Safer Swift Code with Value Types

What do I mean by safety?

Safe, adjective [*]:
- involving little or no risk of mishap, error, etc.
- dependable or trustworthy

[*] Thanks to Rich Hickey for pointing out this great way to open a talk

Should I use a struct or a class?

It depends![4] [5] [6] [7] [8] [9]

[4] GitHub Styleguide: "Unless you require functionality that can only be provided by a class (like identity or deinitializers), implement a struct instead"

[5] Apple Developer Guide: "In practice, this means that most custom data constructs should be classes, not structures"

[6] Apple Swift Blog: "One of the primary reasons to choose value types over reference types is the ability to more easily reason about your code."

[7] http://faq.sealedabstract.com/structs_or_classes/: "when it comes time to actually save any of those models to disk, or push them over a network, or draw them to the screen, or whatever, you need a class. Classes everywhere. Not "immutable valuable types everywhere" like in the Struct Philosophy™"

[8] http://owensd.io/2015/07/05/re-struct-or-class.html

[9] https://www.mikeash.com/pyblog/friday-qa-2015-07-17-when-to-use-swift-structs-and-classes.html

Agenda

1. **What** are Value Types vs. Reference Types
2. **Why** is this topic relevant now?
3. **How:** A Practical Example of a Value Oriented Architecture

Values vs. References

Reference Types

```
class PersonRefType {
  let name:String
  var age:Int

  // ..
}
// 1
let peter = PersonRefType(name: "Peter", age: 36)
// 2
let peter2 = peter
// 3
peter2.age = 25
// peter {"Peter", 25}
// peter2 {"Peter", 25}
```

Value Types

```
struct Person {
   let name:String
   var age:Int
}
// 1
let petra = Person(name:"Petra", age:25)
// 2
var petra2 = petra
// 3
petra2.age = 20
// petra  {"Petra", 25}
// petra2 {"Petra", 20}
```

Why now?

Foundation / C Types

Reference Types:

- `NSArray`
- `NSSet`
- `NSData`

Value Types:

- `NSInteger`
- `Struct`

Swift Standard Library

Reference Types:

- `ManagedBuffer(?)`
- `NonObjectiveCBase(??)`

Value Types:

- `Array`
- `String`
- `Optional`

Enums and Structs in Swift are Powerful

— Can have properties

— Can have methods

— Can conform to protocols

So What Can't They Do?

"Indeed, in contrast to structs, Swift classes support **implementation inheritance**, (limited) reflection, deinitializers, and **multiple owners**."

Andy Matuschak[1]

[1] http://www.objc.io/issue-16/swift-classes-vs-structs.html

We've Already Been Doing This!

```
@property (copy) NSString *userName;
```

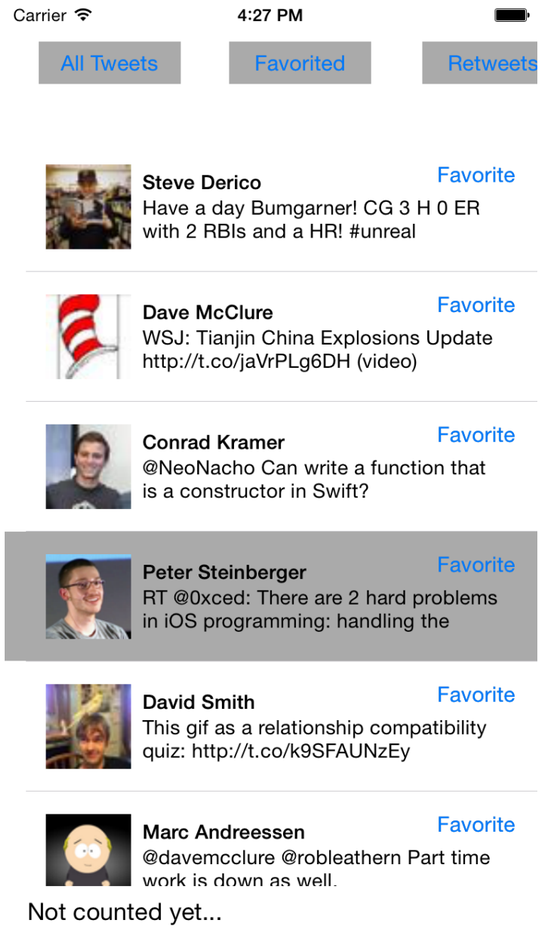
Case Study

A Twitter Client Built on Immutable Value Types

Twitter Client

1. Download the latest 50 tweets and display them
2. Allow to filter tweets (RT only, favorited tweets only, etc.)
3. *Allow user to favorite tweets (should be synced with server)*

Twitter Client



Not counted yet...

