iOS DeCal

lecture 6

Firebase

cs198-001 : fall 2017

announcements

- 3 slip days for snapchat clone part 1 and part 2

  - office hours tonight - Soda 341A

- custom app proposal released

- next week's lab - help with setting up snapchat clone part 2

- final presentation day not finalized. Either Thursday or Friday of dead week

announcements

- if a lab is due Wednesday, it should be completed *before* that Wednesday's lab

- next week's lecture - mid semester review

Overview : Today's Lecture

Sync vs. Async

Recap Closures

Intro to Firebase and BaaS

Managing Users

Saving and Retrieving Data

File Storage

Sync vs. Async Tasks

The problem with network requests

Fact #1: Network requests are *slow.*

The problem with network requests

Fact #1: Network requests are *slow.*

Fact #2: Users hate waiting.

The problem with network requests

Fact #1: Network requests are *slow.*

Fact #2: Users hate waiting.

Fact #3: Users are mean.

The problem with network requests

Fact #1: Network requests are *slow.*

Fact #2: Users hate waiting.

Fact #3: Users are mean.

- We have almost no control over the time it takes to make a request to a server and wait for its response (esp. with poor internet conn.).
- Our goal is to minimize the latency that the user actually sees at any point.
  - Users should never have to sit on a frozen screen.

Synchronous Tasks

Blocks a process until the task is complete

Pros:

- Guarantee that we get results before going on to the next task.
- Somewhat easier implementation (don't have to worry about thread management).

Cons:

- User has to wait for task to finish before being able to do anything else.
  - USERS HATE WAITING!!!

Synchronous Tasks

Blocks a process until the task is complete

Pros:
- Guarantee that we get results before going on to the next task.
- Somewhat easier implementation (don't have to worry about thread management).

Cons:
- User has to wait for task to finish before being able to do anything else.
  - USERS HATE WAITING!!! (& are mean)

Synchronous Tasks: Example

```
func retrieveData {
    let query = PFQuery(classname: "cats")
    let cats = query.findObjects() //
        synchronous call
    for cat in cats {
        // do something
    }
}
```

Asynchronous Tasks

Run out of order, in parallel with the main thread so that code can continue to execute while waiting.

- iOS apps perform network requests in the background
  - Example: loading a TableView and refreshing it once data is returned.
- Introduces a new challenge:
  - What if the next line of code after the network request is evaluated before the request finishes?

Closures Revisited

Closures: self-contained blocks of functionality that can be passed around in your code.

This means we can pass functions around as parameters to other functions!

Why might this be useful for solving our async task problem?

Using Closures as Completion Handlers

Suppose we made an asynchronous network request and wanted to trigger an action only **after** we knew the request had completed.

```swift
func retrieveData {
    let params = [1, 2, 3]
    makeRequest(params: params, completion:
        { (data) in
        if let data = data {
            // do something with data
        }
    })
    // continue rest of code here
}
```

Implementing functions with completion handlers

We usually look at functions with completion handlers as "black boxes" - we assume they do the heavy lifting, and we just tell them what to do at the end.

- What are they doing behind the scenes?

```swift
func makeRequest(params: [Int], completion:
    @escaping (Data?) -> Void) {
    // make some API call
    // get data back from call
    if success {
        completion(data)
    } else {
        completion(nil)
    }
}
```

Firebase

How is data usually stored?

Option 1 – Remotely: Make requests to server and let it do the dirty work of saving/retrieving from a database.

Option 2 – Locally: Use something like SQLite or CoreData independently from a server but with more tedious work in terms of actually managing the database.

Using BaaS tools

Backend as a Service tools provide backend cloud storage support to mobile developers through simple API calls.

- Abstracts away the complexities of database implementation
- No need to write any server-side code
- Many offer a lot of additional tools that simple MySQL/SQLite databases don't support

Using BaaS tools

Backend as a Service tools provide backend cloud storage support to mobile developers through simple API calls.

