Flux

React Developer Series

Peter Munro

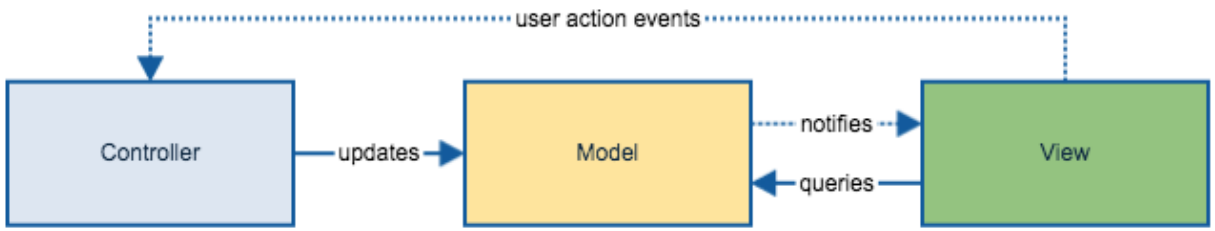
Flux

Flux

Introducing Flux

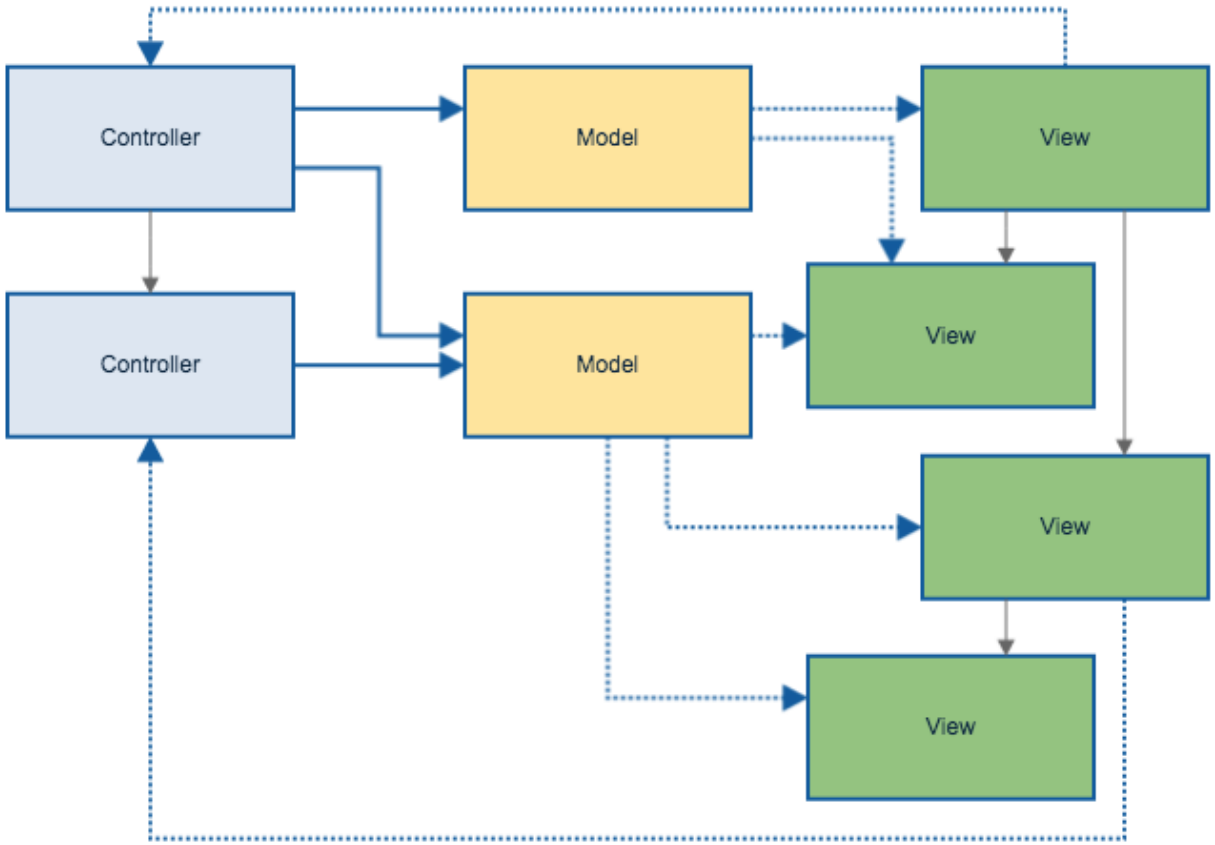
- Flux is not a library or framework
  - Flux is a *design pattern*
- Presented by facebook - Flux website: https://facebook.github.io/flux/