How can we favorite Tweets?

It Is Very Simple with OOP

```
tweet.favorited = true
```

It Is Simple with OOP

```
let lockQueue = dispatch_queue_create("com.happysyncing", nil)
dispatch_sync(lockQueue) {
    tweet.favorited = true
}
```

Is It Simple with OOP?

```
let lockQueue = dispatch_queue_create("com.happysyncing", nil)
dispatch_sync(lockQueue) {
    tweet.favorited = true
    NSNotificationCenter.defaultCenter().
        postNotificationName("Tweet Changed", object: tweet)
}
```

Modeling Change is Hard!

```
let lockQueue = dispatch_queue_create("com.happysyncing", nil)
dispatch_sync(lockQueue) {
    tweet.favorited = true
    NSNotificationCenter.defaultCenter().
        postNotificationName("Tweet Changed", object: tweet)
    tweetAPIClient.markFavorited(tweet.identifier)
}
```

Modeling Change is Hard!

— Protect against unwanted updates

— Distribute new value throughout application

— Understand what the underlying *identity* of an object is and perform update accordingly

Modeling Change is Hard!

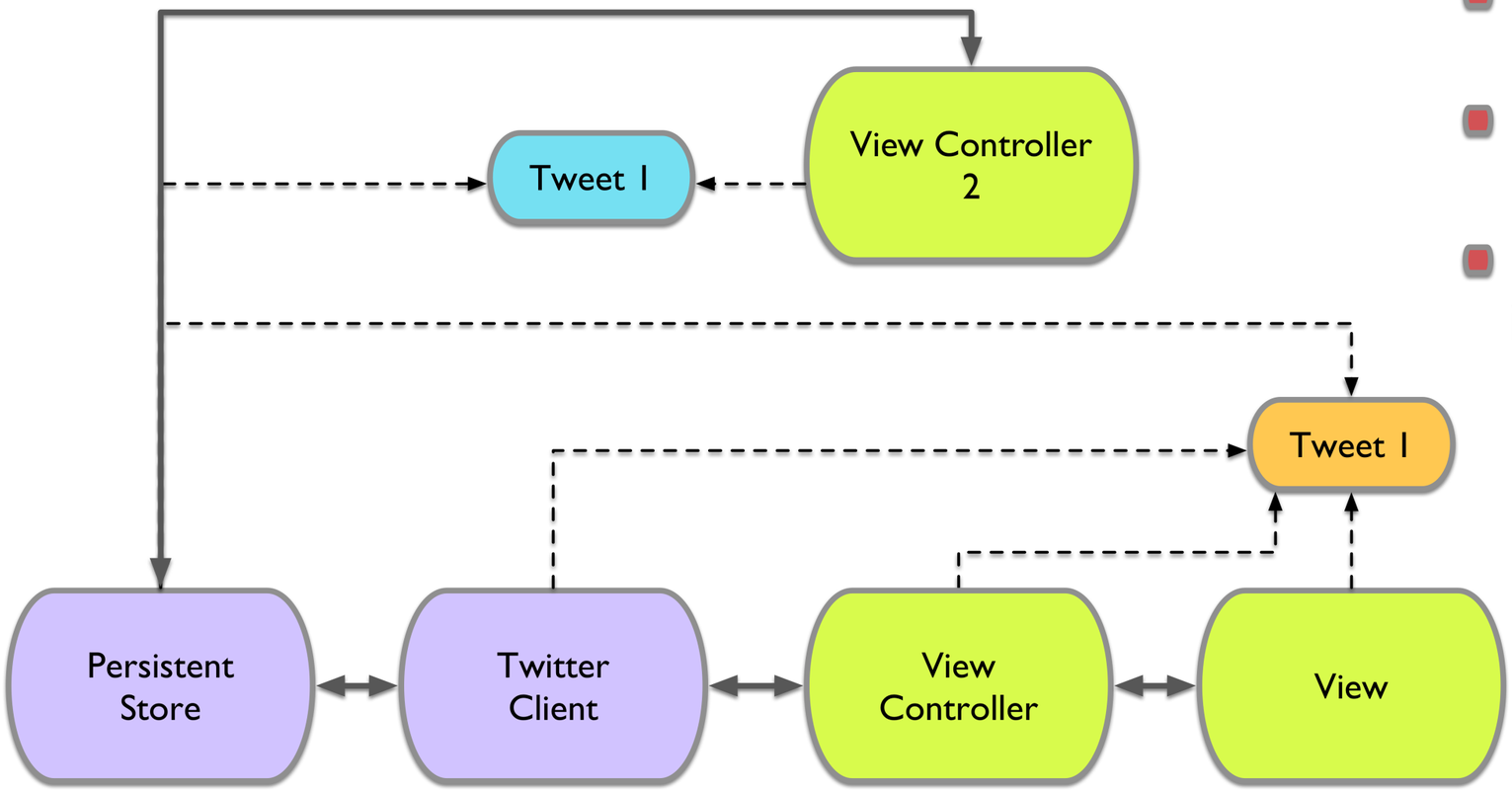
— Protect against unwanted updates

— Distribute new value throughout application

— Understand what the underlying *identity* of an object is and perform update accordingly

