Hello Redux

tl;dr

- Redux is a JavaScript library that tames state management in our applications
- It's not super easy to learn; you have to understand state, pure functions, immutability, and composition.
- Redux is made up of
  - The store
  - Subscriptions
  - Actions
  - Reducers (state-changer functions)
- You dispatch actions to a reducer which generates a new state object -- a very controlled process

It's a library

Let's get this out of the way. Redux is ...

a JavaScript library

- Written by Dan Abramov
- If you want to use redux, include it in your JavaScript program.
- Client-side or server-side? Yes. Either.

You install redux via npm

```
$ npm install --save redux
+ redux@10.3.2
added 3 packages in 6.318s
$
```

You have to include it

```
<script src="redux.js"></script>
```

- Or hopefully you have webpack or another module loader

Why use Redux?

Philosophy behind it

The single responsibility principle is a computer programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. Robert C. Martin expresses the principle as, "A class should have only one reason to change."

- From wikipedia

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

- Redux manages all of the data and access to read/write that data.

SRP simplifies things!

- The cost you pay is you have to learn this tool.

Redux has concepts we need to know

1. State
2. Pure functions
3. Immutability
4. Composition

Concept 1: State

Application state

- A lightswitch can have two states - The "on" state and "off" state
- A color can have 16,777,216 states:

```
{
  red: 0-256,
  green: 0-256,
  blue: 0-256
}
```

- An auto manufacturing plant can have 345,673,568,233,545 states:

```
{
workers with names, addresses, phone numbers, etc
cars with statuses, makes, colors, engines, etc.
balance sheet
assembly line machines
weather conditions,
}
```

The more complex a state becomes, the harder it is ...

to control,
to debug,
to organize

This is what Redux simplifies for you!

Concept 2: Pure functions

Functional programming ideas are at the root of Redux

- NOT OBJECT-ORIENTED
- You Java devs are not going to love this.
- You JavaScript devs are!

- Pop quiz: What does this mean?

```
y = f(x)
```

- Y is a function of X.
- But what does that mean?
- When x changes, y changes in a predictable way.

- Always produce X result from Y input.
- Never change anything external
- Never read from anything external

This is a **pure** function!

Pure

```
function square(x) {
  return x * x;
}
Math.sin(y);
arr.map((item) => item.id);
```

Impure

```
function get(id) {
  fetch(url, {id}).then(p => person=p);
}
Math.random(y);
```

In Redux, state is only changed thru a pure function called a *reducer*

```
newState = f(oldState, actionType, payloadData)
```

Concept 3: Immutability

- Something that is immutable won't change.
- In Redux, state must be treated as immutable.

- So how do you change things then?
- We allow Redux to change state for us as it runs our reducer function.

Concept 4: Composition

Composition

When you create a larger, complex thing that is made up of smaller, simpler things

The smaller things are easier to create and maintain

In Redux we use this with ...

reducers
actions
enhancers

The parts of Redux

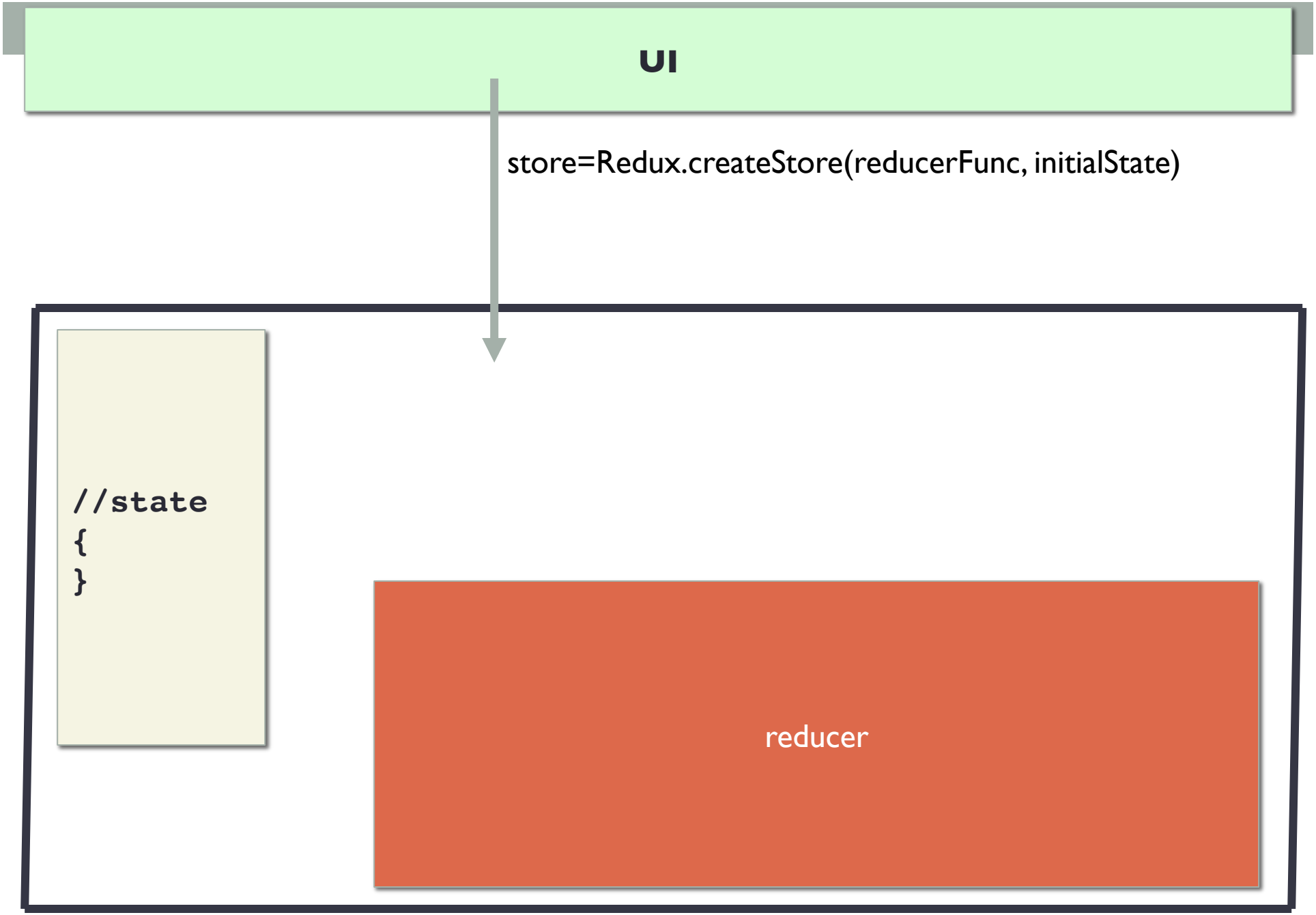
state, store, listeners, actions, and reducers

