# Stanford CS193p

Developing Applications for iOS
Fall 2017-18

CS193p
Fall 2017-18

# Today

◉ Emoji Art

Demo continued ... UITextField to add more Emoji

◉ Persistence

UserDefaults

Property List

Archiving and Codable

Filesystem

Core Data

Cloud Kit

UIDocument

UIDocumentBrowserViewController

# UserDefaults

◉ A very lightweight and limited database

UserDefaults is essentially a very tiny database that persists between launchings of your app.
Great for things like "settings" and such. Do not use it for anything big!

◉ What can you store there?

You are limited in what you can store in UserDefaults: it only stores Property List data.
A Property List is any combo of Array, Dictionary, String, Date, Data or a number (Int, etc.).
This is an old Objective-C API with no type that represents all those, so this API uses Any.
If this were a new, Swift-style API, it would almost certainly not use Any.
(Likely there would be a protocol or some such that those types would implement.)

◉ What does the API look like?

It's "core" functionality is simple. It just stores and retrieves Property Lists by key ...
```
func set(Any?, forKey: String)        // the Any has to be a Property List (or crash)
func object(forKey: String) -> Any? // the Any is guaranteed to be a Property List
```

# UserDefaults

◉ Reading and Writing

You don't usually create one of these databases with UserDefaults().

Instead, you use the static (type) var called standard ...

```
let defaults = UserDefaults.standard
```

Setting a value in the database is easy since the set method takes an Any?.

```
defaults.set(3.1415, forKey: "pi")      // 3.1415 is a Double which is a Property List type
defaults.set([1,2,3,4,5], forKey: "My Array") // Array and Int are both Property Lists
defaults.set(nil, forKey: "Some Setting") // removes any data at that key
```

You can pass anything as the first argument as long as it's a combo of Property List types.

UserDefaults also has convenience API for getting many of the Property List types.

```
func double(forKey: String) -> Double
func array(forKey: String) -> [Any]?                    // returns nil if non-Array at that key
func dictionary(forKey: String) -> [String:Any]? // note that keys in return are Strings
```

The Any in the returned values will, of course, be a Property List type.

# UserDefaults

◎ Saving the database

Your changes will be occasionally autosaved.

But you can force them to be saved at any time with synchronize ...

`if !defaults.synchronize() {` // failed! but not clear what you can do about it `}`

(it's not "free" to synchronize, but it's not that expensive either)

# Archiving

- There are two mechanisms for making ANY object persistent

  A historical method which is how, for example, the storyboard is made persistent.

  A new mechanism in iOS 11 which is supported directly by the Swift language environment.

  We're only going to talk in detail about the second of these.

  Since it's supported by the language, it's much more likely to be the one you'd use.

# Archiving

- NSCoder (old) mechanism

  Requires all objects in an object graph to implement these two methods ...

  func encode(with aCoder: NSCoder)

  init(coder: NSCoder)

  Essentially you store all of your object's data in the coder's dictionary.

  Then you have to be able to initialize your object from a coder's dictionary.

  References within the object graph are handled automatically.

  You can then take an object graph and turn it into a Data (via NSKeyedArchiver).

  And then turn a Data back into an object graph (via NSKeyedUnarchiver).

  Once you have a Data, you can easily write it to a file or otherwise pass it around.

# Archiving

⊙ Codable (new) mechanism

Also is essentially a way to store all the vars of an object into a dictionary.

To do this, all the objects in the graph of objects you want to store must implement Codable.

But the difference is that, for standard Swift types, Swift will do this for you.

If you have non-standard types, you can do something similar to the old method.

Some of the standard types (that you'd recognize) that are automatically Codable by Swift ...

String, Bool, Int, Double, Float

Optional

Array, Dictionary, Set, Data

URL

Date, DateComponents, DateInterval, Calendar

CGFloat, AffineTransform, CGPoint, CGSize, CGRect, CGVector

IndexPath, IndexSet

NSRange

# Archiving

- Codable (new) mechanism

    Once your object graph is all Codable, you can convert it to either JSON or a Property List.

    ```
    let object: MyType = …
    let jsonData: Data? = try? JSONEncoder().encode(object)
    ```

    Note that this encode throws. You can catch and find errors easily (next slide).

    By the way, you can make a String object out of this Data like this …

    ```
    let jsonString = String(data: jsonData!, encoding: .utf8) // JSON is always utf8
    ```

    To get your object graph back from the JSON …

    ```
    if let myObject: MyType = try? JSONDecoder().decode(MyType.self, from: jsonData!) {
    }
    ```

    JSON is not "strongly typed." So things like Date or URL are just strings.

    Swift handles all this automatically and is even configurable, for example …

    ```
    let decoder = JSONDecoder()
    decoder.dateDecodingStrategy = .iso8601 // or .secondsSince1970, etc.
    ```

# Archiving

◎ **Codable** (new) mechanism

Here's what it might look like to `catch` errors during a decoding.

The thing decode `throws` is an enum of type `DecodingError`.

Note how we can get the associated values of the enum similar to how we do with `switch`.

```
do {
    let object = try JSONDecoder().decode(MyType.self, from: jsonData!)
    // success, do something with object
} catch DecodingError.keyNotFound(let key, let context) {
    print("couldn't find key \(key) in JSON: \(context.debugDescription)")
} catch DecodingError.valueNotFound(let type, let context) {


} catch DecodingError.typeMismatch(let type, let context) {


} catch DecodingError.dataCorrupted(let context) {


}
```

# Archiving

Codable **Example**

So how do you make your data types Codable?  Usually you just say so ...

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!
}
```

If your vars are all also Codable (standard types all are), then you're done!

The JSON for this might look like ..

```
{
    "someDate" : "2017-11-05T16:30:00Z",
    "someString" : "Hello",
    "other" : <whatever SomeOtherType looks like in JSON>
}
```

# Archiving

● **Codable** Example

Sometimes JSON keys might have different names than your var names (or not be included).

For example, someDate might be some_date.

You can configure this by adding a private enum to your type called CodingKeys like this ...

```swift
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!

    private enum CodingKeys : String, CodingKey {
        case someDate = "some_date"
        // note that the someString var will now not be included in the JSON
        case other // this key is also called "other" in JSON
    }
}
```

# Archiving

- Codable Example

  You can participate directly in the decoding by implementing the decoding initializer ...

  ```
  struct MyType : Codable {
      var someDate: Date
      var someString: String
      var other: SomeOtherType // SomeOtherType has to be Codable too!

      init(from decoder: Decoder) throws {
          let container = try decoder.container(keyedBy: CodingKeys.self)
          someDate = try container.decode(Date.self, forKey: .someDate)
          // process rest of vars, perhaps validating input, etc. ...

      }
  }
  ```

  Note that this init throws, so we don't need do { } inside it (it will just rethrow).

  Also note the "keys" are from the CodingKeys enum on the previous slide (e.g. .someDate).

# Archiving

- Codable Example

  You can participate directly in the decoding by implementing the decoding initializer ...

  ```swift
  class MyType : Codable {

      var someDate: Date

      var someString: String

      var other: SomeOtherType // SomeOtherType has to be Codable too!


      init(from decoder: Decoder) throws {
          let container = try decoder.container(keyedBy: CodingKeys.self)
          someDate = try container.decode(Date.self, forKey: .someDate)
          // process rest of vars, perhaps validating input, etc. ...
          let superDecoder = try container.superDecoder()
          try super.init(from: superDecoder) // only if class
      }
  }
  ```

  Don't call super.init with your own decoder (use your own container's superDecoder()).

# Archiving

- Codable **Example**

  You can participate directly in the decoding by implementing the decoding initializer ...

