

iOS Dev Accelerator

Week 2 Day 2

- Resizing Images
- Parse
- .gitignore
- UITabBarController
- Stack Data structure

Resizing images

Resizing an Image

- Par for the course with programming, there are a number of different ways to resize an image in iOS
- NSHipster did a great article on a bunch of the different ways, and which way is the fastest.
- Here is the fastest:

```
var size = CGSize(width: 100, height: 100)
UIGraphicsBeginImageContext(size)
originalImage!.drawInRect(CGRect(x: 0, y: 0, width: 100, height: 100))
let thumbnail = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

1. UIGraphicsBeginImageContext() creates bitmap-based graphics context for you to draw in and makes it the 'current context'
2. drawInRect is a method on UIImage which draws the image in the specified rectangle, in the current context, and scales if needed
3. UIGraphicsGetImageFromCurrentImageContext() just pulls the currently drawn image from the context and returns a UIImage

Demo



Parse

Backend for your apps

Parse

- Parse is a mobile app platform that allows you to easily setup and work with a backend hosted with Parse
- The main benefit to you, as an iOS Developer, is that you don't need to know/learn a backend language in order to have a backend.
- Parse's goal is to completely eliminate the need to write any server code for your mobile apps
- The iOS Parse SDK contains all of the classes and resources our apps need to in order to communicate with Parse

Setting up Parse For your App

- First Create a free account at [parse.com](https://www.parse.com)
- Then download the latest Parse iOS Libraries/frameworks:
 - <https://www.parse.com/downloads/ios/parse-library/latest>
- In your Xcode project, create a blank objective-C file. When it asks you if you want to create an Objective-C bridging header file, say yes.

Setting up Parse For your App cont.

- Delete the blank .m objective-c file it created, and then in in your bridging header put the following line:
 - `#import <Parse/Parse.h>`
- Go to the build phases of your app's target and add the following to Link Binary with Libraries....

Things to add to your build phases – Link Binary with Libraries

- `libsqlite3.0.dylib`
- `Foundation.framework`
- `SystemConfiguration.framework`
- `StoreKit.framework`
- `Security.framework`
- `QuartzCore.framework`
- `MobileCoreServices.framework`
- `libz.dylib`
- `CoreLocation.framework`
- `CoreGraphics.framework`
- `AudioToolbox.framework`
- `Bolts.framework`
- `ParseCrashReporting.framework`
- `ParseFacebookUtils.framework`
- `ParseUI.framework`
- `Parse.framework`

To test it out

- In your swift project, add the following code to your app delegate's didFinishLaunchingWithOptions method:

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject:
    AnyObject]?) -> Bool {
    // Override point for customization after application
    launch.

    Parse.setApplicationId("1Xm13mKcLSwNebyeawAZfV7t0owRvJjb0c3
    j5c1v", clientKey:
    "24wzBQfezekydGWEvZY4bMM6qsaCcST2Nc70qoBd")
    let testObject = PFObject(className: "TestObject")
    testObject["foo"] = "bar"
    testObject.save()

    return true
}
```

ApplicationID and clientKey should match those created when you created your first app on Parse

To test it out

- Once you have done that, you should be able to log onto your parse dashboard and see instance of TestObject being uploaded, with the value bar set for the field foo:

Data

TestObject2

+ Add Class

⬆ Import

Cloud Code

Jobs

Logs

Config

+ Row

- Row

+ Col

Security

More ▼

<input type="checkbox"/>	objectId Str...	foo String	createdAt Date ▼	updatedAt Date	ACL ACL	
<input type="checkbox"/>	4SWD8R0ra4	bar	Mar 02, 2015, 20:49	Mar 02, 2015, 20:49	Public Read and Write	
<input type="checkbox"/>	Cio1G9GyWy	bar	Mar 02, 2015, 20:43	Mar 02, 2015, 20:43	Public Read and Write	

