Flux and React

Building Applications with a Unidirectional Data Flow

Flux and React

Building Applications with a Unidirectional Data Flow

Bill Fisher

Jing Chen

Flux and React

https://github.com/facebook/flux

Bill Fisher

Jing Chen

OVERVIEW

Flux and React

Deep dive into stores

Example application

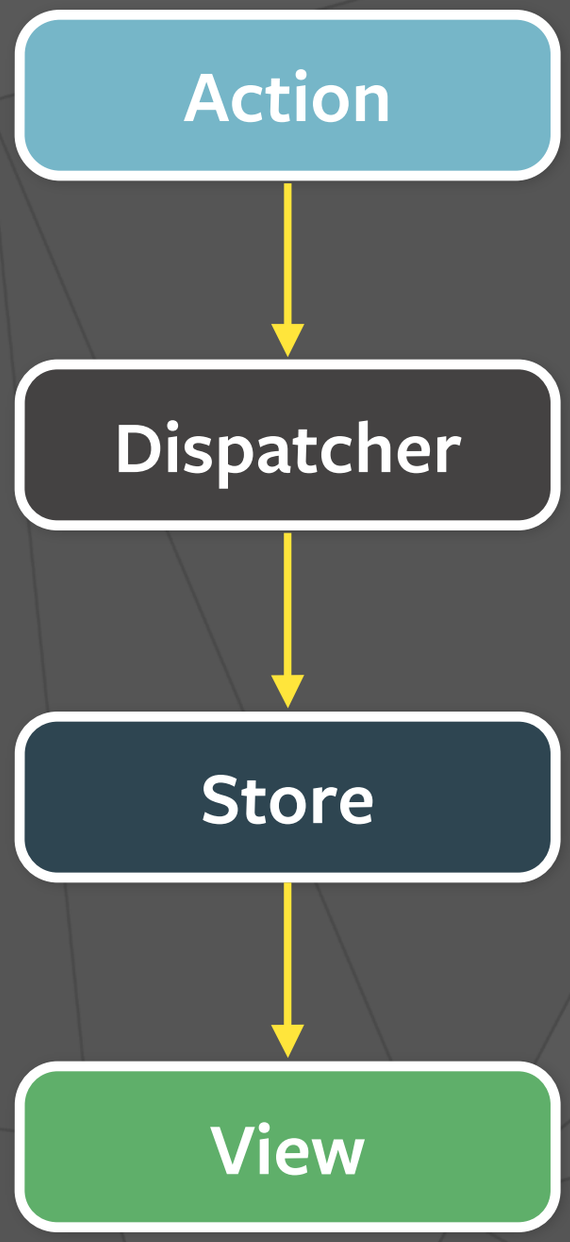
Advanced Concepts

The Future of Flux

FLUX

http://miniurl.com/fluxreact

Unidirectional data flow architecture



We found that by constraining the structure in our apps, we could avoid an explosion in complexity as the app scaled up and increased in features. It takes a little more time to set up, but saves time in maintenance later on.

REACT

JavaScript rendering library

Conceptually re-render on every change

How is this performant? Virtual DOM

Flux and React work really well together, because React allows you to write UI as if every change is handled with a re-render. Under the hood, this is still performant because it uses a virtual DOM - on every update, React diffs the new tree with the previous tree, and applies incremental updates automatically. This feature is useful in conjunction with Flux because updates in Flux usually only specify that something changed, but don't give information on exactly what changed.

This setup allows for a simple mental model, which is what Flux and React are all about.

STORES

We think of Stores as “fat models” - they contain all of the data for a logical domain in a system. In the example chat app, the MessagesStore keeps track of all the messages we know about on the client.

At setup, the store registers a callback with the dispatcher. The only input to the store is through that callback, which is where the store gets all actions. The store can respond to an action if it changes anything in its data, and then emit a change if needed.

One key here is that Stores only have getters as its public interface, there are no setters. Setters would give callers too much control, and make the store vulnerable to ordering of calls that put it in an inconsistent state.

One way to think about a store is as a bag of data and a caretaker for that data. The caretaker watches a stream of actions in the system, and picks the ones that affect its data and updates its data with the metadata data within the action.

Data and application logic for a logical domain

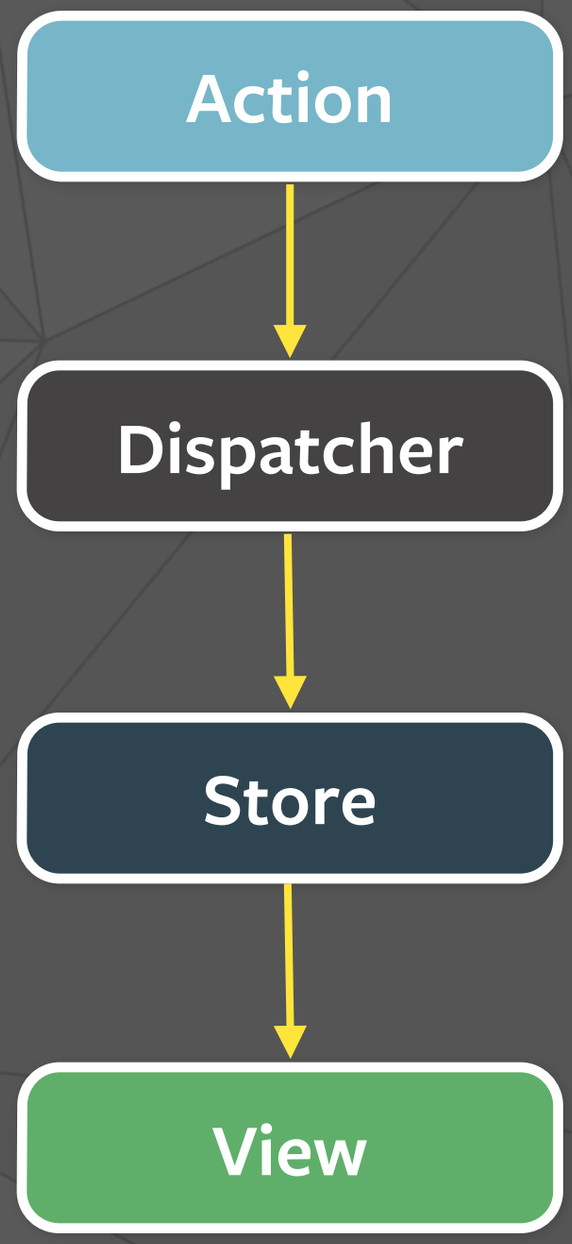
Setup: Register with the dispatcher

Dispatcher calls registered callback

Emits a change event

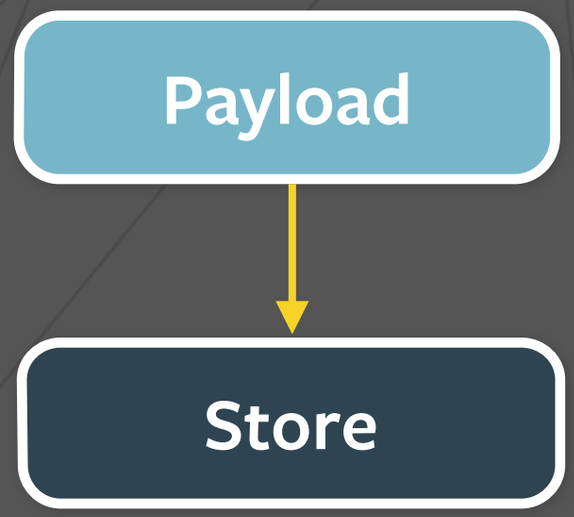
Public interface: Getters, no setters

STORES



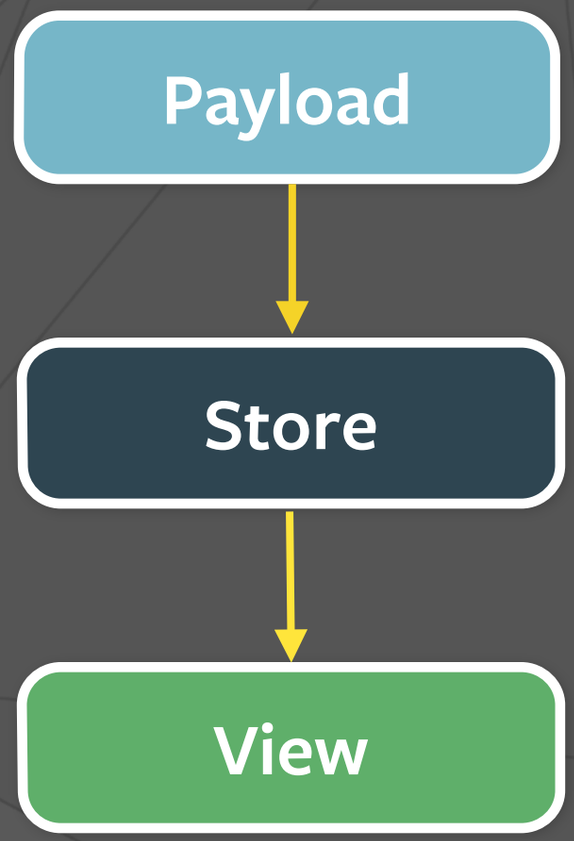
Here’s an overview of the application - the store gets its data from the dispatcher, which is where all updates come in.