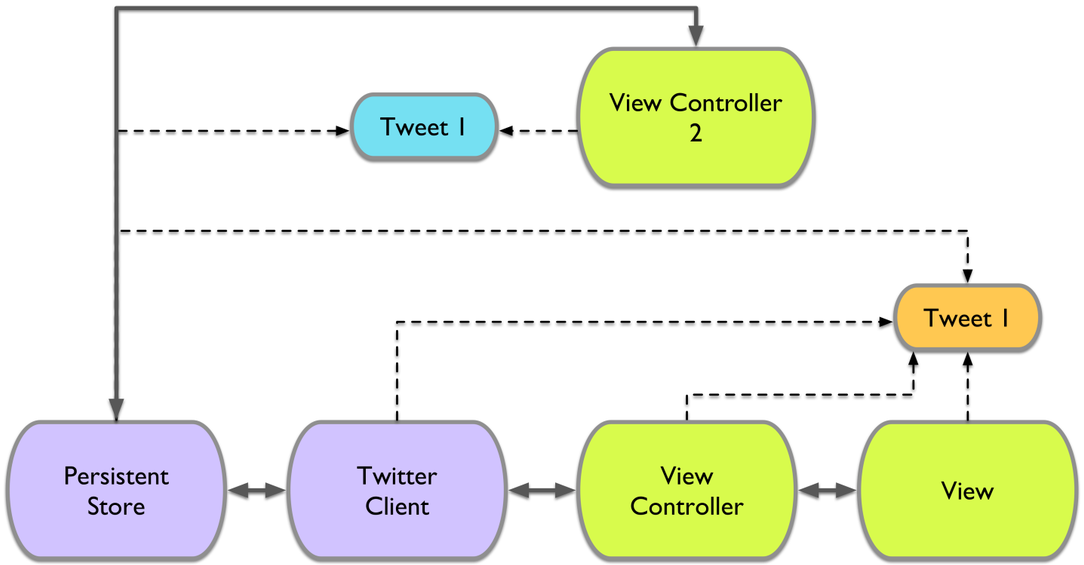
-> **We need to this in all places where we mutate values!**

OOP Data Flow



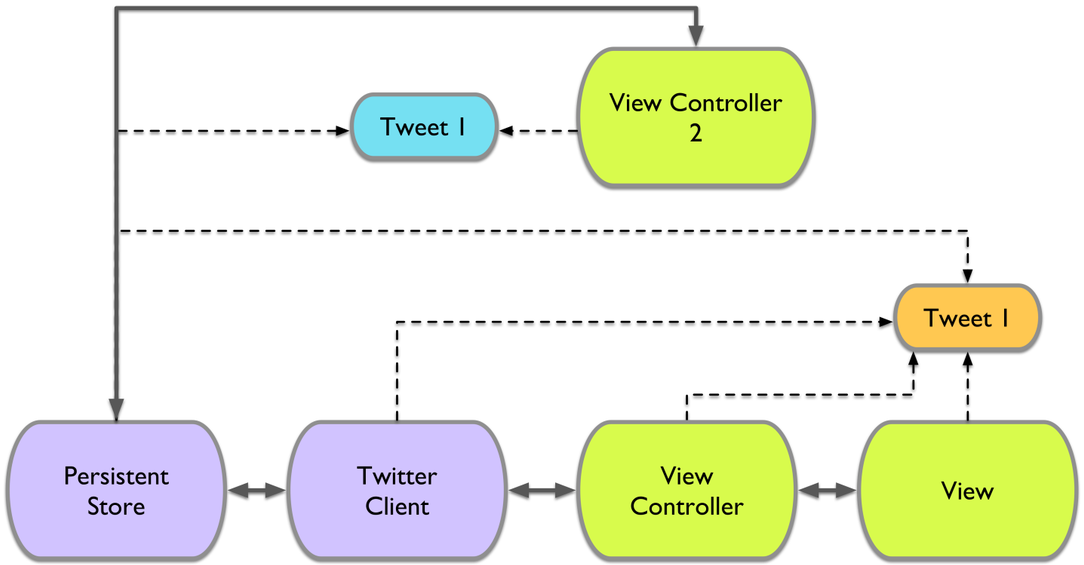
- Multidirectional Data Flow
- Multiple Ownership
- Multiple instances representing identical data

Is the OOP Data Flow Safe?



