THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2017 • SESSION 8

BACKENDS FOR MOBILE APPLICATIONS

CLASS NEWS

CLASS NEWS

- Office hours Thursday from 10-11:30 in Young 308

CLASS NEWS

- Week 7 - CloudKit
  - Assignment 5 assigned
- Week 8 - CloudKit Server Extensions; Swift on the Server
  - Case Study assigned
  - Finish server component of Assignment 5
- Week 9 - Case Studies in Class (Assignment 5 Due)
- Week 10 - Cloud Roundup (Realm, Serverless, etc.)

CLASS NEWS

- Case Studies
  - Find an area of interest in mobile/cloud computing and present to class
  - Ideas:
    - New service (serverless, azure, aws lambda)
    - Technique (sharding :), mysql with app engine)
    - Case study (Snapchat on App Engine, what does XX company use)
    - Deeper dive on topic covered (eg. Google Vision API, etc.)
  - 30 minutes

CLASS NEWS

- Week 7 - CloudKit
  - Assignment 5 assigned
- Week 8 - CloudKit Server Extensions; Swift on the Server
  - Case Study assigned
  - Finish server component of Assignment 5
- Week 9 - Case Studies in Class (Assignment 5 Due)
- Week 10 - Cloud Roundup (Realm, Serverless, etc.)

MAKE-UP CLASS MATERIALS AFTER PRESENTATIONS

CLASS NEWS

- Final Projects
  - Opportunity for you to apply what you learned in class to a project you care about
  - Can use old iOS project or extend assignment
  - Talk me about your ideas early

REVIEW

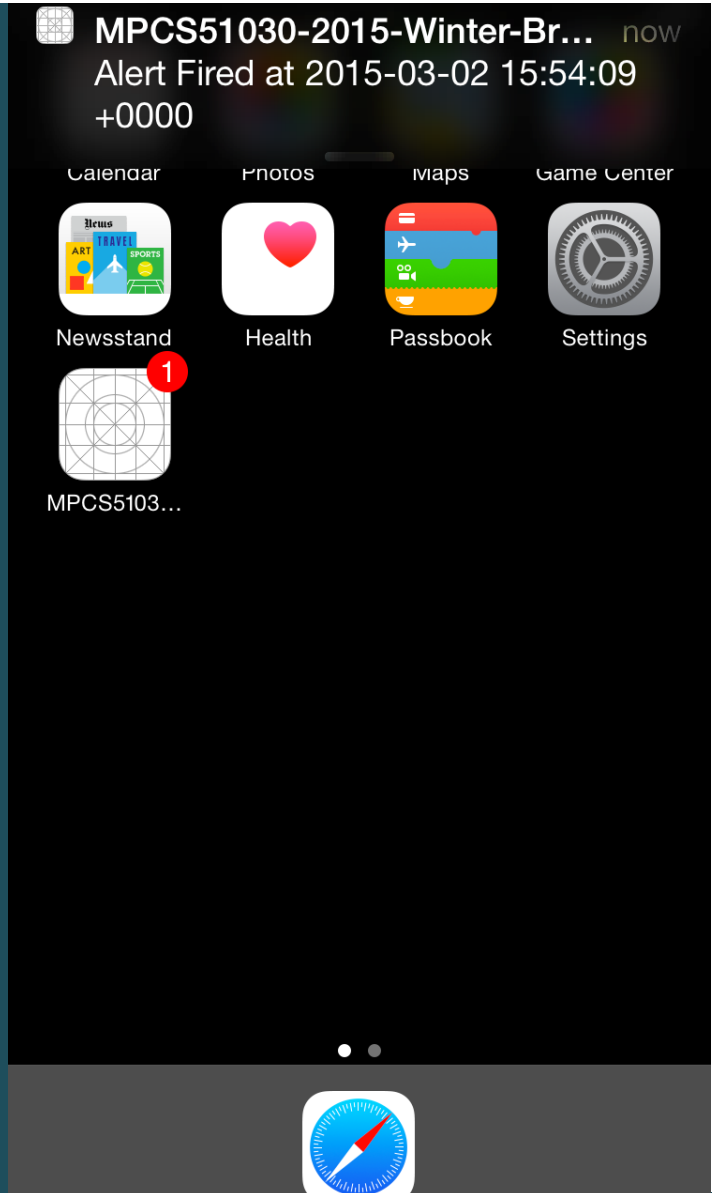
NOTIFICATIONS

NOTIFICATIONS

NOTIFICATIONS

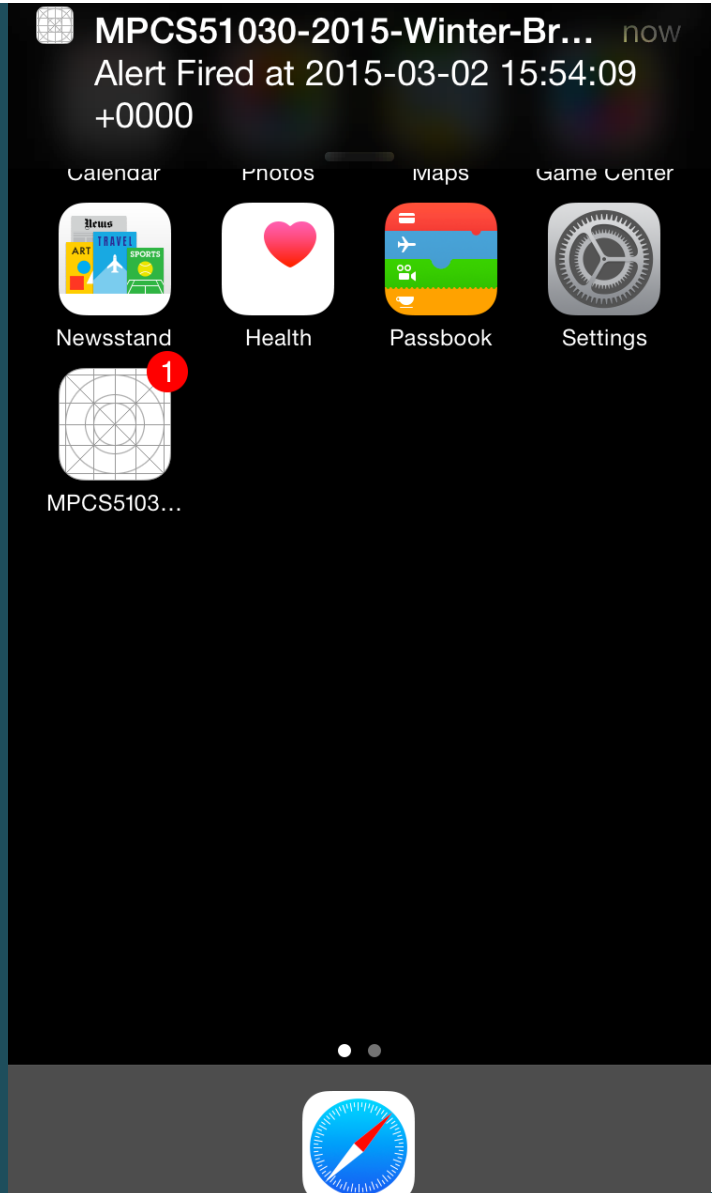
NOTIFICATIONS

- Only one application can run in the foreground
- Notifications give you flexibility in how to interact with users when they're not running your app
  - Messages
  - Game turn
  - Updates available



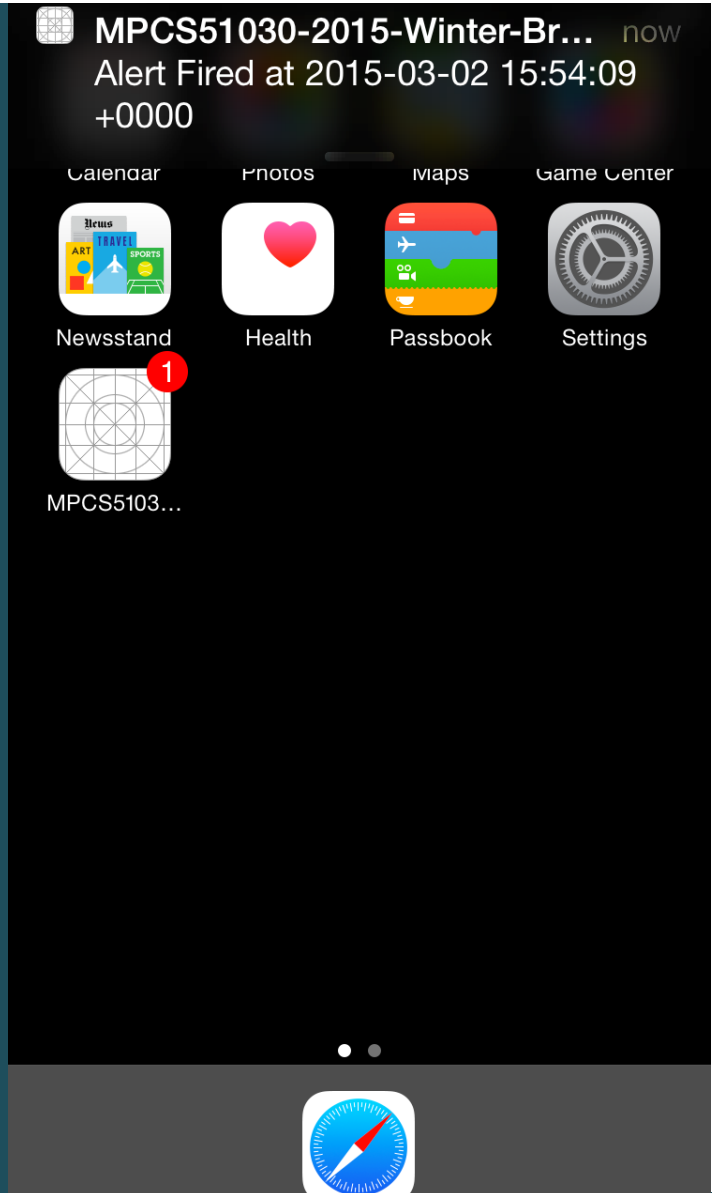
NOTIFICATIONS

- Re-engage your users
  - Notifications can launch your app



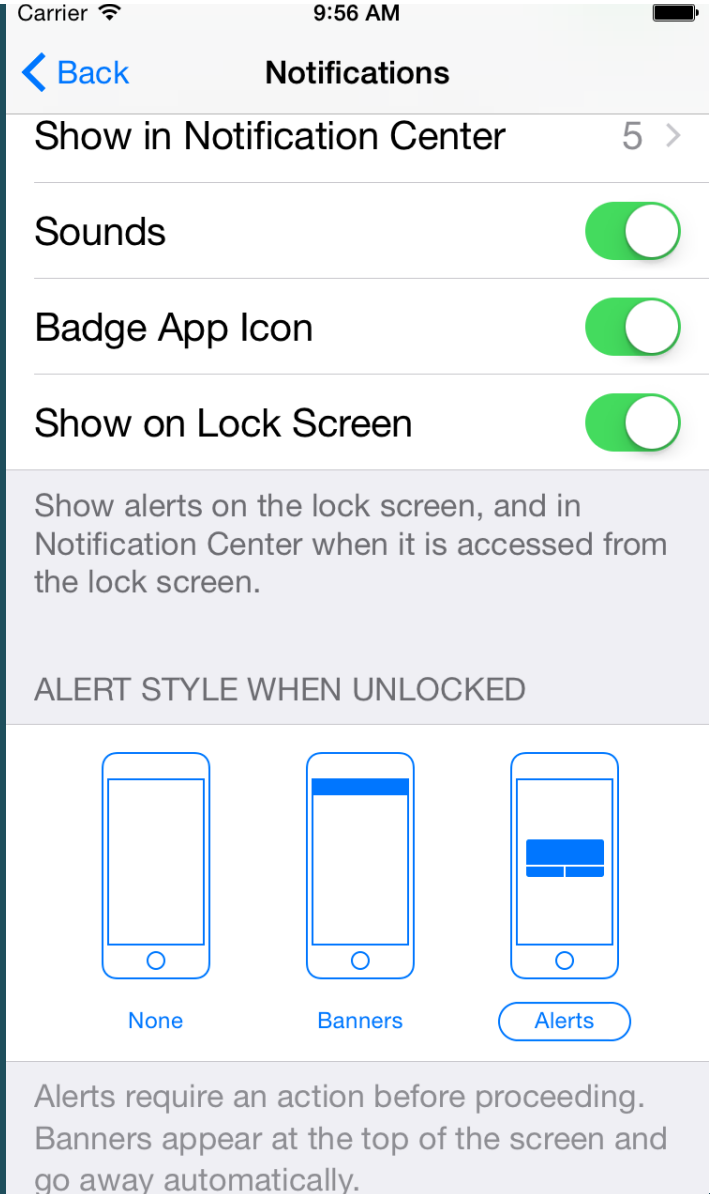
NOTIFICATIONS

- Notification styles
  - Modal
  - Alert
  - Sounds (default, custom)
  - Badges
  - Silent
    - Happens in background
    - “Trigger event”



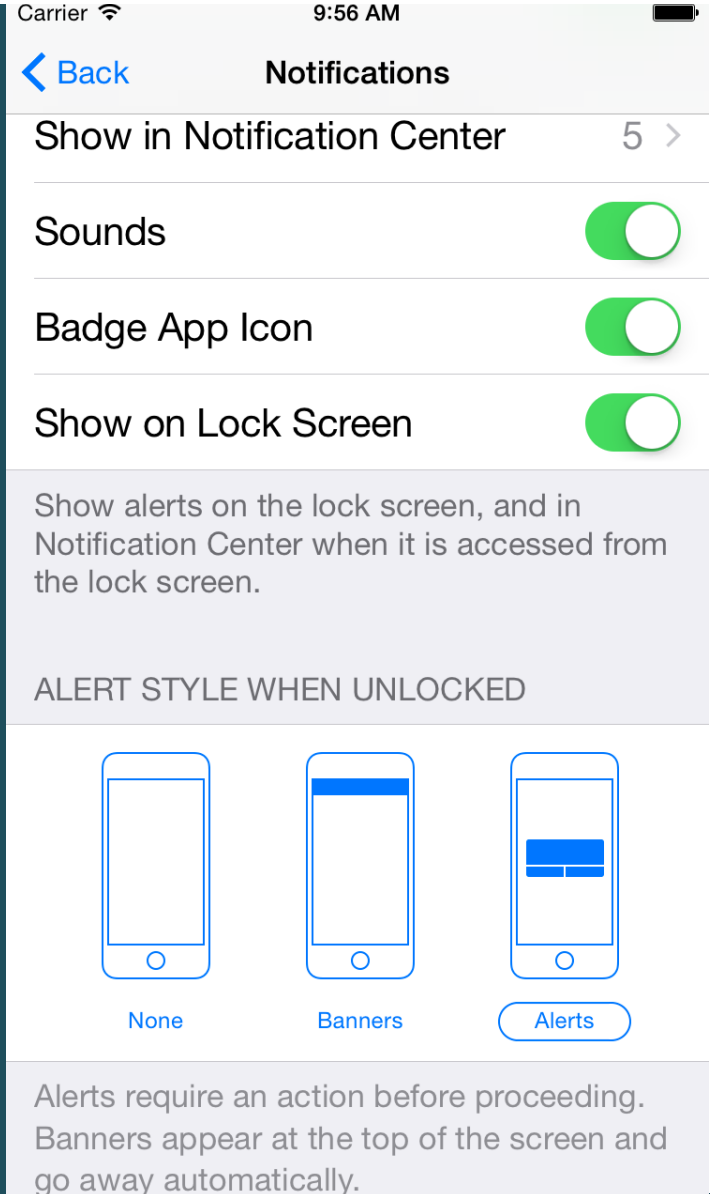
NOTIFICATIONS

- Local Notifications
  - Scheduled by an application
  - Delivered by the iOS on the device
  - “Schedule a notification”



NOTIFICATIONS

- Push Notifications
  - Remote notifications
  - Sent by application's remote server to Apple Push Notification Service (APNS)
  - "Register a notification"



NOTIFICATIONS

Sounds

Badge App Icon

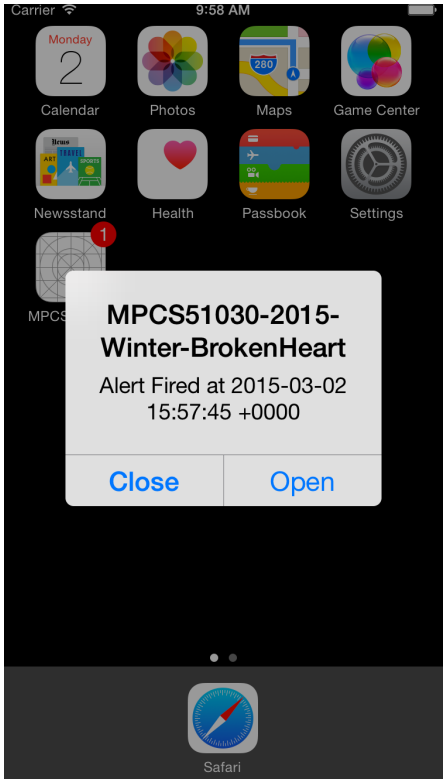
Show on Lock Screen

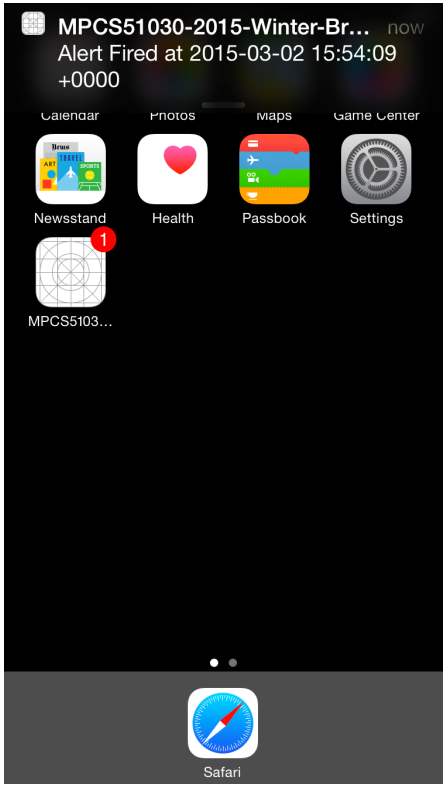
Show alerts on the lock screen, and in Notification Center when it is accessed from the lock screen.

ALERT STYLE WHEN UNLOCKED

None Banners Alerts

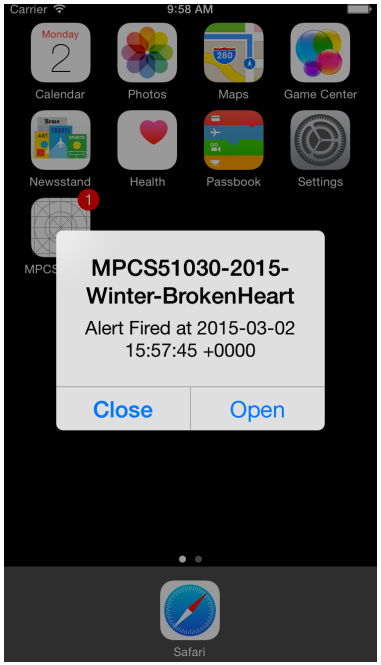
Alerts require an action before proceeding. Banners appear at the top of the screen and go away automatically.



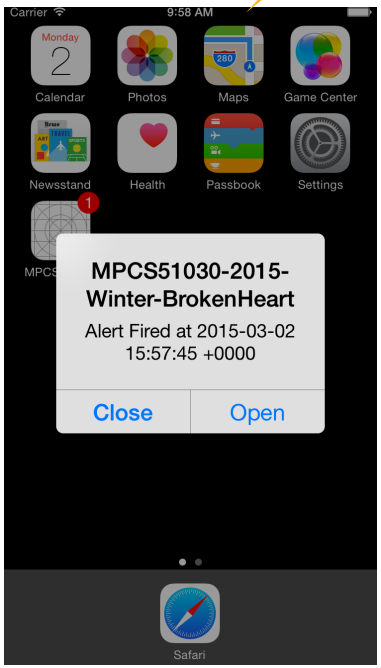


NOTIFICATIONS

Local



Push



Appears the same on devices

NOTIFICATIONS

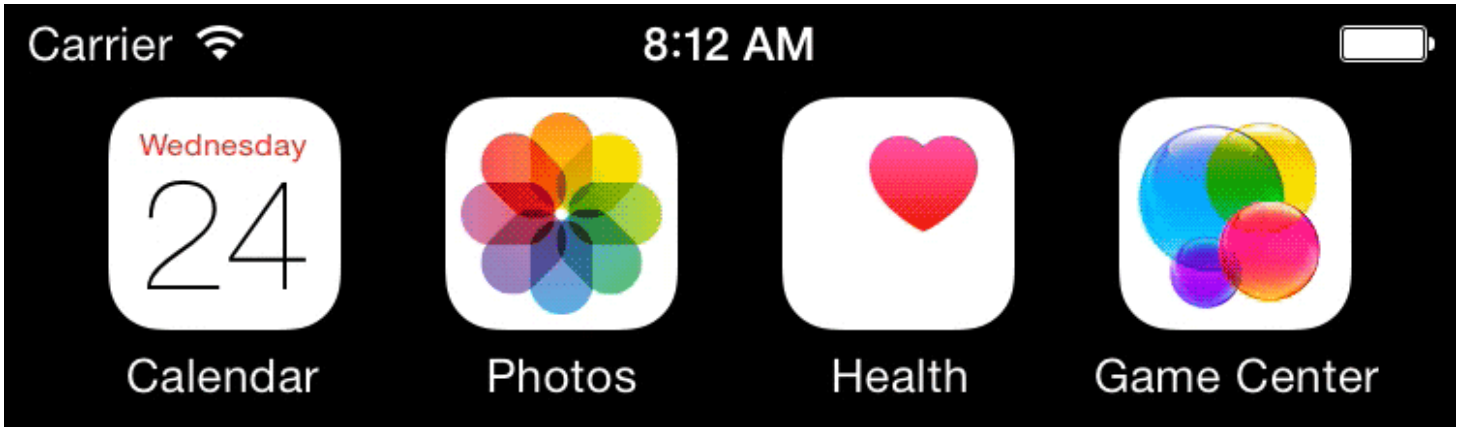
- Handling notifications in app delegate
  - application:didFinishLaunchingWithOptions:
  - application:didReceiveRemoteNotification:
  - application:didReceiveLocalNotification:

Options: local or push
UIApplicationLaunchOptionsLocalNotificationKey

NOTIFICATIONS

- iOS8 introduced interactive notifications
  - Engage with application without opening it

NOTIFICATIONS

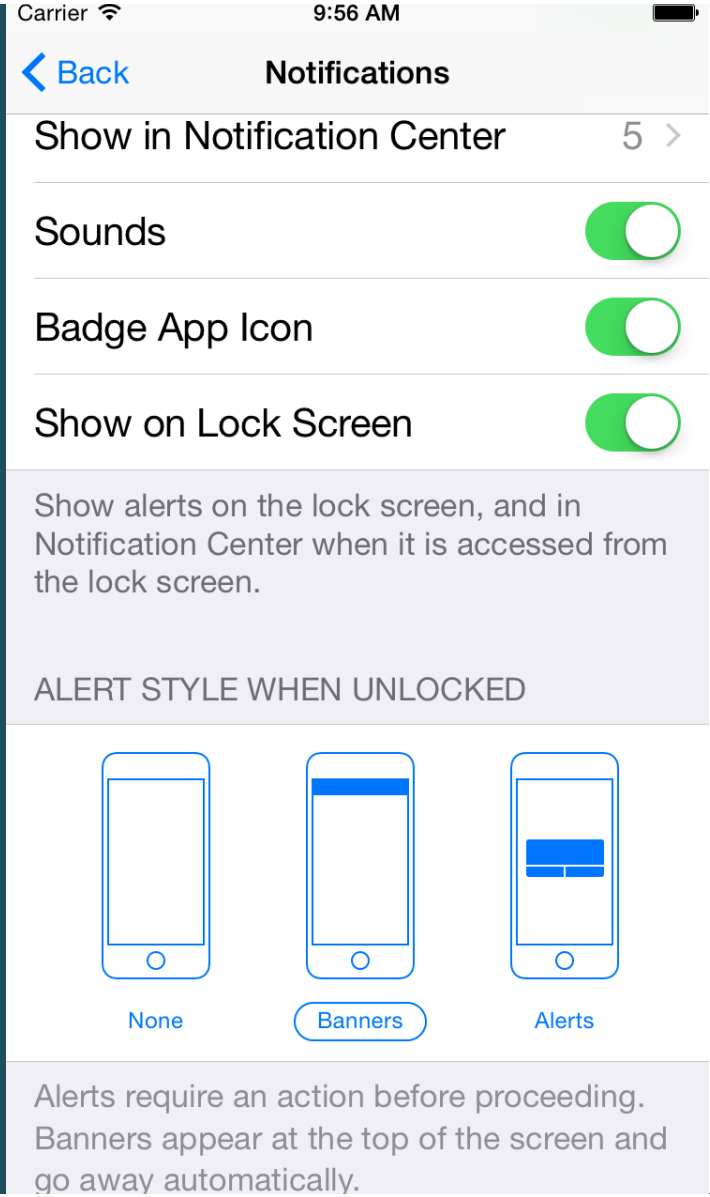


- iOS8 introduced interactive notifications
  - Engage with application without opening it
- iOS9 allows text input (eg. messages) from interactive notifications
- iOS10 unified the API for working with UserNotification Framework

PERMISSIONS

PERMISSIONS

- Remember that users are always in control
  - Keep them informed
  - Do not bother them
    - Of course you are trying to get their permission to bombard them with notifications)
  - Make sure you app still functions without notifications
    - Inform the user of consequences
    - "To be informed on the latest info, enable notifications in Settings" don't forget to show a link to settings

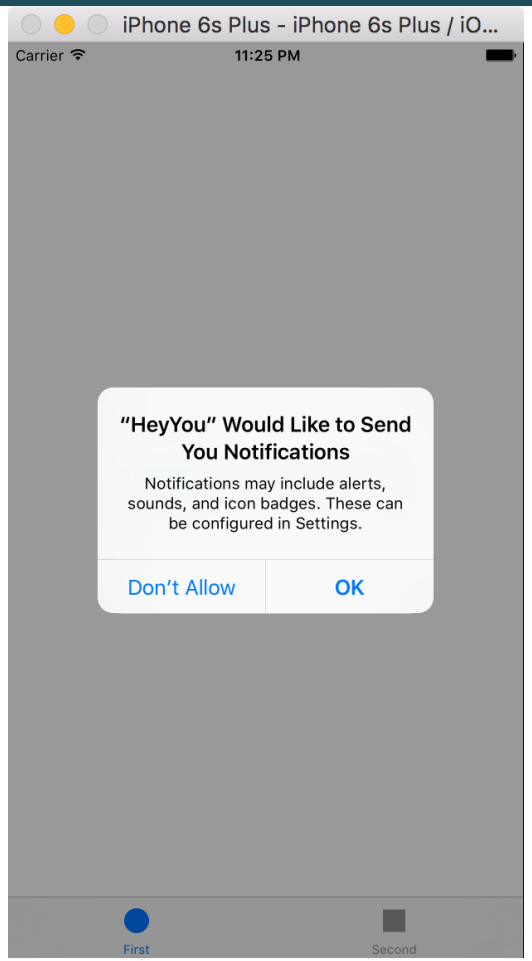


PERMISSIONS

```
let options: UNAuthorizationOptions = [.alert, .sound];
let center = UNUserNotificationCenter.current()
center.requestAuthorization(options: options) {
    (granted, error) in
    if !granted {
        print("Something went wrong")
    }
}
```

- Registering for notifications

PERMISSIONS



Standard Dialogue

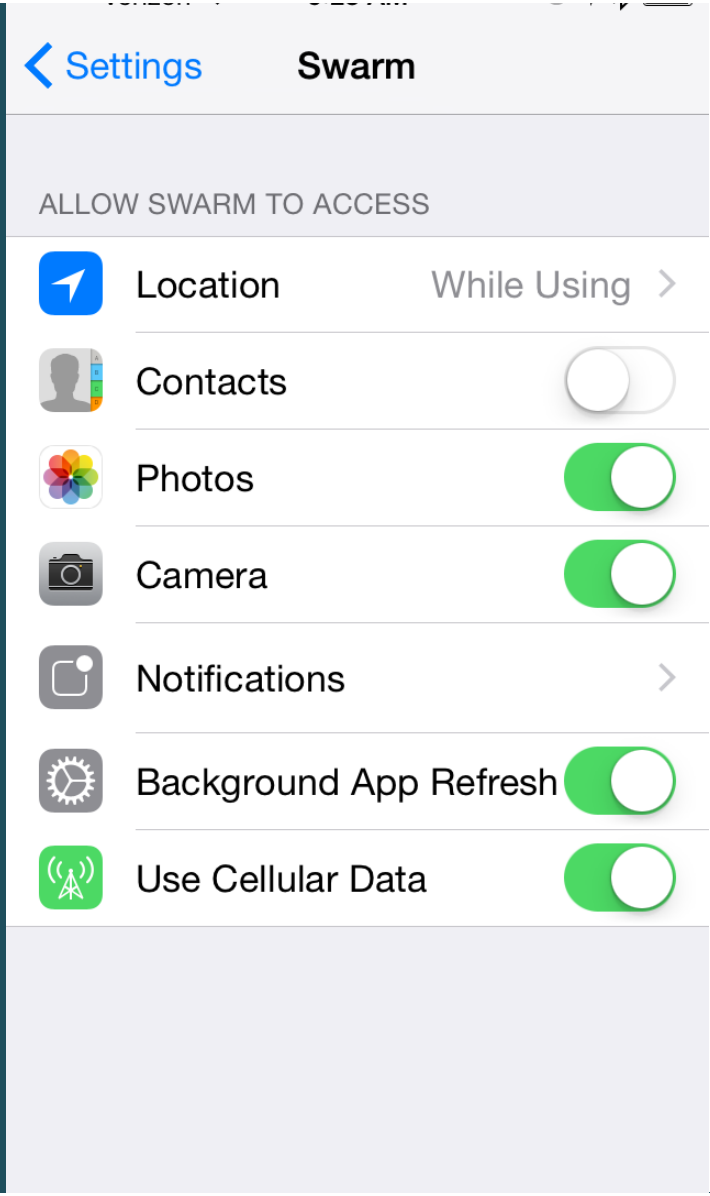
PERMISSIONS

- Consideration on notifications
  - Users may not accept to receive notifications
  - Users can change preferences in settings
- Be prepared to handle all potential use cases

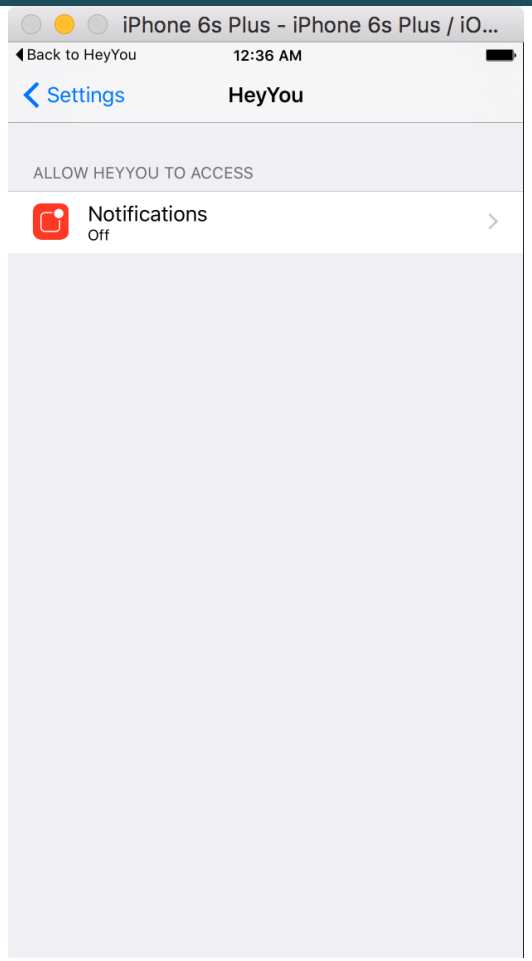
```
center.getNotificationSettings { (settings) in
    if settings.authorizationStatus != .authorized {
        // Notifications not allowed
    }
}
```

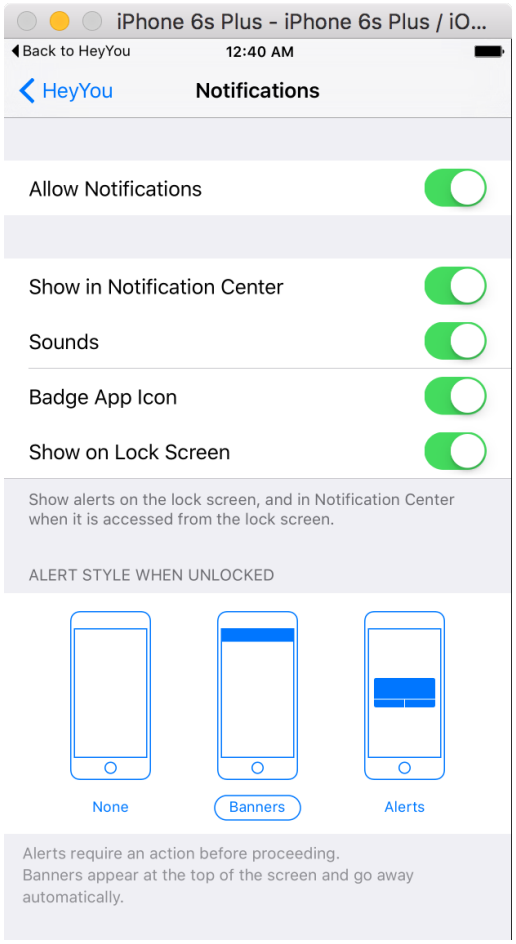
PERMISSIONS

- The old way
  - “Go to Settings > Privacy > Location > OUR_APP”



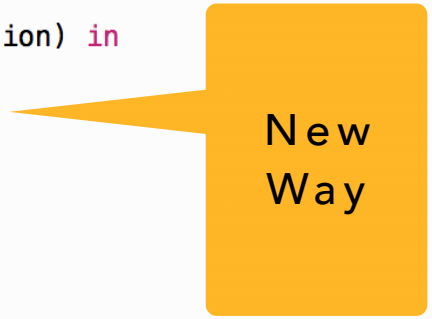
PERMISSIONS





PERMISSIONS

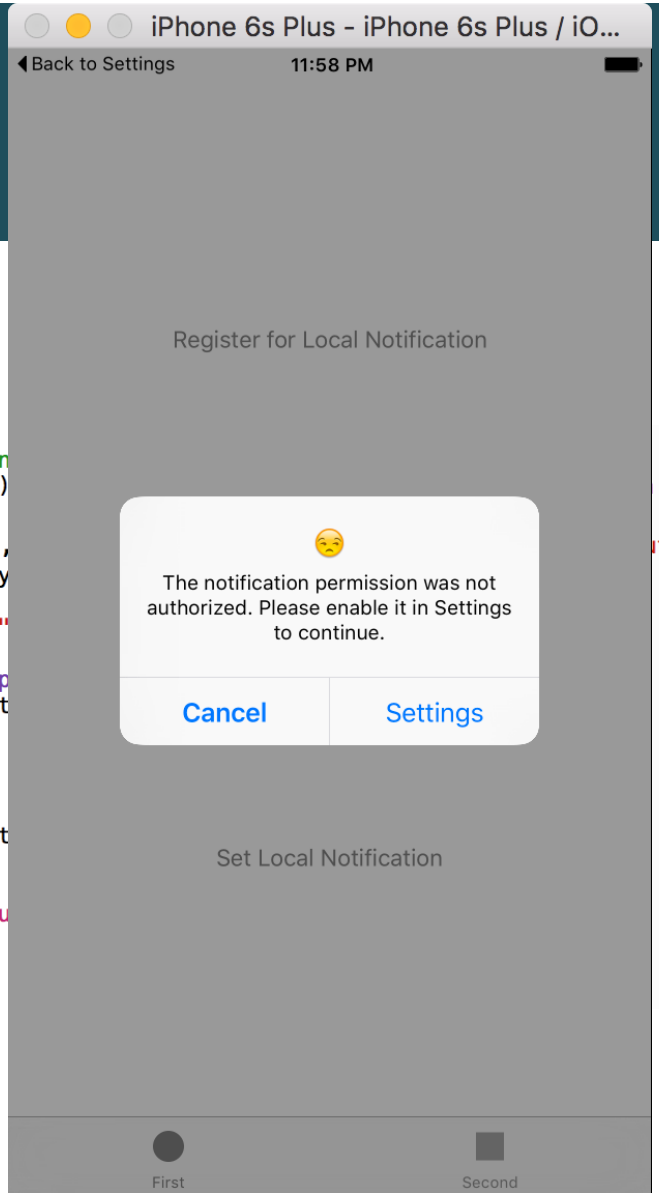
```swift
// Check if we have authorization to send notifications
guard let settings = UIApplication.sharedApplication().currentUserNotificationSettings() else { return }

// If we don't, let the use know and give them the option of going directly
// to the Settings screen for the application
if settings.types == .None {

  // Create an alert view controller
  let alertController = UIAlertController(title: "😒",
                                          message: "The notification permission was not authorized. Please enable it in Sett
                                          preferredStyle: .Alert)
  // Create to go to settings
  let settingsAction = UIAlertAction(title: "Settings", style: .Default) { (alertAction) in
    if let appSettings = NSURL(string: UIApplicationOpenSettingsURLString) {
      UIApplication.sharedApplication().openURL(appSettings)
    }
  }
  alertController.addAction(settingsAction)

  // Create cancel action that does nothing
  let cancelAction = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
  alertController.addAction(cancelAction)

  // Show the alert and exit early
  presentViewController(alertController, animated: true, completion: nil)
  return
}
```



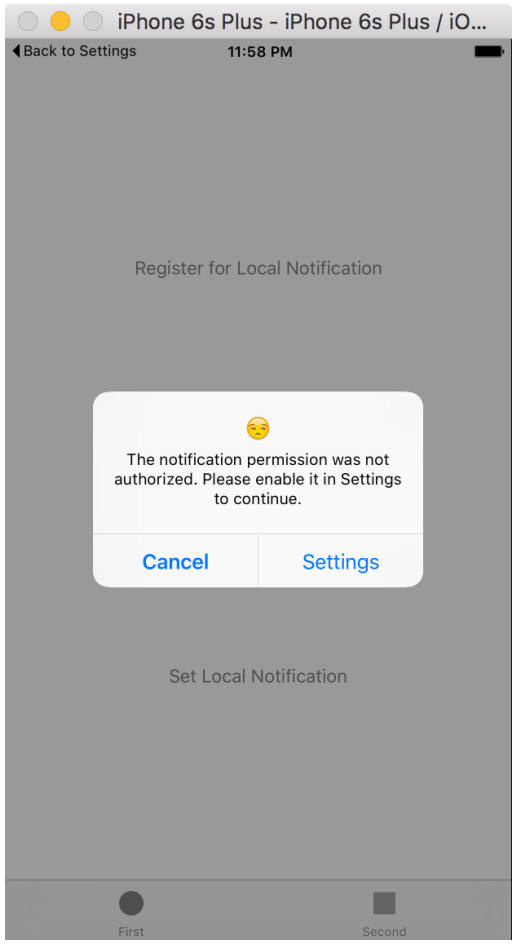
PERMISSIONS

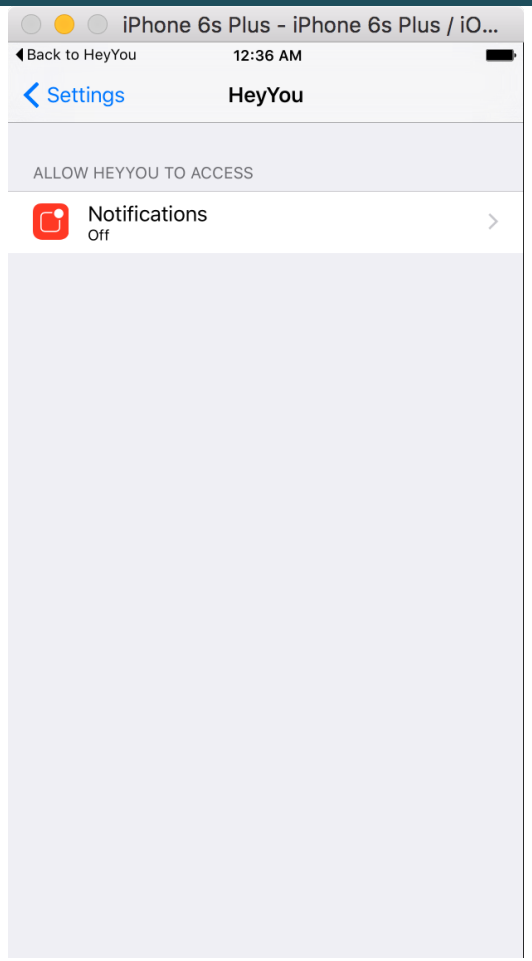
```
// Check if we have authorization to send notification
guard let settings = UIApplication.sharedApplication()
if settings.types == .None {
  let alertController = UIAlertController(title: "😒",
                                          preferredSty

  let settingsAction = UIAlertAction(title: "Settings"

    if let appSettings = NSURL(string: UIApplicationOp
      UIApplication.sharedApplication().openURL(appSet
    }
  }
  alertController.addAction(settingsAction)

  let cancelAction = UIAlertAction(title: "Cancel", st
  alertController.addAction(cancelAction)

  presentViewController(alertController, animated: tru
  return
}
```

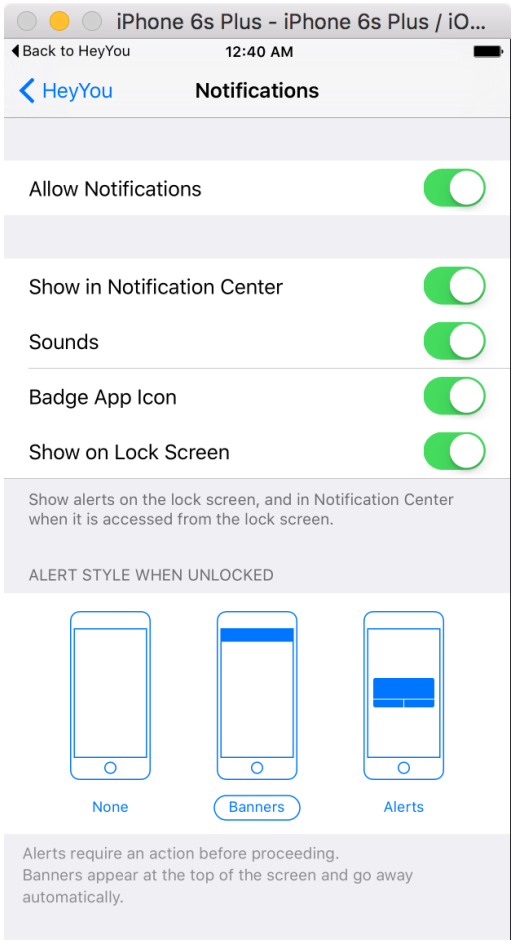
```
}

thorized. Please enable it in Settings to continue.",
```



PERMISSIONS



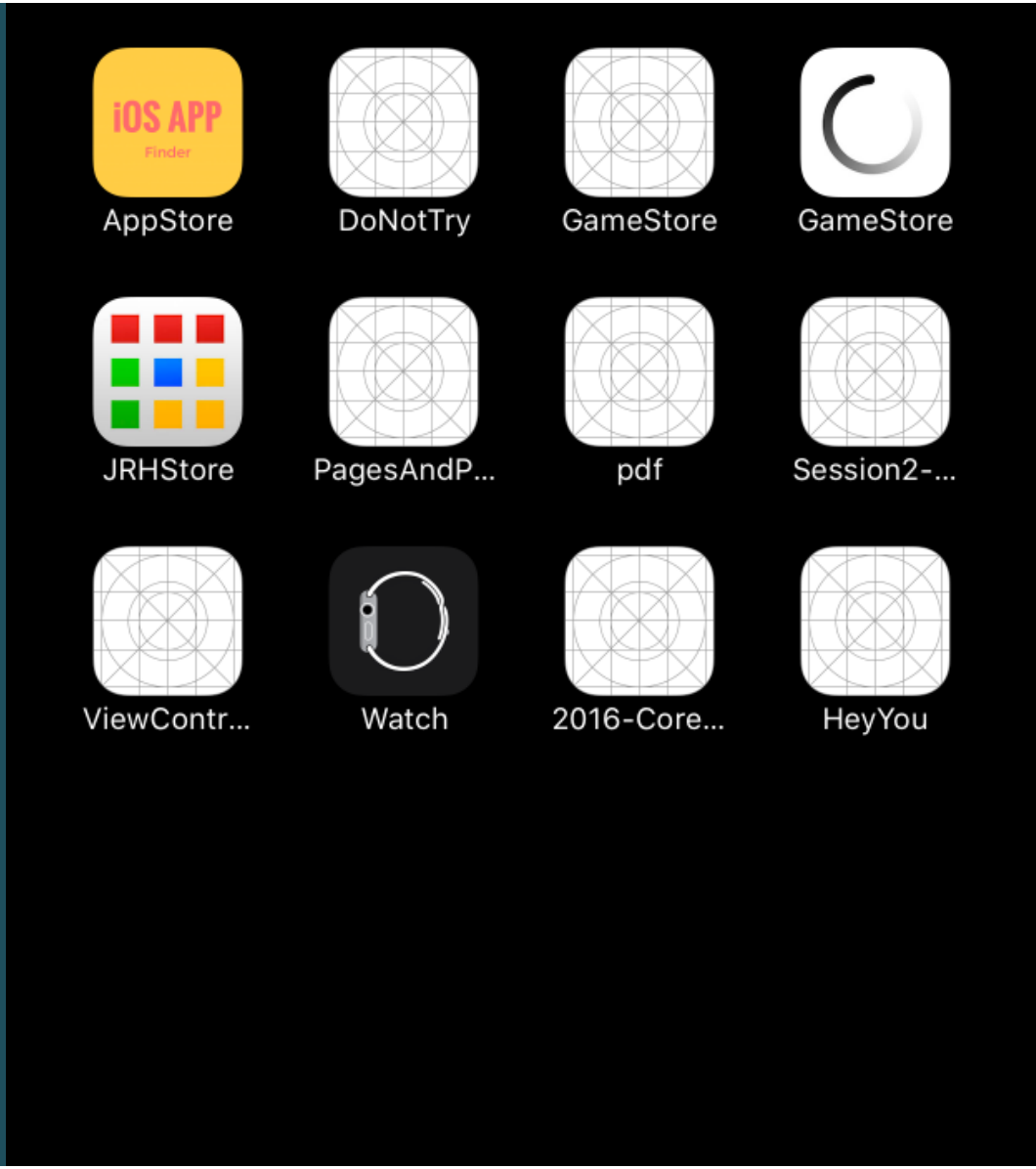




Local Notifications

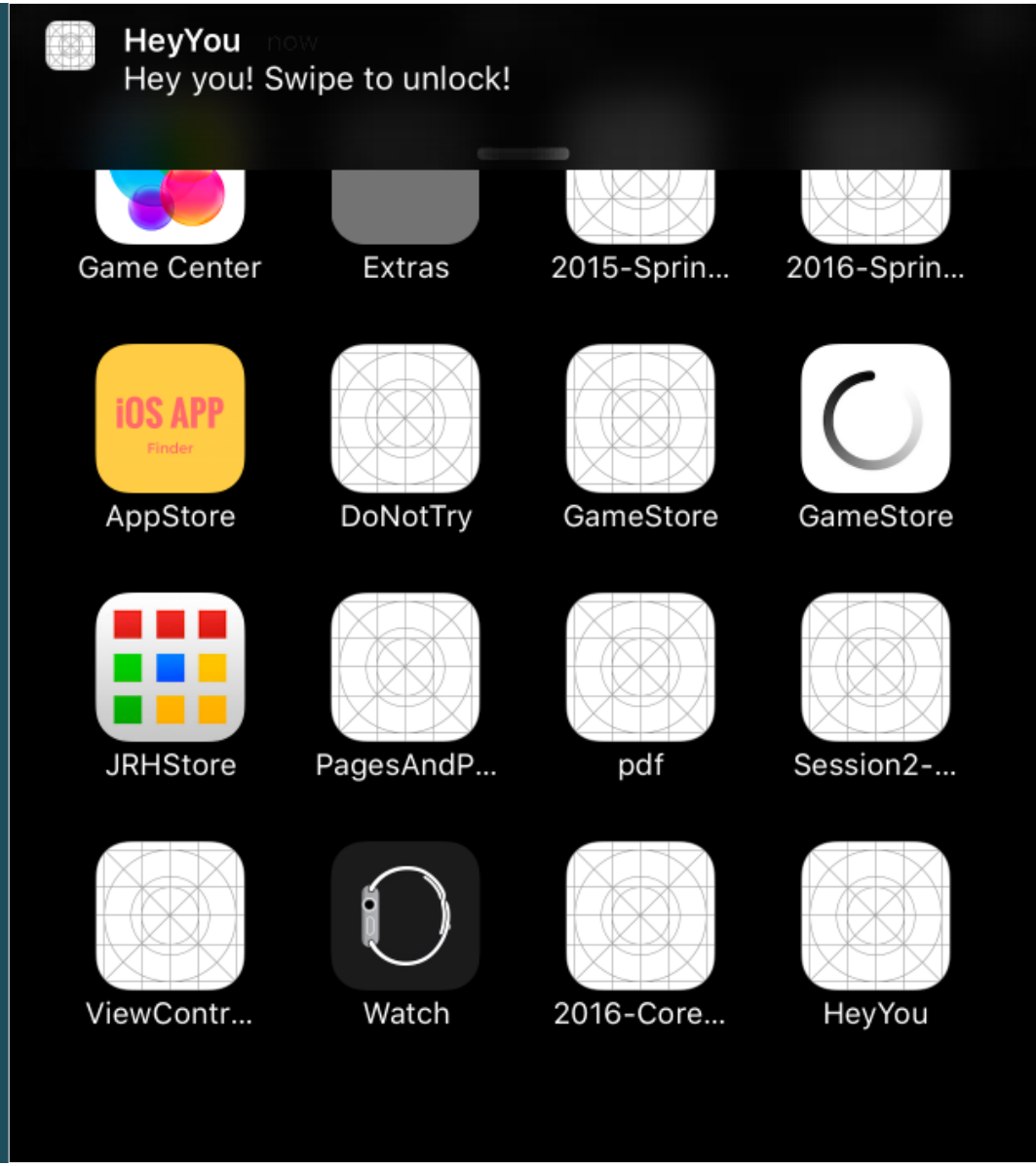
LOCAL NOTIFICATIONS

- Local notifications are delivered by the iOS device



LOCAL NOTIFICATIONS

- Messages are handled differently depending on the state of the application
  - Running
  - Background
  - Terminated
- Developer responsibility to handle each case



