HIT3329 / HIT8329
Creating Data Driven Mobile Applications

Lecture 6
Implementing Networking

Presented by Paul Chapman

Adjunct Lecturer F-ICT
Director, Long Weekend LLC
Swinburne University of Technology

SWINBURNE UNIVERSITY OF TECHNOLOGY

Last Lecture Reviewed

1. Core Data Concepts
2. Creating a Managed Model
3. Retrieving Data
4. CRUD Operations
5. Notification Center*

Questions?

What's On For Today?

1. Simple Networking
2. UI Concerns & Limiting Conditions
3. iOS App Lifecycle
4. Model-View-Controller (MVC)

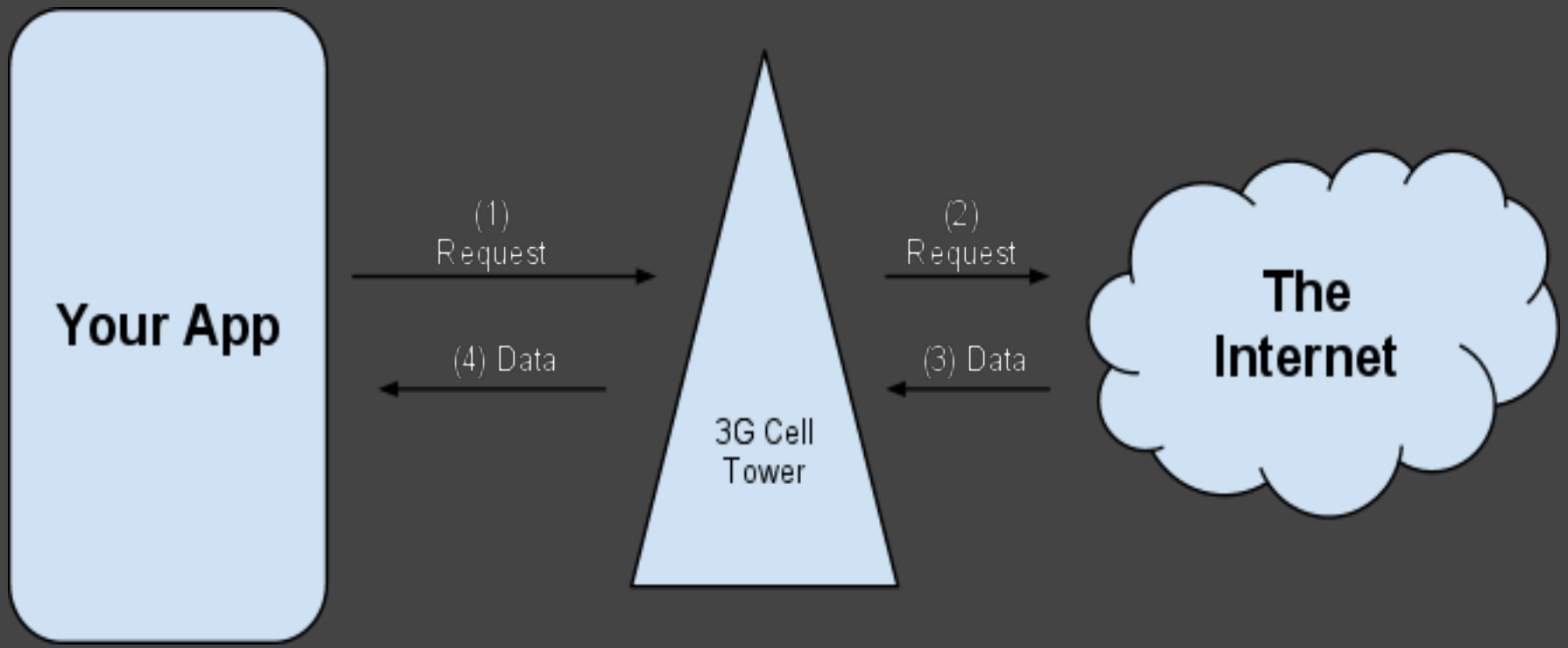
1.0 Accessing the Internet

- Most mobile apps access information stored somewhere else ... like on the Internet
- Cocoa Touch handles this easily

```
NSError *error = NULL;
NSURL *myUrl = [NSURL URLWithString:@"http://www.google.com/"];
NSString *html = [NSString stringWithContentsOfURL:myUrl
                        encoding:NSUTF8StringEncoding
                        error:&error];
```

1.1 Network Latency

Only one problem. Network requests take time, often many seconds ...



1.2 Network Calls Freeze the App

- Executing network requests normally will *freeze* the user interface
- This means:
  - app won't respond to taps
  - user feels app has crashed
- The *foreground thread* freezes whilst the app waits for a network response
- *Synchronous network access* = bad

1.3 The Solution: Multitasking

- NSURLConnection objects *execute in the background* while you do other things, e.g. responding to taps
- This is called *Asynchronous Network Access*

```
NSURL *url = [NSURL URLWithString:@"http://google.com/"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[NSURLConnection connectionWithRequest:request delegate:self];
```

- In the line above connection's *delegate* is self
- Upon completion, a delegate method on self is called, telling us the request has finished

1.4 NSURLConnection Delegate

NSURLConnection will tell the delegate object -- "I received some data", or "I failed"*.

```
// Use this method to process the incoming data
- (void) connection:(NSURLConnection*)connection
     didReceiveData:(NSData*)data
// Use this method to deal with network failures, etc.
- (void) connection:(NSURLConnection*)connection
   didFailWithError:(NSError*)error
```

*** A full list of delegate methods are listed in the NSURLConnection reference:**
http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/Reference/Reference.html