STORES



Stores receive new data in the form a payload, an object literal that is the sole argument to the callback that they registered with the dispatcher.

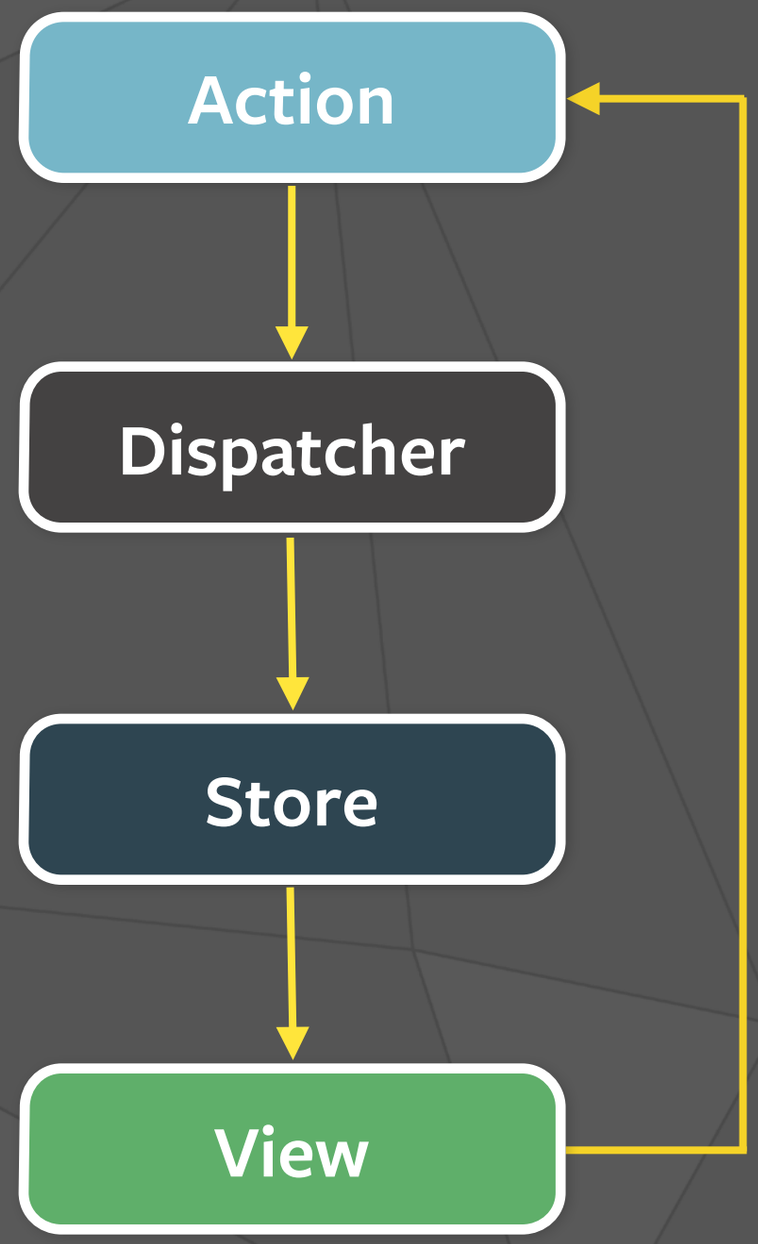
CONTROLLER-VIEWS



The views are React components that listen for change events from the stores. When data within the stores changes, the views pull updated data from the stores through its getters.

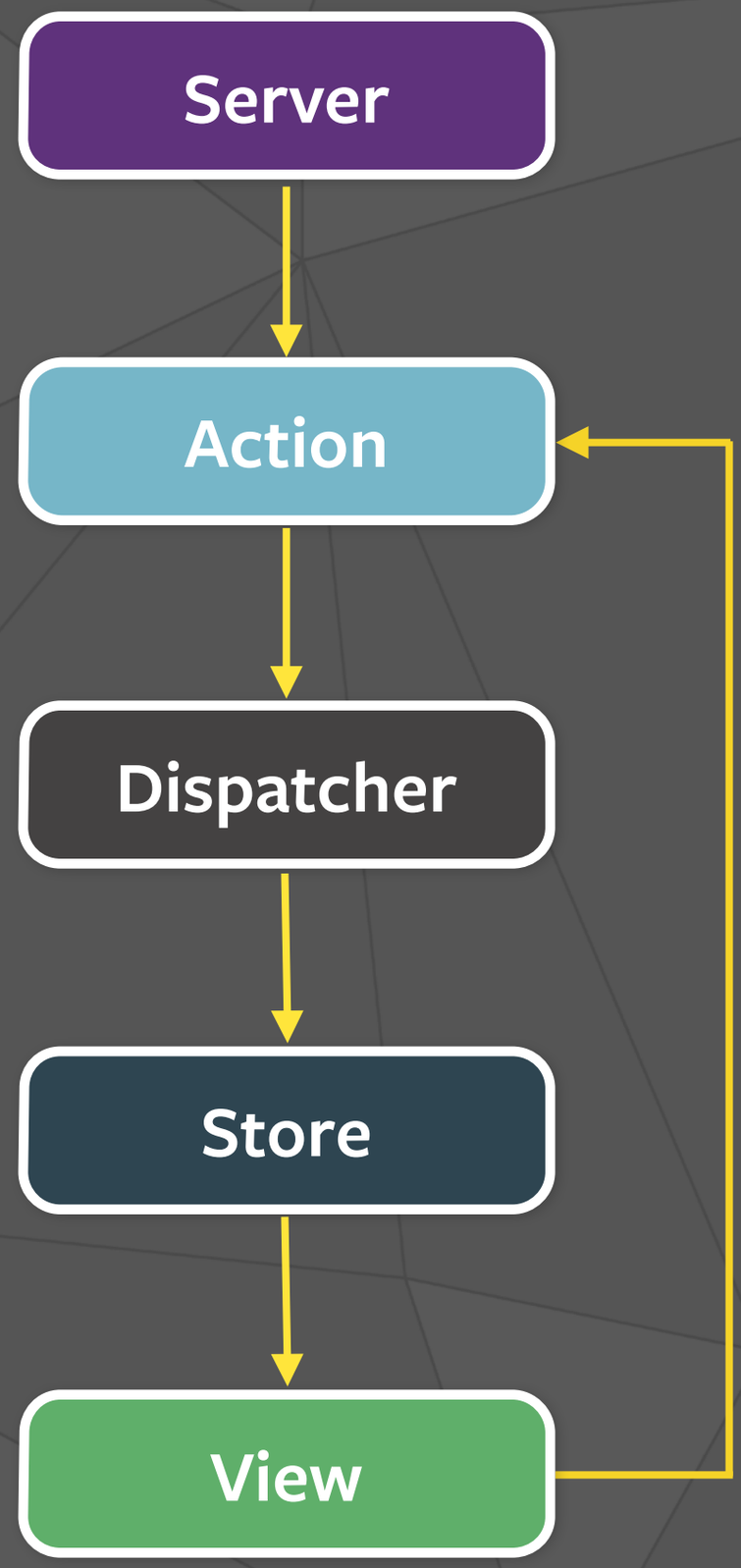
ACTIONS & DISPATCHER

Views handle user input by invoking ActionCreators, which construct actions that represent the intended change. These action payloads are provided to the dispatcher, which is the central hub for these inputs.