Demo

Parse Apps

- On Parse, you will need to create a separate Parse app for every separate mobile app you are going to create that needs a Parse backend.
- Each one of your Parse apps has its own Application ID and client key
- That's how your Xcode project knows which parse app to upload and download data to and from


```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject:
    AnyObject]?) -> Bool {
    // Override point for customization after application
    launch.

    Parse.setApplicationId("1Xm13mKcLSwNebyeawAZfV7t0owRvJjb0c3
    j5c1v", clientKey:
    "24wzBQfezekydGWEvZY4bMM6qsaCcST2Nc70qoBd")
    let testObject = PFObject(className: "TestObject")
    testObject["foo"] = "bar"
    testObject.save()

    return true
}
```

Parse Objects

- Storing and retrieving data on Parse is setup around the class `PFObject`.
- Each `PFObject` contains key value pairings (like a dictionary/JSON) of data
- The data in `PFObjects` is schemaless, which means you don't have to setup schema before you start uploading data. As you upload new key-value pairings, Parse will automatically make a column for it in the table
- Keys must be alphanumeric strings

PFObject

```
let team = PFObject(className: "Team")  
team["name"] = "Seahawks"  
team["location"] = "Seattle"  
team["league"] = "NFL"
```

So the data of this object looks like:
name: "Seahawks", location: "Seattle", league: "NFL"

Demo

Saving your Objects

- So once you have created some data you want to upload to your backend, you just need to call `save` on your objects
- You can call `save` on your objects by simply calling `saveInBackgroundWithBlock()`
- The method has one parameter, which is a closure that acts as a callback once the save has been performed by the server or if it timed out.

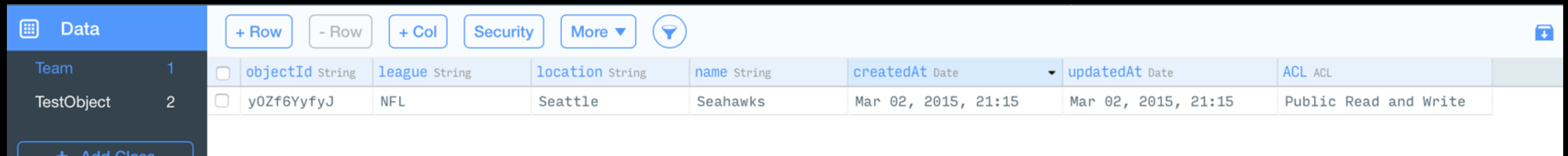
Saving your Objects

```
let team = PFObject(className: "Team")
team["name"] = "Seahawks"
team["location"] = "Seattle"
team["league"] = "NFL"

team.saveInBackgroundWithBlock { (finished, error) -> Void
    in
        if error != nil {
            println("Error during save: %@",error.
                localizedDescription)
        }
    }
}
```

Saving your Objets

- Whats cool about that save is no where did we have to define the scheme for the Team class.
- Parse lazily sets all this stuff up once it sees we are uploading a new class to the backend.



The screenshot shows the Parse dashboard interface. On the left, a sidebar lists 'Team' (1 object) and 'TestObject' (2 objects). The main area displays a table of objects for the 'Team' class. The table has columns for objectId, league, location, name, createdAt, updatedAt, and ACL. A single row is visible with the following data: objectId: y0Zf6YyfyJ, league: NFL, location: Seattle, name: Seahawks, createdAt: Mar 02, 2015, 21:15, updatedAt: Mar 02, 2015, 21:15, and ACL: Public Read and Write.

	objectId String	league String	location String	name String	createdAt Date	updatedAt Date	ACL ACL
<input type="checkbox"/>	y0Zf6YyfyJ	NFL	Seattle	Seahawks	Mar 02, 2015, 21:15	Mar 02, 2015, 21:15	Public Read and Write

- Also take note that there are a few fields that we didn't create explicitly in our code, Parse generates us for these.
- Of particular importance is the objectId field, which is how parse recognizes each individual record in this table. This is guaranteed to be unique.