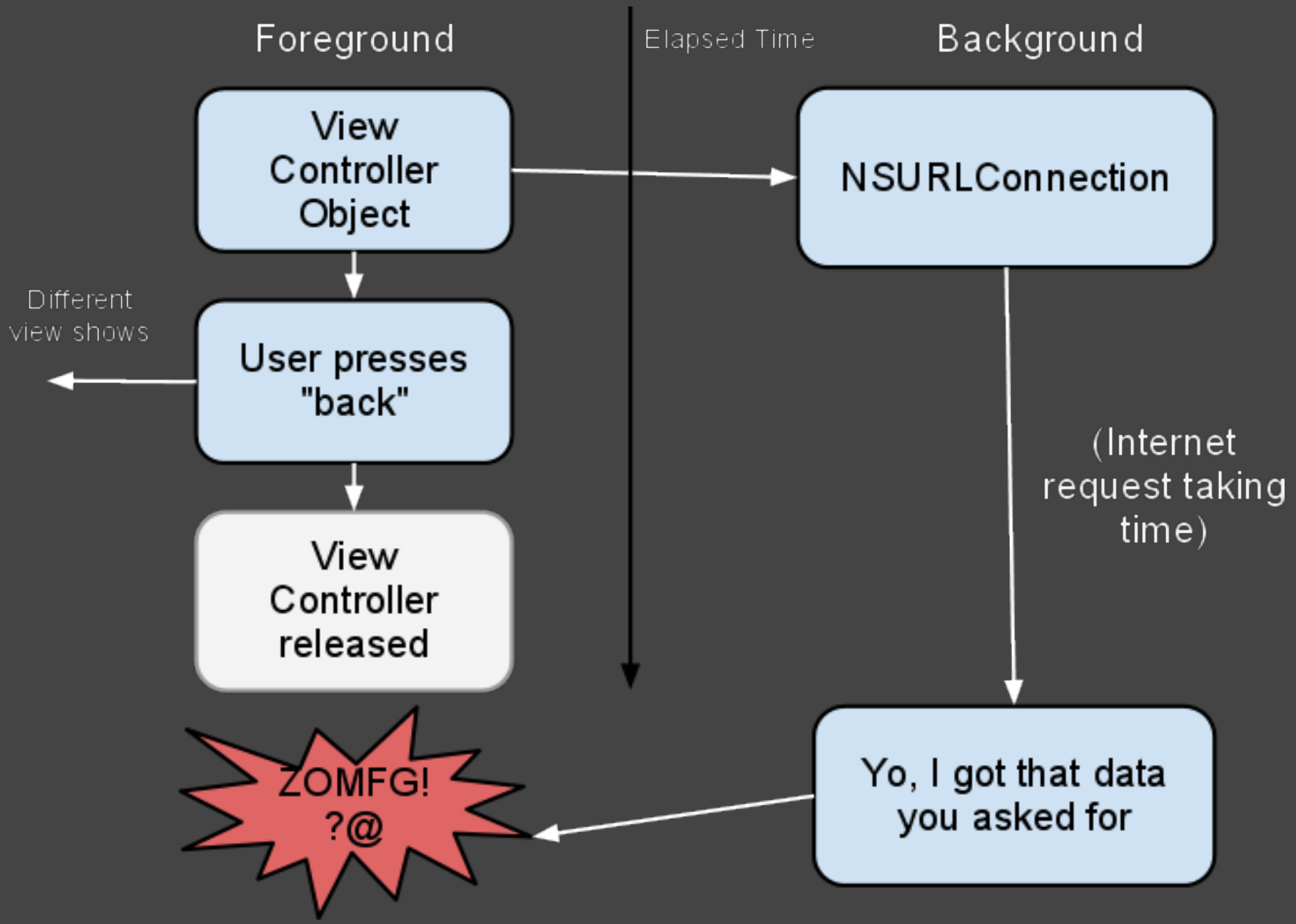
1.5 Multitasking Makes It More Complex

Two difficulties arise due to multitasking:

1. *Race Conditions* - the order in which code runs is not guaranteed

2. We don't know when we're done *until we're done*, so we have to manually assemble the data as it comes in.

1.6 Race to the Bottom

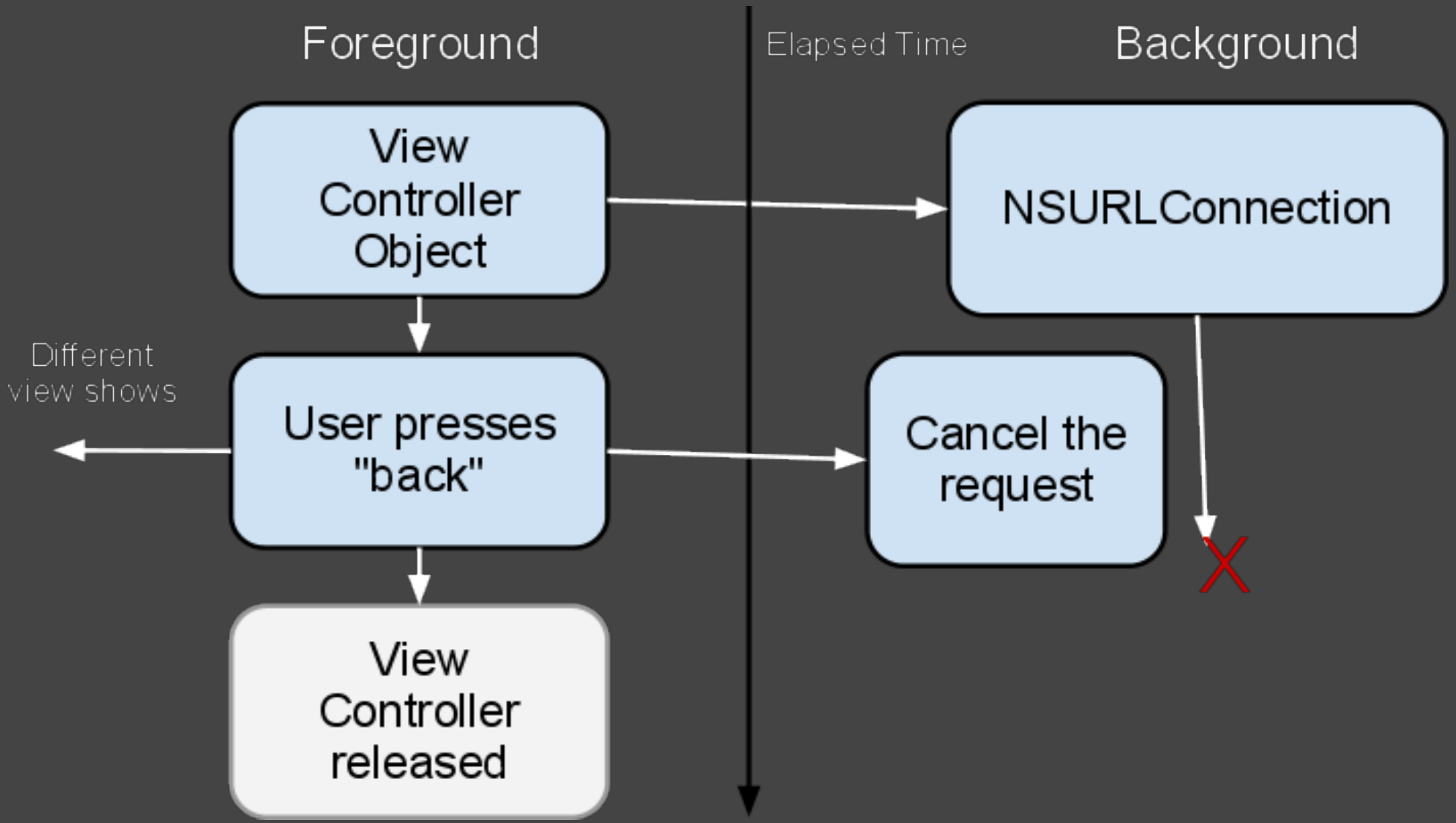


1.7 What's a Race Condition?

When different parts of your code run in a different order than expected, odd things can happen!

1. User launches request to retrieve URL
2. While NSURLConnection is in the background, the user taps a "Back" button on the app's UI
3. The app transitions to the previous screen
4. NSURLConnection finishes, only to crash because its delegate object, the previous screen's view controller, has been deallocated

1.8 Race Condition Fixed



1.9 Solving Race Conditions

You need to remember what might be happening in the background before changing state.

1. User launches request to retrieve URL.
2. While NSURLConnection is waiting in background, the user taps a "Back" button on the app's UI.
3. **The view controller of the current screen cancels the NSURLConnection request.**
4. The app transitions to the previous screen.
5. The connection does not call back because it's been cancelled.

1.10 Aggregating Data As It Comes In

```
// in .h header, also @synthesize'd in .m
@property (retain) NSMutableData *data;
// in .m implementation
- (void) connection: (NSURLConnection*)connection
     didReceiveData:(NSData*)newData {
  [self.data appendData:newData];
}

- (void) connectionDidFinishLoading:(NSURLConnection*)connection {
 // All data now in self.data, do something cool
}
```

1.11 Putting It All Together

- Use NSURLConnection to make URL requests in the background
- Implement its delegate methods so you can track response, failure, etc.
- Cancel the request if you're finished with it before it is finished (i.e. in -dealloc)

What's On For Today?

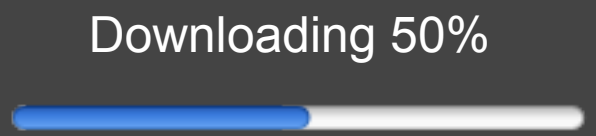
1. Simple Network Access
2. UI Concerns & Limiting Conditions
3. iOS App Lifecycle
4. Model-View-Controller (MVC)

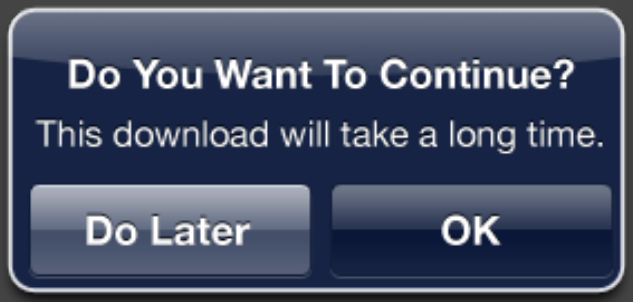
2.0 Thinking About User Experience

- Sometimes, you want to **"block"** the user until a task completes -- i.e. if it is vital to the app's basic function
- Must provide *visual feedback* - what, why, how long
- If possible, offer a cancel option

2.1 Visual Feedback Toolbox

- UILabel
- UIProgressBar
- UIActivityIndicator
- UIAlertView





2.2 User Feedback for "Blocking" Tasks

Any task that takes a non-trivial amount of time to complete (**>500ms on a device**) should have visual user feedback.