LOCAL NOTIFICATIONS

Launch app from notification

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {

  if let notification:UILocalNotification = launchOptions?[UIApplicationLaunchOptionsLocalNotificationKey] as? UILocalNotification {
    print("Launch from local notification: \(notification)")
  }

  return true
}
```

- Test for type of notification when launching from terminated state
- Different behavior
  - Go to a view controller
  - Set some date

LOCAL NOTIFICATIONS

Handle notification from suspended

```
/// Called when the app is in the background
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                            withCompletionHandler completionHandler: @escaping () -> Void) {
  print(response)
  if response.actionIdentifier == UNNotificationDismissActionIdentifier {
    // The user dismissed the notification without taking action
  }
  else if response.actionIdentifier == UNNotificationDefaultActionIdentifier {
    // The user launched the app
    print("AppDelegate: Did Receive: \(response)")
  }
  completionHandler()
}
```

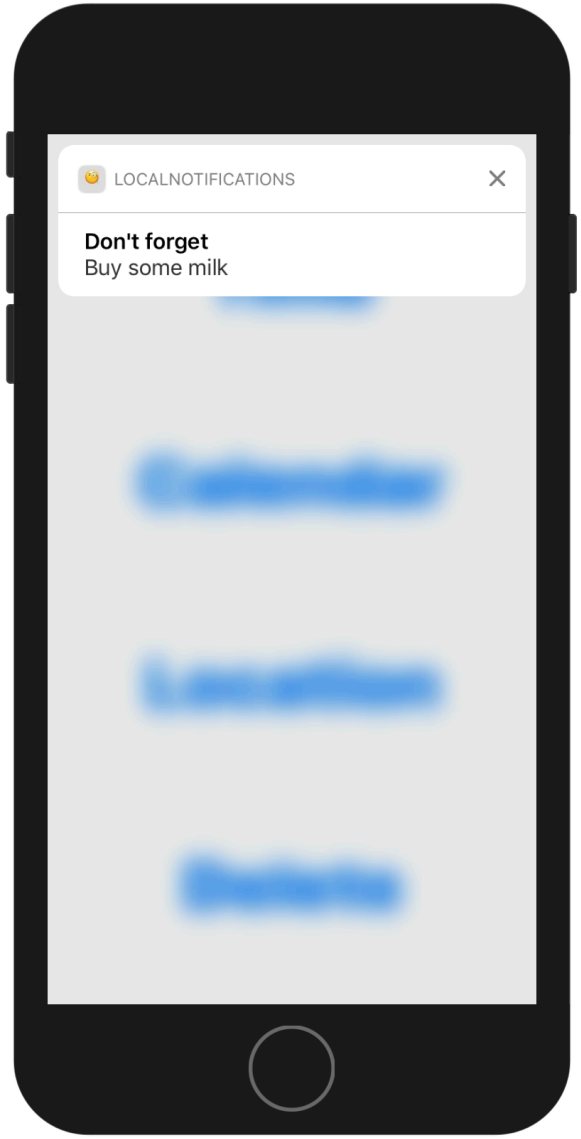
LOCAL NOTIFICATIONS

Have to set delegate early

```
/// Called when the app is in the background
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                            withCompletionHandler completionHandler: @escaping () -> Void) {
  print(response)
  if response.actionIdentifier == UNNotificationDismissActionIdentifier {
    // The user dismissed the notification without taking action
  }
  else if response.actionIdentifier == UNNotificationDefaultActionIdentifier {
    // The user launched the app
    print("AppDelegate: Did Receive: \(response)")
  }
  completionHandler()
}
```

LOCAL NOTIFICATIONS

- Remember to handle all the states (if necessary)
  - App in foreground
  - App suspended
  - App terminated



Scheduling Local Notifications

SCHEDULING NOTIFICATIONS

- Notification request contain content and a trigger

- `title` : String containing the primary reason for the alert.
- `subtitle` : String containing an alert subtitle (if required)
- `body` : String containing the alert message text
- `badge` : Number to show on the app's icon.
- `sound` : A sound to play when the alert is delivered. Use `UNNotificationSound.default()` or create a custom sound from a file.
- `launchImageName` : name of a launch image to use if your app is launched in response to a notification.
- `userInfo` : A dictionary of custom info to pass in the notification
- `attachments` : An array of `UNNotificationAttachment` objects. Use to include audio, image or video content.

Content sent with notification

SCHEDULING NOTIFICATIONS

```
let content = UNMutableNotificationContent()
content.title = "Don't forget"
content.body = "Buy some milk"
content.sound = UNNotificationSound.default()
```

- Notification request contain content and a trigger

SCHEDULING NOTIFICATIONS

- Triggers
  - Time
  - Calendar
  - Location

SCHEDULING NOTIFICATIONS

- Schedule a notification for a number of seconds later

Time trigger

```
let content = UNMutableNotificationContent()
content.title = "Don't forget"
content.body = "Buy some milk"
content.sound = UNNotificationSound.default()

_ = UNTimeIntervalNotificationTrigger(timeInterval: 300,
                                      repeats: false)
```

SCHEDULING NOTIFICATIONS

- Trigger at a specific date and time
- The trigger is created using a date components object which makes it easier for certain repeating intervals

Calendar trigger

```
let content = UNMutableNotificationContent()
content.title = "Don't forget"
content.body = "Buy some milk"
content.sound = UNNotificationSound.default()


let date = Date(timeIntervalSinceNow: 3600)
let triggerDate = Calendar.current.dateComponents([.year,.month,.day,.hour,.minute,.second,],
                                                  from: date)

let trigger = UNCalendarNotificationTrigger(dateMatching: triggerDate,
                                            repeats: false)
```

SCHEDULING NOTIFICATIONS

```
let triggerDaily = Calendar.current.dateComponents([.hour,.minute,.second,], from: date)
let trigger = UNCalendarNotificationTrigger(dateMatching: triggerDaily, repeats: true)


let triggerWeekly = Calendar.current.dateComponents([.weekday,.hour,.minute,.second,], from: date)
let trigger = UNCalendarNotificationTrigger(dateMatching: triggerWeekly, repeats: true)
```

- Repeating notifications

SCHEDULING NOTIFICATIONS

Location trigger

```
let trigger = UNLocationNotificationTrigger(triggerWithRegion:region, repeats:false)
```

- Trigger when a user enters or leaves a geographic region
- The region is specified through a CoreLocation `CLRegion`
- You need to get permissions just like `CoreLocation`

SCHEDULING NOTIFICATIONS

```
UNUserNotificationCenter.current().
    getPendingNotificationRequests {
    requests in
    print(requests)
}
```

- List all the current notifications

SCHEDULING NOTIFICATIONS

```
let center = UNUserNotificationCenter.current()
center.removeAllPendingNotificationRequests()
```

- Delete notifications

SCHEDULING NOTIFICATIONS

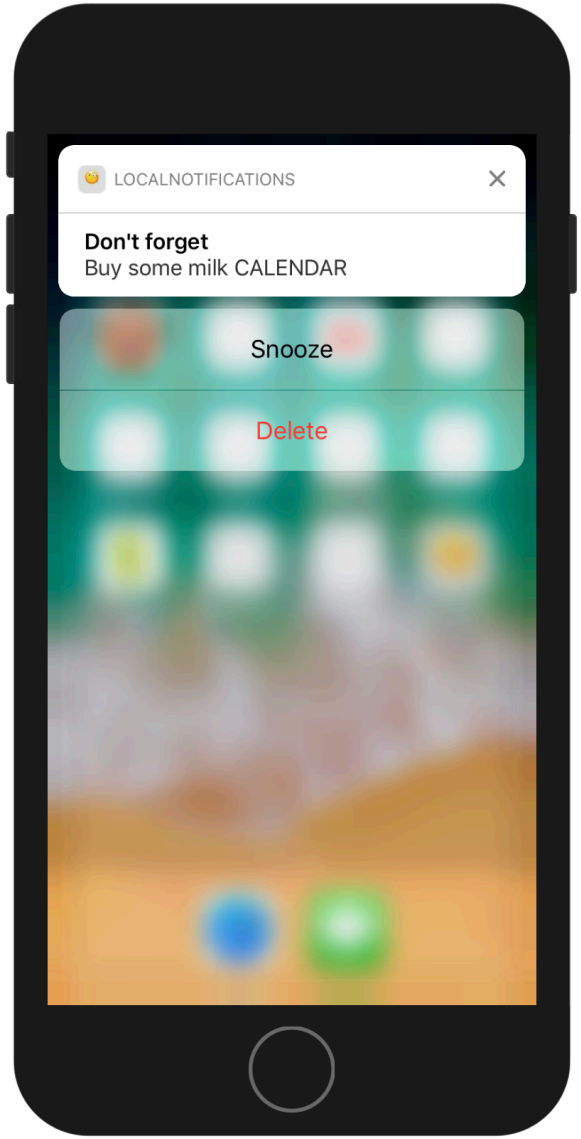
```
@IBAction func tapDelete(_ sender: Any) {
  listNotification()

  let center = UNUserNotificationCenter.current()
  center.removeAllPendingNotificationRequests()

  listNotification()
}
```

- Delete notifications

Notification Actions

NOTIFICATION ACTIONS

- Define custom actions for your notifications



NOTIFICATION ACTIONS

```
// Actions
let snoozeAction = UNNotificationAction(identifier: "Snooze", title: "Snooze", options: [])
let deleteAction = UNNotificationAction(identifier: "DeleteAction", title: "Delete", options:
    [.destructive])
let category = UNNotificationCategory(identifier: "CalendarCategory",
                                      actions: [snoozeAction,deleteAction],
                                      intentIdentifiers: [], options: [])
UNUserNotificationCenter.current().setNotificationCategories([category])
```

NOTIFICATION ACTIONS

```
let content = UNMutableNotificationContent()
content.title = "Don't forget"
content.body = "Buy some milk CALENDAR"
content.sound = UNNotificationSound.default()
content.categoryIdentifier = "CalendarCategory"

let date = Date(timeIntervalSinceNow: 10)
let triggerDate = Calendar.current.dateComponents([.year,.month,.day,.hour,.minute,.second,], from:
    date)
let trigger = UNCalendarNotificationTrigger(dateMatching: triggerDate, repeats: false)

// Actions
let snoozeAction = UNNotificationAction(identifier: "Snooze", title: "Snooze", options: [])
let deleteAction = UNNotificationAction(identifier: "DeleteAction", title: "Delete", options:
    [.destructive])
let category = UNNotificationCategory(identifier: "CalendarCategory",
                                      actions: [snoozeAction,deleteAction],
                                      intentIdentifiers: [], options: [])
UNUserNotificationCenter.current().setNotificationCategories([category])
```

NOTIFICATION ACTIONS

```
/// Called when the app is in the background
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                            withCompletionHandler completionHandler: @escaping () -> Void) {

  print("AppDelegate: Did Receive: \(response)")

  // Determine the user action
   switch response.actionIdentifier {
  case UNNotificationDismissActionIdentifier:
    print("Dismiss Action")
  case UNNotificationDefaultActionIdentifier:
    print("Default")
  case "Snooze":
    print("Snooze")
  case "DeleteAction":
    print("Delete")
  default:
    print("Unknown action")
  }
  completionHandler()
}
```

Switch on the actionIdentifier

Push Notifications

PUSH NOTIFICATIONS

- Sent from Apple's servers to device
- Apple Push Notification Service (APNS) is the gateway for push notification
  - SSL certificate
  - Provisioning
  - Networking code
  - Device tokes

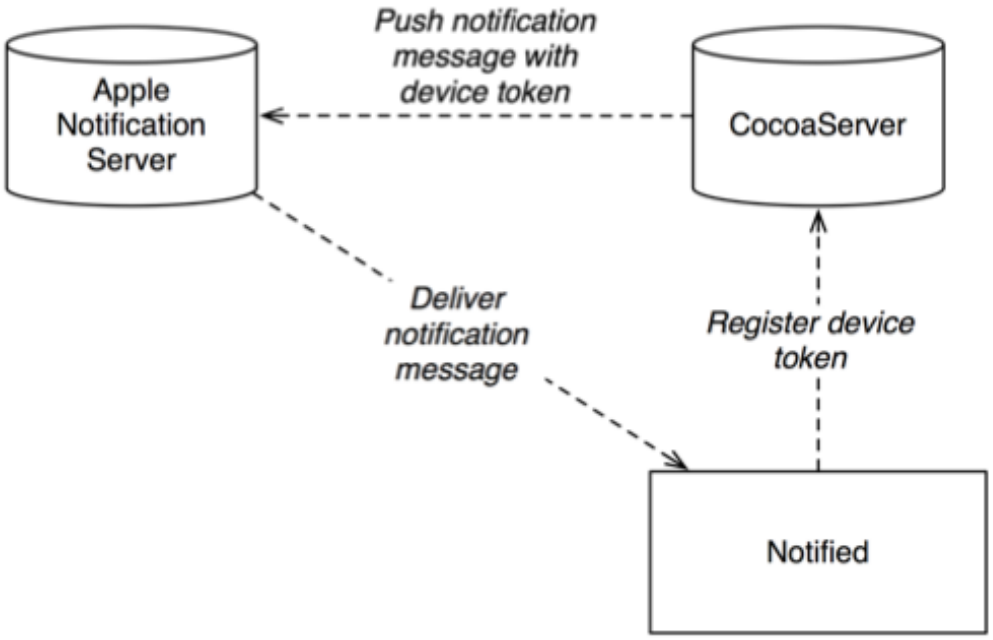
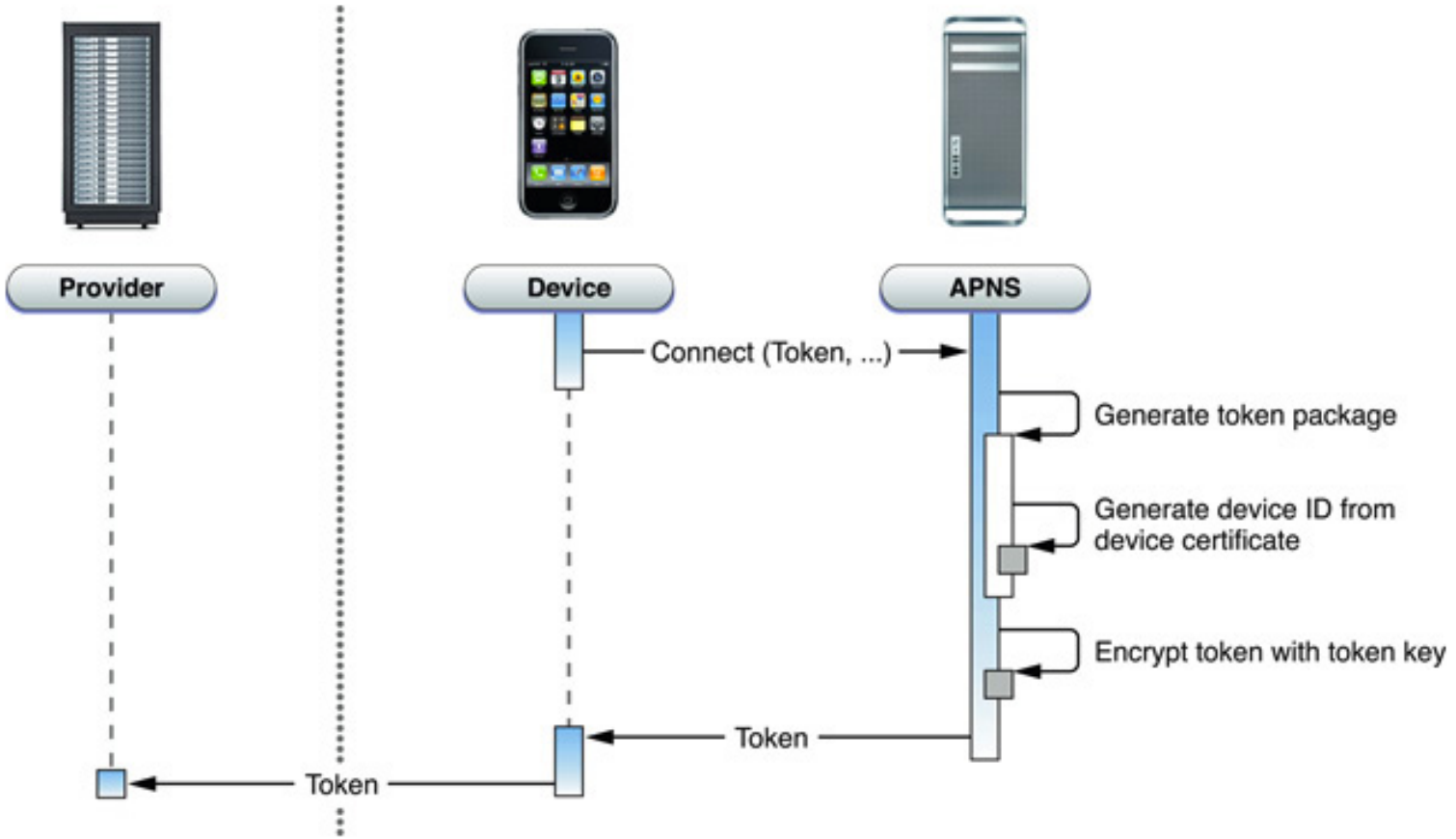


image via Big Nerd Ranch iOS Programming

PUSH NOTIFICATIONS

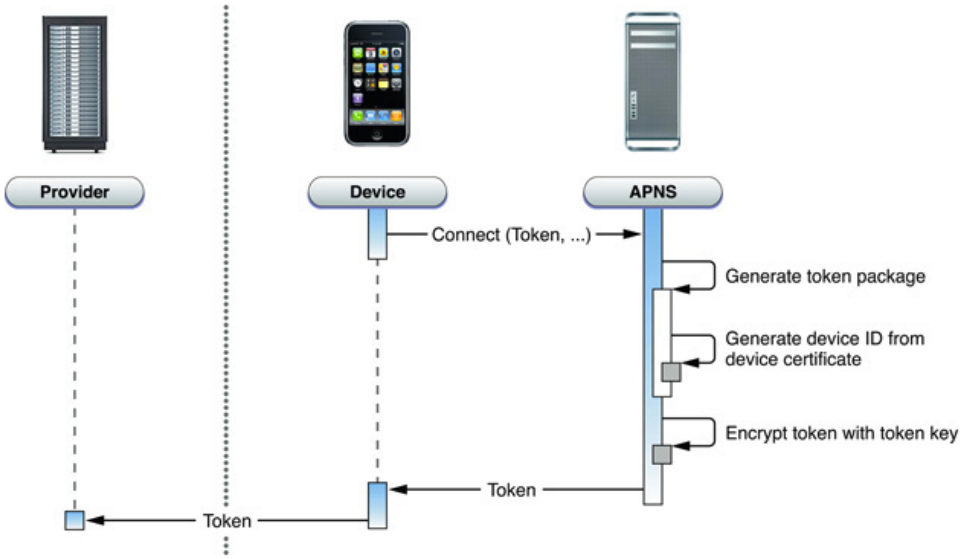
1. An app enables push notification (the user has to confirm that he wishes to receive these notifications)
2. The app receives a “device token”
3. The app sends the device token to your server
4. The server sends a push notification to the Apple Push Notification Service
5. APNS sends the push notification to the user’s device

PUSH NOTIFICATIONS



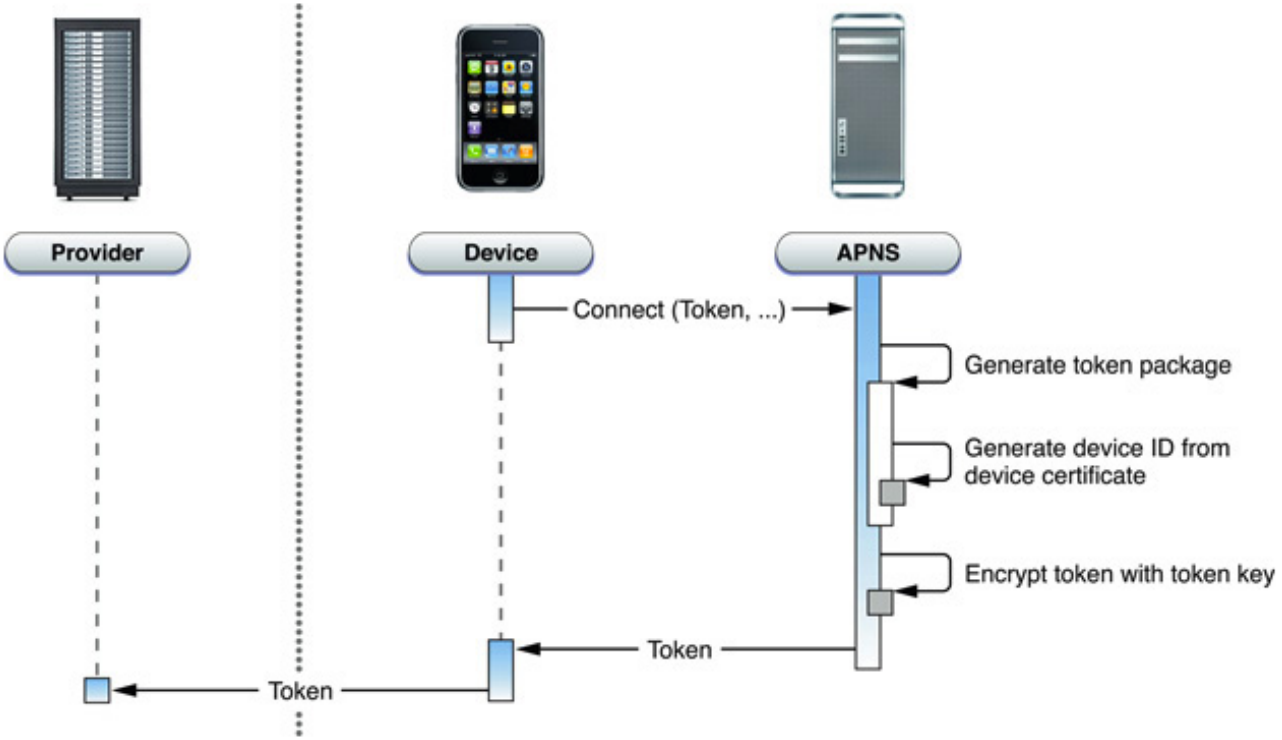
PUSH NOTIFICATION

- Requirements for push notifications
  - iOS device
  - Paid iOS Developer membership
  - Server
    - Ability to run background processes
    - Install SSL certificates
    - Make outing TLS connections on non-standard ports



PUSH NOTIFICATION

- Many third party solutions for push notifications
  - Parse
  - Firebase
  - App Engine
  - ...



PUSH NOTIFICATIONS

