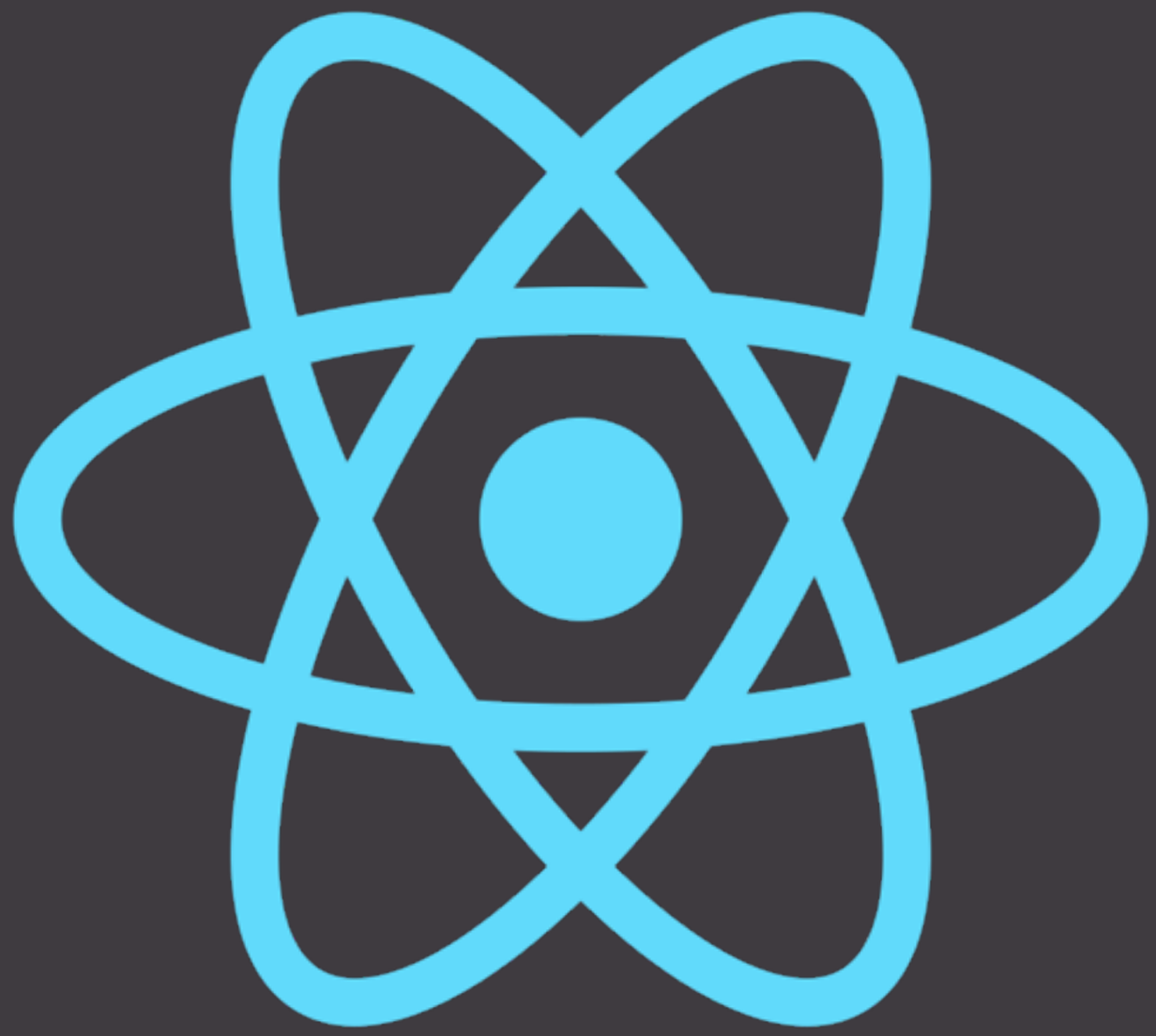
Flux: Those who forget the past...

@jeremymorrell

Flux: Those who forget the past...