Safe, adjective:
- involving little or no risk of mishap, error, etc.
- dependable or trustworthy

Is the OOP Data Flow Safe?



Safe, adjective:
- involving little or no risk of mishap, error, etc.
- dependable or trustworthy

-> A lot of room for error

Modelling Change is Hard!

How Can We Model Change With Immutable Value Types?

How Can We Model Change With Immutable Value Types?

Model change to values as values!

How Can We Model Change With Immutable Value Types?

Model change to values as values:

— Create a new Tweet for every change

How Can We Model Change With Immutable Value Types?

Model change to values as values:

— Create a new Tweet for every change

— Save these mutations in a local change set

How Can We Model Change With Immutable Value Types?

Model change to values as values:

— Create a new Tweet for every change

— Save these mutations in a local change set

— Save the server state in a separate data set

How Can We Model Change With Immutable Value Types?

Model change to values as values:

— Create a new Tweet for every change

— Save these mutations in a local change set

— Save the server state in a separate data set

— Provide a merged view on list of tweets

How Can We Model Change With Immutable Value Types?

Model change to values as values:

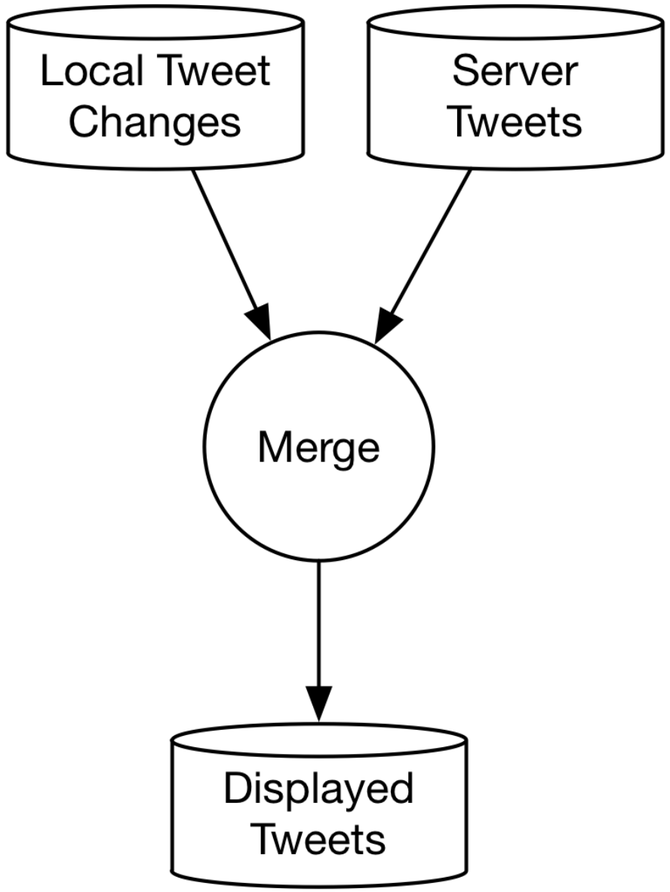
— Create a new Tweet for every change

— Save these mutations in a local change set

— Save the server state in a separate data set

— Provide a merged view on list of tweets

— Provide method to sync local state to server


Local Tweet Changes
Server Tweets
Merge
Displayed Tweets

Where do we store state, how is new state distributed?

Where do we store state, how is new state distributed?