  ```swift
  struct MyType : Codable {
      var someDate: Date
      var someString: String
      var other: SomeOtherType // SomeOtherType has to be Codable too!

      func encode(to encoder: Encoder) throws {
          var container = encoder.container(keyedBy: CodingKeys.self)
          // there are other containers too (e.g. an unkeyed (i.e. array) container) ...
          try container.encode(someDate, forKey: .someDate)
          // encode the rest of vars, perhaps transforming them, etc. ...
      }
  }
  ```

# File System

- Your application sees iOS file system like a normal Unix filesystem
  - It starts at /.
  - There are file protections, of course, like normal Unix, so you can't see everything.
  - In fact, you can only read and write in your application's "sandbox".

- Why sandbox?
  - Security (so no one else can damage your application)
  - Privacy (so no other applications can view your application's data)
  - Cleanup (when you delete an application, everything it has ever written goes with it)

- So what's in this "sandbox"?
  - Application directory — Your executable, .storyboards, .jpgs, etc.; not writeable.
  - Documents directory — Permanent storage created by and always visible to the user.
  - Application Support directory — Permanent storage not seen directly by the user.
  - Caches directory — Store temporary files here (this is not backed up by iTunes).
  - Other directories (see documentation) ...

# File System

◎ Getting a path to these special sandbox directories

FileManager (along with URL) is what you use to find out about what's in the file system.

You can, for example, find the URL to these special system directories like this ...

```
let url: URL = FileManager.default.url(
    for directory: FileManager.SearchPathDirectory.documentDirectory, // for example
    in domainMask: .userDomainMask  // always .userDomainMask on iOS
    appropriateFor: nil, // only meaningful for "replace" file operations
    create: true // whether to create the system directory if it doesn't already exist
)
```

◎ Examples of SearchPathDirectory values

.documentDirectory, .applicationSupportDirectory, .cachesDirectory, etc.

# URL

- Building on top of these system paths

    URL methods:

    ```
    func appendingPathComponent(String) -> URL

    func appendingPathExtension(String) -> URL   // e.g. "jpg"
    ```

- Finding out about what's at the other end of a URL

    ```
    var isFileURL: Bool  // is this a file URL (whether file exists or not) or something else?
    func resourceValues(for keys: [URLResourceKey]) throws -> [URLResourceKey:Any]?
    Example keys: .creationDateKey, .isDirectoryKey, .fileSizeKey
    ```

# File System

◈ Data

Reading binary data from a URL ...

`init(contentsOf: URL, options: Data.ReadingOptions) throws`

Writing binary data to a URL ...

`func write(to url: URL, options: Data.WritingOptions) throws -> Bool`

The options can be things like .atomic (write to tmp file, then swap) or .withoutOverwriting.

Notice that this function throws.

# File System

◉ FileManager

Provides utility operations.

e.g., `fileExists(atPath: String) -> Bool`

Can create and enumerate directories; move, copy, delete files; etc.

Thread safe (as long as a given instance is only ever used in one thread).

Also has a `delegate` with lots of "should" methods (to do an operation or proceed after an error).

And plenty more.  Check out the documentation.

# Core Data

◉ Database

 Sometimes you need to store large amounts of data locally in a database.

 And you need to search through it in an efficient, sophisticated manner.

◉ Enter Core Data

 Object-oriented database.

 Very, very powerful framework in iOS (unfortunately no time to cover it this quarter).

 Check out Winter of 2016-17's iTunesU for a full pair of lectures on it!

◉ It's a way of creating an object graph backed by a database

 Usually backed by SQL (but also can do XML or just in memory).

◉ How does it work?

 Create a visual mapping (using Xcode tool) between database and objects.

 Create and query for objects using object-oriented API.

 Access the "columns in the database table" using vars on those objects.
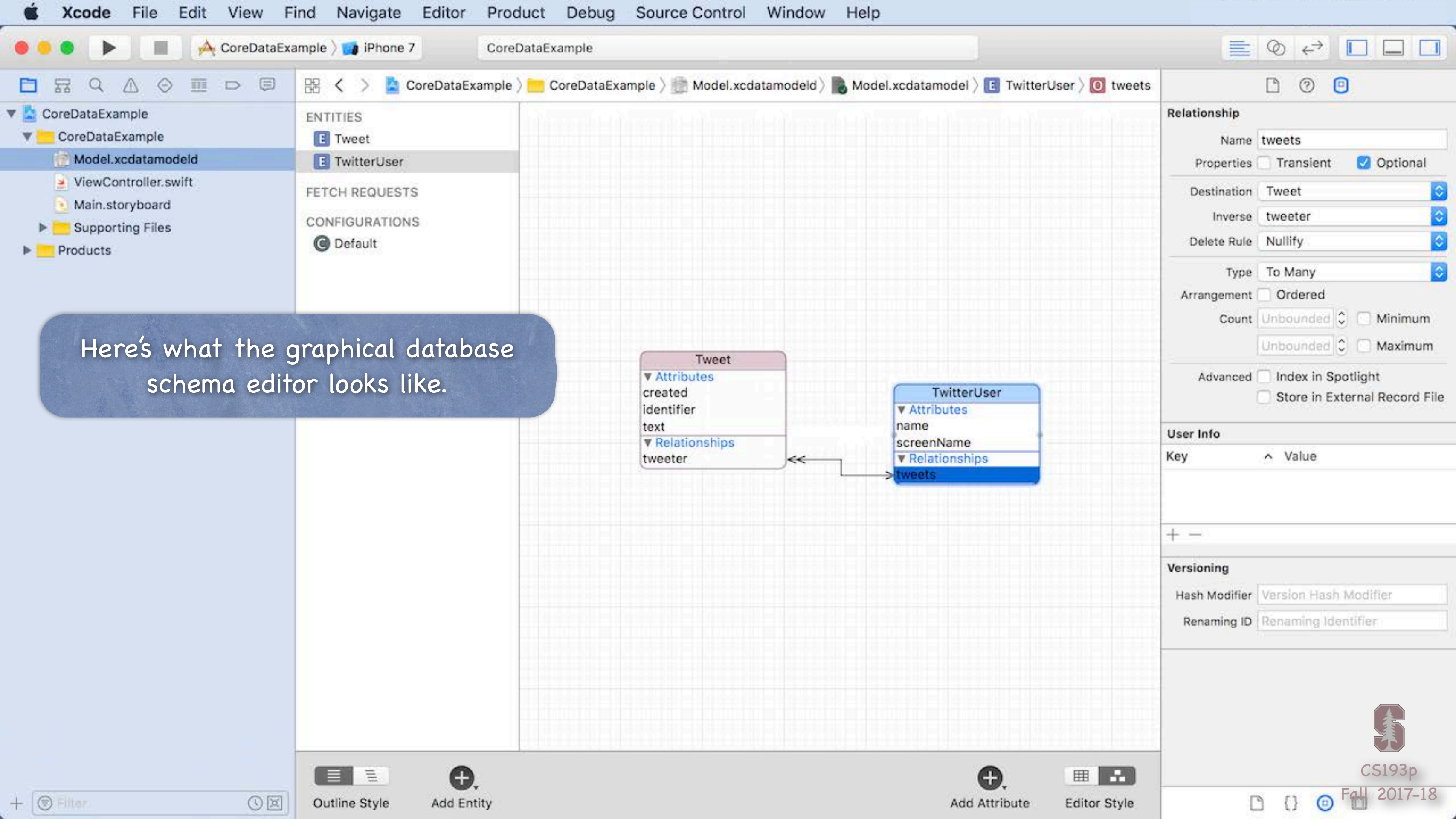
Here's what the graphical database schema editor looks like.
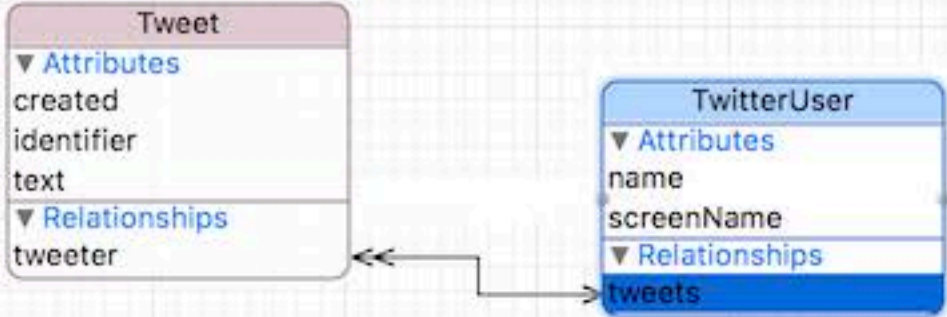
# Core Data

- So how do you access all of this stuff in your code?

  Core Data is access via an NSManagedObjectContext.

  It is the hub around which all Core Data activity turns.

  The code that the Use Core Data button adds creates one for you in your AppDelegate.

# Core Data

What does it look like to create/update objects in the database?

It looks a lot like accessing normal Swift objects ...

