iOS Dev Accelerator
Week 7 Day 3

- Memory Management & Dog Leashes
- App Distribution
- Multiple Storyboards
- NSURLCache

Memory Management

No animals were hurt during the making of these slides

Memory Management

- “Application memory management is the process of allocating memory during your program’s runtime, using it, and freeing it when you are done with it”
- Objective-C has two methods of application memory management:
  - Manual retain-release, or MRR, allows you to explicitly manage memory by keeping track of objects you own using reference counting.
  - Automatic Reference Counting, or ARC, uses the same reference counting that MRR does, but it inserts the appropriate memory management method calls for you at compile-time. Swift uses ARC as well.
- **Understanding the specifics of a languages memory management system is critical to effectively learning the language.**

Memory Problems

- There are two general problems that good memory practices will help you avoid:
  - Freeing or overwriting data that is still in use, causing corruption and crashes (dangling pointer)
  - Not freeing data that is no longer in use, which is a memory leak. Leaks needlessly increase the memory footprint of your app, often times to levels that will cause your app to be terminated.
- The main difference between ARC and a garbage collection environment is that ARC does not automatically break retain cycles for you

Disabling ARC

- To disable ARC for your entire project, go to the Build Settings of your target, scroll down until you see ‘Apple LLVM 6.0 - Language - Objective-C’ and set Automatic Reference Counting to No
- To disable ARC for an individual file, go to the Build Phases of your target, look in Compile Sources, and click to the right of the name of the file you want to disable ARC. enter in “-fno-objc-arc” without the quotes.

Demo

Reference Counting

- Objective-C uses reference counting for its memory management system.
- EVERY object has a reference counter that is increased or decreased
- When you want to keep an object alive, you can increase its reference counter, or retain count.
- When you are no longer interested in keeping that object alive, you can decrease the retain count.
- An object with a retain count of 0 is free to be destroyed by the system.

Object Ownership

- Memory management is modeled after the concept of object ownership.
- An object can have many owners.
- As long as an object has one owner, it is kept alive.
- If an object has no owners, it is destroyed by the run time.

Being the owner

- An object is always created with a retain count of 1
- So anytime you instant an object using alloc, new, copy, or mutable copy, the retain count is set to 1. Which means you can say you ‘own’ the object.
- **If a method you called to create an object doesn't contain those words, you don't own that object.**
- You can also ‘own’ any object by calling retain on it, which increases the retain count by 1

```
- (NSString *)fullName {
    NSString *string = [NSString stringWithFormat:@"%@ %@",
                                self.firstName, self.lastName];
    return string;
}
```

We don’t own string because we didn't use alloc,new,copy

```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
```

We own aPerson because we used alloc

Relinquishing ownership

- When you are no longer interested in keeping the object alive, you can call release on the object
- this decreases the object's retain count by 1.
- if the object's retain count is now 0 after your release, it is free to be destroyed by the system

```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
    // ...
    [aPerson release];
}
```

Autorelease

- You can use autorelease when you need to send a deferred release message.
- Typically when returning an object from a method.
- You need autorelease because regular release would possibly release the object before you could return it from the method:

```
- (NSString *)fullName {
    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
                                        self.firstName, self.lastName] autorelease];
    return string;
}
```

This gives the object that sent this message a chance to retain this object before it is released

Retain count

- Every object has a retain count property.
- **Never use this property**
- **You should never need to manually access an object's retain count; The value returned is not accurate. Don't do it.**

Dealloc

- NSObject provides a method called dealloc.
- Dealloc is invoked automatically when an object has no owners and its memory is reclaimed.
- The role of dealloc is to free the objects own memory, and to dispose of any other objects or resources its holding onto, like properties.
- **You never call dealloc directly, the system calls it for you.**

Dealloc Example

```
@interface Person : NSObject
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

Accessor methods and MM

- When your class has a property thats an object, you need to make sure that object isn't released while your class is still using it.
- You need to claim ownership for this. And then you need to relinquish ownership when appropriate.
- Harness the power of accessor methods to make this simple and easy.

Accessor methods and MM

```
- (NSNumber *)count {
    return _count;
}
```

getter can just return the object