...are doomed to debug it

@jeremymorrell



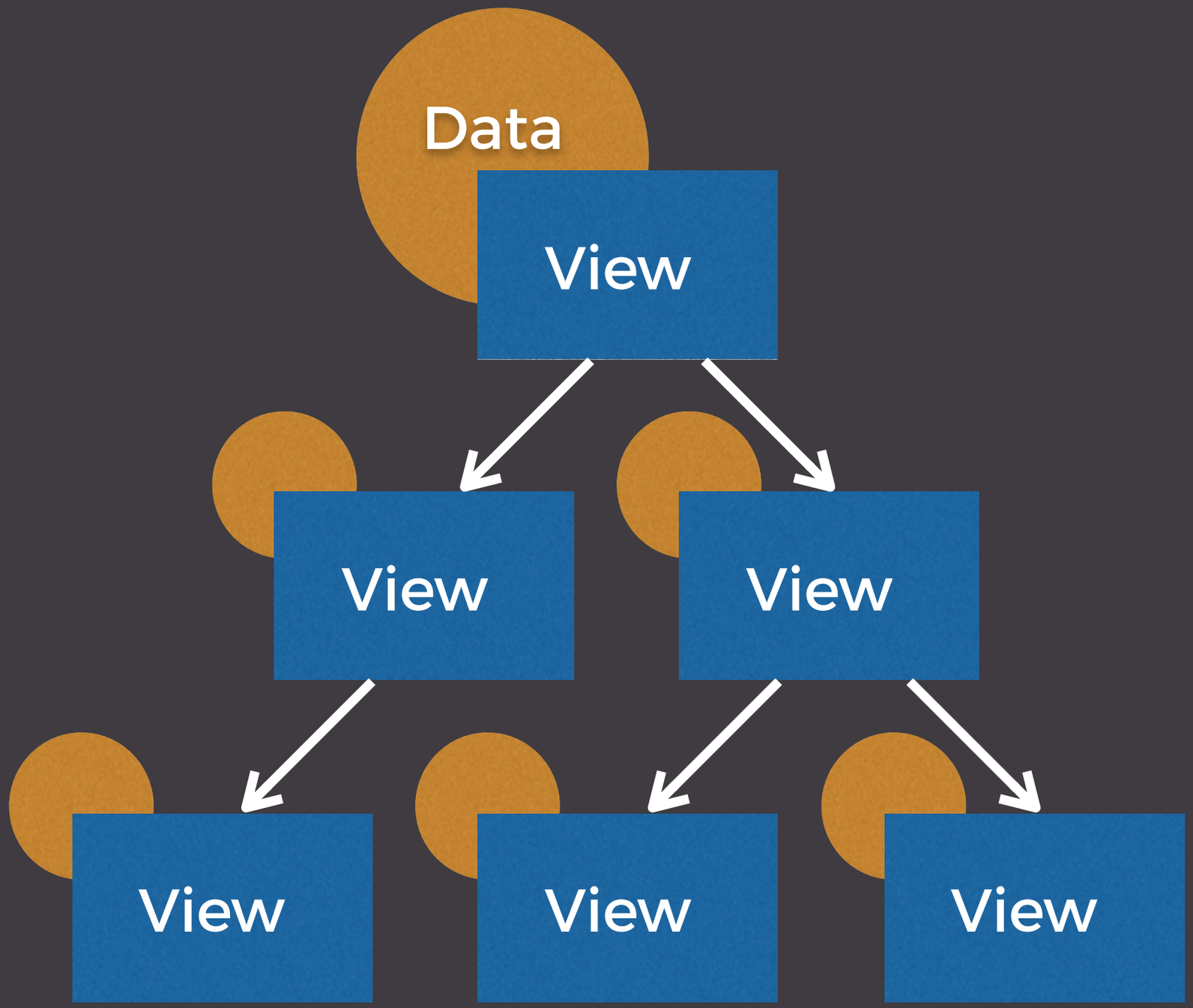
This talk isn’t about React specifically
But we do need to understand one thing about it

react(**data**) → UI

what's special about react is the way I can think about my views
your application data is passed in at the root
and the UI produced is a function of that data
with the same data as input, it will always produce the same output
when the data changes I just re-run the function and React will update the UI

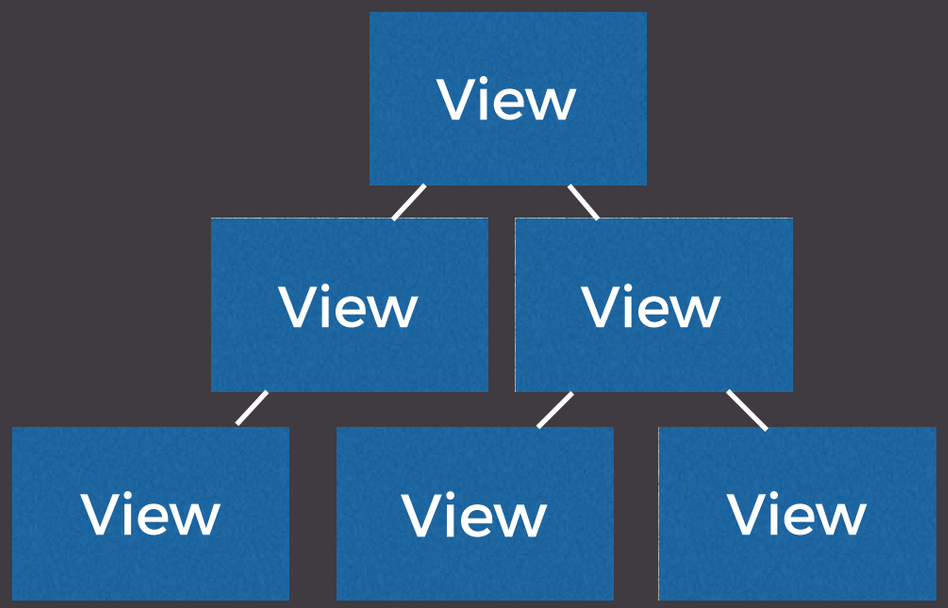
```
function render(data) {
  return `
    <h1>
      Hola, ${data.name}!
    </h1>
  `;
}

document.body.innerHTML = render({
  name: "JSConf UY"
});
```