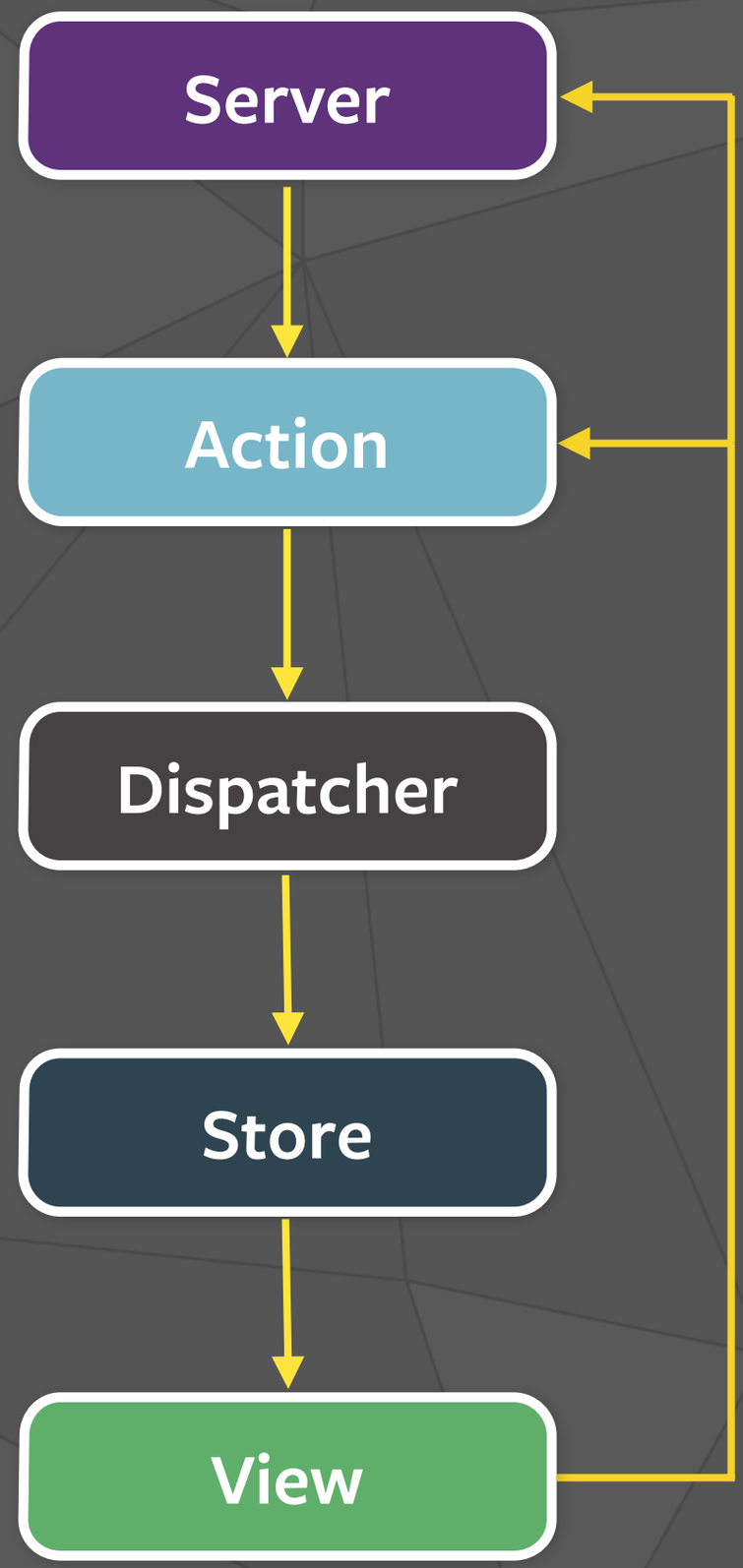
SERVER INTERACTIONS

Web APIs can create actions just like user interactions, and both plug into the same data flow. This includes the initial payload of data - the server provides a special initial data action to the dispatcher and the stores use this to populate its initial data.



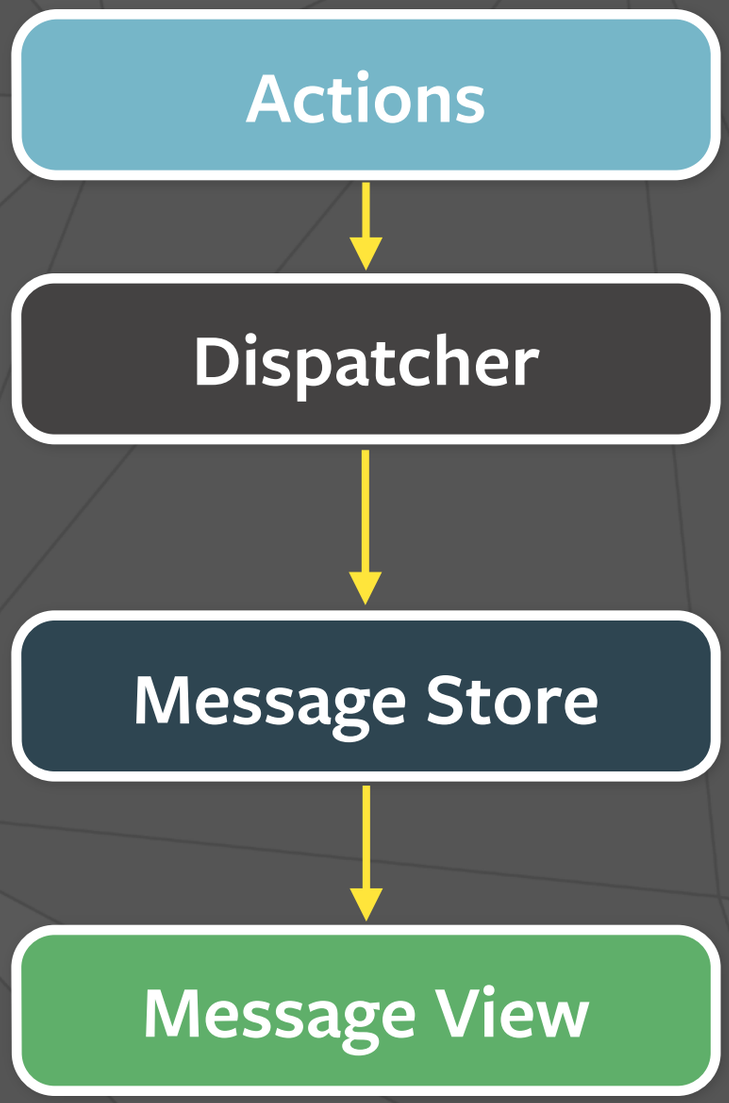
SERVER INTERACTIONS

ActionCreators can also initiate writes to the server, which returns its action response to the dispatcher. The initial action from the ActionCreator is then considered an optimistic action, used to display the expected response from the server before the action is confirmed.



ADDING THREADS

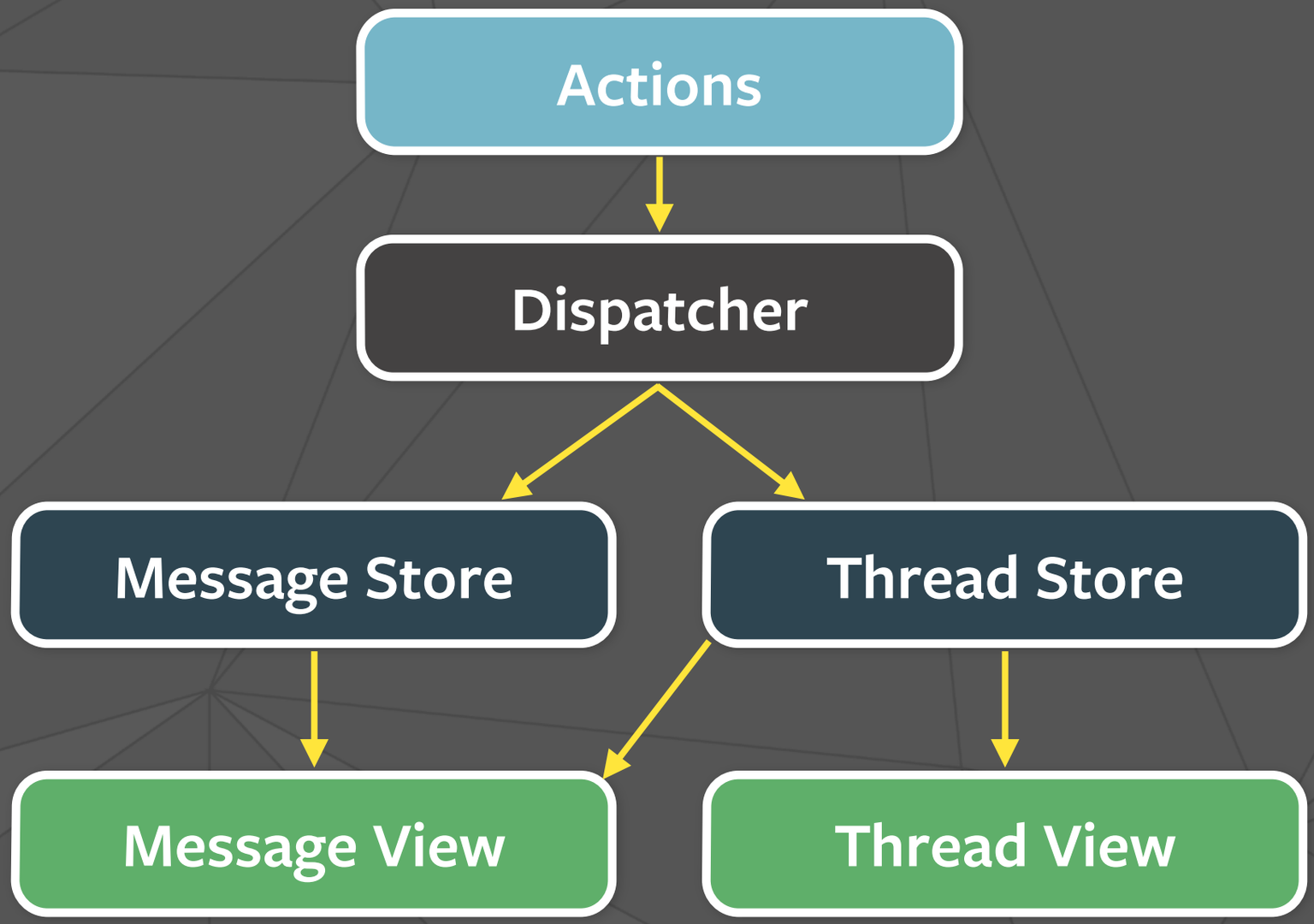
With our chat example, the MessageStore is the one store, and the MessageView listens to changes in the MessageStore