- Network access, file downloads
- Processing image/audio data (filters, etc)
- File I/O - large file copying
- Updating GPS coordinates

2.3 Implementing a "Modal Activity View"

```
// launch connection
conn = [NSURLConnection connectionWithRequest:request
                                     delegate:self];

// Create black semi-transparent view over our screen
// (Assume defined @property (retain) UIView *modalView)
self.modalView = [[UIView alloc]
                  initWithFrame:self.view.frame];
self.modalView.alpha = 0.5f;
self.modalView.backgroundColor = [UIColor blackColor];

// Put a label on our shaded black modal view
UILabel *myLabel = [[[UILabel alloc] init] autorelease];
myLabel.text = @"Please Wait";
[self.modalView addSubview:myLabel];

[self.view addSubview:self.modalView];
```

2.4 Removing the Modal View

After the request finishes, or fails, put your user back in control:

```
// Removes view from screen, release it
[self.modalView removeFromSuperview];
self.modalView = nil;
```

** This technique works because the modalView covers the view beneath entirely. Therefore it "absorbs" all touches and doesn't pass any events to the view below, even though it is still visible (i.e. due to the modalView being half-transparent).*

2.5 Limited Conditions

- Airplane mode, overseas roaming, no service, out of data on prepaid plan, bad weather, server is down, WiFi connection went to sleep... the list goes on.
- NEVER ASSUME you have a valid Internet connection.
- Same applies for GPS/location services.

2.6 Handling Limited Conditions

- Implement a "failure" handler for when network, GPS, or any other data service is unavailable
- Try first, and if at first you don't succeed, handle the failure and notify the user how to remedy

2.7 Network Failure Example

```
- (void) connection: (NSURLConnection*)connection
   didFailWithError:(NSError*)error {
  if (error.domain == NSURLErrorDomain) {
    UIAlertView *alert = [[UIAlertView alloc]
           initWithTitle:@"Network Problem"
                 message:error.localizedDescription
                delegate:nil
       cancelButtonTitle:nil
       otherButtonTitles:nil];
    [alert show];
    [alert release];
  }
}
```

Source:
http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Constants/Reference/reference.html

What's On For Today?

1. Simple Network Access
2. UI Concerns & Limiting Conditions
3. iOS App Lifecycle
4. Model-View-Controller (MVC)

3.0 The iOS App Lifecycle

Remember the Application Object from Lecture 3 (and its delegate, AppDelegate)?

```
- (void)applicationDidFinishLaunching:
  (UIApplication*)application {
    self.viewController = [[SomeViewController alloc]
                           initWithNibName:nil
                                    bundle:nil];
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}
```

