

Build a WhatsApp clone for iOS using AWS

With Amazon Web Services, developers get a powerful platform that allows them to easily create a WhatsApp-like messaging application. Here, we will show how you can develop a social messaging app using a variety of AWS technologies.

AWS Sample Messenger Features

A typical messaging application should include features such as:

1. Log into the app using Facebook or Google+ accounts
2. Create chat rooms and invite friends from the address book
3. Send/Receive messages
4. Store and share pictures
5. Measure adoption with key metrics

Most functionalities in a messaging app require a lot of work, both inside the app and at the backend. For starters, the developer has to write the code to integrate sign-in functionality of each of the social messaging platforms; he/she also has to create an infrastructure that is capable of handling millions of chat rooms and billions of messages every day. In addition to all this, he/she must also keep track of consumers' usage patterns to enhance the ease-of-use and maximize revenue.

With the strength of its infrastructure, AWS makes it extremely easy for developers to handle all these processes. By providing a single hub to work with, and automating a lot of the infrastructural work, AWS significantly cuts down on app development time; it reduces time needed to market the app, and minimizes operational cost.

To give you an idea of how AWS simplifies app development, we will list the various AWS technologies that we'll be using to build our messaging app.

AWS Mobile Hub

To quickly setup, configure and customize features that make our iOS app dynamic. AWS Mobile Hub will also help us cut down on development time.

Amazon Cognito Identity

To log into our messaging app, using Facebook and/or Google+ accounts

Amazon Mobile Analytics

To capture user actions for UX analysis

Amazon DynamoDB

To create and manage chat rooms, and store user messages

Amazon S3

To store the images shared by users in chat rooms

Amazon SNS

To send Push Notifications to chat room participants when a new message is sent to the chat room

Amazon Device Farm

To test sample application across multiple devices

Prerequisites

- Prior knowledge of SWIFT
- AWS account
- A working Xcode 7.1 or newer version

1. Project Setup

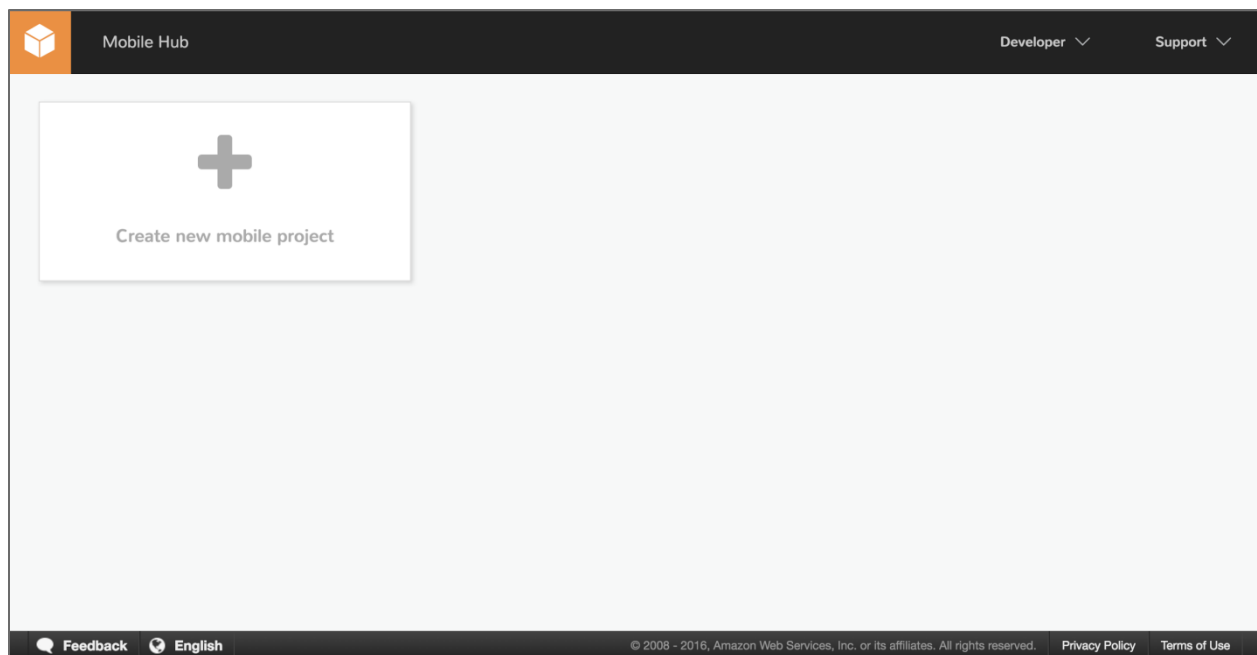
With this background, we are now ready to start developing our application. We will begin with creating an iOS project using **Amazon Mobile Hub** which is a powerful console to configure complex projects easily and conveniently through a single interface.

[What is Amazon Mobile Hub?](#)

- Go to [Mobile Hub Console](#) and log onto your AWS developer account. The AWS Mobile Hub Console lets you to conveniently create, edit and delete your projects.