For the purposes of this talk you can think about React as one giant template function
Every time the data changes, we re-render the template,
and just blow the old view away
This makes it much easier to reason about what's happening in our view layer

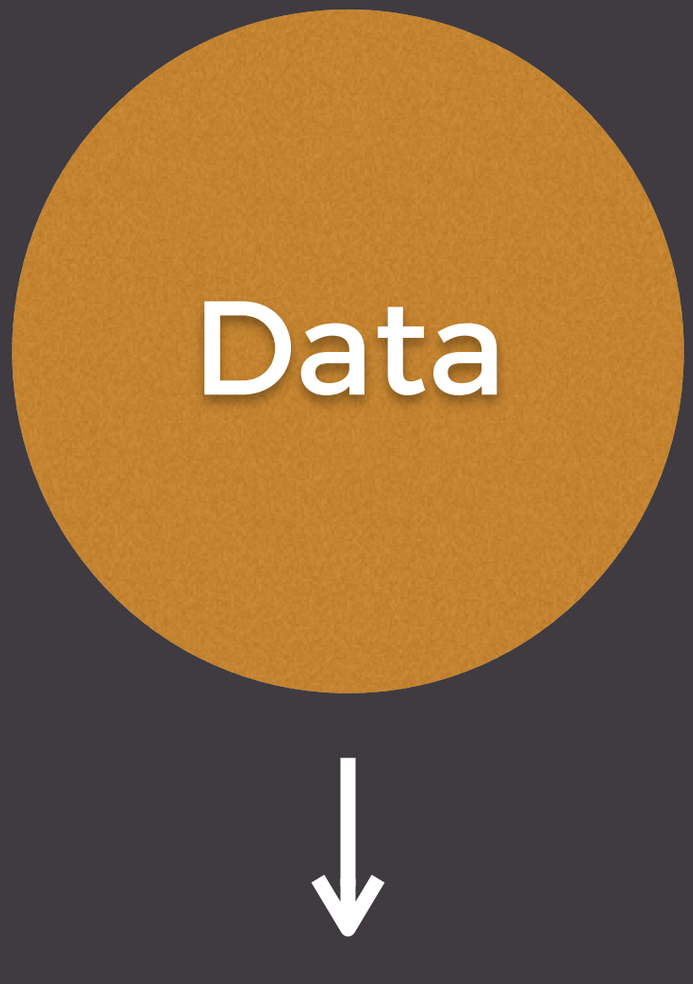


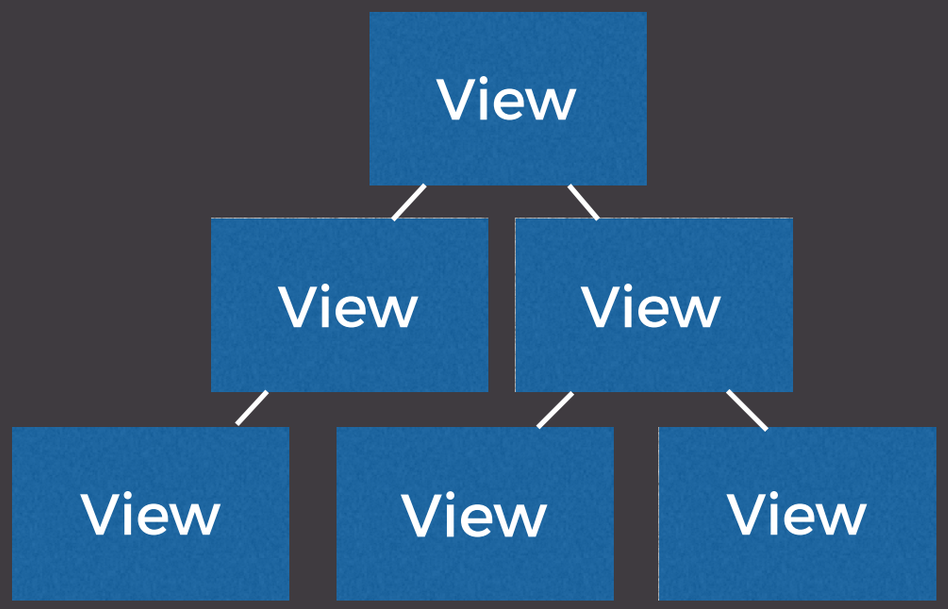
- But then you create a new problem
- Previously our apps looked something like this
- Views living right next to the data they needed