- Abstracts away the complexities of database implementation
- No need to write any server-side code
- Many offer a lot of additional tools that simple MySQL/SQLite databases don't support
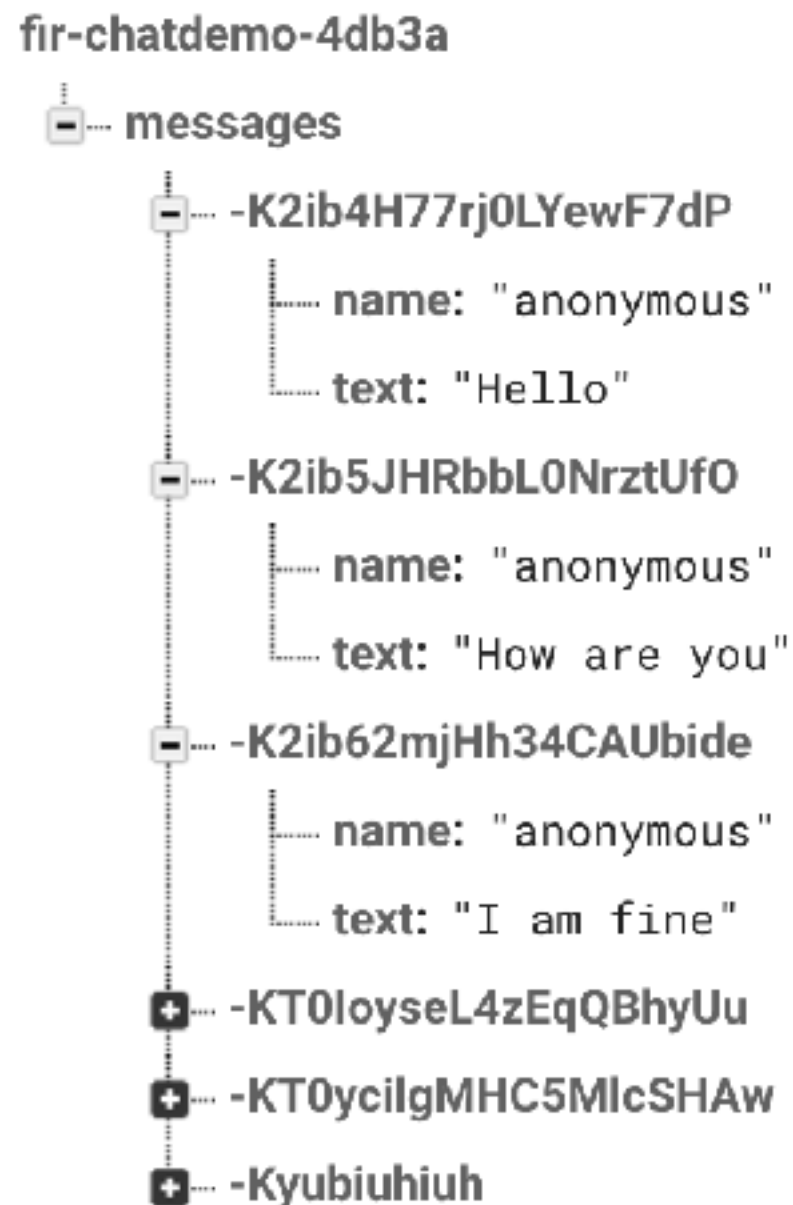
Why Firebase?

1. It's real-time! Allows us to update the view as soon as something in the database changes
2. Has strong support for iOS and Android as well as Web, Unity, C++
3. Thorough documentation - see https://firebase.google.com/docs/ios/setup
4. Can be easily incorporated into project via Cocoapods
5. Supports not only simple data storage but also authentication, file storage, cloud messaging, analytics, and more.
6. Biggest competitor, Parse, shut down in 2015.

How does Firebase work?

Firebase is built on a NoSQL database
  - Literally no SQL involved - data stored as a JSON tree

```
fir-chatdemo-4db3a
  messages
    -K2ib4H77rj0LYewF7dP
      name: "anonymous"
      text: "Hello"
    -K2ib5JHRbbL0NrztUfO
      name: "anonymous"
      text: "How are you"
    -K2ib62mjHh34CAUbide
      name: "anonymous"
      text: "I am fine"
    -KT0loyseL4zEqQBhyUu
    -KT0ycilgMHC5MlcSHAw
    -Kyubiuhiuh
```

- Data represented as a set of nodes, each with corresponding child nodes
- Retrieve data within app as a dictionary with key-value pairs.

Configuring Firebase

Initializing Firebase

```swift
import Firebase

class AppDelegate: UIApplicationDelegate... {

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions...) -> Bool {

        FirebaseApp.configure()

        return true
    }
}
```

Managing Users

User-Driven Data

For any application, we need to be able to:

- Create accounts for users
- Store a user's authentication state
- Store a user's basic information (name, profile pic, etc)
- Associate data objects (messages, photos, etc.) to the user who created them.

