# Trickline Introduction

@paulcbetts, @charlie

# OK So, React?

We've decided our glorious future will be React-based 🎉

# Sooo, time to...

# Componentize Everything!

Converting jQuery + Handlebars views to React will be a challenge

But it might not even be the Hard Part™

We can no longer have all users and channels in memory.

— *Abraham Lincoln*

# Wait, What?

The hard part is converting components from a data model that assumes we have everything, to one where you might not.

```
// Easy!
TS.model.channels[0].name
>>> devel-react

// We'll never be able to debug this as this gets bigger
TS.models.getChannel(0).then(x => dieInside());
```

The old model is *straightforward*. New developers can program against it fairly easily. We need a model that retains that ease for Most People.

# Managing data changing over time is What Slack Does

*Almost every object* in Slack can be sourced from:

- HTTPS method call
- RTM event
- Our local cache

We should get Really Really Good at these kinds of objects and solve their problems in a generalized way.

# Redux is Great...

Redux is great – we've used it in the Desktop app and it provides some 🆒 benefits:

- Your whole app state in one tree: *Introspectible!*
- Actions for everything: *Debuggable!*
- Reducers for everything: *Functionally elegant!*

# But it is Insufficient 😔

- Redux doesn't solve the core issue: partial models.
- Having your app state in one object might cause us to run *towards* this problem rather than away from it.

# Trickline

Demo

# Our Goals

- The amount of memory we use is proportional to the number of things on screen.
  - Nothing on screen? No(*) memory usage.
- Writing views should be super easy, and reading the implementation of views should be a joy.
  - Polluting every view with fetching and retries and caching will make every view a disaster.

# Our Goals

- The way that data gets *into* Slack should be completely unrelated to the way that devs *access* data.
  - Views don't really care where data comes from, they just say what they want.
- Electron apps don't have to use 2GB of memory, and can be *really fast*. Prove it.

Wait, are you building a Slack client tho???

# Wait, are you building a Slack client tho???

- Our goal is to build *pieces* that we can end up using anywhere.
- While these pieces go really well *together*, you're not opting-in to a Capital-F *Framework*. Each piece works standalone!
- The UI you saw is basically the world's most involved integration test – it's a way to exercise the developer experience.

# Updatable: A Lazy Promise

- Updatables are like a Promise that doesn't necessarily do its work immediately
- You can always get the current value of an Updatable, though it may be null

```
// Okay
console.log(generalChannel.value.name);
>>> announcements-general

// Better
generalChannel.get().then(value => console.log(value.name));
>>> announcements-general
```

# Updatable: A Lazy Promise

- Updatables can change *more than once* (as opposed to a Promise, that only `thens` once).
- Updatables let you listen for when an object changes:

```
let currentName = generalChannel.value.name;
generalChannel.subscribe(channel => {
  if (currentName === channel.name) return;

  currentName = channel.name;
  console.log(`The new general channel name is ${currentName}!`);
});
```

# Updatables know how to get the latest version of themselves

```
// Fetch the very latest channel name
generalChannel.invalidate();
generalChannel.get().then(value => console.log(value.name));
```

# Updatables know how to get the latest version of themselves

- Invalidate can also be used to handle models that are incomplete:

```
// If we don't know enough about a channel (because, say, we got it from
// `users.counts`), fill it up with another API call
channel = generalChannel.value;

if (!channel.topic) {
  generalChannel.invalidate();
  channel = await generalChannel.get();
}

console.log(channel.topic);
```

# SparseMap - like an on-demand Map

- Knows how to create Updatables for a certain "class" of thing (users, channels)

```
// Always returns an Updatable of _something_
const myChannel = channelList.listen('C032AB90');

myChannel.get().then(channel => console.log(channel.name));
>>> "random"
```

# SparseMap - like an on-demand Map

- If we've seen that data recently, 🎉 - if not, you might make a network request, or receive stale data.
- In the future, Updatables will be able to tell you when they've last updated, or you can request that certain fields be present.