```
let context: NSManagedObjectContext = ...
if let tweet = Tweet(context: context) {
    tweet.text = "140 characters of pure joy"
    tweet.created = Date()
    let joe = TwitterUser(context: tweet.managedObjectContext)
    tweet.tweeter = joe
    tweet.tweeter.name = "Joe Schmo"
}
```

# Core Data

⊙ Deleting objects

```
context.delete(tweet)
```

⊙ Saving changes

You must explicitly save any changes to a context, but note that this throws.

```
do {
    try context.save()
} catch {
    // deal with error
}
```

However, we usually use a UIManagedDocument which autosaves for us.

More on UIDocument-based code in a few slides ...

# Querying

◉ Searching for objects in the database

Let's say we want to query for all TwitterUsers …

```
let request: NSFetchRequest<TwitterUser> = TwitterUser.fetchRequest()
```

… who have created a tweet in the last 24 hours …

```
let yesterday = Date(timeIntervalSinceNow:-24*60*60) as NSDate
request.predicate = NSPredicate(format: "any tweets.created > %@", yesterday)
```

… sorted by the TwitterUser's name …

```
request.sortDescriptors = [NSSortDescriptor(key: "name", ascending: true)]
```

```
let context: NSManagedObjectContext = ...
let recentTweeters = try? context.fetch(request)
```

Returns an <u>empty</u> Array (not `nil`) if it succeeds and there are <u>no matches</u> in the database.

Returns an Array of NSManagedObjects (or subclasses thereof) if there were any matches.

And obviously the `try` fails if the fetch fails.

# Core Data

◉ **And so much more!**

Very efficient

Support for multithreading

Close integration with UITableView (for obvious reasons)

Optimistic locking (deleteConflictsForObject)

Rolling back unsaved changes

Undo/Redo

Staleness (how long after a fetch until a refetch of an object is required?)

Etc., etc. ...

# Cloud Kit

◉ Cloud Kit

A database in the cloud. Simple to use, but with very basic "database" operations.

Since it's on the network, accessing the database could be slow or even impossible.