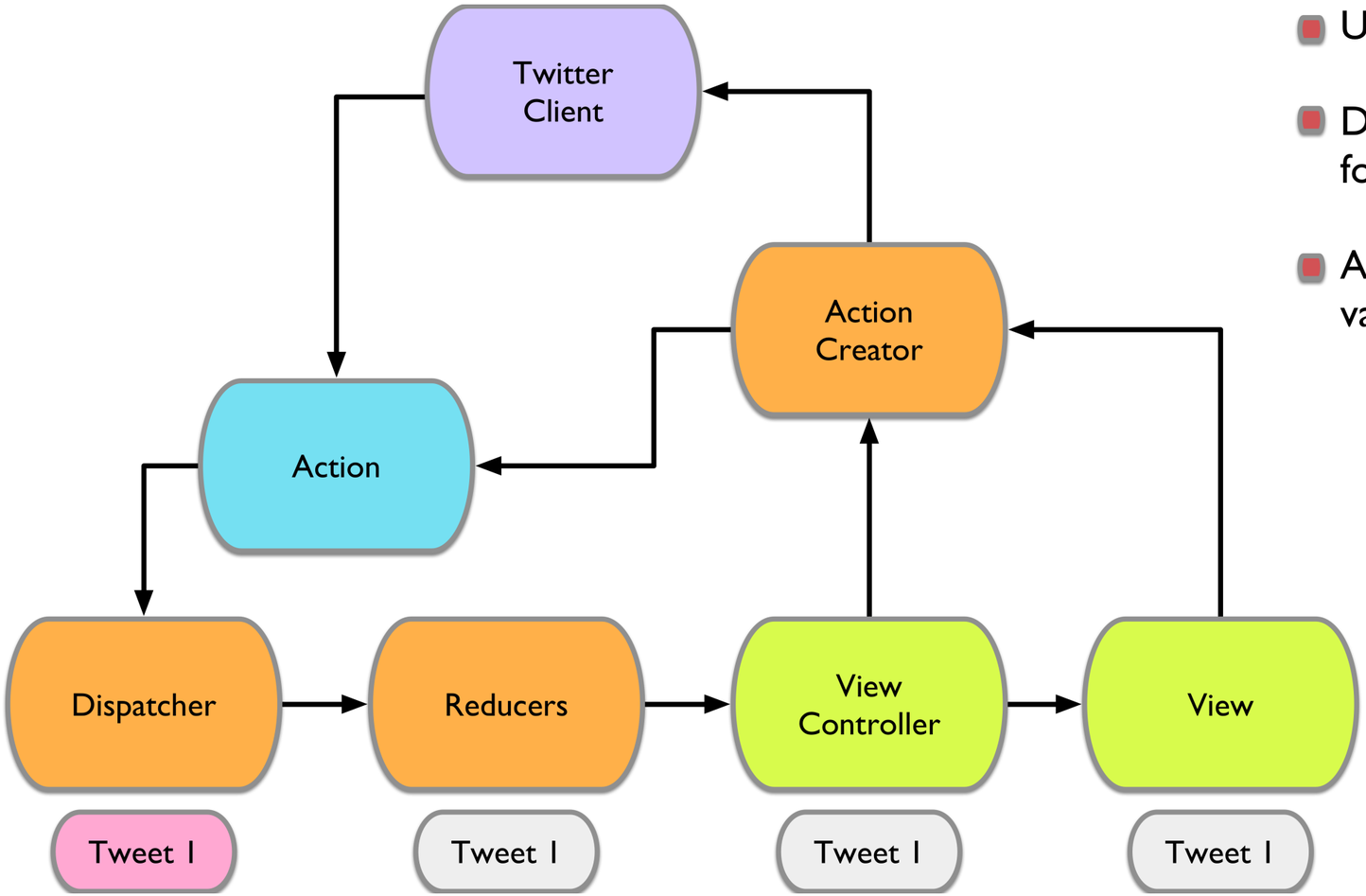
Drawing inspiration from the JavaScript world [11] [12]:

-> **Unidirectional Data Flow**

[11] https://facebook.github.io/flux/

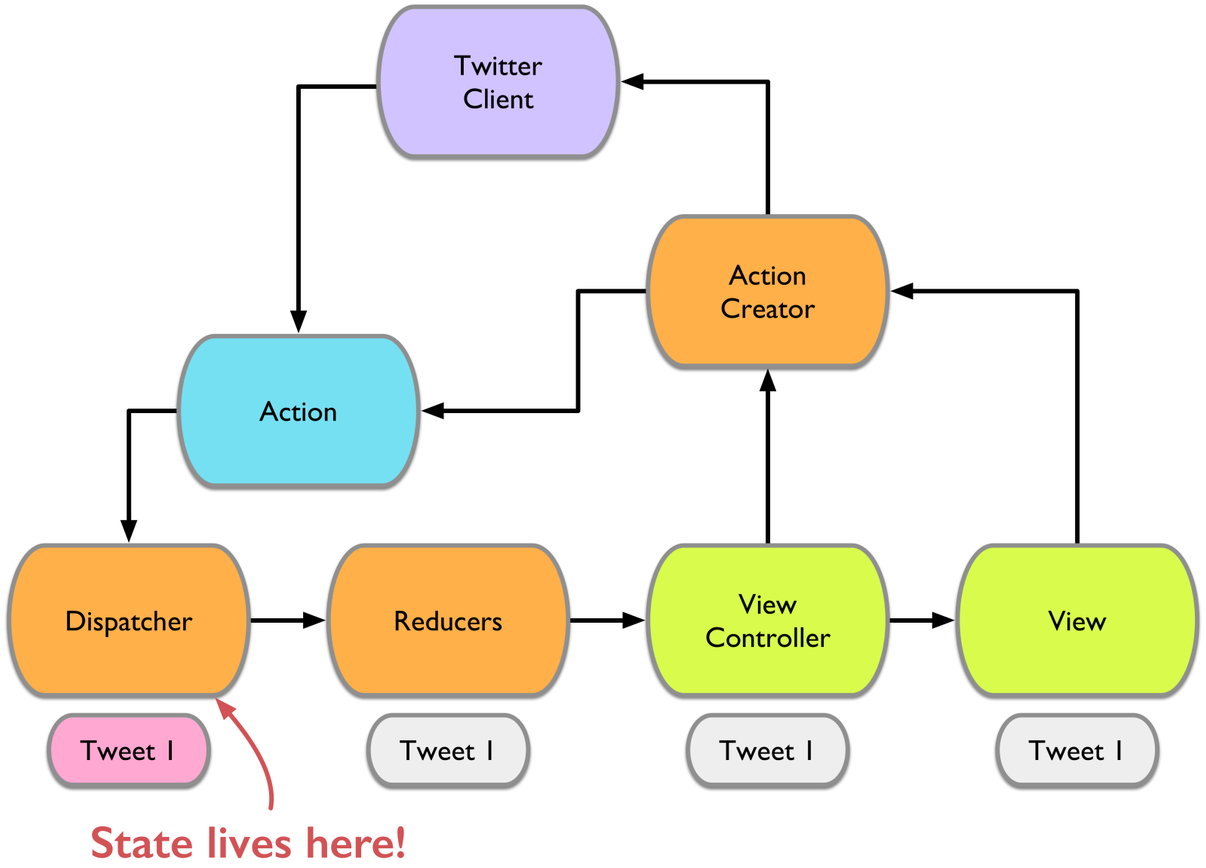
[12] https://github.com/rackt/redux

Unidirectional data flow



- Unidirectional data flow
- Dispatcher is only source of truth for current state
- Application only deals with immutable value types

Unidirectional data flow



- **ActionCreator**: Provides interface for business logic
- **Action**: Represents an individual state mutation
- **Dispatcher**: Stores state, invokes reducers, publishes new state
- **Reducers**: Pure functions, combine current state and action to produce a new state

Implementing unidirectional data flow

Representing State

— State will be stored in the Dispatcher in the following form:

```
typealias TimelineState = (serverState: [Tweet], localState: [Tweet])

typealias TimelineMergedState = (serverState: [Tweet],
                                 localState: [Tweet],
                                mergedState: [Tweet])
```

Defining Actions

— Each action describes an atomic mutation to the application state

— Action Creators vend these Actions Reducers implement them

```
enum Action {
  case Mount
  case FavoriteTweet(Tweet)
  case UnfavoriteTweet(Tweet)
  case SetServerState([Tweet])
  case SetLocalState([Tweet])
}
```

Favoriting a Tweet

Within the `TimelineViewController` we trigger the state change by dispatching an Action Creator:

```
func didFavorite(tweetTableViewCell:TweetTableViewCell) {
  let currentTweet = tweetTableViewCell.tweet!

  if (currentTweet.isFavorited) {
    timelineDispatcher.dispatch { TimelineActionCreator.unfavoriteTweet(currentTweet) }
  } else {
    timelineDispatcher.dispatch { TimelineActionCreator.favoriteTweet(currentTweet) }
  }

  timelineDispatcher.dispatch { TimelineActionCreator.syncFavorites() }
}
```

Favoriting a Tweet

Within the `ActionCreator` we generate an according action:

```
static func favoriteTweet(tweet: Tweet) -> ActionCreator {
  return { state, dispatcher -> Action? in
    return .FavoriteTweet(tweet)
  }
}
```

Favoriting a Tweet

Within the `Reducer` we generate a new state based on the old one:

```
struct TimelineReducers {
  //...
  static func favoriteTweet(var state: TimelineState, tweet: Tweet) -> TimelineState {
    //...
  }

}
```

Favoriting a Tweet

```
static func favoriteTweet(var state: TimelineState, tweet: Tweet) -> TimelineState {
  let newTweet = Tweet(
    content: tweet.content,
    identifier: tweet.identifier,
    user: tweet.user,
    type: tweet.type,
    favoriteCount: tweet.favoriteCount,
    isFavorited: true
  )

  let tweetIndex = find(state.localState, tweet)

  if let tweetIndex = tweetIndex {
    // if we have stored local state for this tweet previously, override here
    state.localState[tweetIndex] = newTweet
  } else {
    // else append new state
    state.localState.append(newTweet)
  }

  return state
}
```