Data



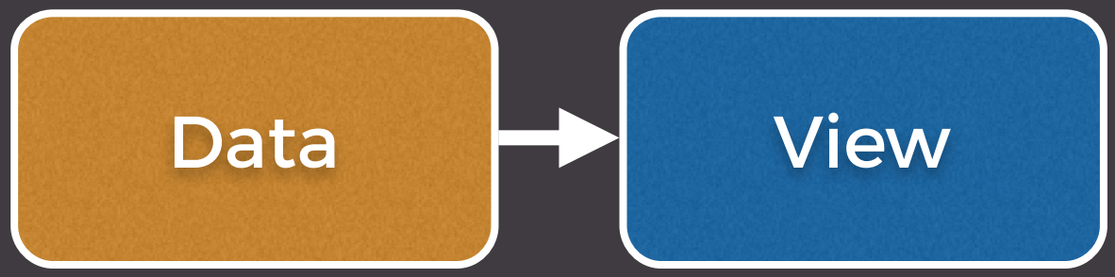
- But with React your data lives outside of this view hierarchy
- I can now easily reason about my view layer
- How can I structure my application so that it’s easy to reason about my data?



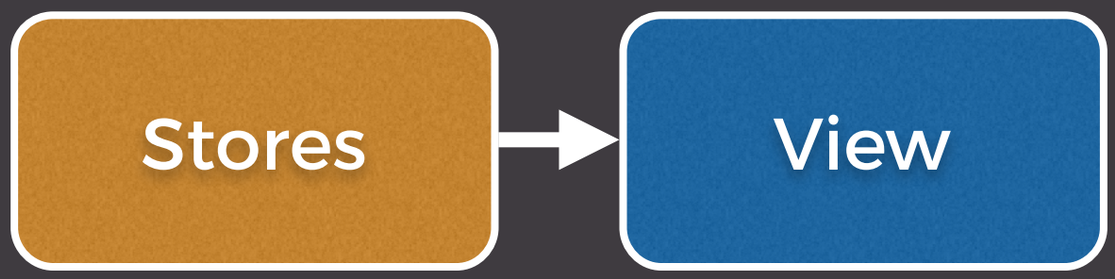


- But with React your data lives outside of this view hierarchy
- I can now easily reason about my view layer
- How can I structure my application so that it’s easy to reason about my data?

The solution that's been working for us as we develop our large applications is an architecture that we call Flux



Our ideal view of the world looks like this
Data completely separate from the view
We know that when the data changes we can re-render our view
So let’s add that functionality into our data layer and change the name

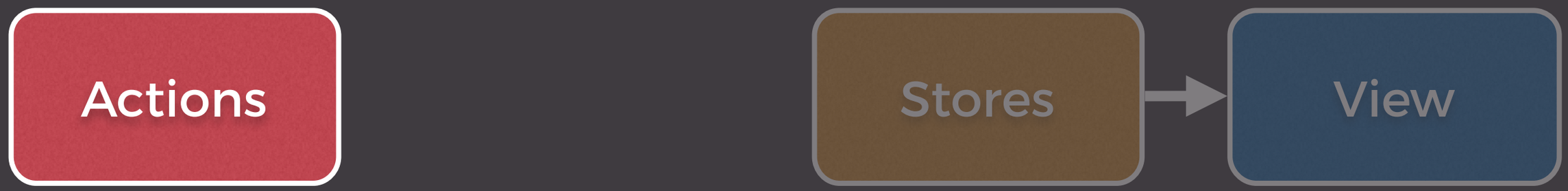


Stores hold data, and signal when something has changed

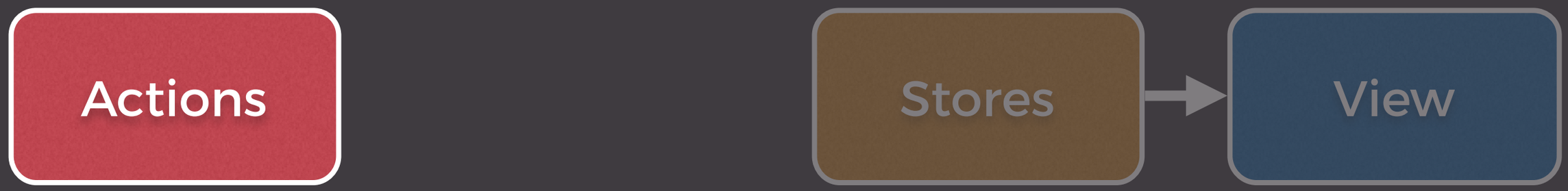
Views subscribe to the stores that contain the data that it needs