This requires some thoughtful programming.

No time for this one either this quarter, but check Spring of 2015-16's iTunesU for full demo.

◉ Important Components

Record Type - like a class or struct

Fields - like vars in a class or struct

Record - an "instance" of a Record Type

Reference - a "pointer" to another Record

Database - a place where Records are stored

Zone - a sub-area of a Database

Container - collection of Databases

Query - an Database search

Subscription - a "standing Query" which sends push notifications when changes occur

# Cloud Kit

You must enable iCloud in your Project Settings
Under Capabilities tab, turn on iCloud (On/Off switch).
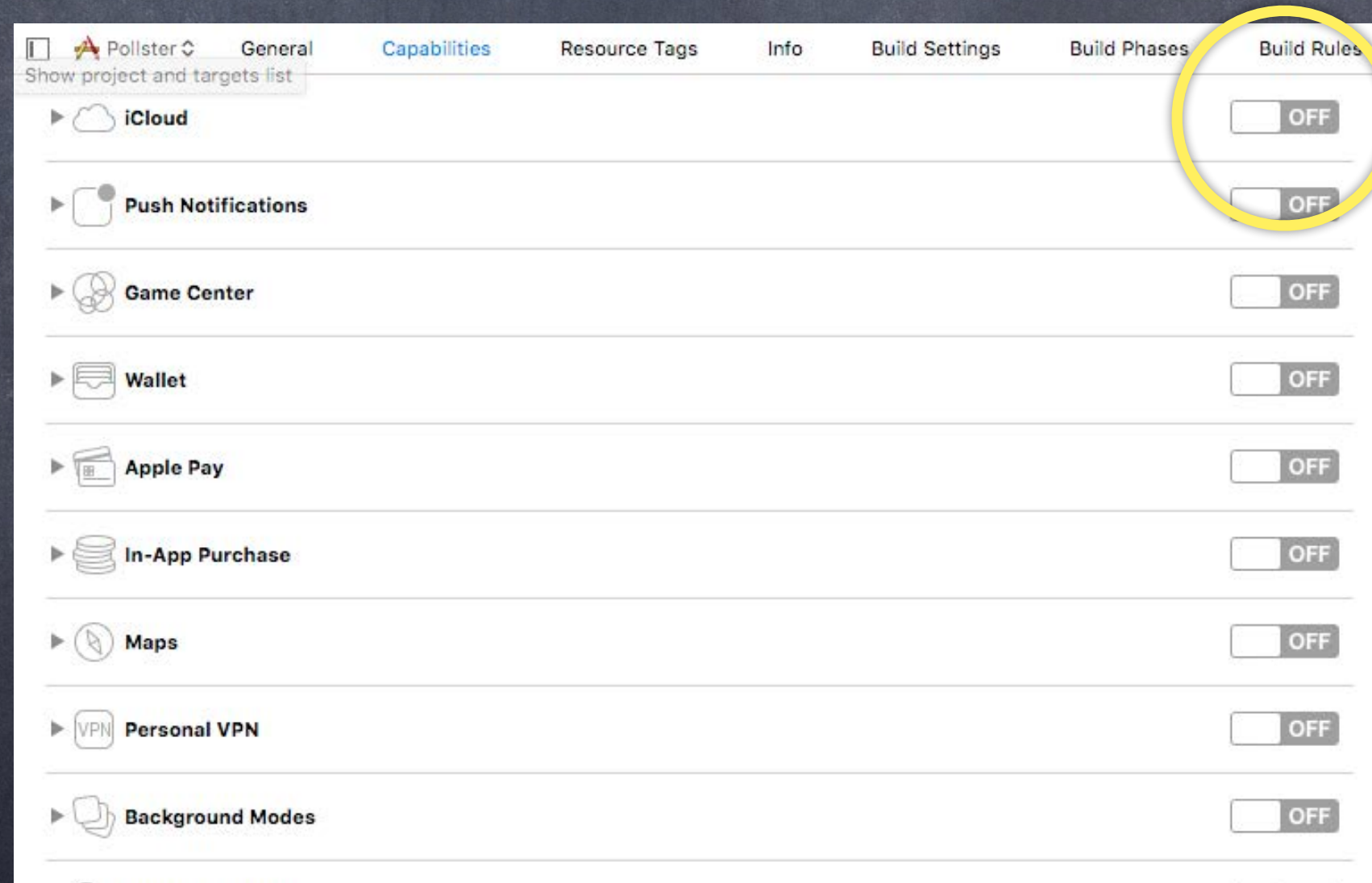
# Cloud Kit

☁ You must enable iCloud in your Project Settings

Under Capabilities tab, turn on iCloud (On/Off switch).

Then, choose CloudKit from the Services.
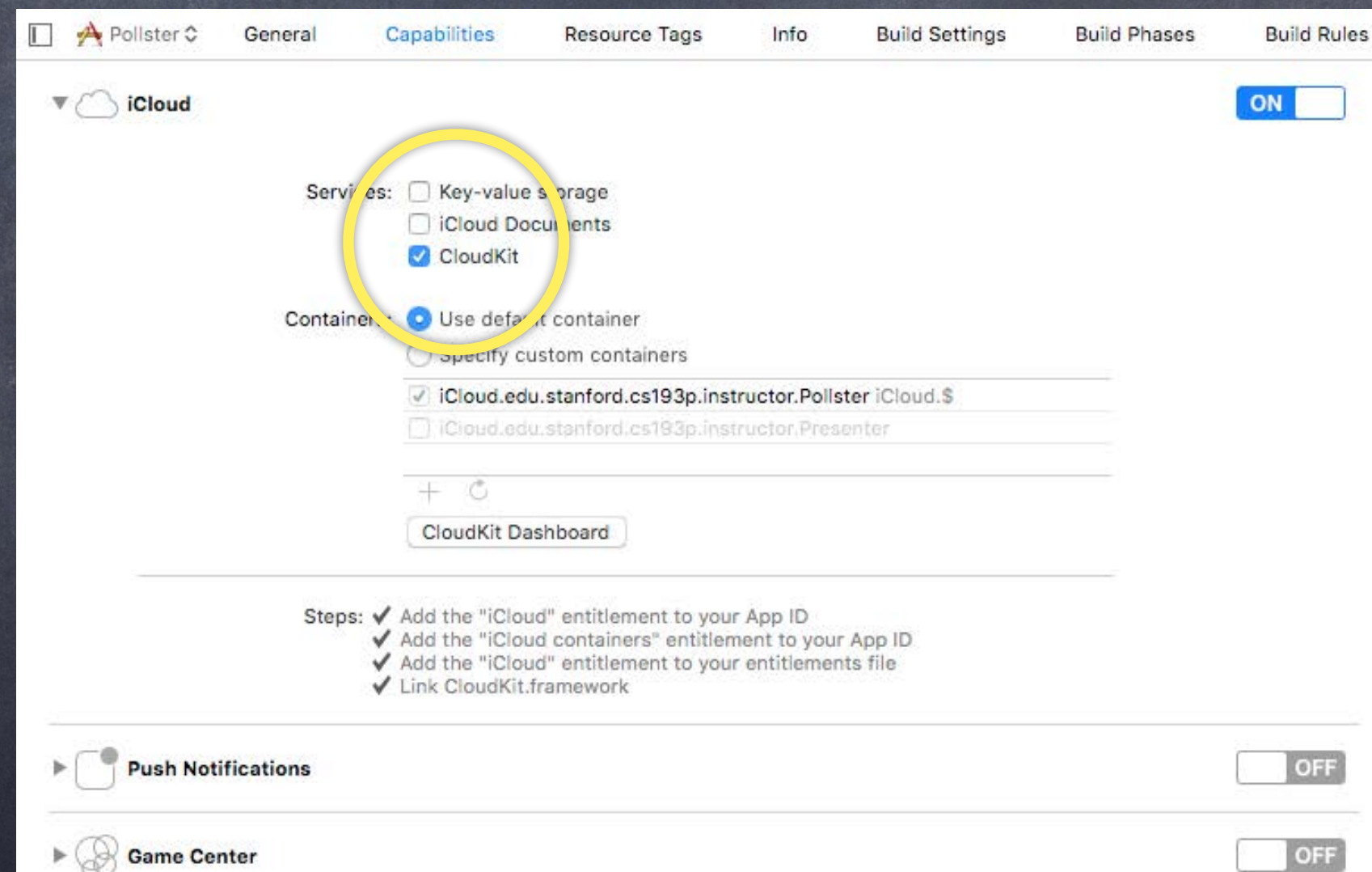
# Cloud Kit

⚙ You must enable iCloud in your Project Settings

Under Capabilities tab, turn on iCloud (On/Off switch).

Then, choose CloudKit from the Services.

You'll also see a CloudKit Dashboard button which will take you to the Cloud Kit Dashboard.
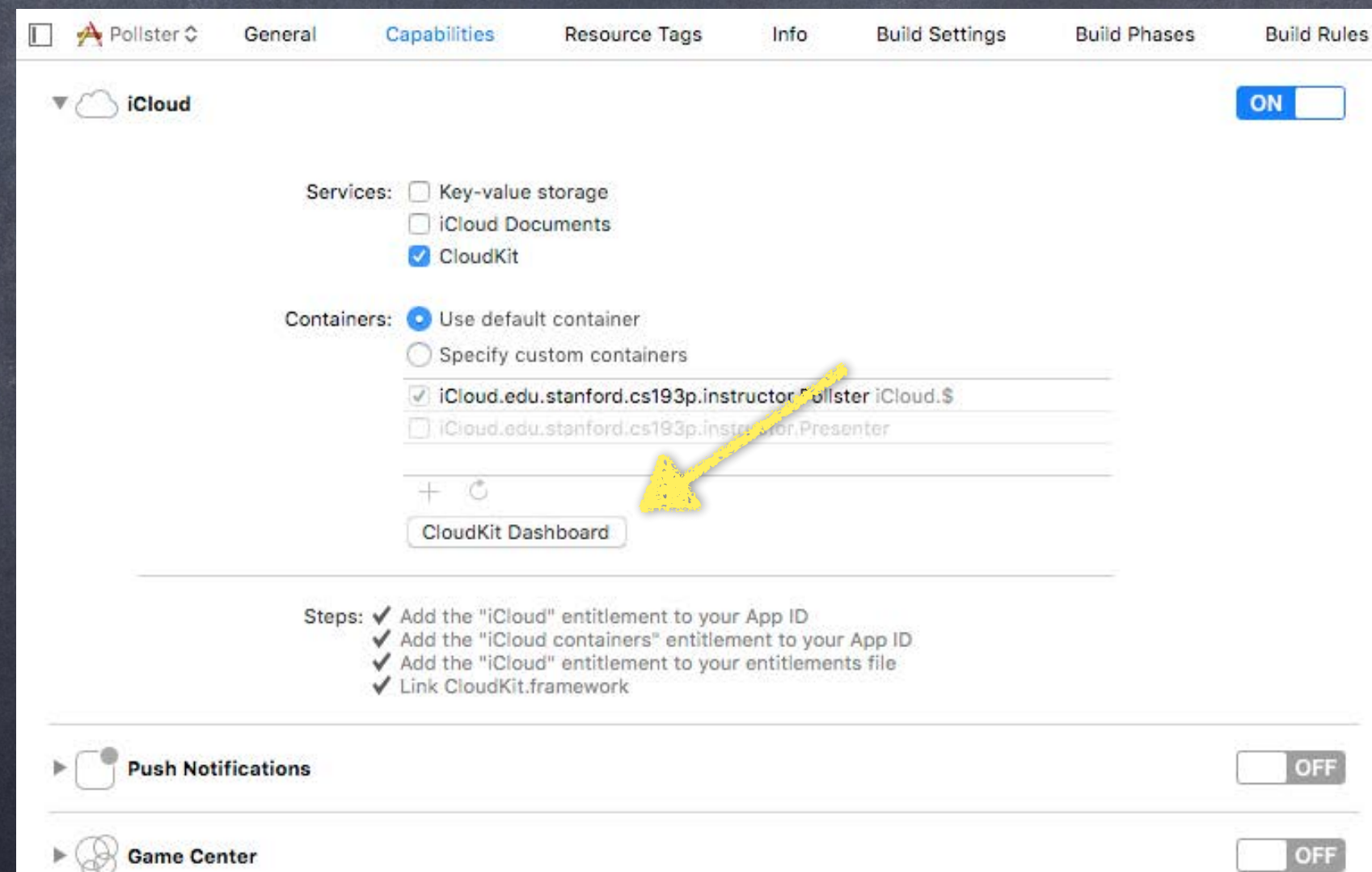
# Cloud Kit

⊚ Cloud Kit Dashboard

A web-based UI to look at everything you are storing.

Shows you all your Record Types and Fields as well as the data in Records.

You can add new Record Types and Fields and also turn on/off indexes for various Fields.

# Cloud Kit

◎ Dynamic Schema Creation

But you don't have to create your schema in the Dashboard.

You can create it "organically" by simply creating and storing things in the database.

When you store a record with a new, never-before-seen Record Type, it will create that type.

Or if you add a Field to a Record, it will automatically create a Field for it in the database.

This only works during Development, not once you deploy to your users.

# Cloud Kit

- What it looks like to create a record in a database