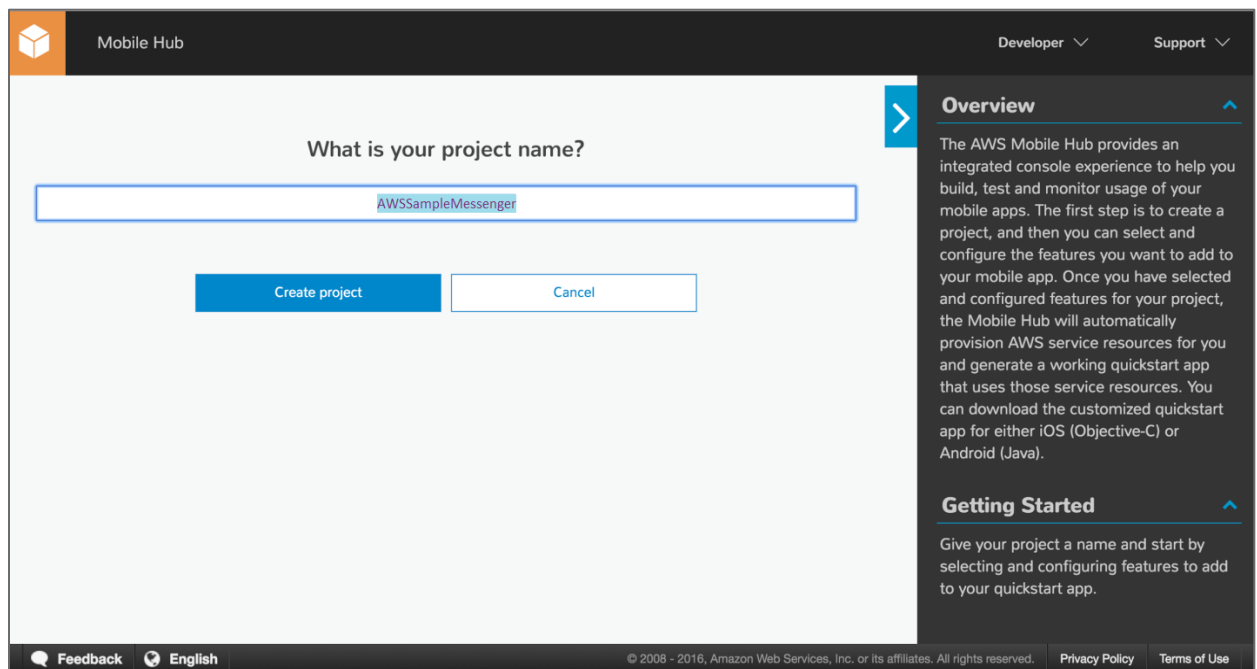
I don't have an AWS developer account. [Create a new account.](#)

- Select "Create new mobile project" from the dashboard



Screen 1: AWS Mobile Hub Dashboard

- Enter the name of your project. For this tutorial, we will be naming our project `AWSSampleMessenger`.



Screen 2: Creating a new project

- Now simply click on “Create Project” to create a basic project, ready for further configuration and addition of AWS functionalities.

2. Social Networks Sign-in

Our sample application will allow users to sign-in using their existing social network accounts, such as Facebook and Google+. This can be easily done using **Amazon Cognito Identity** service. With the basic project in place, we can now add the *Cognito Identity* feature using Mobile Hub.

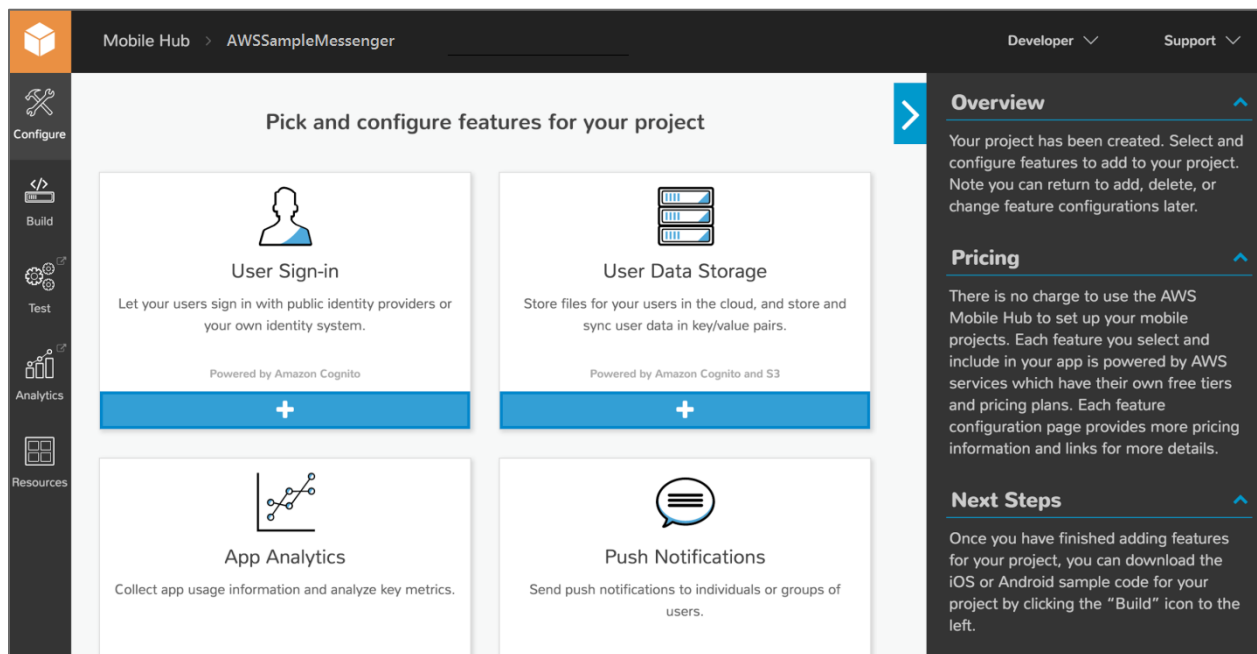
For more information about Cognito identity, visit [Amazon Cognito](#)

What does it cost?

For your first 50,000 monthly active users (MAUs), Amazon Cognito Identity is completely **free**.