```
{"aps": {
    "badge": 100,
    "alert": "Push Notification Test",
    "sound": "SoundFile.aif"},
    "type":"websiteUrl",
    "data":"http://mySite.com"
}
```

Manage the badges in your code (not incremental)

Include sound files in your bundle

- Push notification payload
  - JSON dictionary
  - <256 bytes

Pass any key-values in the notification

PUSH NOTIFICATION

- Receiving notification require app delegate methods to be implemented
- Code path is the same for local notification

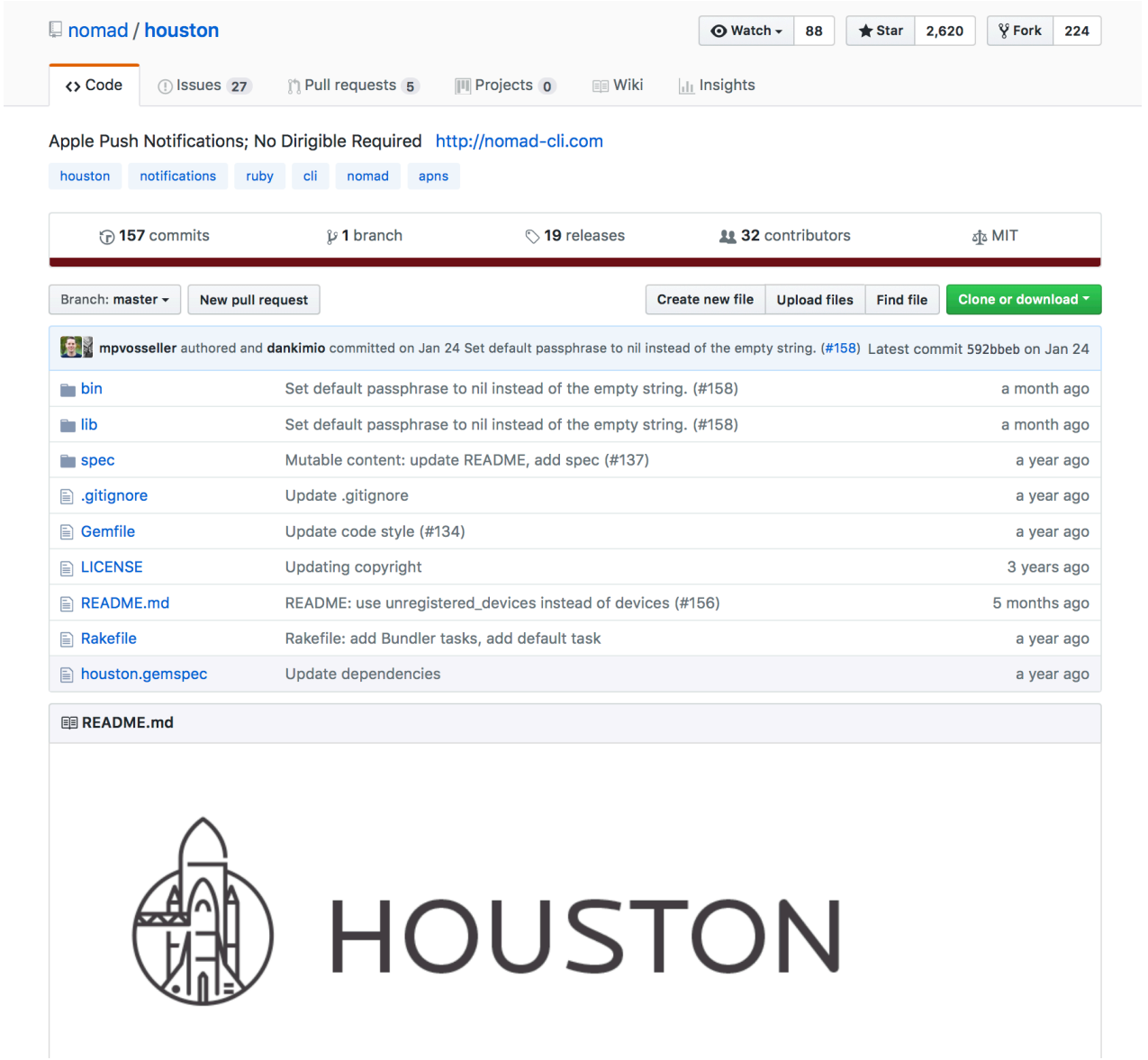
```swift
/// Called when the app is in the background
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                            withCompletionHandler completionHandler: @escaping () -> Void) {
  print(response)
  if response.actionIdentifier == UNNotificationDismissActionIdentifier {
    // The user dismissed the notification without taking action
  }
  else if response.actionIdentifier == UNNotificationDefaultActionIdentifier {
    // The user launched the app
    print("AppDelegate: Did Receive: \(response)")
  }
  completionHandler()
}
```

PUSH NOTIFICATION

- There are many third party services for setting up push notifications

PUSH NOTIFICATION

- There are many third party services for setting up push notifications



NOTIFICATIONS WRAP-UP

- Notifications keep users engage by opening your app
- Notifications are not guaranteed to be delivered by Apple
- Tips and tricks:
  - Push was important before multi-tasking in iOS4, but many uses can be implemented as local notifications
  - Third-party option for remote notifications
  - Use CloudKit for remote notifications via subscriptions
- Don’t abuse notifications…this is one thing that Apple seems to regularly enforce in the App Store Guidelines
  - “Push notifications can not be used for advertising”

CLOUDKIT API

WEB SERVICE API

- Javascript API
- Matches native CloudKit API
- No intermediate servers
- New notes web app built with CloudKit JS

JS

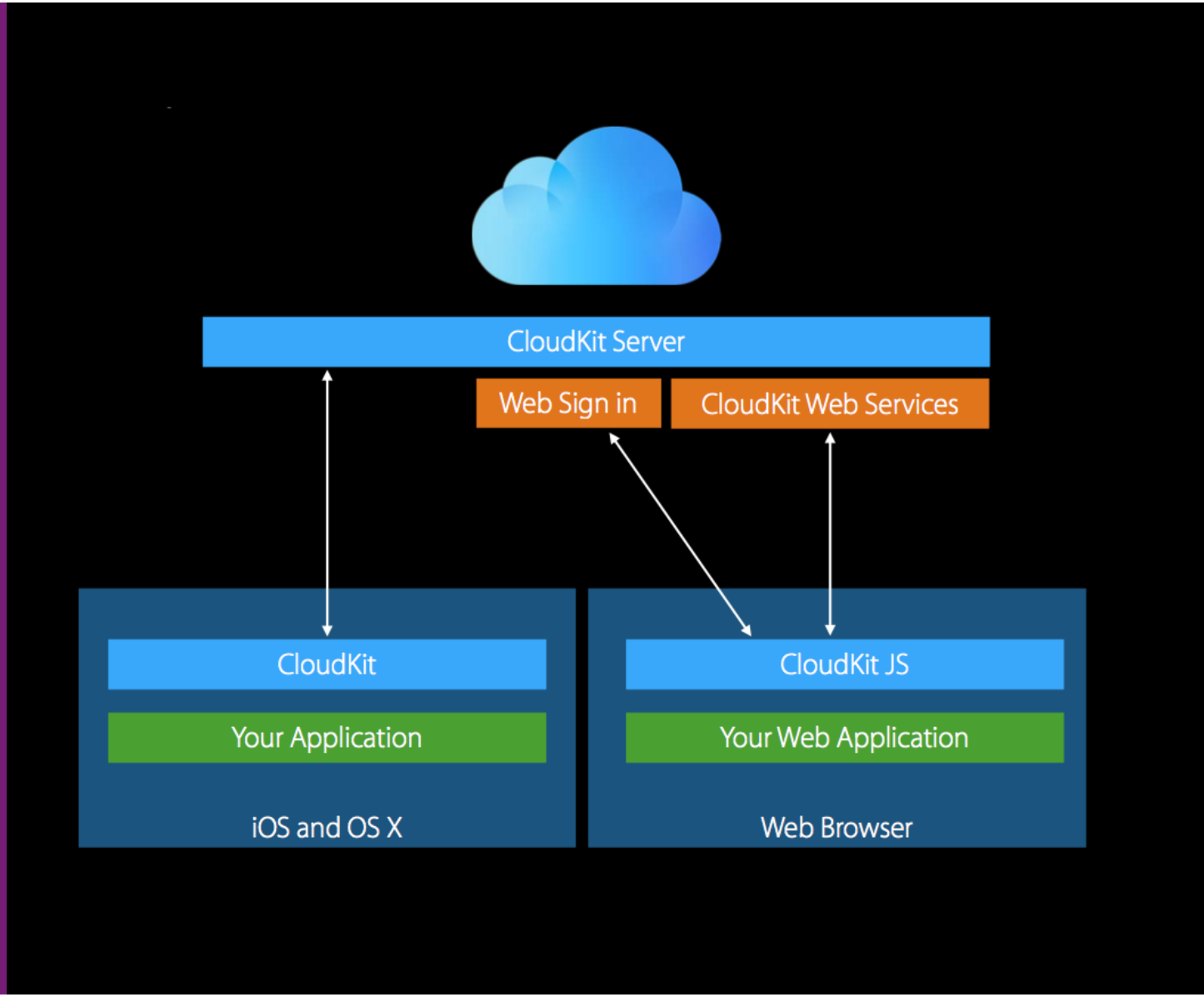
```
<SCRIPT SRC="HTTPS://CDN.APPLE-CLOUDKIT.COM/CK/1/CLOUDKIT.JS" />
```

WEB SERVICE API

- Public and private database access
- Record operations
- Assets
- Query
- Subscriptions and notifications
- User discoverability
- Sync

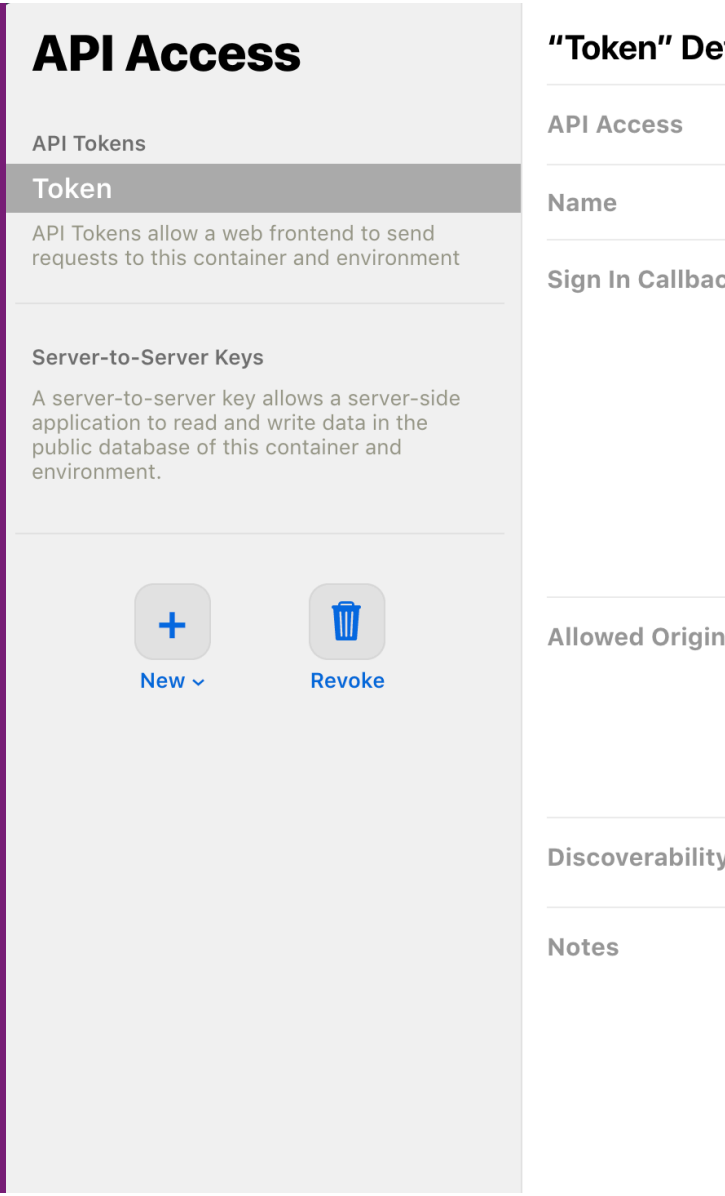
WEB SERVICES

- New server-to-server capabilities
- Add servers to cover the areas that cloud kit doesn't cover



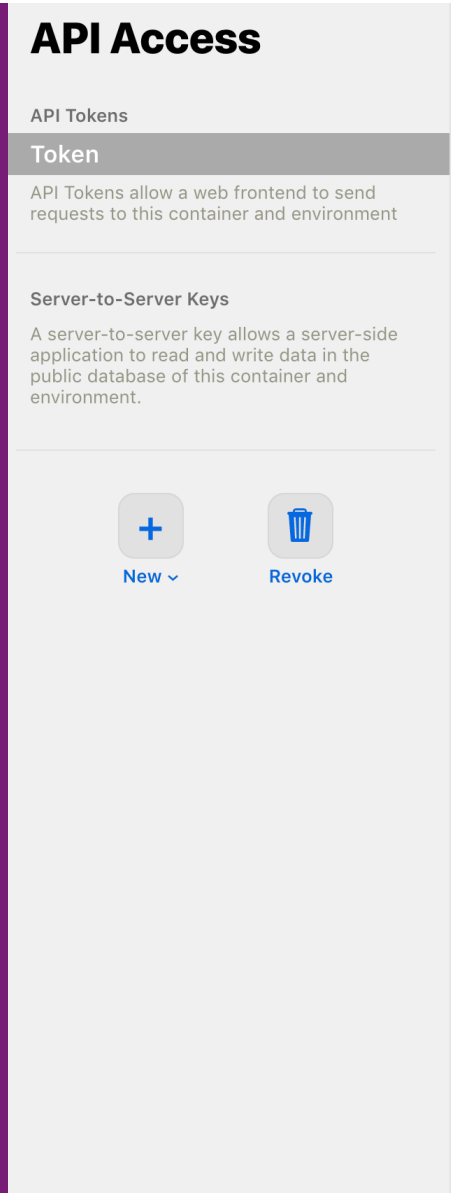
CLOUDKIT JS

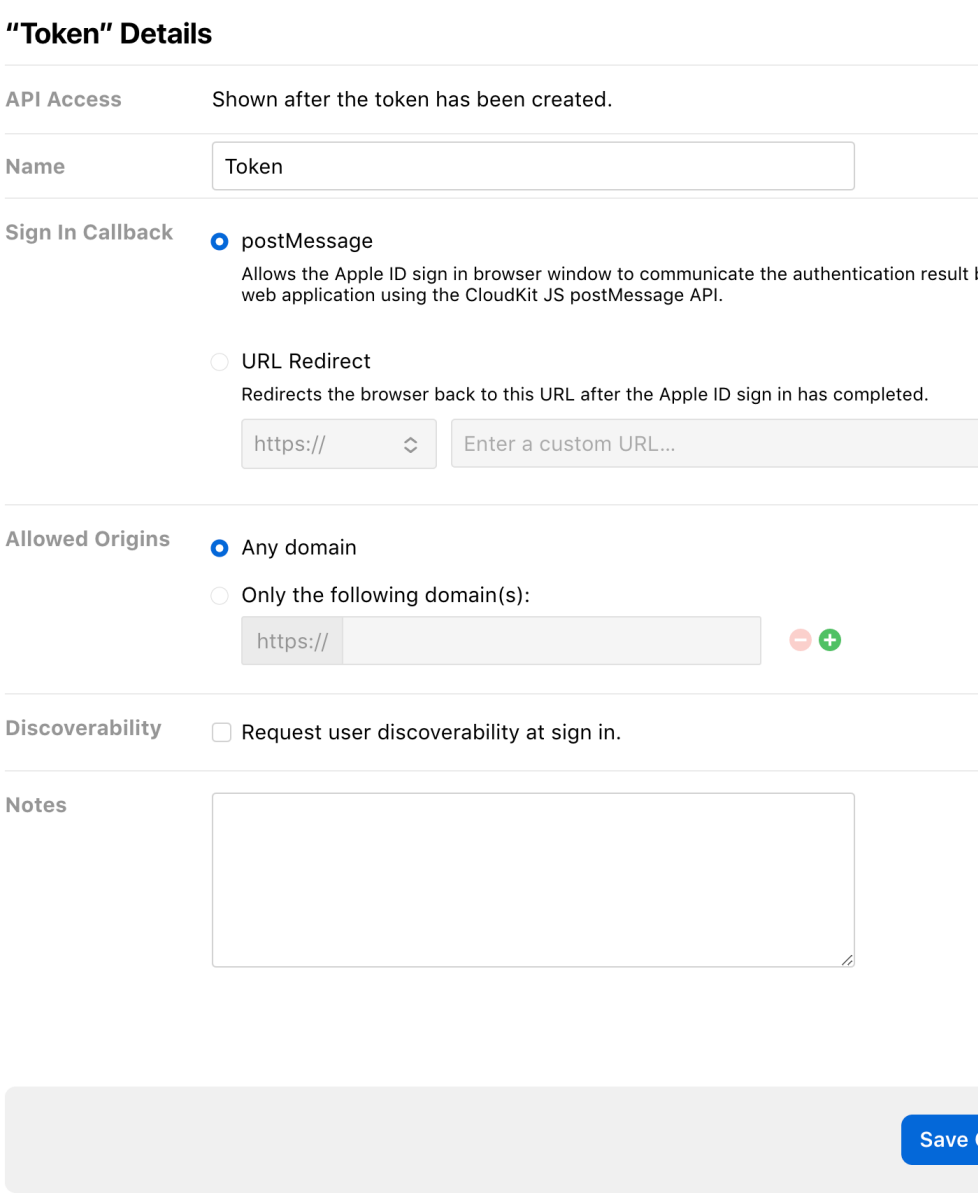
- An API token is used to allow a web front-end to send requests to this container and environment
- A server-to-server key allows a server-side application to read and write data in the public database of this container & environment



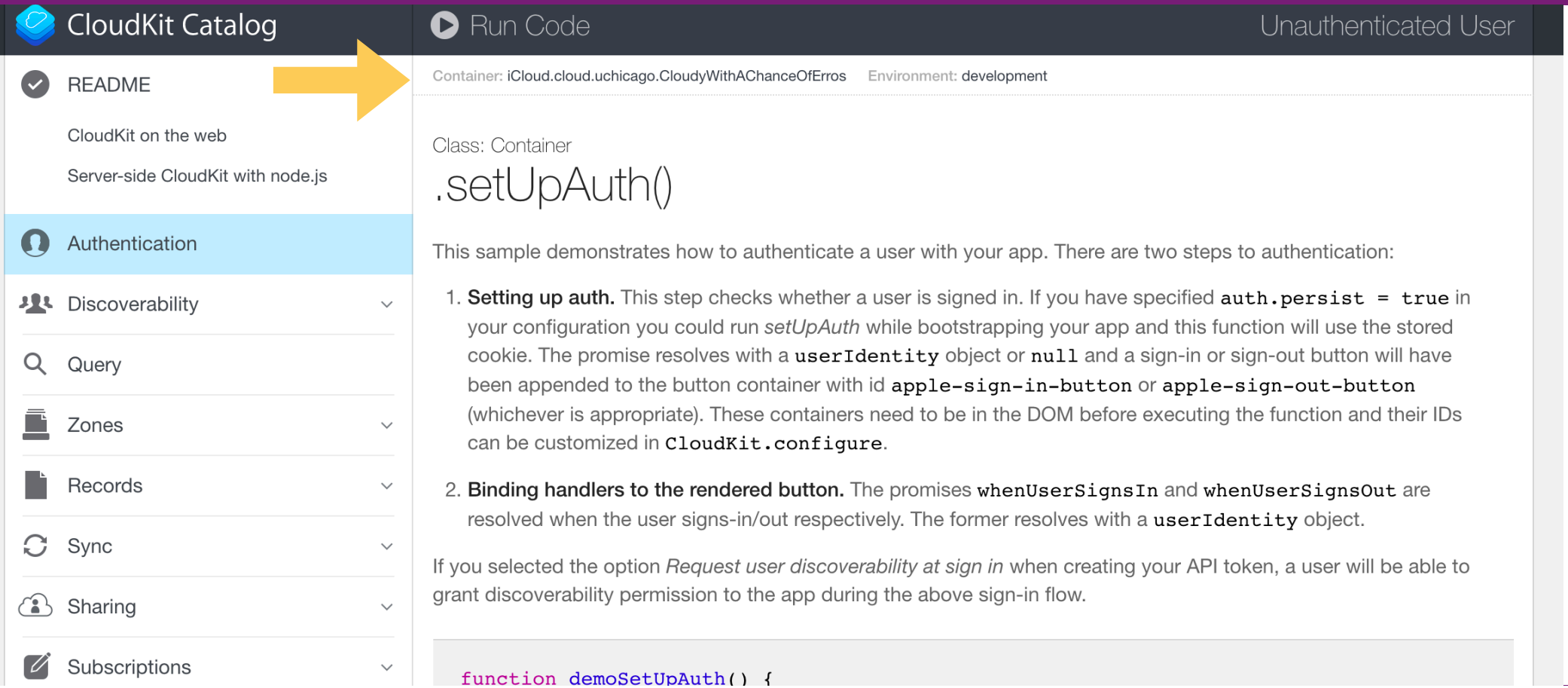
CLOUDKIT JS

- Create a new token



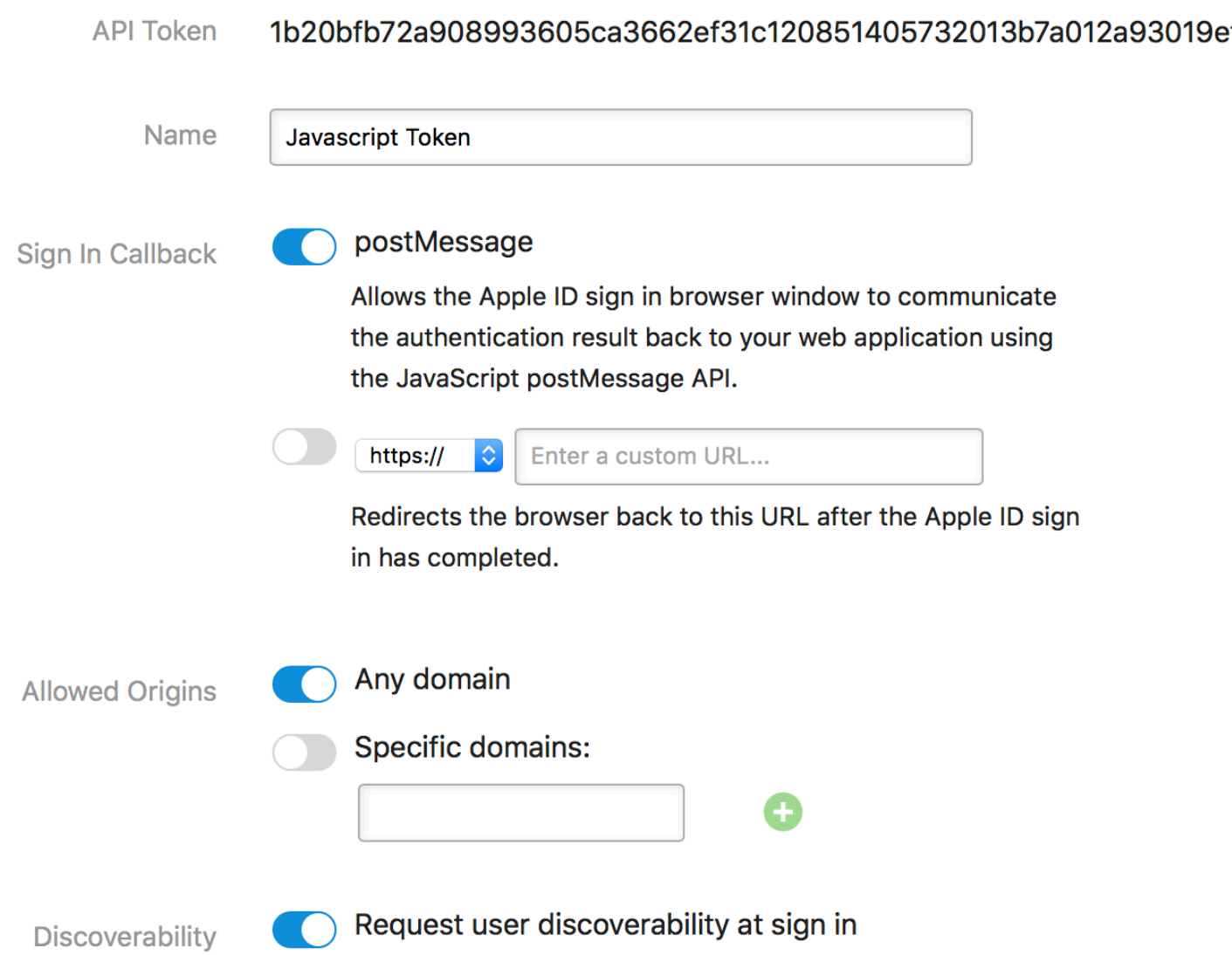


CLOUDKIT JS


CloudKit Catalog
Run Code
Unauthenticated User
README
CloudKit on the web
Server-side CloudKit with node.js
Authentication
Discoverability
Query
Zones
Records
Sync
Sharing
Subscriptions
Container: iCloud.cloud.uchicago.CloudyWithAChanceOfErros
Environment: development
Class: Container
.setUpAuth()
This sample demonstrates how to authenticate a user with your app. There are two steps to authentication:
1. Setting up auth. This step checks whether a user is signed in. If you have specified auth.persist = true in your configuration you could run setUpAuth while bootstrapping your app and this function will use the stored cookie. The promise resolves with a userIdentity object or null and a sign-in or sign-out button will have been appended to the button container with id apple-sign-in-button or apple-sign-out-button (whichever is appropriate). These containers need to be in the DOM before executing the function and their IDs can be customized in CloudKit.configure.
2. Binding handlers to the rendered button. The promises whenUserSignsIn and whenUserSignsOut are resolved when the user signs-in/out respectively. The former resolves with a userIdentity object.
If you selected the option Request user discoverability at sign in when creating your API token, a user will be able to grant discoverability permission to the app during the above sign-in flow.
function demoSetUpAuth() {

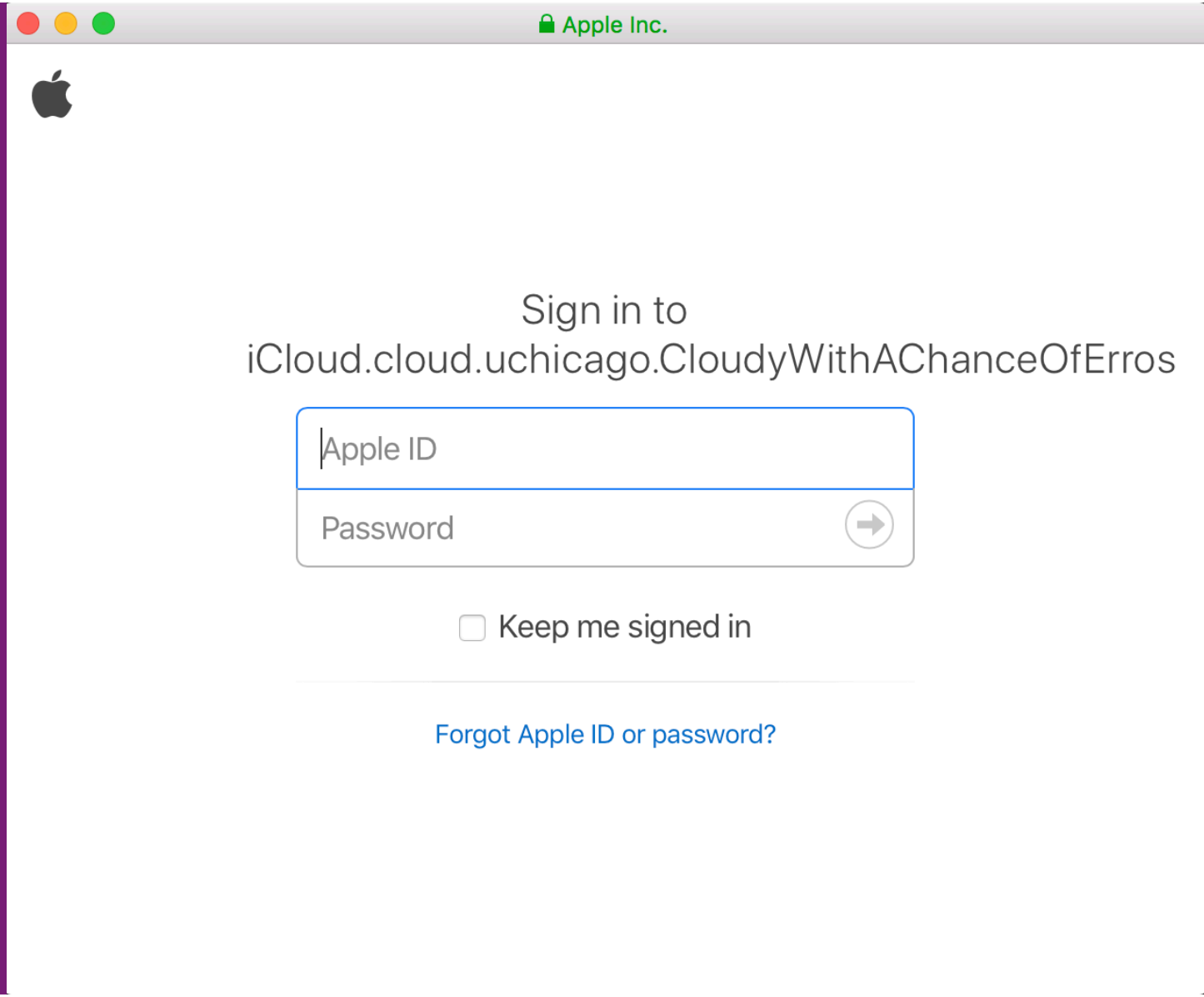
CLOUDKIT JS

- Token is used to communicate with iCloud



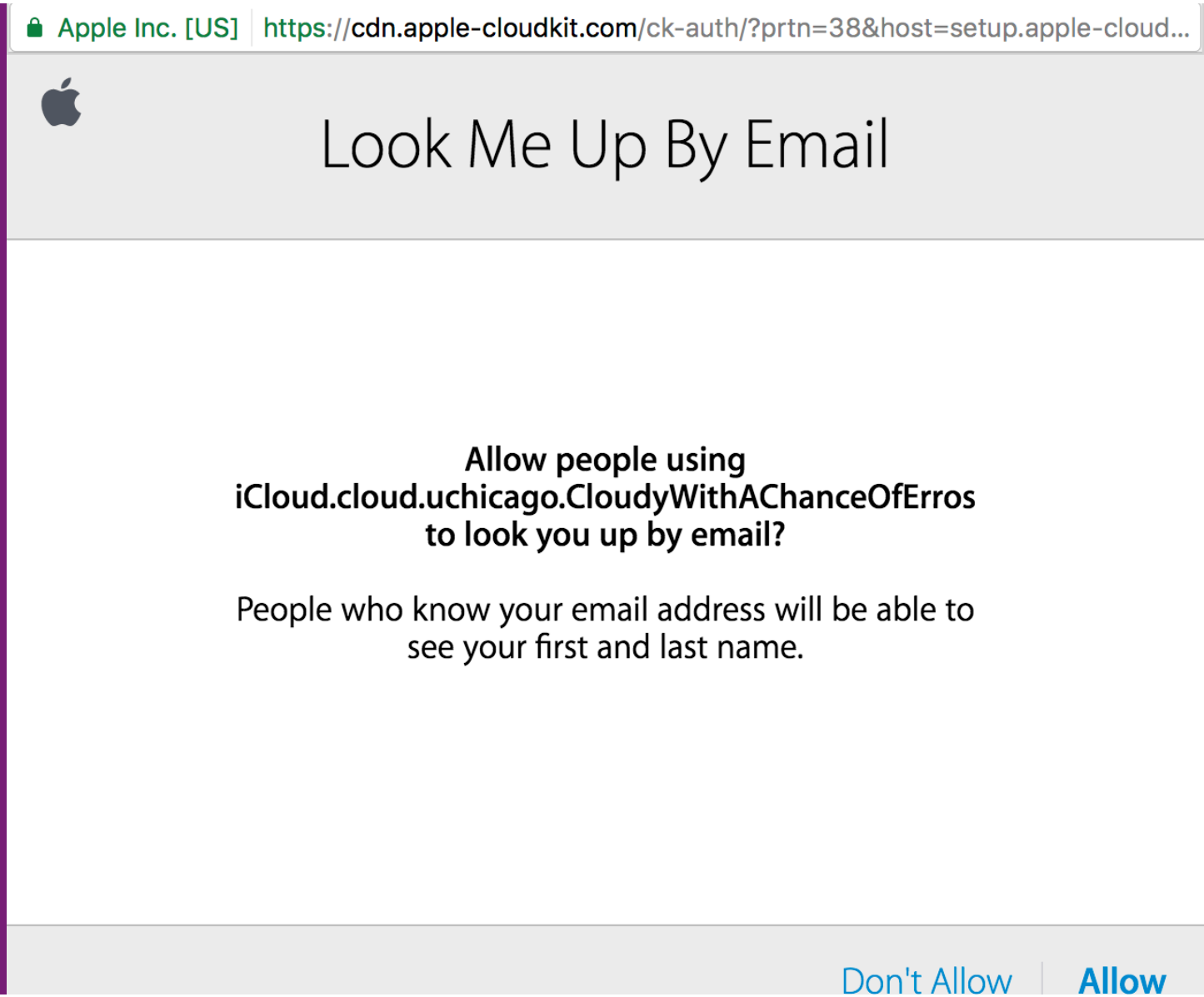
CLOUDKIT JS

- Authentication happens with Apple provided UI



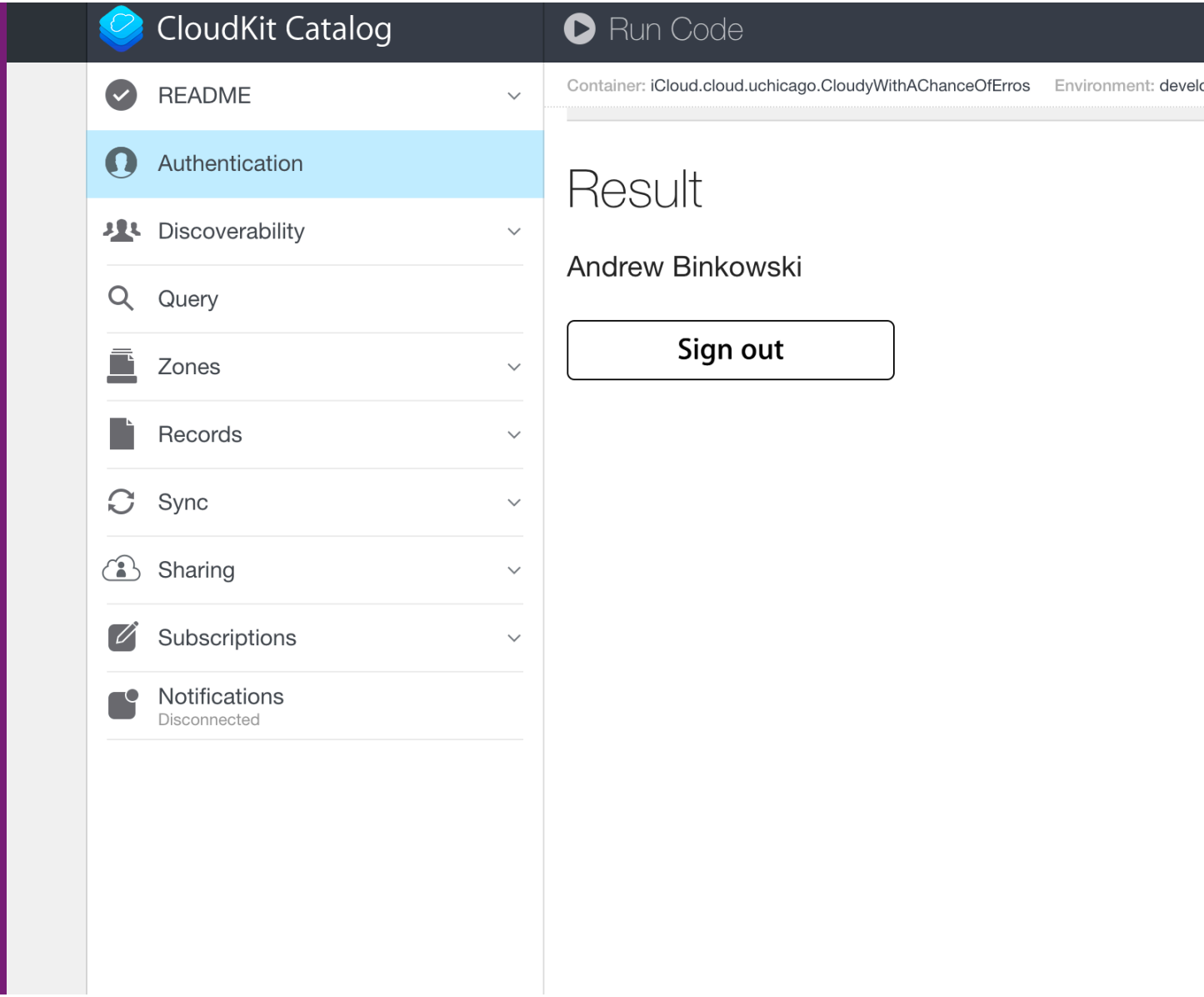
CLOUDKIT JS

- Test directly in CloudKitCatalog web app



CLOUDKIT JS

- Test directly in CloudKitCatalog web app



SERVER-TO-SERVER WITH CLOUDIKIT

SERVER-TO-SERVER

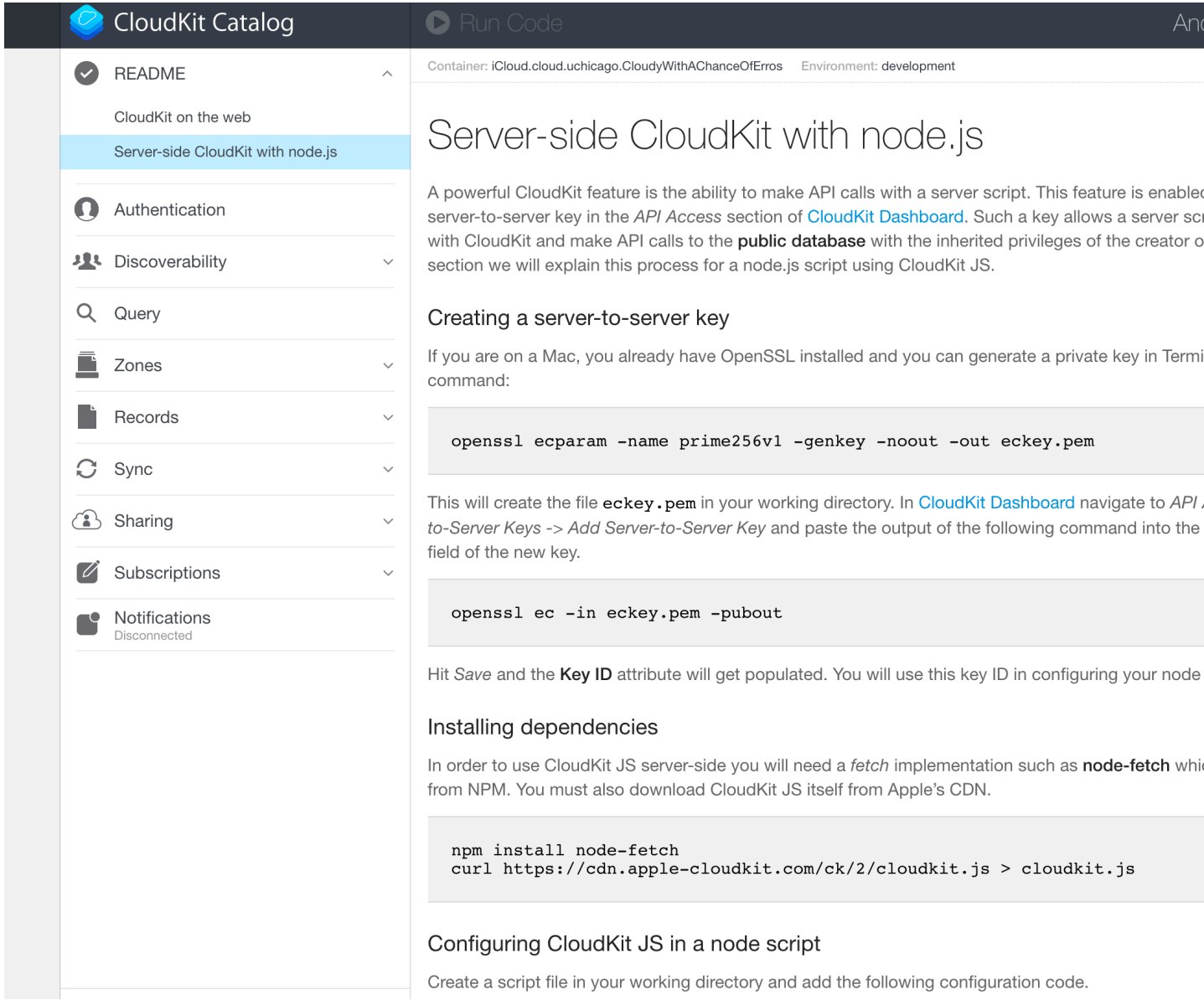
- Late 2016 Apple opened server-to-server communication
- Allows a server to communicate with CloudKit using a server-to-server key
- The missing link in data processing and server side logic 🤔

SERVER-TO-SERVER

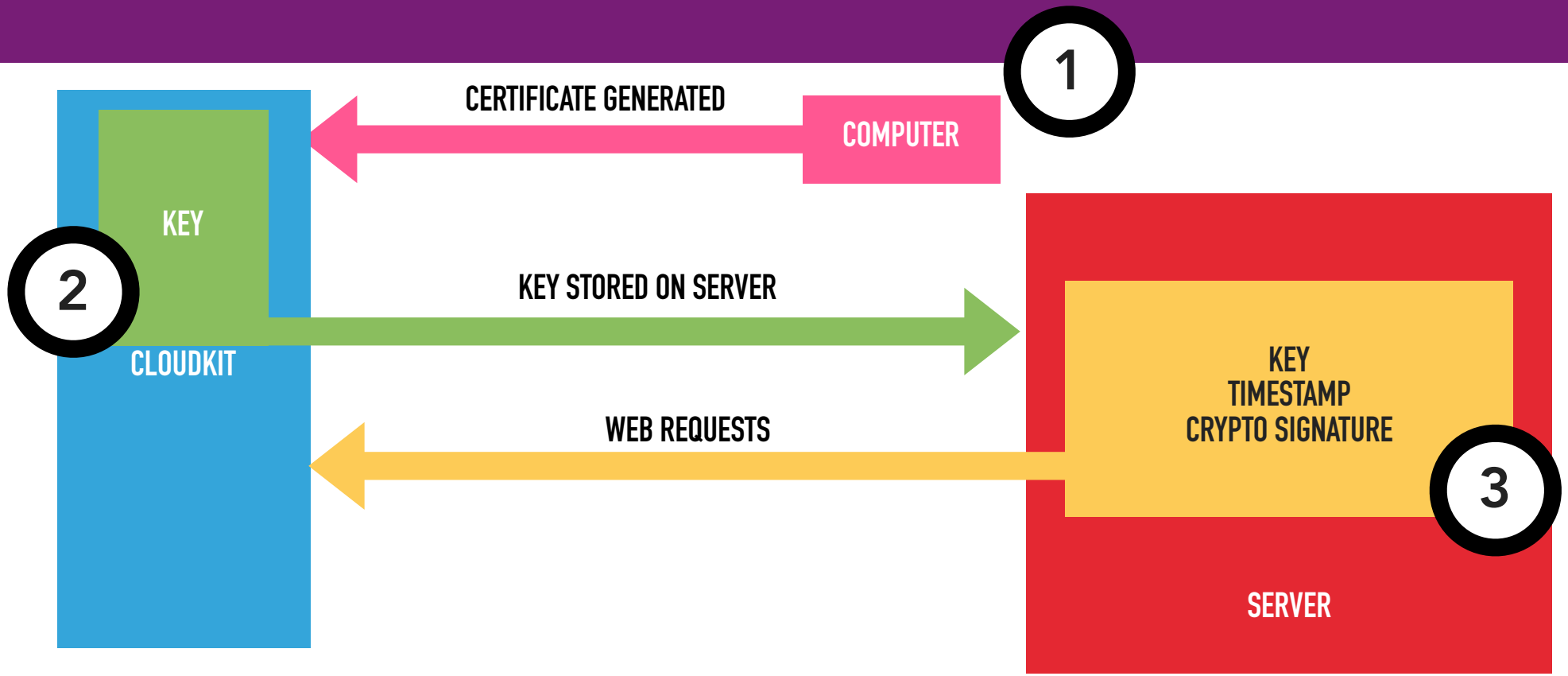
- API calls can only be made to the public database
- Calls run as with the inherited privileges of the creator of the key

SERVER-TO-SERVER

- Apple provides demo for node.js (supported by App Engine Flex)
- 👎



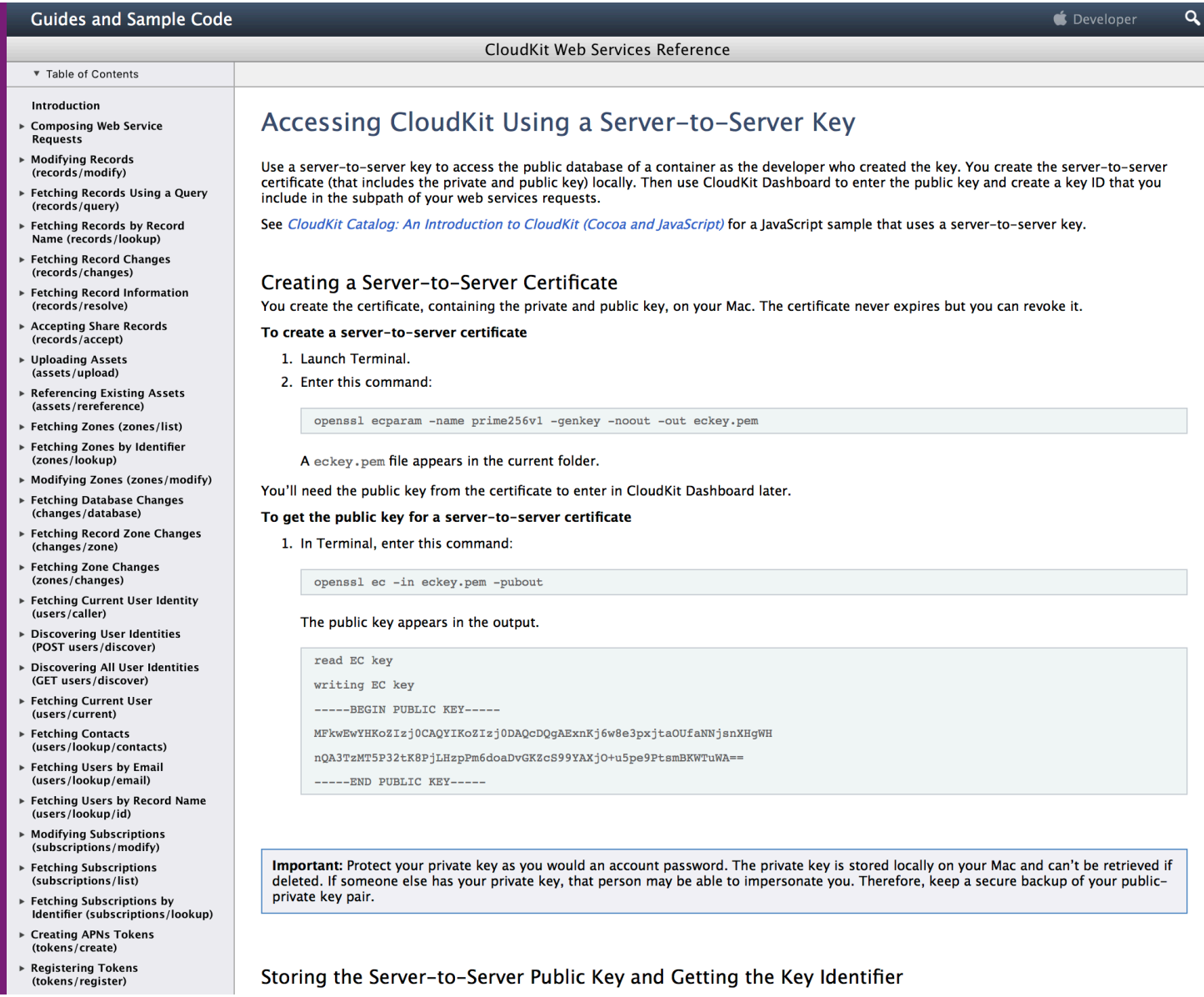
SERVER-TO-SERVER



SERVER-TO-SERVER KEY

SERVER-TO-SERVER KEY

- Create a public/private key pair to communicate with the iCloud servers



Guides and Sample Code

Developer

CloudKit Web Services Reference

Table of Contents

Accessing CloudKit Using a Server-to-Server Key

Use a server-to-server key to access the public database of a container as the developer who created the key. You create the server-to-server certificate (that includes the private and public key) locally. Then use CloudKit Dashboard to enter the public key and create a key ID that you include in the subpath of your web services requests.

See *CloudKit Catalog: An Introduction to CloudKit (Cocoa and JavaScript)* for a JavaScript sample that uses a server-to-server key.

Creating a Server-to-Server Certificate

You create the certificate, containing the private and public key, on your Mac. The certificate never expires but you can revoke it.

To create a server-to-server certificate

1. Launch Terminal.
2. Enter this command:

```
openssl ecparam -name prime256v1 -genkey -noout -out eckey.pem
```

A `eckey.pem` file appears in the current folder.

You'll need the public key from the certificate to enter in CloudKit Dashboard later.

To get the public key for a server-to-server certificate

1. In Terminal, enter this command:

```
openssl ec -in eckey.pem -pubout
```

The public key appears in the output.

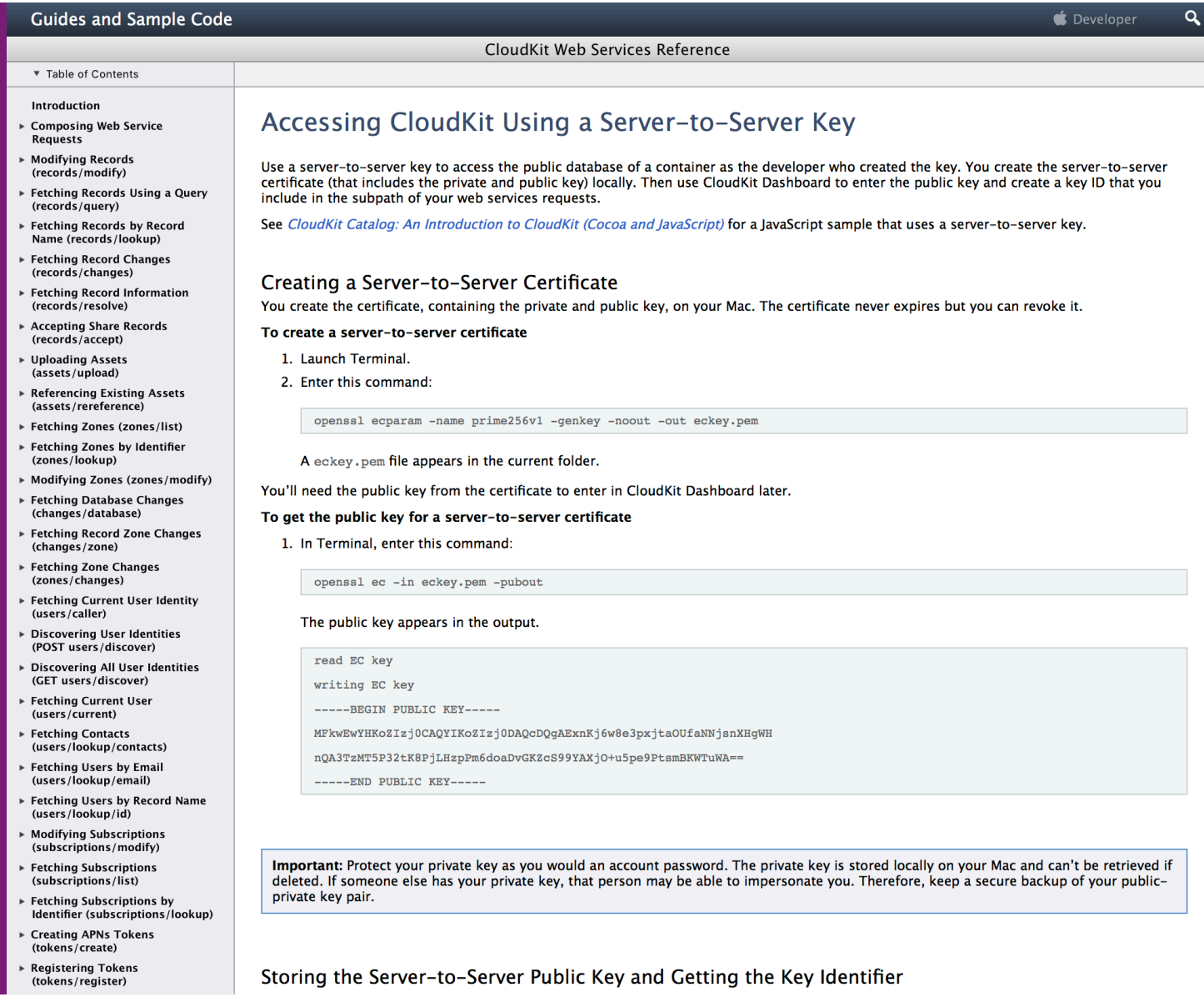
```
read EC key
writing EC key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAExnKj6w8e3pxjtaOUfaNNjsnXHgWH
nQA3TzMT5P32tK8PjLHzpPm6doaDvGKZcS99YAXjO+u5pe9PtsmBKWTuWA==
-----END PUBLIC KEY-----
```

Important: Protect your private key as you would an account password. The private key is stored locally on your Mac and can't be retrieved if deleted. If someone else has your private key, that person may be able to impersonate you. Therefore, keep a secure backup of your public-private key pair.

Storing the Server-to-Server Public Key and Getting the Key Identifier

SERVER-TO-SERVER KEY

- https://developer.apple.com/library/content/documentation/DataManagement/Conceptual/CloudKitWebServicesReference/SettingUpWebServices/SettingUpWebServices.html#//apple_ref/doc/uid/TP40015240-CH24-SW6



Guides and Sample Code

Developer

CloudKit Web Services Reference

Table of Contents

Accessing CloudKit Using a Server-to-Server Key

Use a server-to-server key to access the public database of a container as the developer who created the key. You create the server-to-server certificate (that includes the private and public key) locally. Then use CloudKit Dashboard to enter the public key and create a key ID that you include in the subpath of your web services requests.

See *CloudKit Catalog: An Introduction to CloudKit (Cocoa and JavaScript)* for a JavaScript sample that uses a server-to-server key.

Creating a Server-to-Server Certificate

You create the certificate, containing the private and public key, on your Mac. The certificate never expires but you can revoke it.

To create a server-to-server certificate

1. Launch Terminal.
2. Enter this command:

```
openssl ecparam -name prime256v1 -genkey -noout -out eckey.pem
```

A `eckey.pem` file appears in the current folder.

You'll need the public key from the certificate to enter in CloudKit Dashboard later.

To get the public key for a server-to-server certificate

1. In Terminal, enter this command:

```
openssl ec -in eckey.pem -pubout
```

The public key appears in the output.

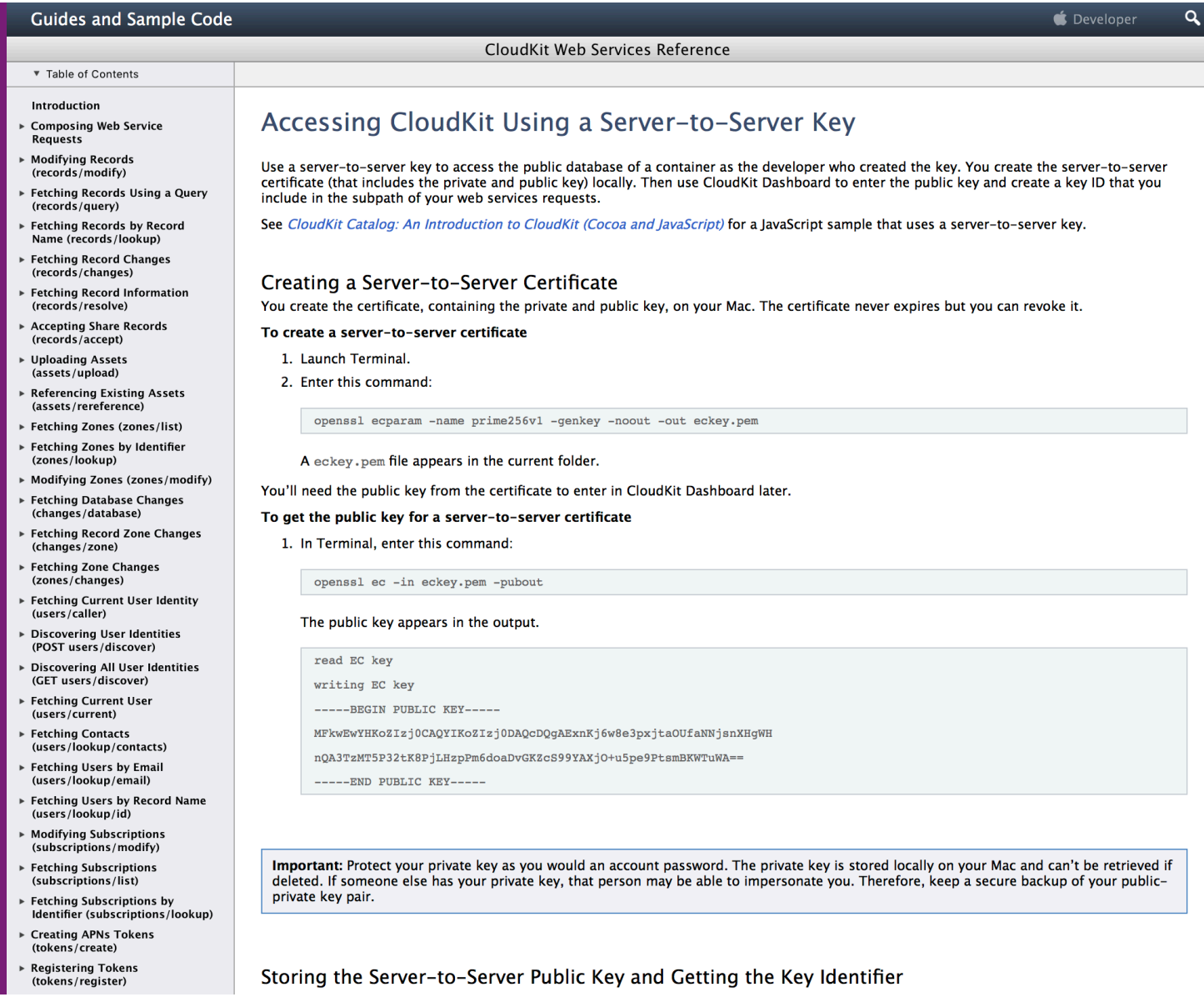
```
read EC key
writing EC key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAExnKj6w8e3pxjtaOUfaNNjsnXHgWH
nQA3TzMT5P32tK8PjLHzpPm6doaDvGKZcS99YAXjO+u5pe9PtsmBKWTuWA==
-----END PUBLIC KEY-----
```

Important: Protect your private key as you would an account password. The private key is stored locally on your Mac and can't be retrieved if deleted. If someone else has your private key, that person may be able to impersonate you. Therefore, keep a secure backup of your public-private key pair.

Storing the Server-to-Server Public Key and Getting the Key Identifier

SERVER-TO-SERVER KEY

- Protect your key
  - Back it up
  - Don't post on GitHub as part of project
- Can be revoked if loose track of it



Guides and Sample Code

Developer

CloudKit Web Services Reference

Table of Contents

Accessing CloudKit Using a Server-to-Server Key

Use a server-to-server key to access the public database of a container as the developer who created the key. You create the server-to-server certificate (that includes the private and public key) locally. Then use CloudKit Dashboard to enter the public key and create a key ID that you include in the subpath of your web services requests.

See *CloudKit Catalog: An Introduction to CloudKit (Cocoa and JavaScript)* for a JavaScript sample that uses a server-to-server key.

Creating a Server-to-Server Certificate

You create the certificate, containing the private and public key, on your Mac. The certificate never expires but you can revoke it.

To create a server-to-server certificate

1. Launch Terminal.
2. Enter this command:

```
openssl ecparam -name prime256v1 -genkey -noout -out eckey.pem
```

A `eckey.pem` file appears in the current folder.

You'll need the public key from the certificate to enter in CloudKit Dashboard later.

To get the public key for a server-to-server certificate

1. In Terminal, enter this command:

```
openssl ec -in eckey.pem -pubout
```

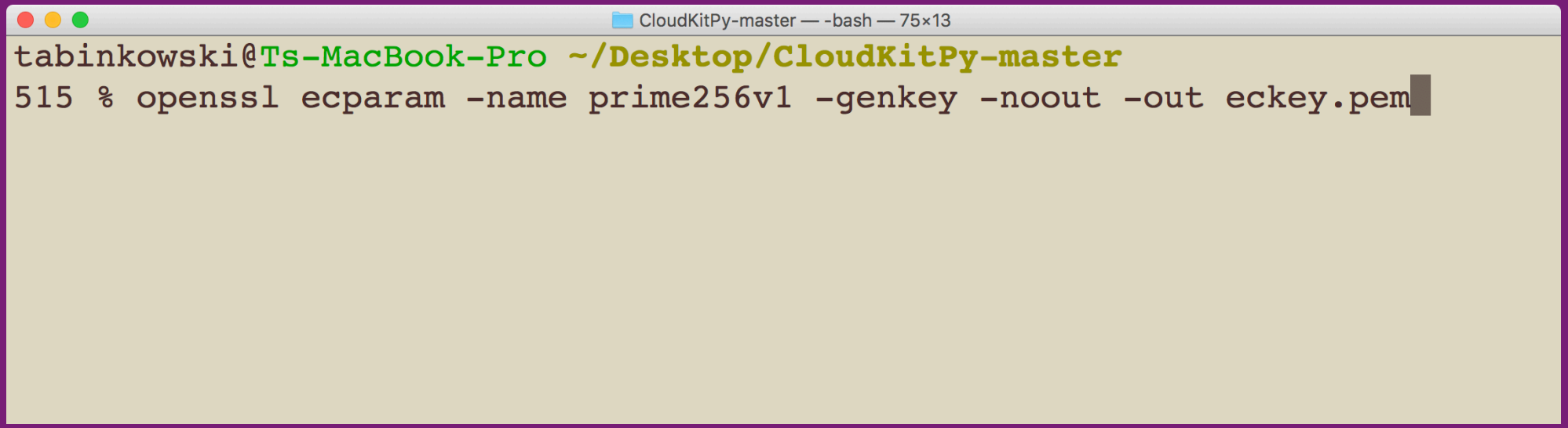
The public key appears in the output.