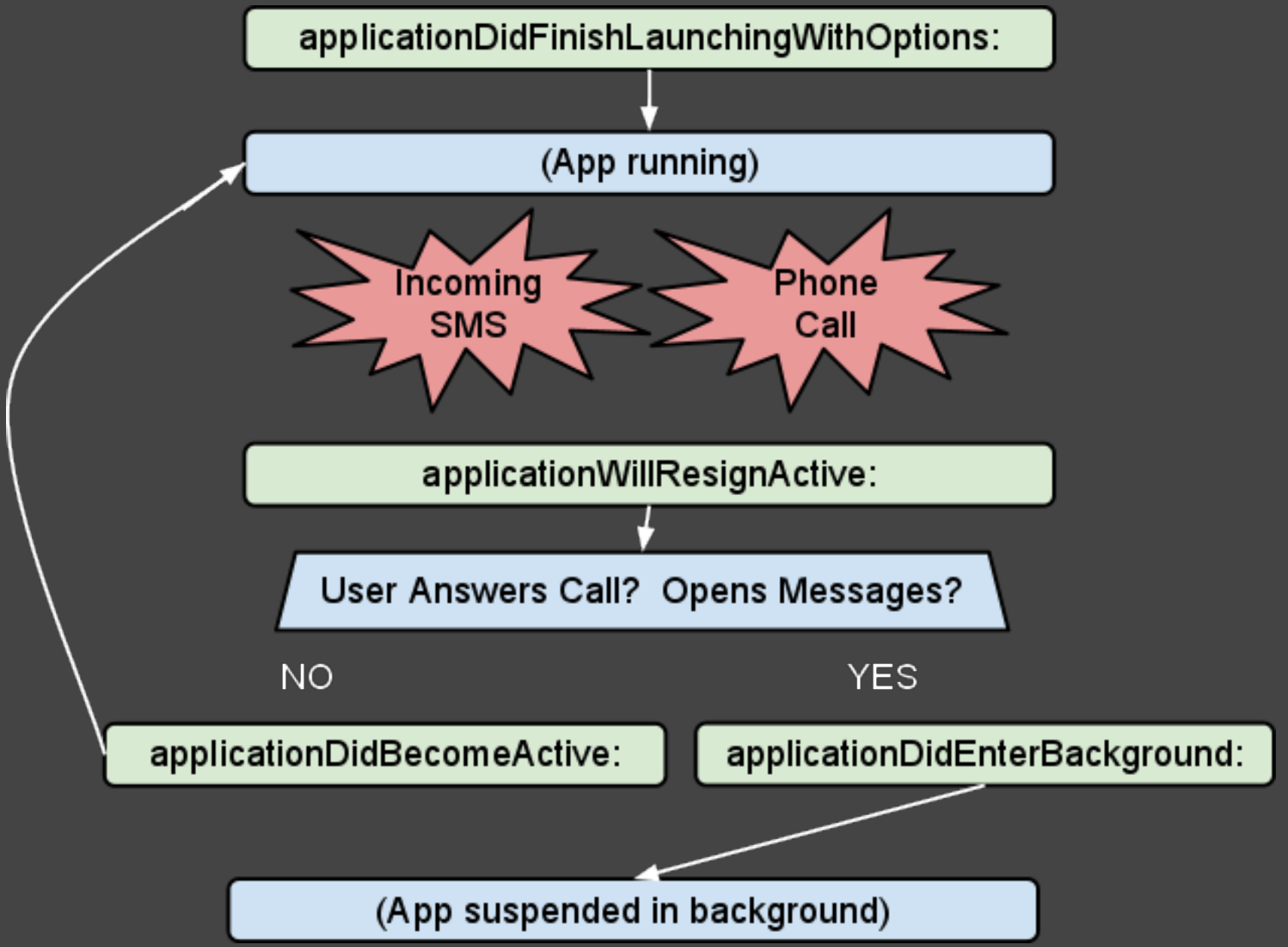
3.1 Application States

applicationDidFinishLaunching:withOptions: is just one of many ***Delegate Methods*** UIApplication calls on the delegate object as the app changes state.

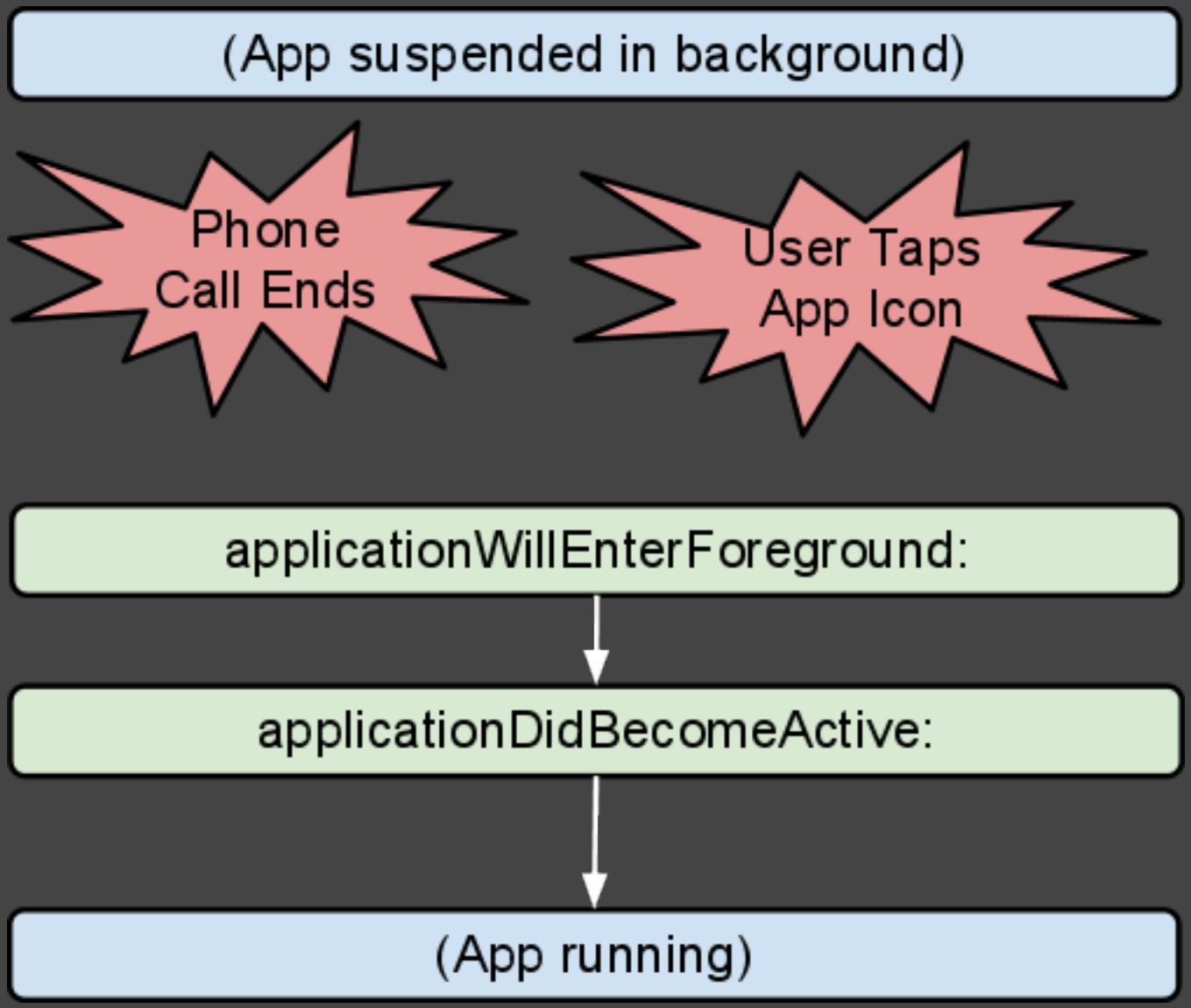
- (void) applicationDidBecomeActive:
- (void) applicationWillResignActive:
- (void) applicationDidEnterBackground:
- (void) applicationWillEnterForeground:

See: http://developer.apple.com/library/ios/#documentation/uikit/reference/UIApplicationDelegate_Protocol/Reference/Reference.html

3.2 App Lifecycle I



3.3 App Lifecycle II



3.4 Application State Notifications

In addition to UIApplicationDelegate calls, UIApplication will ***post notifications*** when each event occurs.