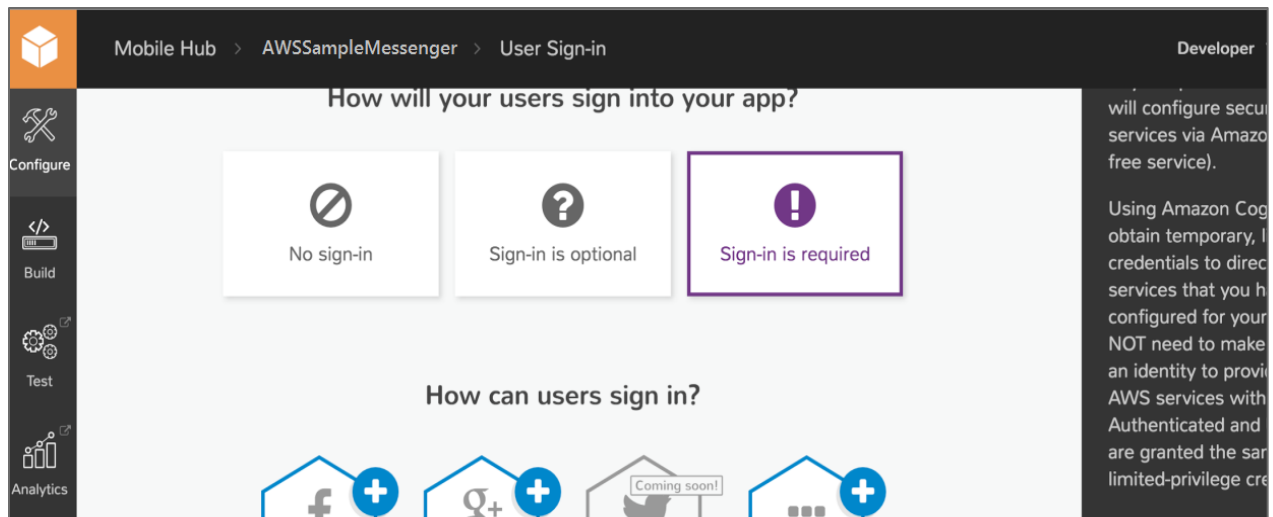
Click [here](#) for information about pricing for MAUs above the free tier

- From the Mobile Hub dashboard, select the project name and click on the + button on the “User Sign-in” box.



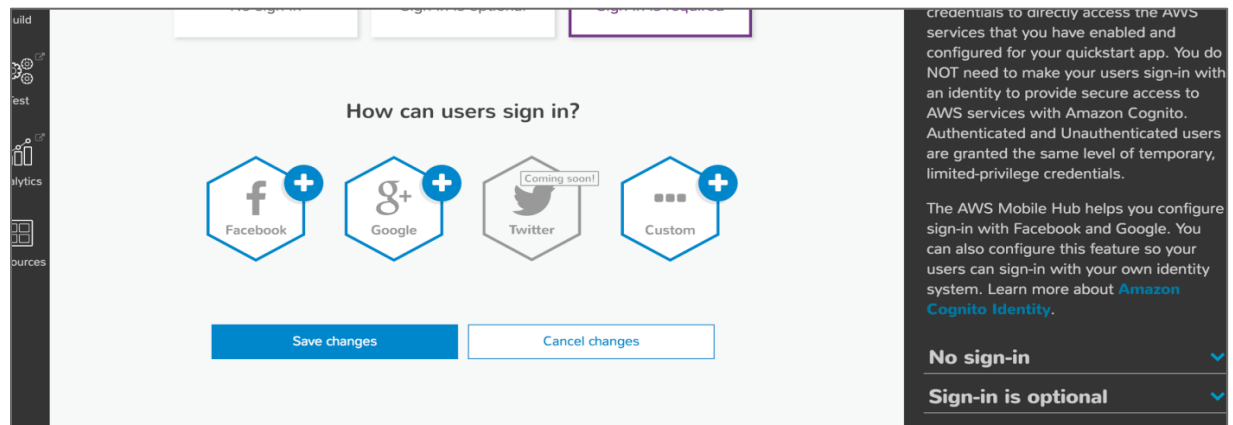
Screen 3: Project customizable features

- Since our application will enforce sign-in *before* anyone can use the application, we will select “Sign-in is required”.



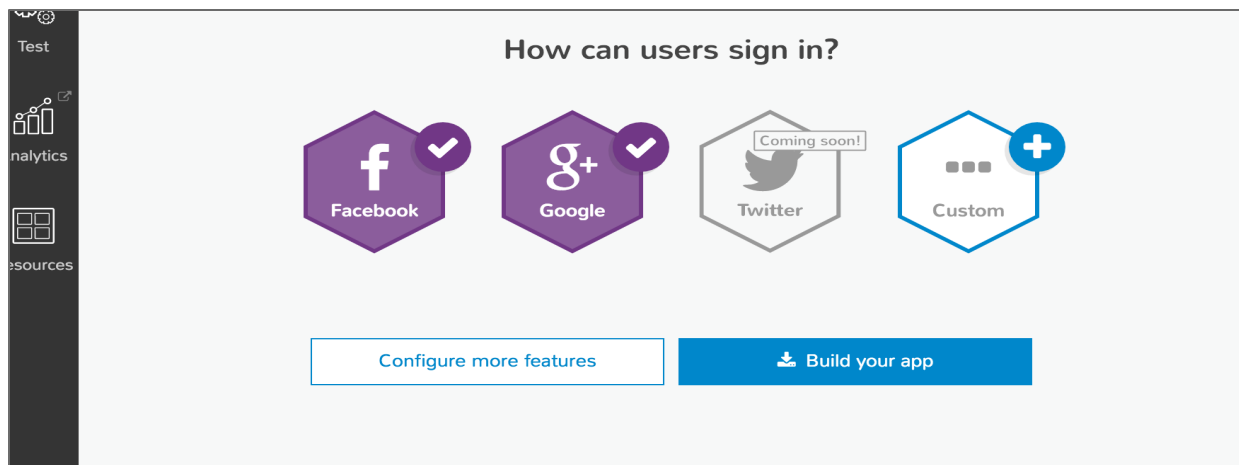
Screen 4: Enabling social media log-in

- To configure Facebook for your app, click on the + button on the Facebook logo and follow the instructions provided at: <https://developers.facebook.com/docs/ios/getting-started#appid>