```
read EC key
writing EC key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAExnKj6w8e3pxjtaOUfaNNjsnXHgWH
nQA3TzMT5P32tK8PjLHzpPm6doaDvGKZcS99YAXjO+u5pe9PtsmBKWTuWA==
-----END PUBLIC KEY-----
```

Important: Protect your private key as you would an account password. The private key is stored locally on your Mac and can't be retrieved if deleted. If someone else has your private key, that person may be able to impersonate you. Therefore, keep a secure backup of your public-private key pair.

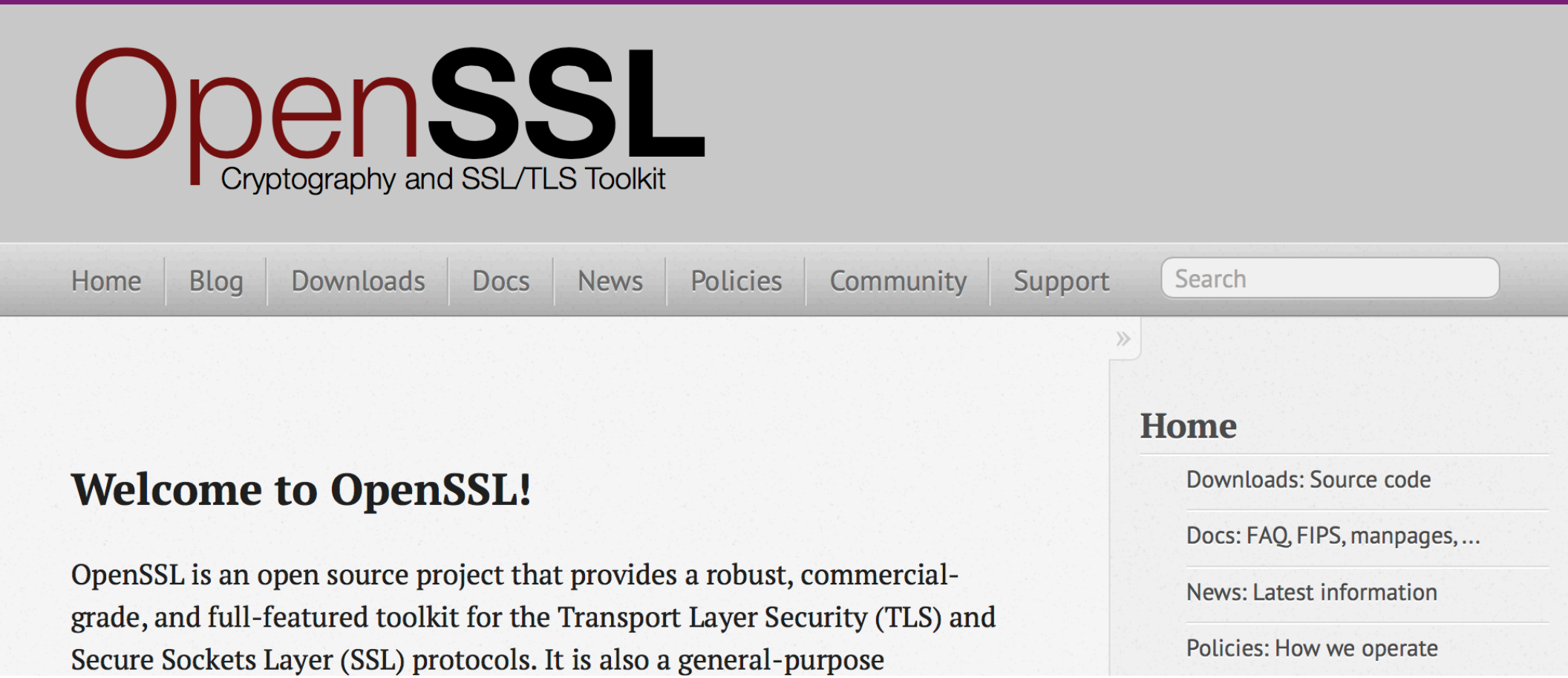
Storing the Server-to-Server Public Key and Getting the Key Identifier

SERVER-TO-SERVER KEY

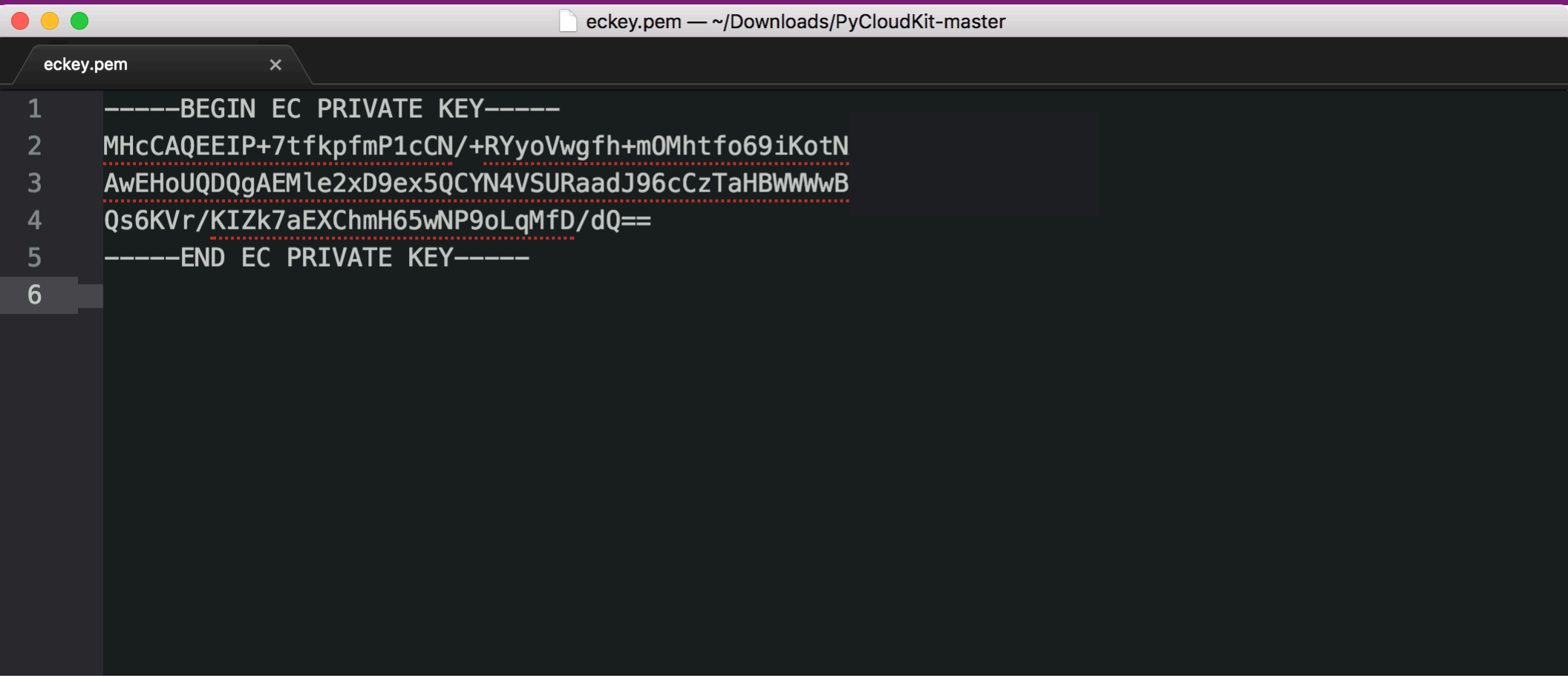


- Create the server-to-server certificate
- A eckey.pem file will be generated

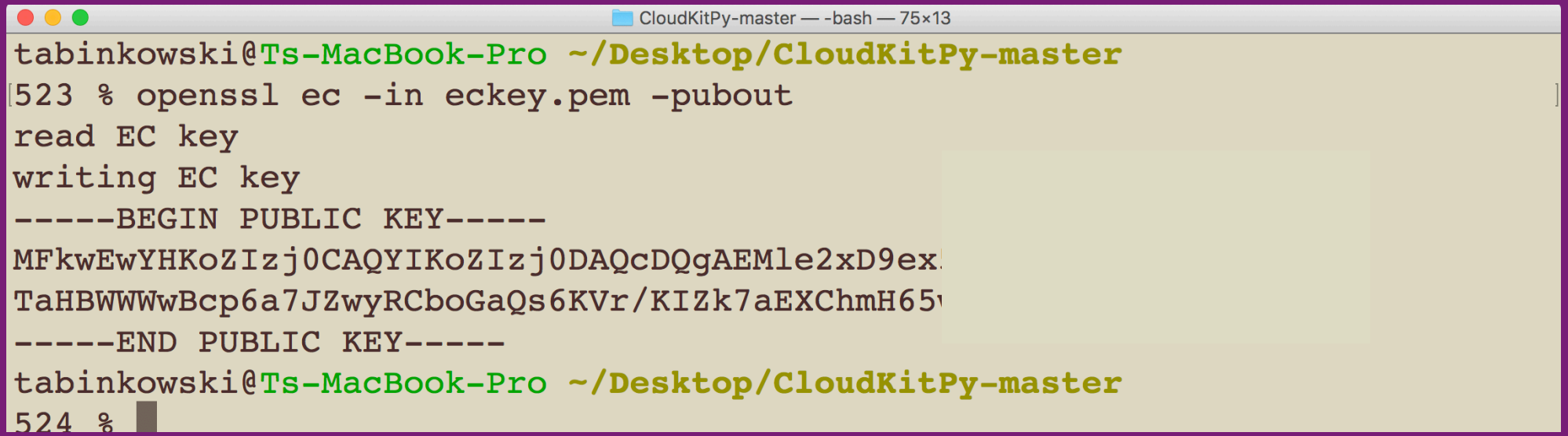
SERVER-TO-SERVER KEY



SERVER-TO-SERVER KEY

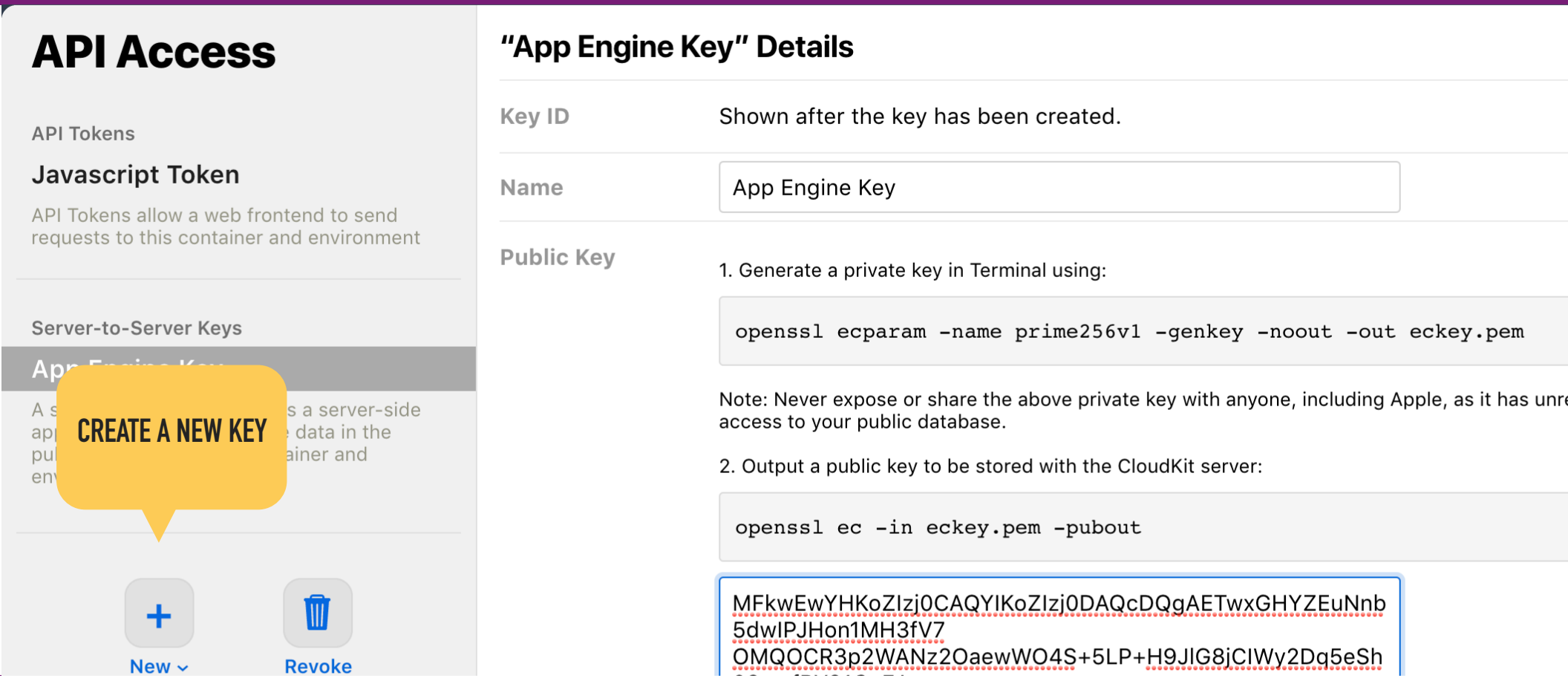


SERVER-TO-SERVER KEY



- Generate the public key
- Enter in CloudKit console

SERVER-TO-SERVER KEY



SERVER-TO-SERVER KEY

API Access

API Tokens

Javascript Token

API Tokens allow a web frontend to send requests to this container and environment

Server-to-Server Keys

App Engine Key

A server-to-server key allows a server-side application to read and write data in the public database of this container and environment.

"App Engine Key" Details

Key ID	99e04a281f141f4bce822833bd8e87827b914
Name	App Engine Key
Public Key	MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE OMQOCR3p2WANz2OaewWO4S+5LP+H9JIG
Notes	

WEB SERVICE REQUESTS

WEB SERVICE REQUESTS

- When using a server-to-server key, you sign the web service request
- Key is authentication
- Developer allowed access

To create a signed request using the server-to-server certificate in your keychain

1. Concatenate the following parameters and separate them with colons.

```
[Current date]:[Request body]:[Web service URL subpath]
```

Current date
The ISO8601 representation of the current date (without milliseconds)—for example, `2016-01-25T22:15:`

Request body
The base64 string encoded SHA-256 hash of the body.

Web service URL subpath
The URL described in The Web Service URL but without the `[path]` component, as in:

```
/database/1/iCloud.com.example.gkumar.MyApp/development/public/records/query
```

Note: If you include the API token subpath, described in Accessing CloudKit Using an API Token, the requ

2. Compute the ECDSA signature of this message with your private key.
3. Add the following request headers.

```
X-Apple-CloudKit-Request-KeyID: [keyID]
X-Apple-CloudKit-Request-ISO8601Date: [date]
X-Apple-CloudKit-Request-SignatureV1: [signature]
```

keyID
The identifier for the server-to-server key obtained from CloudKit Dashboard, described in Storing the Ser
Getting the Key Identifier.

date
The ISO8601 representation of the current date (without milliseconds).

signature
The signature created in Step 2.

WEB SERVICE REQUESTS

- All requests must have proper headers
- Only valid for 10 minutes

```
X-Apple-CloudKit-Request-KeyID: [keyID]

X-Apple-CloudKit-Request-ISO8601Date: [date]

X-Apple-CloudKit-Request-SignatureV1: [signature]
```

keyID
The identifier for the server-to-server key obtained from CloudKit Da
Getting the Key Identifier.

date
The ISO8601 representation of the current date (without milliseconds)

signature
The signature created in Step 2.

WEB SERVICE REQUESTS

QUERY

```
URL:
https://api.apple-cloudkit.com/database/1/
iCloud.cloud.uchicago.CloudyWithAChanceOfErros/development/
public/records/query

Data:  {"query": {"recordType": "joke"}}

Headers:  {
'X-Apple-CloudKit-Request-ISO8601Date': '2017-05-10T04:33:33Z',
'X-Apple-CloudKit-Request-SignatureV1':u'MEUCIQ...TfA=',
'X-Apple-CloudKit-Request-KeyID':'ffdb30c23add...00191'}
```

WEB SERVICE REQUESTS

- **records/resolve:** Fetches information about records using GUIDs, described in Fetching Record Information (records/resolve).
- **records/accept:** Accepts a share on behalf of the current user, described in Accepting Share Records (records/accept).
- **changes/database:** Fetches changed zones in a database, described in Fetching Database Changes (changes/database).
- **changes/zone:** Fetches changed records in the specified zones, described in Fetching Record Zone Changes (changes/zone).
- **users/caller:** Fetches information about the current user, described in Fetching Current User (users/current).
- **GET users/discover:** Fetches all user identities in the current user's address book, described in Discovering All User Identities (GET users/discover).
- **POST users/discover:** Fetches all users in the specified array, described in Discovering User Identities (POST users/discover).

Updated endpoints to support sharing records:

- **records/changes:** Added `shared` as database value, described in Fetching Record Changes (records/changes).
- **records/modify:** Added `shared` as database value, described in Modifying Records (records/modify).
- **records/lookup:** Added `shared` as database value, described in Fetching Records by Record Name (records/lookup).

- API Endpoints for CloudKit
- Look at CloudKit catalog for API examples

WEB SERVICE REQUEST QUERY

WEB SERVICE REQUESTS

- API for fetching records

Fetching Records Using a Query (records/query)

You can fetch records using a query.

Path

```
POST [path]/database/[version]/[container]/[environment]/[database]/records/query
```

Parameters

path
The URL to the CloudKit web service, which is `https://api.apple-cloudkit.com`.

version
The protocol version—currently, 1.

container
A unique identifier for the app's container. The container ID should begin with `iCloud.`.

environment
The version of the app's container. Pass `development` to use the environment that is not accessible by apps available on the store. environment that is accessible by development apps and apps available on the store.

database
The database to store the data within the container. The possible values are:

`public`
The database that is accessible to all users of the app.

`private`
The database that contains private data that is visible only to the current user.

`shared`
The database that contains records shared with the current user.

WEB SERVICE REQUESTS

Request

The POST request is a JSON dictionary containing the following keys:

Key	Description
`zoneID`	A dictionary that identifies a record zone in the database, described in Zone ID Dictionary.
`resultsLimit`	The maximum number of records to fetch. The default is the maximum number of records in a response that is allowed, described in Data Size Limits.
`query`	The query to apply, described in Query Dictionary. This key is required.
`continuationMarker`	The location of the last batch of results. Use this key when the results of a previous fetch exceeds the maximum. See Response. The default value is `null`.
`desiredKeys`	An array of strings containing record field names that limits the amount of data returned in this operation. Only the fields specified in the array are returned. The default is `null`, which fetches all record fields.
`zoneWide`	A Boolean value determining whether all zones should be searched. This key is ignored if `zoneID` is non-`null`. To search all zones, set to `true`. To search the default zone only, set to `false`.
`numbersAsStrings`	A Boolean value indicating whether number fields should be represented by strings. The default value is `false`.

- POST request consists of a JSON dictionary

WEB SERVICE REQUESTS

- Example query

```
"{
  "zoneID": {
    "zoneName": "myCustomZone",
  },
  "query": {
    "recordType": "myRecordType",
    "filterBy": [
      {
        "systemFieldName": "createdUserRecordName",
        "comparator": "EQUALS",
        "fieldValue": {
          "value": {
            "recordName": "recordA",
          },
          "type": "REFERENCE"
        }
      }
    ],
    "sortBy": [
      {
        "systemFieldName": "createdTimestamp",
        "ascending": false
      }
    ]
  }
}"
```

WEB SERVICE REQUESTS

- Example query operations

Record Operation Dictionary

The operation dictionary keys are:

Key	Description
`operationType`	The type of operation. Possible values are described in Operation Type Values. This key is required.
`record`	A dictionary representing the record to modify, as described in Record Dictionary. This key is required.
`desiredKeys`	An array of strings containing record field names that limits the amount of data returned in this operation. Only the fields specified in the array are returned. The default is `null`, which fetches all record fields. This `desiredKeys` setting overrides the `desiredKeys` setting in the enclosing dictionary.

Operation Type Values

The possible values for the `operationType` key are:

Value	Description
`create`	Create a new record. This operation fails if a record with the same record name already exists.
`update`	Update an existing record. Only the fields you specify are changed.
`forceUpdate`	Update an existing record regardless of conflicts. Creates a record if it doesn't exist.
`replace`	Replace a record with the specified record. The fields whose values you do not specify are set to `null`.
`forceReplace`	Replace a record with the specified record regardless of conflicts. Creates a record if it doesn't exist.
`delete`	Delete the specified record.
`forceDelete`	Delete the specified record regardless of conflicts.

WEB SERVICE REQUESTS

Response

An array of dictionaries describing the results of the operation. The dictionary contains the following keys:

Key	Description
`records`	An array containing a result dictionary for each record requested. If successful, the result dictionary contains the keys described in Record Dictionary. If unsuccessful, the result dictionary contains the keys described in Record Fetch Error Dictionary.
`continuationMarker`	If included, indicates that there are more results matching this query. To fetch the other results, pass the value of the `continuationMarker` key as the value of the `continuationMarker` key in another query.

- Response contains dictionary of records
- Returned results are limited by data (unless specified)

WEB SERVICE REQUESTS

- Returned results are limited by data (unless specified by the `resultsLimit` key in request)
- `continuationMarker` for the next batch

Data Size Limits

These are the limits on the size of data sent to and from the CloudKit server.

Property	Value
Maximum number of operations in a request	200
Maximum number of records in a response	200
Maximum number of tokens in a request	200
Maximum record size (not including Asset fields)	1 MB
Maximum file size of an Asset field	50 MB
Maximum number of source references to a single target where the action is delete self	750

WEB SERVICE REQUESTS

```
URL:
https://api.apple-cloudkit.com/database/1/
iCloud.cloud.uchicago.CloudyWithAChanceOfErros/development/
public/records/query

Data:  {"query": {"recordType": "joke"}}

Headers:  {
'X-Apple-CloudKit-Request-ISO8601Date': '2017-05-10T04:33:33Z',
'X-Apple-CloudKit-Request-SignatureV1':u'MEUCIQ...TfA=',
'X-Apple-CloudKit-Request-KeyID':'ffdb30c23add...00191'}
```

API URL

QUERY JSON

MODIFY RECORDS

WEB SERVICE REQUESTS

- Apply multiple types of operations
  - Creating
  - Updating
  - Replacing
  - Deleting
- Multiple records can be modified in a single request

Path

```
POST [path]/database/[version]/[container]/[environment]/[database]/records/modify
```

Parameters

path
The URL to the CloudKit web service, which is `https://api.apple-cloudkit.com`.

version
The protocol version—currently, 1.

container
A unique identifier for the app's container. The container ID begins with `iCloud.`.

environment
The version of the app's container. Pass `development` to use the environment that is not accessible by apps available on the store. environment that is accessible by development apps and apps available on the store.

database
The database to store the data within the container. The possible values are:

`public`
The database that is accessible to all users of the app.

`private`
The database that contains private data that is visible only to the current user.

`shared`
The database that contains records shared with the current user.

WEB SERVICE REQUESTS

Key	Description
`operations`	An array of dictionaries defining the operations to apply to records in the database. The dictionary keys are described in Record Operation Dictionary. See Data Size Limits for maximum number of operations allowed. This key is required.
`zoneID`	A dictionary that identifies a record zone in the database, described in Zone ID Dictionary.
`atomic`	A Boolean value indicating whether the entire operation fails when one or more operations fail. If `true`, the entire request fails if one operation fails. If `false`, some operations may succeed and others may fail. The default value is `true`. Note this property only applies to custom zones.
`desiredKeys`	An array of strings containing record field names that limit the amount of data returned in the enclosing operation dictionaries. Only the fields specified in the array are returned. The default is `null`, which fetches all record fields.
`numbersAsStrings`	A Boolean value indicating whether number fields should be represented by strings. The default value is `false`.

- POST request consists of a JSON dictionary

WEB SERVICE REQUESTS

```
{
    "operationType" : "create",
    "record" : {
        "recordType" : "Artist",
        "fields" : {
            "firstName" : {"value" : "Mei"},
            "lastName" : {"value" : "Chen"}
        }
        "recordName" : "Mei Chen"
    },
}
```

YOUR CUSTOM FIELDS

OMIT RECORDNAME AND IT WILL BE AUTOMATICALLY GENERATED AS UUID

- POST request consists of a JSON dictionary
- Schema is not on-demand for JS API

WEB SERVICE REQUESTS

ADD A RECORD

```
URL:  https://api.apple-cloudkit.com/database/1/
iCloud.cloud.uchicago.CloudyWithAChanceOfErros/development/
public/records/modify

Data:  {"operations": [{"record": {"fields": {"joke": {"value":
{"recordName": "60d82b7d-32ef-4e91-9c12-585c92a3bb89",
"action": "DELETE_SELF", "zoneID:": {"zoneName":
"_defaultZone"}}}}, "day": {"value": "today"}}, "recordType":
"Daily"}, "operationType": "create"}]}

Headers:  {
'X-Apple-CloudKit-Request-ISO8601Date': '2017-05-10T04:33:33Z',
'X-Apple-CloudKit-Request-SignatureV1':u'MEUCIQ...TfA=',
'X-Apple-CloudKit-Request-KeyID':'ffdb30c23add...00191'}
```

WEB SERVICE REQUESTS

ADD A RECORD

```
{"operations": [
 {"record":
  {"fields":
   {"joke":
    {"value":
     {"recordName": "60d82b7d-32ef-4e91-9c12-585c92a3bb89",
      "action": "DELETE_SELF",
      "zoneID:": {"zoneName": "_defaultZone"}
     }
   },
   "day": {"value": "today"}},
   "recordType": "Daily"},
   "operationType": "create"}]
}
```

WEB SERVICE REQUESTS

Response

The response is an array of dictionaries describing the results of the operation. The dictionary contains a single key:

Key	Description
`records`	An array containing a result dictionary for each operation in the request. If successful, the result dictionary contains the keys described in Record Dictionary. If unsuccessful, the result dictionary contains the keys described in Record Fetch Error Dictionary.

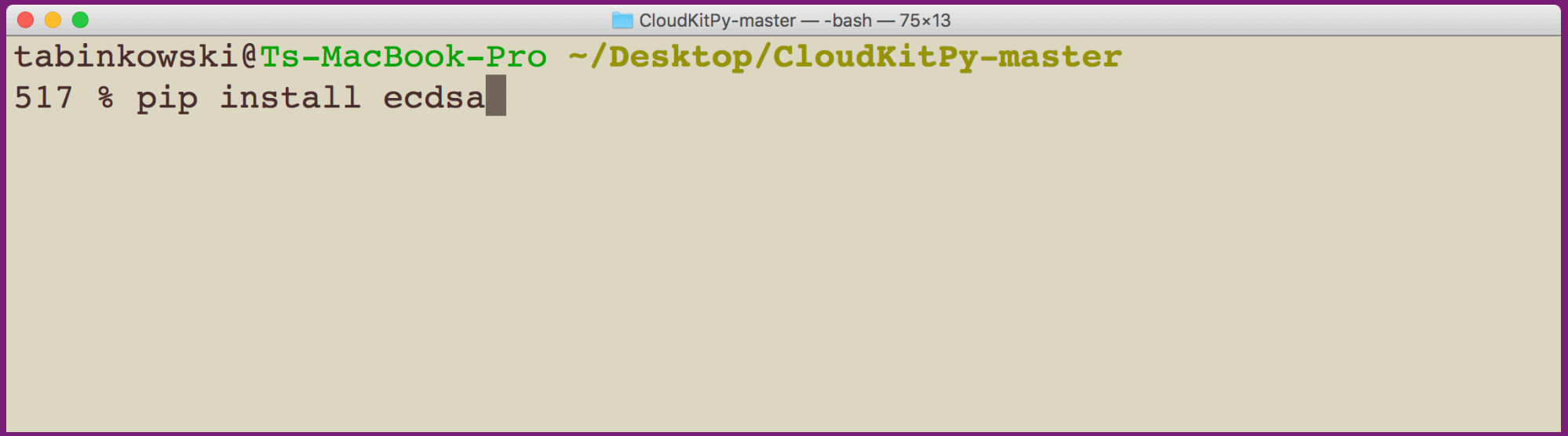
- Response is a dictionary with more information than you probably need

WEB SERVICE REQUESTS

- Response dictionary


```
{
    "created": {
        "deviceID": "2",
        "timestamp": 1494288550303,
        "userRecordName": "_7b892f2a3eeaddd6afc77ff7158f69d9"
    },
    "fields": {
        "question": {
            "type": "STRING",
            "value": "Why?\n"
        },
        "response": {
            "type": "STRING",
            "value": "That's why!"
        }
    },
    "modified": {
        "deviceID": "2",
        "timestamp": 1494386749697,
        "userRecordName": "_7b892f2a3eeaddd6afc77ff7158f69d9"
    },
    "pluginFields": {},
    "recordChangeTag": "j2gstkiq",
    "recordName": "60d82b7d-32ef-4e91-9c12-585c92a3bb89",
    "recordType": "joke",
    "zoneID": {
        "ownerRecordName": "_7b892f2a3eeaddd6afc77ff7158f69d9",
        "zoneName": "_defaultZone"
    }
```

SERVER-TO-SERVER KEY

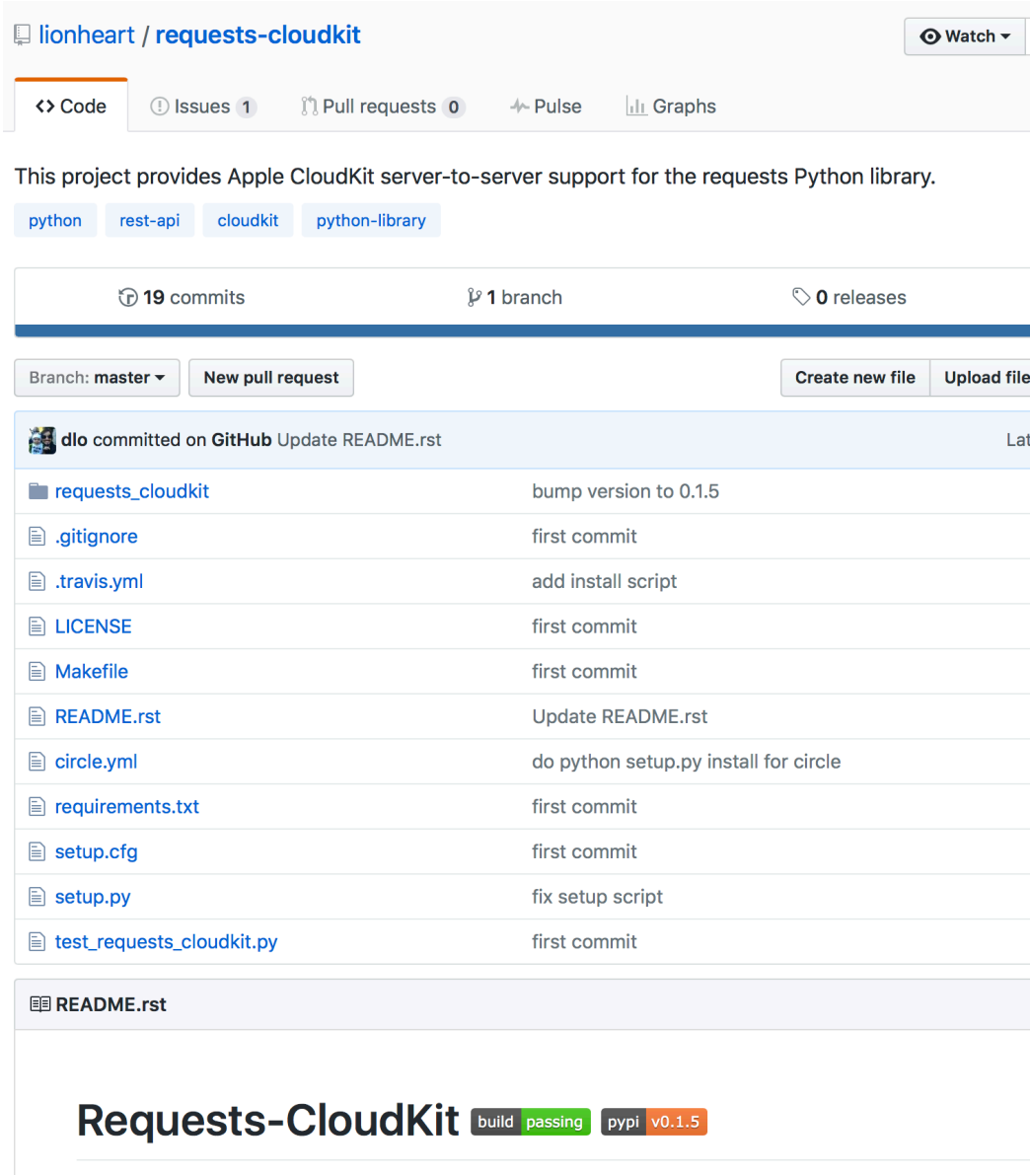


- Elliptic Curve Digital Signature Algorithm used to sign requests
- Cryptography approach for public/private key agreement

SERVER-TO-SERVER DEMO

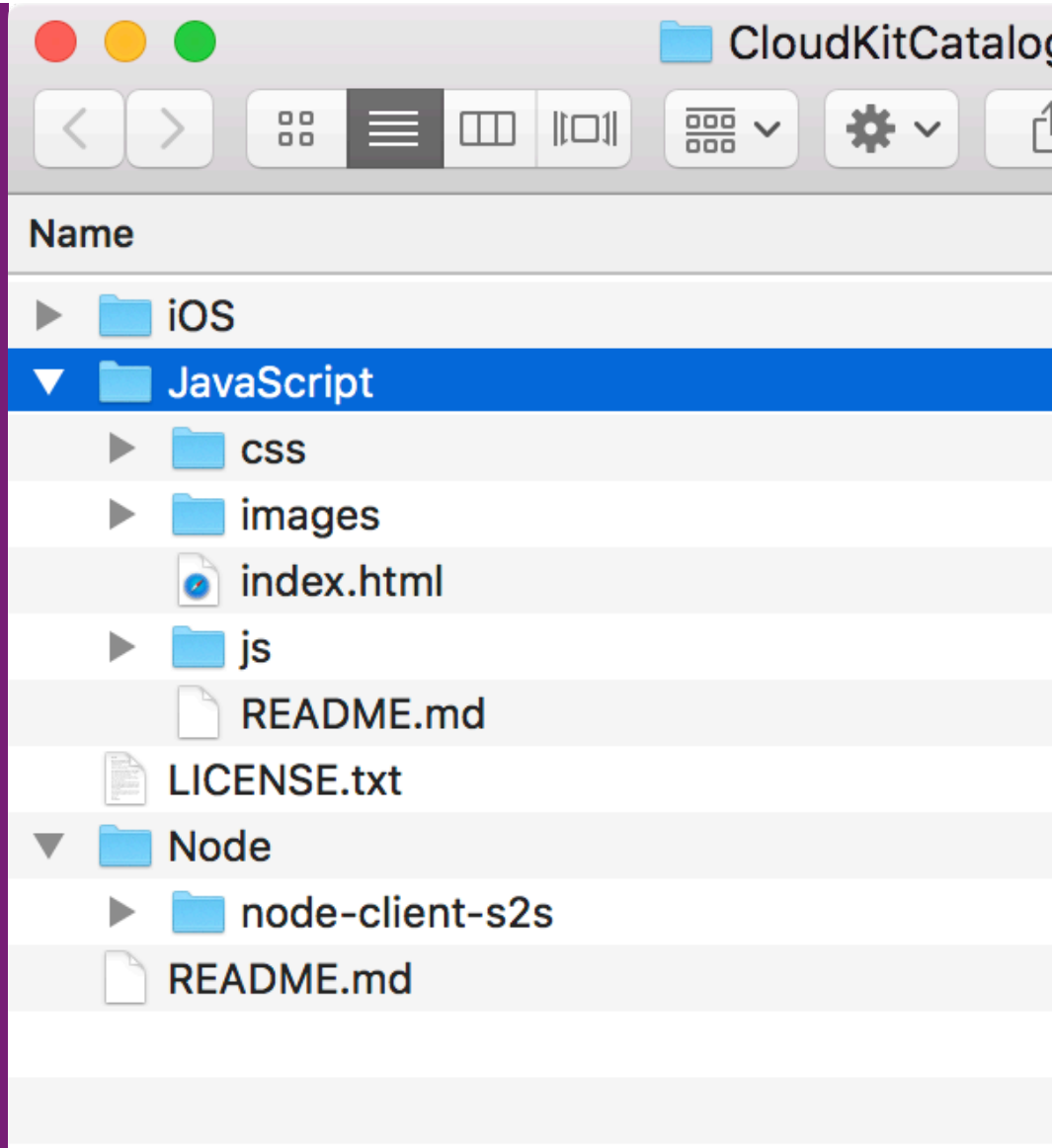
SERVER-TO-SERVER KEY

- CloudKit API is JSON based so any language can be used
- Surprisingly few number of existing libraries
  - Even less that actually work



SERVER-TO-SERVER KEY

- Apple has a sample project showing iOS, JS, and Node sharing a CloudKit container
- CloudKitCatalogAnIntroduction ToCloudKitCocoaJavaScript

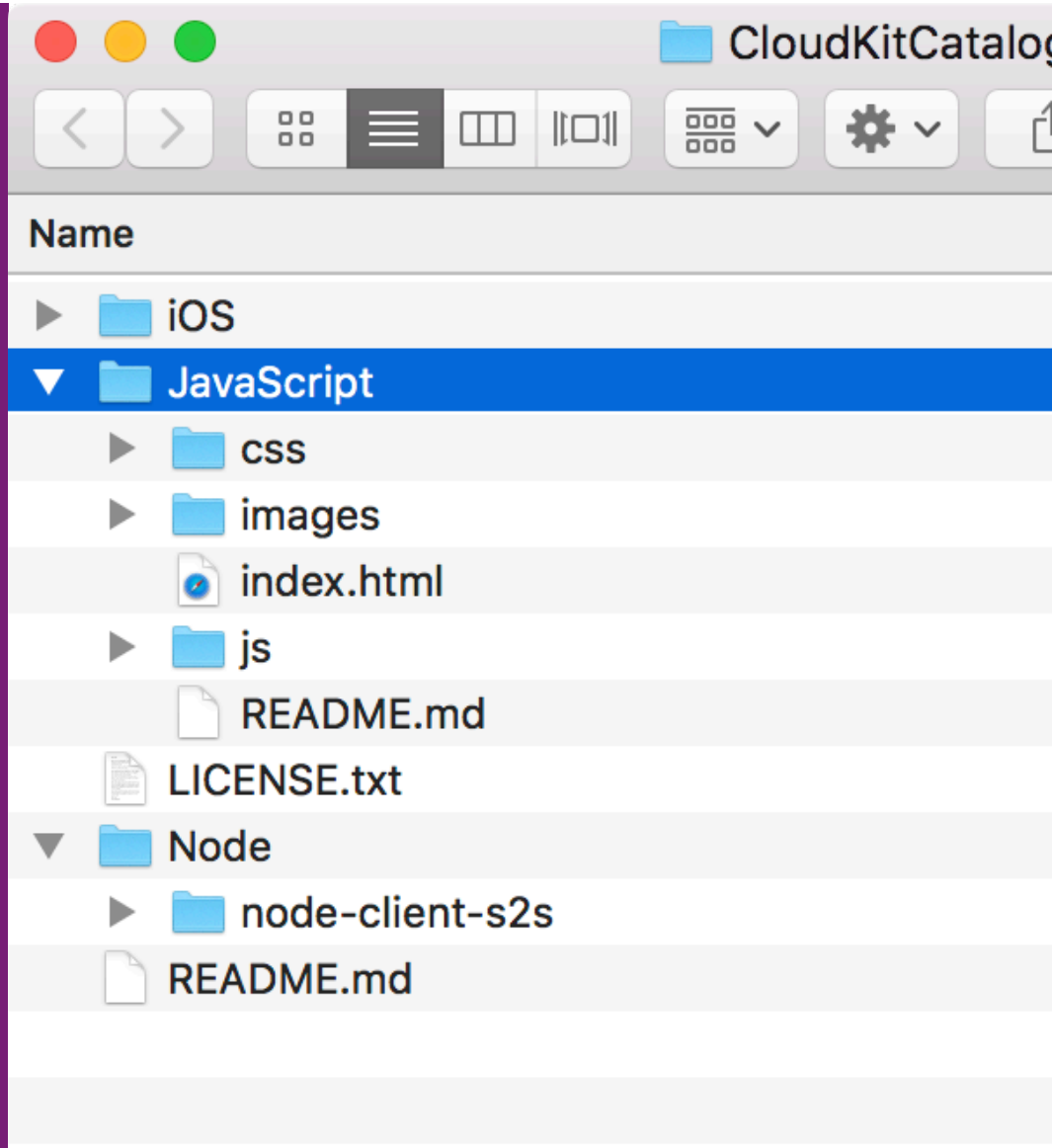


SERVER-TO-SERVER KEY

```
/*
 Copyright (C) 2016 Apple Inc. All Rights Reserved.
 See LICENSE.txt for this sample's licensing information

 Abstract:
 This node script uses a server-to-server key to make public database calls with CloudKit JS
 */


process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";

(function() {
  var fetch = require('node-fetch');

  var CloudKit = require('./cloudkit');
  var containerConfig = require('./config');

  // A utility function for printing results to the console.
  var println = function(key,value) {
    console.log("--> " + key + ":");
    console.log(value);
    console.log();
  };

  //CloudKit configuration
  CloudKit.configure({
```

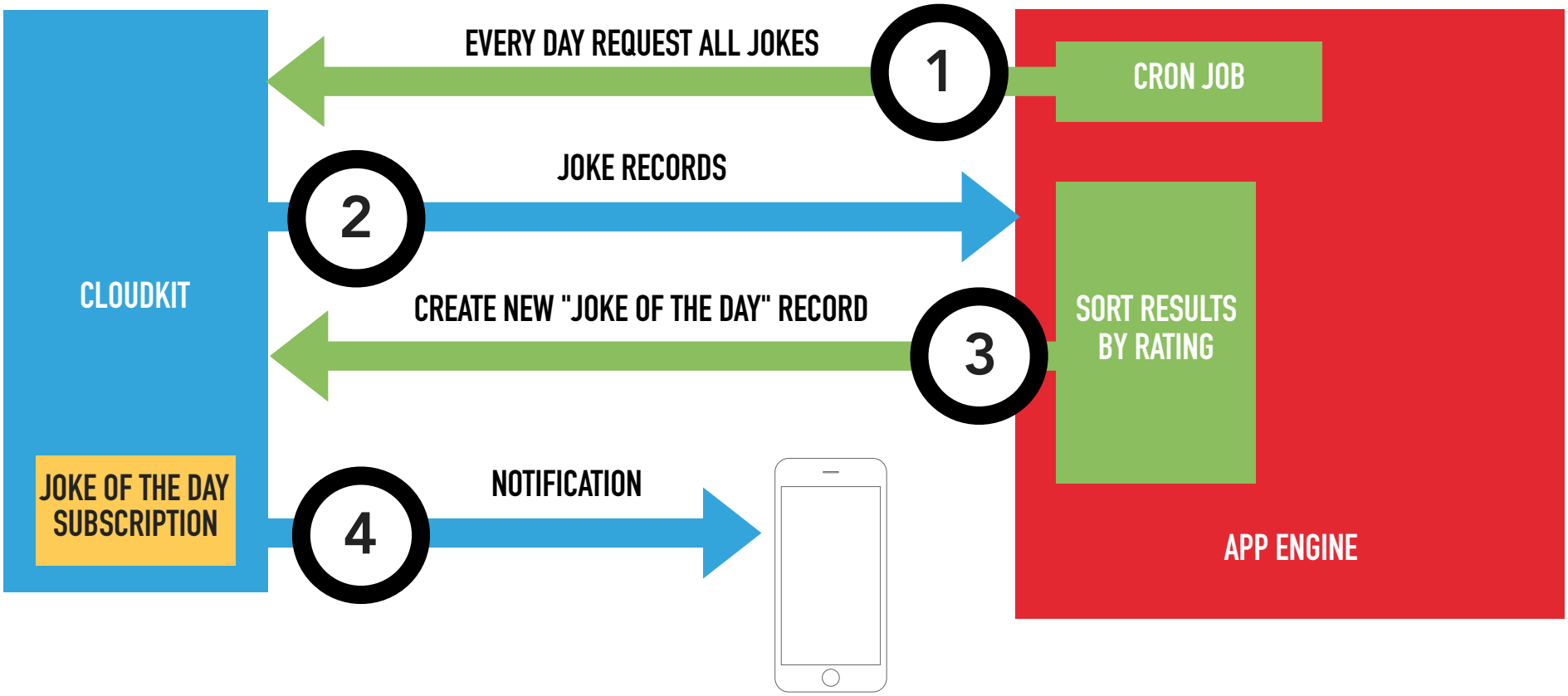
SERVER-TO-SERVER KEY

- Python so that we can use App Engine



CLOUDKIT HELPER

SERVER-TO-SERVER KEY



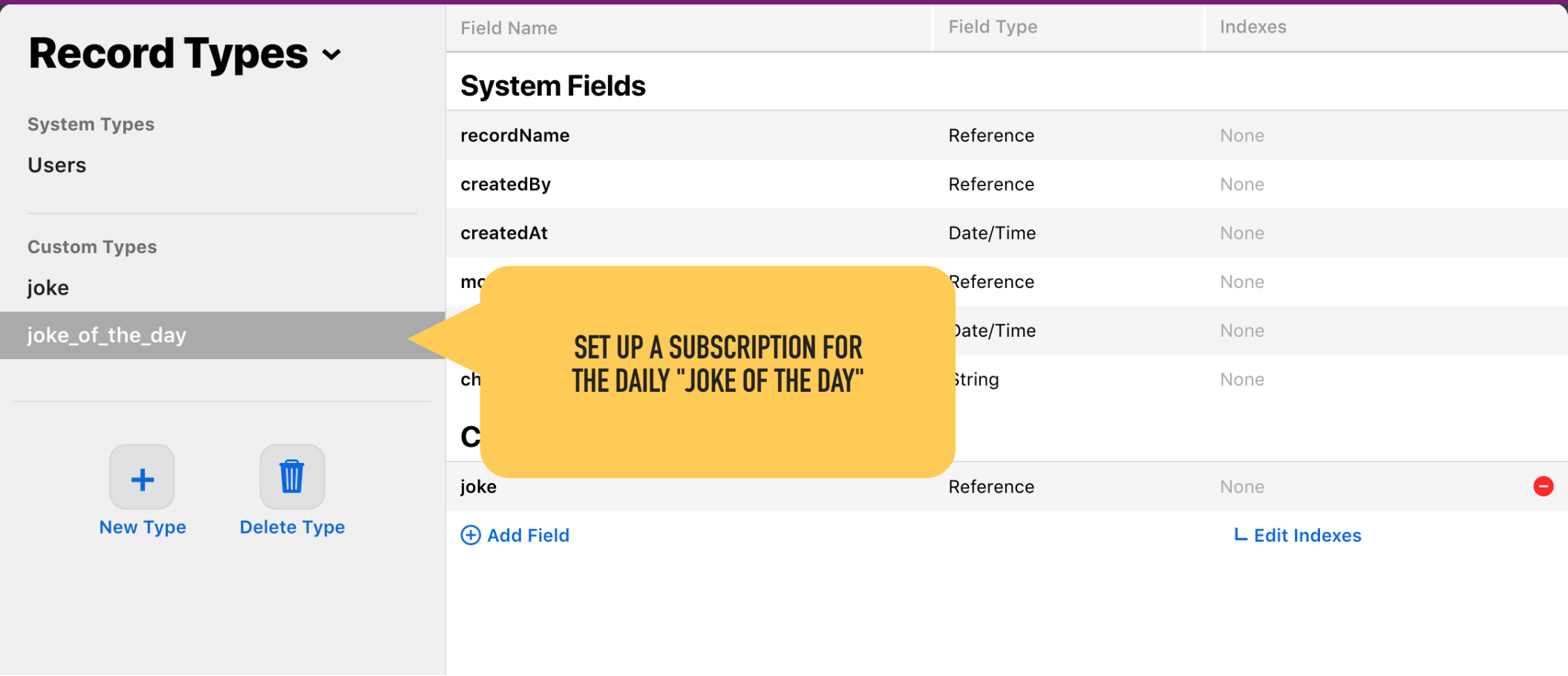
SERVER-TO-SERVER KEY

- Python so that we can use App Engine

```
#
#
# Based off the following
#  - CloudKitCatalog (c) 2016, Apple
#  - PyCloudKit. Created on 09.02.2016  (c) 2015 Andreas Schulz
#  - requests-cloudkit Copyright 2016 Lionheart Software LLC


from __future__ import print_function
import ecdsa
import base64
import hashlib
import datetime
import sys
import json

from urllib2 import HTTPPasswordMgrWithDefaultRealm, HTTPBasicAuthHandler, Request, build_opener
from urllib import urlencode


KEY_ID = 'ffdb30c23addd822e851e15cdd30ea819251d2fbb9bc59c4803f006dcc900191'
CONTAINER = 'iCloud.cloud.uchicago.CloudyWithAChanceOfErros'

def cloudkit_request(cloudkit_resource_url, data):
    """Uses HTTP GET or POST to interact with CloudKit. If data is empty, Uses
    GET, else, POSTs the data.
    """

    # Get ISO 8601 date, cut milliseconds.
    date = datetime.datetime.utcnow().isoformat()[:-7] + 'Z'

    # Load JSON request from config.
    _hash = hashlib.sha256(data.encode('utf-8')).digest()
    body = base64.b64encode(_hash).decode('utf-8')

    # Construct URL to CloudKit container.
    web_service_url = '/database/1/' + CONTAINER  + cloudkit_resource_url

    # Load API key from config.
    key_id = KEY_ID

    # Read out certificate file corresponding to API key.
    with open('eckey.pem', 'r') as pem_file:
        signing_key = ecdsa.SigningKey.from_pem(pem_file.read())

    # Construct payload.
```

IOS

IOS


Record Types
System Types
Users
Custom Types
joke
joke_of_the_day
New Type
Delete Type
Field Name
Field Type
Indexes
System Fields
recordName
Reference
None
createdBy
Reference
None
createdAt
Date/Time
None
Reference
None
Date/Time
None
String
None
joke
Reference
None
Add Field
Edit Indexes
SET UP A SUBSCRIPTION FOR
THE DAILY "JOKE OF THE DAY"

IOS

- Setup subscription for joke of the day
- Remember subscriptions are tied to individual users

```
func registerJokeOfTheDaySubscriptions() {
  // Unique identifier for the subscription
  let uuid: UUID = UUID()
  let identifier = "\(uuid)-joke-of-the-day"

  // Create the notification that will be delivered
  let notificationInfo = CKSubscription.NotificationInfo()
  notificationInfo.alertBody = "The Joke of the Day is here! 😂"
  notificationInfo.shouldBadge = true
  notificationInfo.shouldSendContentAvailable = true
  //notificationInfo.desiredKeys = ["question"]

  // Create the subscription object
  let subscription = CKQuerySubscription(recordType: "joke_of_the_day",
                                         predicate: NSPredicate(value: true),
                                         subscriptionID: identifier,
                                         options: [
                                          CKQuerySubscription.Options.firesOnRecordCre

  subscription.notificationInfo = notificationInfo

  // Save subscription
  currentDB.save(subscription, completionHandler: ({returnRecord, error in
    if let err = error {
      print("JOTD: subscription failed \(err.localizedDescription)")
    } else {
      print("JOTD: subscription set up")
    }
  }))
}
```

IOS

```
// Create the notification that will be delivered
let notificationInfo = CKSubscription.NotificationInfo()
notificationInfo.alertBody = "The Joke of the Day is here! 😂"
notificationInfo.shouldBadge = true
notificationInfo.shouldSendContentAvailable = true
notificationInfo.desiredKeys = ["joke"]

// Create the subscription object
le                                    ecordType: "joke_of_the_day",
                                      predicate: NSPredicate(value: true),
                                      subscriptionID: identifier,
                                      options: [
                                       CKQuerySubscription.Options.firesOnRecordCrea
```

KEYS TO SEND WITH RECORD TYPE

- Could send the joke (or make it generic)

IOS

```
}, AnyHashable("ck"): {
    ce = 2;
    cid = "iCloud.mobi.uchicago.twenty-nineteen-cloudkit";
    ckuserid = "_c364d6b1fb322f17cd2346d1b82667a2";
    nid = "257010ef-4903-4116-873a-bfcb3b619137";
    qry =     {
        af =         {
            joke = "3CCB6936-49BD-4CDF-8FAB-63F362C5D1B8";
        };
        zid = "_defaultZone";
        zoid = "_defaultOwner";
    };
}]
```

KEYS TO SEND WITH RECORD TYPE