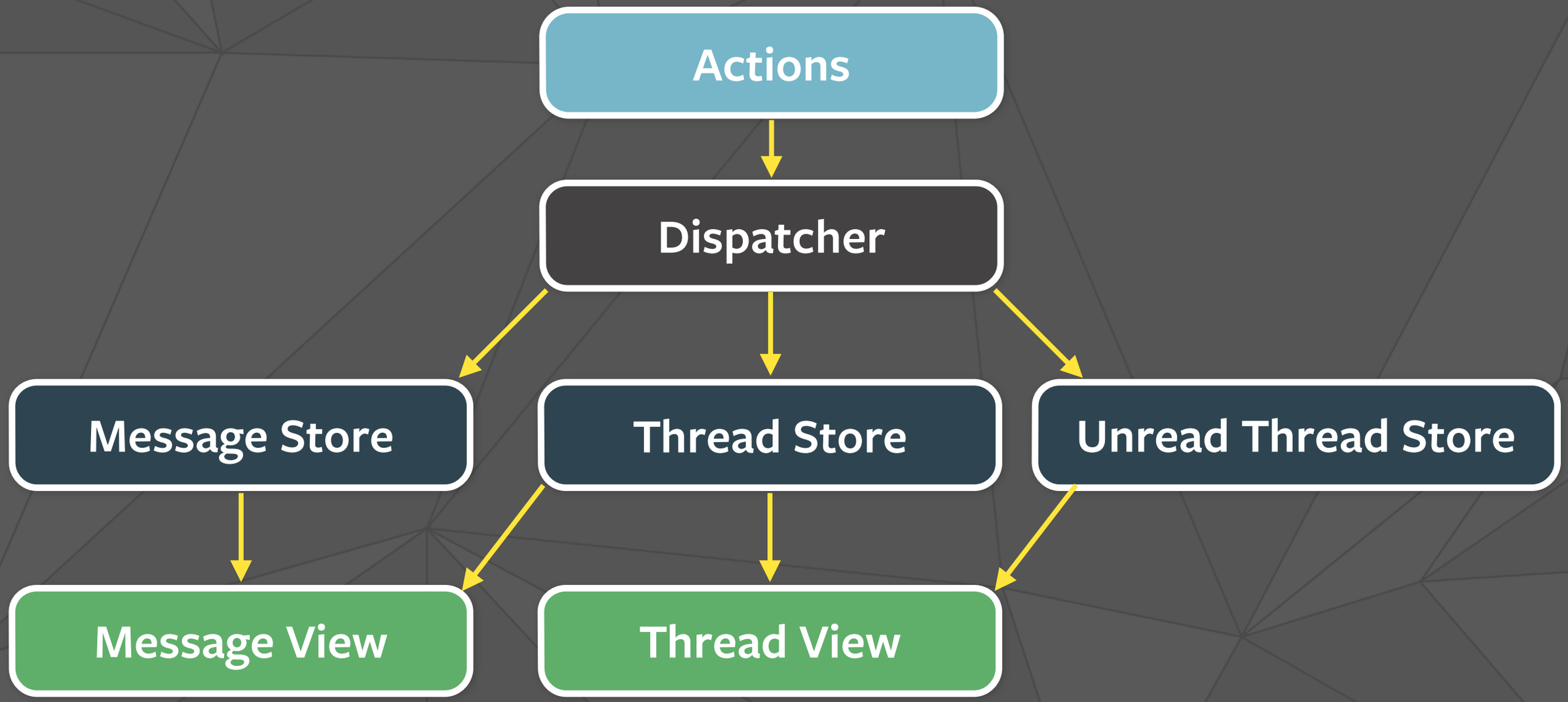
ADDING THREADS

If we also want to track threads, then we can add the ThreadStore, which both the MessageView and the ThreadView pull from.

(Switch to example code)



UNREAD THREADS



MARSHALING CALLBACKS

Dispatcher's waitFor( ) method

Sequenced updates

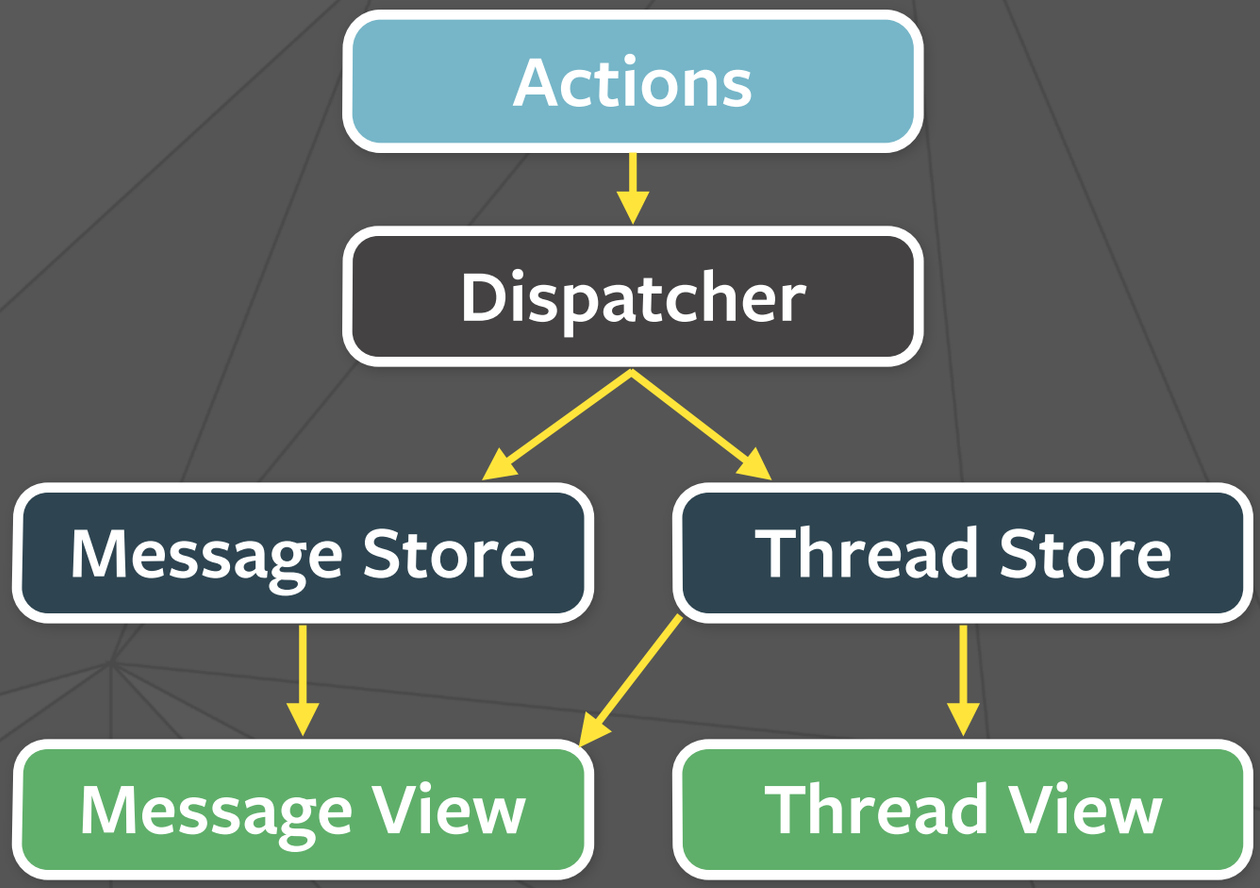
Allows for the creation of a hierarchy of stores

To handle the dependency between UnreadThreadsStore and ThreadStore, we can use Dispatcher's waitFor method to specify that UnreadThreadsStore wants to wait until ThreadStore has processed the action. This allows us to create a hierarchy of stores where data can be derived from data in other stores.