Screen 5: Configuring different social media networks

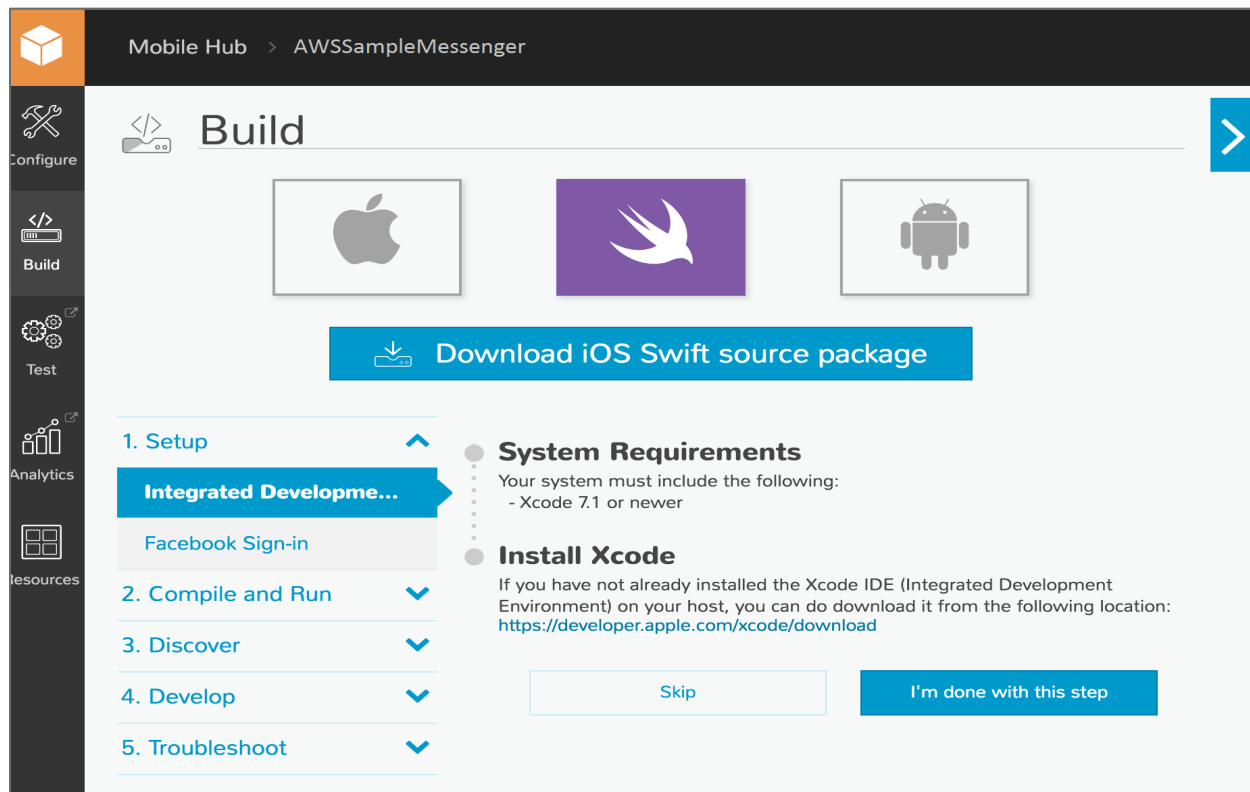
- To configure Google+ for your app, click on the + button on the Google+ logo and follow the instructions provided at: https://developers.google.com/+/mobile/ios/getting-started#before_you_begin
- After successfully configuring social media log-in, click on “Save Changes”.
- Once everything is configured successfully, click on “Build your app”.



Screen 6: Building the app

Note: At this stage, Mobile Hub has brought in the required *Cognito Identity* feature, but we still need to select the platform and the programming language. Mobile Hub supports Android, Objective-C based iOS and Swift-based iOS platforms. For this sample application, we want to use Swift for iOS programming.

- First select the box for the iOS Swift platform logo and then click on “Download iOS Swift source package”.