```
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}
```

setter must release the old object and retain the new

Using the retain attribute on a property does this for you automatically but obviously if you override the setter, you need to do it yourself

When not to use Accessor Methods

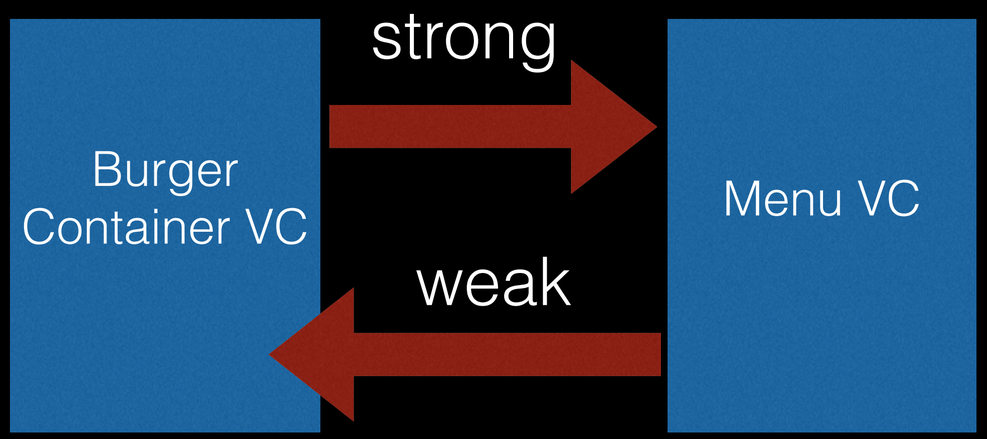
- Don’t use the accessor methods in your init methods and dealloc. This eliminates the chance of a bug related to a setter executing or accessing code that hasn't been setup yet.

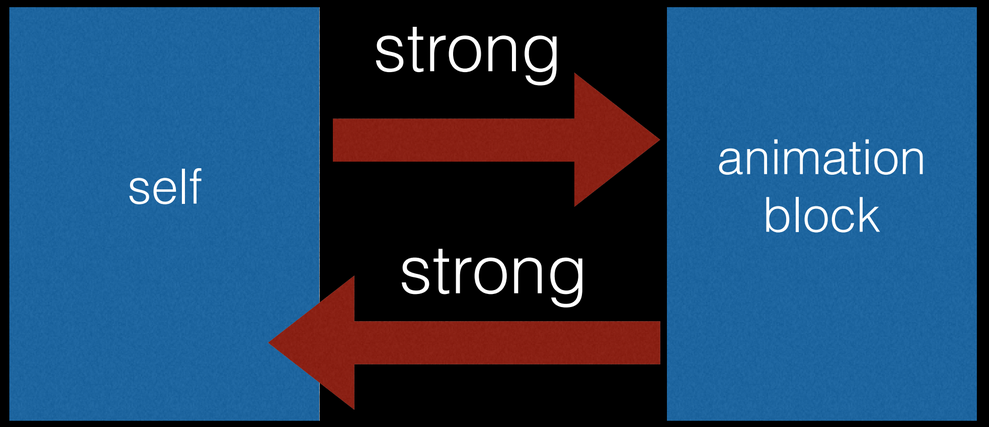
```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}
```