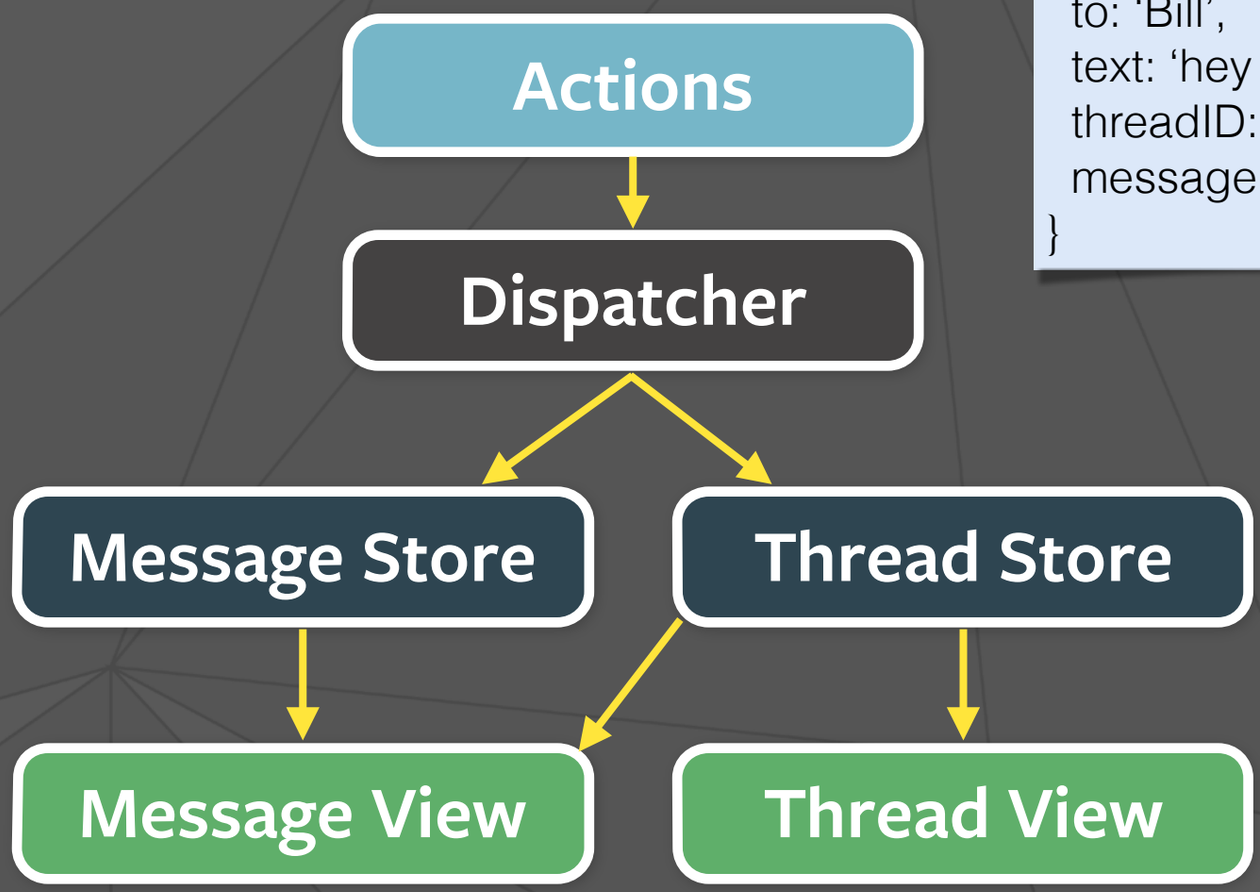
EVOLVING THE APP

As we said at the beginning, Flux makes it a little harder to start building the app, but it should make it easier to add features to the app and maintain it as complexity grows. Let's take a look at a few examples of how features would fit in to our example chat app.

START A NEW CONVERSATION



START A NEW CONVERSATION

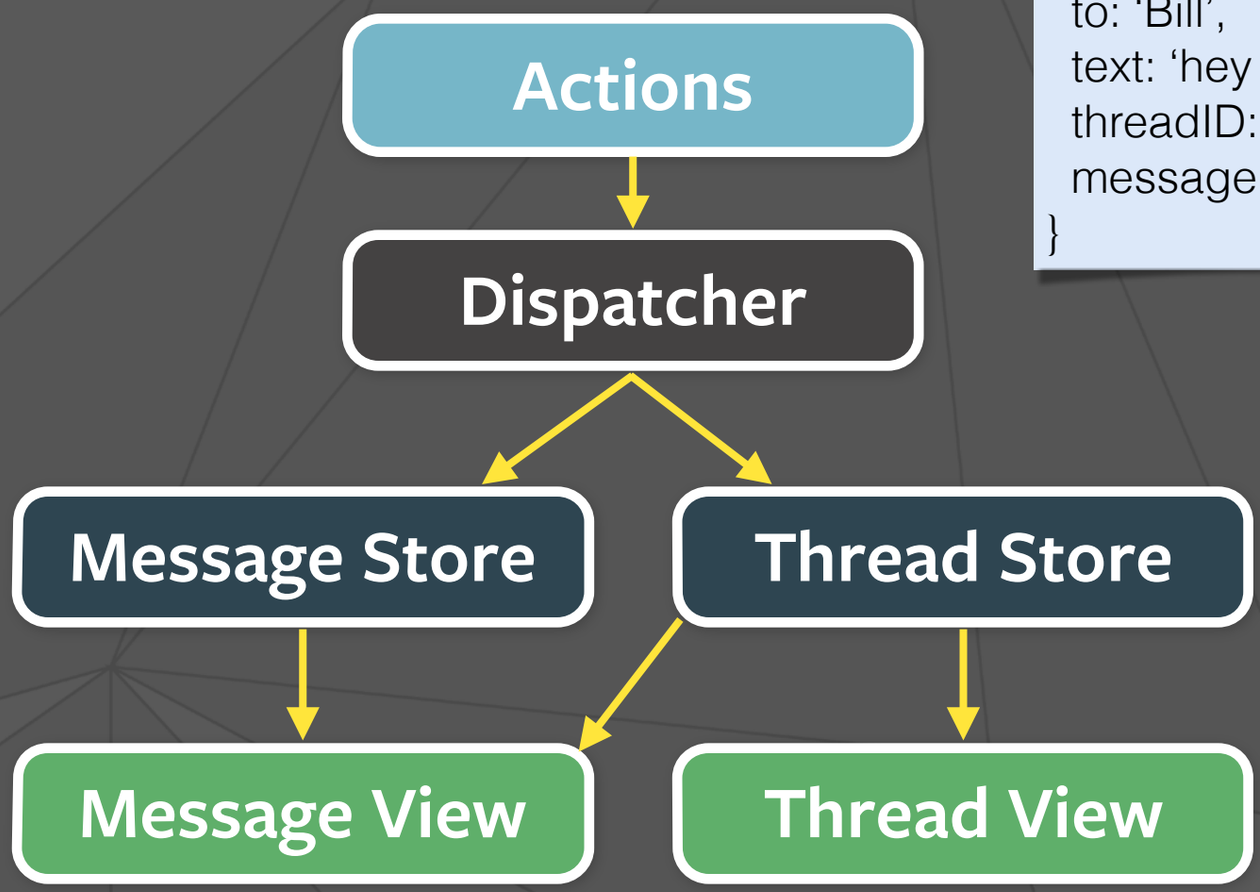



```
action = {
  type: NEW_THREAD,
  to: 'Bill',
  text: 'hey Bill',
  threadID: '5e93696f',
  messageID: '0272fac4',
}
```

Let's say we want to start a new conversation from the viewer, Jing, with Bill. We'll use first names as IDs here for simplicity :)

The action would consist of a series of metadata about the new thread and first message on that thread.

START A NEW CONVERSATION



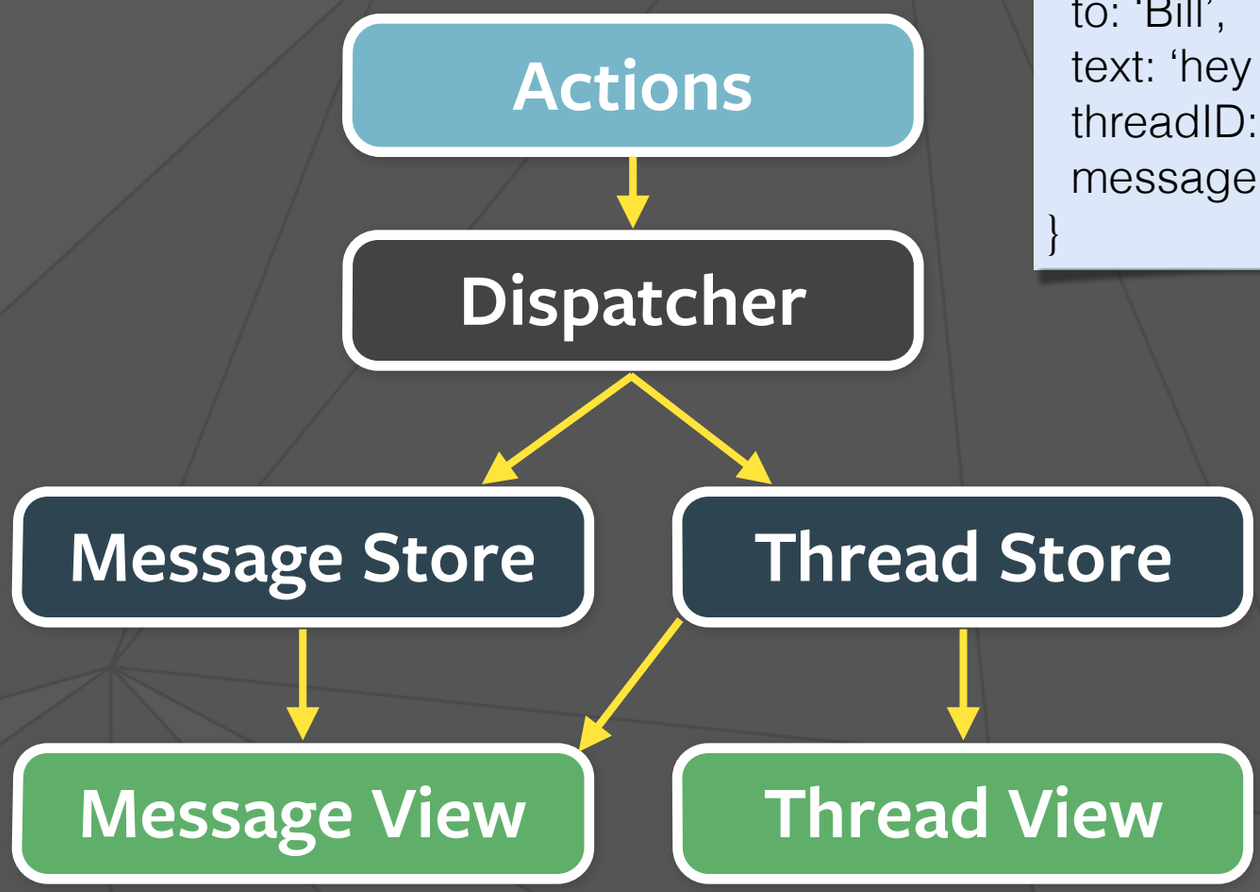

```
action = {
  type: NEW_THREAD,
  to: ‘Bill’,
  text: ‘hey Bill’,
  threadID: ‘5e93696f’,
  messageID: ‘0272fac4’,
}
```

```
Add to map of threads:
‘5e93696f’: {
  participants: [‘Bill’, ‘Jing’],
  messageList: [‘0272fac4’]
}
```

When the ThreadStore gets this action, it can look at the metadata to construct its new thread info. We know that the participants on the list are Bill and the viewer, Jing. And that the first message on that thread has the messageID from the action.