The parts of Redux

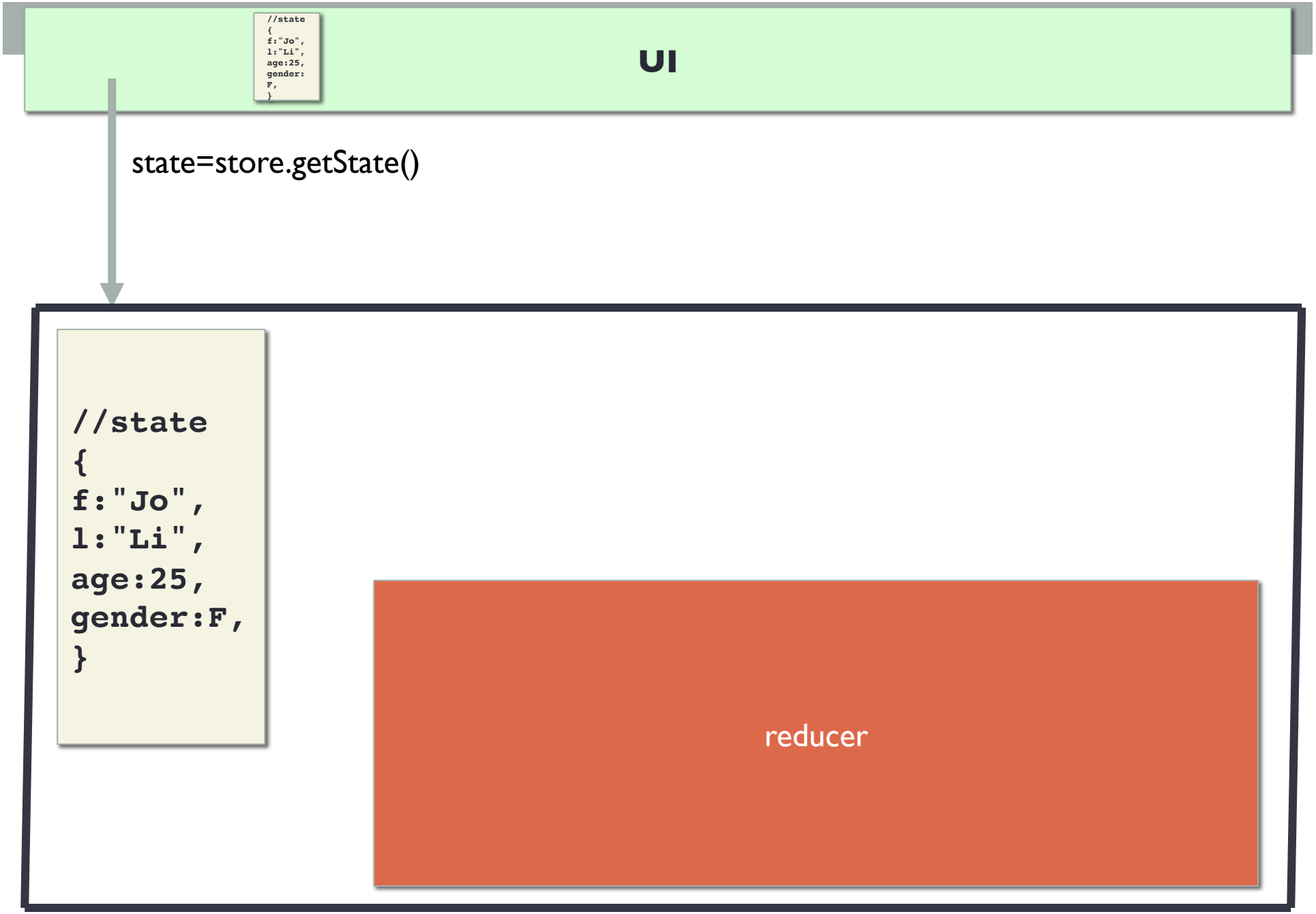
1. The store
2. Subscriptions
3. Actions
4. Reducers

Part 1: The store

- A single object which encapsulates the state of your application
- State is a single JSON object
- No functions. No logic.
- Reading state
- Altering state
- Principle: You must NEVER change state directly


UI
store=Redux.createStore(reducerFunc, initialState)
//state
{
}
reducer

The store


//state
{
f:"Jo",
l:"Li",
age:25,
gender:
F,
}
UI
state=store.getState()
//state
{
f:"Jo",
l:"Li",
age:25,
gender:F,
}
reducer