```
// Create the notification that will
let notificationInfo = CKSubsc
notificationInfo.alertBody =
notificationInfo.shouldBadge = true
notificationInfo.shouldSendContentAv
notificationInfo.desiredKeys = ["jok
```

KEYS TO SEND WITH RECORD TYPE

```
(recordType: "joke_of_the_day",
 predicate: NSPredicate(value: true),
 subscriptionID: identifier,
 options: [
  CKQuerySubscription.Options.firesOnRecordCrea
```

- Could send the joke (or make it generic)

APP ENGINE

APP ENGINE

- App Engine microservice to run cron job to process the data
  - Request all jokes
  - Process to find top rated joke
  - Add a new record to CloudKit representing joke of the day (trigger subscription)

```
# app.yaml
#

runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /.*
  script: main.app
```

APP ENGINE

- We don't need to store anything on App Engine to accomplish this functionality

```
# app.yaml
#

runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /.*
  script: main.app
```

APP ENGINE

- Adapt our cloudkit_helper for app engine
  - Replace print()
  - …

```
from __future__ import print_function
import ecdsa
import base64
import hashlib
import datetime
import sys
import json

from urllib2 import HTTPPasswordMgrWithDefaultRealm, HTTPBasicAuthHandler, Request, build_opener
from urllib import urlencode


KEY_ID = 'ffdb30c23addd822e851e15cdd30ea819251d2fbb9bc59c4803f006dcc900191'
CONTAINER = 'iCloud.cloud.uchicago.CloudyWithAChanceOfErros'

def cloudkit_request(cloudkit_resource_url, data):
    """Uses HTTP GET or POST to interact with CloudKit. If data is empty, Uses
    GET, else, POSTs the data.
    """

    # Get ISO 8601 date, cut milliseconds.
    date = datetime.datetime.utcnow().isoformat()[:-7] + 'Z'

    # Load JSON request from config.
    _hash = hashlib.sha256(data.encode('utf-8')).digest()
    body = base64.b64encode(_hash).decode('utf-8')

    # Construct URL to CloudKit container.
    web_service_url = '/database/1/' + CONTAINER  + cloudkit_resource_url

    # Load API key from config.
    key_id = KEY_ID
```

ecdsa is not a standard module on app engine

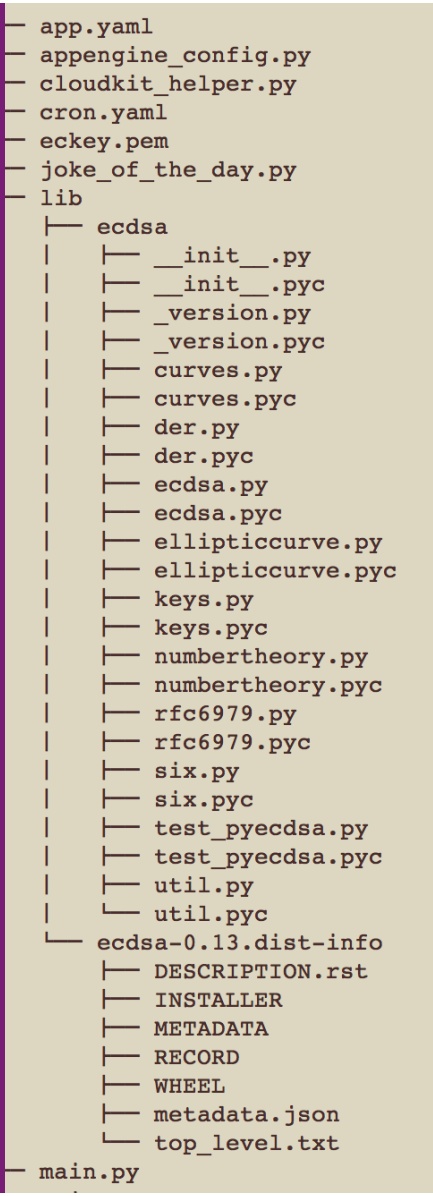
APP ENGINE

INSTALL THE AN EXTERNAL LIBRARY
IN YOUR PROJECT

```
cd gae_dir

mkdir lib

# pip install -t lib/ <library_name>
pip install -t lib/ ecdsa
```

APP ENGINE

- Installed in full in the same directory
- Allows you to use it as pure python code



APP ENGINE

- Add a new file: appengine_config.py

```
# appengine_config.py
#

from google.appengine.ext import vendor

# Add any libraries install in the "lib" folder.
vendor.add('lib')
```

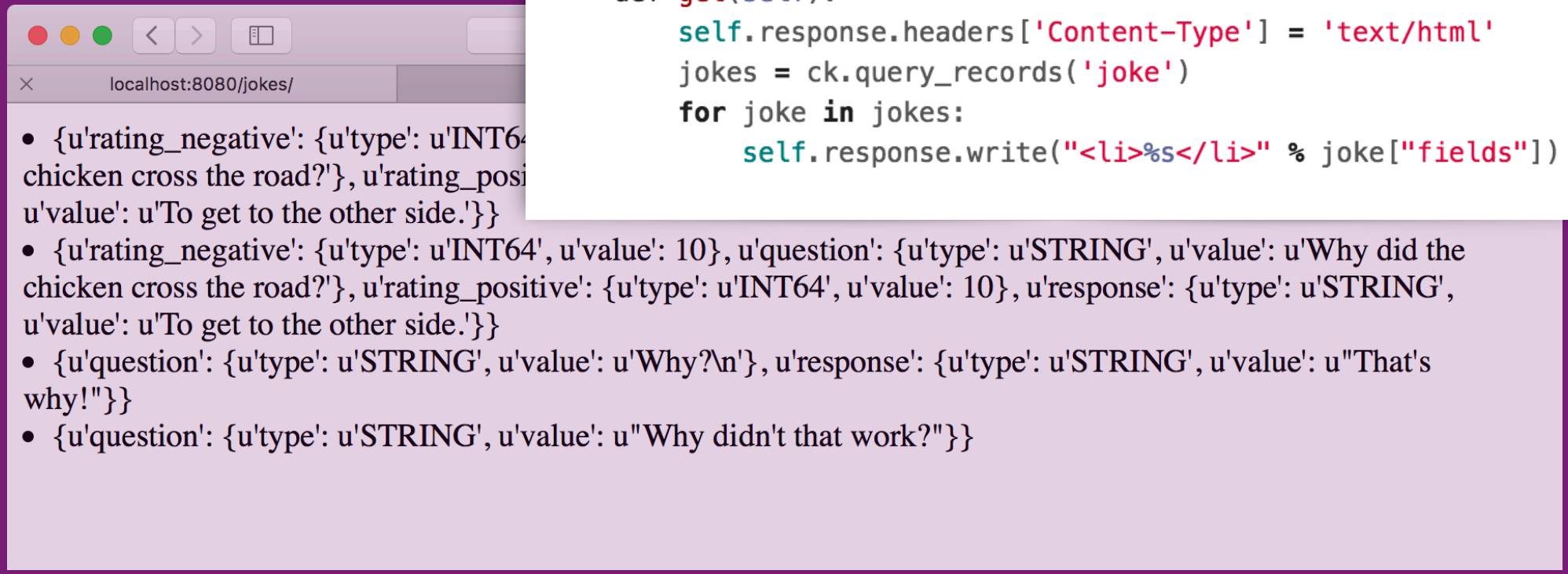
APP ENGINE

- main.py
- Handlers
  - Processing
  - Posting record

```
import webapp2

import sys
import json

import cloudkit_helper as ck
from joke_of_the_day import JokeOfTheDay

class MainPage(webapp2.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/html'
        self.response.write("home")

class ProcessJokes(webapp2.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/html'
        jokes = ck.query_records('joke')
        for joke in jokes:
            self.response.write("<li>%s</li>" % joke["fields"])


app = webapp2.WSGIApplication([
    ('/', MainPage),
    ('/jokes/', ProcessJokes),
    ('/tasks/jokeoftheday/', JokeOfTheDay),
], debug=True)
```

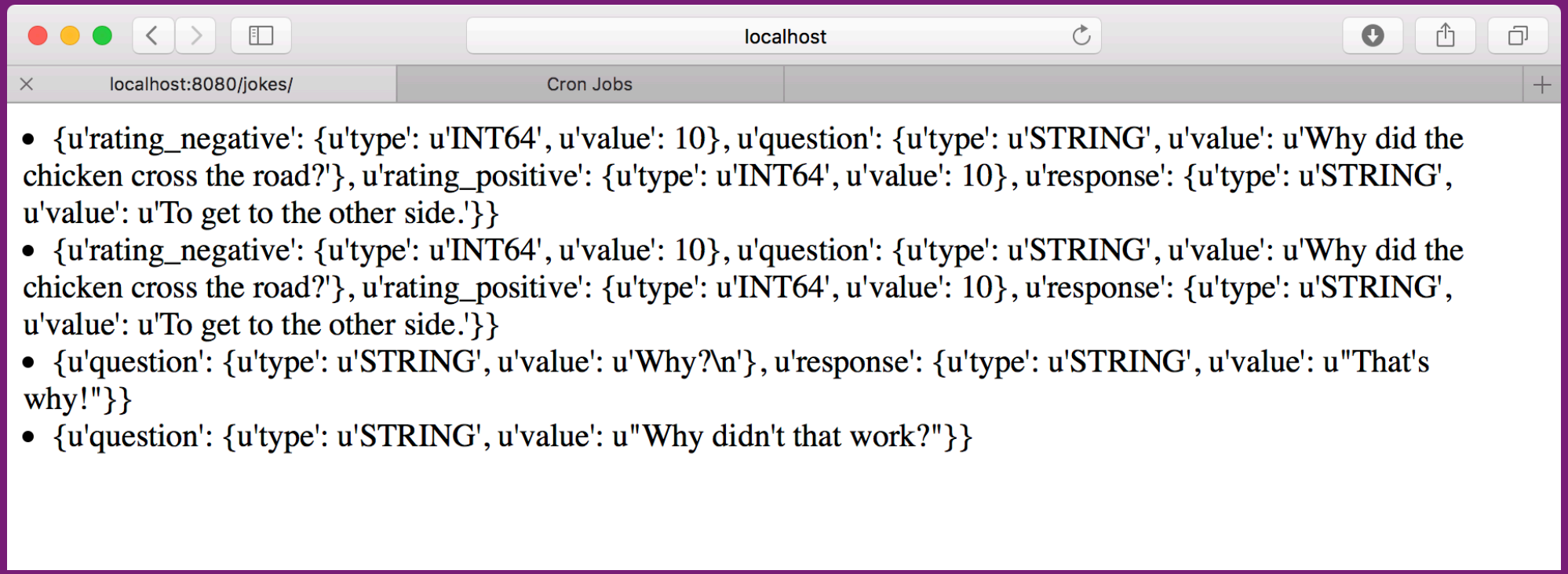
APP ENGINE

```
class ProcessJokes(webapp2.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/html'
        jokes = ck.query_records('joke')
        for joke in jokes:
            self.response.write("<li>%s</li>" % joke["fields"])
```



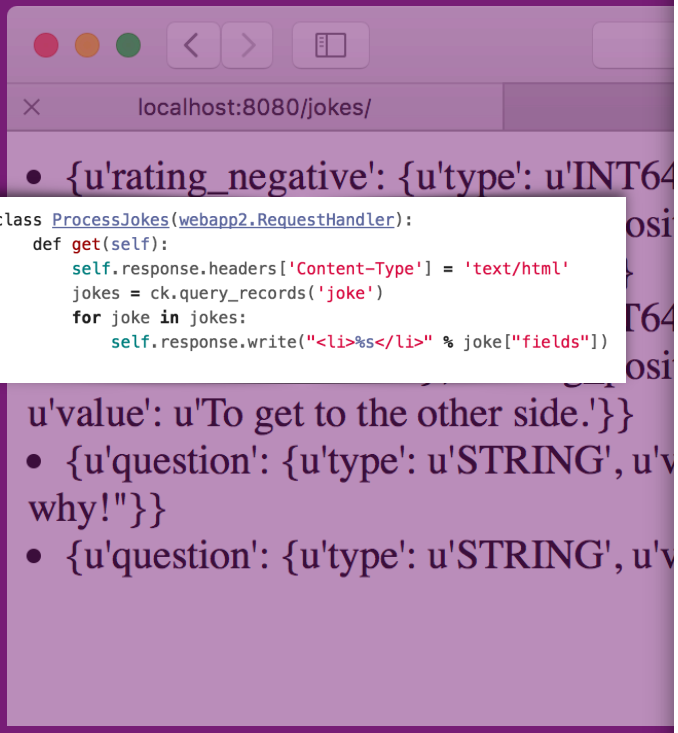
- http://localhost:8080/jokes/

APP ENGINE



- http://localhost:8080/jokes/

APP ENGINE




```
class ProcessJokes(webapp2.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/html'
        jokes = ck.query_records('joke')
        for joke in jokes:
            self.response.write("<li>%s</li>" % joke["fields"])
```

```
def query_records(record_type):
    """Queries CloudKit for all records of type record_type."""
    json_query = {
        'query': {
            'recordType': record_type
        }
    }

    records = []
    while True:
        result_query_authors = cloudkit_request(
            '/development/public/records/query',
            json.dumps(json_query))
        result_query_authors = json.loads(result_query_authors['content'])

        records += result_query_authors['records']

        if 'continuationMarker' in result_query_authors.keys():
            json_query['continuationMarker'] = \
                result_query_authors['continuationMarker']
        else:
            break
```

REQUEST WITH CURSORS

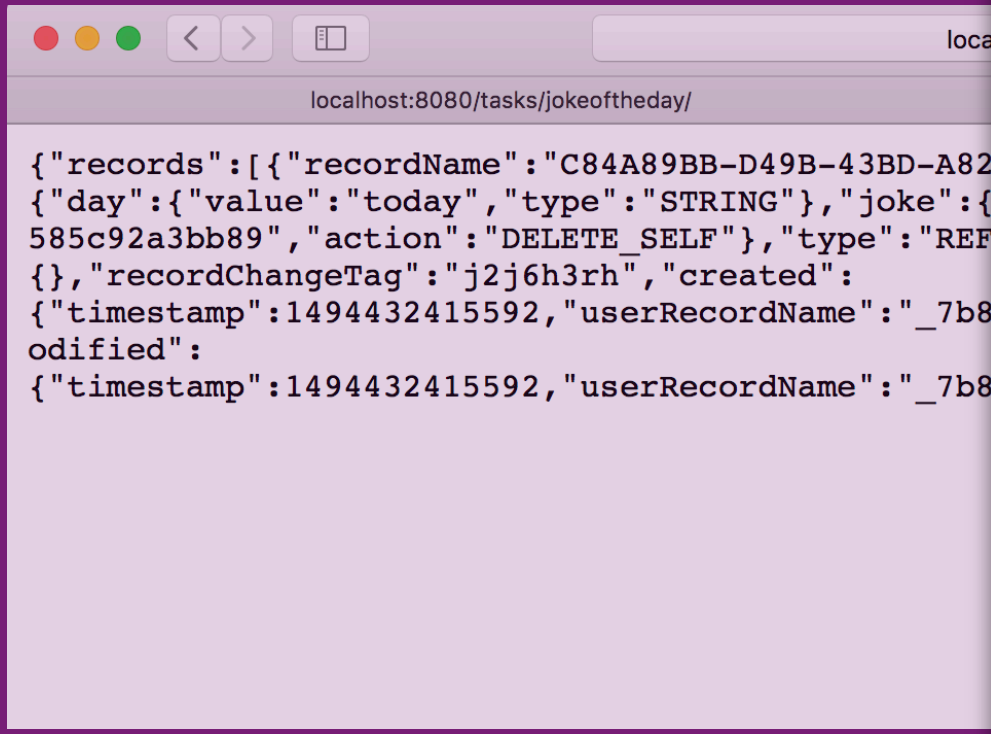
- http://localhost:8080/jok

APP ENGINE

```
    def get(self):
        self.response.headers['Content-Type'] = 'text/html'
        jokes = ck.query_records('joke')
        for joke in jokes:
            self.response.write("<li>%s</li>" % joke["fields"])


app = webapp2.WSGIApplication([
    ('/', MainPage),
    ('/jokes/', ProcessJokes),
    ('/tasks/jokeoftheday/', JokeOfTheDay),
], debug=True)
```

- http://localhost:8080/tasks/jokeoftheday/

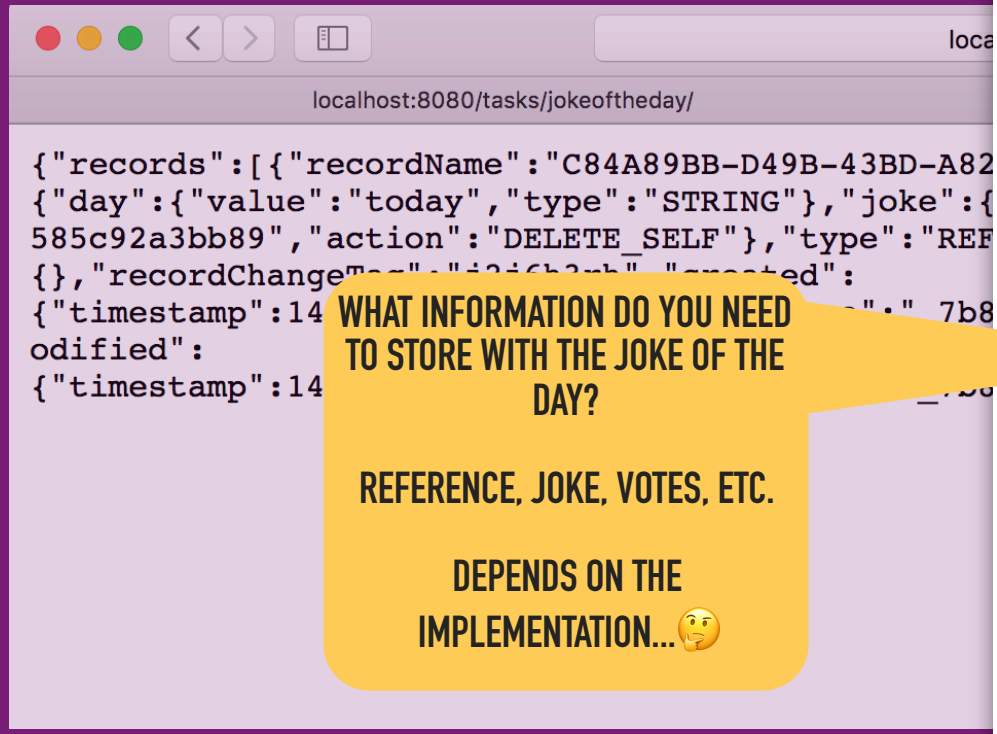
APP ENGINE



- http://localhost:8080/tasks/jokeofthe

```
class JokeOfTheDay(webapp2.RequestHandler):
    #
    # Create a new joke of the day record and post it to iCloud
    # Note: We are hard coming the reference here, you should get
    # if from the processing of the daily joke ratings
    def get(self):


        new_joke_of_the_day_data = {
        'operations': [{
            'operationType': 'create',
            'record': {
                'recordType': 'Daily',
                'fields': {
                    'day': {'value': 'today'},
                    'joke': {
                        'value': {
                            'recordName': '60d82b7d-32ef-4e91-9c12-585c92a3bb89',
                        'zoneID:': {
                            'zoneName': '_defaultZone'
                            },
                            'action': 'DELETE_SELF'
                        }
                    }
                }
            }
        }]
        }

        #print('Posting operation to create quote...')
        result_modify_jokes = ck.cloudkit_request(
        '/development/public/records/modify',
        json.dumps(new_joke_of_the_day_data))

        self.response.headers['Content-Type'] = 'text/json'
        self.response.write(result_modify_jokes['content'])
```

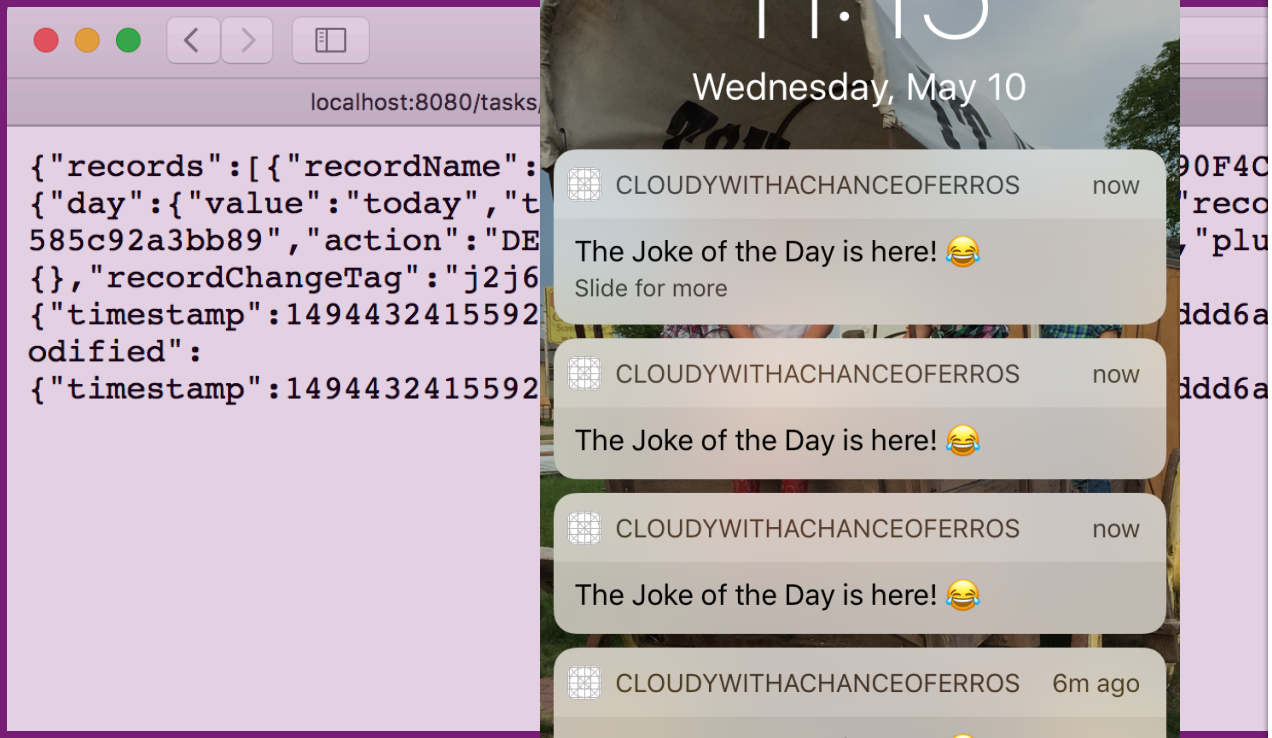
APP ENGINE




```
class JokeOfTheDay(webapp2.RequestHandler):
    #
    # Create a new joke of the day record and post it to iCloud
    # Note: We are hard coming the reference here, you should get
    # if from the processing of the daily joke ratings
    def get(self):


        new_joke_of_the_day_data = {
        'operations': [{
            'operationType': 'create',
            'record': {
                'recordType': 'Daily',
                'fields': {
                    'day': {'value': 'today'},
                    'joke': {
                        'value': {
                            'recordName': '60d82b7d-32ef-4e91-9c12-585c92a3bb89',
                        'zoneID:': {
                            'zoneName': '_defaultZone'
                            },
                            'action': 'DELETE_SELF'
                        }
                    }
                }
            }
        }]
        }

        #print('Posting operation to create quote...')
        result_modify_jokes = ck.cloudkit_request(
        '/development/public/records/modify',
        json.dumps(new_joke_of_the_day_data))

        self.response.headers['Content-Type'] = 'text/json'
        self.response.write(result_modify_jokes['content'])
```

- http://localhost:8080/tasks/jokeofthe

APP ENGINE



- http://localhost:808


```python
def cloudkit_request(cloudkit_resource_url, data):
    """Uses HTTP GET or POST to interact with CloudKit. If data is empty, Uses
    GET, else, POSTs the data.
    """

    # Get ISO 8601 date, cut milliseconds.
    date = datetime.datetime.utcnow().isoformat()[:-7] + 'Z'

    # Load JSON request from config.
    _hash = hashlib.sha256(data.encode('utf-8')).digest()
    body = base64.b64encode(_hash).decode('utf-8')

    # Construct URL to CloudKit container.
    web_service_url = '/database/1/' + CONTAINER  + cloudkit_resource_url

    # Load API key from config.
    key_id = KEY_ID

    # Read out certificate file corresponding to API key.
    with open('eckey.pem', 'r') as pem_file:
        signing_key = ecdsa.SigningKey.from_pem(pem_file.read())

    # Construct payload.
    unsigned_data = ':'.join([date, body, web_service_url]).encode('utf-8')

    # Sign payload via certificate.
    signed_data = signing_key.sign(unsigned_data,
                                   hashfunc=hashlib.sha256,
                                   sigencode=ecdsa.util.sigencode_der)

    signature = base64.b64encode(signed_data).decode('utf-8')

    headers = {
        'X-Apple-CloudKit-Request-KeyID': key_id,
        'X-Apple-CloudKit-Request-ISO8601Date': date,
        'X-Apple-CloudKit-Request-SignatureV1': signature
    }

    if data:
        req_type = 'POST'
    else:
        req_type = 'GET'

    result = curl('https://api.apple-cloudkit.com' + web_service_url,
                  req_type=req_type,
                  data=data,
                  headers=headers)

    return result
```

APP ENGINE

```yaml
# cron.yaml
#
# To test in developement server goto http://localhost:8000/cron
#

cron:
- description: daily joke summary and notification
  url: /tasks/jokeoftheday/
  schedule: every day 08:00
  retry_parameters:
    min_backoff_seconds: 60
    max_doublings: 5
```

- Add a new file: cron.yaml

APP ENGINE

CRON JOB

Google App Engine

Development SDK 1.9.50

dev~None

- Instances
- Datastore Viewer
- Datastore Indexes
- Datastore Stats
- Interactive Console
- Memcache Viewer
- Blobstore Viewer
- Task Queues
- Cron Jobs

Cron Jobs

Request to /tasks/jokeoftheday/ succeeded!

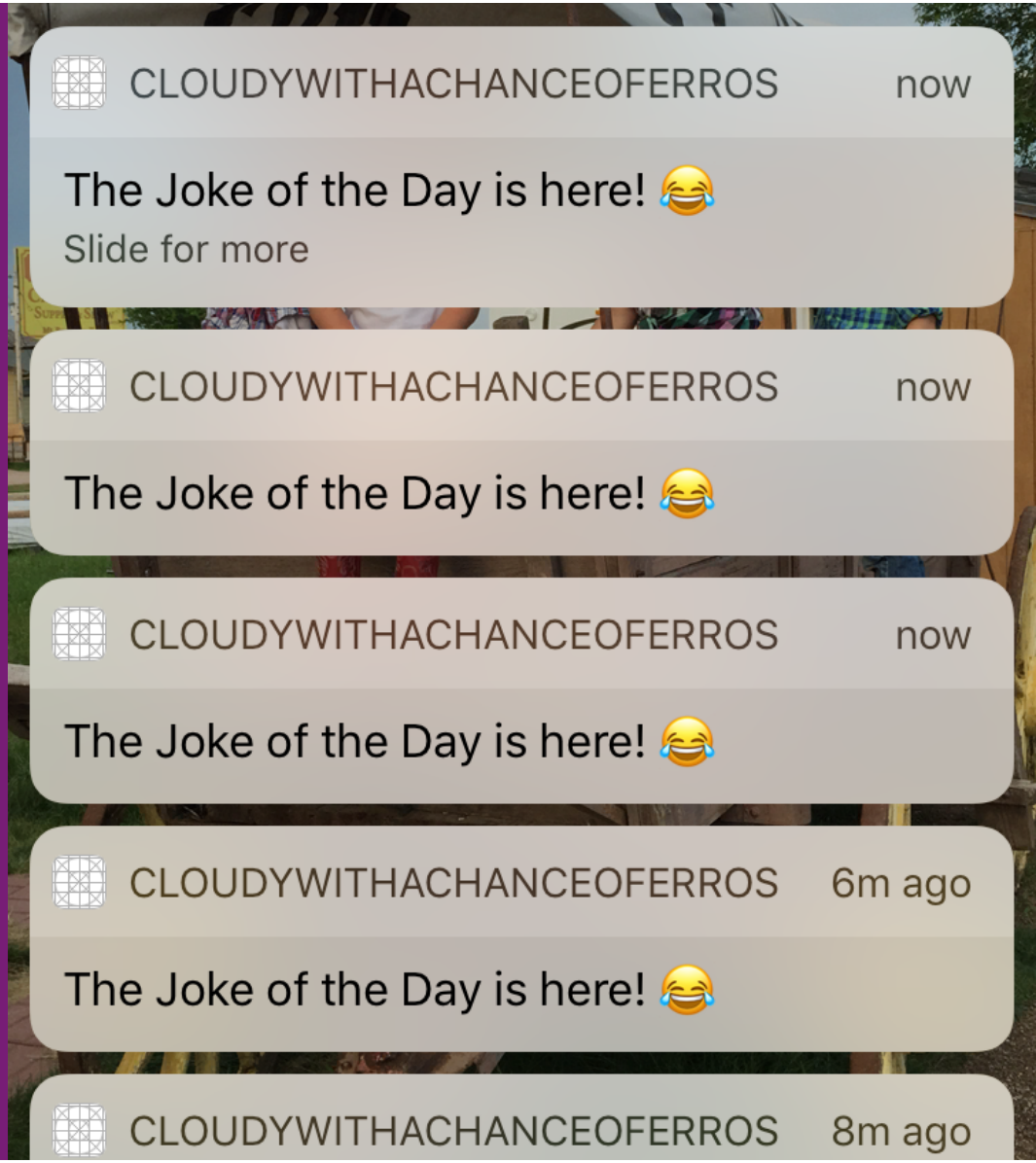
Cron Job	Schedule
/tasks/jokeoftheday/ daily joke summary and notification	**every day 08:00** Run now In production, this would run at these times: 2017-05-11 08:00:00Z 16:20:41.791990 from now 2017-05-12 08:00:00Z 1 day, 16:20:41.791990 from now 2017-05-13 08:00:00Z 2 days, 16:20:41.791990 from now

BREAK TIME 

CUSTOM NOTIFICATIONS FROM SUBSCRIPTIONS

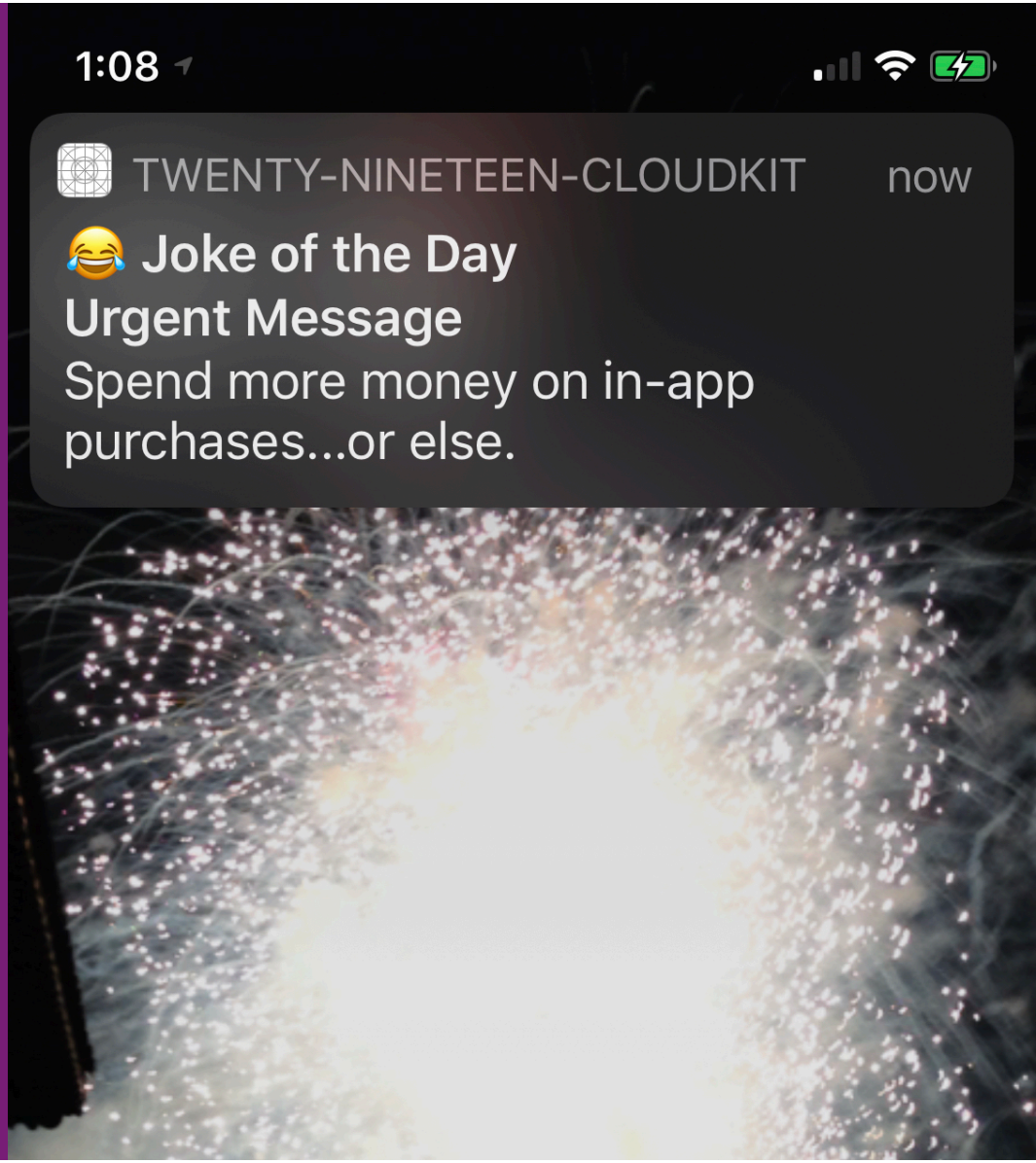
CUSTOM NOTIFICATIONS

- Generic notifications from subscriptions
- This may not be the great user experience we want



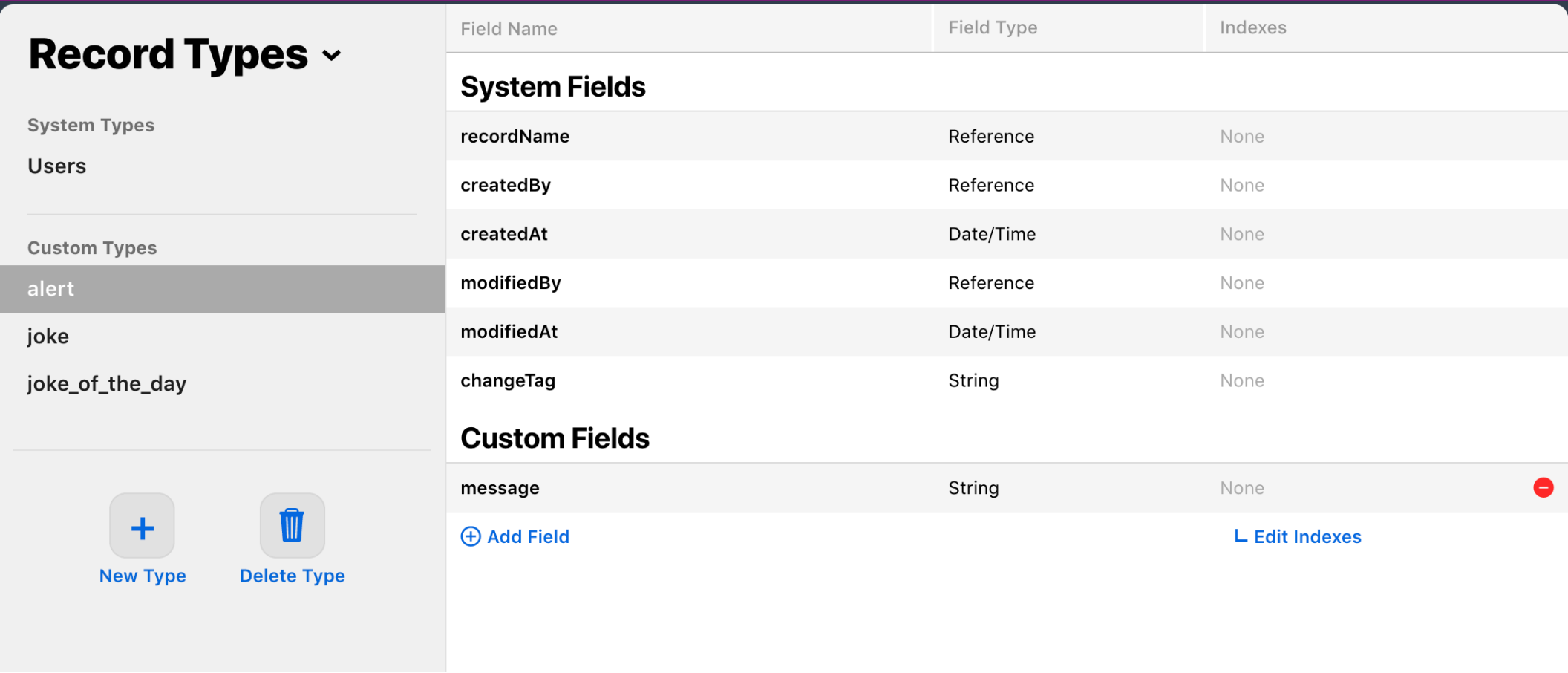
CUSTOM NOTIFICATIONS

- Generic notifications from subscriptions
- This may not be the great user experience we want



CUSTOM NOTIFICATIONS

CUSTOM NOTIFICATIONS



CUSTOM NOTIFICATIONS

```
func registerSilentAlertSubscription() {
  let uuid: UUID = UUID()
  let identifier = "\(uuid)-alert"


  // Create the notification that will be delivered
  let notificationInfo = CKSubscription.NotificationInfo()
  notificationInfo.shouldSendContentAvailable = true
  notificationInfo.desiredKeys = ["message"]

  // Create the subscription
  let subscription = CKQuerySubscription(recordType: "alert",
                                         predicate: NSPredicate(value: true),
                                         subscriptionID: identifier,
                                         options: [CKQuerySubscription.Options.firesOnRecordCreation])

  subscription.notificationInfo = notificationInfo
  CKContainer.default().publicCloudDatabase.save(subscription,
                                                 completionHandler: ({returnRecord, error in
                                                  if let err = error {
                                                    print("ALERT: subscription failed \(err.localizedDescription)")
                                                  } else {
```

MESSAGE IS PASSED ALL THE WAY THROUGH TO THE IOS APP

CUSTOM NOTIFICATIONS

CLOUDKIT HELPER

CUSTOM NOTIFICATIONS

- alert.py
- Prompt user for a message and then adds a new record to the public database
  - Triggers the alert

```python
import sys
import json

import cloudkit_helper as ck

def add_alert(message):
    """Create a new alert with a custom message"""

    data = {
        'operations': [{
            'operationType': 'create',
            'record': {
                'recordType': 'alert',
                'fields': {
                    'message': {
                        'value': message,
                    }
                }
            }
        }]
    }

    print('Posting operation to create quote...')
    result = ck.cloudkit_request(
        '/development/public/records/modify', json.dumps(data))
    print(result['content'])


def main():
    message = input("What would you like to say to your users?\n")
    add_alert(message)


if __name__ == '__main__':
    main()
```

CUSTOM NOTIFICATIONS

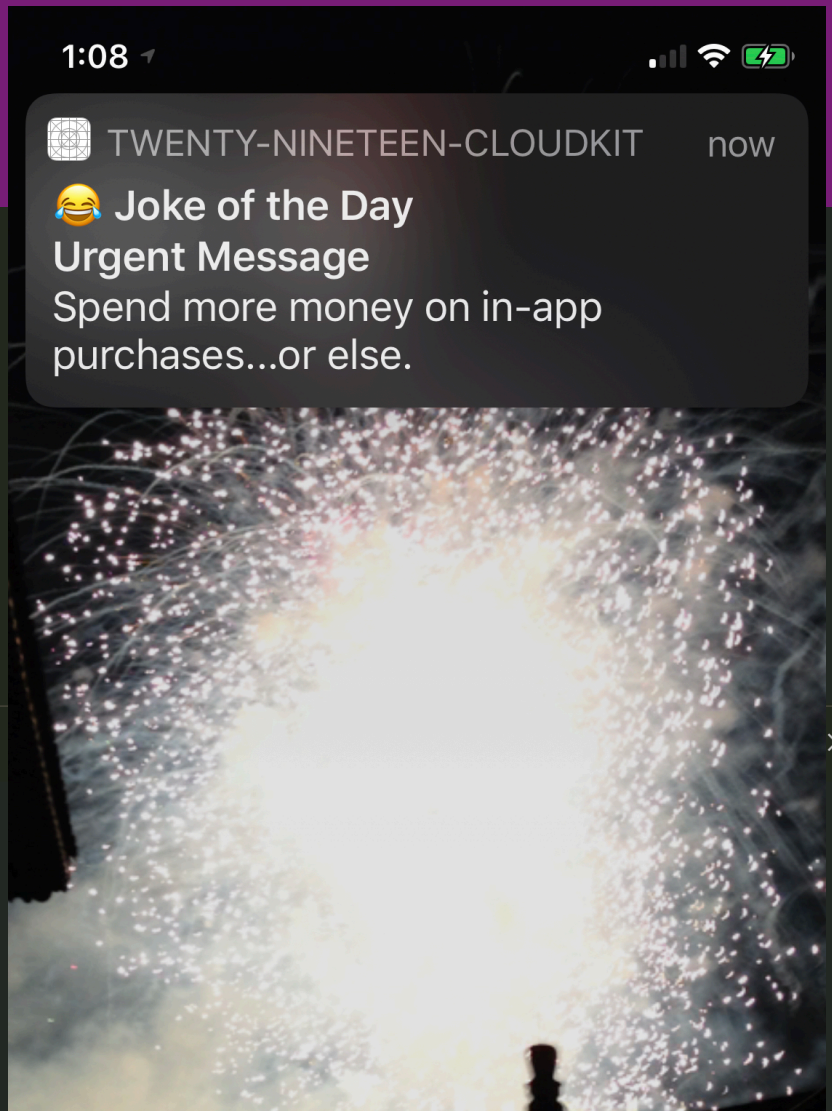
```
def add_alert(message):
    """Create a new alert with a custom message"""

    data = {
        'operations': [{
            'operationType': 'create',
            'record': {
                'recordType': 'alert',
                'fields': {
                    'message': {
                        'value': message,
                    }
                }
            }
        }]
    }

    print('Posting operation to create quote...')
    result = ck.cloudkit_request(
        '/development/public/records/modify', json.dumps(data))
    print(result['content'])
```

CUSTOM NOTIFICATIONS

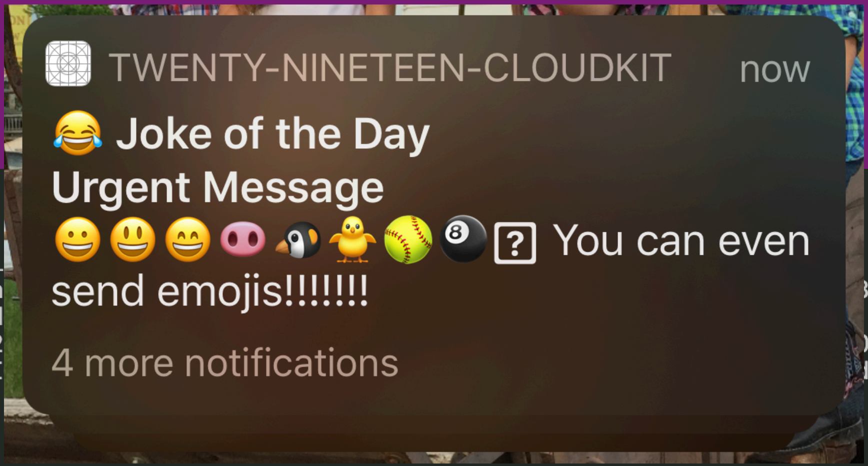
```
12                'record' : {
13                    'recordType': 'alert',
14                    'fields': {
15                        'message': {
16                            'value': message,
17                        }
18                    }
19                }
20            }]
21        }
22
23        print('Posting operation to create quote...')
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
3-2
What would you like to say to your users?
Buy more in app purchases...or else
```



CUSTOM NOTIFICATIONS



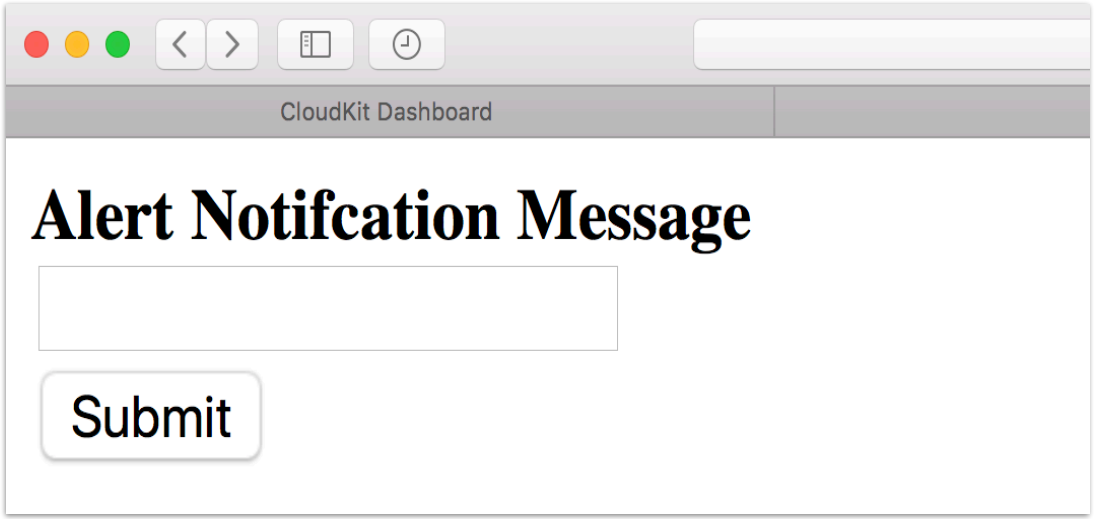

```
What would you like to say to your users?
😀😃😄🐽🐧🐥🥎🎱 You can even send emojis!!!!!!!
Posting operation to create quote...
URL:  https://api.apple-cloudkit.com/database/1/iCloud.mobi.uchicago.twenty-ni
Data:  {"operations": [{"operationType": "create", "record": {"recordType": "a          3\ud
83d\ude04\ud83d\udc3d\ud83d\udc27\ud83d\udc25\ud83e\udd4e\ud83c\udfb1\ud83e\ud
Headers:  {'X-Apple-CloudKit-Request-KeyID': '99e04a281f141f4bce822833bd8e8782          )860
1Date': '2019-11-12T19:29:27Z', 'X-Apple-CloudKit-Request-SignatureV1': 'MEYCI          JGB0
BlmlJE6fNLf2UmpUZQsVC9s+'}
Params:  None
{
  "records" : [ {
    "recordName" : "0B361BB6-5E9D-4D10-8531-F607A96C480B",
    "recordType" : "alert",
    "fields" : {
      "message" : {
        "value" : "\uD83D\uDE00\uD83D\uDE03\uD83D\uDE04\uD83D\uDC3D\uD83D\uDC27\uD83D\uDC25\uD83E\uDD4E\uD83C\uDFB1\uD83E\uDE80 You can even send
 mojis!!!!!!!",
        "type" : "STRING"
      }
    },
    "pluginFields" : { },
    "recordChangeTag" : "k2w91wdk",
    "created" : {
      "timestamp" : 1573586969406,
      "userRecordName" : "_8a76cb86f6390ecf95f548796a270cc8",
      ...
      06,
      cb86f6390ecf95f548796a270cc8",
```


EMOJIS SURVIVE

CUSTOM NOTIFICATIONS

- Create an amazing form that takes some text for the custom message