- UIApplicationWillEnterForeground
- UIApplicationDidEnterBackground
- UIApplicationWillResignActive
- UIApplicationDidBecomeActive

3.5 Your App in the Background

- Users also put your app into the background by pressing the Home button
- Unlike networking in the background (while your app is in the foreground), *you cannot run code while in the background**

* There are exceptions (streaming audio, VoIP, background GPS), but these are beyond the scope of this course.

3.6 Application State Transitions

- When going to the background:
  - Save any user data
  - Cancel any ongoing network connection
  - Stop timers, stop graphics drawing code
- Reverse as appropriate on enter foreground
- App becomes "inactive" when lock button pressed, "active" when unlocked again

Great reference & diagram:
http://www.cocoanetics.com/2010/07/understanding-ios-4-backgrounding-and-delegate-messaging/

3.7 Networking & Application State

- Even if the UI inside the app is "blocked", *users can still close your app any time*
- Objects that access the Internet should listen for the `UIApplicationDidEnterBackground` notification and act appropriately
- It is not ideal to restart a network connection after receiving `UIApplicationWillEnterForeground`

3.8 Abandoning a transfer

```
- (void) cancelTransfer {
  if (self.connection == nil) {
    return;  // if we're not active, no worries
  }
  // Assume NSURLConnection object is @property
  [self.connection cancel];
  self.connection.delegate = nil;

  // Get rid of our blocking view
  [self.modalView removeFromSuperview];
  self.modalView = nil;
}

// Somewhere else in the class, probably -init method
[[NSNotificationCenter defaultCenter]
   addObserver:self selector:@selector(cancelTransfer)
     name:UIApplicationDidEnterBackground object:nil];
```

What's On For Today?

1. Simple Network Access
2. UI Concerns & Limiting Conditions
3. iOS App Lifecycle
4. Model-View-Controller (MVC)

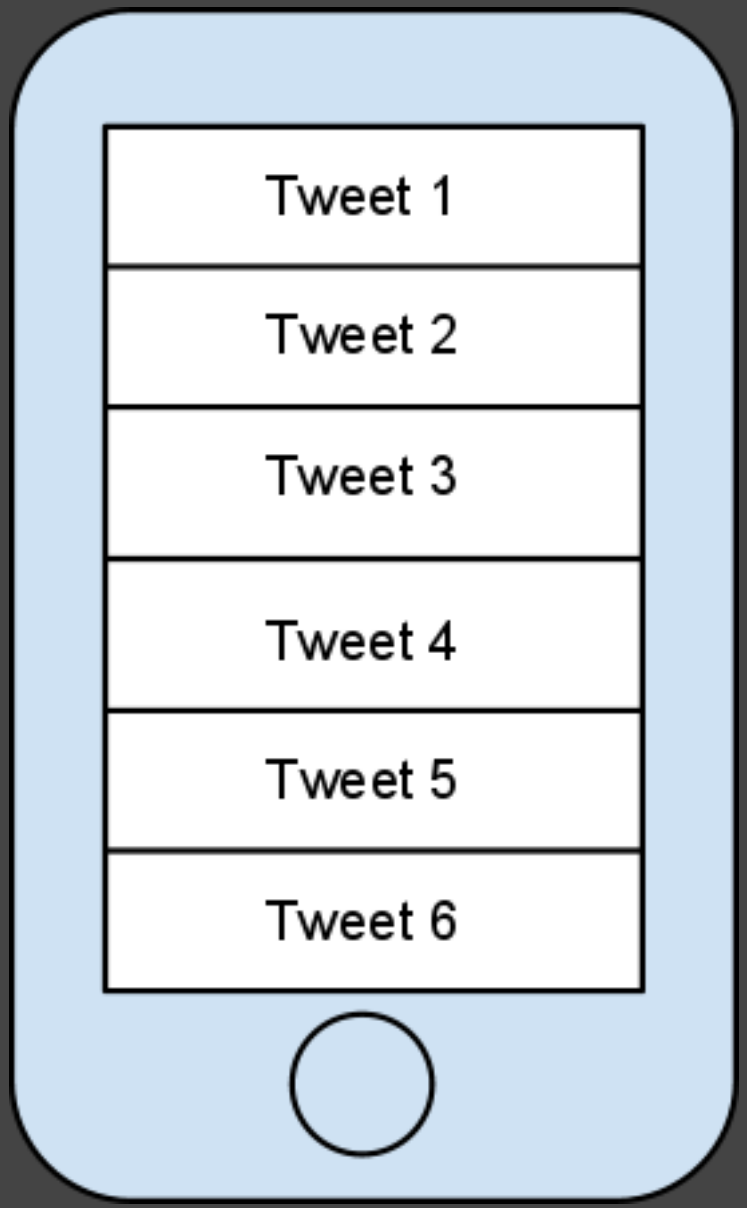
4.0 Networking & Model-View-Controller

- Any network request is likely to get **data** as part of your application's *data model*
- While it's possible to put networking code into a view controller class, it tightly couples your data to the view
- This makes it less reusable, more brittle
- What should we do?