Firebase allows us to handle this by assigning unique user ID's

Firebase Users

For any user, Firebase stores:
- A unique user ID
- Email address
- Display name
- Photo URL

Firebase maintains a `FIRUser` instance which keeps track of the current user.
- Persists the user's state so that closing the app or losing connection doesn't sign the user out.

Creating a new user

```swift
import FirebaseAuth

Auth.auth().createUser(withEmail: emailText, password:
    passwordText) { (user, error) in

    if let error = error {
      // handle error
    } else {
      // do something
    }

}
```

Signing in

```swift
Auth.auth().signIn(withEmail: emailText, password:
passwordText) { (user, error) in

  if let error = error {
    // handle error
  } else {
    // do something
  }

}
```

Setting a user's display name

```swift
let changeRequest = user!.createProfileChangeRequest()
changeRequest.displayName = newName

changeRequest.commitChanges() { (error) in
  if let error = error {
    // handle error
  }

}
```

Getting the current user

If we want to access the properties of the currently signed in user, we can do something like:

```
let user = Auth.auth().currentUser
let name = user?.displayName
let email = user?.email
let uid = user?.uid
let photoURL = user?.photoURL
```

We can also use the currentUser variable to check if a user is already signed in (instead of logging in every time).
However, it is safer to use a listener:

```
Auth.auth().addStateDidChangeListener() { (auth, user) in
  // do something with user
}
```

Saving/Retrieving Data

Structuring Data

Recall that data is stored on Firebase as a JSON tree.

- Each time we add data to the tree, it becomes a node in the tree with a key and value.
- We can access a value in the tree by following its key-path in the tree.
- If we attempt to access a node in the database, we get access to all of its children as well.
  - Potential pitfalls of this?

```
"chats": {
  "one": {
    "title": "Historical Tech Pioneers",
    "messages": {
      "m1": { "sender": "ghopper", "message": "Relay malfunction found. Cause:
      "m2": { ... },
      // a very long list of messages
    }
  },
  "two": { ... }
}
```

Writing Data to Firebase

Create a reference to the root node:

```swift
import FirebaseDatabase
let dbRef = Database.database().reference()
```

Save data to a node:

```swift
dbRef.child("Users").child(user.uid).setValue(username,
    forKey: "username")
```

We can also specify the entire path directly:

```swift
dbRef.child("Users/\(user.uid)/username").setValue(username)
```

Save multiple values to a node:

```swift
let dict: [String: Any] = ["email": email,
                            "preferences": preferences]
dbRef.child("Users/\(user.uid)").setValue(dict)
```

Note: You can also create unique ID's for children using the childByAutoID function.

Reading Data from Firebase

Create a listener (called when a particular node changes):

```swift
let refHandle = dbRef.child("Users/\(user.uid)").observe(.value,
    with: { (snapshot) in

    if snapshot.exists() {
        if let userDict = snapshot.value as? [String : Any] {
            let newValue = userDict["username"] as! String
        }
    }
})
```

Note that the code inside the closure will execute *every* time the user's node on Firebase (or any of its children) changes.
- We can also query Firebase a single time by calling the observeSingleEvent function instead.

Storing Files

How does Firebase store files?

Firebase's database is only capable of storing numbers, arrays, dictionaries, and strings.

What if we want to store an image? (e.g. Snapchat Clone)

Firebase has a separate module for storage where we can upload all of our files - then we can just store its path in the storage section as a string in the database.

Store an image on Firebase

Just like with the database, we need a reference to the root node of the storage module:

```swift
import FirebaseStorage
let storageRef = Storage.storage().reference()
```

Then we can upload a file to a specific path as:

```swift
storageRef.child("images/img.jpg").putData(imageData,
metadata: nil) { (metadata, error) in

    guard let metadata = metadata else {
        //handle error
        return
    }
    let downloadURL = metadata.downloadURL
}
```

Download an image from Firebase

We can download an image either by using its path:

```swift
let imageRef = storageRef.child("images/img.jpg")
imageRef.getData(maxSize: 1*1024*1024) { (data, error) in
    if let error = error {
        // handle error
    } else {
        let image = UIImage(data: data!)
    }
}
```

Or by its download URL:

```swift
let imageRef = Storage.storage().reference(forURL: downloadURL)
imageRef.getData(maxSize: 1*1024*1024) { (data, error) in
    if let error = error {
        // handle error
    } else {
        let image = UIImage(data: data!)
    }
}
```