Screen 7: Downloading the source package

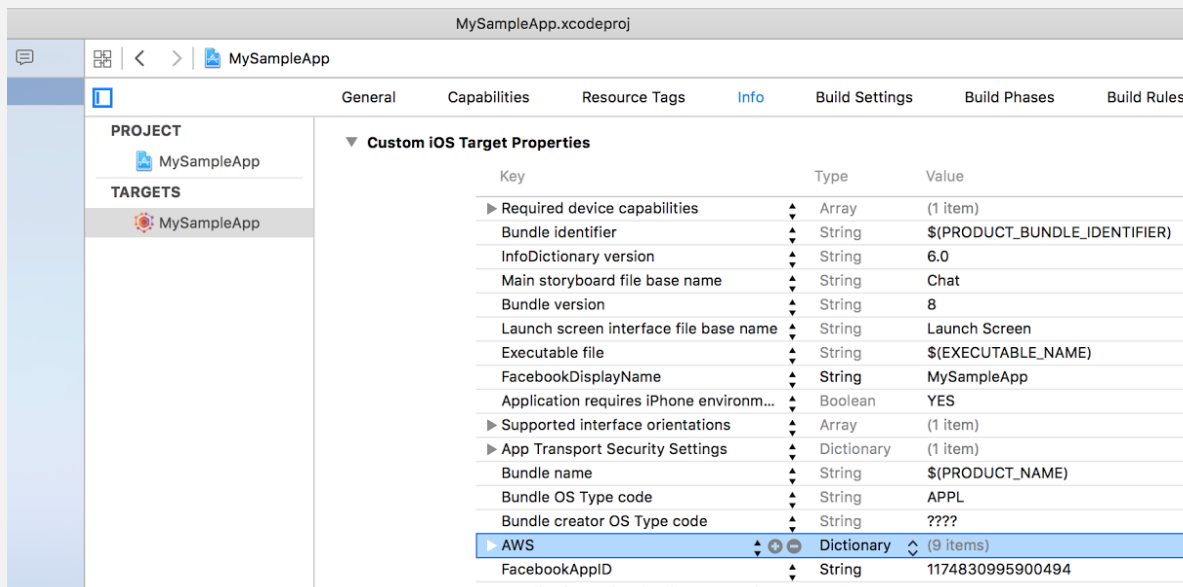
How to use sample application source code

As you know, any application built with AWS services, needs to be associate with your Amazon Developer Account. Similarly, services including Facebook login, Google+ login, AWS SNS etc. require some configuration that is specific to you. The accompanying source code for the sample messenger application does not contain any reference to the Amazon Developer Account or other services. Before you can use this source code, you have to do some configuration. Here are a couple of ways to go about it.

Option # 1

In this option, you will reuse the sample application project and add AWS, Facebook and Google+ specific configuration to the project.

- Download and select the AWS Sample Messenger Project in Xcode and press “Info”
- Expand the AWS key



- Make the following changes to the specified keys:

Key	Modification
MobileAnalytics	Enter AppId values
MobileHub	Enter ProjectClientId values
UserFileManager	Enter S3Bucket values
PushManager	Enter SNSPlatformApplicationARN and SNSConfiguredTopics values
ContentManager	Enter S3Bucket value
CredentialsProvider	Enter PoolId value
IdentityManager	Enter Google ClientId value
FacebookAppID	Enter FacebookAppID value

- Create the following DynamoDB tables in Amazon Mobile Hub
 - UserProfile
 - ChatRoom
 - Conversation

Note: Attributes for these tables can be found in Section 4 of this tutorial, under ‘Sample Database Schema’

Option # 2

- Create and configure your own project on Amazon Mobile Hub as described in this tutorial
- Download the source code for the AWS Sample Messenger Project.
- Drag and drop the Chat folder from the sample project into your own project
- Select the Chat storyboard as the Main Interface in your project’s “General” Tab
- Copy All code from the sample project’s AppDelegate file and paste into your project’s AppDelegate file

- Once these steps are done, simply run the iOS code.

At this stage, if you launch the application, you will automatically get a splash screen that Amazon Mobile Hub has generated for you. You will also be able to see the following sign-in screen [Screen 8] that allows you to login, using your Facebook and/or Google+ account(s) [depending on which network(s) you enabled during configuration].

Amazon Cognito Identity does the heavy lifting for you while still giving you the flexibility to customize these screens to meet your specific needs.



Screen 8: Sample sign-in screen

3. Amazon Mobile Analytics

So we have the sample project in place, and have created it to allow users to sign in from their social media accounts. At this stage, we need to incorporate another feature into our project; one that is a must-have for any consumer-facing application – Analytics. Among other things, an analytics feature would allow us to evaluate a user's app usage from different social media networks.

In-app analytic tools help generate useful data regarding an app's consumption, impression and reach. By incorporating analytics early into your app, you can effectively extract, quantify, organize and analyze data about important use cases, which can help you make data-driven decisions to monetize your app and increase user engagement.

Amazon Mobile Analytics helps you measure your app usage by automatically generating and compiling analytic reports. This data can help you track key trends about your users, evaluate and extrapolate app revenue, gauge user retention, and monitor custom in-app behavior events.

So let's begin by incorporating Amazon Mobile Analytics into our application. The goal is to

For more information about analytics, visit [Amazon Mobile Analytics](#)

