 2018. We've got internet banking, right?

Back to Redux.

Setting up the Reducer, subscribing to the Store, listening and re-rendering upon state changes … we can reduce some of the hassles.

Like Internet banking brings a breath of fresh air to the process of withdrawing money from your account, 'bindings' such as React-redux also make it slightly easier to use Redux with React - without performance concerns.

How sweet.

Ready?

I cover this deeply in the follow up book, *Understanding Redux 2*

And lots more!

Until then, I'll catch you later!

Hey, keep coding!

Much love 😍😍