Demo

Fetching your objects

- So we know how to create and save our objects, but how do we fetch them from the cloud?
- We do this using the PFQuery class.
- The PFQuery class offers many different ways to fetch objects from your backend.
- Lets look at a few of the most common and powerful ones

Fetching an object with an ID

- If you know the objectId of the object you are trying to fetch, use getObjectInBackgroundWithId:block:() to fetch a single object back

```
let query = PFQuery(className: "Team")

query.getObjectInBackgroundWithId("y0Zf6YyfyJ", block: {
    (teamObject, error) -> Void in

    let name = teamObject["name"] as! String

    println("the name of the team we fetched is \
(teamObject)")
})
```

Fetching a group of objects

- You can run a query against an entire class of objects, without an id, by using the `findObjectsInBackgroundWithBlock()` method

```
let query = PFQuery(className: "Team")

query.findObjectsInBackgroundWithBlock { (results, error) -
    > Void in
    println(results.count)
    for object in results {
        //do something for each object
    }
}
```

Fetching a group of objects

- So the previous query will return ALL the objects (up to a limit of 1000 per query, default is 100) in that class. What if we want to be more specific? We can attach filters to our queries:

```
let query = PFQuery(className: "Team")
query.whereKey("name", containsString: "Sea")
query.findObjectsInBackgroundWithBlock { (results, error) -
    > Void in
    println(results.count)
    for object in results {
        //do something for each object
    }
}
```


Query filters

- Theres a ton!

```
Self! whereKey(key: String!, containedIn: [AnyObject]!)
Self! whereKey(key: String!, containsAllObjectsInArray: [AnyObject]!)
Self! whereKey(key: String!, containsString: String!)
Self! whereKey(key: String!, doesNotMatchKey: String!, inQuery: PFQuery!)
Self! whereKey(key: String!, doesNotMatchQuery: PFQuery!)
Self! whereKey(key: String!, equalTo: AnyObject!)
Self! whereKey(key: String!, greaterThan: AnyObject!)
Self! whereKey(key: String!, greaterThanOrEqualTo: AnyObject!)
```

- While its recommended you use these methods, you can use NSPredicate class if you are coming in with some CoreData knowledge, of if you need to do some more advanced fetching.

Demo

Saving files to your backend

- The PFFile class lets you store files in the cloud that would be too large to put inside regular PFObjects.
- PFFile is needed for things like images, videos, music, any other binary data (up to 10mb at a time)
- There is a pretty simple workflow you follow to upload your files to the cloud

PFFiles workflow

- First thing is to convert the file you are trying to upload (like a UIImage) into NSData.
- And then create a PFFile with the data:

```
let image = UIImage(named: "MyImage")  
let imageData = UIImagePNGRepresentation(image)  
let file = PFFile(name: "test.png", data: imageData)
```

The name parameter of the PFFile does not have to be unique server side, but the file type is required so the parse server knows how to handle the file

PFFiles workflow part 2

- You then need to associate the PFFile with a PFObject, and then save it to the cloud

```
let image = UIImage(named: "seahawks")
let imageData = UIImagePNGRepresentation(image)
let file = PFFile(name: "test.png", data: imageData)

let imagePost = PFObject(className: "ImagePost")
imagePost["image"] = file
imagePost["title"] = "Cray Cray"
imagePost.saveInBackgroundWithBlock { (succeeded, error) ->
    Void in
    println("save completed!")
}
```

Demo

Get that file back from the cloud

- To get back files, like getting back an image to show, you first need to run a regular PFQuery to return the PFObject(s) that contain the data you need
- You then need to access that particular field on the PFObject that contains the data, creating a PFFile to hold the reference
- And finally call getDataInBackground() to download the actual data. The data doesn't come down when you do your PFQuery. That would make PFQuerys very slow.