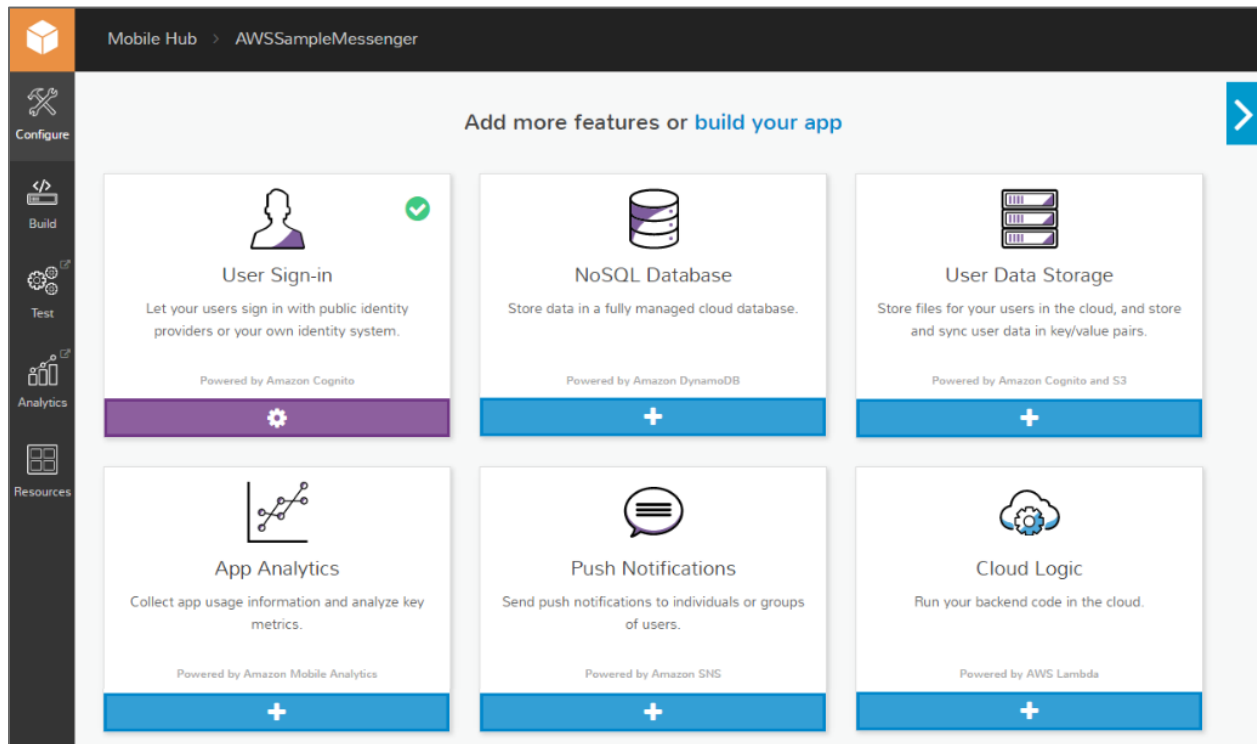
What does it cost?

For up to 100 million events per month, Mobile Analytics is completely **free**.

Click [here](#) for information about pricing for events above the free tier

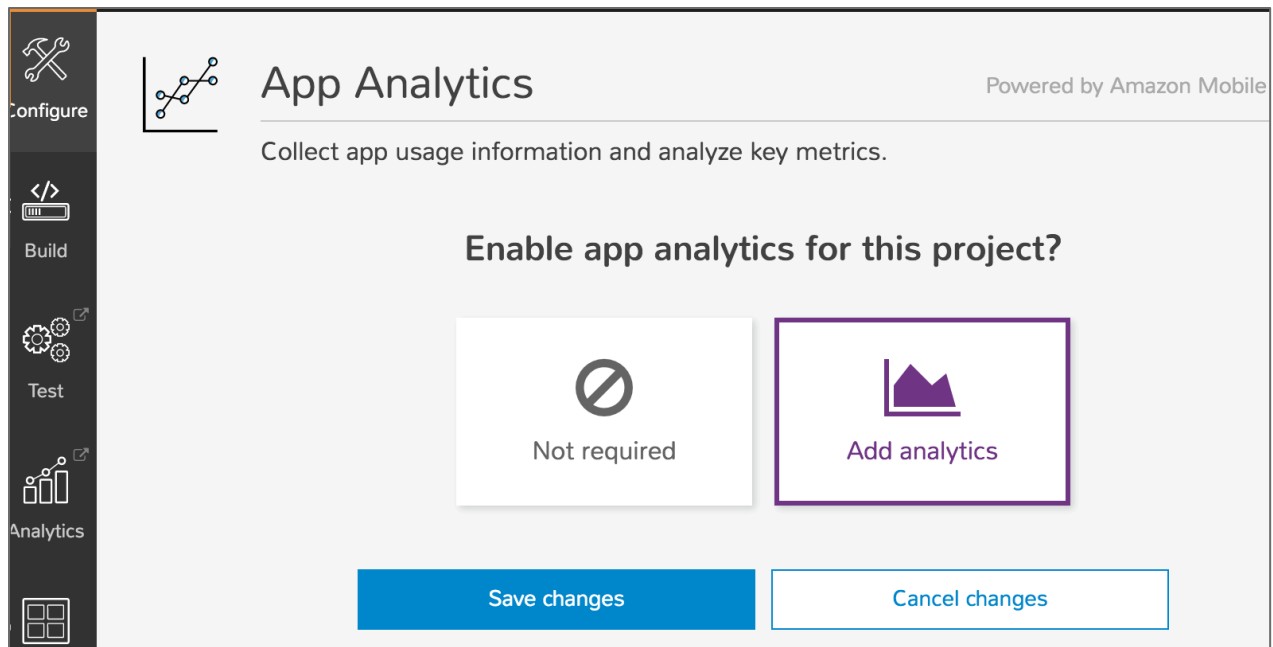
track whether the user logged in using his/her Facebook or Google+ account.

- In order to do that, go to the project dashboard on Mobile Hub and click on the + button on the “App Analytics” box.