4.1 Implementing a Twitter Client

- The world needs MORE Twitter apps
- So imagine we're writing a Twitter app for the UN that reads UN data feeds
- Let's step through how MVC will help us keep our code flexible

4.2 First Approach



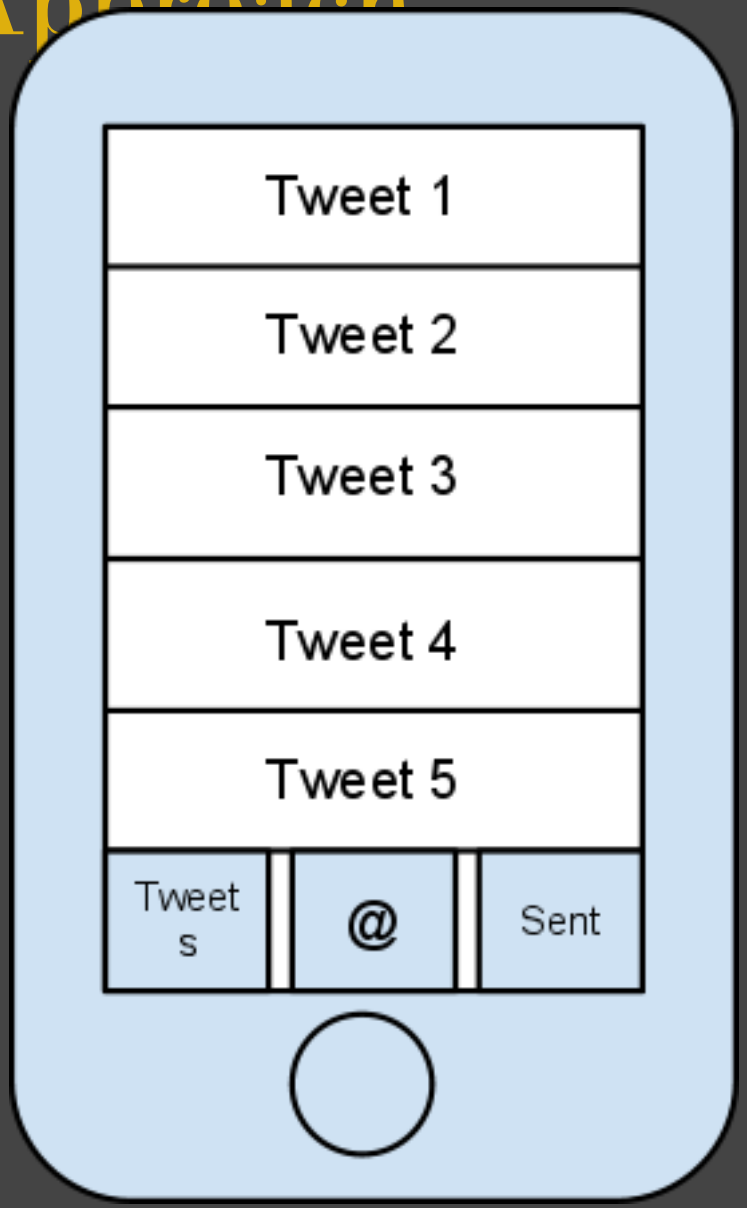
- Use UITableViewController subclass to display tweets out of an NSArray
- When app loads, a network request gets recent tweets
- When the network call is finished, an NSArray of tweets is populated and [self.tableView reloadData] is called

4.3 Additional Features

Our Twitter client works well, but now we have a new feature request:

> *"I want to have 3 displays: my timeline, my @mentions timeline, and my sent tweets timeline. Maybe we can implement a tab bar controller??"*

4.4 New Features using our First Approach



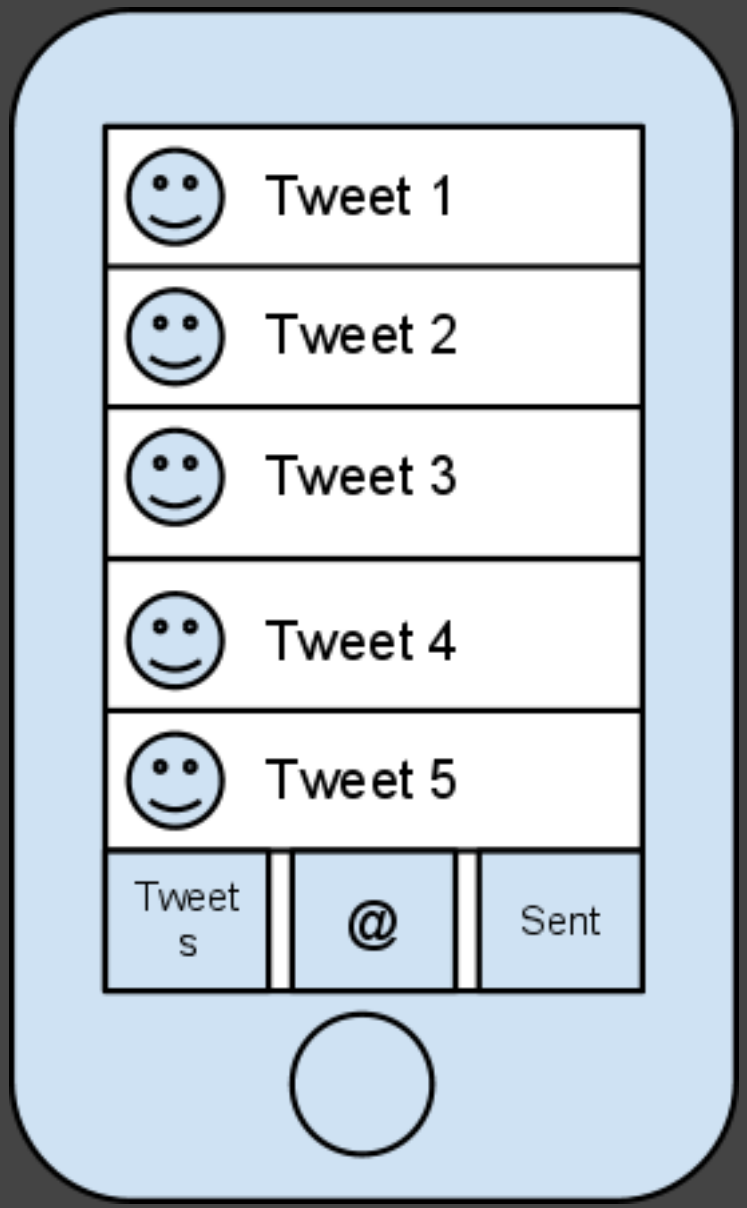
We have 3 UITableViewController subclasses, one for each screen with a different timeline, each reading slightly different Tweet data from the network when loaded

4.5 MORE Additional Features

Here we go again:

"Wouldn't it be cool to put each person's picture/avatar next to their tweets?"