```
def get(self):
    self.response.headers['Content-Type'] = 'text/html'
    html = """
    <form action="/alerts/" method="post">
    <b>Alert Notifcation Message</b><br>
    <input type="text" name="message" value=""><br>
    <input type="submit" value="Submit">
    </form>
    """
    self.response.write(html)
```



IOS APP

CUSTOM NOTIFICATIONS

- Handle the remote notification
  - Read the APNS data
  - Extract the message
  - Create local notification

```
/// Called for push notifications
/// In this case, we are getting the changed record from cloudkit and then creating a new notification
func application(_ application: UIApplication,
                 didReceiveRemoteNotification userInfo: [AnyHashable : Any],
                 fetchCompletionHandler completionHandler: @escaping (UIBackgr

  // Get the APS notification data
  print(userInfo)
  let aps = userInfo["aps"] as! [String: AnyObject]

  // Do Something with the content available data
  let contentAvailable = aps["content-available"] as! Int
  if contentAvailable == 1 {
    print("APS: \(aps)")
    let cloudKitInfo = userInfo["ck"] as! [String: AnyObject]
    let recordId = cloudKitInfo["qry"]?["rid"] as! String
    let field = cloudKitInfo["qry"]?["af"] as! [String: AnyObject]

    // A message indicates that we have a received a silent notification
    if let message = field["message"] as? String {
      local_notification_from_silent(message: message, recordId: recordId)
    }
    completionHandler(.newData)

  } else {
    completionHandler(.noData)
  }
}
```

CUSTOM NOTIFICATIONS

SILENT NOTIFICATION CONTENT

```
[AnyHashable("aps"): {
    "content-available" = 1;
}, AnyHashable("ck"): {
    ce = 2;
    cid = "iCloud.cloud.uchicago.Cloudy...AChanceOfErros";
    nid = "2b661f39-5617-4a1c-a88...09cd9eae7a";
    qry =     {
        af =         {
            message = "Where are my emojis?";
        };
        dbs = 2;
        fo = 1;
        rid = "A0915CF5-E4E7-4C76-856E-48CE01ADCACE";
        sid = "14725FE6-9C8F-4BDC-96FF-3815E5FE1E3B-alert";
        zid = "_defaultZone";
        zoid = "_defaultOwner";
    };
}]
```

FIELD I REQUESTED

RECORD THAT MATCHED THE QUERY

CUSTOM NOTIFICATIONS

```
func local_notification_from_silent(message: String, recordId: String) {
  print("Creating local notification")

  // Create notification content
  let content = UNMutableNotificationContent()
  content.title = "😂 Joke of the Day"
  content.subtitle = "Urgent Message"
  content.body = message
  content.sound = UNNotificationSound.default

  // Set up trigger
  let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 1,repeats: false)

  // Create the notification request
  let center = UNUserNotificationCenter.current()
  let identifier = recordId
  let request = UNNotificationRequest(identifier: identifier,
                                      content: content, trigger: trigger)
  center.add(request, withCompletionHandler: { (error) in
    if let error = error {
      print("Something went wrong: \(error)")
    }
  })
```

CREATE A NOTIFICATION

CUSTOM NOTIFICATIONS

```
func local_notification_from_silent(message: String, recordId: String) {
  print("Creating local notification")

  // Create notification content
  let content = UNMutableNotificationContent()
  content.title = "😂 Joke of the Day"
  content.subtitle = "Urgent Message"
  content.body = message
  content.sound = UNNotificationSound.default

  // Set up trigger
  let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 1,repeats: false)

  // Create the notification request
  let center = UNUserNotificationCenter.current()
  let identifier = recordId
  let request = UNNotificationRequest(identifier: identifier,
                                      content: content, trigger: trigger)
  center.add(request, withCompletionHandler: { (error) in
    if let error = error {
      print("Something went wrong: \(error)")
    }
  })
```

CREATE A TRIGGER

TIME, 1 SEC

CUSTOM NOTIFICATIONS

```
func local_notification_from_silent(message: String, recordId: String) {
  print("Creating local notification")

  // Create notification content
  let content = UNMutableNotificationContent()
  content.title = "😂 Joke of the Day"
  content.subtitle = "Urgent Message"
  content.body = message
  content.sound = UNNotificationSound.default

  // Set up trigger
  let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 1,repeats: false)

  // Create the notification request
  let center = UNUserNotificationCenter.current()
  let identifier = recordId
  let request = UNNotificationRequest(identifier: identifier,
                                      content: content, trigger: trigger)
  center.add(request, withCompletionHandler: { (error) in
    if let error = error {
      print("Something went wrong: \(error)")
    }
  })
```

ADD TO NOTIFICATION CENTER

CORE DATA

CORE DATA

Technique	Pros	Cons
Archiving	Allows ordered relationships (arrays, not sets). Easy to deal with versioning.	Reads all the objects in (no faulting). No incremental updates.
Web Service	Makes it easy to share data with other devices and applications.	Requires a server and a connection to the internet.
Core Data	Lazy fetches by default. Incremental updates.	Versioning is awkward (but can certainly be done using an `NSModelMapping`). No real ordered relationships (at the time this is being written).

- Options for data storage and persistence

CORE DATA

- A framework that provides object-relational mapping
  - Converts objective-c objects into data stored on disk
- Creates a visual mapping between database and objects
- An object-oriented API that conforms to MVC paradigm
- API for constructing detailed queries
- Access the data using properties on the objects

Introduction to Core Data Programming Guide

The Core Data framework provides generalized and automated solutions to common tasks associate
and object graph management, including persistence.

Who Should Read This Document

You should read this document to gain an understanding of the Core Data framework. *You must be
of Cocoa development, including the Objective-C language and memory management.*

Important: Although this document provides a thorough treatment of the fundamentals of the Cor
simply reading from start to finish is not a good strategy for learning how to use the technology ef
should typically augment your understanding by following the related tutorials provided in the Refe
description of the recommended learning path, see *Core Data Starting Point*.

Organization of This Document

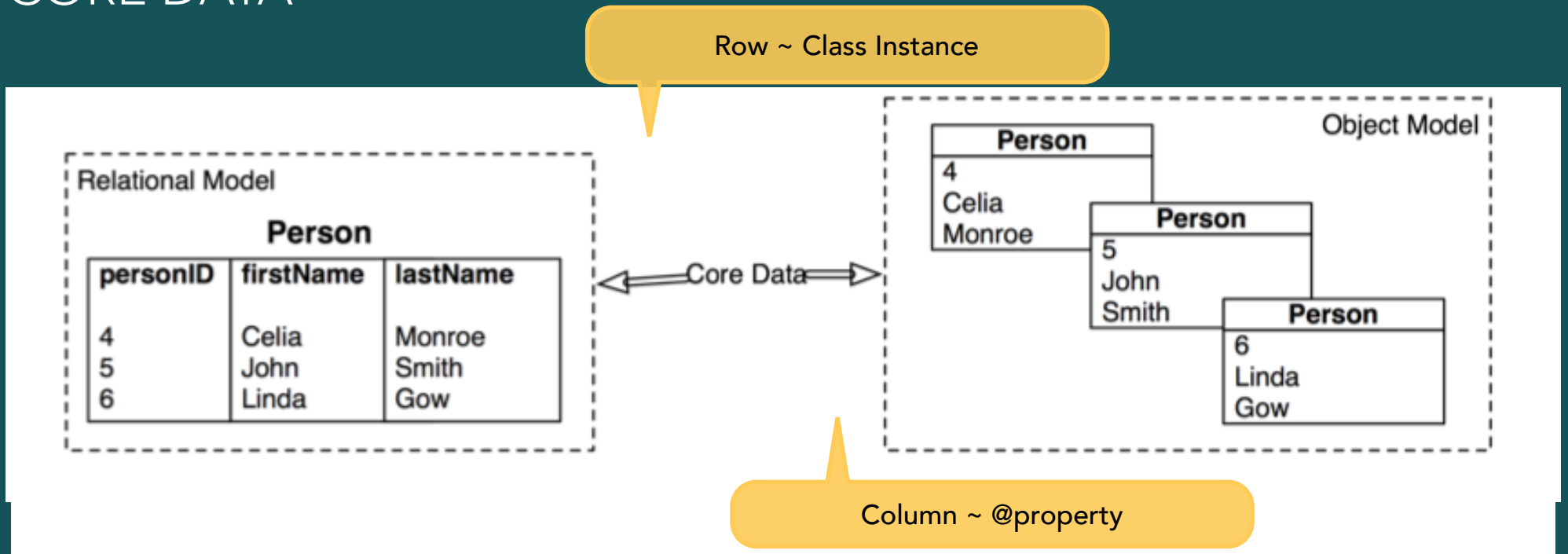
The following articles explain the problems the Core Data Framework addresses, the solutions it pro
functionality, and common tasks you might perform:

- Technology Overview describes what Core Data is and why you might choose to use it.
- Core Data Basics describes the fundamental architecture of the technology.
- Managed Object Models describes the features of a managed object model.
- Using a Managed Object Model describes how you use a managed object model in your applicatio
- Managed Objects describes the features of a managed object, the `NSManagedObject` class, and h

CORE DATA

- Core Data is an object graph management and persistence framework
- Core Data is not a database!

CORE DATA



- Row =~ an instance of the class

- Core Data model file is the description of every entity and their attributes in your application

CORE DATA

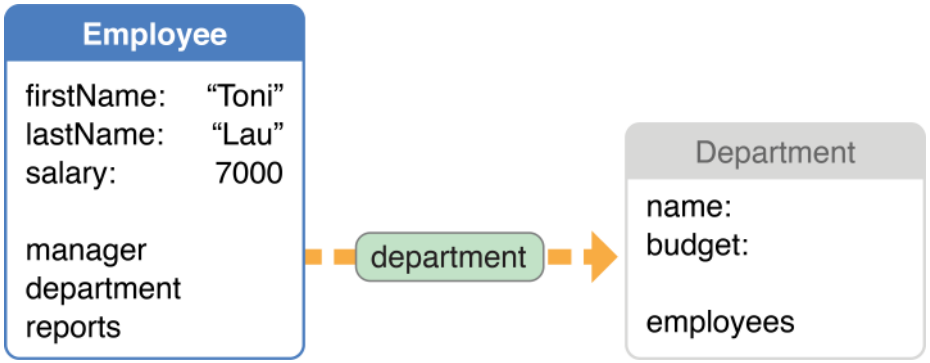
WHAT YOU SHOULD KNOW ABOUT CORE DATA

- Object graph store
  - Model layer represent state of your app
- Persistence
  - Read/write state of model objects to disk
- Querying
  - SQL-like parameter based queries
- Versioning
    - Lightweight migration

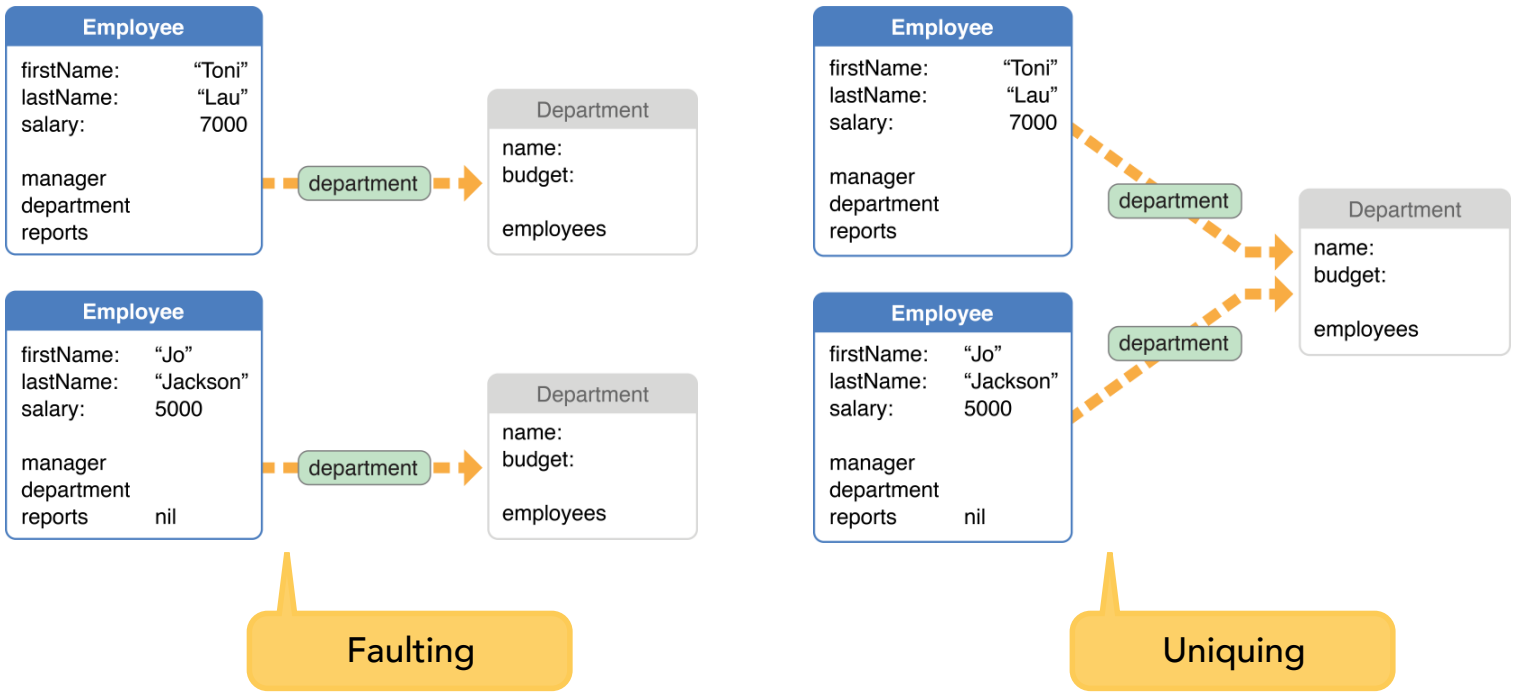
CORE DATA

WHAT YOU PROBABLY DO NOT KNOW

- Change tracking
- Optimize data work flows while in memory
- Faulting
  - Reduce application's memory usage
- Uniquing
  - Ensures that, in a given managed object context, you never have more than one managed object to represent a given record



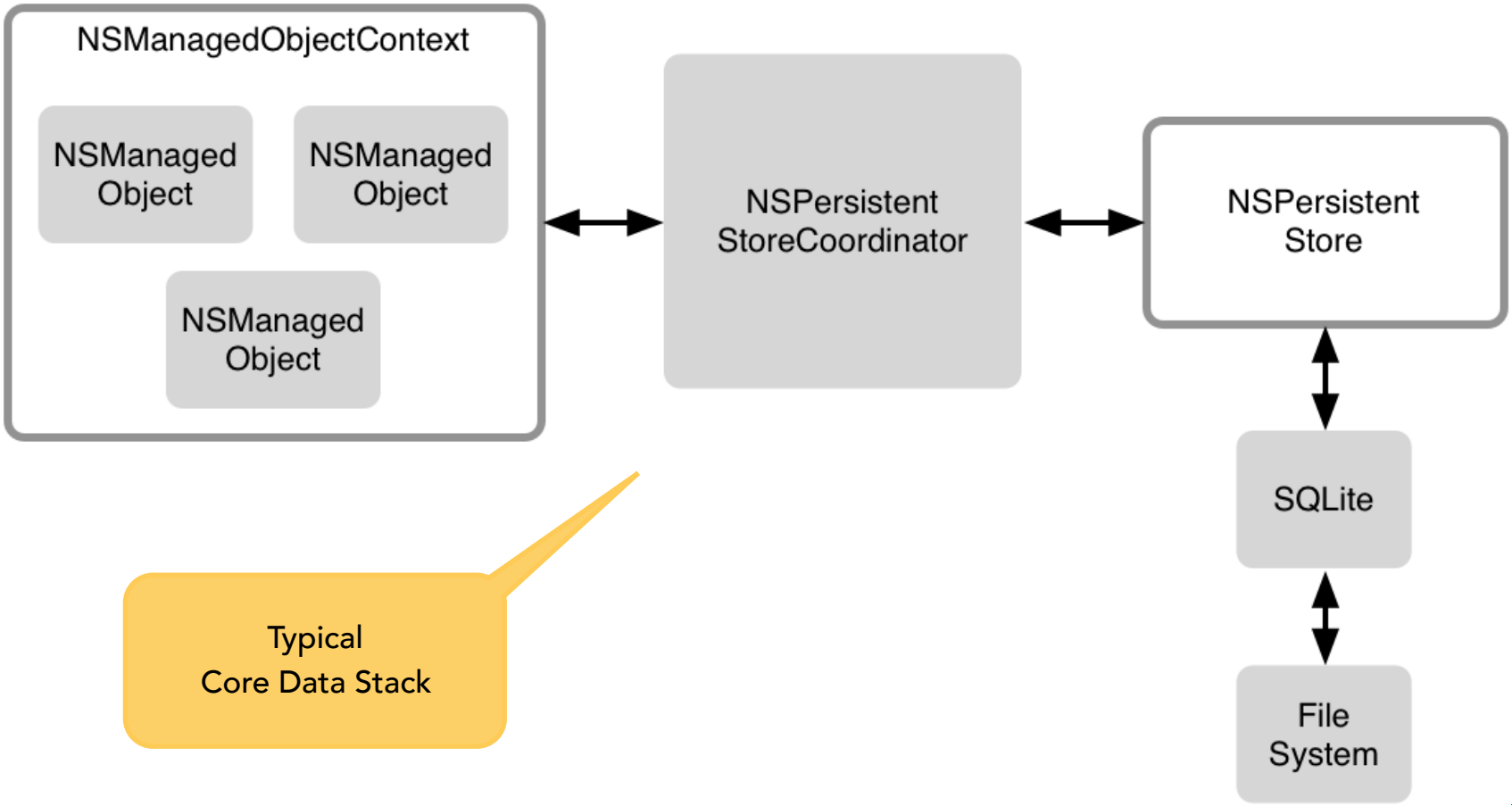
CORE DATA



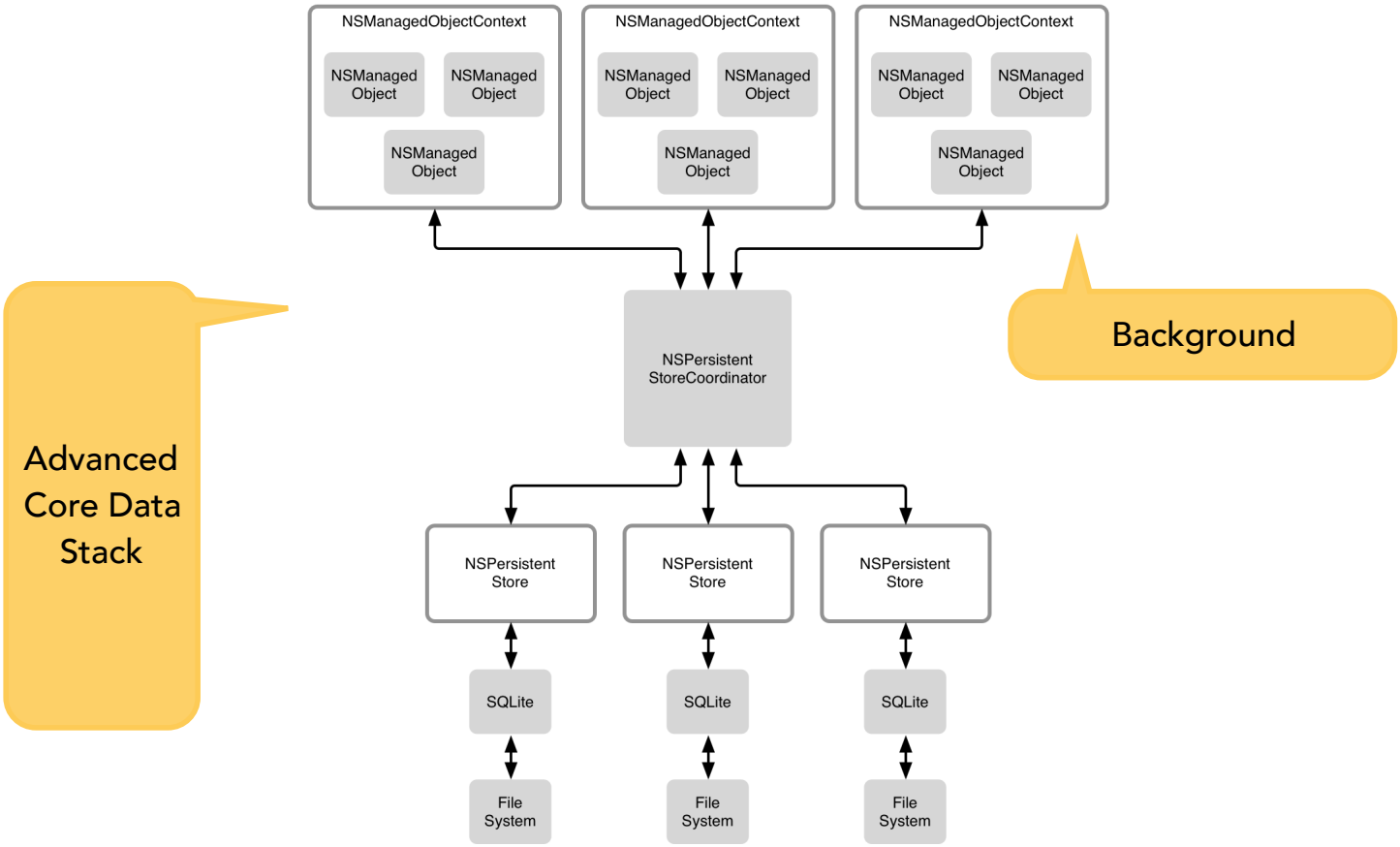
Faulting is a mechanism Core Data employs to reduce your application's memory usage

Uniquing ensures that you never have more than one managed object to represent a given record

CORE DATA

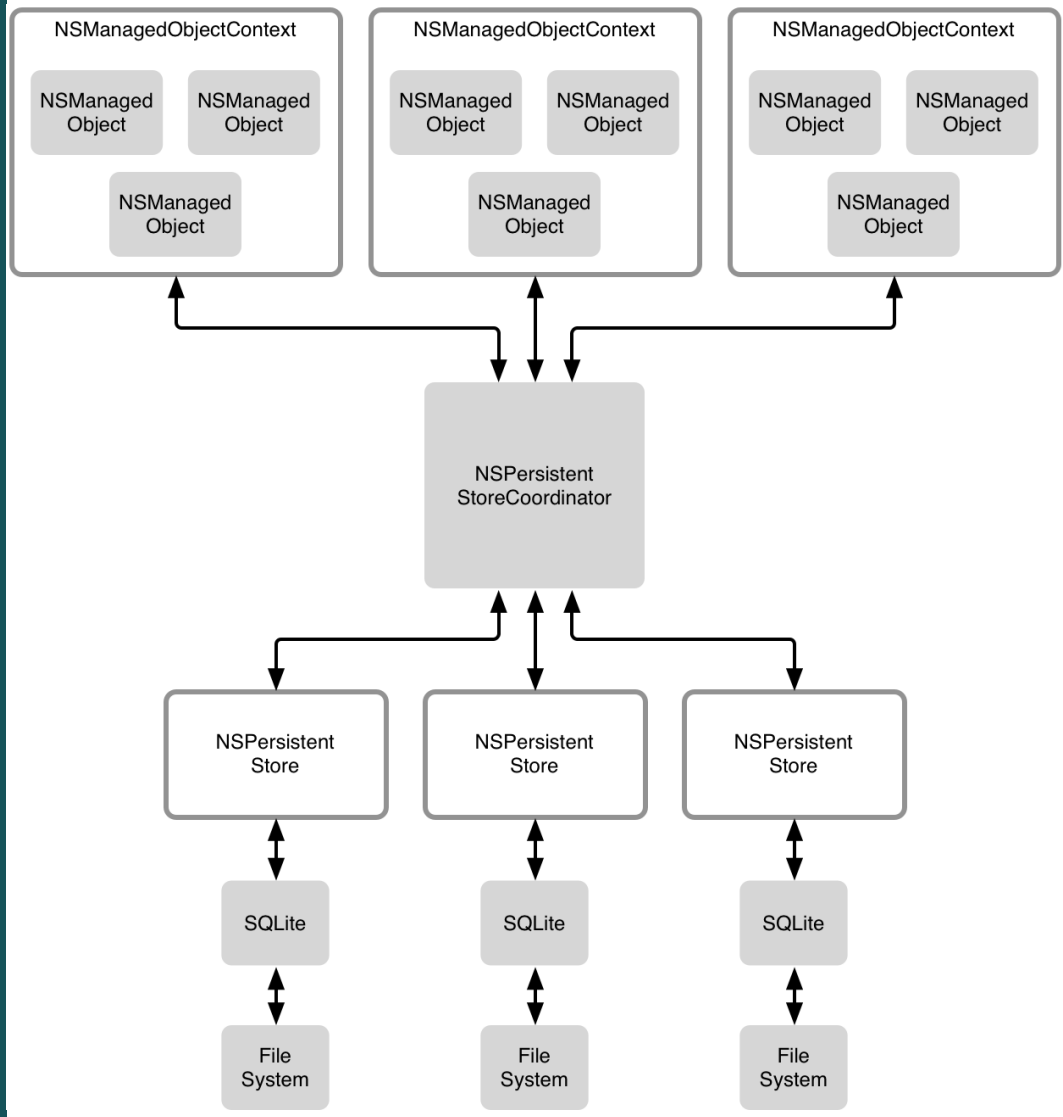

NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSPersistent StoreCoordinator
NSPersistent Store
SQLite
File System
Typical
Core Data Stack

CORE DATA


NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSManagedObjectContext
NSManaged Object
NSManaged Object
NSManaged Object
NSPersistent StoreCoordinator
NSPersistent Store
NSPersistent Store
NSPersistent Store
SQLite
SQLite
SQLite
File System
File System
File System
Advanced Core Data Stack
Background

CORE DATA

- Multiple Managed Object Context (MOC) setups
  - MOC only know about their objects
  - Objects only know about their MOC
- Persistent store doesn't care about what its storing



CORE DATA

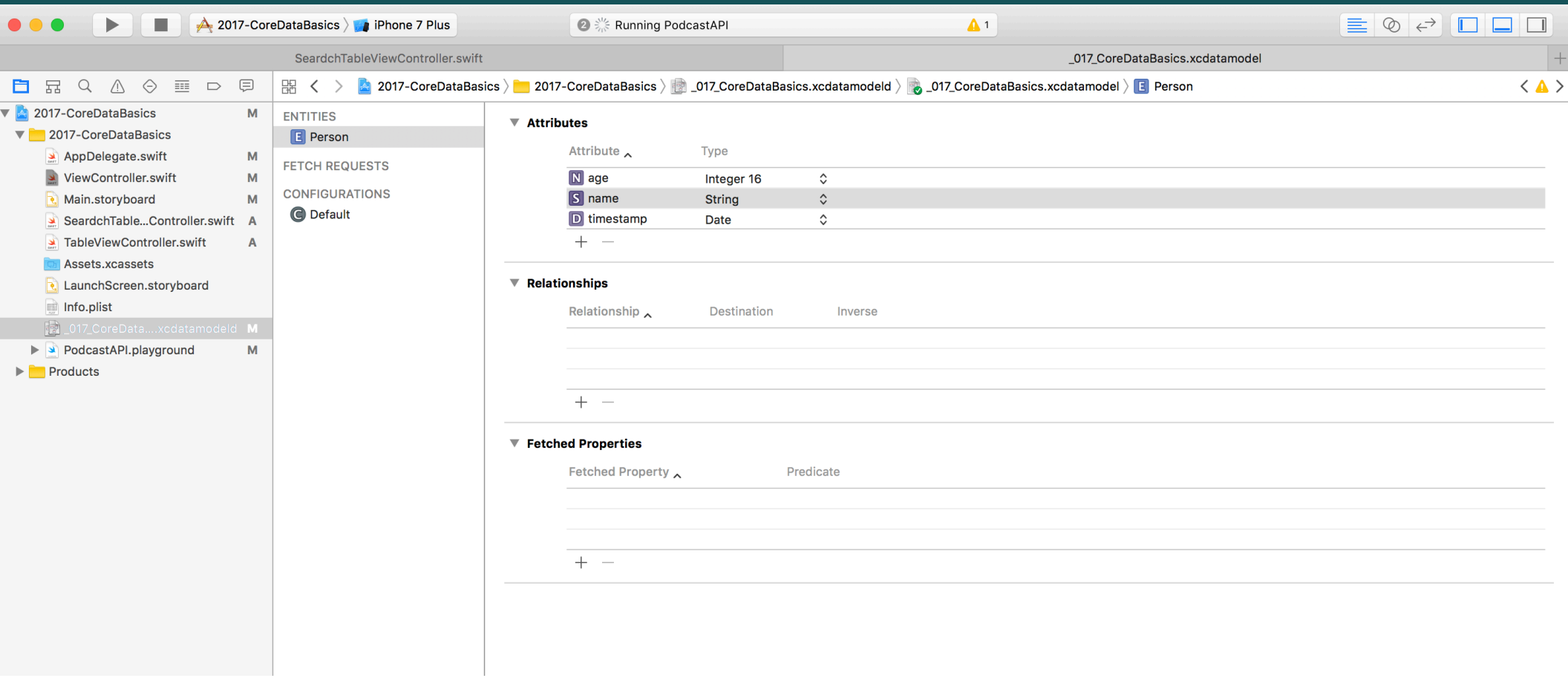
APPLICATION FLOW

- Setup Core Data Stack
  - Object model
  - Persistent store
  - MOC

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, e
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
                in a shipping application, although it may be useful during

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or di
             * The persistent store is not accessible, due to permissions or
             * The device is out of space.
             * The store could not be migrated to the current model version.
             Check the error message to determine what the actual problem wa
             */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
                in a shipping application, although it may be useful during
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA

APPLICATION FLOW



CORE DATA

APPLICATION FLOW

- Create in instance of your entity and set the property values

```
var person = Person(context: context!)
person.name = "Homer"
person.age = 42
person.timestamp = Date() as NSDate

person = Person(context: context!)
person.name = "Marge2"
person.age = 41
person.timestamp =  Date() as NSDate
```

CORE DATA

APPLICATION FLOW

- Changes are kept in memory until you explicitly call 'save'
- What happens?
  - MOC figures out what has changed
  - The managed object context then passes these changes on to the persistent store coordinator
  - The persistent store coordinator coordinates with the store (SQLite3 db) to write into the SQL database on disk

```
func saveContext () {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            // Replace this implementation with code to handle the error
            // appropriately.
            // abort() causes the application to generate a crash log and
            // terminate. You should not use this function in a shipping
            // application, although it may be useful during development.
            let nserror = error as NSError
            NSLog("Unresolved error \(nserror), \(nserror.userInfo)")
            abort()
        }
    }
}
```

CORE DATA

APPLICATION FLOW

- Make a fetch request
- Core Data brings back entity objects
- Fine grain control on the performance

```
// Fetch our person objects
let personFetchRequest: NSFetchRequest<Person> = Person.fetchRequest()

do {
  let fetchedEntities = try context?.fetch(personFetchRequest)

  for user in fetchedEntities! {
    print("> \(user.name!) ##########")
  }

} catch {
  // Do something in response to error condition
}
}
```

CORE DATA

Why Should You Use Core Data?

Already done for you

There are a number of reasons why it may be appropriate for you to use Core Data. One of the simplest metrics is that, with Core Data, the amount of code you write to support the model layer of your application is typically 50% to 70% smaller as measured by lines of code. This is primarily due to the features listed above—the features Core Data provides are features you don't have to implement yourself. Moreover they're features you don't have to test yourself, and in particular you don't have to optimize yourself.

It's a mature technology

Core Data has a mature code base whose quality is maintained through unit tests, and is used daily by millions of customers in a wide variety of applications. The framework has been highly optimized over several releases. It takes advantage of information provided in the model and runtime features not typically employed in application-level code. Moreover, in addition to providing excellent security and error-handling, it offers best memory scalability of any competing solution. Put another way: you could spend a long time carefully crafting your own solution optimized for a particular problem domain, and not gain any performance advantage over what Core Data offers for free for any application.

Xcode integration

In addition to the benefits of the framework itself, Core Data integrates well with the OS X tool chain. The model design tools allow you to create your schema graphically, quickly and easily. You can use templates in the Instruments application to measure Core Data's performance, and to debug various problems. On OS X desktop, Core Data also integrates with Interface Builder to allow you to create user interfaces from your model. These aspects help to further shorten application design, implementation, and debugging cycles.

CORE DATA

What Core Data Is Not

NOT A RDB

Having given an overview of what Core Data is and does, and why it may be useful, it is also useful to correct some common misperceptions and state what it is not.

- Core Data is not a relational database or a relational database management system (RDBMS).

  Core Data provides an infrastructure for change management and for saving objects to and retrieving them from storage. It can use SQLite as one of its persistent store types. It is not, though, in and of itself a database. (To emphasize this point: you could for example use just an in-memory store in your application. You could use Core Data for change tracking and management, but never actually save any data in a file.)

- Core Data is not a silver bullet.

  Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

- Core Data does not rely on Cocoa bindings.

  Core Data integrates well with Cocoa bindings and leverages the same technologies—and used together they can significantly reduce the amount of code you have to write—but it is possible to use Core Data without bindings. You can readily create a Core Data application without a user interface (see *Core Data Utility Tutorial*).

"real-world"?

CORE DATA

DOWNFALLS

```
NSManagedObjectModel *model = <#Get a model#>;
NSFetchRequest *requestTemplate = [[NSFetchRequest alloc] init];
NSEntityDescription *publicationEntity =
          [[model entitiesByName] objectForKey:@"Publication"];
[requestTemplate setEntity:publicationEntity];

NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:
                                  @"(mainAuthor.firstName like[cd] $FIRST_NAME) AND \
                                  (mainAuthor.lastName like[cd] $LAST_NAME) AND \
                                  (publicationDate > $DATE)"];
[requestTemplate setPredicate:predicateTemplate];
[model setFetchRequestTemplate:requestTemplate
                       forName:@"PublicationsForAuthorSinceDate"];
```

- Verbose (simple fetch)
- Versioning
- Historical misunderstanding

CORE DATA STACK IN IOS10

CORE DATA

APPLICATION FLOW

- This was tremendously simplified in iOS10
- Most criticism of CoreData it probably is outdated

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, e
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
                in a shipping application, although it may be useful during

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or di
             * The persistent store is not accessible, due to permissions or
             * The device is out of space.
             * The store could not be migrated to the current model version.
             Check the error message to determine what the actual problem wa
             */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error app
            // fatalError() causes the application to generate a crash log a
                in a shipping application, although it may be useful during
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA

APPLICATION FLOW

- Setup Core Data Stack
  - Object model
  - Persistent store
  - MOC

Old way had to set up all parts of the stack manually

```
// MARK: - Core Data stack

lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file. This code uses a directory named "mobi.uchicago._0
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional. It is a fatal error for the application not
    let modelURL = NSBundle.mainBundle().URLForResource("_016_CoreDataStack", withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implementation creates and returns a coordinator, having adde
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should not use this function in a shipping
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}()

lazy var managedObjectContext: NSManagedObjectContext = {
    // Returns the managed object context for the application (which is already bound to the persistent store coordinator for t
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()
```

CORE DATA

```
_managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:[self modelURL]];
_persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                        initWithManagedObjectModel:self.managedObjectModel];


_managedObjectContext = [[NSManagedObjectContext alloc]
                                        initWithConcurrencyType:NSMainQueueConcurrencyType];
```

```
self.managedObjectContext.persistentStoreCoordinator = self.persistentStoreCoordinator;
[self.managedObjectContext.persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                                    configuration:nil
                                                                              URL:[self storeURL]
                                                                          options:nil
                                                                            error:&error];
```

- Application Flow: Setup Core Data Stack

CORE DATA

```
let entity = NSEntityDescription.entityForName("Standard", inManagedObjectContext: context)
self.init(entity: entity!, insertIntoManagedObjectContext: context)
self.identifier = identifier
self.category = data["category"] as? String
self.stateStandard = data["stateStandard"] as? String
self.subCategory = data["subCategory"] as? String
self.uuid = data["uuid"] as? String
```

- Create/Edit entities

CORE DATA

APPLICATION FLOW

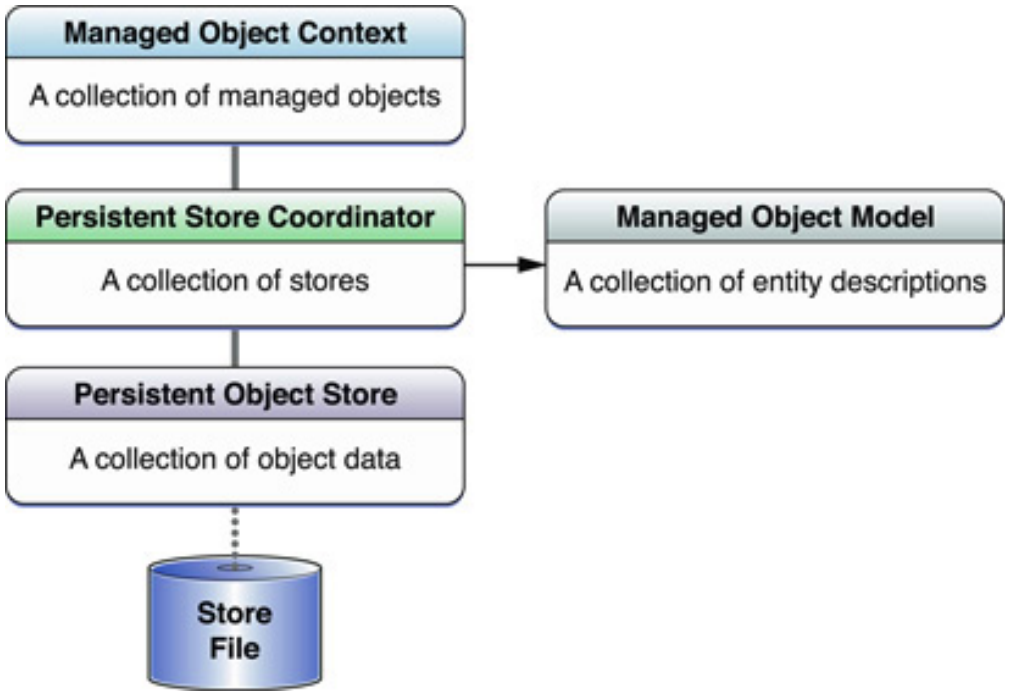
```
guard self._fetchedResultsController == nil else {
  return self._fetchedResultsController!
}
let fetchRequest = NSFetchRequest(entityName: "Lesson")
let sectionSortDescriptor = NSSortDescriptor(key: "currentIndexSection", ascending: true)
let rowSortDescriptor = NSSortDescriptor(key: "currentIndexRow", ascending: true)
fetchRequest.sortDescriptors = [sectionSortDescriptor,rowSortDescriptor]
if let cp = currentPredicate {
  fetchRequest.predicate = cp
} else {
  fetchRequest.predicate = defaultPredicate
}

let frc = NSFetchedResultsController(fetchRequest: fetchRequest, managedObjectContext: self.context, sectionNameKeyPath: "currentIndexSection", cacheName: nil)
frc.delegate = self
self._fetchedResultsController = frc
return self._fetchedResultsController!
```

- Fetch

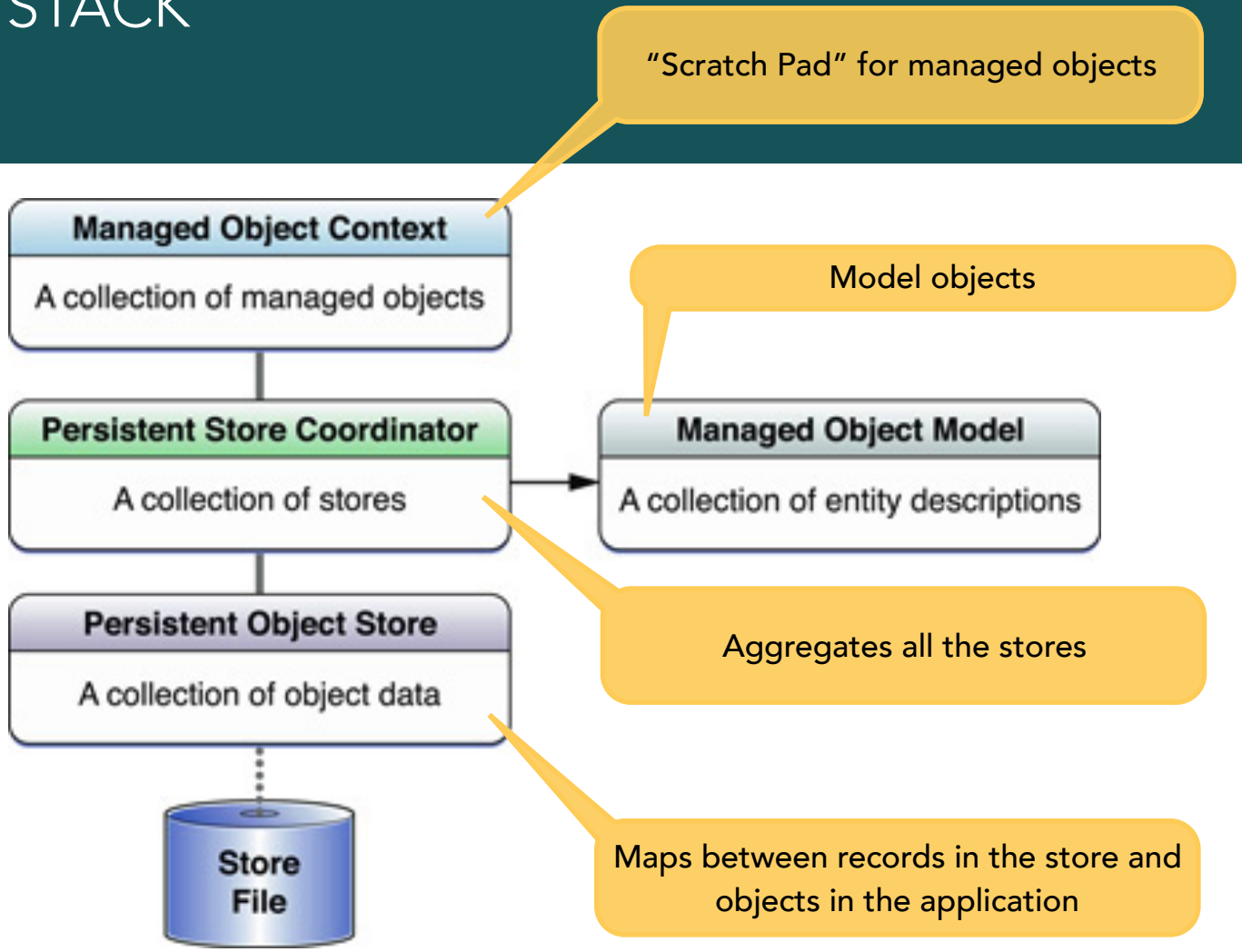
CORE DATA STACK

CORE DATA STACK

- Core Data stack
  - One or more managed object contexts
  - A single persistent store coordinator
  - Persistent store

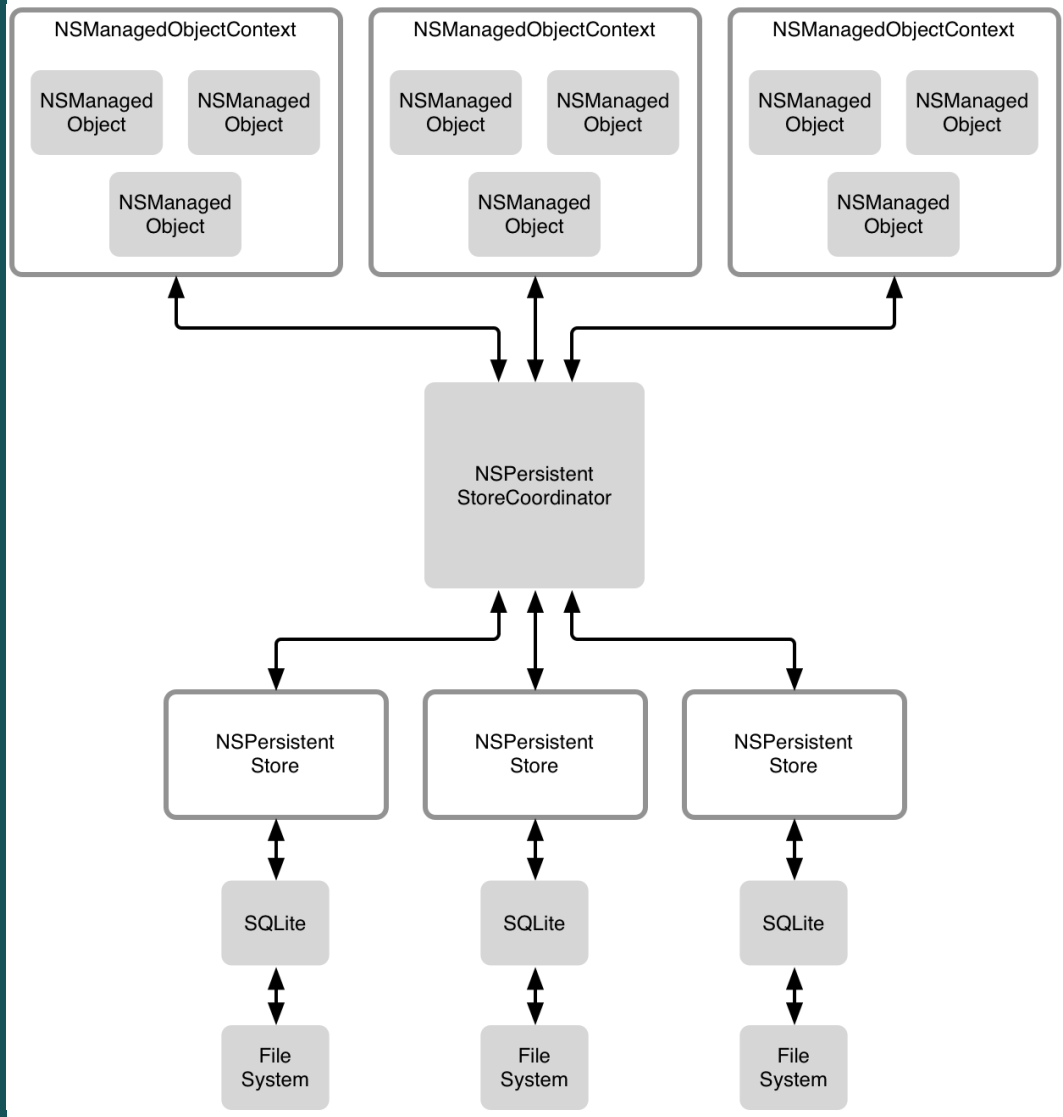


CORE DATA STACK



CORE DATA STACK

- Persistent Store Coordinator defines stack
- Can be multiple contexts, stores, etc.



CORE DATA STACK

XCODE TEMPLATE

- Xcode will create codes stubs for working the persistent store
- AppDelegate with have a property called `persistentContainer`
- These are fairly complete and will not need to be dramatically altered

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "_017_CoreDataBasics")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate. You should not use this f
                useful during development.

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or disallows writing.
             * The persistent store is not accessible, due to permissions or data protection when the device is loc
             * The device is out of space.
             * The store could not be migrated to the current model version.
             Check the error message to determine what the actual problem was.
             */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()
```