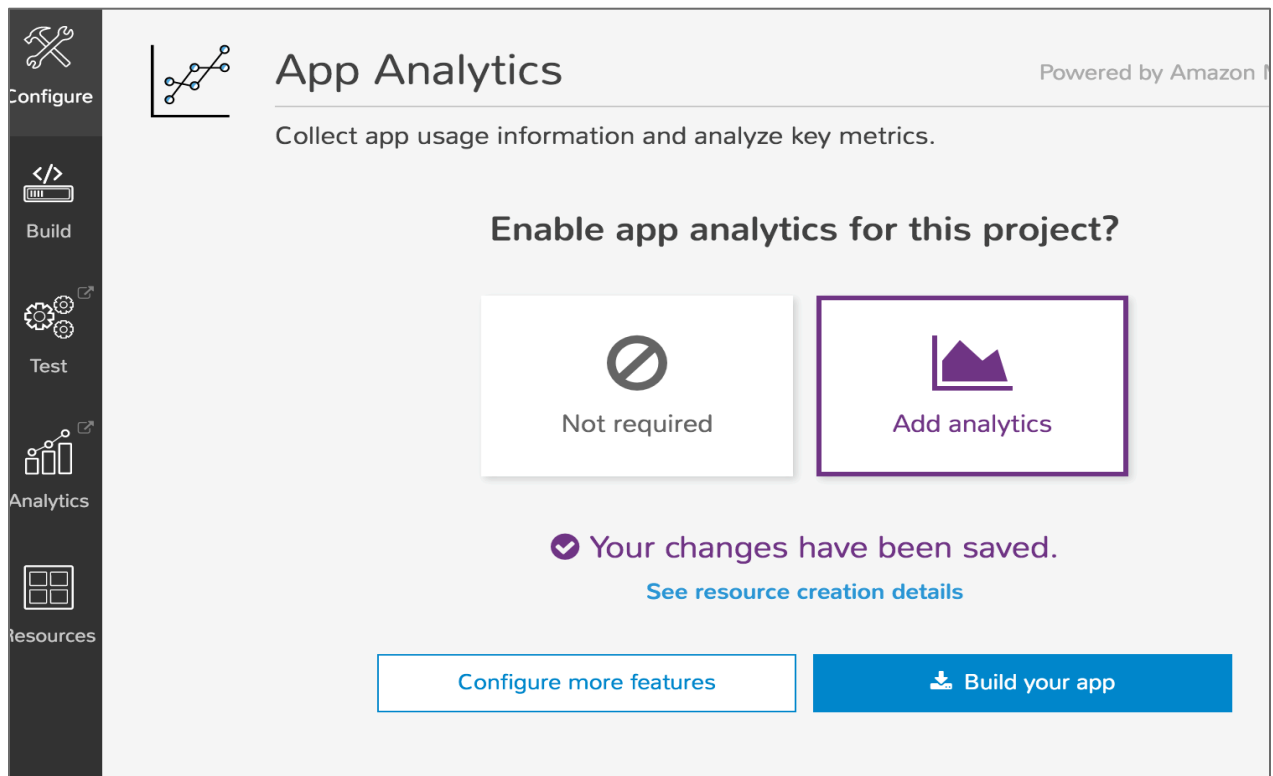
Screen 9 Project features

- On the resulting screen, select “Add Analytics”, and click on “Save Changes”.



Screen 10 App Analytics feature

- Once the changes are successfully saved, simply click on the “Build your app” button.

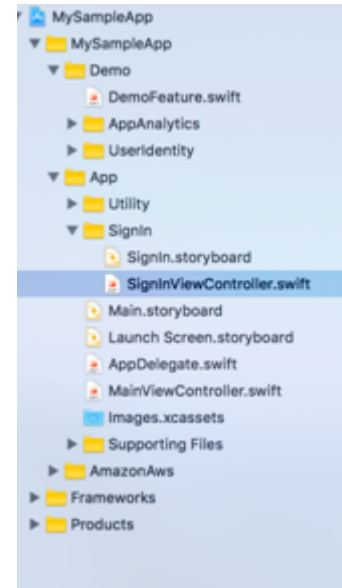


Screen 11 Building the app after enabling App Analytics

- Now, like we did earlier for Cognito Identity, choose the box with the iOS Swift platform logo, and download source package [Screen 7].

So far we have added the Amazon Mobile Analytics package in the iOS project. Now we will add an Analytics-specific source code to the project.

- Open the downloaded project in Xcode
- Find and select filename
“SignInViewController.swift” from the project navigator.
- Add the following code to track user actions in the `func handleFacebookLogin()` and `func handleGoogleLogin()` functions of the above class.