Data updates, re-render the view, we know this stuff

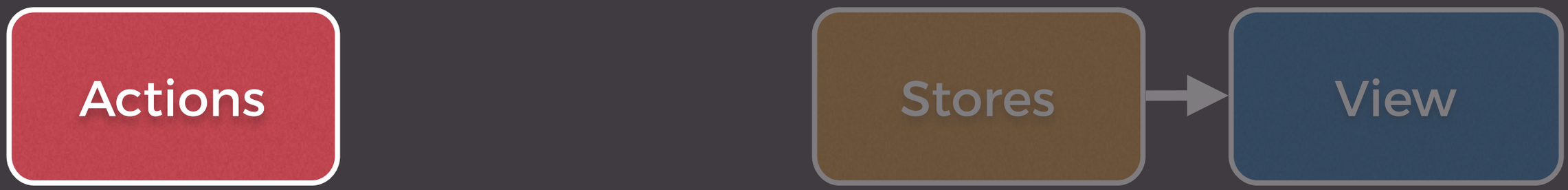
This tends to be pretty intuitive for frontend developers

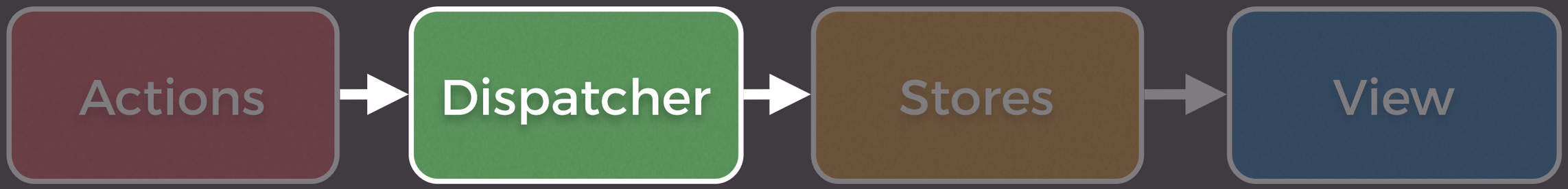


Flux introduces a concept called Actions
less intuitive for most of us
NOT DOM EVENTS



Actions are loosely defined as “things that happen in your app”
Examples:
liking a post on newsfeed,
leaving a comment,
requesting search results,
changing your password


Actions
Stores
View



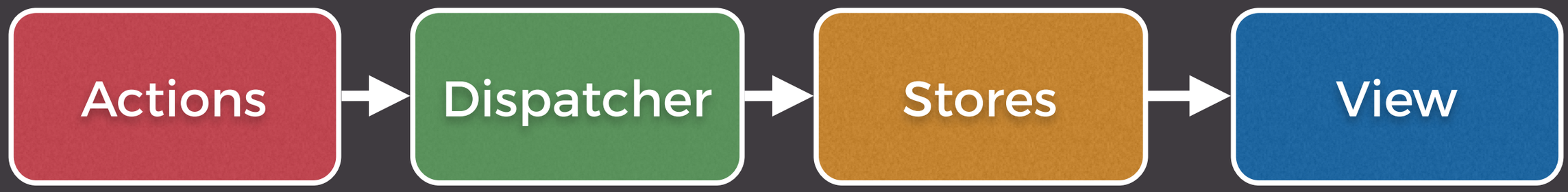
The dispatcher trips people up some times

receives actions and passes them to every registered store

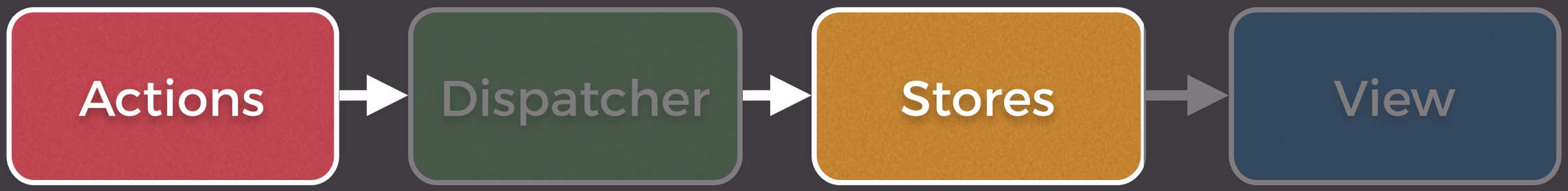
* Every action passes through the dispatcher

* Every action is passed through every store

It handles dependencies between stores, but today we don’t have to think about that



So I click on a button,
that generates an action
the dispatcher passes that to each store
stores update themselves in response
view re-renders



For this talk we can basically ignore the dispatcher and view layers
I want to focus on the interaction between actions and stores
Still abstract, let's get a concrete example

Bank Account

Bank Account

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250
		$250

With every transaction, we update another value called balance

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250
		$250

These transactions are how we're interacting with our bank. They modify the state our of account.