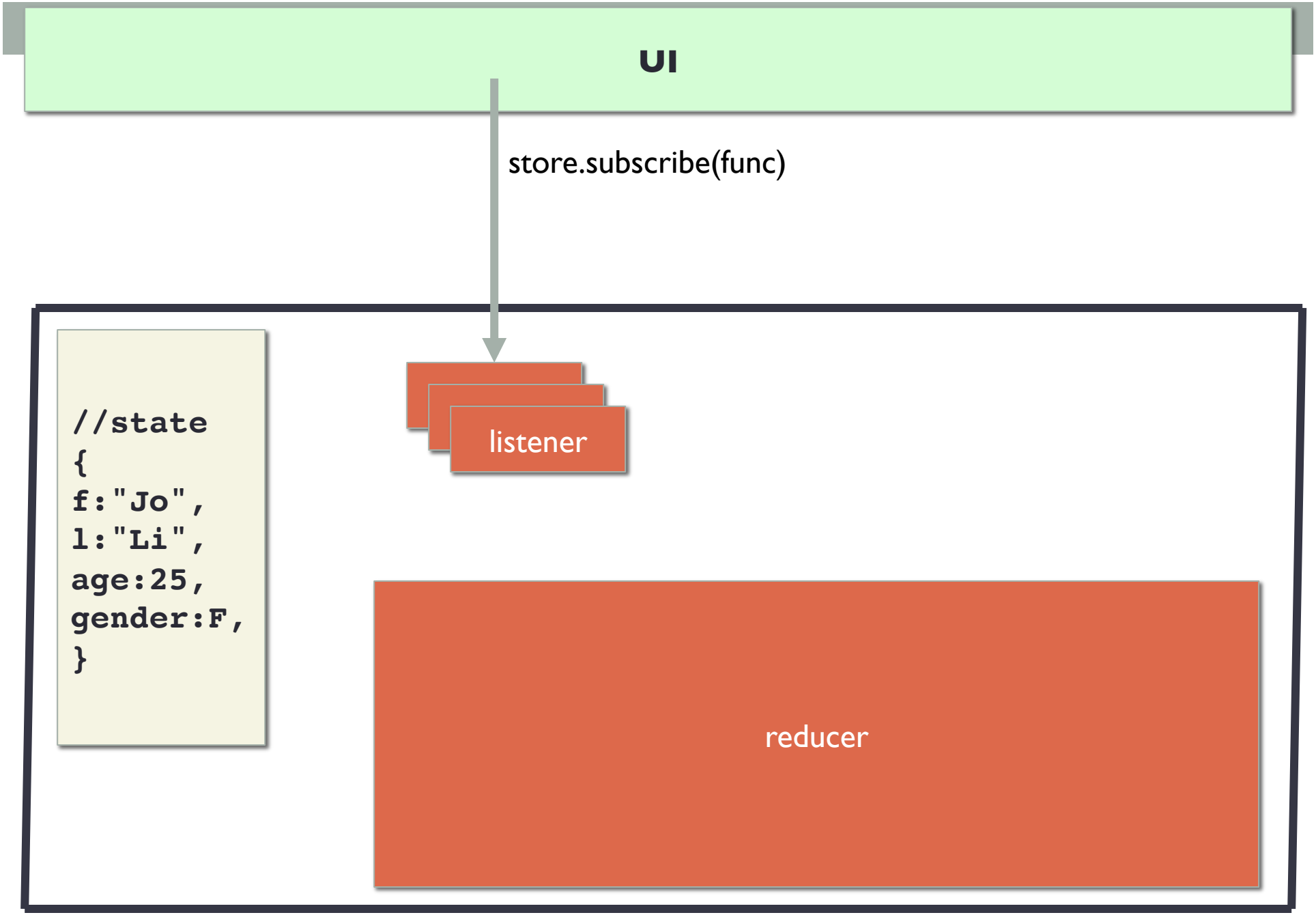
The store

Part 2: Subscriptions

- When data changes, you want your app to respond.
- ie. re-render the UI so the new data appears.

- To make that happen, you can register a JavaScript callback to be run whenever any dispatch happens.
- This is called *subscribing*

- The store maintains a list of subscriptions and automatically runs them when a dispatch happens


UI
store.subscribe(func)
//state
{
f:"Jo",
l:"Li",
age:25,
gender:F,
}
listener
reducer

The store

Part 3: Actions

- An structure that describes the change to be made to the state.
- Always has a type
  - A string
  - One member of a finite pre-defined set
  - Kind of like an enum
- Usually has a payload
  - The data that should be changed
- Examples:

```
const action = {type:"HAVE_A_BIRTHDAY"};
const action = {
  type:"SET_COURSE",
  heading: "207-mark-99",
  speed:"warp 1.0"
};
```

There is a **finite** list of things that can be done to change state.
That finite list defines all of your actions

Actions are always *dispatched* to change state.

Dispatching sends an action to the reducer

Part 4: Reducers

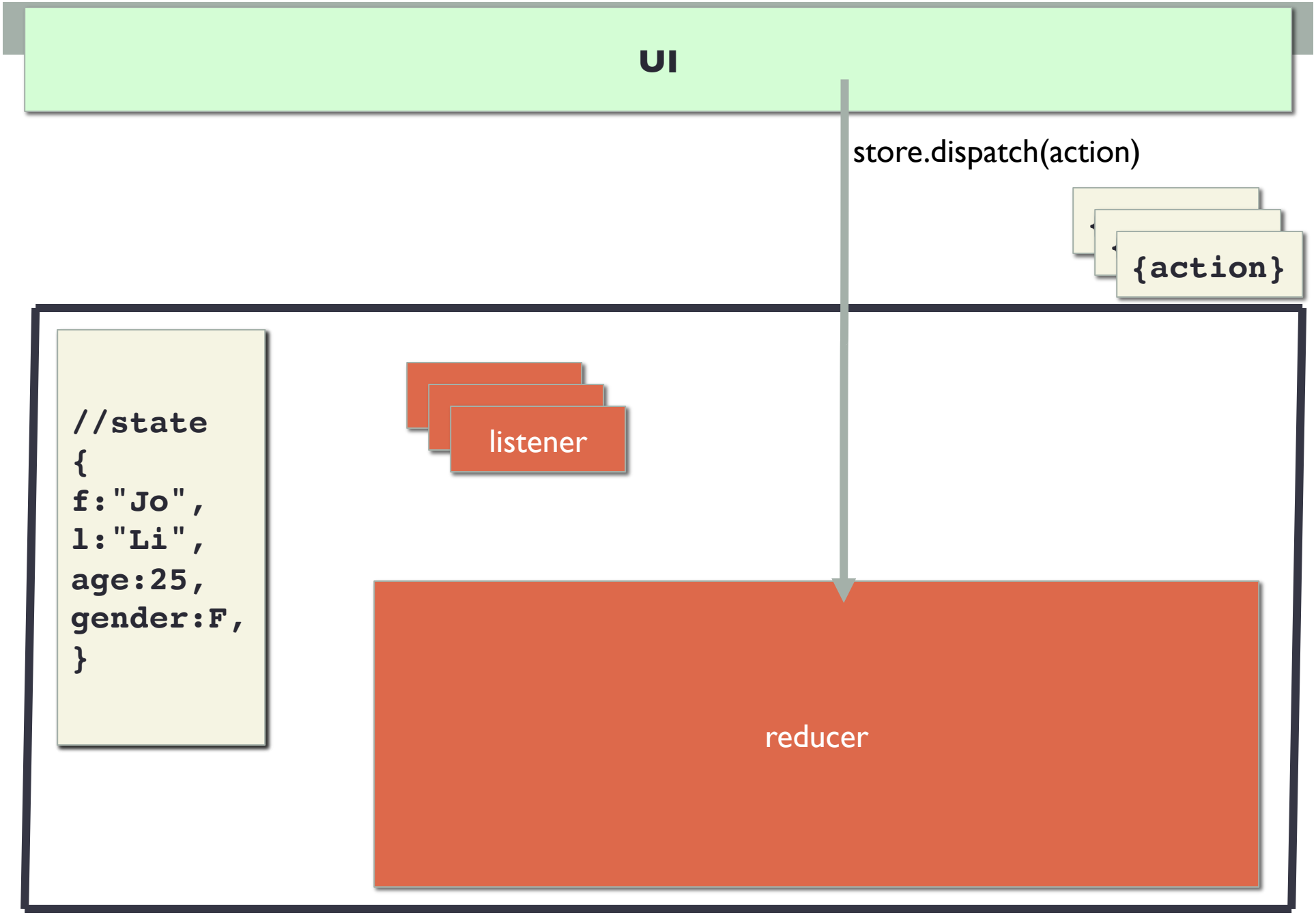
- Reducers are functions that know how to mutate (change) state, once given an action.

```
NewState = fn(oldState, action);
```

- We activate the reducer by *dispatching* an action.
- A pure and synchronous function! Thus it cannot change state. It can read the oldState as a parameter and will return a copy of that state that is different by the action.
- There will always be one case for each action.

Behind the scenes, Redux does something like this:

```
function dispatch(action) {
  const newState = reducer(state, action);
  state = newState;
  for (let func of subscriptions)
    func()
}
```


UI
store.dispatch(action)
{action}
//state
{
f:"Jo",
l:"Li",
age:25,
gender:F,
}
listener
reducer

The store

There's more!

- But this gives us plenty to think about for now.
- More to come later.

tl;dr

- Redux is a JavaScript library that tames state management in our applications
- It's not super easy to learn; you have to understand state, pure functions, immutability, and composition.
- Redux is made up of
  - The store
  - Subscriptions
  - Actions
  - Reducers (state-changer functions)
- You dispatch actions to a reducer which generates a new state object -- a very controlled process

Creating the store

The store is one JavaScript object that holds everything related to state

- The state itself
- Controlled way to read state
- Controlled ways to change state
- A list of things to do when state changes

- So, how do you create the store?