MVC



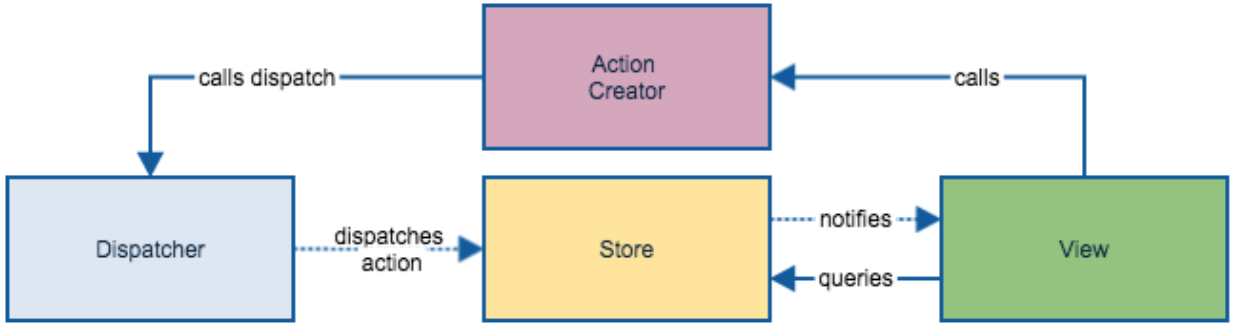
- Simple MVC apps are readily understandable

Scaling MVC



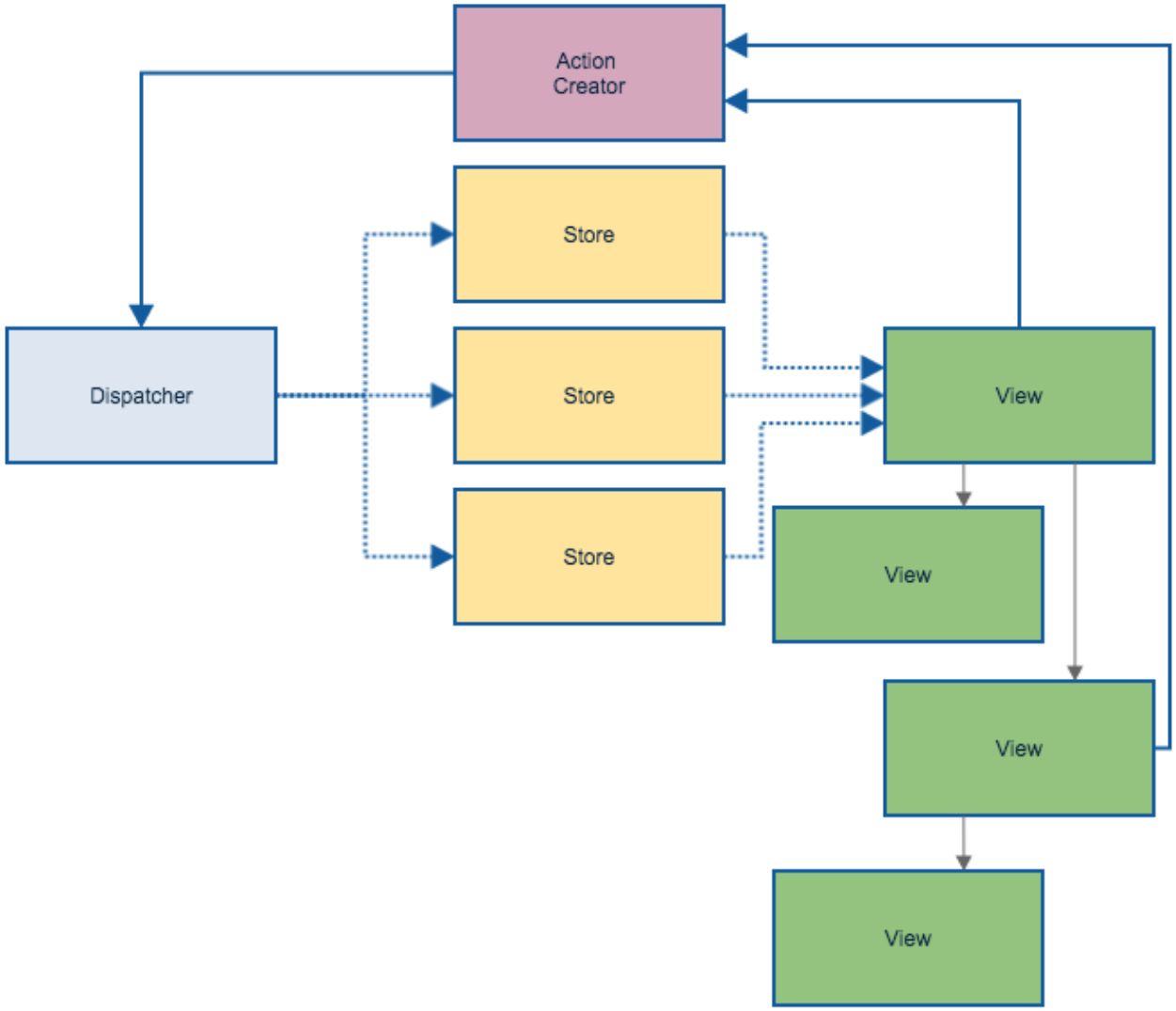
- More complex apps are harder to reason about. The problem: *cascading updates*