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250

NOTE: If we perform the same transactions, same order, these results will be the same
The balance is derived data
In flux terms, the transactions on the left are our actions and the balance on the right is a value that we would track in a store

$250

Actions should be like newspapers

“Actions should be like newspapers, reporting on something that has happened in the world.”
- Bill Fisher @ Fluent
They might look something like:

```
{
  type: Actions.WITHDREW_FROM_ACCOUNT,
  data: {
    accountID: 7,
    amount: 50,
    date: 1429468551933,
    location: { ... }
  }
}
```

Two fields

type

details about that action

```
{
  type: Actions.DEPOSITED_INTO_ACCOUNT,
  data: {
    accountID: 7,
    amount: 500,
    date: 1429468551933,
    location: { ... }
  }
}
```

note past tense for the action name.
“Something that happened”
So what would our store code look like?

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDREW_FROM_ACCOUNT:
      balance -= action.data.amount;
      break;
    case Actions.DEPOSITED_INTO_ACCOUNT:
      balance += action.data.amount;
      break;
    ...
  }
}
```

This would be inside a store that tracks account balance

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDREW_FROM_ACCOUNT:
      balance -= action.data.amount;
      break;
    case Actions.DEPOSITED_INTO_ACCOUNT:
      balance += action.data.amount;
      break;
    ...
  }
}
```

The dispatcher makes sure that every action in the app invokes onDispatch on every store

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDREW_FROM_ACCOUNT:
      balance -= action.data.amount;
      break;
    case Actions.DEPOSITED_INTO_ACCOUNT:
      balance += action.data.amount;
      break;
    ...
  }
}
```

When we withdraw money, we decrement

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDREW_FROM_ACCOUNT:
      balance -= action.data.amount;
      break;
    case Actions.DEPOSITED_INTO_ACCOUNT:
      balance += action.data.amount;
      break;
    ...
  }
}
```

And when we deposit money we increment

After this method, the store emits a change, and the view re-renders

```
let balance = 0;

function onDispatch(action) {
  ...
}
```

```
function getBalance() {
  return balance;
}
```

We also need to get the data out

The view layer would call getBalance when it renders

Stores are not observable objects

At least not in the way we generally think of them
It's tempting to think of stores as just models that live outside of your view hierarchy
but stores do not behave like the traditional models that we think of (O.o)
How so?

```
model.balance
```

```
store.getBalance()
```

We have getters, true

```
Object.observe(model, changes => {
  // update the view
});
```

```
store.subscribe(() => {
  // re-render the app
});
```

And we can subscribe to changes, so that’s not too different

```
model.balance = oneMillionDollars;
// ... ?
```

But there’s no equivalent for a setter
You can’t call up your bank and tell them that your balance is now one million dollars
Stores update in response to actions, but there’s no way to update just one value,
or just one store
ACTIONS become the ONLY WAY to MODIFY our state
There’s an important result of this fact

Stores are a function of the actions fired on them

```
f(state, [...actions]) → newState
```

Given a set state, the transition to another state given a set of actions is deterministic.
If I fire the same sequence of actions in my app, I will end up with the exact same state
Source of truth is actually the stream of events
Stores are a “cache”
This is a reduce, the stores are accumulators

But bank transactions are async...

We need to take care to not accidentally mutate state without an action though

My previous example wasn’t complete.
We have to request a transaction

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDRAWAL_REQUESTED:
      requestWithdrawal(
        action.data.accountId,
        action.data.amount
      ).then(
        res => balance -= res.amount;
      );
      break;
    ...
  }
}
```

A first attempt might look like this

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDRAWAL_REQUESTED:
      requestWithdrawal(
        action.data.accountId,
        action.data.amount
      ).then(
        res => balance -= res.amount;
      );
      break;
    ...
  }
}
```

New Action

Make a request, and when the response comes back, update the value

The store updates with the correct value

and the view will render correctly

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDRAWAL_REQUESTED:
      requestWithdrawal(
        action.data.accountId,
        action.data.amount
      ).then(
        res => balance -= res.amount;
      );
      break;
    ...
  }
}
```

But now there is a mutation of our data that's not in this stream of actions

If we re-apply our actions we end up in a different state

If something else needed to know about the withdrawal, now it can't

Harder to reason about our app

Async operations need to fire actions

The way around this is to always fire actions at the end of an async req