- To create the store, you run Redux's createStore method

createStore()

Syntax:

store = createStore(reducer, initialState);

Where ...

- *reducer* is a function that receives a state object and an action object and returns a state object. In other words …

```
reducer = (state, action) => state
```

- *initialState* is a plain object

The simplest possible store

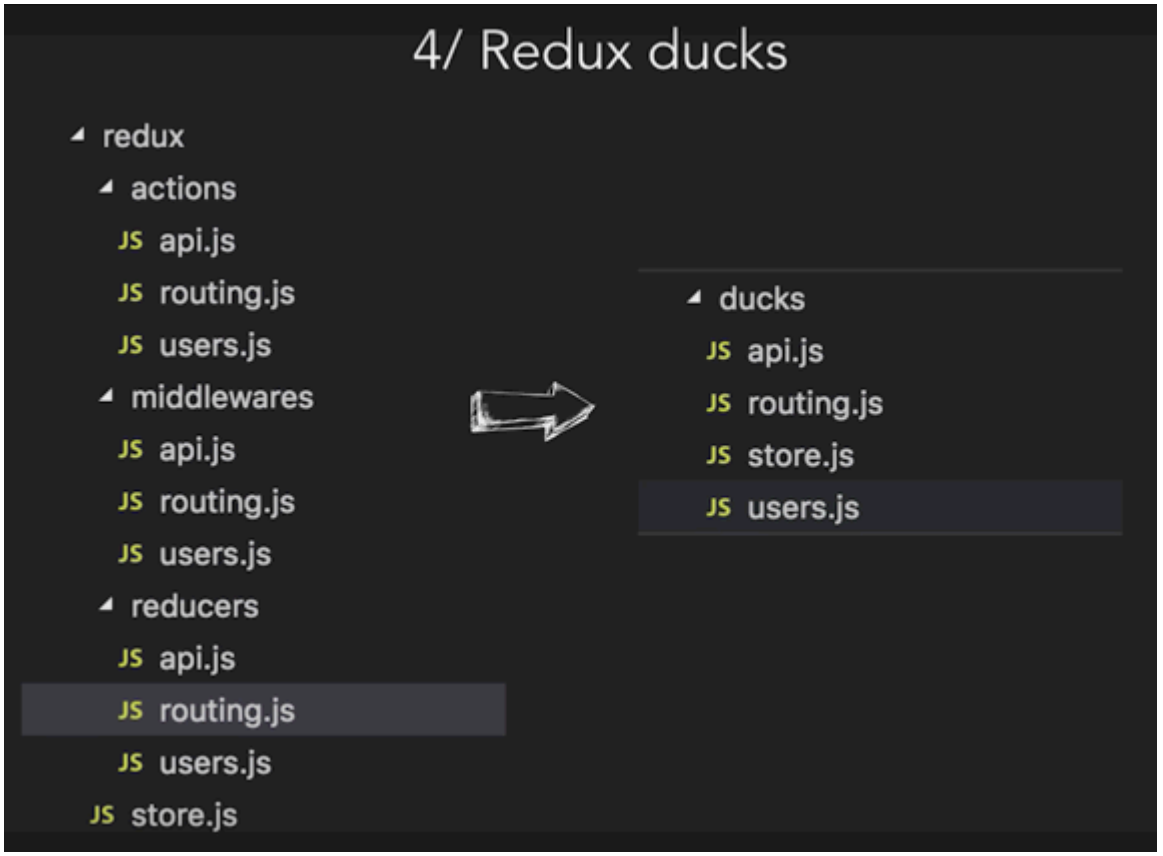
```
import { createStore } from 'redux';

const reducer = (state, action) => state;
const initialState = {};
export const store =
  createStore(reducer, initialState);
// Now we can do things with the store!
```

Hands on creating the store

Redux ducks

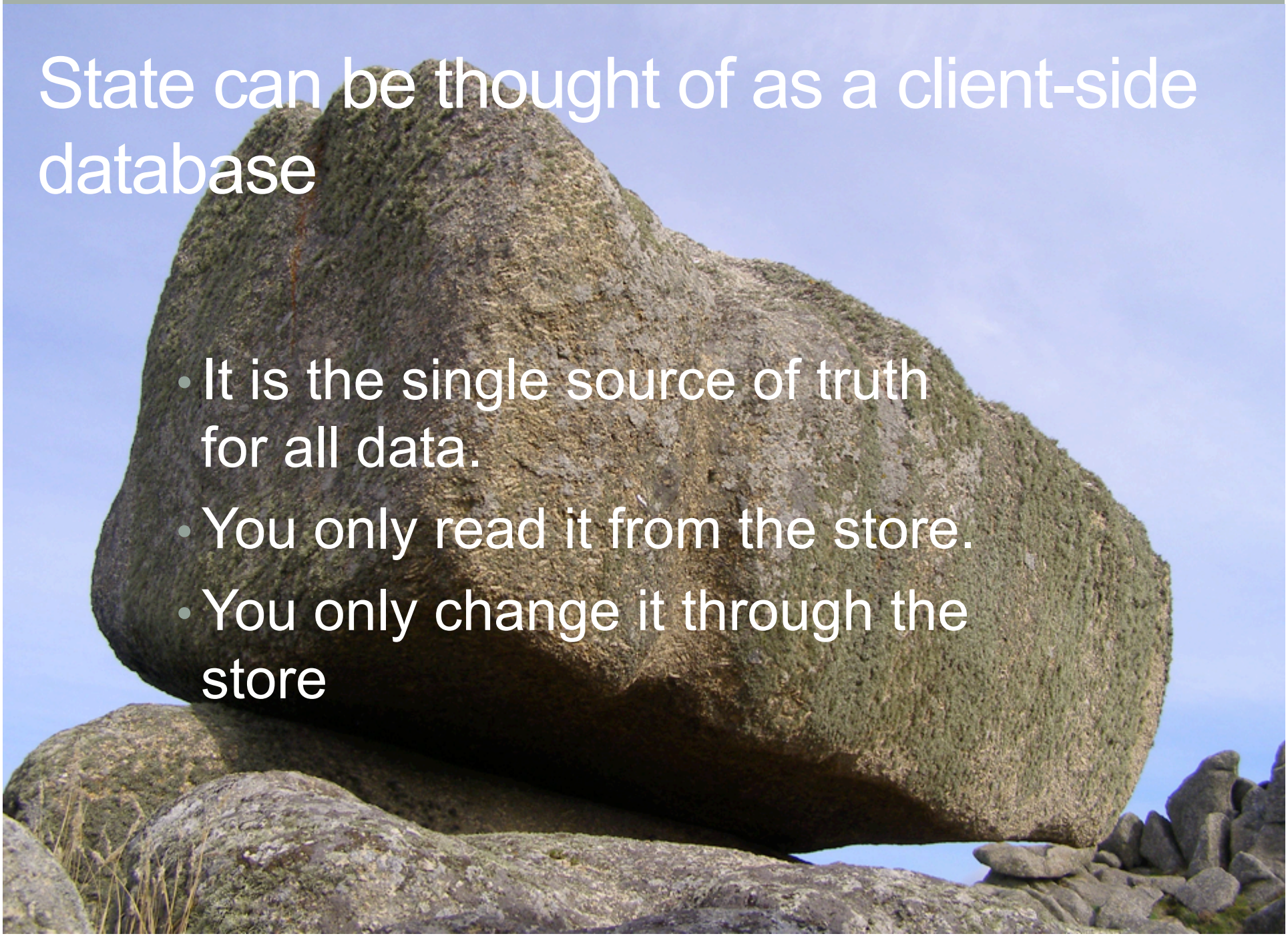
- Rather than having folders for (middleware, action constants, action types, reducers, models, etc), you have one file that has those things together:



tl;dr

- The store is where all the action happens!
- You create it by calling Redux's createStore() method
- It receives a reducer function and an initial state object