```
- (void)dealloc {
    [_count release];
    [super dealloc];
}
```

Retain Cycles

- Retaining an object creates a strong reference to that object.
- An object cannot be dealloc'd until all of its strong references are released.
- A retain cycle happens when two objects have strong references to each other.


strong
Burger
Container VC
Menu VC
weak


strong
self
animation
block
strong

Use weak references to avoid retain cycles

- A weak reference is a non-owning relationship where the source object does not retain the object to which it has a reference.
- Cocoa conventions dictate that parent objects should have strong references to their children, and children have weak references to their parents.
- Examples of weak/assign references in Cocoa: data sources, delegates, and notification observers.
- Thats why you need to specifically unregister for notifications when your object is about to be released, since the notification center only keeps **assign** references to objects who sign up for notifications. If you didn't unregistered, the notification center may try to deliver a notification to a released object. CRASH
- **Whenever you can, use weak. Weak properties will automatically be set to nil when the object is set to nil. Assign doesn't do this!**

AutoRelease Pool Blocks

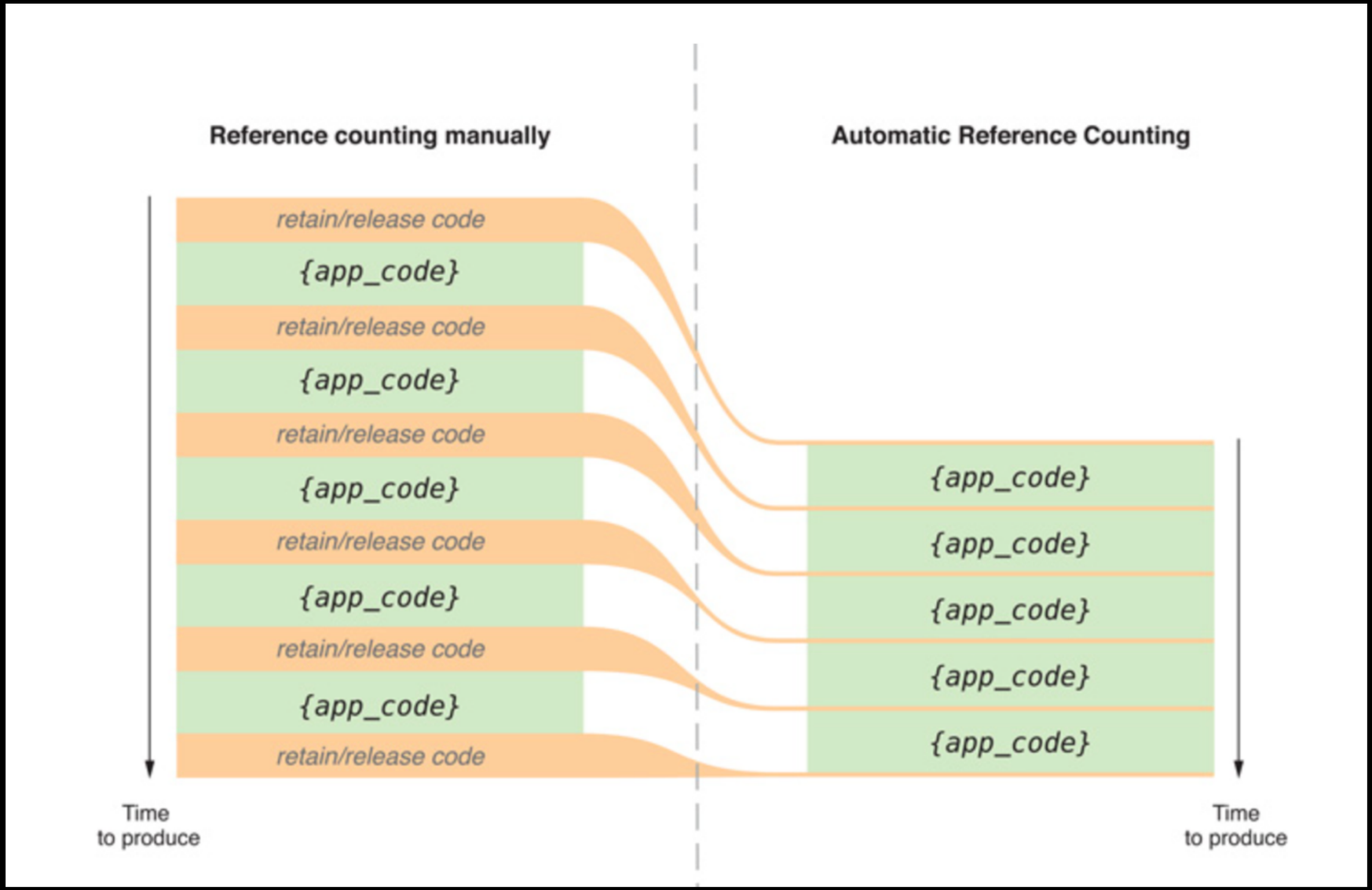
- “Autorelease pool blocks provide a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately (such as when you return an object from a method)”
- You usually don't need to create your own.
- 3 common scenarios when you create one:
  - If you are writing a command-line tool.
  - You write a loop that creates many temporary objects.
  - If you spawn a secondary thread.

AutoRelease Pool Blocks

```
for (NSURL *url in urls) {

    @autoreleasepool {
        NSError *error;
        NSString *fileContents = [NSString stringWithContentsOfURL:url
                                          encoding:NSUTF8StringEncoding error:&error];
        /* Process the string, creating and autoreleasing more objects. */
    }
}
```

ARC


Reference counting manually
Automatic Reference Counting
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
{app_code}
{app_code}
{app_code}
{app_code}
{app_code}
Time to produce
Time to produce

Keeping objects alive while in use

- Objects can be released at any time.
- Like when an object's parent is released, and the parent was the only owner of the child, the child is released too.
- Collections own the objects they contain. When an object is removed from a collection like an array or dictionary, if the collection was the only owner then the object is released.
- So when you really need to keep an object alive while in use, call retain on it, execute your code, and then release it.

Automatic Reference Counting (ARC)

- ARC enforces new rules: You cannot invoke dealloc, retain,release,retainCount, or autorelease.
- ARC introduced weak and strong property attributes. Strong is the default.
- You can explicitly create weak references with __weak qualifier while declaring a variable.
- Compiler will insert appropriate memory management calls for you.

ARC

