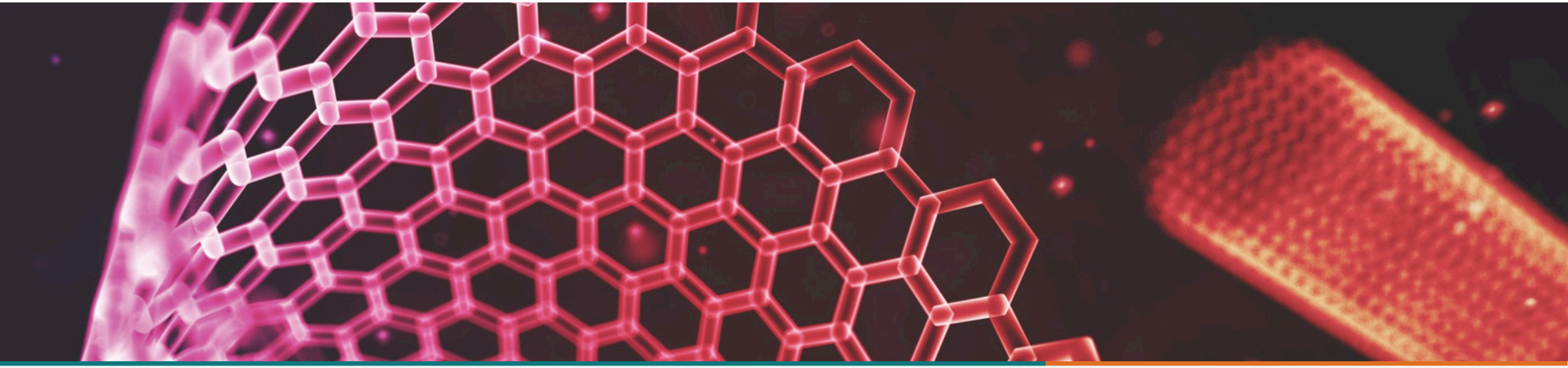
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

Redux



STEVENS
INSTITUTE of TECHNOLOGY

Schaefer School of
Engineering & Science

stevens.edu

Patrick Hill
Adjunct Professor
Computer Science Department
Patrick.Hill@stevens.edu

What is Redux?

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

You can use Redux together with React, or with any other view library.

Redux is a way to manage an application state and move it to an **external global store**.

A React Component State Example

```
1.  import React, { Component } from "react";
2.
3.  class ExampleComponent extends Component {
4.    constructor() {
5.      super();
6.
7.      this.state = {
8.        articles: [
9.          { title: "React Redux Tutorial for Beginners", id: 1 },
10.         { title: "Redux e React: cos'è Redux e come usarlo con React", id: 2
    }
11.       ]
12.     };
13.   }
14.
15.   render() {
16.     const { articles } = this.state;
17.     return <ul>{articles.map(el => <li key={el.id}>{el.title}</li>)}</ul>;
18.   }
19. }
```

What Problem Does Redux Solve?

Redux solves a problem that might not be clear in the beginning: it helps giving **each React component** the **exact piece of state** it needs.

Redux holds up the **state** within a **single location**.

Also with Redux the **logic for fetching and managing the state** lives **outside React**.

The benefits of this approach might be not so evident. Things will look clear as soon as you'll get your feet wet with Redux.

When Should You Use Redux?

Redux is ideal for medium to big apps, and you should only use it when you have trouble managing the state with the default state management of React, or the other library you use.

Simple apps should not need it at all (and there's nothing wrong with simple apps).