START A NEW CONVERSATION

```
action = {
  type: NEW_THREAD,
  to: 'Bill',
  text: 'hey Bill',
  threadID: '5e93696f',
  messageID: '0272fac4',
}
```

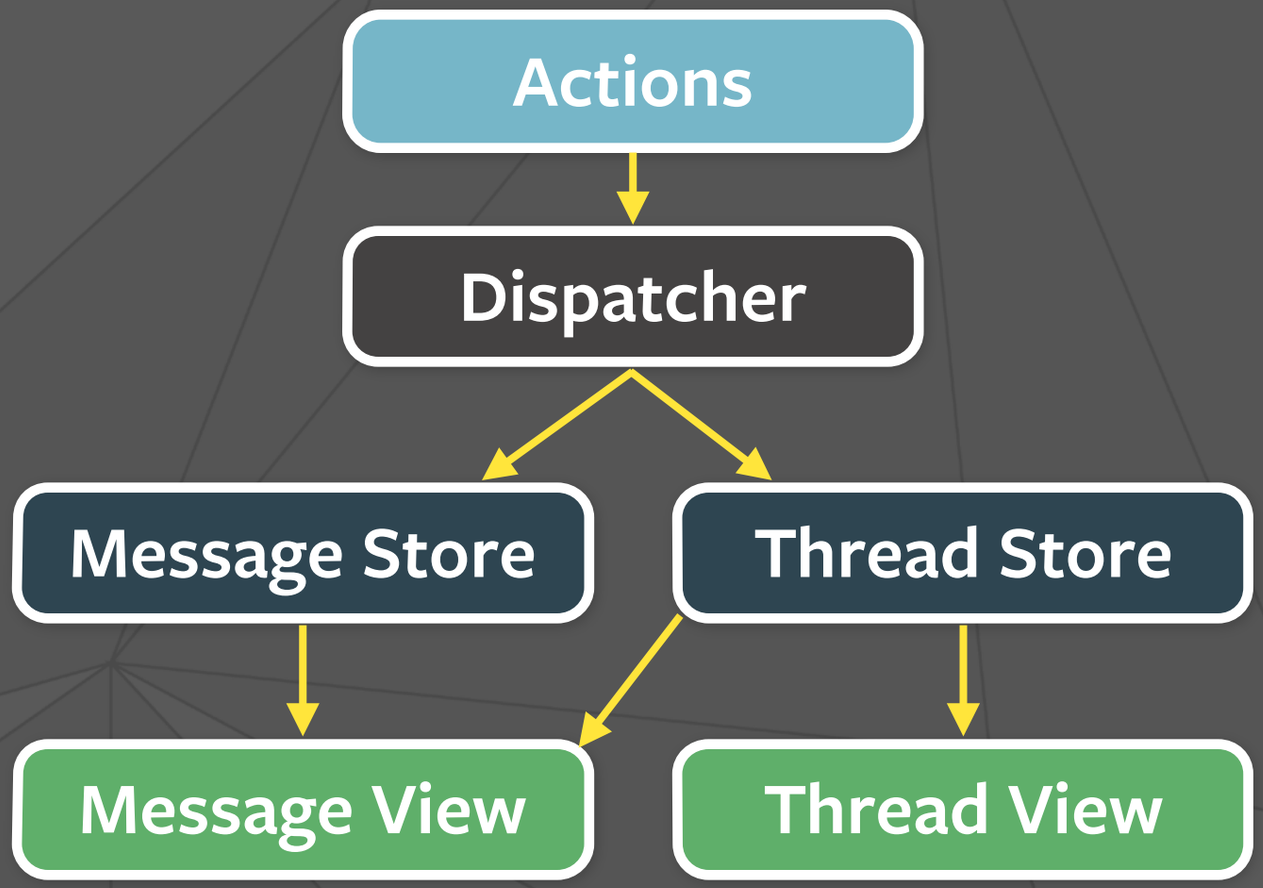



```
Add to map of messages:
'0272fac4' : {
  text: 'hey Bill',
  thread: '5e93696f',
  author: 'Jing'
}
```

```
Add to map of threads:
'5e93696f': {
  participants: ['Bill', 'Jing'],
  messageList: ['0272fac4']
}
```

The MessageStore extracts info from the action in a similar way. With this setup, the logic for how the data is updated is in the same module as the data. When you want to add the ability to add a new thread with a message, you only have to think about how it affects each individual store, rather than think about everything that should change as a result of that action.

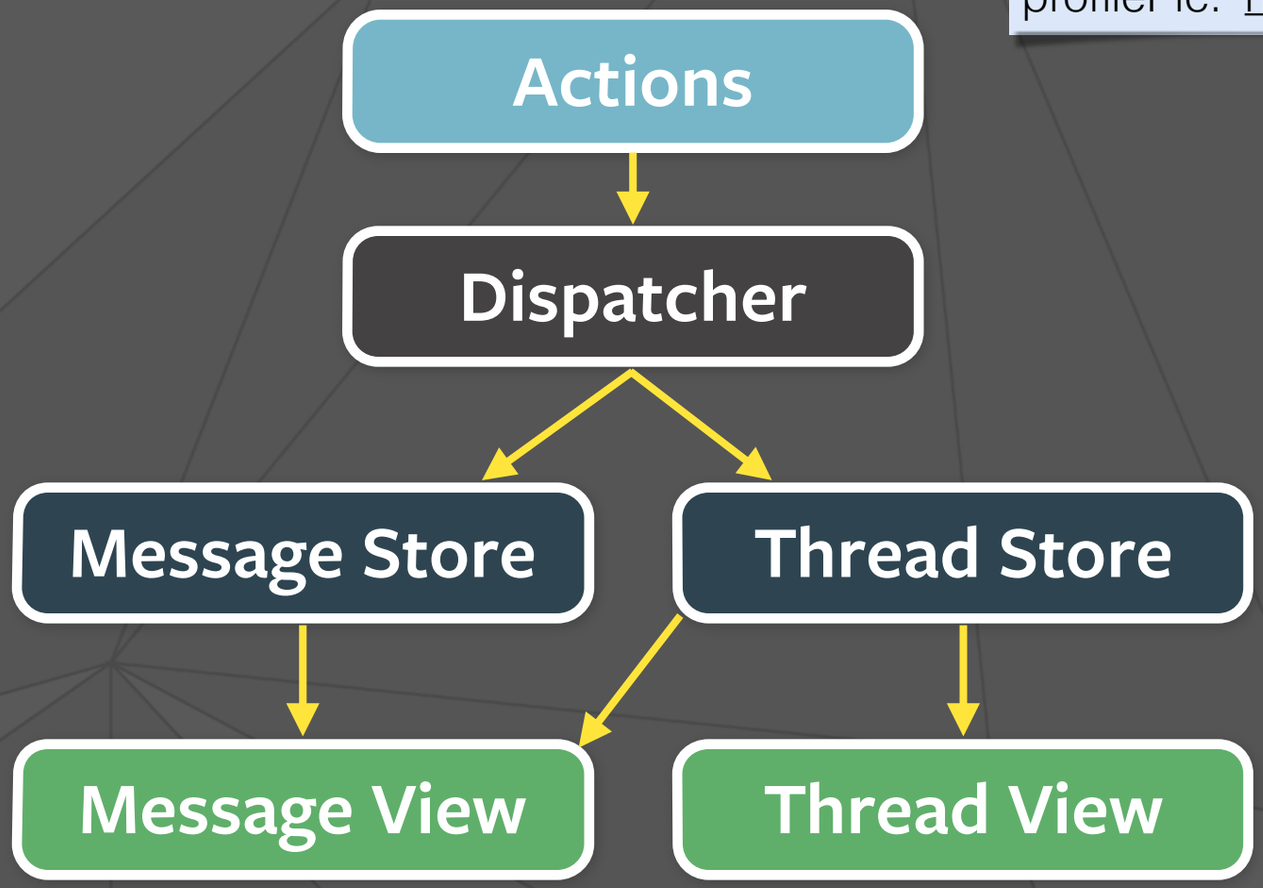
GROUP THREADS



Similarly, let's consider what would change if we wanted to add a group chat feature, with the ability to add and remove participants