4.6 MORE New Features



- Here's where our design begins to break down:

  We have to change *all 3* UITableViewController subclasses to add support for images!

  This is quickly becoming unmaintainable - duplicated code that is tightly-coupled

4.7 The MVC Approach

The view controller *controls* the environment -- it shouldn't worry about whether the tweets it is displaying are mentions, sent tweets or a plain timeline - they are all tweets!

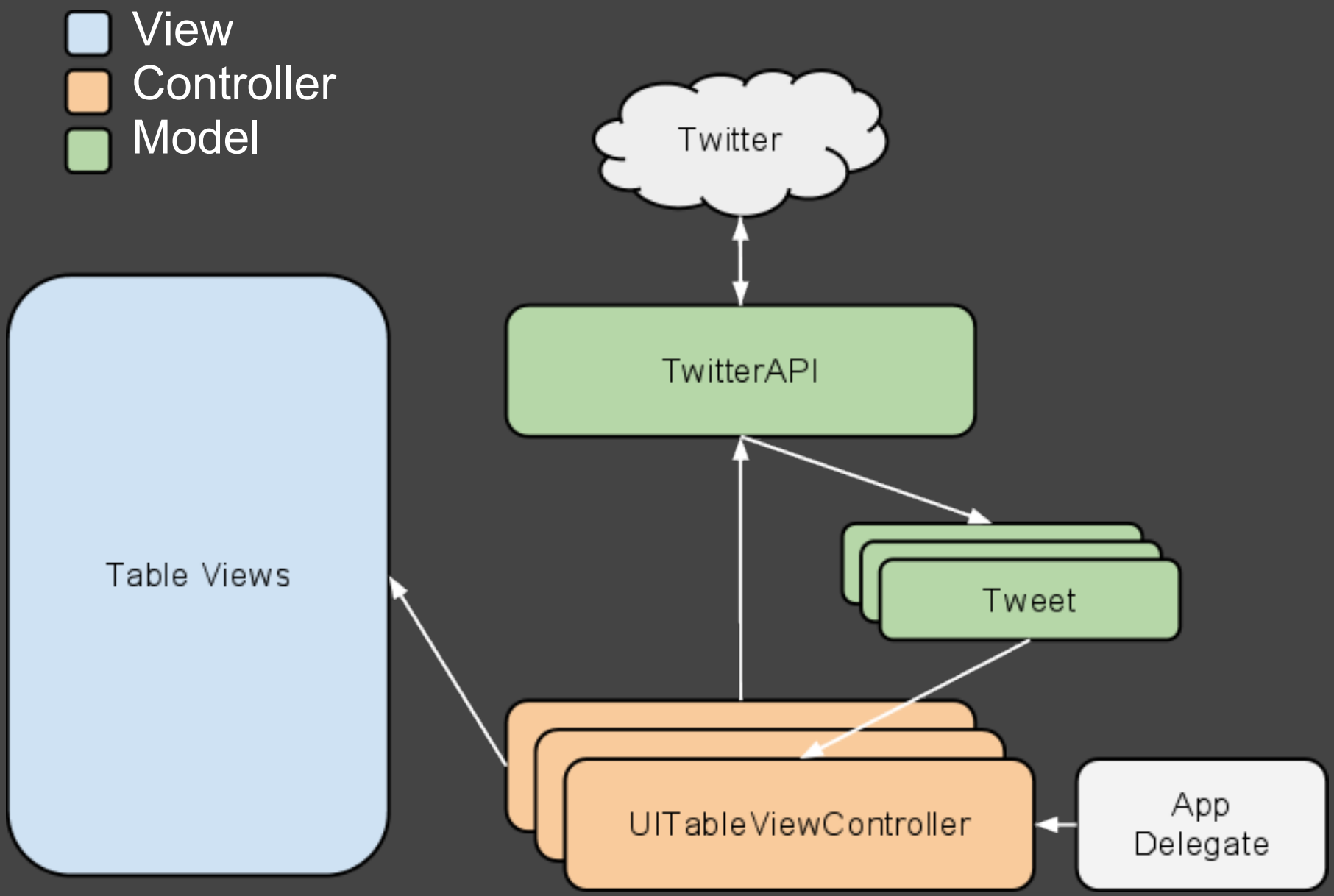
We need a *data model* object to represent a single tweet - our UITableViewController can handle those.

4.7 The MVC Approach Class Design

As a general idea, the view controller should worry about program flow and *control*, not what type of tweet we are displaying

We need a *data model* object to represent a single tweet - and a UITableViewController can handle those without caring about anything else

4.8 The MVC Approach Class Design II



4.9 iOS MVC Summary

- In iOS, *views* are mostly inside XIB files
- The key point is to keep *data models* and *controllers* as separate as is possible, usually as different classes
- It is easy to have a *view controller* do too many tasks

What We Covered Today

1. Simple Networking
2. UI Concerns & Limiting Conditions
3. iOS App Lifecycle
4. Model-View-Controller (MVC)

End of Lecture 6

1. Lab
2. Assignments
3. Kaboom