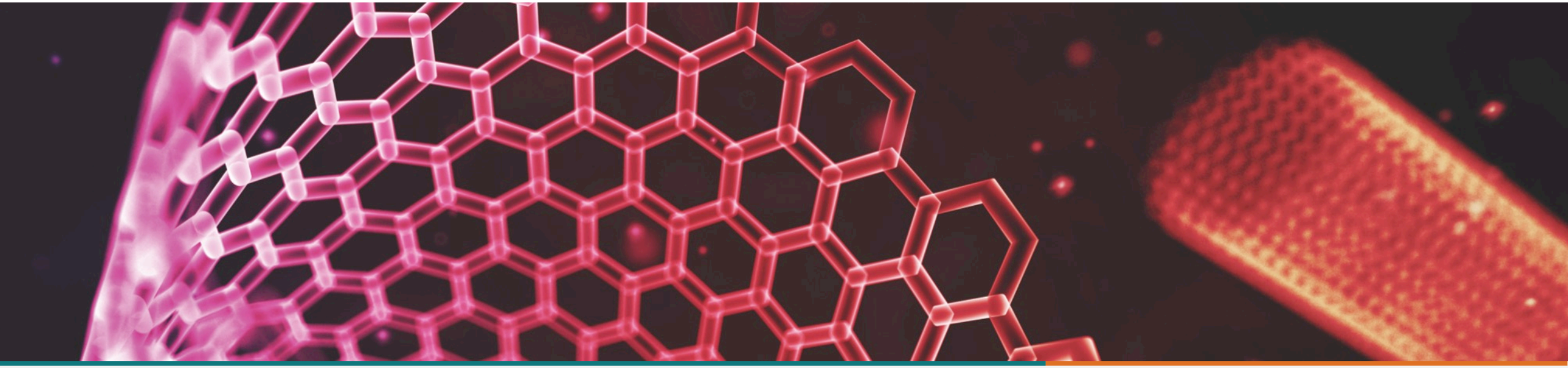
Immutable State Tree

In Redux, the whole state of the application is represented by **one** JavaScript object, called **State** or **State Tree**.

We call it **Immutable State Tree** because it is read only: it can't be changed directly.

It can only be changed by dispatching an **Action**.

Redux Terms



Redux Terms

- Store
- Reducers
- Actions
- Action Creators
- Dispatch

Store: What is a Store?

The **Store** is an object that:

- **Holds the state** of the app
- **Exposes the state** via ***getState()***
- Allows to **update the state** via ***dispatch()***
- Allows to (un)register as a **state change listener** using subscribe()

A store is unique in the app.

Store

Here is an example of importing and creating a store:

```
import { createStore } from 'redux'
import listManager from './reducers'
let store = createStore(listManager)
```

Reducer: What is a Reducer?

A **reducer** is a function that consumes actions and generates a new state. Reducers are supposed to be pure and should not actually generate non-pure data. When given an identical action, a reducer should return an identical state.

A pure function takes an input and returns an output without changing the input nor anything else. Thus, a reducer returns a completely new state tree object that substitutes the previous one.

What a Reducer Should Not Do

A reducer should be a pure function, so it should:

- Never mutate its arguments
- Never mutate the state, but instead create a new one with **Object.assign({}, ...)**
- Never generate side-effects (no API calls changing anything)
- Never call non-pure functions, functions that change their output based on factors other than their input (e.g. Date.now() or Math.random())

Reducer Example

```
const userReducer = (state={}, action)=>{
    switch (action.type){
      case "CHANGE_USER":{
        state = {...state, name: action.name, location: action.location}
        break;
      }
      case "CHANGE_NAME":{
        state = {...state, name: action.name};
        break;
      }
      case "CHANGE_LOCATION":{
        state = {...state, location: action.location};
        break;
      }

    }
    return state;
  }

  module.exports = userReducer;
```

Another Reducer Example

```
const postsReducer = (state = [], action) => {
    if (action.type ==="ADD_POST"){
        return [...state,action.post];
    }
    return state;
}
module.exports = postsReducer;
```

Combining Reducers

The **combineReducers** helper function turns an object whose values are different reducing functions into a single reducing function you can pass to createStore.

The resulting reducer calls every child reducer and gathers their results into a single state object. **The state produced by** combineReducers() **namespaces the states of each reducer under their keys as passed to** combineReducers()

Combining Reducers Example

```
const redux = require('redux');
const userReducer = require("./users");
const postReducer = require("./posts");
const counter = require ('./counter')

const reducers = redux.combineReducers({user: userReducer,posts: postReducer, counter: counter})

module.exports = reducers;
```

Actions

An **action** is a payload of information that describes how the store should be updated. All actions should be objects, with a type field that indicates what the action type is.