GROUP THREADS

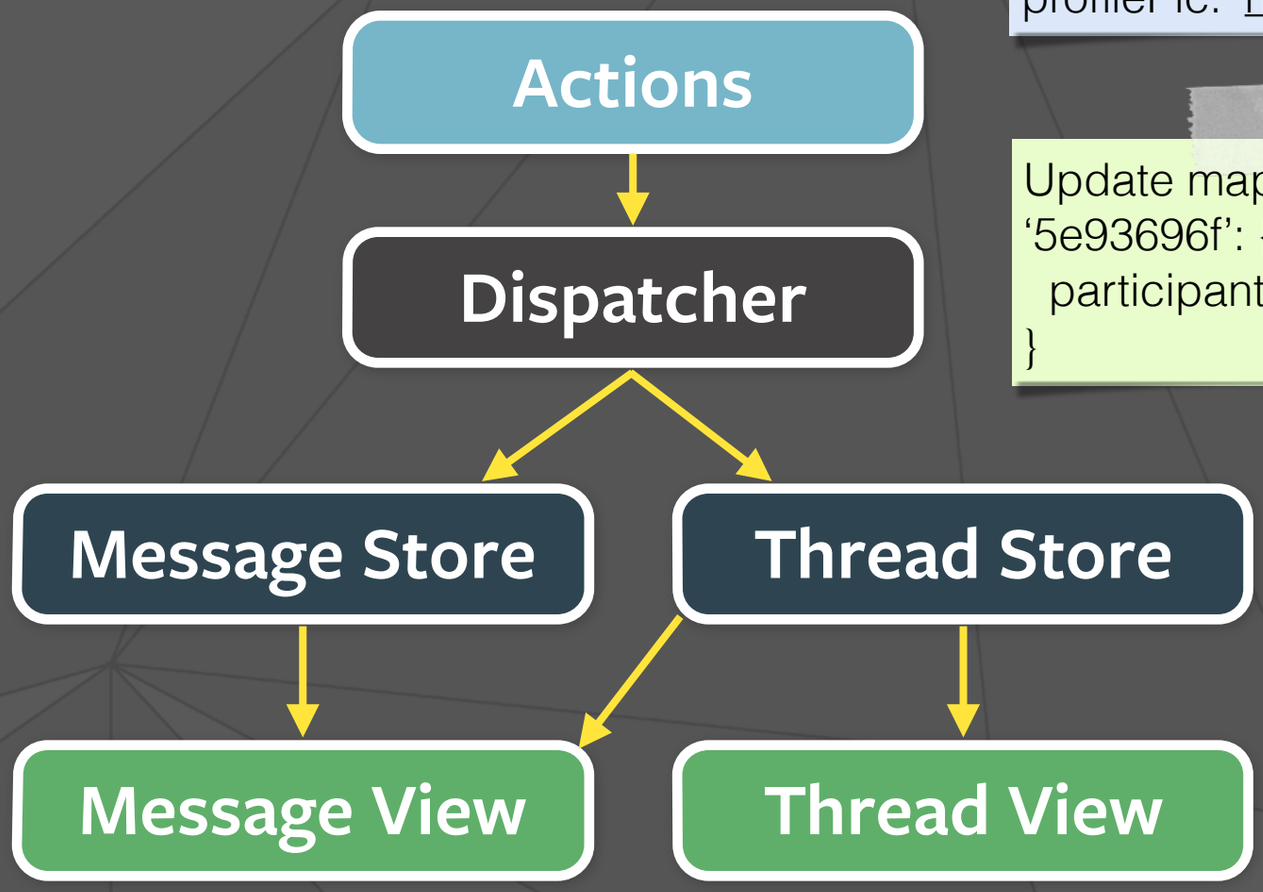
type: add_to_thread,
threadID: ‘5e93696f’,
newParticipant: ‘Chris’,
name: ‘Christopher Chedeau’
profilePic: ‘http://www…'



We have an action coming in that indicates we want to add a new participant to a particular thread, as well as some metadata about the new participant

GROUP THREADS

type: add_to_thread,
threadID: ‘5e93696f’,
newParticipant: ‘Chris’,
name: ‘Christopher Chedeau’
profilePic: ‘http://www…'

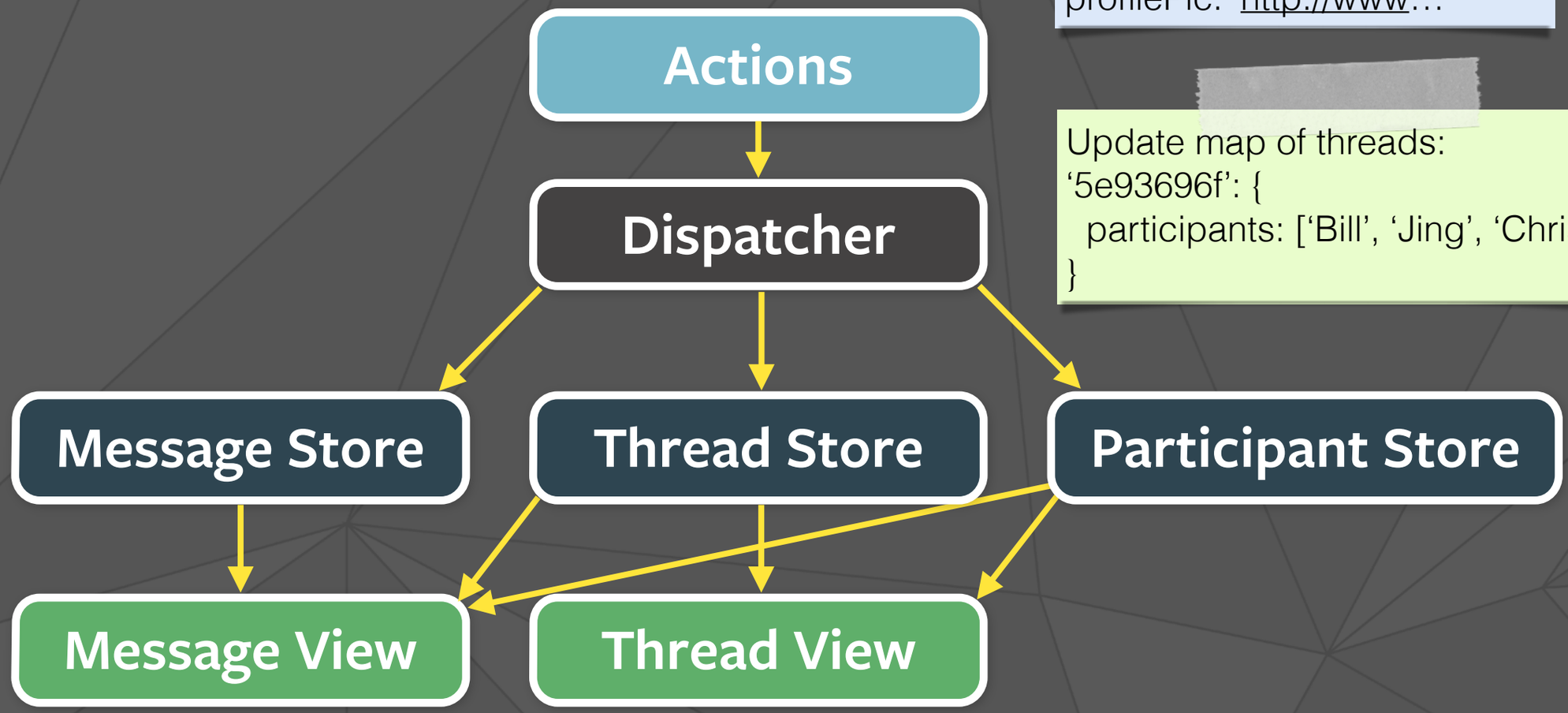


Update map of threads:
‘5e93696f’: {
participants: [‘Bill’, ‘Jing’, ‘Chris’]
}

In this case, we update the participant list in the ThreadStore (still using first names as IDs), but it doesn’t quite make sense to store the full name and profile picture in the ThreadStore…