```
function requestWithdrawal(account, amount) {
  requestWithdrawal(account, amount)
    .done(
      res => dispatch({
        type: Actions.WITHDREW_FROM_ACCOUNT,
        data: { ... }
      }),
      err => dispatch({
        type: Actions.WITHDRAWAL_FAILED,
        data: { ... }
      });
    );
}
```

You might do it this way, outside of the store

```
function requestWithdrawal(account, amount) {
  requestWithdrawal(account, amount)
    .done(
      res => dispatch({
        type: Actions.WITHDREW_FROM_ACCOUNT,
        data: { ... }
      }),
      err => dispatch({
        type: Actions.WITHDRAWAL_FAILED,
        data: { ... }
      });
    );
}
```

If the request succeeds, we fire the action from earlier

```
function requestWithdrawal(account, amount) {
  requestWithdrawal(account, amount)
    .done(
      res => dispatch({
        type: Actions.WITHDREW_FROM_ACCOUNT,
        data: { ... }
      }),
      err => dispatch({
        type: Actions.WITHDRAWAL_FAILED,
        data: { ... }
      });
    );
}
```

If it fails, something else will want to know

Stores are a way of asking a question

Stores are a convenience

Given list of all transactions that I've ever made, can I afford to buy lunch?

This is what we used to have to do balancing a checkbook (ask your parents)

We decide what stores to have based on what questions we want to ask

Let’s ask a new question

Account balance is probably not the only question we’ll need to ask of this data

In large systems many different subsystems may need to know about what’s happening

Because every action is passed to every store we create more stores

Your withdrawal has failed

Let’s ask a new question

So your designer wants the app to notify the user when a withdrawal has failed

```
{
  type: Actions.SHOW_NOTIFICATION,
  data: {
    message: "Your withdrawal has failed",
    ...
  }
}
```

At first we might consider doing this

```
{
  type: Actions.SHOW_NOTIFICATION,
  data: {
    message: "Your withdrawal has failed",
    ...
  }
}
```

But this isn’t a good action

SHOW_NOTIFICATION is a command, not “something that happened”

Now, I have to sprinkle this action all around the application

We’re trying to get around the lack of a setter and talk to a particular store

Actions are not elaborate setters

- Actions are like newspapers

want to implement like this

```
let messages = [];

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDRAWAL_FAILED:
      messages.push("Your withdrawal has failed");
      break;
    case Actions.NOTIFICATION_DISMISSED:
      messages = [];
      break;
    ...
  }
}
```

Our view layer simply renders a notification for each value in messages

Empty -> no notification

When a withdrawal fails, messages now has a value

view re-renders

Your withdrawal has failed

```
let messages = [];

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDRAWAL_FAILED:
      messages.push("Your withdrawal has failed");
      break;
    case Actions.NOTIFICATION_DISMISSED:
      messages = [];
      break;
    ...
  }
}
```

and we have a notification,

when the user interacts with the view or a time limit is reached

the dismiss action is fired and it's not longer rendered

Maintain separation of concerns.

The code firing the action has no idea the notification system is listening.

Actions are the change in your app

Actions represent mutations of your app state

Explicit, easy to find the places that could trigger a particular action, I can search for it

```
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
Actions.WITHDRAWAL_REQUESTED
Actions.WITHDRAWAL_FAILED
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
Actions.WITHDRAWAL_REQUESTED
Actions.WITHDRAWAL_FAILED
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
```

our app looks like this when it's running

every action passes through the dispatcher

can log them all out

I use this at work to understand new sections of the UI that I haven't worked on before

```
let balance = 0;