Favoriting a Tweet

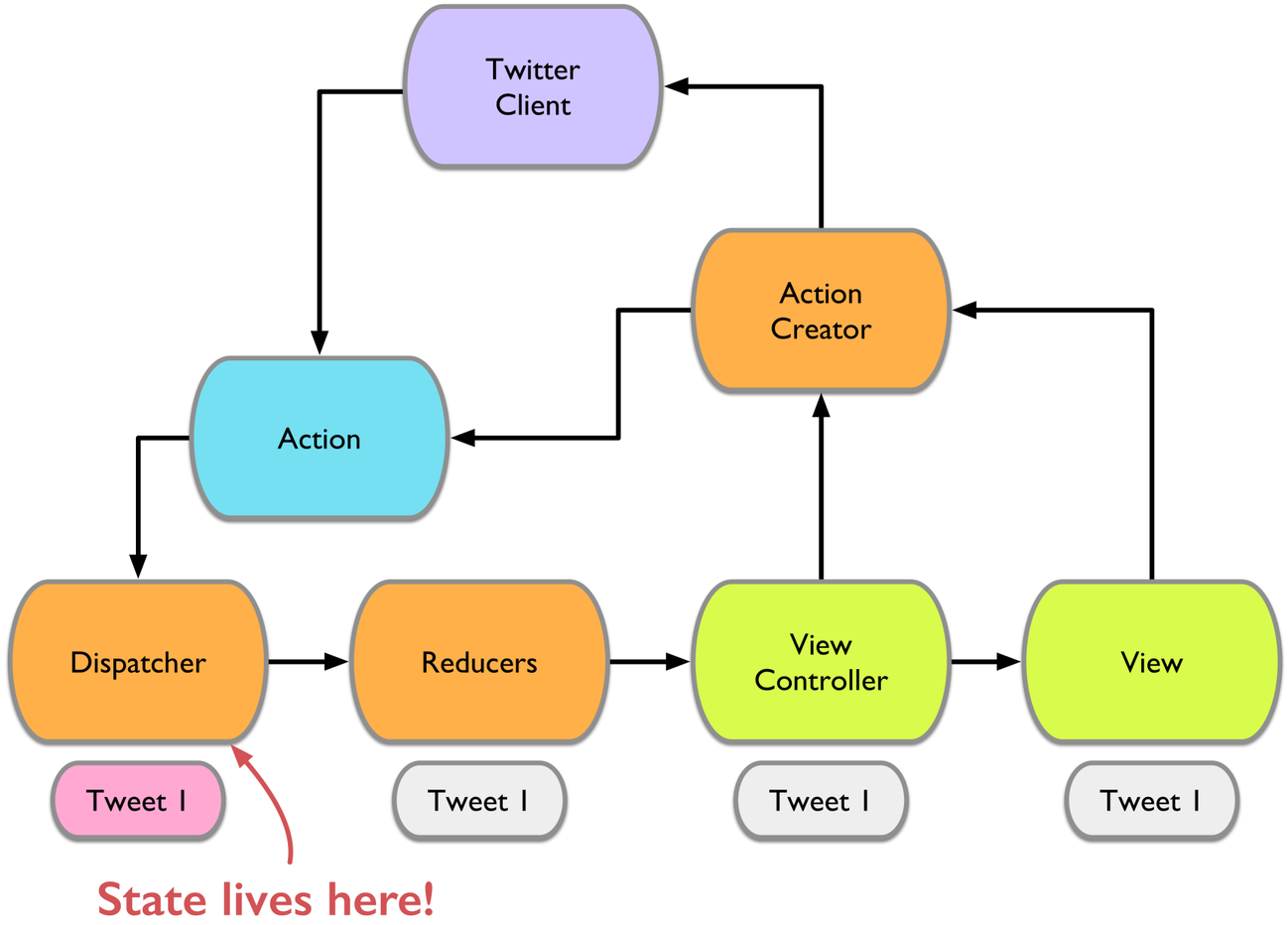
Then the `Dispatcher` forwards the new state to its subscribers, which inlcudes the `TimelineViewController`:

```
extension TimelineViewController: TimelineSubscriber {

  func newState(state: TimelineMergedState) {
    // table view is reloaded when 'tweets' is set
    tweets = state.mergedState
  }

}
```