```
// The following declaration is a synonym for: @property(retain) MyClass *myObject;
@property(strong) MyClass *myObject;

// The following declaration is similar to "@property(assign) MyClass *myObject;"
// except that if the MyClass instance is deallocated,
// the property value is set to nil instead of remaining as a dangling pointer.
@property(weak) MyClass *myObject;
```

Weak self and blocks

- When referring to self inside a block (or closure), best practice is to create a weak reference in self before the block and use that reference instead of self.
- Since blocks and closures capture strong references to all objects accessed in their bodies, if the object represented by self were to get a strong pointer to the block or closure, then we would have a retain cycle.

Blocks and __weak self

- Be careful accessing self inside a block, or you could create a retain cycle
- Use __weak self and __strong self to avoid this
- You may see this referred to as the @weakify pattern

http://aceontech.com/objc/ios/2014/01/10/weakify-a-more-elegant-solution-to-weakself.html

Blocks and __weak self

```
// Create a weak reference to self
__weak typeof(self)weakSelf = self;

[self.context performBlock:^{
    // Create a strong reference to self, based on the previous weak reference.
    // This prevents a direct strong reference so we don't get
    // into a retain cycle to self.
    // Also prevents self from becoming nil half-way through execution.

    __strong typeof(weakSelf)strongSelf = weakSelf;

    // Do something else

    NSError *error;
    [strongSelf.context save:&error];

    // Do something else
}];
```

App Distribution

Bundle ID

- Your Bundle ID is how both Apple and your device recognizes your app.
- Your app's Bundle Identifier must be unique to be registered with Apple
- Usually written out in reverse DNS notation (ie com.myCompany.myApp)
- The Bundle ID you have set in your App's Xcode project MUST match the Bundle ID you have assigned to the App on the iOS Dev Center.
- In Xcode, the Bundle ID is stored in the Info.plist, and is later copied into your app's bundle when you build.
- In Member center, you create an App ID that matches the app's bundle ID.
- In iTunes Connect, you enter the Bundle ID to identify your app, after your first version is available on the store, you cannot change your bundle ID EVER AGAIN.

Teams!

- Each Xcode project is associated with a single team.
- If you enroll as an individual, you're considered a one-person team.
- The team account is used to store the certificates, identifiers, and profiles needed to provision your app.
- All iOS apps needs to be provisioned to run on a device.

Team Provisioning Profile

- When you set your team, Xcode ‘may’ attempt to create your code signing identity and development provisioning profile.
- Xcode creates a specialized development provisioning profile called a team provisioning profile that it manages for you.
- A team provisioning profile allows an app to be signed and run by all team members on all their devices.

Provisioning Profile Creation

- Here are the steps Xcode takes when creating your provisioning profile:

1. Requests your development certificate
2. Registers the iOS device chosen in the Scheme popup menu
3. Creates an App ID that matches your app's bundle ID and enables services
4. Creates a team provisioning profile that contains these assets
5. Sets your project's code signing build settings accordingly

Version Number & Build String

- The version number of an app is 3 positive integers separated by periods (ex: 1.0.4)
- The first digit is a major release, the 2nd is a minor release, and the third is a maintenance release.
- Build String represents an iteration of the bundle and contain letters and numbers. Change it whenever you distribute a new build of your app for testing.

Code Signing

- “Code Signing your app lets users trust your app has been created by a source known to Apple and that it hasn't been tampered with”
- The **signing identity** is a public-private key pair that Apple issues. The private key is stored in your keychain and used to generate a signature. The certificate contains the public key and identifies you as the owner of the key pair.
- To sign an app, you also need an intermediate certificate, which is automatically installed in your keychain when you install Xcode.
- You use Xcode to create your signing identity and sign your app. Your signing identity is added to your keychain after creation and the corresponding certificate is stored in the member center.
- A **development certificate** identifies you, as a team member, in a development provisioning profile that allows your apps signed by you to launch on devices.
- A **distribution certificate** identifies your team or organization in a distribution provisioning profile and allows you to submit your app to the store.
- You can view your Signing Identifies and Provisioning Profiles in Xcode if you need to troubleshoot them! Everything should match what you see in Member Center.

Submitting your app

1. Create a distribution certificate for your app in Xcode.
2. Create a store distribution provisioning profile on Member Center.
3. Archive and Validate your app in Xcode.
4. Create your App on iTunes Connect
5. Submit your app binary using Xcode or application Loader.
6. Finalize all the info on iTunes Connect and add the build you submitted from Xcode

Demo

Multiple Storyboards

Multiple Storyboards

- When you are building an app with a storyboard, sometimes the storyboard can be become so large and all-encompassing that it becomes unwieldy to use (both hard to find your way around, and also takes forever to load)
- In these cases, it can be very beneficial to break the large storyboard into separate smaller storyboards.

Multiple Storyboards Workflow

1. Add new storyboard's via File->New->File->iOS->Interface->Storyboard
2. Give the storyboard a name that describes what it feature or area of the app it is gong to encompass
3. In code, when you need to transition to a VC in another storyboard, use UIStoryboard's storyboardWithName:bunde: method to get a reference to your next storyboard.
4. Then use instantiateViewControllerWithIdentifier: on that storyboard to get a reference to first VC you want to show from that storyboard.

Demo

NSURLCache

Sourced from NSHipster

NSURLCache

- NSURLCache provides both in memory and on disk caching.
- Any connections handled by either NSURLConnection or NSURLSession(yay!) can be handled by NSURLCache
- Network caching greatly reduces the number of requests to the server, which both helps us the developer (rate limiting, download slowness) and the user (speed, better user experience)

NSURLCache

- When a request has finished loading its response from the server, a cached response will be saved locally.
- The next time the same request is made, the locally-cached response will be returned immediately, without having to connect to the server.
- This is all done automatically for you. Its not difference in code if you are downloading something for the first time, or if you are downloading someone you have already gotten, which then uses the cache.

NSURLCache Setup

- NSURLCache has an init that takes in memory capacity (in bytes), disk capacity (in bytes), and disk path (string)
- The capacities should be at least 50MB or more. It seems the url cache doesn't work properly if you dont have a size approaching that ammount.
- The disk patch can be any string. It saves it in the appropriate directory for you
- You can then associate the urlCache with the NSURLSession.sharedSession’s configuration.

NSURLCache Setup