```
let db = CKContainer.default.publicCloudDatabase
let tweet = CKRecord("Tweet")
tweet["text"] = "140 characters of pure joy"
let tweeter = CKRecord("TwitterUser")
tweet["tweeter"] = CKReference(record: tweeter, action: .deleteSelf)
db.save(tweet) { (savedRecord: CKRecord?, error: NSError?) -> Void in
    if error == nil {
        // hooray!
    } else if error?.errorCode == CKErrorCode.NotAuthenticated.rawValue {
        // tell user he or she has to be logged in to iCloud for this to work!
    } else {
        // report other errors (there are 29 different CKErrorCodes!)
    }
}
```

# Cloud Kit

◉ What it looks like to query for records in a database

```
let db = CKContainer.default.publicCloudDatabase
let predicate = NSPredicate(format: "text contains %@", searchString)
let query = CKQuery(recordType: "Tweet", predicate: predicate)
db.perform(query) { (records: [CKRecord]?, error: NSError?) in
    if error == nil {
        // records will be an array of matching CKRecords
    } else if error?.errorCode == CKErrorCode.NotAuthenticated.rawValue {
        // tell user he or she has to be logged in to iCloud for this to work!
    } else {
        // report other errors (there are 29 different CKErrorCodes!)
    }
}
```

# Cloud Kit

🌀 Standing Queries (aka Subscriptions)

One of the coolest features of Cloud Kit is its ability to send push notifications on changes.

All you do is register an NSPredicate and whenever the database changes to match it, boom!

Unfortunately, we don't have time to discuss push notifications this quarter.

If you're interested, check out the UserNotifications framework.

# UIDocument

🌀 **When to use UIDocument**

If your application stores user information in a way the user perceives as a "document".
If you just want iOS to manage the primary storage of user information.

🌀 **What does UIDocument do?**

Manages all interaction with storage devices (not just filesystem, but also iCloud, Box, etc.).
Provides asynchronous opening, writing, reading and closing of files.
Autosaves your document data.
Makes integration with iOS 11's new Files application essentially free.

🌀 **What do you need to do to make UIDocument work?**

Subclass UIDocument to add vars to hold the Model of your MVC that shows your "document".
Then implement one method that writes the Model to a Data and one that reads it from a Data.
That's it.
Now you can use UIDocument's opening, saving and closing methods as needed.
You can also use its "document has changed" method (or implement undo) to get autosaving.

# UIDocument

◉ Subclassing `UIDocument`

For simple documents, there's nothing to do here except add your Model as a `var` ...

```
class EmojiArtDocument: UIDocument {
    var emojiArt: EmojiArt?
}
```

There are, of course, methods you can override, but usually you don't need to.

◉ Creating a `UIDocument`

Figure out where you want your document to be stored in the filesystem ...

```
var url = FileManager.urls(for: .documentDirectory, in: .userDomainMask).first!
url = url.appendingPathComponent("Untitled.foo")
```

Instantiate your subclass by passing that url to UIDocument's only initializer ...

```
let myDocument = EmojiArtDocument(fileURL: url)
```

... then (eventually) set your Model var(s) on your newly created UIDocument subclass ...

```
myDocument.emojiArt = ...
```

# UIDocument

◉ Creating a Data for your Model

Override this method in your UIDocument subclass to convert your Model into a Data.

```
override func contents(forType typeName: String) throws -> Any {
    return emojiArt converted into a Data
}
```

Note that the return type is Any ... that's because your file format can also be a FileWrapper.

A FileWrapper represents a directory full of files that make up your document.

If for some reason you can't create a Data or FileWrapper, you can throw an error here.

We'll see where that thrown error ends up in a couple of slides.

The forType is a UTI (type identifier) calculated from your fileURL's file extension (e.g. .jpg)

We'll see in a few slides how to declare what sorts of files (the UTIs) your app deals with.

# UIDocument

🌀 Turning a Data into a Model

Override this method in your UIDocument subclass to a Data into an instance of your Model.

```
override func load(fromContents contents: Any, ofType typeName: String?) throws {
    emojiArt = contents converted into an EmojiArt
}
```

Again, you can throw here if you can't create a document from the passed contents.

# UIDocument

◉ Ready to go!

Now you can open your document (i.e. get your Model) ...

```
myDocument.open { success in
    if success {
        // your Model var(s) (e.g. emojiArt) is/are ready to use
    } else {
        // there was a problem, check documentState
    }
}
```

This method is asynchronous!

The closure is called on the same thread you call open from (the main thread usually).

We'll see more about documentState in a couple of slides.

# UIDocument

◉ Saving your document

You can let your document know that the Model has changed with this method ...

`myDocument.updateChangeCount(.done)`

... or you can use UIDocument's undoManager (no time to cover that, unfortunately!)

UIDocument will save your changes automatically at the next best opportunity.

Or you can force a save using this method ...

```
let url = myDocument.fileURL // or something else if you want "save as"
myDocument.save(to url: URL, for: UIDocumentSaveOperation) { success in
    if success {
        // your Model has successfully been saved
    } else {
        // there was a problem, check documentState
    }
}
```

`UIDocumentSaveOperation` is either `.forCreating` or `.forOverwriting`.

# UIDocument

🌀 Closing your document

When you are finished using a document for now, close it ...