# A ViewModel is a Model Of A View

- Testing React components using tree diffs requires constant maintenance
- Testing against Plain Ol' Objects is easier
- Our ViewModels have the unique property that, you can listen to changes on them

```
myChannel.changed
  .subscribe(x => console.log(`${x.property} is now ${x.value}`));
```

# Let's make that a bit easier

```
when(myChannel, x => x.unreadCount)
  .subscribe(x => console.log(`Unread count is now ${x}`));

myChannel.unreadCount = 5;
>>> Unread count is now 5
```

# Models and Updatables 2Gether In Love

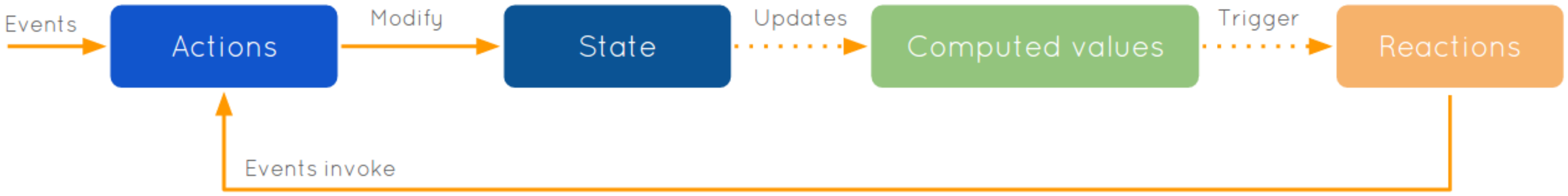
- ViewModels make it easy to turn Updatables into Properties.
- Conveniently, this means we don't really have to think about Subscribing.

# Models and Updatables 2Gether In Love

```
export class UserViewModel extends Model {
  @fromObservable model: User;
  @fromObservable displayName: string;
  @fromObservable profileImage: string;

  // This User is just an object from `users.info` or an RTM event
  constructor(Updatable<User> model, id: string, api: Api) {
    super();

    // Always keep a current copy of the User
    model.toProperty(this, 'model');

    // The DisplayName updates whenever the Model changes
    when(this, x => x.model)
      .map(user => user ? user.real_name || user.name : '')
      .toProperty(this, 'displayName');

    // The ProfileImage updates whenever the Model changes, and if it's
    // initially empty, give them a default
    when(this, x => x.model)
      .map(user => {
        if (!user) return defaultAvatar;
        return user.profile.image_48;
      })
      .toProperty(this, 'profileImage');
  }
}
```

# Is this *Not Invented Here* syndrome??? 🤔

- Most of these concepts aren't new; much of it feels like 🧱 **MobX**
- MobX is a popular Redux alternative based on **`Observables`**
- We could ~~pilfer~~ reuse MobX implementations if we prefer their API




*Events invoke actions. Actions are the only thing that modify state and may have other side effects.*

*State is observable and minimally defined. Should not contain redundant or derivable data. Can be a graph, contain classes, arrays, refs, etc.*

*Computed values are values that can be derived from the state using a pure function. Will be updated automatically by MobX and optimized away if not in use.*

*Reactions are like computed values and react to state changes. But they produce a side effect instead of a value, like updating the UI.*

```
@action onClick = () => {
  this.props.todo.done = true;
}
```

```
@observable todos = [{
  title: "learn MobX",
  done: false
}]
```

```
@computed get completedTodos() {
  return this.todos.filter(
    todo => todo.done
  )
}
```

```
const Todos = observer({ todos } =>
  <ul>
    todos.map(todo => <TodoView ... />
  </ul>
)
```

Part Two Coming Soon

# THE OUTLINE

**Why even?**

- We can't have everything in memory any more. Full stop.
- The average developer shouldn't have to think about RTM.start vs users.counts to build features
  - When views all know about fetching data, it makes changing the data sources Difficult
- How would we design the Slack data model in 2017, given that Teams aren't going to mean anything, and shared / enterprise