Actions are plain JavaScript objects that represent payloads of information that send data from your application to your store. Actions have a type and an optional payload.

Most changes in an application that uses Redux start off with an event that is triggered by a user either directly or indirectly. Events such as clicking on a button, selecting an item from a dropdown menu, hovering on a particular element or an AJAX request that just returned some data.

Even the initial loading of a page can be an occasion to dispatch an action. Actions are often dispatched using an action creator.

What is an Action Creator?

An **action creator** is a function that will generate an action; you would pass parameters to the action creator, and it would generate an appropriate action

Depending on what the action is, reducers can choose to return a new version of their piece of state. The newly returned piece of state then gets piped into the application state, which then gets piped back into our React app, which then causes all of our components to re-render.

So let's say a user clicks on a button, we then call an action creator which is a function that returns an action. That action has a type that describes the type of action that was just triggered.

Action Creator Example

```
changeUser = (name, location) => ({
  type: "CHANGE_USER",
  name,
  location
});

changeUserName = (name) => ({
  type: "CHANGE_NAME",
  name
});

changeUserLocation = (location) => ({
  type: "CHANGE_LOCATION",
  location
});

addPost =(post) =>({
  type: "ADD_POST",
  post

})
incCounter =(number) =>({
  type: "INC",
  number: number

})
decCounter =(number) =>({
  type: "DEC",
  number: number

})

module.exports = {changeUser, addPost,incCounter,decCounter, changeUserLocation,changeUserName};
```

You usually run action creators in combination with triggering the dispatcher:

```
store.dispatch(actions.changeUser('Patrick', 'New York City'))
store.dispatch(actions.addPost('My First Post'))
store.dispatch(actions.addPost('My Second Post'))
```

dispatch() and subscribe()

dispatch(action)

Dispatches an action. This is the only way to trigger a state change.

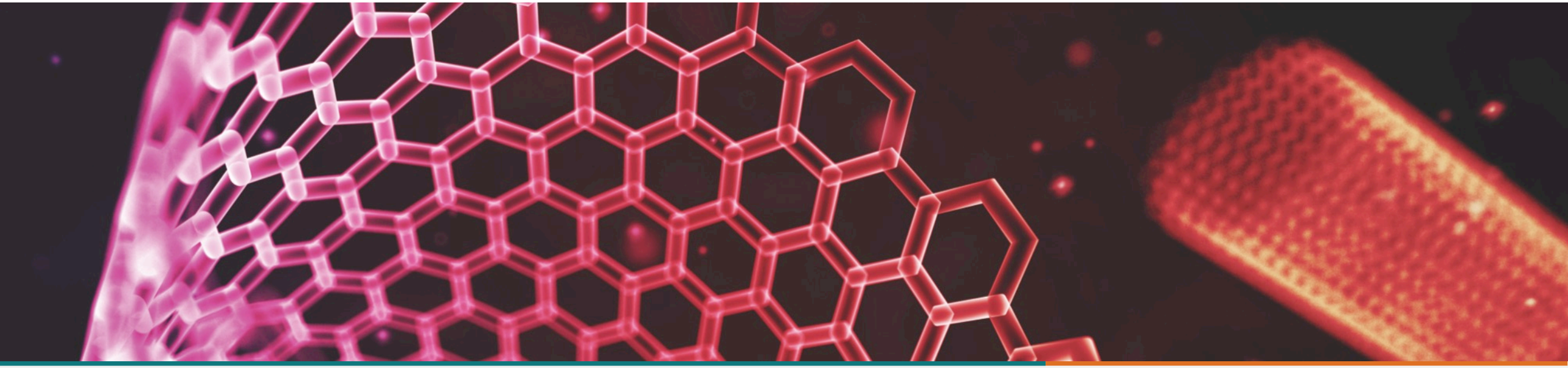
```
store.dispatch(actions.changeUser('Patrick', 'New York City'));
store.dispatch(actions.addPost('My First Post'));
store.dispatch(actions.addPost('My Second Post'));
store.dispatch(actions.incCounter(10));

console.log(store.getState().user);
```

subscribe(listener)