CORE DATA STACK

XCODE TEMPLATE

- Provides a function to save the current store

```
// MARK: - Core Data Saving support

func saveContext () {
    print("Saving...")
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
            dump()


        } catch {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate.
                useful during development.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

CORE DATA STACK

WORKING WITH DATA

- Keys are NSString (i.e. 'thumbnailData','date', 'name')
- The value is whatever is stored
  - Numbers and booleans are NSNumber type objects
  - “To-many” relationships are NSSet
  - Single relationships are NSManagedObjects
  - Binary data is NSData

CORE DATA STACK

WORKING WITH DATA

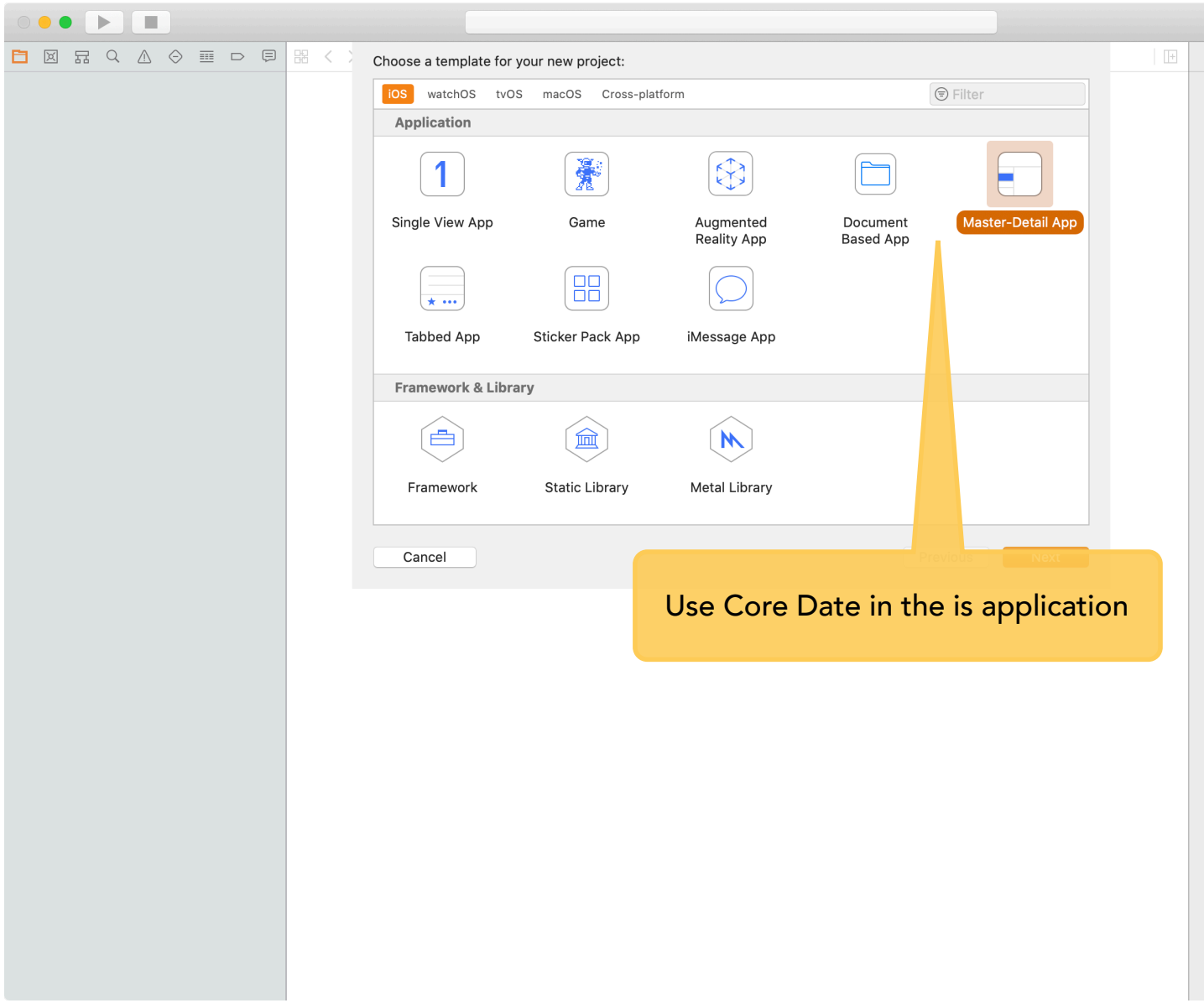
- Behind the scenes
  - SQL is generated
  - Results are fetched (lazily)
- Writes to database are only done in memory
  - Need to save them or you will loose data