State and Subscriptions



State can be thought of as a client-side database

- It is the single source of truth for all data.
- You only read it from the store.
- You only change it through the store

We should set the initial state when we're creating the store.

```
const initialState = {
  first: "Jo",
  last: "Bennett",
  city: "Tallahassee",
  state: "Florida"
}
export const store =
  createStore(reducer, initialState);
```

- This method is self-documenting.

Hands-on state

What should go in state and what should not?

- Does the data change?
- Should the data be the same between pages/views?
- Should the data be the same when you refresh this page/view?
- If all are yes, then it goes in state. If not, no.

Subscriptions

When state changes, we should redraw the UI

- Different data => different display

- Should we call a redraw() UI function?
- What if we have to change 2 things upon a data change?
- 20?
- 200?
- Do we really want to keep track of all of that?

We can subscribe to every state change

- We know we'll always dispatch an action to change state
- Redux allows us to register X functions to run every time state is changed through a dispatch:

- In the UI:

```
store.subscribe(redraw);
function redraw() {
  // Logic to re-render the UI goes here
}
```

Redux keeps track of all subscriptions no matter where they were registered

- So, you can subscribe in lots of places in the UI and Redux fires them all when the state changes.

- And it doesn't have to be just the UI. Register any function that needs to run when state changes.
  - Upload data to a server
  - Save to local storage
  - Push it through a socket to another client
  - Etc.

Hands-on subscriptions

Actions and Reducers 101

tl;dr

- ALL changes to state can be described by an action
- An action is an object with a type and an optional payload
- The reducer is a function that describes how state can change
- A reducer always has this shape:

```
reducer = (oldState, action) => newState;
```

- Several common mistakes can be avoided if you watch out for them:
  - Reference assigning accidentally
  - Not returning state
  - Mutating state directly

We must only allow state to change in a controlled way

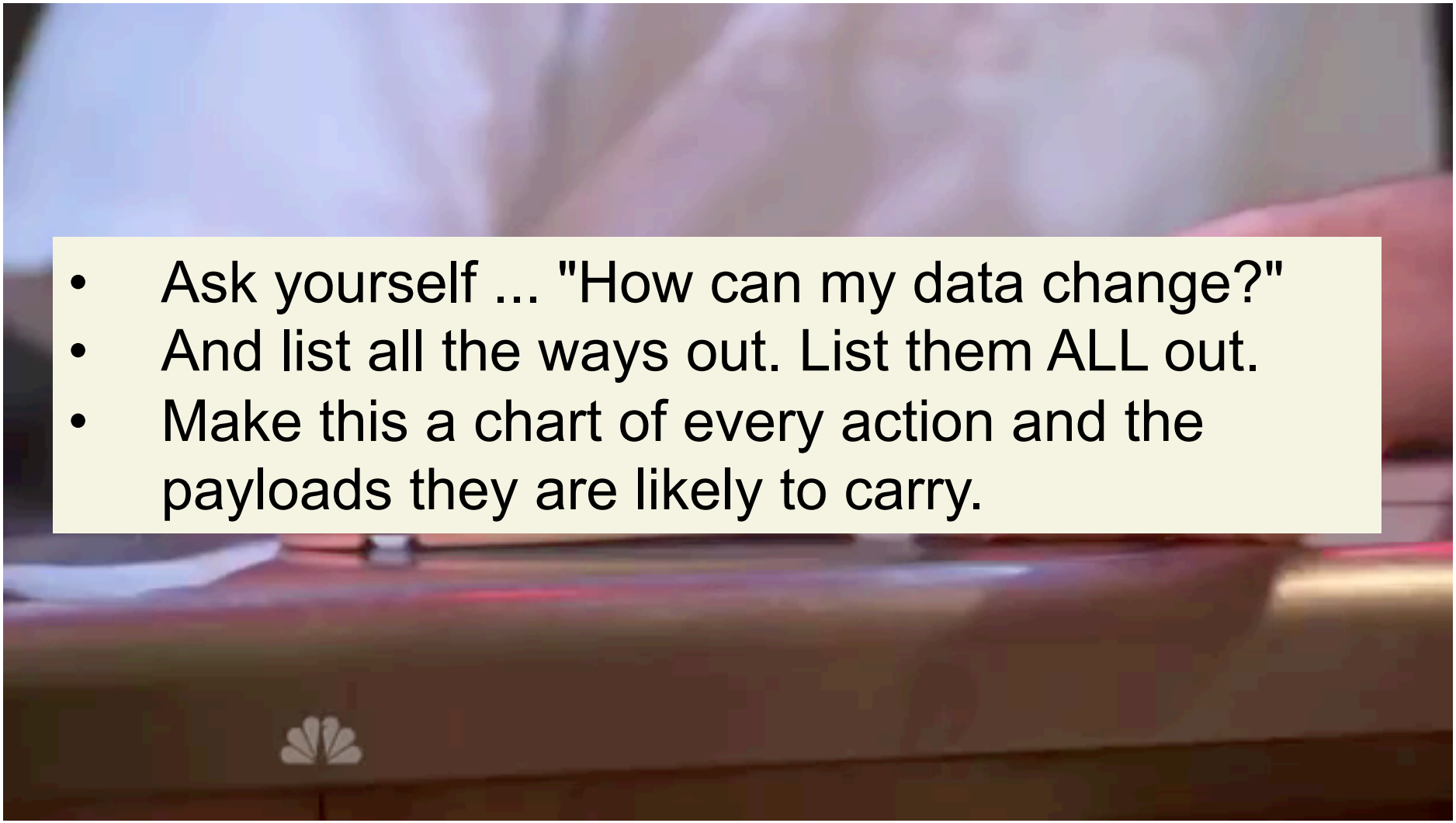
When done right, state will only change when we dispatch an action that is handled in a reducer.

Actions

An action is an object which fully describes the change to be made to state

- It always has a "type" property which is a string
- It usually has a payload ... a series of additional properties that may be needed.
  - "Move forward? How many spaces?"
  - "Increase balance? By how much?"
  - "Change the name? To what?"

You'll begin by listing out all the valid ways that data can change



- Ask yourself ... "How can my data change?"
- And list all the ways out. List them ALL out.
- Make this a chart of every action and the payloads they are likely to carry.