Flux



- Flux is designed for client-side scalability
  - Reasoning about our app should be easier
- One-way data flow
  - Data only enters your app via *Actions*

Scaling Flux



View

- Simply a React component
- facebook often refer to them as *ViewControllers*
    - They exhibit characteristics of both MVC Views and Controllers
- Recap: no two-way binding!

Action

- Pure JS object/data
- An Action contains two things:
  1. A `type`
  2. A *payload*: the new data

Example Action Types

- UI: `CREATE_CONTACT`, `DELETE_CONTACT`
- Server: `RECEIVE_CONTACT_UPDATE`, `RECEIVE_CONTACT_DELETE`

ActionCreator

- Creates actions:
  - from **user interactions** (touch events, mouse, keyboard etc)
  - from API **server responses**
- Forwards actions to the *dispatcher*

Dispatcher

- A central hub
  - A singleton
- **Stores register themselves** with the dispatcher
  - Dispatcher maintains a list of registered stores
- Dispatcher is simple
  - Really just a way of propagating actions to stores
  - May also offer:
    - ordering of updates
    - synchronous updates (guarantee that next action doesn't start before the current one has completed)
- facebook offers a dispatcher implementation

Store

A store:

- holds all client-side data
- has no direct setter methods
  - nothing changes the store from outside
  - only one way to get data in: via a dispatcher callback
- receives *every* action
  - it chooses which actions to respond to and which to ignore, via the action's `type`
- allows client to register/unregister change listeners
- emits `change` events when its state changes
  - views update themselves from stores via getter methods