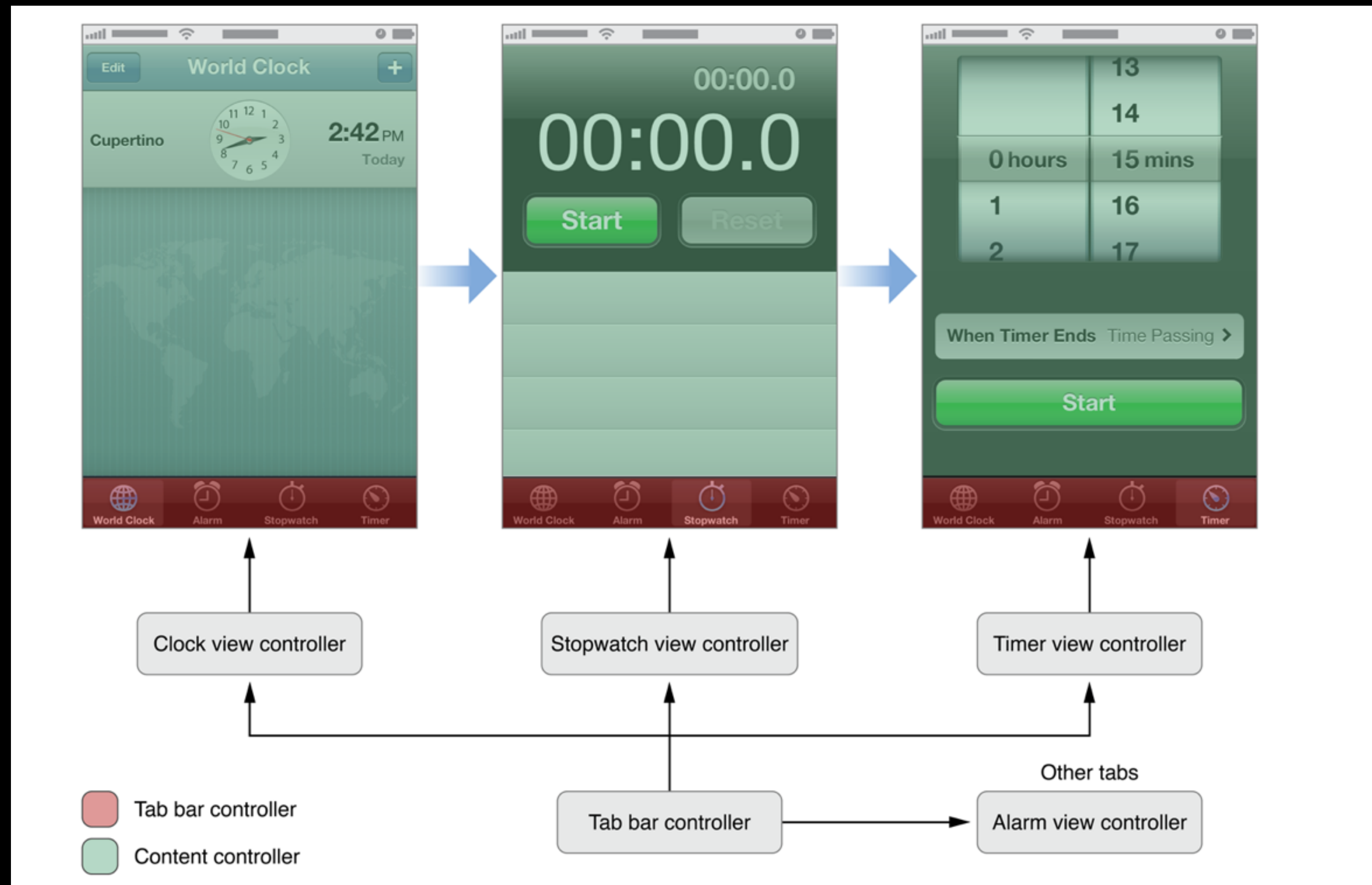
Get that file back from the cloud

```
let imageFile = myPost["image"] as! PFFile

imageFile.getDataInBackgroundWithBlock { (data, error) ->
    Void in

    if error == nil {
        let image = UIImage(data: data)
    }
}
```