Adds a change listener. It will be called any time an action is dispatched, and some part of the state tree may potentially have changed. You may then call getState() to read the current state tree inside the callback.

```
store.subscribe(() => {
    console.log('State Changed', store.getState())
})
```

Installing Redux



Installing Redux

To install Redux, we just install the npm module:

- **npm install redux**

This allows us to use redux within our application. While Redux is most often paired with React we do not need React to use Redux. We will see a simple example of using Redux with an Express server to demonstrate that you do not need to use React to use Redux.

To install Redux and use it with React we need to install another npm module:

- **react-redux**

React and Redux

Redux has been around for a while now. React now has the Context API what now does pretty much everything that Redux has done. Even though React has the Context API, Redux is still very popular for state management and a lot of developers prefer it over the Context API.

React bindings are not included in Redux by default. You need to install them explicitly we will use the react-redux package:

- **npm install react-redux**

react-redux

The most important method in react-redux that you'll work with is **connect**

What does react-redux's **connect** do? Unsurprisingly it connects a React component with the Redux store.

You will use connect with two or three arguments depending on the use case. The fundamental things to know are:

the **mapStateToProps** function

the **mapDispatchToProps** function

react-redux

What does mapStateToProps do in react-redux? mapStateToProps does exactly what its name suggests: it **connects a part of the Redux state** to the props of a React component. By doing so a connected React component will have access to the exact part of the store it needs.

What does mapDispatchToProps do in react-redux? mapDispatchToProps does something similar, but for actions. **mapDispatchToProps connects Redux actions to React props**. This way a connected React component will be able to dispatch actions.

We will see an example of using React, Redux and react-redux in our lecture code.

react-redux

We saw that **mapStateToProps** connects a portion of the Redux state to the props of a React component. You may wonder: is this enough for connecting Redux with React? No, it's not.

To start off **connecting Redux with React we're going to use Provider**.

Provider is an high order component coming from react-redux.

In layman's terms, Provider wraps up your React application and makes it aware of the entire Redux's store. We have seen providers in both our React Lecture on **useContext** as well as in GraphQL

Why so? We saw that in Redux the store manages everything. React must talk to the store for accessing the state and dispatching actions.

react-redux

```
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import reducer from './reducers/index';
import logger from 'redux-logger';

// We create a store with our combined reducers and add the logger middleware
// Note: Open your dev tools to see how the state changes
const store = createStore(reducer, applyMiddleware(logger));

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

react-redux updates with hooks!

Now that React has hooks, there are some small changes to how we use react-redux.

When we want to tell a component what state to use from the store, we use the **useSelector** hook

```
import { useSelector } from 'react-redux';
```

We then tell the component which piece of state to use (in the example below, its using the array of todo objects from the store):

```
const allTodos = useSelector((state) => state.todos);
```

We can also get the whole state tree like so:

```
const allState = useSelector((state) => state);
```

react-redux updates with hooks!

Now that React has hooks, there are some small changes to how we use react-redux.

When we want to tell a component what state to use from the store, we use the **useSelector** hook

```
import { useSelector } from 'react-redux';
```

We then tell the component which piece of state to use (in the example below, its using the array of todo objects from the store):

```
const allTodos = useSelector((state) => state.todos);
```

We can also get the whole state tree like so:

```
const allState = useSelector((state) => state);
```

react-redux updates with hooks!

For issuing dispatch, we use the **useDispatch** hook:

```
import { useDispatch } from 'react-redux';
```

We then set a variable to use useDispatch:

```
const dispatch = useDispatch();
```

Now And then we call a dispatch like so:

```
dispatch(actions.addUser(data.name, data.email));
```

Questions?