Ordering of Updates

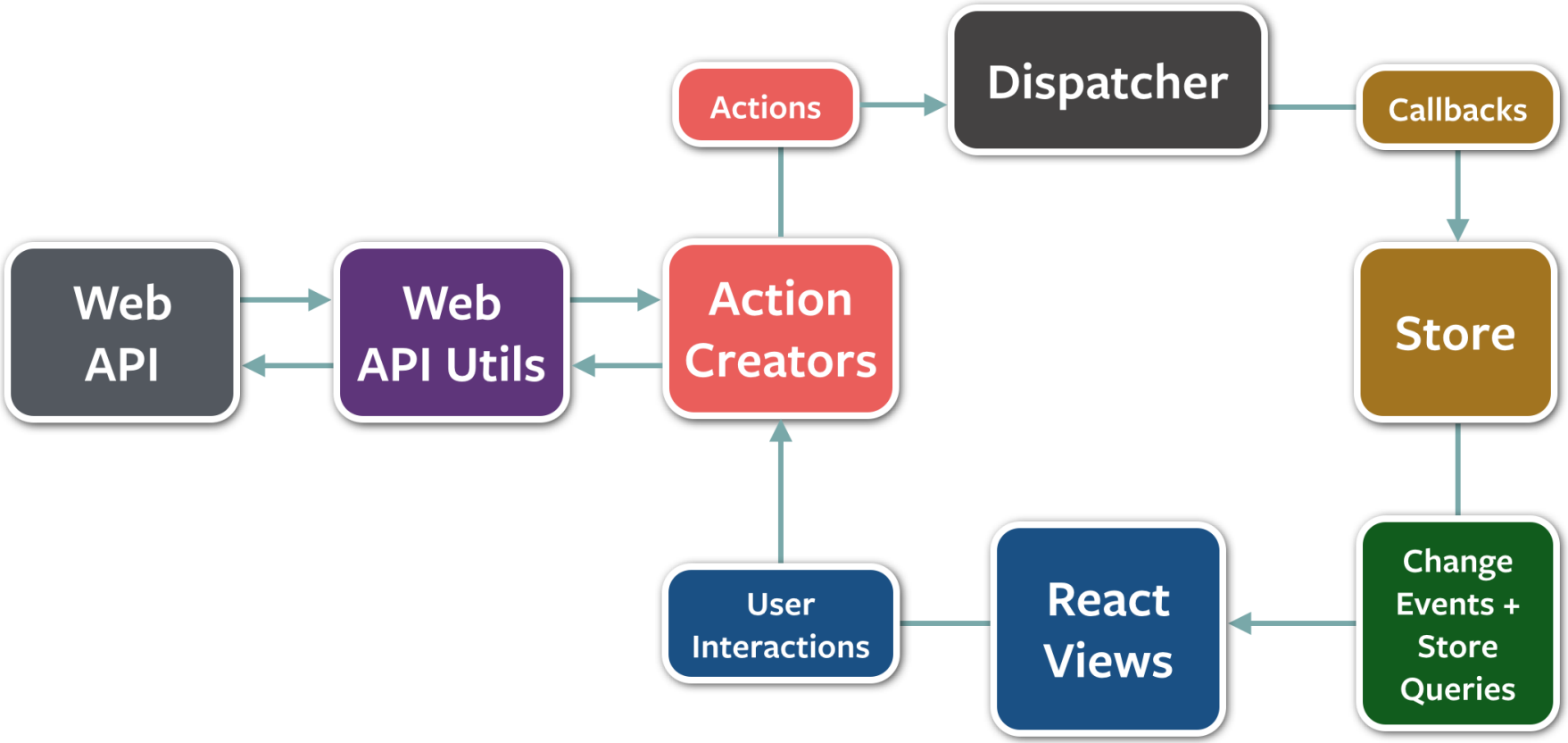
- Store A may need Store B to update itself first
  - Store A may compute values from it
  - So dispatcher must invoke Store B's callback first
- Store A therefore informs dispatcher of this dependency
  - `MyDispatcher.waitFor(storeB)`

Dispatcher Methods

So a dispatcher can be implemented with three methods:

- `dispatch()`
  - Propagate action to stores
- `register()`
  - Called by store to register itself with dispatcher
- `waitFor()`
  - Called by store to inform dispatcher of a dependency

But how to communicate with API Server?



- Reto Schläpfer documented his experience in Async requests with React.js and Flux, revisited

Best Practices

- *All data* is kept in stores
  - Views can have transitory state, but nothing they want to persist
    - Destroying the view shouldn't matter
- *All data changes* happen *only* via actions
- Views declare the data they need
- Actions are just fire-and-forget
  - They don't have callbacks

Flux Implementations

- Many implementations exist
  - Alt
  - Fluxible
  - Fluxxor
  - Marty
  - McFly
  - NuclearJS
  - Facebook's dispatcher in their Flux code repository
  - ...
- And evolutions of Flux
  - Redux