Demo



Tab Bar Controllers

Tab Bar Controllers

- A Tab bar controller is a container view controller that you use to divide your app into two or more **distinct modes of operations**.
- The tab bar has multiple tabs, each representing a child view controller.
- Selecting a tab causes the tab bar controller to display the associated view controller's view on screen.
- In general, you should use a tab bar if your app displays different types of data to the user, or displays the data in different ways

Getting a tab bar controller into your app

- You can instantiate a `UITabBarController` in code with a regular initializer.
- Or the easier way, embed them via storyboard, just like you do with a navigation controller

Demo

Tab Bar Controller facts

- Everything UIViewController has a property called tabBarController, which points to the tab bar controller it is in, IF it is actually in a tab bar controller (just like the navigationController property)
- UITabBarController has a property called viewControllers which is an array of the view controllers it is managing. They are ordered from left to right.
- By default, the label for each 'item' in the tab bar is the title of the view controller it contains.

UITabBarItem

- Every view controller has a UITabBarItem property called tabBarItem
- This item dictates how the view controller is represented in the tab bar
- If you want to provide custom images as the tab icons, you use the UITabBarItem initializer that takes in a title, image and selectedImage.
- Creating custom icons for the tab bar sucks. It has to be only alpha channels. See Apple's HIG for a guidelines on creating icons
- I will provide you custom icons for this project. You can buy icons specifically designed for the tab bar from many different icon websites as well.

Demo

.gitignore

- Often times you will have a set of files which git shouldn't track.
- These are generally automatically generated logs, file system files, large SDKs or resources, or even secret API keys.
- In these cases, you can create a file called .gitignore that contains a list of the files or file patterns you want ignored by git.
- In this .gitignore file, you can list specific files, directories, or specific file types you want ignored.
- Use a # for comment lines
- github has a master list of best practice .gitignores for many programming languages.

.gitignore workflow

- Change directory in terminal to the root directory of your project (where the xcodeproj file lives)
- run the command `touch .gitignore`
- run `open .gitignore`
- paste in the list of files/directories to ignore and save the file
- add the .gitignore file (add it individually instead of with `git add -A`)
- Once the .gitignore is added, git will ignore any files listed in the .gitignore.
- It is important that adding the .gitignore be the first file you add to your git repository.