```
myDocument.close { success in
    if success {
        // your Model has successfully been saved and closed
        // use the open method again if you want to use it
    } else {
        // there was a problem, check documentState
    }
}
```

# UIDocument

⊙ Document State

As all this goes on, your document transitions to various states.

You can find out what state it is in using this var ...

`var documentState: UIDocumentState`

Possible values ...

`.normal` — document is open and ready for use!

`.closed` — document is closed and must be opened to be used

`.savingError` — document couldn't be saved (override `handleError` if you want to know why)

`.editingDisabled` — the document cannot currently be edited (so don't let your UI do that)

`.progressAvailable` — how far a large document is in getting loaded (check `progress` var)

`.inConflict` — someone edited this document somewhere else (iCloud)

To resolve conflicts, you access the conflicting versions with ...

`NSFileVersion.unresolvedConflictVersionsOfItem(at url: URL) -> [NSFileVersion]?`

For the best UI, you could give your user the choice of which version to use.

Or, if your document's contents are "mergeable", you could even do that.

documentState can be "observed" using the `UIDocumentStateChanged` notification (more later).

# UIDocument

 Thumbnail

You can specify a thumbnail image for your UIDocument.

It can make it much easier for users to find the document they want in Files, for example.

Essentially you are going to override the UIDocument method which sets file attributes.

The attributes are returned as a dictionary.

One of the keys is for the thumbnail (it's a convoluted key) ...

```swift
override func fileAttributesToWrite(to url: URL, for operation: UIDocumentSaveOperation)
    throws -> [AnyHashable : Any] {
    var attributes = try super.fileAttributesToWrite(to: url, for: saveOperation)
    if let thumbnail: UIImage = … {
        attributes[URLResourceKey.thumbnailDictionaryKey] =
            [URLThumbnailDictionaryItem.NSThumbnail1024x1024SizeKey:thumbnail]
    }
    return attributes

}
```

It does not have to be 1024x1024 (it seems to have a minimum size, not sure what).

# UIDocument

⊚ Other

```
var localizedName: String

var hasUnsavedChanges: Bool

var fileModificationDate: Date?