The Reducer

A reducer is a pure function that receives in the old state and an action and returns a different state object

It MUST have this shape:

```
(oldState, action) => newState
```

All reducers do something like this:

```
if (action.type === "foo") {
  const newState = getCopyOfOldState(oldState);
  newState.prop1 = action.newValue1;
  newState.prop2 = action.newValue2;
  return newState;
}
```

Most folks use a switch statement

```
const reducer = (state, action) => {
  if (!action) return state;
  switch (action.type) {
    case "SET_FIRST":
      return {...state, first:action.first};
    case "SET_ZIP":
      return {...state, zip:action.zipcode};
    default:
      return state;
  }
}
```

You can easily mutate objects with the object spread operator

The ellipsis operator *spreads* out the properties of an object

```
const newState = {...oldState};
```

(The above simply makes a copy of the object)

To add a new property

If key1 doesn't already exist, it adds it to the new object

```
const newState = {...oldState, key1:value1};
```

To update an existing property

If key1 already exists, it clobbers the old value

```
const newState = {...oldState, key1:value1};
```

You can easily mutate arrays with the array spread operator

The ellipsis operator *spreads* out the elements of an array

```
const newState = [...oldState];
```

(The above simply makes a copy of the array)

To add a new element

Adds value1 to the end of the new array

```
const newState = [...oldState, value1];
```

To update an existing element

```
const idx = oldState.indexOf(oldValue);
const newState = [...oldState.slice(0,idx),
                  newValue, ...oldState.slice(idx+1)];
```

How to avoid the worst Redux mistakes

Some people have to learn the hard way. :-(

Rookie mistake 1: Reference assigning

```
const newState = oldState
newState.prop1 = action.newValue1
```

- State will change here, but change detection fails because Redux does essentially this behind the scenes:

```
if (newState !== oldState)
  runAllTheListeners()
```

- When you change newState, you're also changing oldState
- Redux will only recognize that state has changed when a deep comparison of oldState is different from newState.

Rookie mistake 2: Not returning state

- If you don't return a state, the dispatch causes state to be undefined!
- Always return the old state if ...

1. There is no action
2. Action has an unknown type

Rookie mistake 3: Changing state directly

- You can totally do this...

```
const state = myStore.getState();
state.prop1 = "Some new value";
```

And it absolutely works! State will change. But there's never a good reason to do this

- Lots of reasons why NOT to ...

1. Subscriptions won't fire
2. Other devs can't find where state is changing
3. Impossible to debug

tl;dr

- ALL changes to state can be described by an action
- An action is an object with a type and an optional payload
- The reducer is a function that describes how state can change
- A reducer always has this shape:

```
reducer = (oldState, action) => newState;
```

- Several common mistakes can be avoided if you watch out for them:
  - Reference assigning accidentally
  - Not returning state
  - Mutating state directly

Advanced Actions

Cool things to make actions more understandable

tl;dr

Certain action practices may not give you additional capabilities but they can make your code cleaner:

- Action type constants
- Action type enumeration
- Action creators
- Action creator enumeration

Action constants

Q: What happens if you have a reducer with an action of

```
case "SET_FIRSTNAME":
return {...person, first: action.first}
```

And you dispatch like this:

```
const first = txtBox1.value;
store.dispatch({type:"SET_FRISTNAME", first});
```

A: Nothing happens. No errors are thrown. Try debugging that!!

Q: Now what happens if you changed the reducer like this:

```
const SET_FIRSTNAME = "SET_FIRSTNAME";
...
case SET_FIRSTNAME:
return {...person, first: action.first}
```

And try this:

```
const SET_FIRSTNAME = "SET_FIRSTNAME";
...
const first = txtBox1.value;
store.dispatch({type:SET_FRISTNAME, first});
```

A: JavaScript throws during development

So some devs create an action-types.js

```
export const SET_LATLON = "SET_LATLON";
export const SET_PRECIP_PROBABILITY=
"SET_PRECIP_PROBABILITY";
export const SET_PRECIP_TYPE = "
SET_PRECIP_TYPE";
```

- Then, everywhere one is used ...

```
import { SET_LATLON } from './action-types.';
...
case SET_LATLON:
// Do stuff
```

The action enumeration

Note: This technique may not be popular yet, but it is very clean

- Not a requirement, just a good idea.
- Eliminates magic strings and thus typos

Create an action "enumeration"

- All uppercased. Underscores between the words.

```
const ActionTypes = {
  ADD_FOO: "ADD_FOO",
  REMOVE_FOO: "REMOVE_FOO",
  SUBMIT_ORDER: "SUBMIT_ORDER",
  SET_FOO: "SET_FOO",
};
export default ActionTypes;
```

To use your enumeration...

```
import actionTypes from './action-types';
...
case actionTypes.SET_LATLON:
// Do stuff
```

Action creators

Actions are tough to remember also

Again, you're coding and you're trying to set the user's location. what does the action look like?

```
store.dispatch( /* what goes here? */)
```

Maybe you remember the action type, but what else is in the payload?

Solution: Use an action creator!

```
function setUserAction(first, last, birthdate)
{
  return {
    type: SET_USER, first, last, birthdate
  };
}
```

- Then you

```
store.dispatch(setUserAction("April",
  "Ludgate", aDate));
```

- This works because the function returns a properly formatted action.

Action enumerations

Note: This is an original concept so you won't find it by searching online.

- Not that others haven't discovered it independently also.

If we use action creators, we don't have to remember the action type and payload.

But you still need to remember the name of the action creator!

Solution: Put all of your action creators in an enumeration

In actions.js

```
const setUser =
  (first, last, birthdate) =>
    ({type:SET_USER, first, last, birthdate});
const setPassword =
  pass => ({type:SET_PASS, pass});
// etc., etc.
export const actions = {
  setUser,
  setPassword,
  /* etc. etc. */
};
```

Then to dispatch an action ...

```
import { actions } from 'actions.js';
...
store.dispatch(actions.
```

- ... and your IDE's intellisense kicks in so you can pick an action from the list and get prompted for the arguments to pass in.

```
store.dispatch(actions.setUser(
  "Chris","Trager",someDate));
```

tl;dr

Certain action practices may not give you additional capabilities but they can make your code cleaner:

- Action type constants
- Action type enumeration
- Action creators
- Action creator enumeration

Composing Reducers

tl;dr

- The reducer are the most complex part of Redux, mostly because state is a complex object
- We can greatly simplify the reducer by decomposing it into sub-reducers each handing a slice of the state called sub-states
- We then compose a root reducer from our sub-reducers
- Each sub-reducer is much easier to understand and maintain

Reminder: What do these do?

```
const newObj = {...oldObj, prop1:val1, prop2:val2};

const obj2 = { key1: someFunc(someval) };

store.dispatch({type:"notarealtype"});
```

The more complicated state becomes, the more complicated the reducer must become

For instance ...