Demo

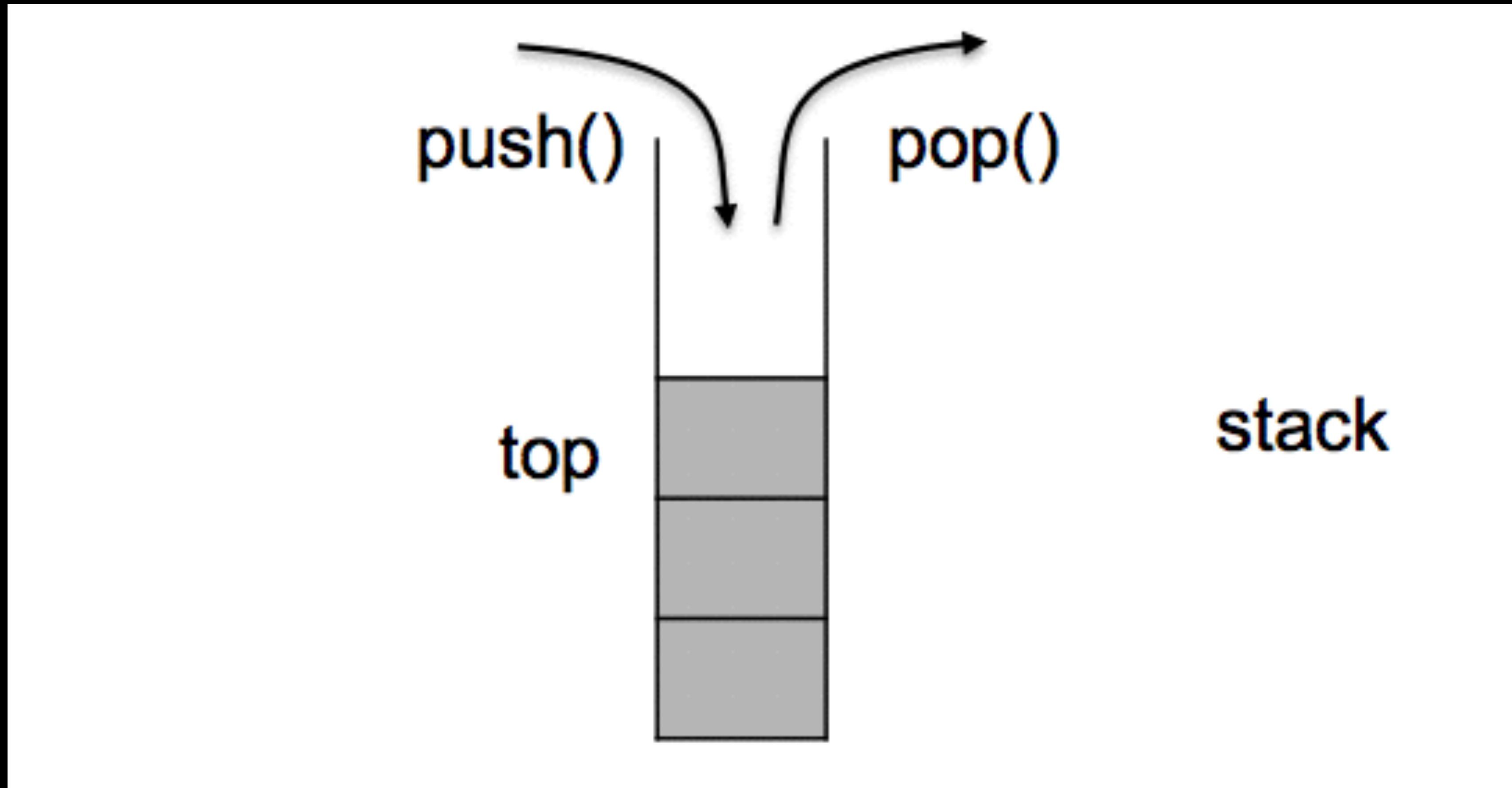
Stack Data structure

- “The stack data structure is one of the most important data structures in computer science” -Wikipedia (so you know its legit)
- Used in navigation controllers!
- And also in pretty much every programming language ever.

Stack Metaphor

- To understand a stack, think of a deck of playing cards that is face down.
- We can easily access the card that is on top.
- There are 2 things we can do to access the card on top:
 - peek at it, but leave it on the deck
 - pop it off the deck. When we pop something off a stack, we are taking it off the stack
- When you want to put something onto the stack, we call it pushing onto the stack
- A stack is considered LIFO. Last in, first out. This means the last thing we added to the deck (pushed) is the first thing that gets taken off (popped)

Stack Visual



Call stack

- “In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program.”
- In most high level programming languages, the implementation of the call stack is abstracted away for us, but its still valuable to know what it is and how it works.
- When a function/method is called, it is pushed onto the stack. Any local variables are created and stored on the stack as well. If the called function calls another function, that 2nd function is pushed onto the stack as well. This keeps happening until all the functions have returned and they are all popped off the stack.

call stack demo

Implementing a stack

- There are two easy ways to implement a stack: using an array and using a linked list.
- Since we haven't learned linked lists yet, we will focus on the array way for now.

stack implementation demo