function onDispatch(action) {
  switch (action.type) {
    case Actions.WITHDREW_FROM_ACCOUNT:
      balance -= action.data.amount;
      break;
    case Actions.DEPOSITED_INTO_ACCOUNT:
      balance += action.data.amount;
      break;
    ...
  }
}
```

When looking at a store the actions that can modify it are explicit

This is the exhaustive list

This helps narrow the scope of what I need to understand in a large system, especially if we keep the stores small

Make changes with confidence

This allows us to keep moving fast, even as our systems get large

Those who forget the past...

So here’s where I try to justify the title

Account Balance: -$10

You open your bank account and see that you now have -10 dollars as your balance
WHAT HAPPENED?
A user sends you a screenshot of your app in a weird state: I HAVE A BUG
This is the same situation
Repro please

Bank Account

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250
		$250

If this is our bank account we have a history to look at
If this is our app, we are missing most of this data

Bank Account

-$10

We’re trying to debug using only the final value and our knowledge of the system

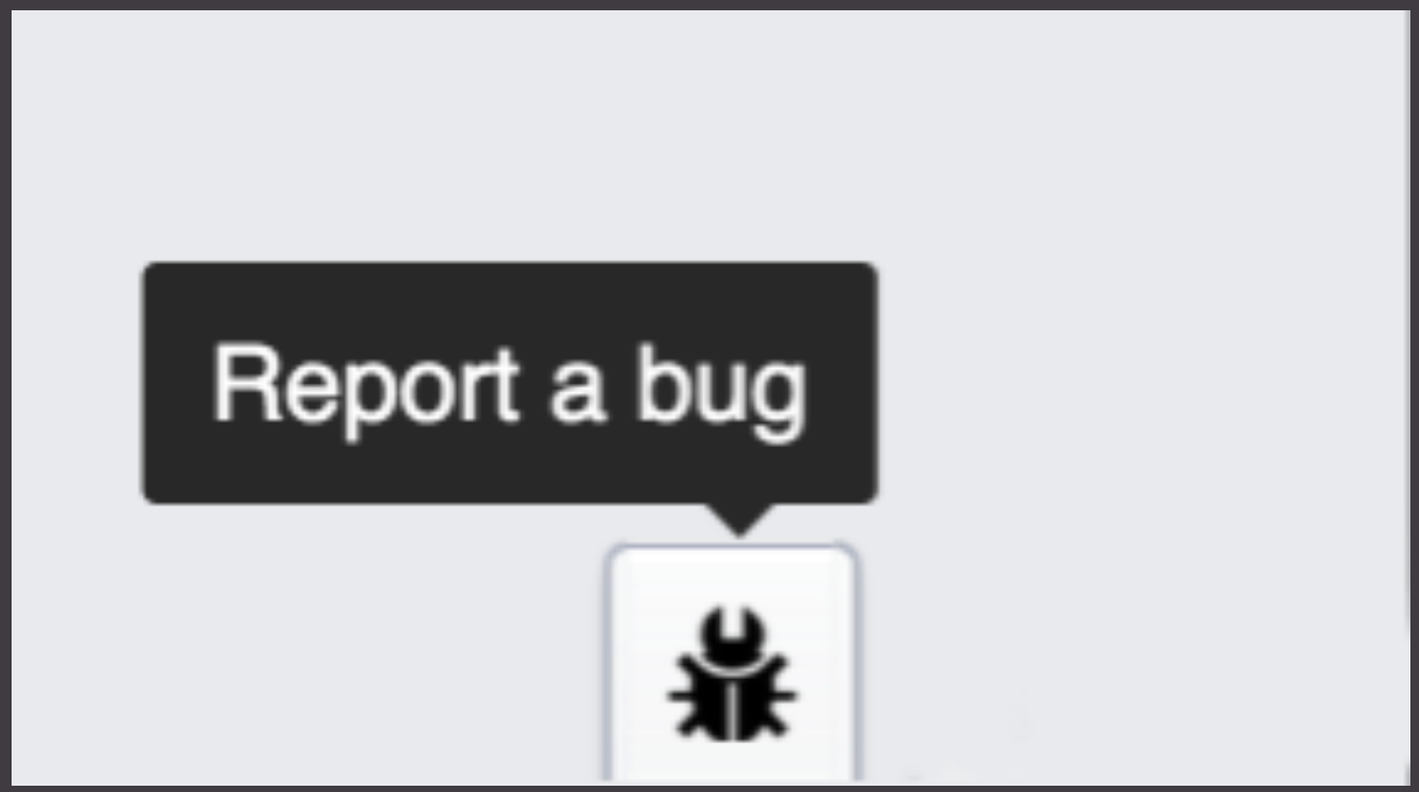
Bank Account

Transaction	Amount
Create Account	$0
Deposit	$200
Withdrawal	($50)
Withdrawal	($160)
	-$10

This is what we really want

```
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
Actions.WITHDRAWAL_REQUESTED
Actions.WITHDRAWAL_FAILED
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
Actions.WITHDRAWAL_REQUESTED
Actions.WITHDRAWAL_FAILED
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
Actions.USER_CHANGED_PASSWORD
Actions.USER_UPDATED_PHONE_NUMBER
Actions.WITHDRAWAL_REQUESTED
Actions.WITHDRAWAL_FAILED
Actions.DEPOSIT_REQUESTED
Actions.DEPOSITED_INTO_ACCOUNT
```

But we have exactly that! We just need to save them off



At Facebook we did that for one of our flux apps
When an employee filed a bug, they could choose to send off all of the actions that happened that session

```
f(state, [...actions]) → newState
```

Because of this property, not only can I see how they got there
I can literally re-play their actions and see exactly what they saw
every intermediate step

Those who forget the past are doomed to debug it

But we can only do this because we make our mutations explicit and keep a history
So the next time someone sends you a screenshot of your app in a weird state…

¡Gracias!