Simple

- To set data which is immediately inside the state object:

```
case SET_GENDER:
return {...state, gender: action.gender};
```

More complex

- To set data which is inside an embedded object in state:

```
case SET_FIRST_NAME:
return {...state, name: {...state.name, first:
action.first}};
```

Even more complex

- To set data which is 2 levels deep:

```
case SET_COMPANY_CEO_NAME:
return {...state, company: {...state.company,
{ ceo: {...state.company.ceo, name:
action.name } } } } };
```

Solution: Create sub-reducers

- Sub-reducer is a reducer function that handle a slice of state.

- Remember that a reducer is a function that returns a state object so ...

1. We'll split the state into sub-states
2. We'll handle each sub-state with a sub-reducer
3. We'll combine the sub-reducers to form the main reducer

For example:

- Pseudo-code ...

```
rootReducer = (state, action) => return {
  name: nameReducer(state.name, action),
  location: locationReducer(state.location, action),
  picture: pictureReducer(state.picture, action),
  ...
};
```

Remember, name, location, and picture are just objects.

Two options

1. Use combineReducers
   - Constrained to a square-shaped data
   - Easier for another dev to understand
2. Do it ourselves manually
   - More control and flexibility
   - More code to write

Using Redux's combineReducers function

combineReducers() is a function built-in to Redux

- It's easy to use but limited.
- It assumes you're naming the sub-reducer function the same as its key in the state object.
- For example ...

```
import { createStore, combineReducers } from 'redux';

const name = (state, action) => {
  // Here, "state" is only the object under
  // the "name" key.
}
const location= (state, action) => {
  // Here, "state" is only the object under
  // the "location" key.
}
const picture= (state, action) => {
  // Here, "state" is only the object under
  // the "picture" key.
}

const rootReducer = combineReducers(name, location,
picture);
```

Manually combining reducers

If that is too limiting, we can do the same thing manually

We will do essentially the same thing as combineReducers() except that we have more control and flexibility.

Combining manually

```
const rootReducer = (state, action) => {
  return {
    ...personReducer(state, action),
    name: nameReducer(state.name, action),
    location: locationReducer(state.location, action),
    picture: pictureReducer(state.picture, action)
  }
}
```

- And you could do this multiple levels deep.
- As many nested levels as you like.

tl;dr

- The reducer are the most complex part of Redux, mostly because state is a complex object
- We can greatly simplify the reducer by decomposing it into sub-reducers each handing a slice of the state called sub-states
- We then compose a root reducer from our sub-reducers
- Each sub-reducer is much easier to understand and maintain

Redux middleware

tl;dr

- Reducers must be pure but sometimes we need side-effects
- Middleware are functions that are allowed to have side-effects
- Writing them isn't easy because of the strange syntax but they are extremely powerful and yet clean
- To use them you must register them with the store and all dispatches will be run through all middleware

The middleware pattern is an expression of the open-closed principle

It allows the developer to insert as many interim functions into the stream of processes without making changes to the processes themselves.

Middleware has a "next()" function

- In any middleware function, you can call next() to pass control to the *next* process.
- We can string any number of middlewares along, each doing it's own specialized functionality until we have a truly beautiful and powerful machine.

Redux middleware are functions that give you more control over what is happening

- Redux allows the insertion of functions into the flow between the dispatch method and the reducer invocation.
- Each can read state and the current action
- Each can introduce additional steps to the process and can alter the process based on logic ...
  - stop the current action or pass it along the chain
  - dispatch new actions -- as many as you like
  - alter the action before it is passed along

What kinds of things will you do with middleware?

- Saving state to local storage and restoring it on startup
- Logging actions and pre/post state for each dispatch
- All asynchronous processing
  - Ajax calls
  - setTimeout() and/or setInterval()
  - Anything involving promises or async/await functions
- Complicated, multi-step processes
- Long-running processes
- Processes that need to re-dispatch or dispatch further actions
- Processes that may have side-effects (non-pure functions)

Shape of middleware

Redux expects your middleware to conform to a strange interface

Warning: While the concept of middleware isn't that tough to understand, the code can be. To smooth that out, just view the code as a recipe and don't waste brainpower on understanding why it is the way it is.

All middleware must be shaped like this:

```
function (store) {
  return function (next) {
    return function (action) {
      // Do stuff in here
    }
  }
}
```

Or written differently ...

```
store => next => action => {
  // Do stuff in here
}
```

But most accurately ...

```
({getState, dispatch}) => next => action => {
  // Do stuff in here
}
```

Therefore the middleware layer has access to four things:

1. the action object being dispatched
2. the dispatch() method
3. the getState() method
4. the next() method

```
{getState, dispatch} => next => action => {
  next(action);
}
```

Example: Do nothing but pass control along

```
{getState, dispatch} => next => action => {
  if (action.type === types.FOO)
    dispatch(actions.doThing());
  next(action);
}
```

Example: Conditionally dispatch a new action

```
{getState, dispatch} => next => action => {
  if (someCondition)
    action.prop = "Some new value";
  next(action);
}
```

Example: Alter the action in progress

```
{getState, dispatch} => next => action => {
  next(action);
  runAFunction();
}
```

Example: Run some code AFTER the next step returns is run

```
{getState, dispatch} => next => action => {
  if (getState().someCondition)
    // Do nothing
  else
    next(action);
}
```

Example: Under certain conditions, abort the dispatch

How to register middleware

The store must be made aware of middleware

- It turns out that createStore actually has this shape:

```
createStore(
  reducerFunction, // The main reducer
  initialState,    // The default state
  ...enhancers     // A list of store "enhancers"
)
```

- Enhancers make the store better. Middleware can be converted to an enhancer using the applyMiddle:

```
const enhancerList = applyMiddleware(...functions);
```

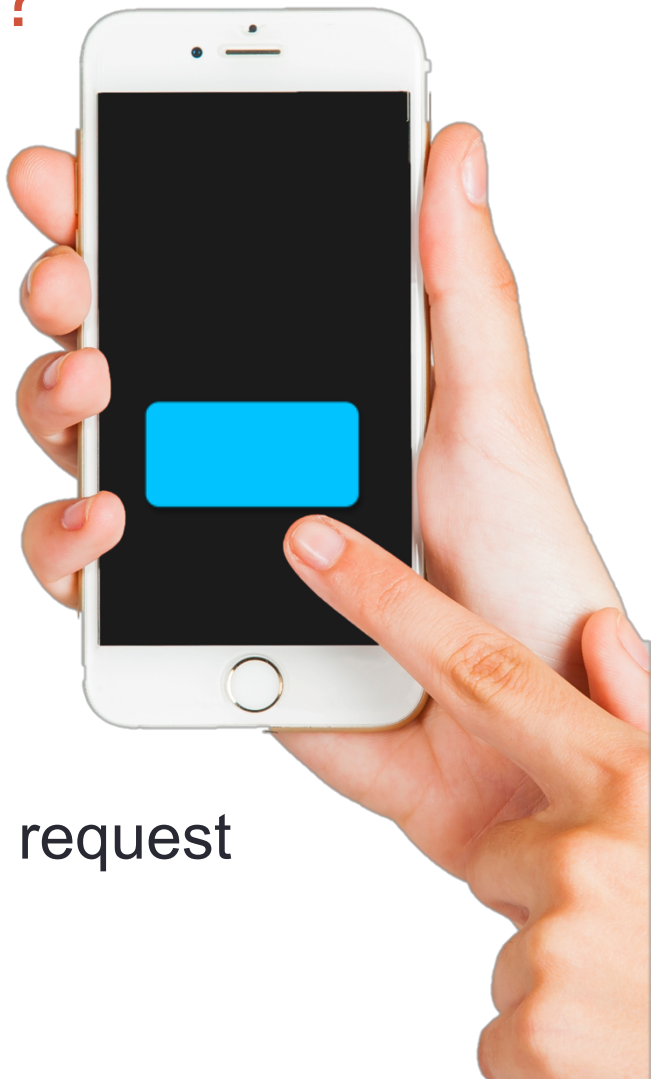
tl;dr

- Reducers must be pure but sometimes we need side-effects
- Middleware are functions that are allowed to have side-effects
- Writing them isn't easy because of the strange syntax but they are extremely powerful and yet clean
- To use them you must register them with the store and all dispatches will be run through all middleware