Favoriting a Tweet: Recap



Syncing Change

Syncing Change

1. Iterate over each mutation in the local change set

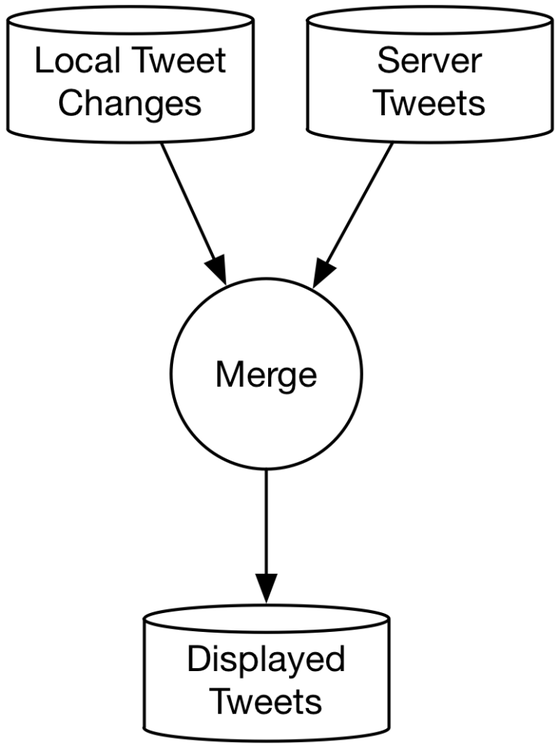
Syncing Change

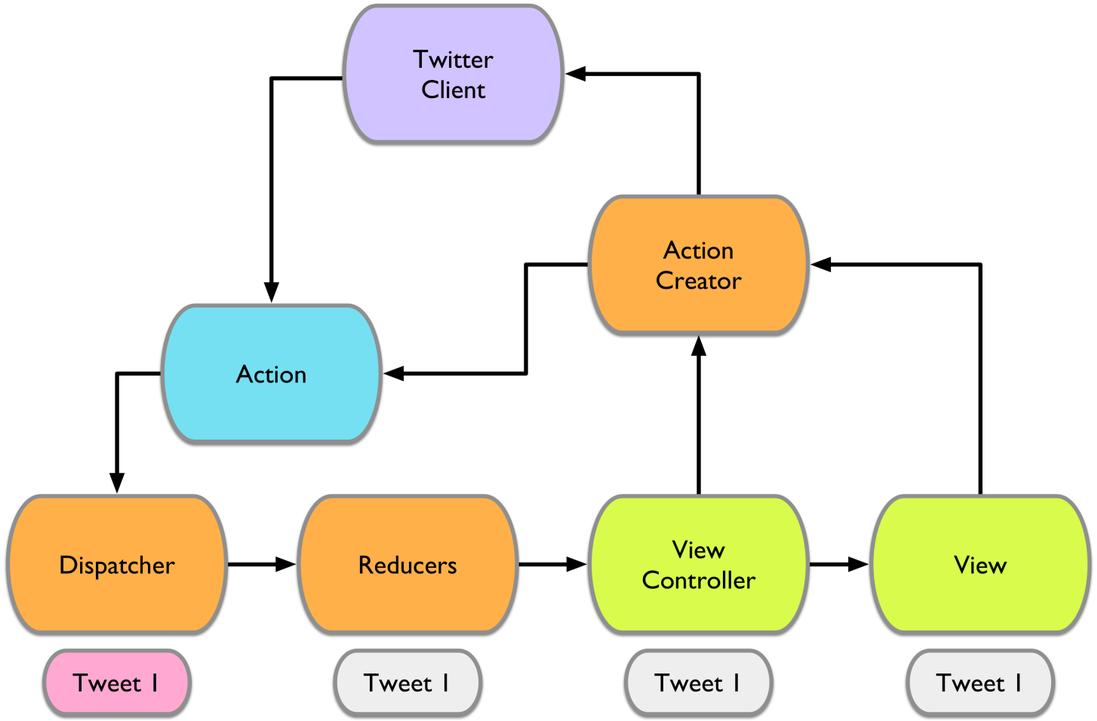
1. Iterate over each mutation in the local change set
2. Generate API request that syncs that local change to server

Syncing Change

1. Iterate over each mutation in the local change set
2. Generate API request that syncs that local change to server
3. Upon each API response:
   - If success: remove tweet from local change set
   - If failure: leave tweet in local change set

A Twitter Client built on Immutable Value Types





Should I use a struct or a class?

It's an architectural question!

It's not about `struct` vs. `class`

It's about:
- Shared state vs. isolated state
- Mutable state vs. immutable state [10]

[10] WWDC 2014, Session 229, Advanced iOS Application Architecture and Patterns

Downsides of a Value Oriented Architecture

— Unconventional: can be an issue when working on larger teams

— Can be difficult to integrate with frameworks that rely on reference semantics, e.g. Core Data

Benefits of a Value Oriented Architecture

— Confidence that no one will change our data under the covers

— State modification & propagation needs to be handled explicitly and can be implemented in one place

— Modeling change as data opens opportunities, e.g. simpler client server sync

— Easier to test

The Value Mindset

Twitter: @benjaminencz

Project & Slides: http://bit.ly/swift-values

Related, great talks: Controlling Complexity [20], The Value of Values [21]

[20] https://realm.io/news/andy-matuschak-controlling-complexity/

[21] http://www.infoq.com/presentations/Value-Values