GROUP THREADS

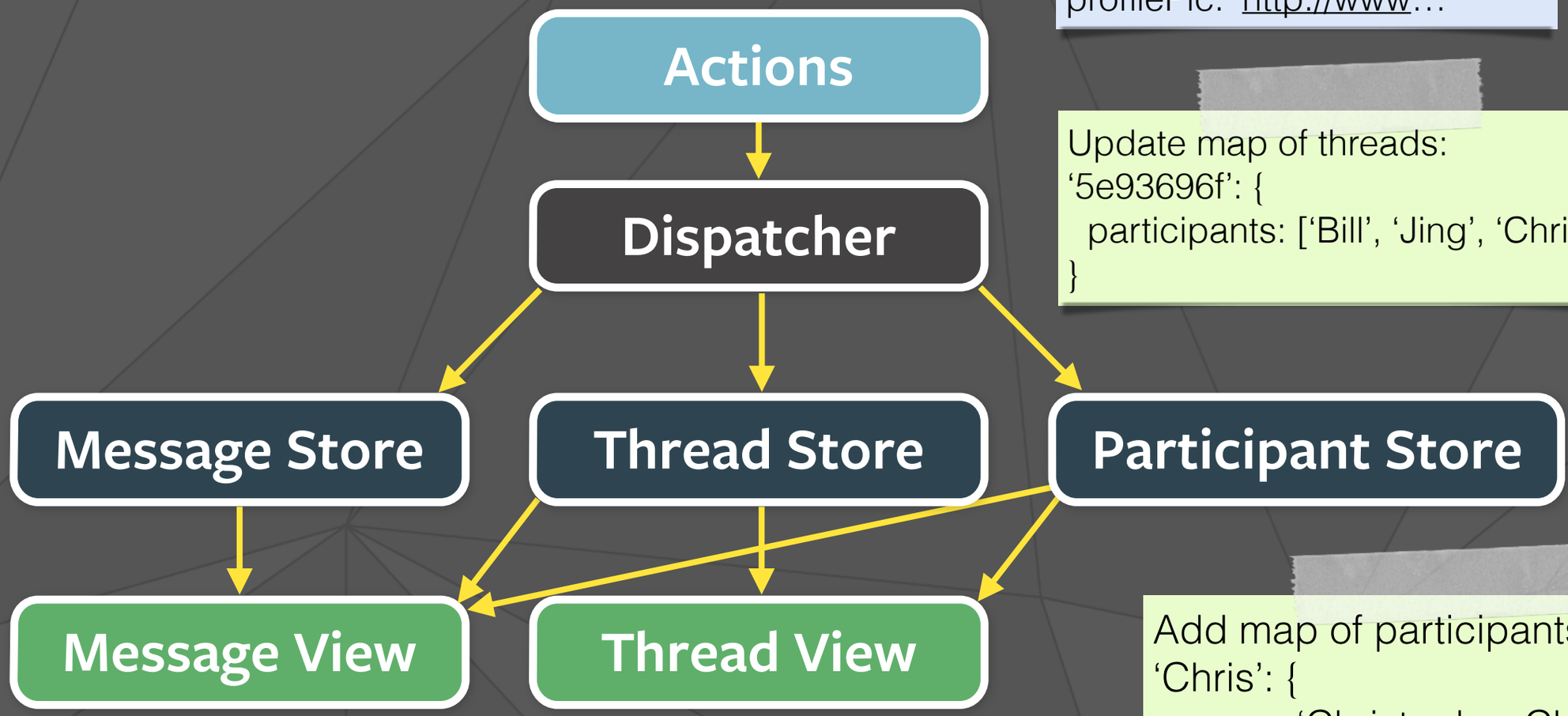
type: add_to_thread,
threadID: ‘5e93696f’,
newParticipant: ‘Chris’,
name: ‘Christopher Chedeau’
profilePic: ‘http://www…'



Update map of threads:
‘5e93696f’: {
participants: [‘Bill’, ‘Jing’, ‘Chris’]
}

In that case, we can add a third store that stores details about participants, indexed by ID

GROUP THREADS



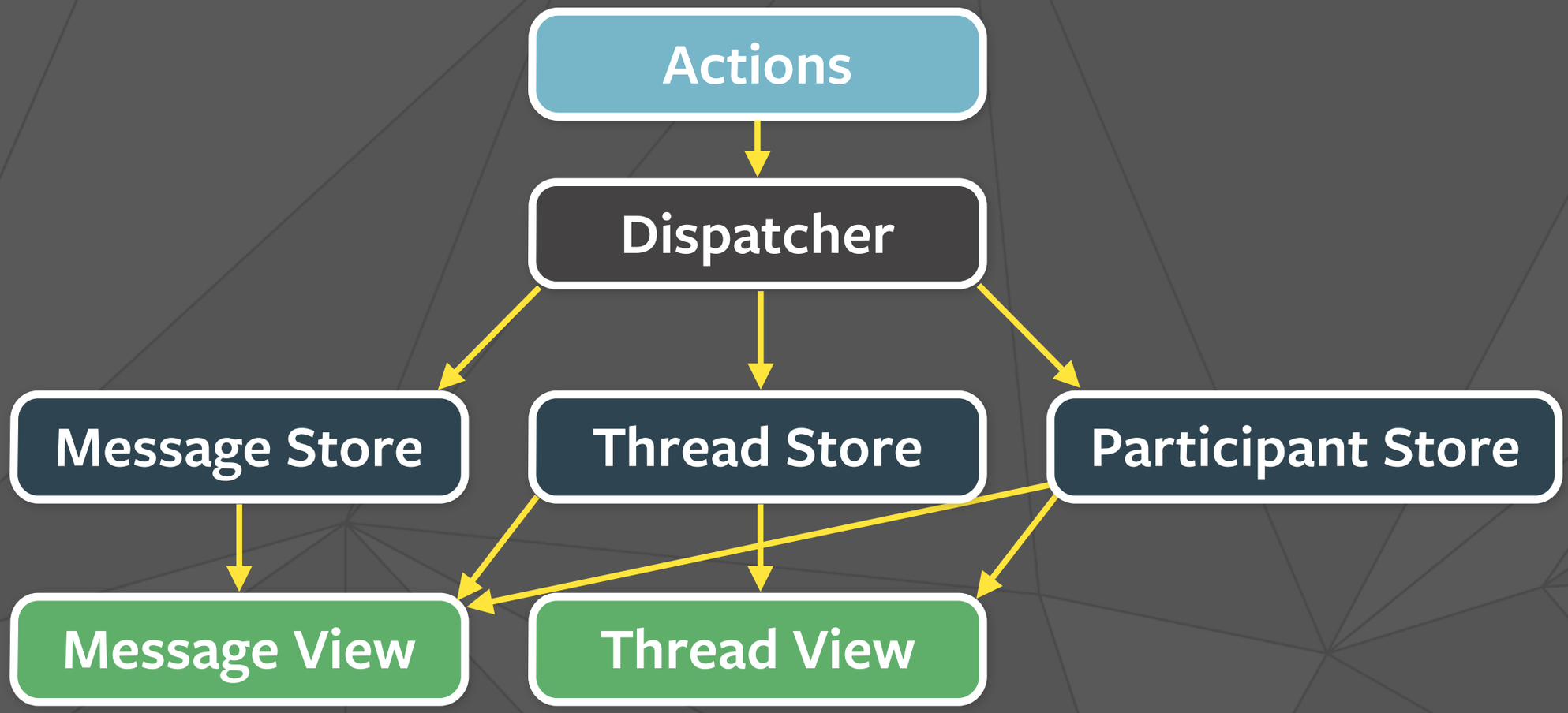
type: add_to_thread,
threadID: ‘5e93696f’,
newParticipant: ‘Chris’,
name: ‘Christopher Chedeau’
profilePic: ‘http://www…'

Update map of threads:
‘5e93696f’: {
  participants: [‘Bill’, ‘Jing’, ‘Chris’]
}

Add map of participants:
‘Chris’: {
  name: ‘Christopher Chedeau’,
  profilePic: ‘http://www…'
}

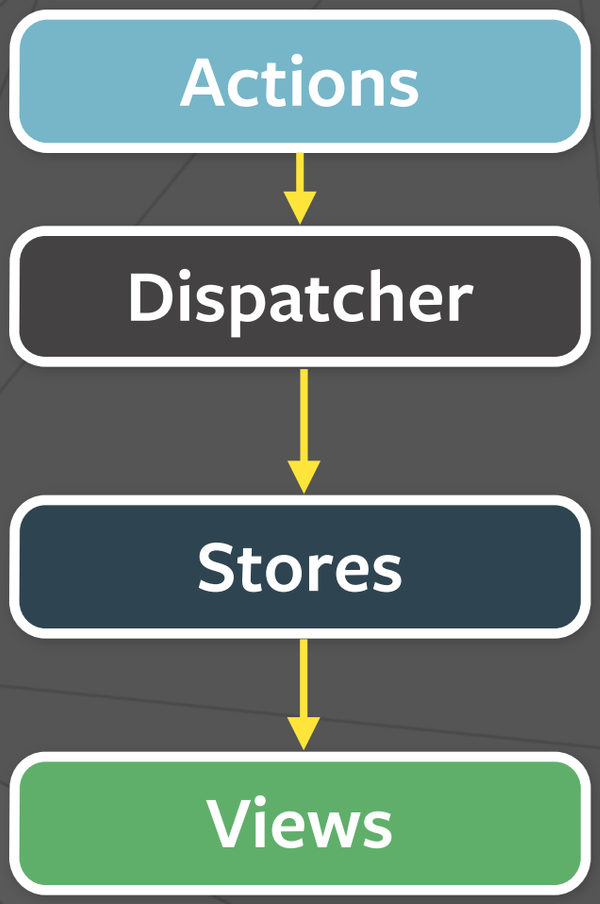
And when it sees this new participant, it can grab the participant info if it doesn’t already have it. The ThreadStore can then reference participants by ID, and views can go to the ParticipantStore for profile photo

DATA FLOW



At this point, the diagram is pretty complicated, and we're not even including the UnreadThreadsStore that we showed earlier

REDUCES TO...



But the idea is that even if the stores and views expand, the underlying structure is still simple. When you make a change or try to debug a problem, you can zoom in to specific stores and only think about how the action should affect each store. There's no cascade of effects from one change, which could happen when the updates are bi-directional.

WHERE DOES FLUX FIT IN

Dataflow Programming

CQRS

Reactive

FLUX

MVC

Functional and FRP

Flux is certainly influenced by the MVC pattern, but it is also has similarities to other programming paradigms.

THE FUTURE OF FLUX

Open source Dispatcher ✓ http://facebook.github.io/flux/docs/dispatcher.html

Example chat app ✓ https://github.com/facebook/flux/tree/master/examples/flux-chat

Flux boilerplate and CLI tools soon!

Needed: Utils, Routers

MORE FLUX

Documentation:
http://facebook.github.io/react/docs/flux-overview.html

TodoMVC Tutorial:
http://facebook.github.io/react/docs/flux-todo-list.html

Screencasts:
https://egghead.io/series/react-flux-architecture

Tools:
http://fluxxor.com/

THANK YOU!