```
func saveContext () {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            // Replace this implementation with code to handle the error
            // appropriately.
            // abort() causes the application to generate a crash log and
            // terminate. You should not use this function in a shipping
            // application, although it may be useful during development.
            let nserror = error as NSError
            NSLog("Unresolved error \(nserror), \(nserror.userInfo)")
            abort()
        }
    }
}
```

CREATE A CORE DATA MODEL

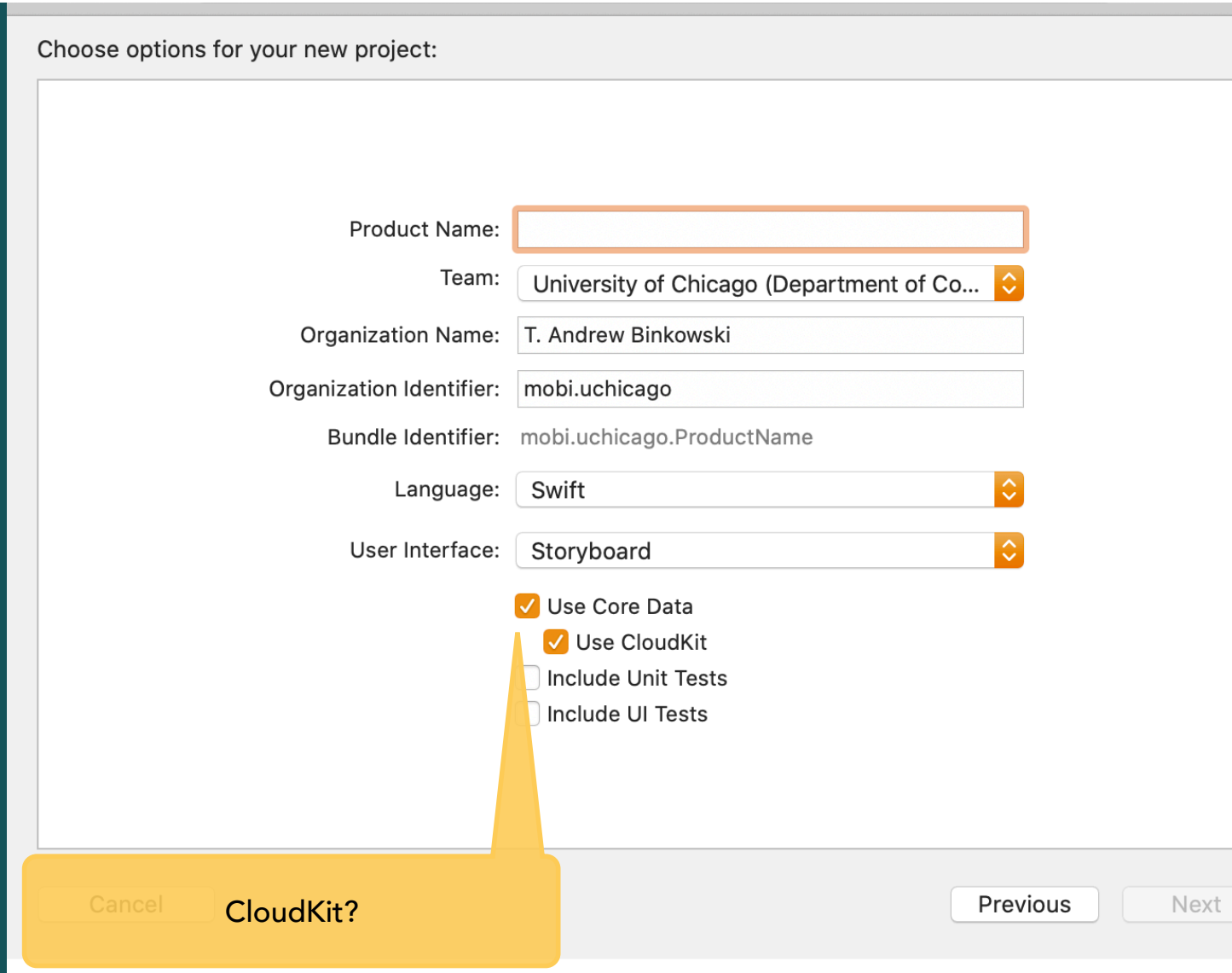
CORE DATA MODEL

- Provide a simple data model (.xcdatamodel)
- Xcode will generate the boiler plate methods



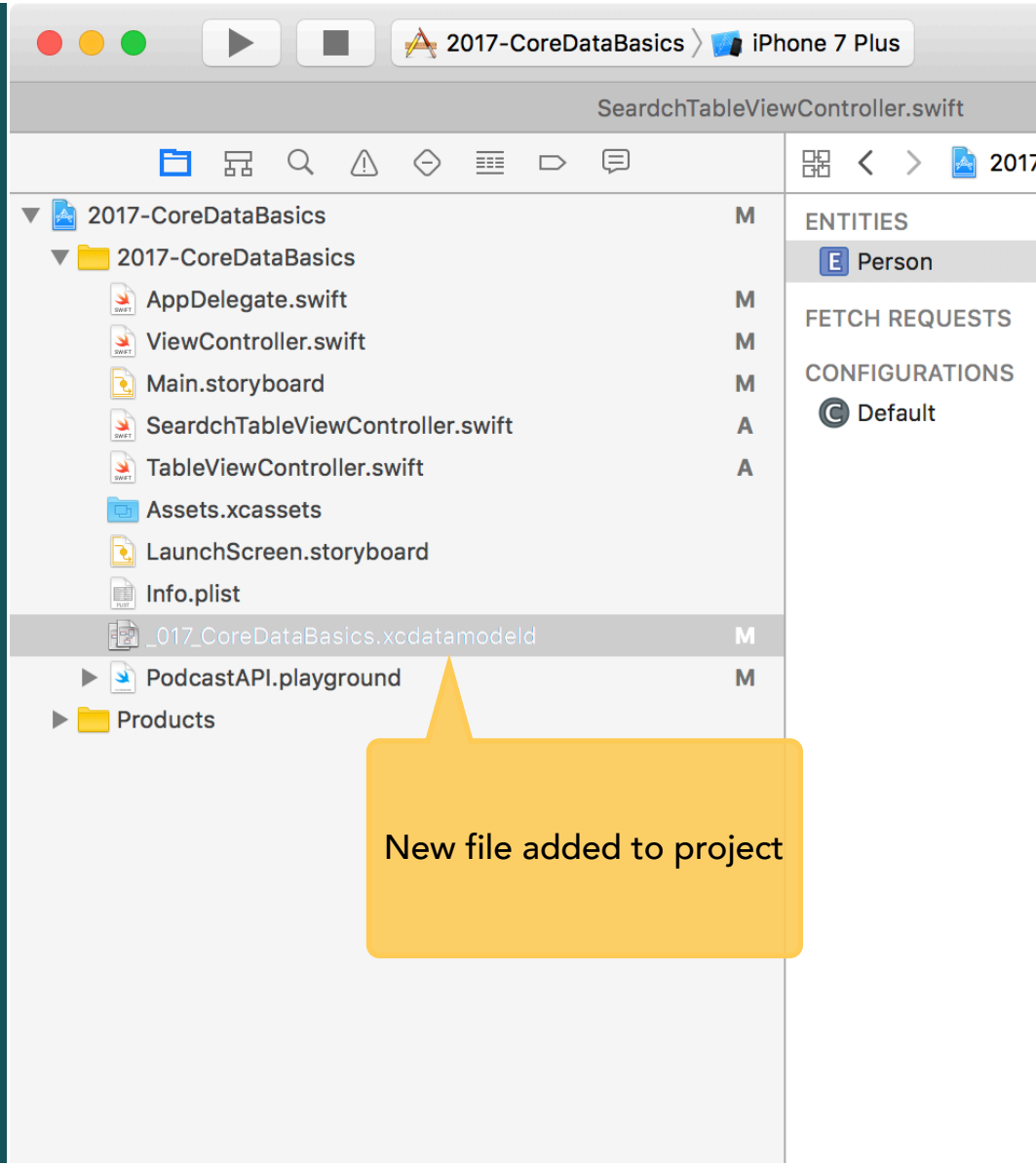
CORE DATA MODEL

- Provide a simple data model (.xcdatamodel)
- Xcode will generate the boiler plate methods



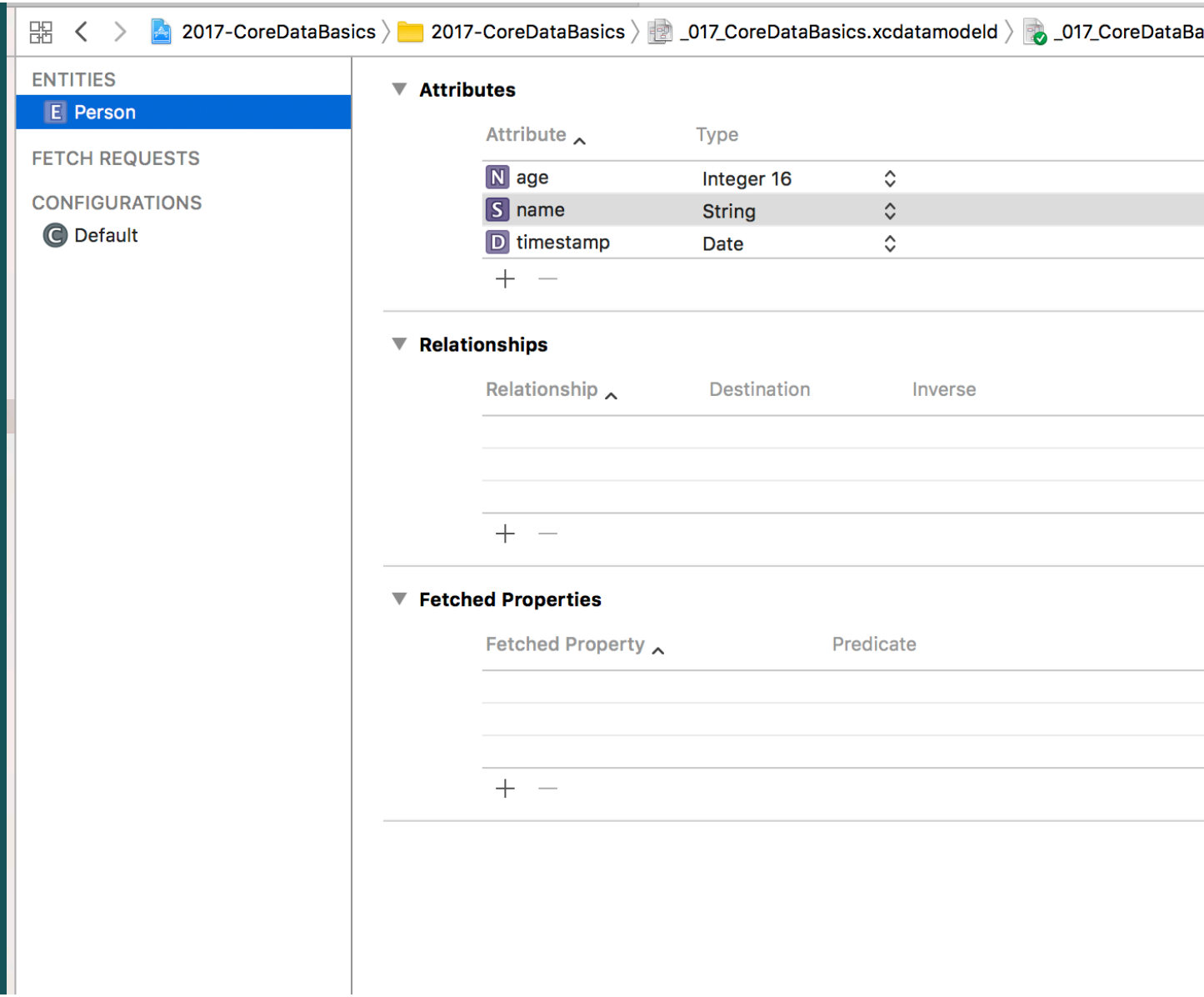
CORE DATA

- .xcdatamodeld is a directory
  - Stores data model
  - Provides built-in versioning
- Changing your data model will break your app
  - Think about the changes you may make

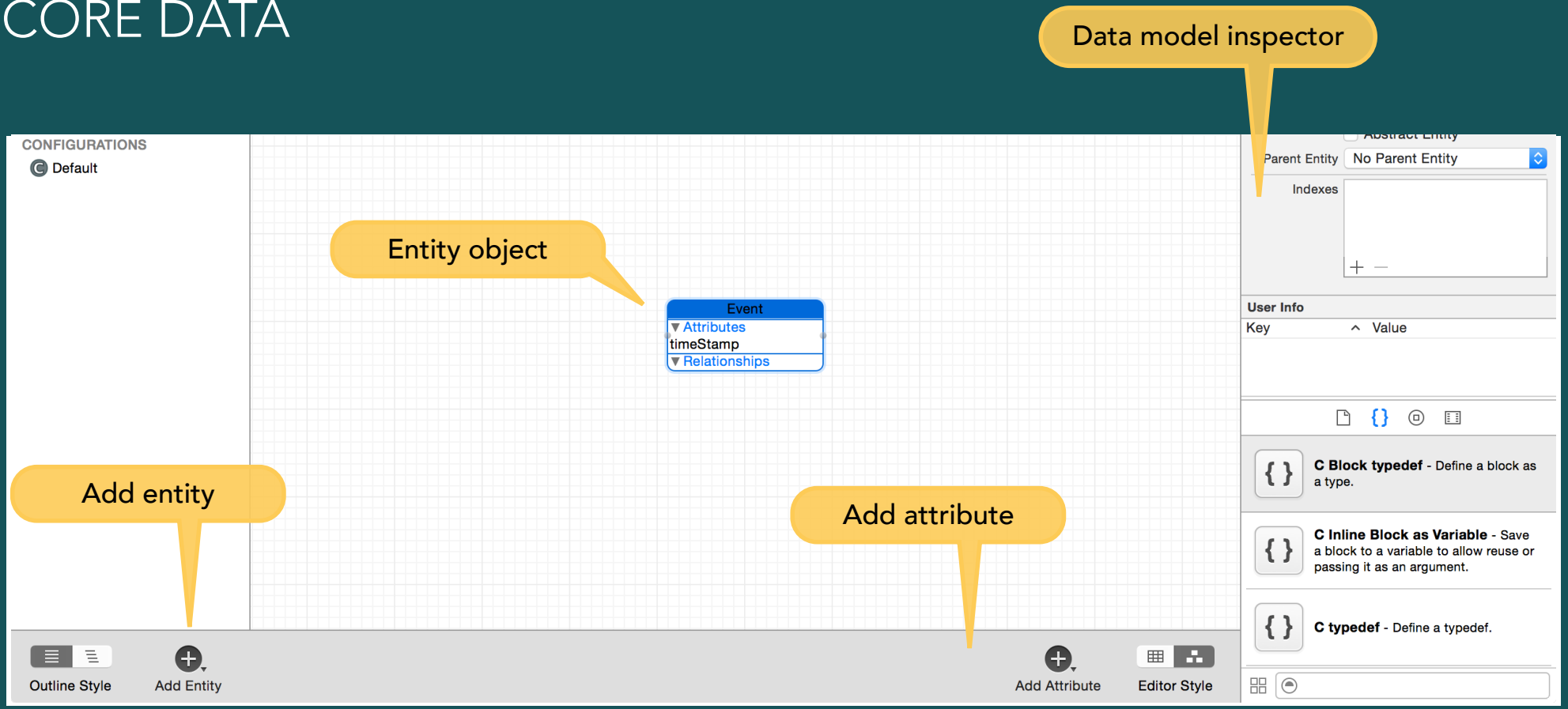


CORE DATA

- Model
  is .xcdatamodel file
- Tasks
  - Add entities
  - Define attributes
- Attributes and relationships of an entity equivalent to @properties of a class



CORE DATA

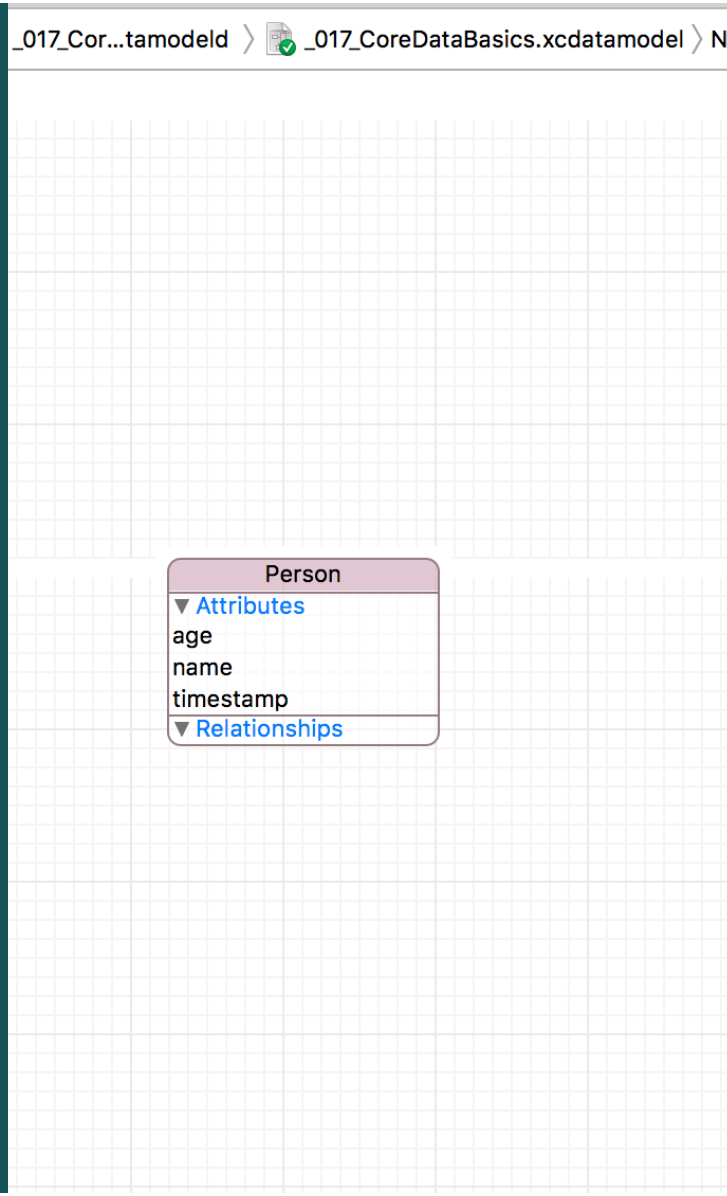


- Entity maps a database table to an object in our code

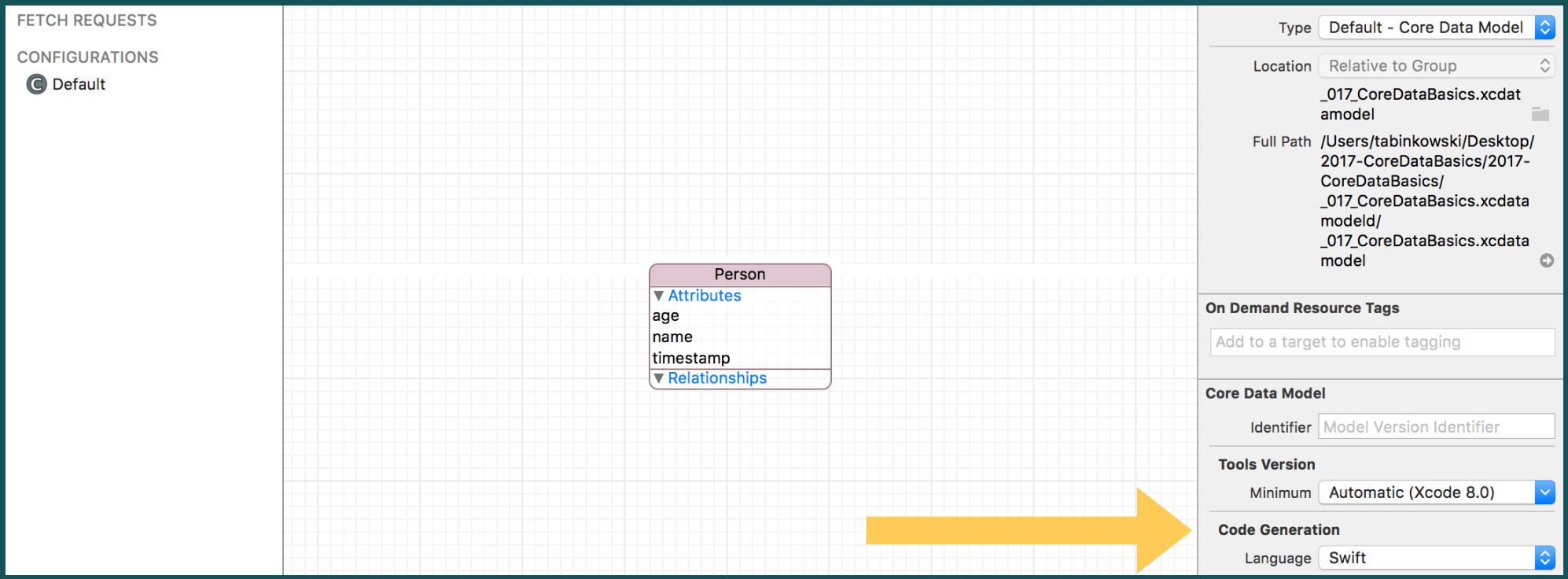
CORE DATA

- NSManagedObject
  - Subclass of NSObject
  - Access you Core Data model and data
  - The returned object of a Core Data query
  - Key-Value pairs represent the properties of an entity
  - Not thread safe (i.e. do work on main thread)

- Subclass NSManagedObject to access the properties
  - Can also get them using key-value coding with a subclass (this is uncommon)

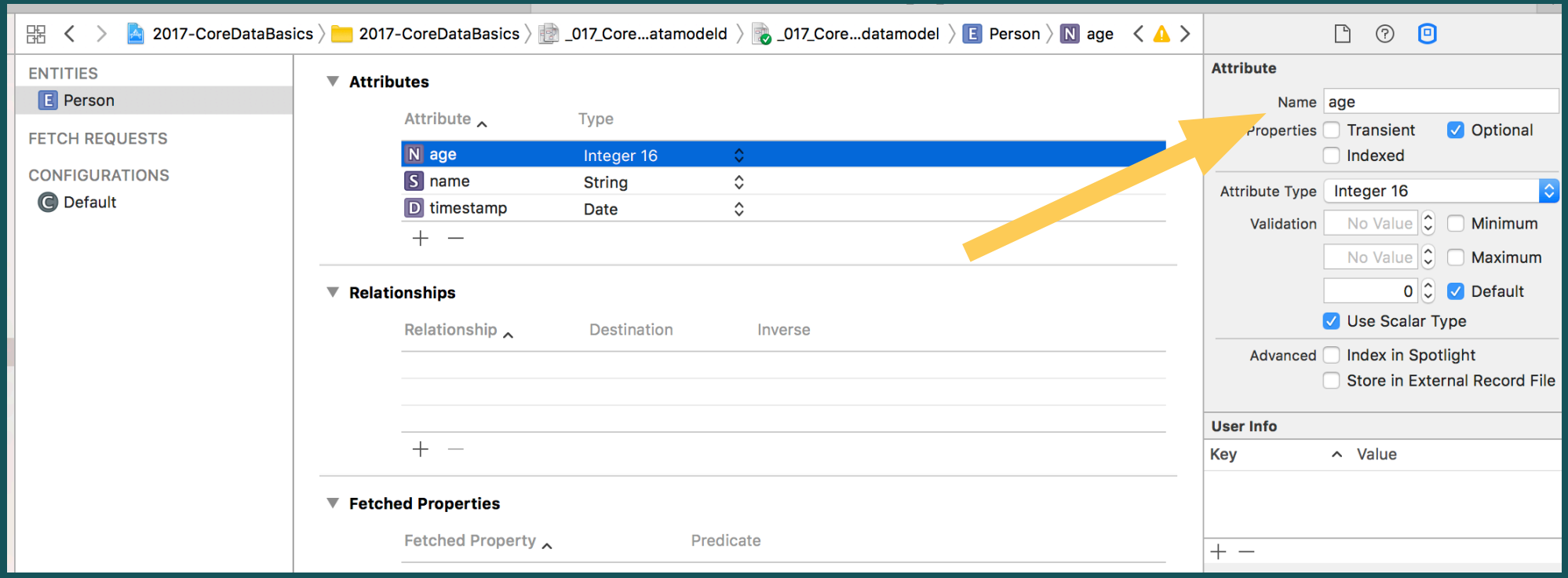


CORE DATA



- Xcode will generate subclass files of NSManagedObject for you

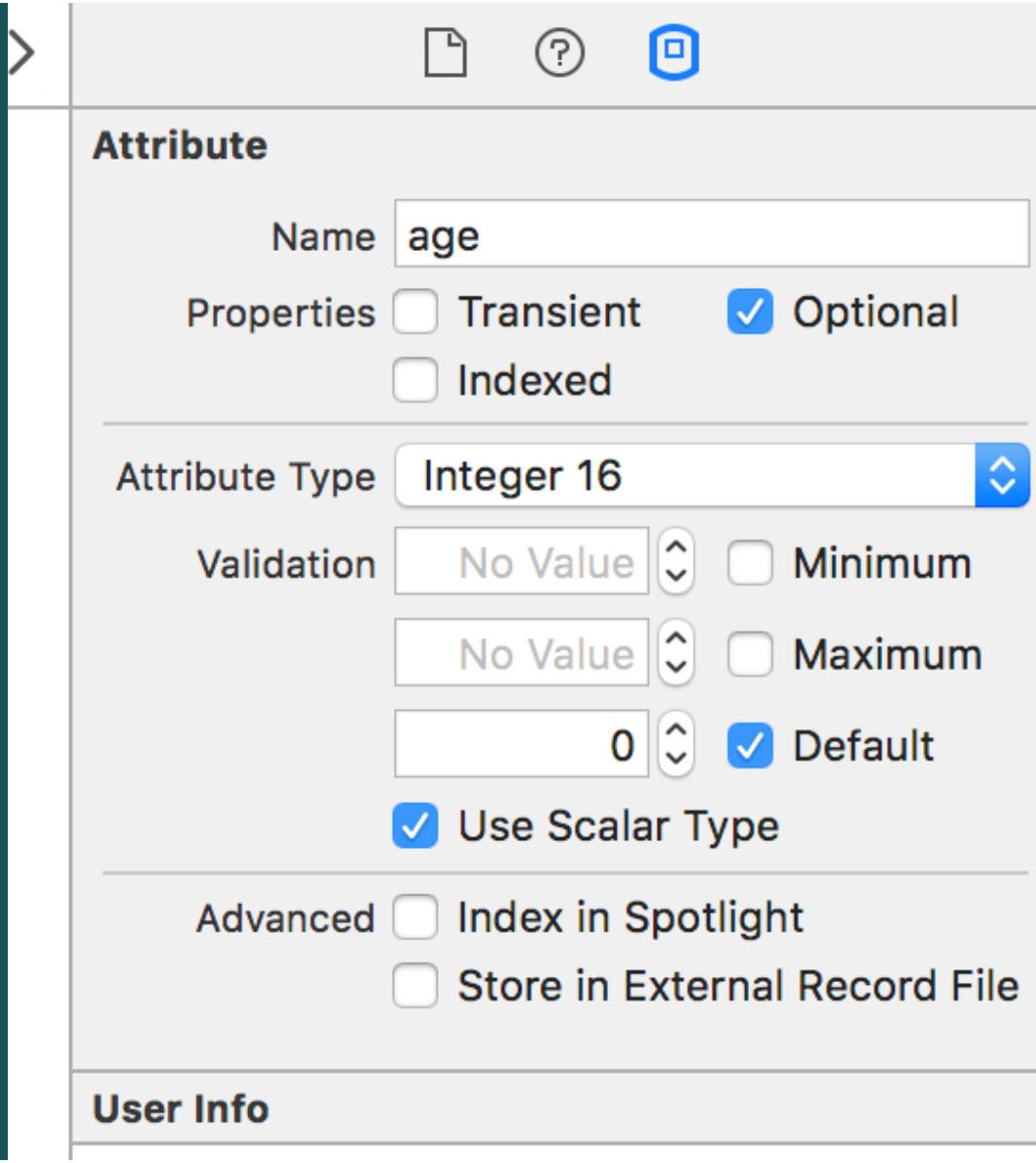
CORE DATA



- Attributes map to a database column
- Will be able to access via @property in code

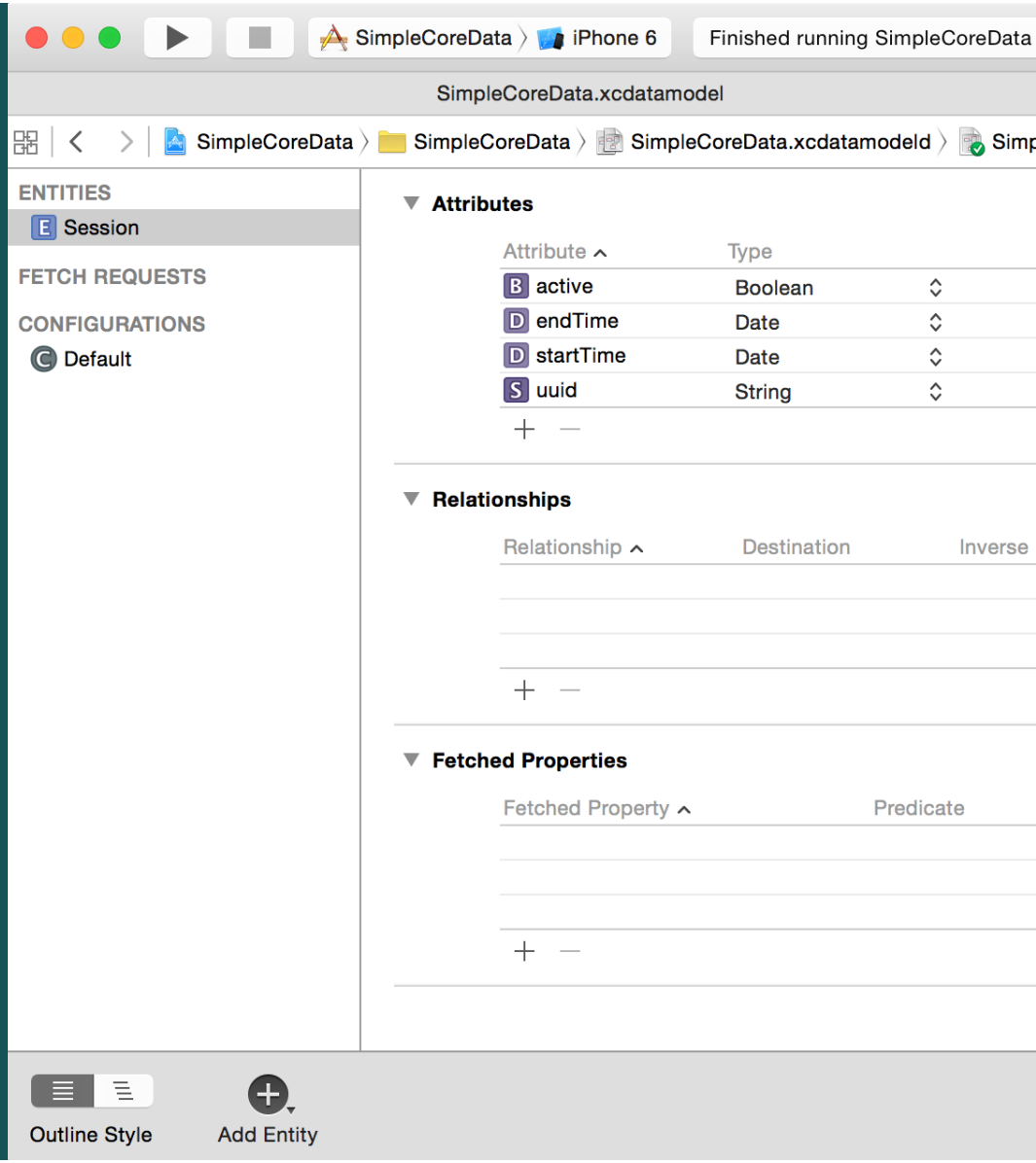
CORE DATA

- Attributes
  - Type
    - Maps to foundation object
  - Optional
  - Transient
    - Create a property that exist on object, but have no backing
  - Indexed
    - Create an index in the SQLite backend
    - Only what you will search on



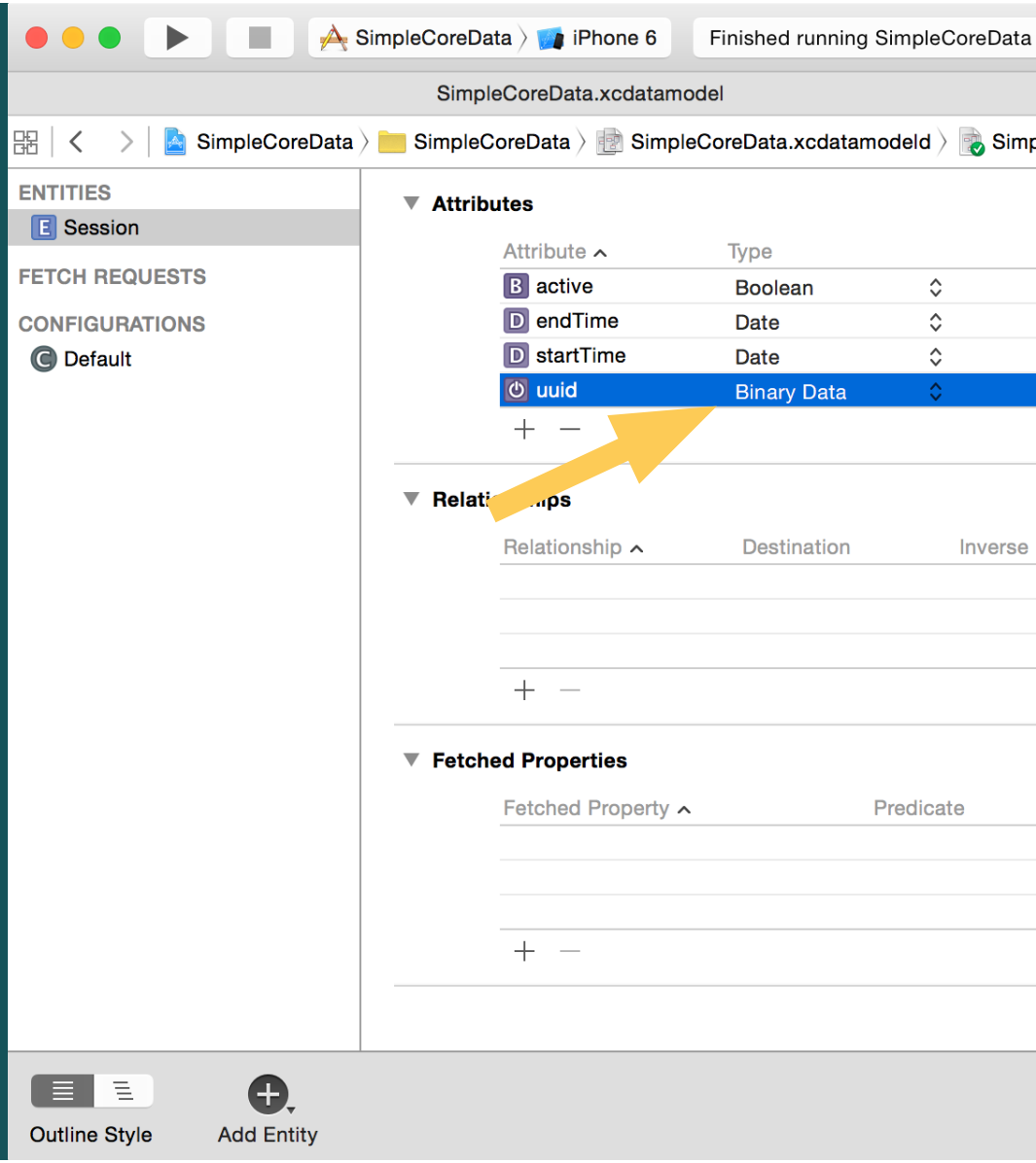
CORE DATA

- Attribute types
  - Foundation types
    - NSData for everything else
  - Binary Data for holding images
  - Transformable for nonstandard object types



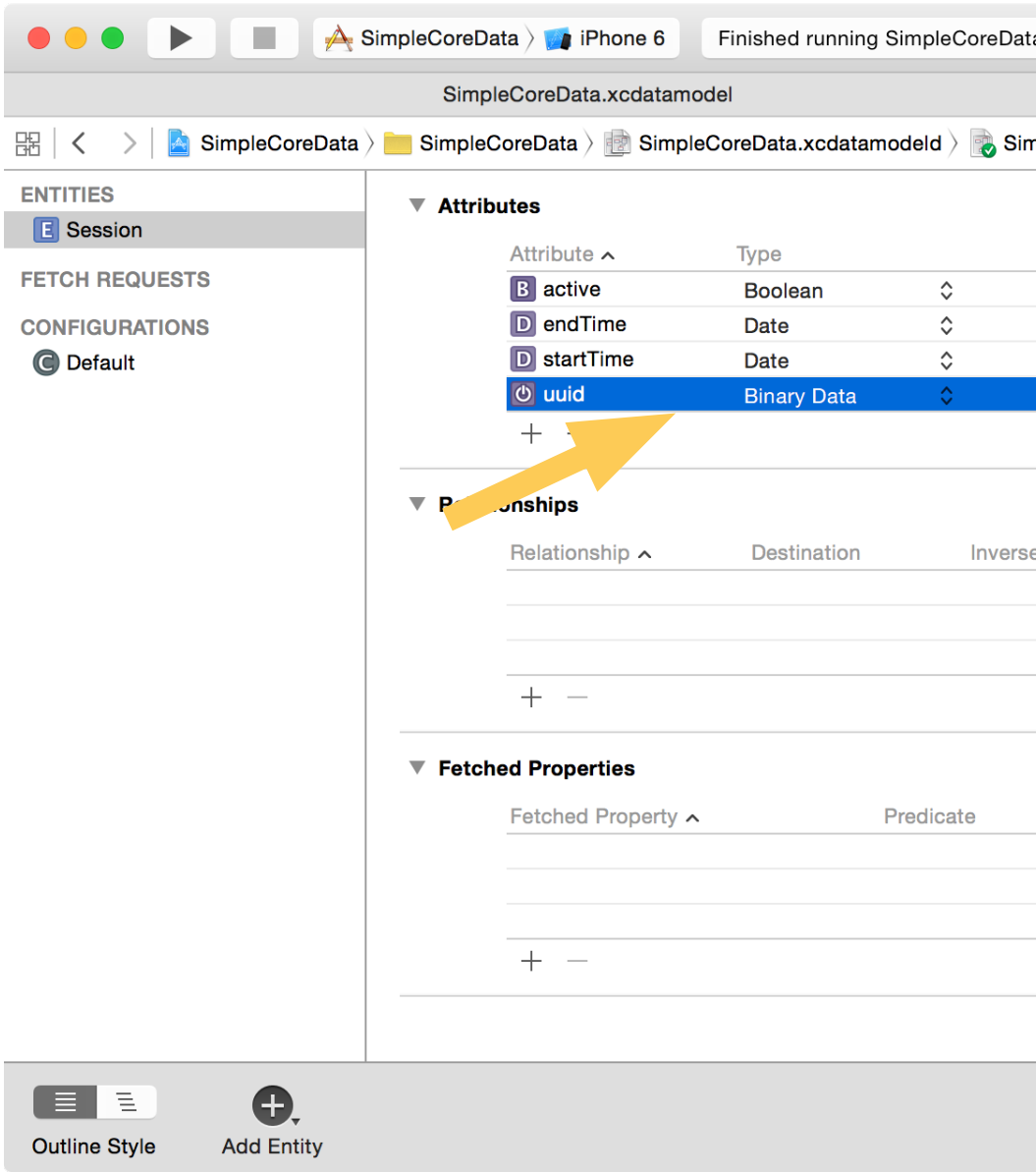
CORE DATA

- External Binary Data
  - New in iOS5
- When enabled, Core Data heuristically decides on a per-value basis
  - Save the data directly in the database
  - Store a URI to a separate file



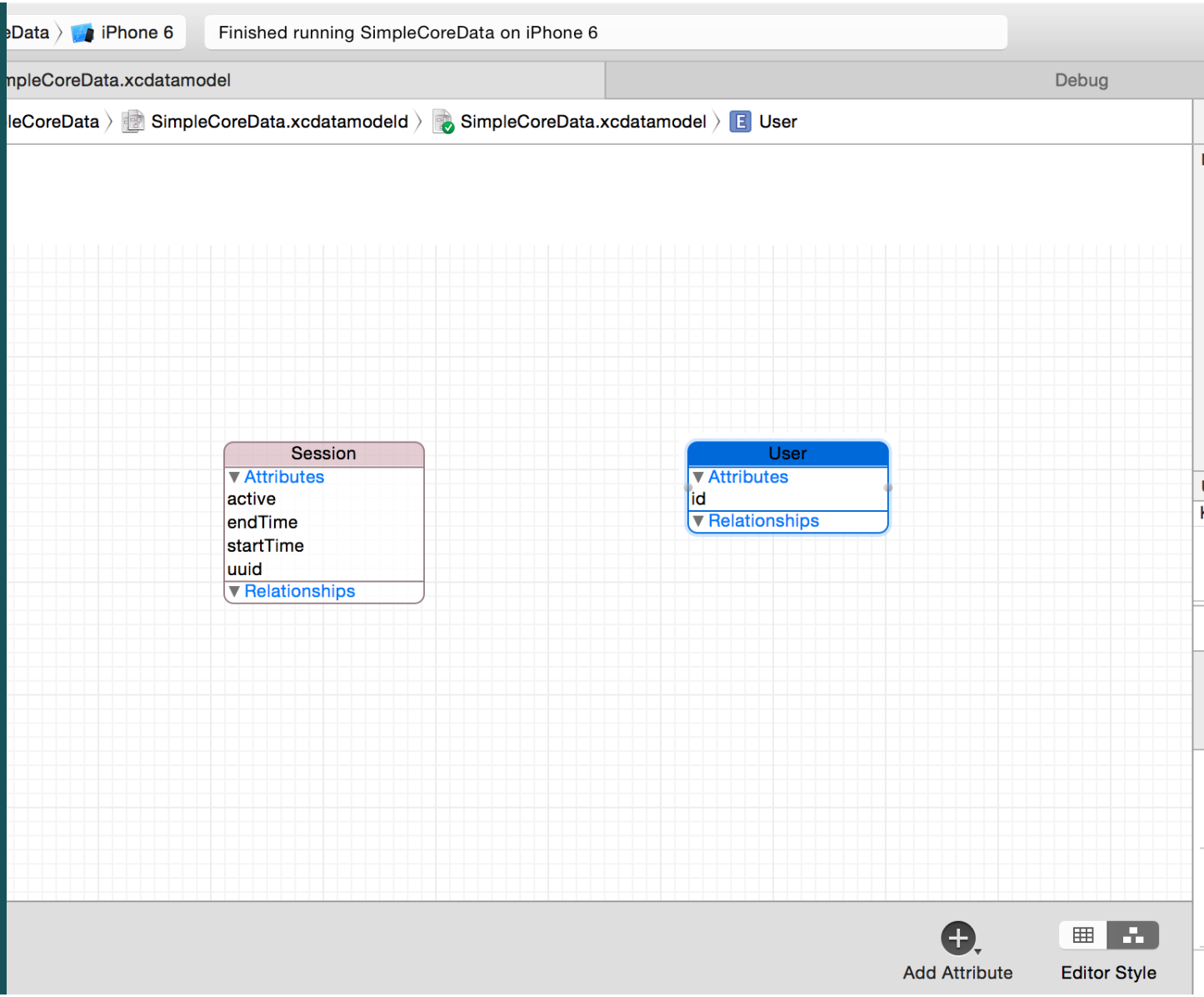
CORE DATA

- Image storage guidelines
  - < 100 kb store in entity
  - < 1 mb store in a separate entity to avoid performance issues
  - > 1 mb store on disk and reference the path storing nonstandard object types within Core Data

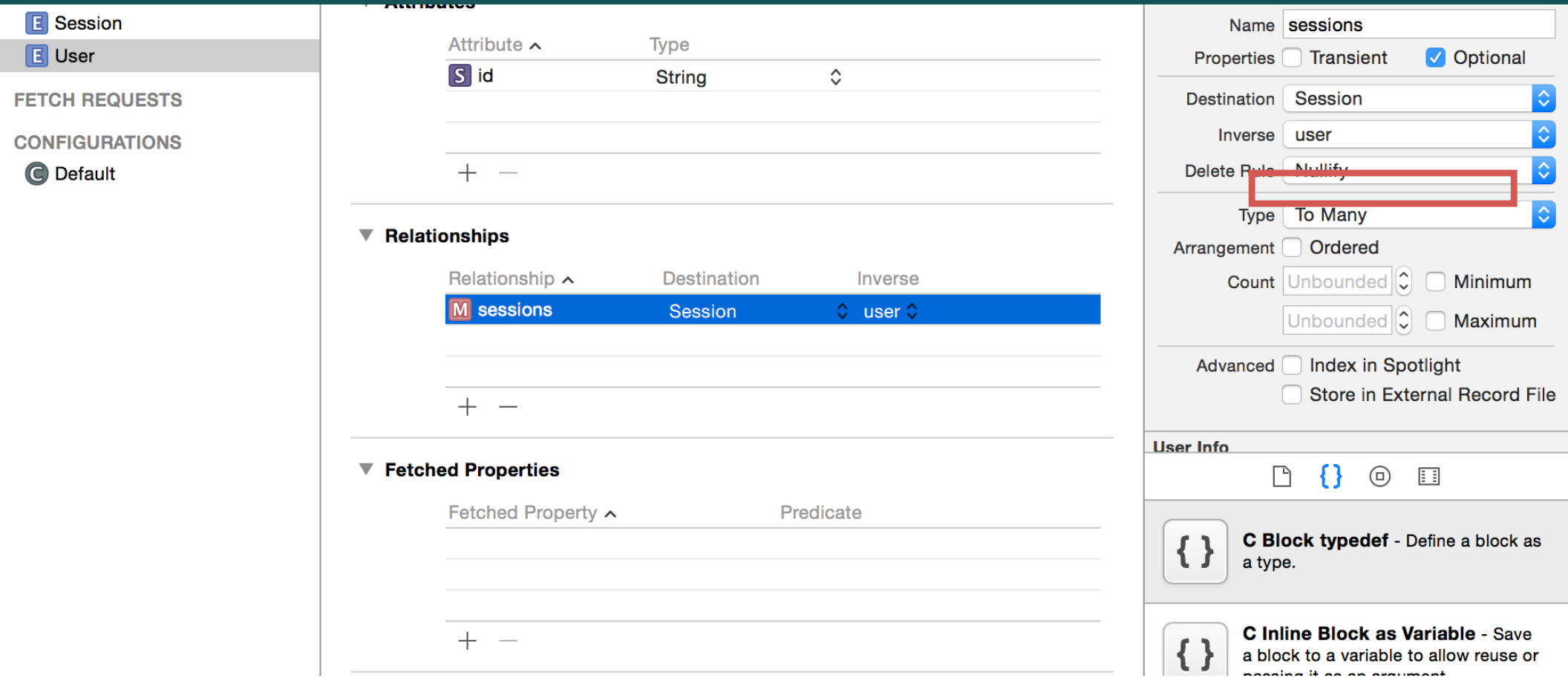


CORE DATA

- Entities can have relationships
- Example: Users and Session
  - A user can have multiple sessions
  - A session has only one user

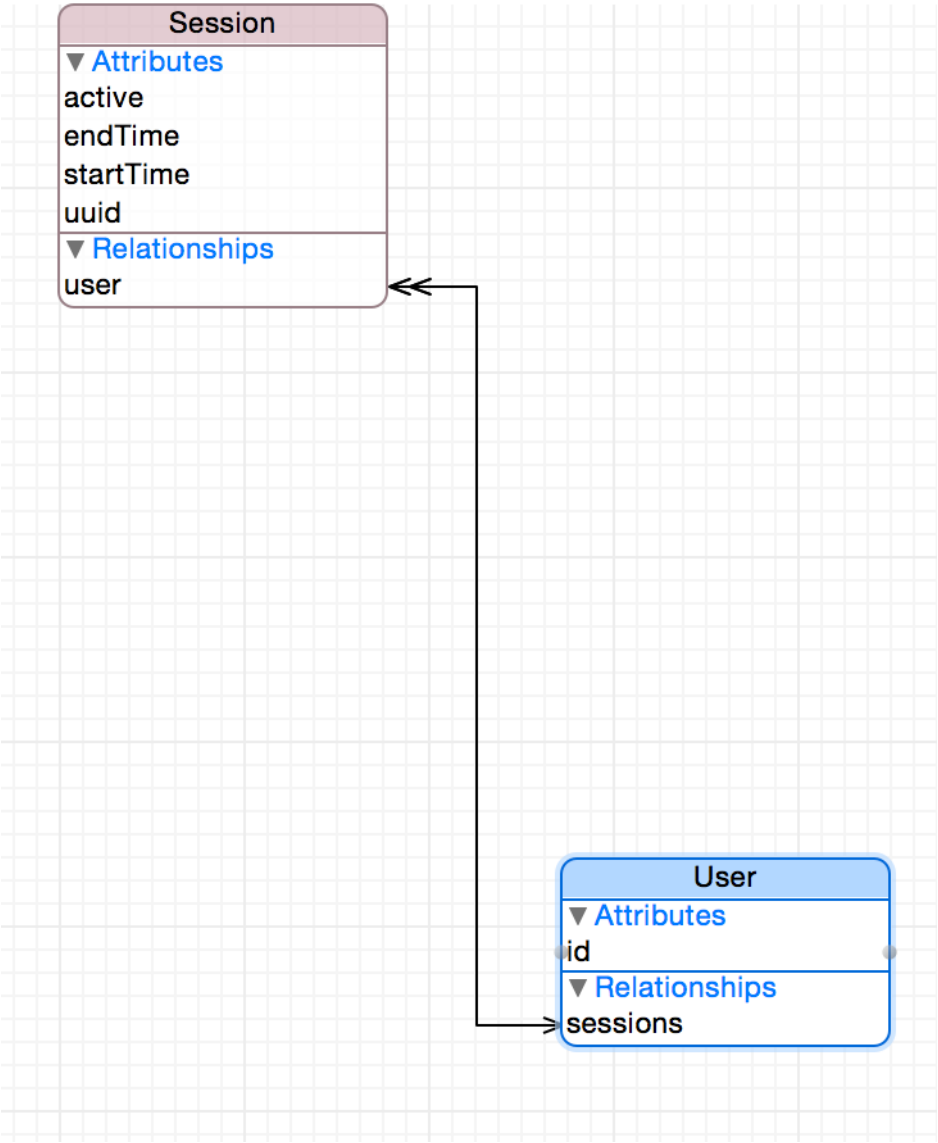


CORE DATA


Session
User
FETCH REQUESTS
CONFIGURATIONS
Default
Attribute
Type
id
String
Relationships
Relationship
Destination
Inverse
sessions
Session
user
Fetched Properties
Fetched Property
Predicate
Name
sessions
Properties
Transient
Optional
Destination
Session
Inverse
user
Delete Rule
Nullify
Type
To Many
Arrangement
Ordered
Count
Unbounded
Minimum
Unbounded
Maximum
Advanced
Index in Spotlight
Store in External Record File
User Info
C Block typedef - Define a block as a type.
C Inline Block as Variable - Save a block to a variable to allow reuse or

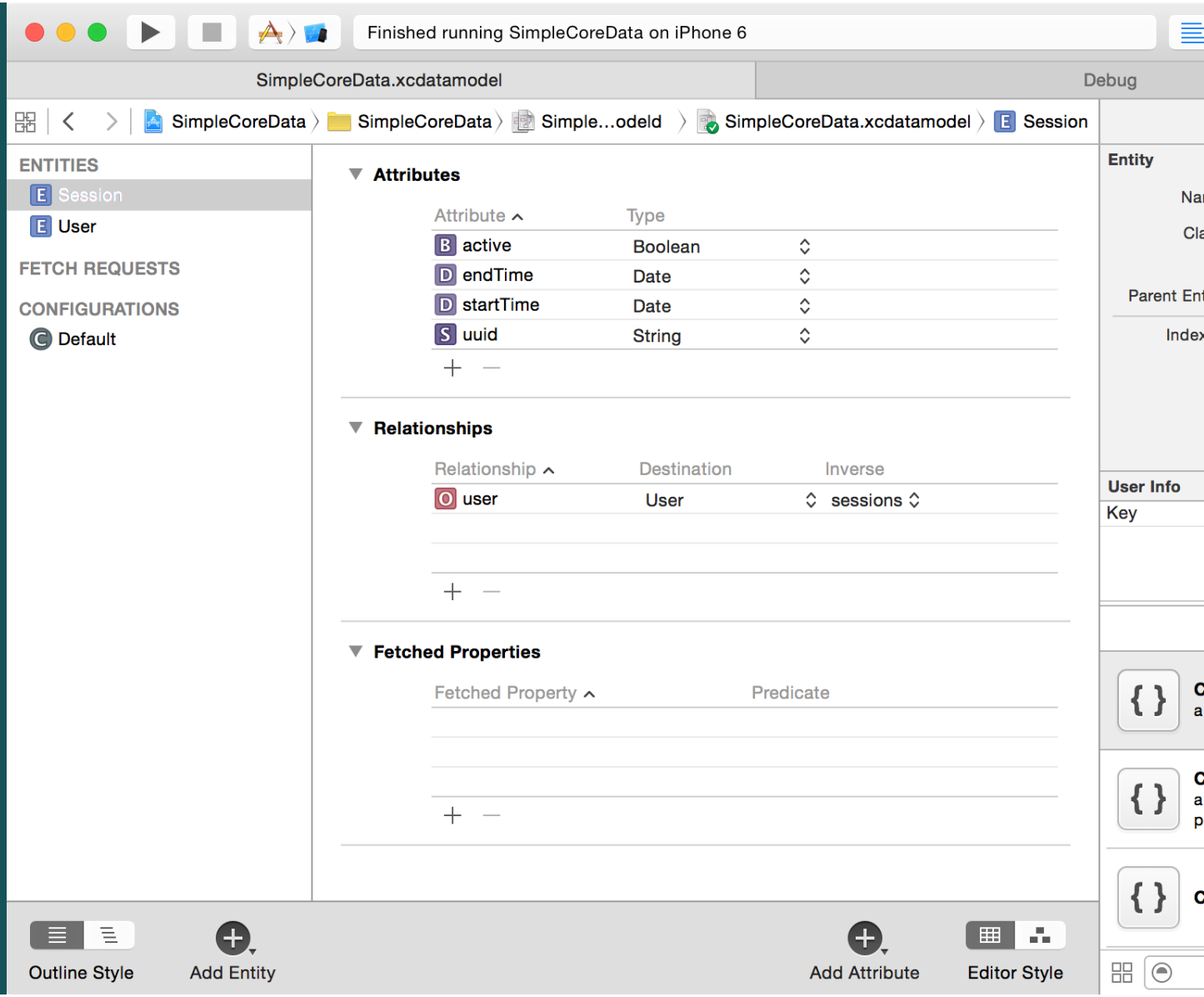
CORE DATA

- "To Many" relationship
  - Each user can have many different sessions
- Maps to an `NSSet` of `NSManagedObjects`
  - Remember that sets have no order guarantee



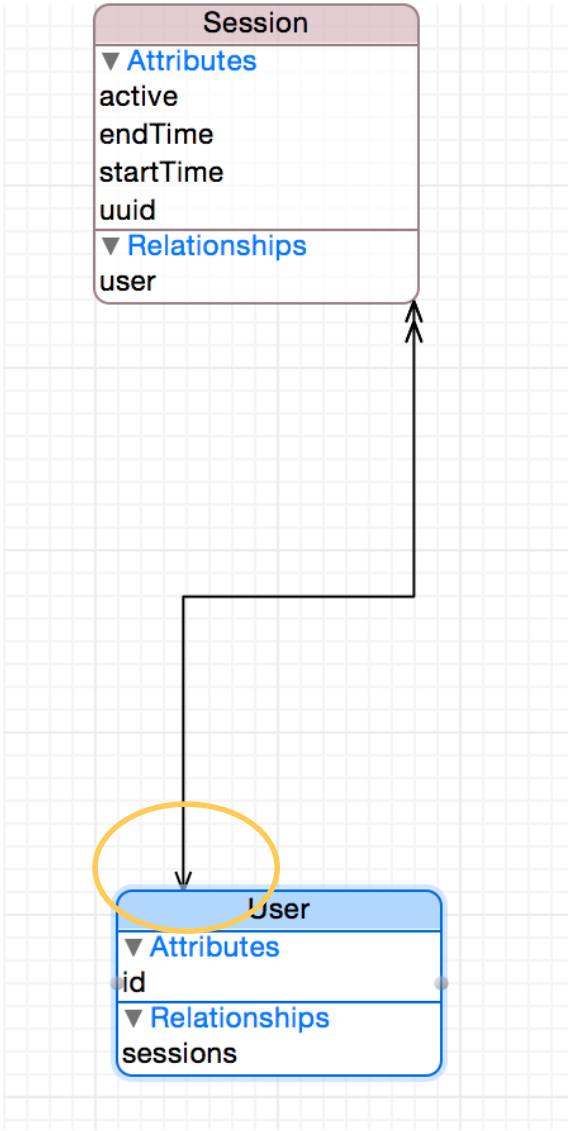
CORE DATA

- Inverse relationship from user to sessions
  - Each session can have only 1 user

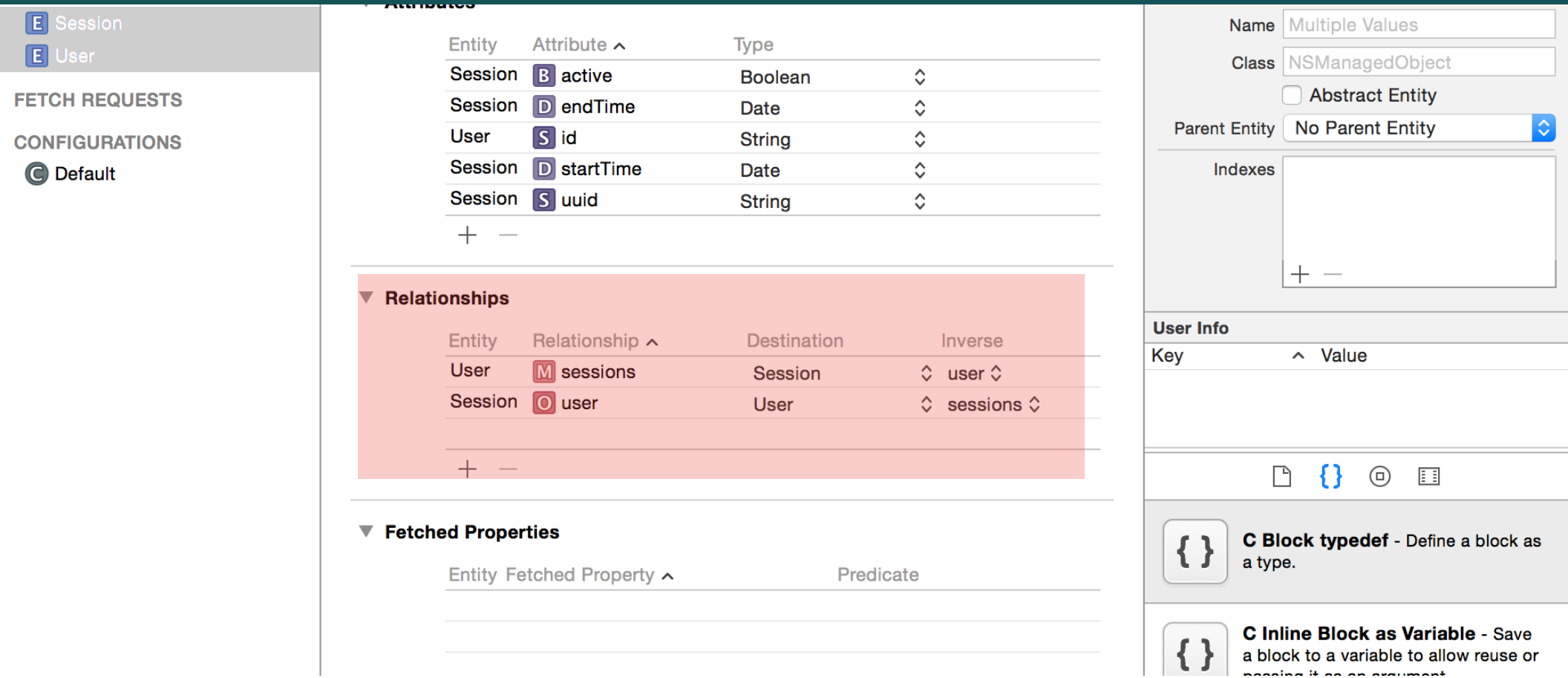


CORE DATA

- Inverse goes only one way
- Maps to a single NSManagedObject



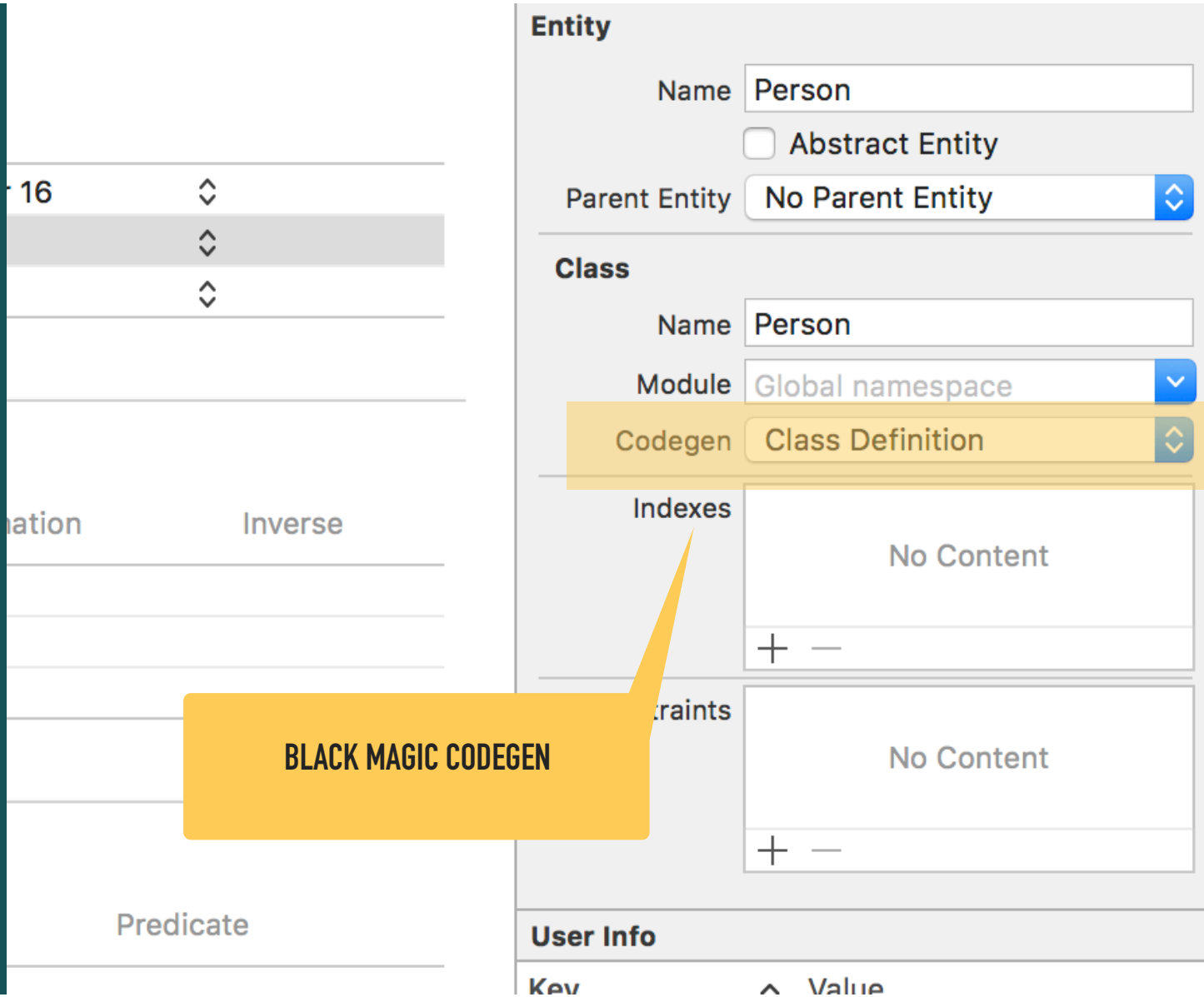
CORE DATA


E Session
E User
FETCH REQUESTS
CONFIGURATIONS
C Default
Entity Attribute Type
Session B active Boolean
Session D endTime Date
User S id String
Session D startTime Date
Session S uuid String
Relationships
Entity Relationship Destination Inverse
User M sessions Session user
Session O user User sessions
Fetched Properties
Entity Fetched Property Predicate
Name Multiple Values
Class NSManagedObject
Abstract Entity
Parent Entity No Parent Entity
Indexes
User Info
Key Value
C Block typedef - Define a block as a type.
C Inline Block as Variable - Save a block to a variable to allow reuse or

NSMANAGEDOBJECT SUBCLASS IN IOS10

NSMANAGEDOBJECT SUBCLASS IN IOS10

- Xcode 8+ `codegen` will create subclasses for you at compile time
  - You don't ever see them



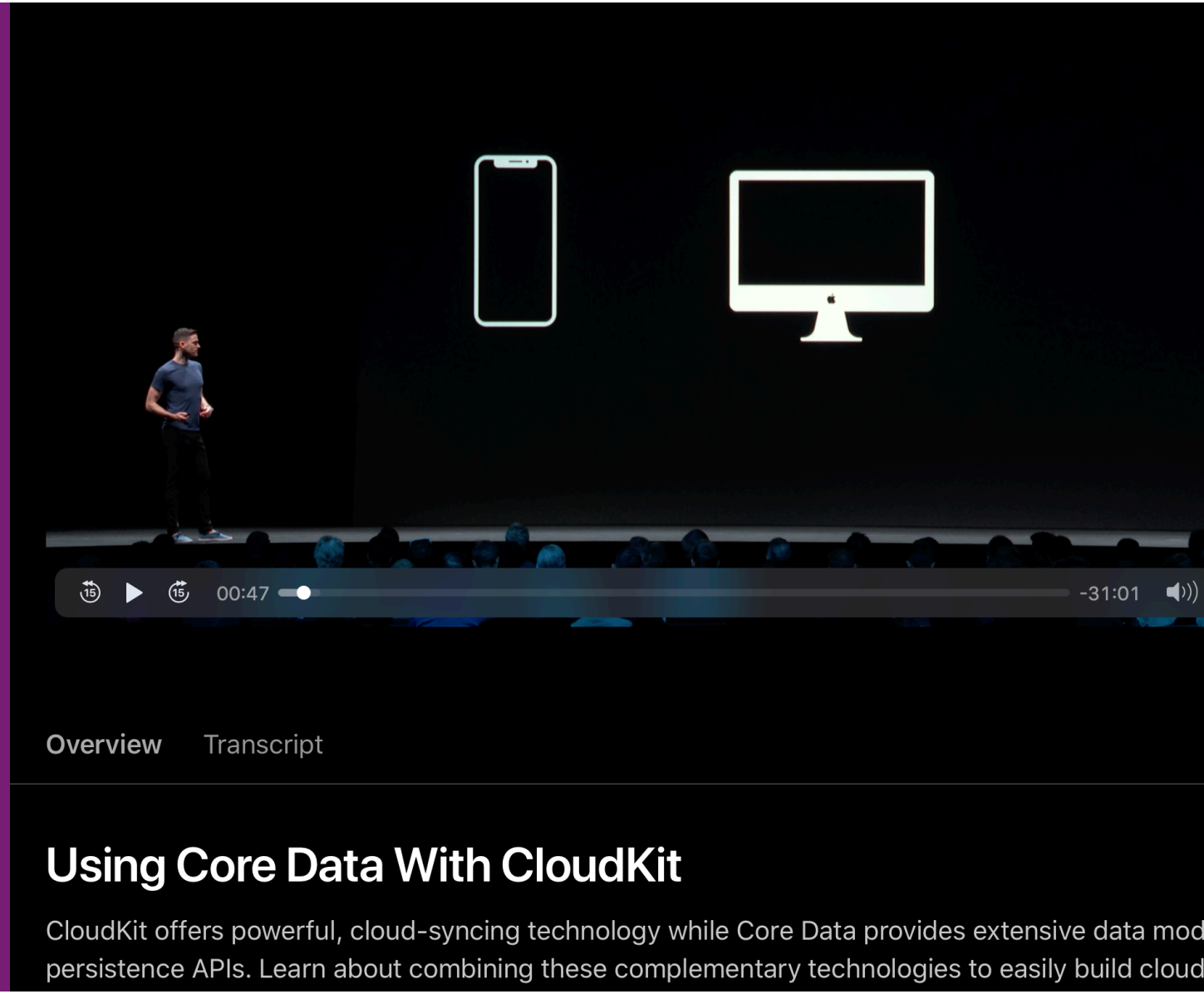
NSMANAGEDOBJECT SUBCLASS IN IOS10

- Thats it 😀

CLOUDKIT + COREDATA
TOGETHER AT LAST

CLOUDKIT + COREDATA

- WWDC 2019 the dream becomes reality



CLOUDKIT + COREDATA

- WWDC 2019 the dream becomes reality

Setting Up Core Data with CloudKit

Set up the classes and capabilities that sync your store to CloudKit.

Overview

To sync your Core Data store to CloudKit, you enable the CloudKit capability for your app. You also set up the Core Data stack with a persistent container that is capable of managing one or more local persistent stores that are backed by a CloudKit private database.

Configure a New Xcode Project

When you create a new project, you specify whether you want to add support for Core Data with CloudKit directly from the project setup interface. The resulting project instantiates an `NSPersistentCloudKitContainer` in your app's delegate. Once you enable CloudKit in your project, you use this container to manage one or more local stores that are backed with a CloudKit database.

1. Choose File > New > Project to create a new project.
2. Select a project template to use as the starting point for your project, and click Next.
3. Select the Use Core Data and Use CloudKit checkboxes.

CLOUDKIT + COREDATA

- Replicates data model in CloudKit
- Schedule notifications to sync data

Article

Setting Up Core Data wit

Set up the classes and capabilities that sync y

Overview

To sync your Core Data store to CloudKit, you enable the also set up the Core Data stack with a persistent contain more local persistent stores that are backed by a CloudK

Configure a New Xcode Project

When you create a new project, you specify whether you CloudKit directly from the project setup interface. The re `NSPersistentCloudKitContainer` in your app's de project, you use this container to manage one or more lo CloudKit database.

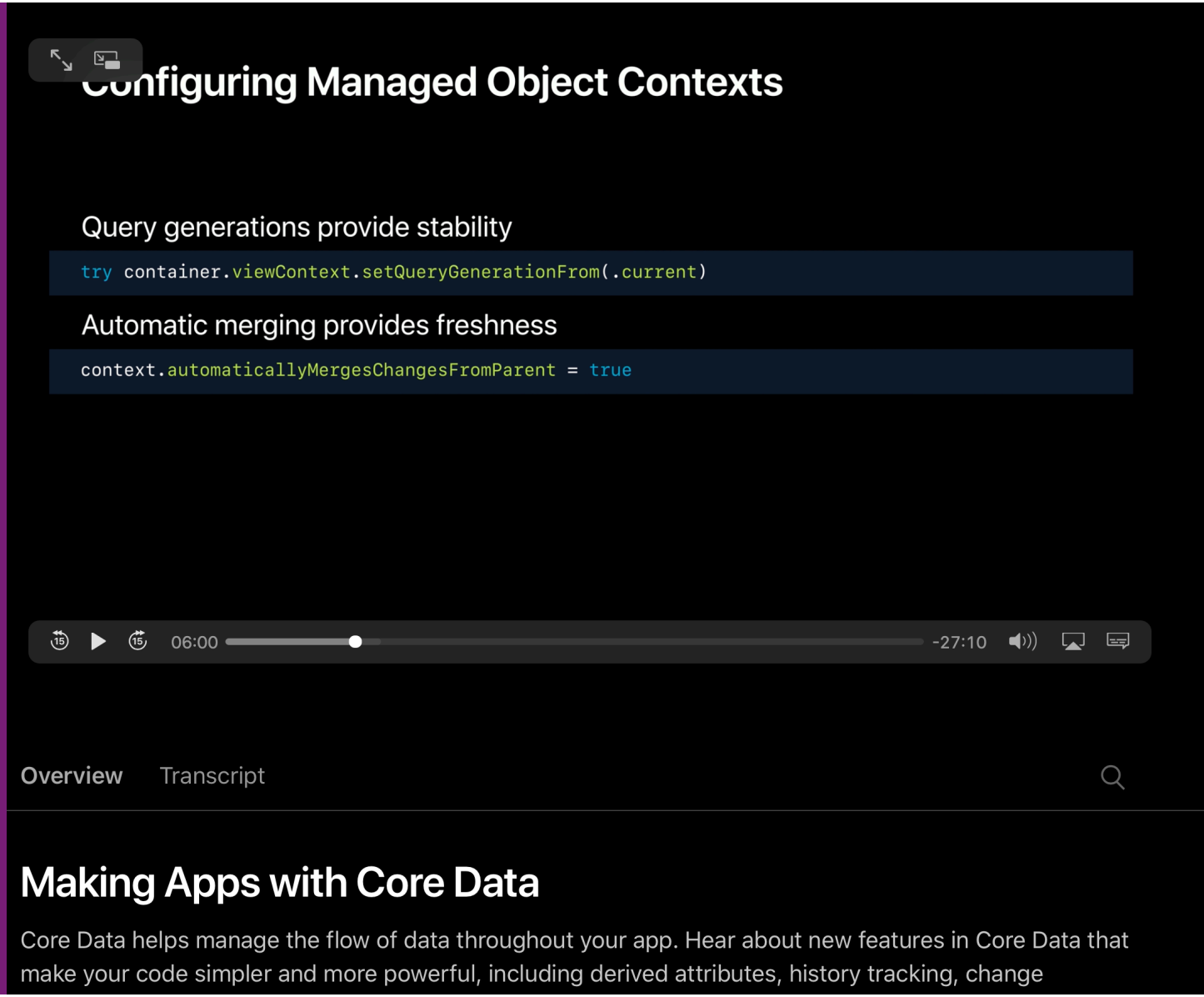
1. Choose File > New > Project to create a new projec
2. Select a project template to use as the starting poin
3. Select the Use Core Data and Use CloudKit checkb
4. Enter any other project details and click Next.

CLOUDKIT + COREDATA

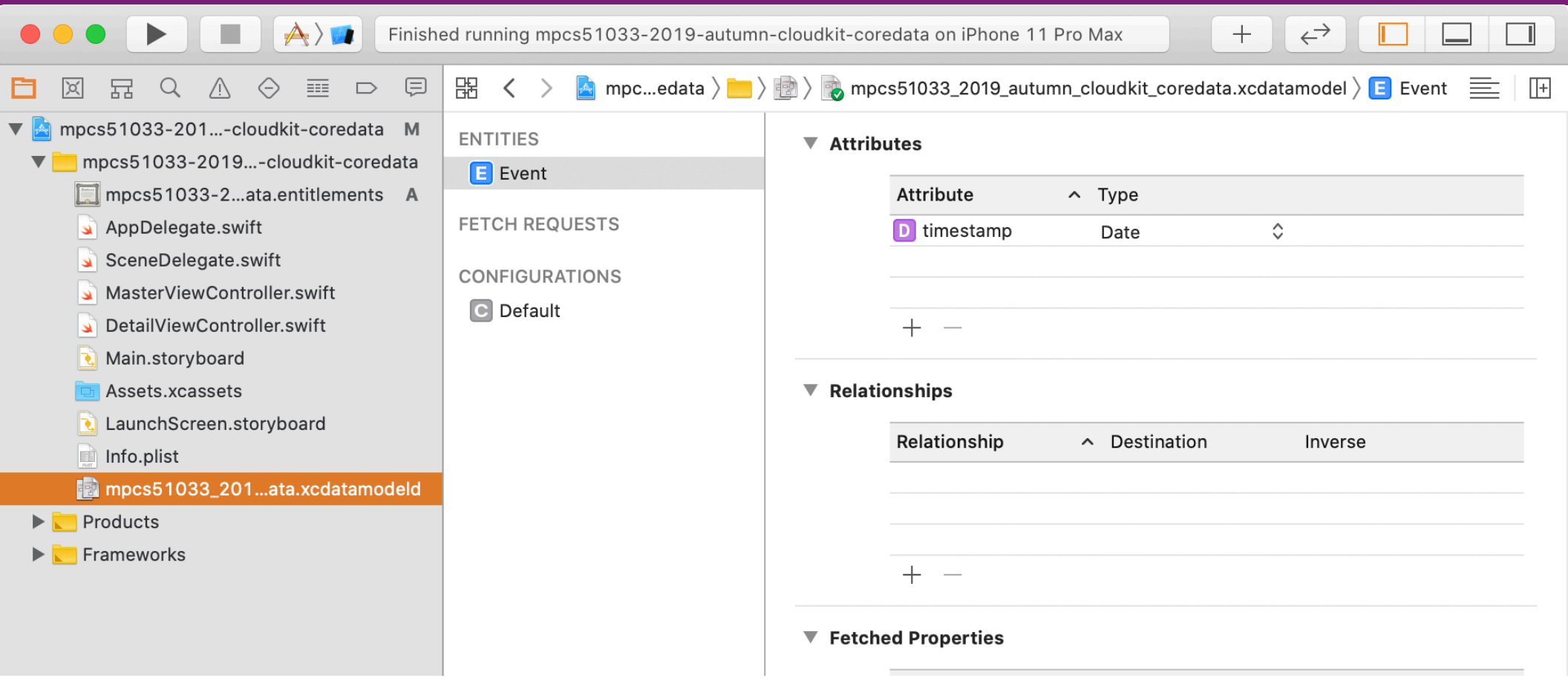
	Core Data	CloudKit
Objects	NSManagedObject	CKRecord
Models	NSManagedObjectModel	Schema
Stores	NSPersistentStore	CKRecordZone / CKDatabase

CLOUDKIT + COREDATA

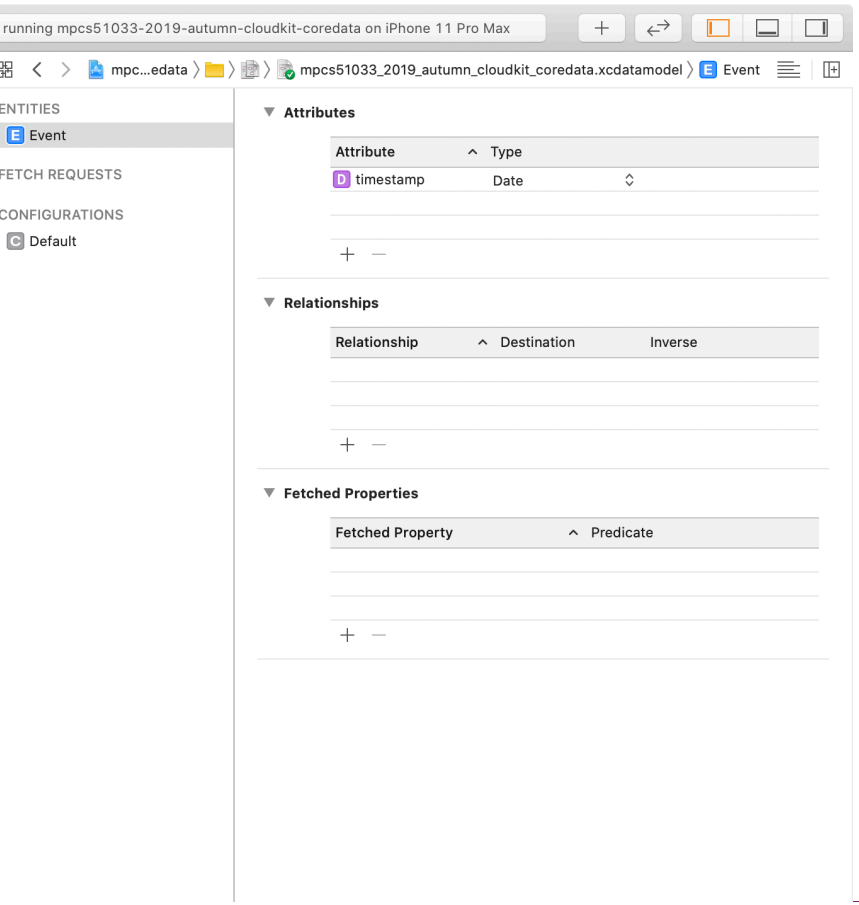
- WWDC 2019 Making App With Core Dat
  - https://developer.apple.com/videos/play/wwdc2019/230

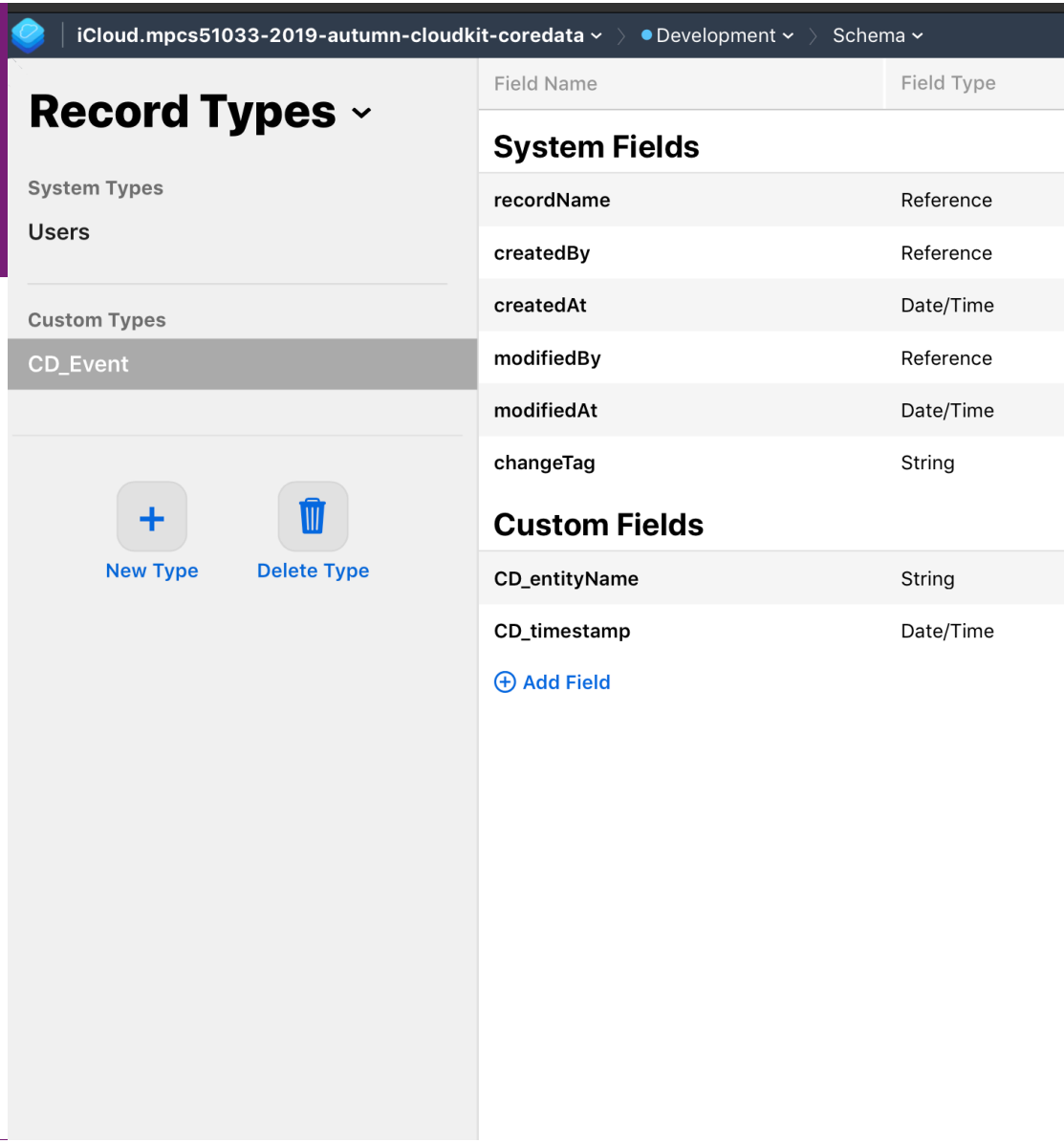


CLOUDKIT + COREDATA

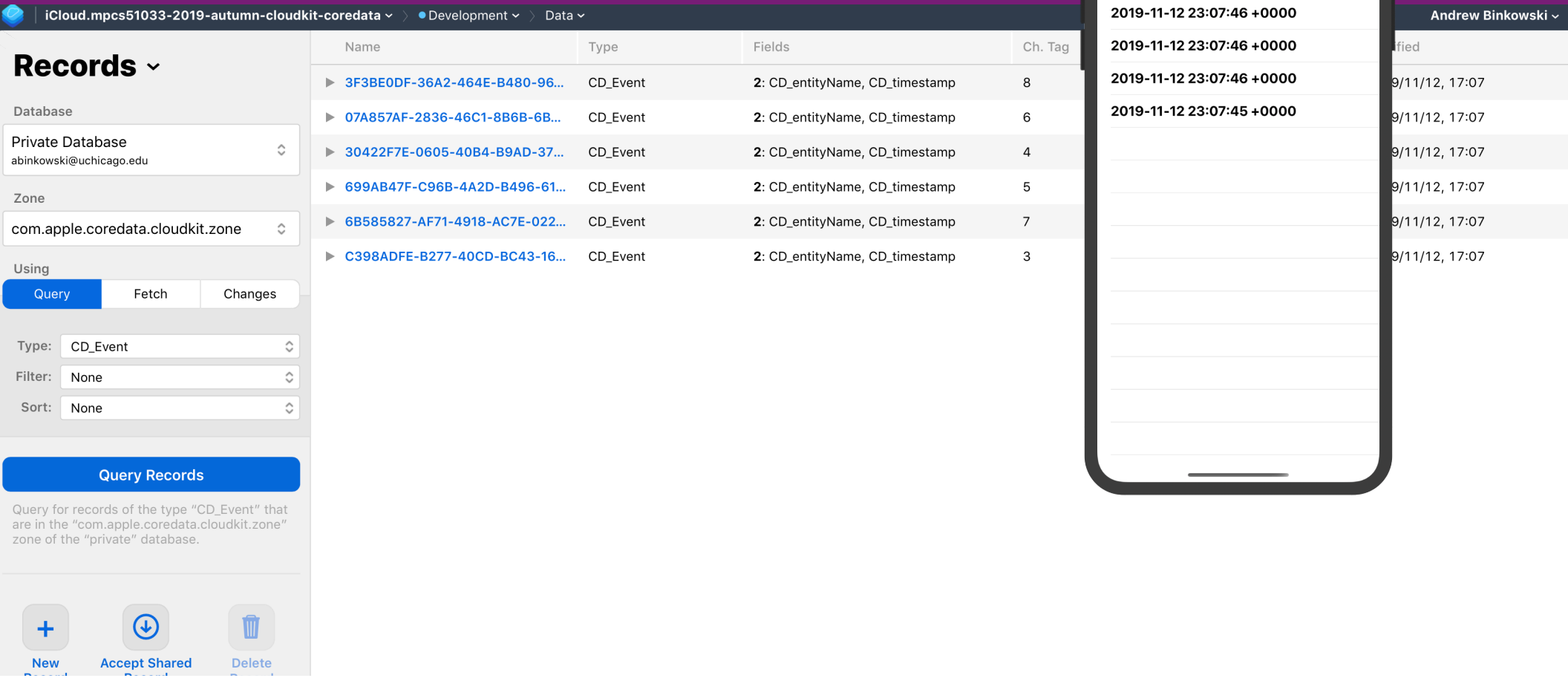


CLOUDKIT + COREDATA





CLOUDKIT + COREDATA



CLOUDKIT + COREDATA

CLOUD

```
// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentCloudKitCo
    /*
     The persistent container for the applicati  This implementation
     creates and returns a container, having  aded the store for the
     application to it. This property is o ional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentCloudKitContainer(name:
        "mpcs51033_2019_autumn_cloudkit_coredata")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropriately.
            // fatalError() causes the application to generate a crash log and terminate.
                You should not use this function in a shipping application, although it may
                be useful during development.
```

CLOUDKIT + COREDATA

- Leave it alone
- Create another store to self manage other data

CLOUDKIT + COREDATA

- Reactions
  - Great when it works
  - Terrible when it doesn't

Article

Setting Up Core Data wit

Set up the classes and capabilities that sync y

Overview

To sync your Core Data store to CloudKit, you enable the also set up the Core Data stack with a persistent contain more local persistent stores that are backed by a CloudK

Configure a New Xcode Project

When you create a new project, you specify whether you CloudKit directly from the project setup interface. The re `NSPersistentCloudKitContainer` in your app's de project, you use this container to manage one or more lo CloudKit database.

1. Choose File > New > Project to create a new projec
2. Select a project template to use as the starting poin
3. Select the Use Core Data and Use CloudKit checkb
4. Enter any other project details and click Next.

THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2019 • SESSION 7

BACKENDS FOR MOBILE APPLICATIONS