var userActivity: NSUserActivity? // iCloud documents only
```
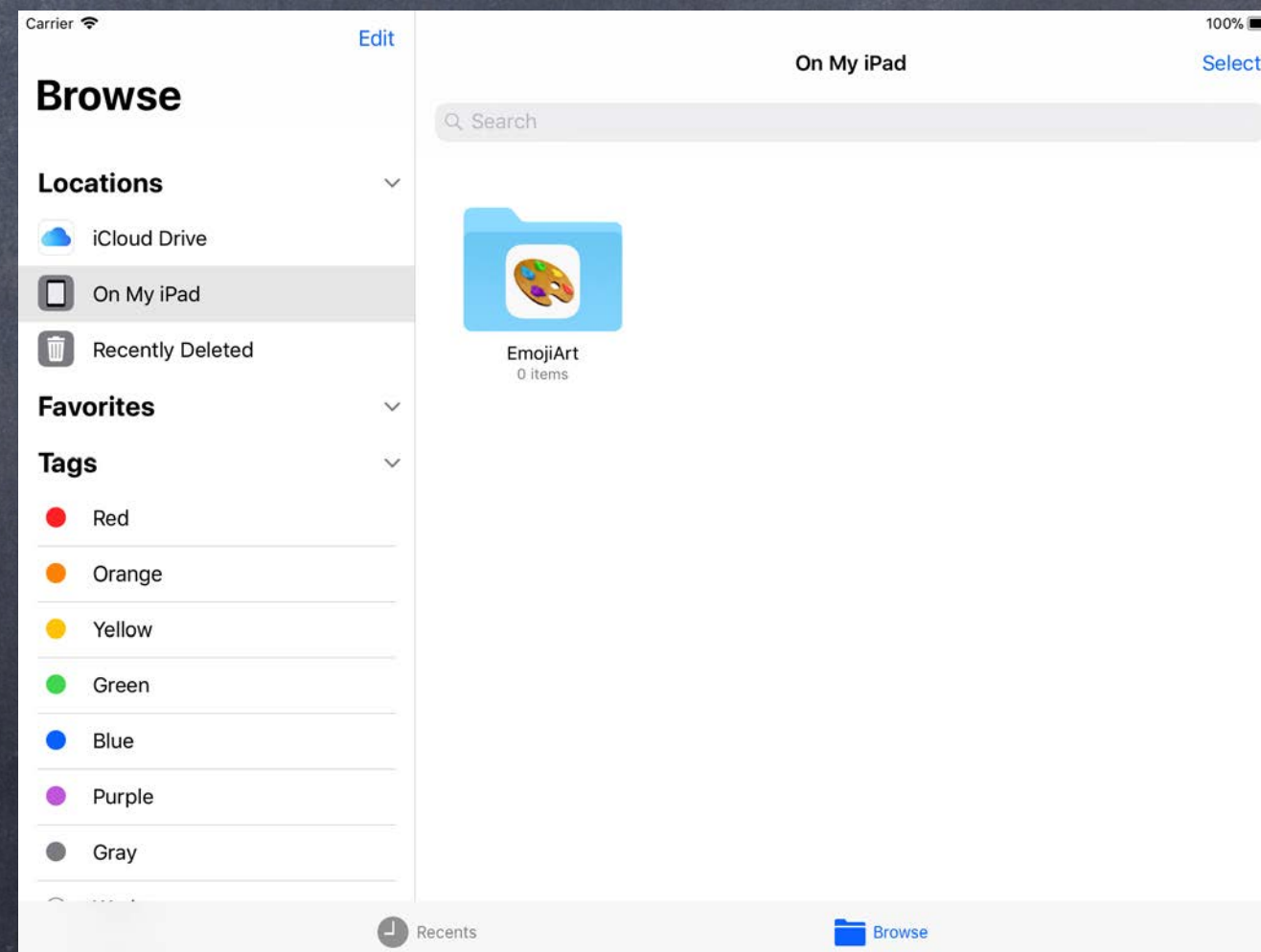
# UIDocumentBrowserViewController

⬡ Managing user documents

You probably want users to be able to easily manage their documents in a document-based app.

Choosing files to open, renaming files, moving them, accessing iCloud drive, etc.

The UIDocumentBrowserViewController (UIDBVC) does all of this for you.

Using UIDocument to store your document makes leveraging this UIDBVC easy.

# UIDocumentBrowserViewController

🌀 Managing user documents

You probably want users to be able to easily manage their documents in a document-based app.

Choosing files to open, renaming files, moving them, accessing iCloud drive, etc.
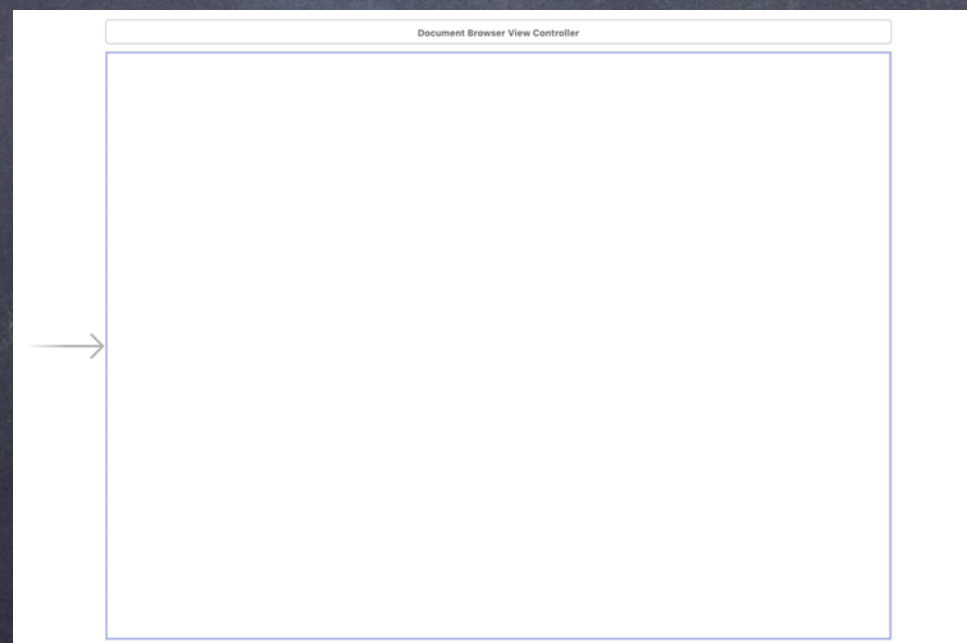
The UIDocumentBrowserViewController (UIDBVC) does all of this for you.

Using UIDocument to store your document makes leveraging this UIDBVC easy.

🌀 Using the UIDocumentBrowserViewController

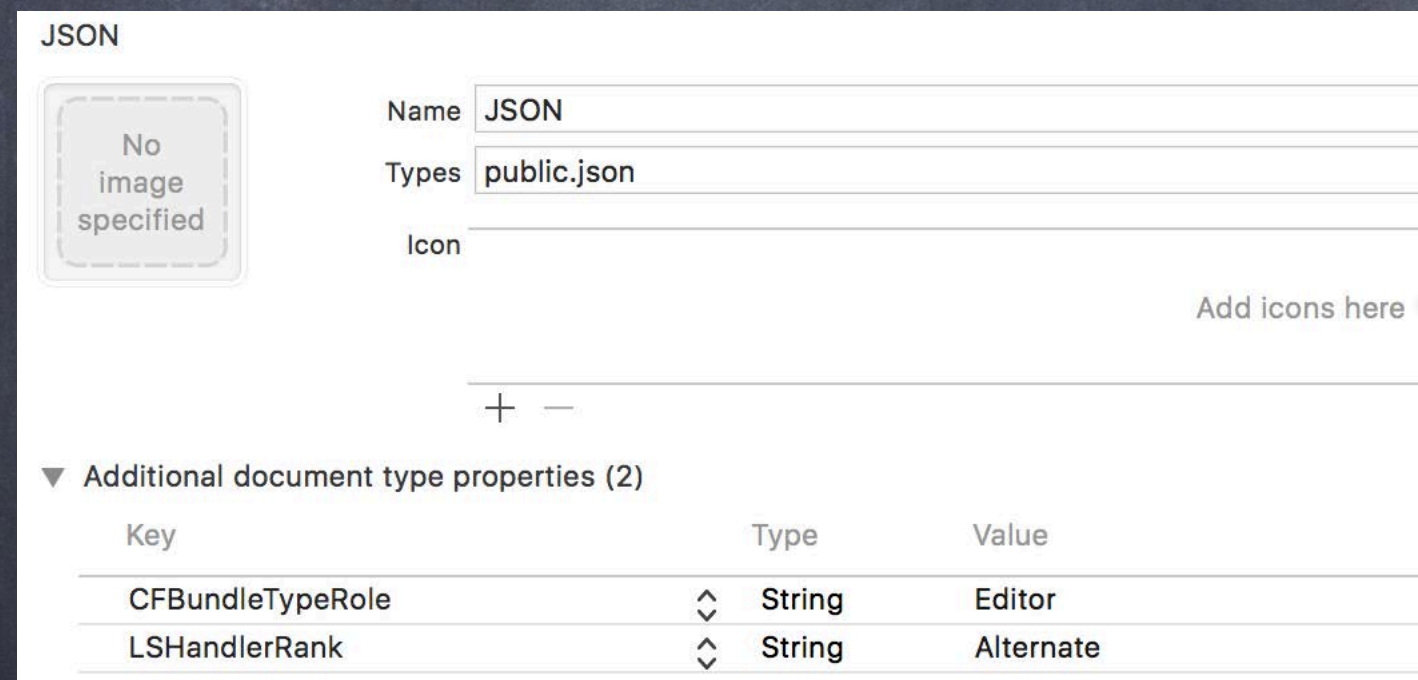It has to be the root view controller in your storyboard (i.e. the arrow points to it).

Your document-editing MVC will then be presented modally on top of (i.e. takes over the screen).

# UIDocumentBrowserViewController

## What document types can you open?

To use the UIDBVC, you have to register which <u>types</u> your application uses.

You do this in the Project Settings in the Info tab with your Target selected.

In the Document Types area, add the types you support.

Here's what it looks like to support JSON files ...



You can add an icon for the file type too.

The Types field is the UTI of the type you want to support (e.g. `public.json`, `public.image`).

The CFBundleTypeRole and LSHandlerRank say how you handle this kind of document.

Are you the primary editor and owner of this type or is it just something you can open?

# UIDocumentBrowserViewController

🌀 Declaring your own document type

You might have a custom document type that your application edits

You can add this under Exported UTIs in the same place in Project Settings

Here's an example of adding an "emojiart" type of document ...

▼ **Exported UTIs (1)**

EmojiArt

| Description | EmojiArt |
| Identifier | edu.stanford.cs193p.emojiart |
| Conforms To | public.data |

Small Icon   None

Large Icon

▼ Additional exported UTI properties (1)

| Key | | Type | Value |
| --- | --- | --- | --- |
| ▼ UTTypeTagSpecification | ⇕ | Dictionary | (1 item) |
| public.filename-extension | | String | emojiart |

This is the "UTI" that we keep referring to.
It's like `public.json` is for JSON.

EmojiArt

No image specified

| Name | EmojiArt |
| Types | edu.stanford.cs193p.emojiart |

Icon

Add icons here

\+  \−

... and then add it as a supported Document Type

▼ Additional document type properties (2)

| Key | | Type | Value |
| --- | --- | --- | --- |
| CFBundleTypeRole | ⇕ | String | Editor |
| LSHandlerRank | ⇕ | String | Owner |

# UIDocumentBrowserViewController

- **Opening documents at the request of other apps (including Files)**

  A user can now click on a document of your type in Files (or another app can ask to open one)

  When this happens, your AppDelegate gets a message sent to it ...

  ```
  func application(UIApplication, open: URL, options: [UIApplicationOpenURLOptionsKey:Any]) -> Bool
  ```

  We haven't discussed the AppDelegate yet, but it's just a swift file with some methods in it.

  Inside here, you can ask your UIDocumentBrowserViewController to show this document ...