```
let cache = NSURLCache(memoryCapacity: 60 * 1024 * 1024,
  diskCapacity: 200 * 1024 * 1024, diskPath: "test.urlcache")

NSURLSession.sharedSession().configuration.URLCache = cache
```

NSURLCachePolicy

- When you make an NSURLRequest, you can specify the NSURLCachePolicy:
- **UseProtocolCachePolicy:** Default behavior.
- **ReloadIngoringLocalCacheData:** Doesn't use cache.
- **ReturnCacheDataElseLoad:** Use the cache (no matter how out of date) or if no cached response exists, load from network.
- **ReturnCacheDataDontLoad:** Offline mode: use the cache (no matter how out of date), but don’t load from network.

HTTP Caching Semantics

- NSURLCache is designed to work with both FTP and HTTP/HTTPS protocols.
- Via HTTP, the way NSURLCache works is via headers.
- By default, NSURLRequest will use the current time to determine if a cached response be returned.
- If you want more precise control, you can set the following headers yourself:
  - **If-Modified-Since:** This request header corresponds to the Last-Modified response header. Set the value of this to the Last-Modified value received from the last request to the same endpoint
  - **If-None-Match:** This request header corresponds to the Etag value received previously for the last request to that endpoint.

HTTP Caching Semantics

- An NSHTTPURLResponse contains a set of HTTP headers, which can include the following directives for how that response should be cached:
  - **Cache-Control:** This header must be present in the response from the server to enable HTTP caching by the client. The value of this header may include information like its max-age (how long to cache the reponse), and whether the response may be cached with public or private access, or no-cache at all. See RFC 2616 for more details

HTTP Caching Semantics

- In addition to Cache-Control, the server may add addtitional headers that can be used to conditionally request information as needed (as mentioned in the previous section)
  - **Last-Modified:** The value of this header corresponds to the date and time when the requested resource was last changed.
  - **Etag:** An abbreviation for “entity-tag”, this is an identifier that represents the contents requested resource. In practice, this Etag value could be like something like an MD5 hash. This is usefull if the resources you are requested are dynamically generated, and wont have a ‘Last-Modified’

Demo

Multiple Storyboards Workflow

1. Add new storyboard's via File->New->File->iOS->Interface->Storyboard
2. Give the storyboard a name that describes what it feature or area of the app it is gong to encompass
3. In code, when you need to transition to a VC in another storyboard, use UIStoryboard's storyboardWithName:bunde: method to get a reference to your next storyboard.
4. Then use instantiateViewControllerWithIdentifier: on that storyboard to get a reference to first VC you want to show from that storyboard.

Demo