Ajax with Redux

Let's say the user clicks a button which gets data from the server and displays it.

When should the dispatch happen?

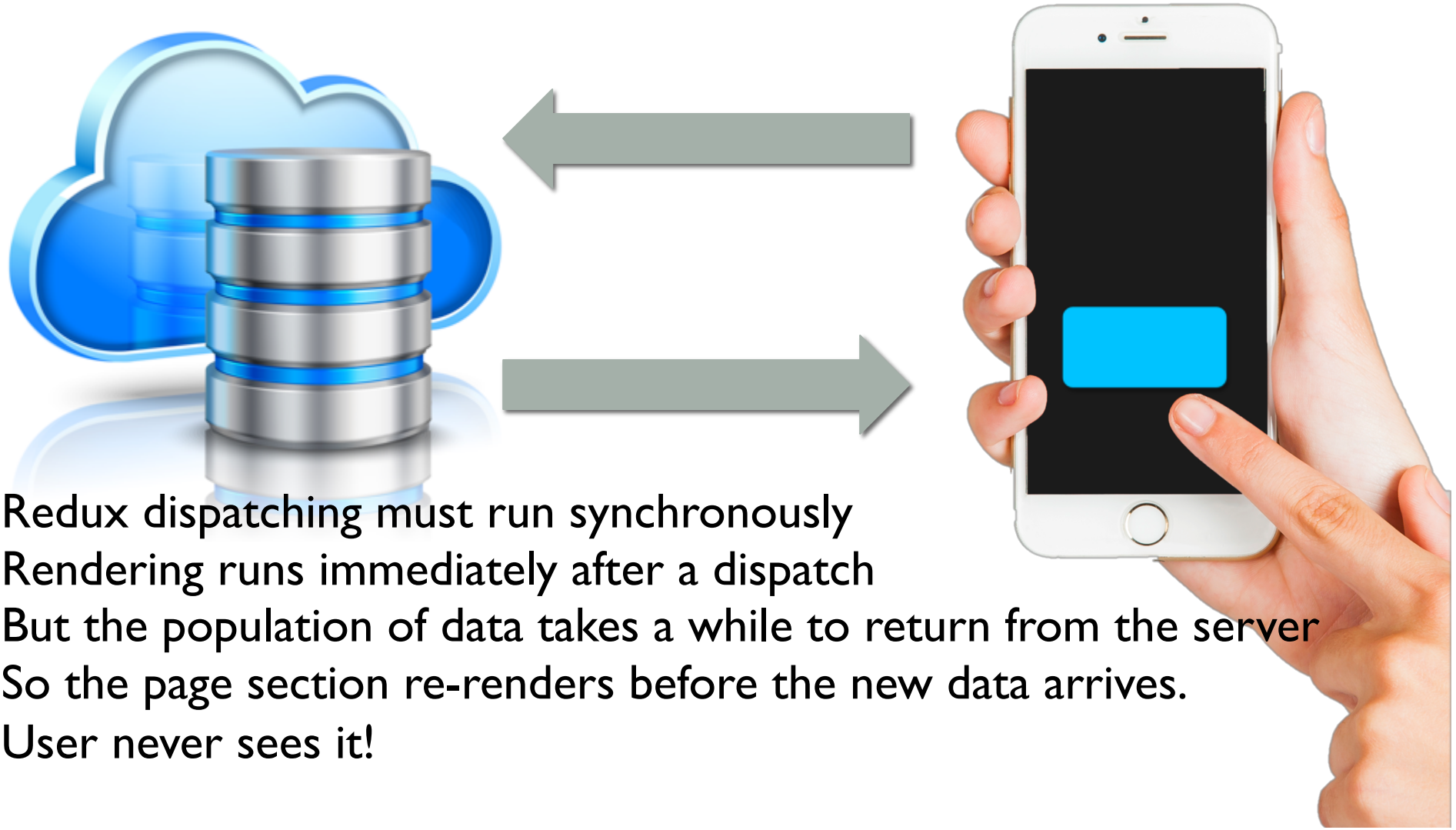


On click?

Yes, because we need to send the request

But ....

... the request is asynchronous



Redux dispatching must run synchronously
Rendering runs immediately after a dispatch
But the population of data takes a while to return from the server
So the page section re-renders before the new data arrives.
User never sees it!

If only we could dispatch an action, but delay the reducer until we get a response from the server.

Hmm. Is there a thing that allows us to run asynchronously and have access to the dispatch function?

Middleware!

- So no matter how you do it, middleware must be involved when you make Ajax calls.
- Here's one method...

To run Ajax calls cleanly through Redux ...

1. Create the reducer that will run when an Ajax response is received
2. Create the middleware function that will run when the Ajax call is dispatched
3. Register the middleware
4. Run the middleware by dispatching an action

Shall we? ➔

1. Create the reducer that will run when an Ajax response is received

```
function reducer(state, action) { {
 switch (action.type) {
  case OTHER_TYPES_HERE:
   return {...state, other:"payloads"};
  case SET_PERSON:
   return {...state, person:action.person};
  case OTHER_STUFF_BELOW:
   return {...state, foo:action.bar};
  default:
   return state
 };
}
```

This is the data that will come from the server

2. Create the middleware function that will eventually run

```
const getPersonMiddleware =
{getState, dispatch} => next => action => {
  if (action.type === FETCH_PERSON) {
    fetch(`/api/person/${action.personId}`)
      .then(res => res.json())
      .then(person => dispatch({
        type: SET_PERSON, person
      })
    );
  }
  next(action);
}
```

3. Register the middleware

```
const middleware =
  applyMiddleware(getPersonMiddleware);
const store =
  createStore(
    rootReducer,
    initialState,
    middleware);
```

4. Run the middleware by dispatching an action

```
// Do this on, say, a button click or whatever
this.handleClick = evt =>
    store.dispatch({
      type: FETCH_PERSON,
      personId
    });
```

- Note the absence of any response or anything. It is just set it and forget it. We rely on the middleware to dispatch a synchronous SET_PERSON action and the subscription to refresh the UI when an Ajax response is received.

Hands on fetching Ajax data