  ```
  let uidbvc = window?.rootViewController as? UIDBVC // since it's "arrowed" in storyboard
  uidbvc.revealDocument(at: URL, importIfNeeded: true) { (url, error) in
      if error != nil {
          // present a UIDocument at url modally (more on how to do this in a moment)
      } else {
          // handle the error
      }
  }
  ```

# UIDocumentBrowserViewController
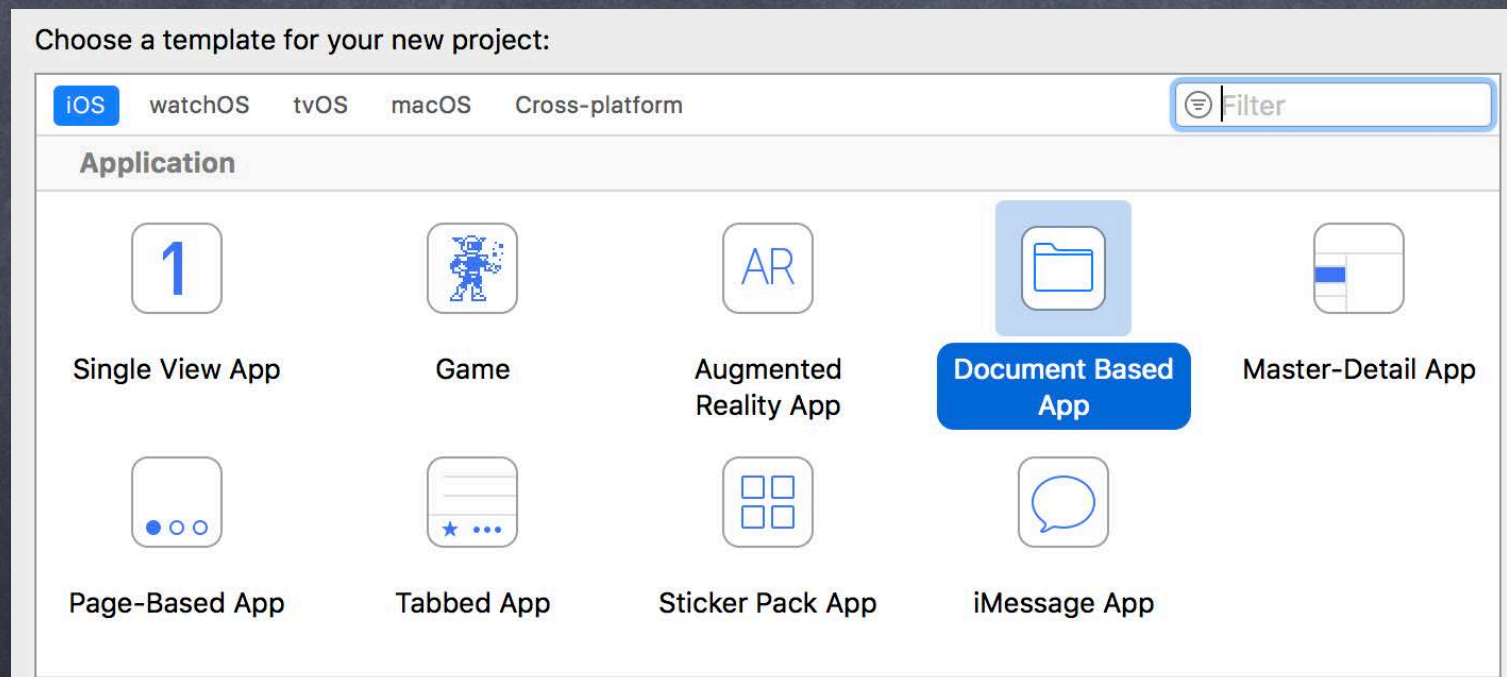
- **Enabling UIDocumentBrowserViewController**

  To make all this work, you need an entry in your Info.plist.

  You can add it the same way we added AppTransportSecurity (right click in Info.plist).

  | Supports Document Browser | ⇕ | Boolean | YES |
  | --- | --- | --- | --- |

- **Xcode template**

  Luckily, an Xcode template exists to do all of the above configuration for us

# UIDocumentBrowserViewController

⊚ What is in the template?

A stub for Document Types in Project Settings (supports `public.image` file types)

The `Info.plist` entry Supports Document Browser = YES

The code in AppDelegate to reveal a document

A stubbed out UIDocument subclass (with empty `contents` and `load(fromContents)` methods)

A stubbed out MVC to display a document (just calls UIDocument's `open` and `close` methods)

A subclass of UIDocumentBrowserViewController (with almost everything implemented)

⊚ What you need to do ...

1. Use your UIDocument subclass instead of the stubbed out one

2. Use your document-viewing MVC code (already using UIDocument) instead of stub

3. Add code to UIDBVC subclass to ...

    a. configure the UIDBVC (allow multiple selection? creation of new documents? etc.)

    b. specify the url of a template document to copy to create new documents

    c. present your document-viewing MVC modally given the url of a document

4. Update the Document Types in Project Settings to be your types (instead of `public.image`)

# UIDocumentBrowserViewController

🌀 **Steps 1 and 2**

As long as you properly implement UIDocument in your MVC, this is no extra work

🌀 **Step 3a: Configuring the UIDBVC**

This happens in its `viewDidLoad` ...

```
override func viewDidLoad() {
    super.viewDidLoad()
    delegate = self        // the guts of making UIDBVC work are in its delegate methods
    allowsDocumentCreation = true
    allowsPickingMultipleItems = true
    browserUserInterfaceStyle = .dark
    view.tintColor = .white
}
```

Set these as you wish.

# UIDocumentBrowserViewController

⚙ **Steps 3b: Specifying the "new document" template URL**

This happens in this UIDBVC delegate method ...

```
func documentBrowser(_ controller: UIDBVC,
    didRequestDocumentCreationWithHandler handler: @escaping (URL?, UIDBVC.ImportMode) -> Void
) {
    let url: URL? = … // where your blank, template document can be found
    importHandler(url, .copy or .move)
}
```

Usually you would specify `.copy`, but you could create a new template each time and `.move`.

Likely you would have some code here that creates that blank template (or ship with your app).

# UIDocumentBrowserViewController

⊘ Aside: Presenting an MVC without segueing

We haven't covered how to present MVCs in any other way except by segueing.

So let's cover it now!

It's very easy. You present a new MVC from an existing MVC using `present(animated:)` ...

```
let newVC: UIViewController = …
existingVC.present(newVC, animated: true) {
    // completion handler called when the presentation completes animating
    // (can be left out entirely if you don't need to do anything upon completion)
}
```

The real trick is "where do I get newMVC from?"

Answer: you get it from your storyboard using its identifier which you set in Identity Inspector

```
let storyboard = UIStoryboard(name: "Main", bundle: nil) // Main.storyboard
if let newVC = storyboard.instantiateViewController(withIdentifier: "foo") as? MyDocVC {
    // "prepare" newMVC and then present(animated:) it
}
```

# UIDocumentBrowserViewController

🌀 Steps 3c: Presenting your document MVC modally

The Xcode template stubs out a function called `presentDocument(at: URL)` to do this ...

```
func presentDocument(at url: URL) {
    let story = UIStoryboard(name: "Main", bundle: nil)
    if let docvc = story.instantiateViewController(withIdentifier: "DocVC") as? DocVC {
        docvc.document = MyDocument(fileURL: url)
        present(docvc, animated: true)
    }
}
```

You can call this function anything you want.

But the point is that it takes a URL to one of your documents and you show it.

The Xcode template then calls this from the appropriate `delegate` methods in UIDBVC.

That's all you have to do to get UIDBVC working.

# UIDocumentBrowserViewController

🌀 Step 4: Specifying your types

Unless your app opens `public.image` files, you'll need to change that in Project Settings

For your homework, for example, you'll probably need to invent a new type for Image Gallery

# Demo Code

Download the <u>demo code</u> from today's lecture.