Screen 12 Locating file in Xcode

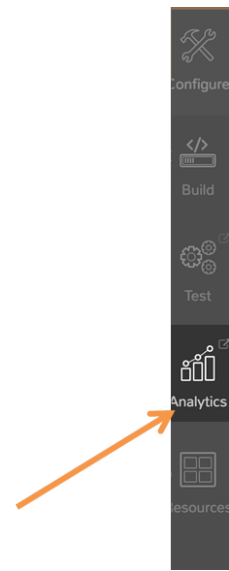
```
func handleFacebookLogin() {
    let eventClient = AWSMobileClient.sharedInstance.mobileAnalytics.eventClient
    let event = eventClient.createEventWithEventType("SignIn")
    event.addAttribute("Facebook", forKey: "LoggedIn")
    eventClient.recordEvent(event)

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), {() -> Void
in
        eventClient.submitEvents()
    })
    handleLoginWithSignInProvider(AWSFacebookSignInProvider.sharedInstance())
}

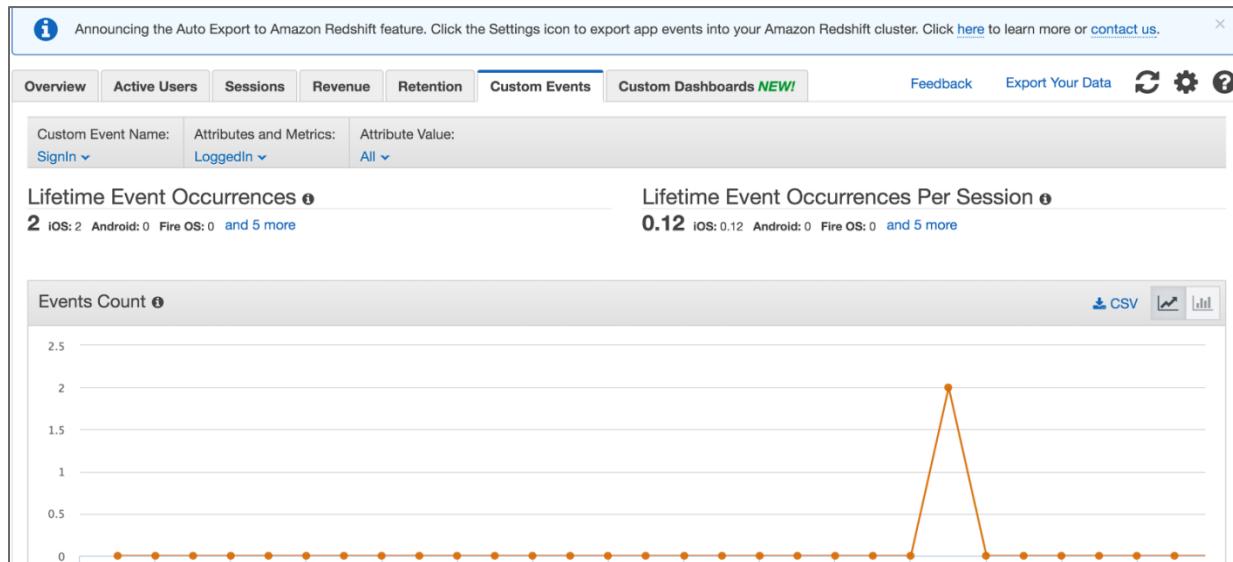
func handleGoogleLogin() {
    let eventClient = AWSMobileClient.sharedInstance.mobileAnalytics.eventClient
    let event = eventClient.createEventWithEventType("SignIn")
    event.addAttribute("Google+", forKey: "LoggedIn")
    eventClient.recordEvent(event)

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), {() -> Void
in
        eventClient.submitEvents()
    })
    handleLoginWithSignInProvider(AWSGoogleSignInProvider.sharedInstance())
}
```

This is all you need to track the Facebook and Google+ login usage. You can now view the resulting data by going to the Analytics tab in the left panel in Mobile Hub and view all the analytics, as shown in Screen 14.



Screen 13: Analytics tab on Mobile Hub



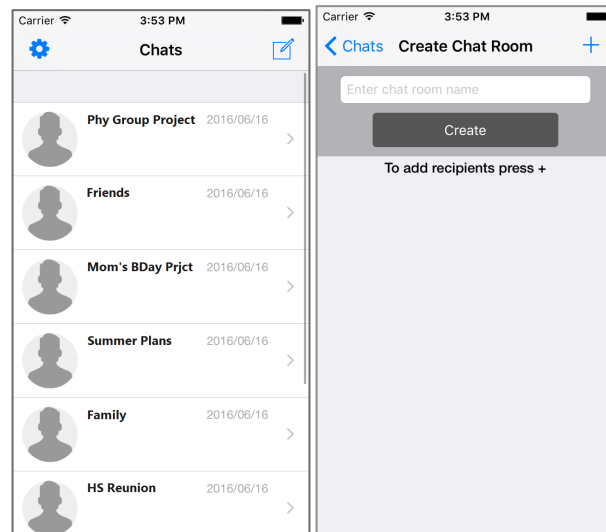
Screen 14: App Analytics console

4. Amazon NoSQL Database

This brings us to the development of the main functionality of our messenger application – the ability to allow users to create chat rooms, add participants to individual chat rooms, and send/receive messages in a chat room. Our goal is to create a messaging app that, when logged into, welcomes the user with a dashboard screen similar to Screen 15.

How does it work?

The user can create a new chat room by pressing the icon on the top-right corner of the first screen. Similarly, when the user presses an individual chat room, he/she is presented with a detailed view of said chat room.



Screen 15 Sample app dashboard & creating a chat room

This application needs to be able to handle the potential of hundreds of thousands of active chat rooms, if not more. It also needs to be capable of handling millions of messages being sent and received simultaneously.

Traditional relational databases do not scale well in such situations. Hence, the solution is to use a NoSQL database; this is where **Amazon DynamoDB** comes in. Amazon DynamoDB is Amazon's cloud-based NoSQL Database service, which allows you to create a database and have it managed by a service that stores and retrieves your in-app data. For our sample messenger, the *Amazon NoSQL Database* feature would allow us to store information about chat rooms, conversations, and chat room recipients.

For more information about Amazon DynamoDB, visit [Amazon DynamoDB](#).

What does it cost?

DynamoDB customers can get started with a **free tier**.

Visit [DynamoDB Pricing](#) for more information about all the pricing tiers available.

Sample Database Schema

To implement Amazon DynamoDB, we must first create a database schema. This database schema would help us define how the data would be organized, as well as the relationship between different aspects of the data. For this particular application, we need to create four different tables, each with a different set of attributes.

1. UserProfile (Protected)

userId(PK) - String	phone(SK) – String	Name - String	pushTargetArn – String set
----------------------------	---------------------------	----------------------	-----------------------------------

2. ChatRoom (Protected)

userId(PK) – String	chatRoomId(SK) – String	createdAt(SK) – String	name - String	recipients - String set
----------------------------	--------------------------------	-------------------------------	----------------------	--------------------------------

Index name: ByCreationDate
Partition key: chatRoom Id – String
Sort key: createdAt – String

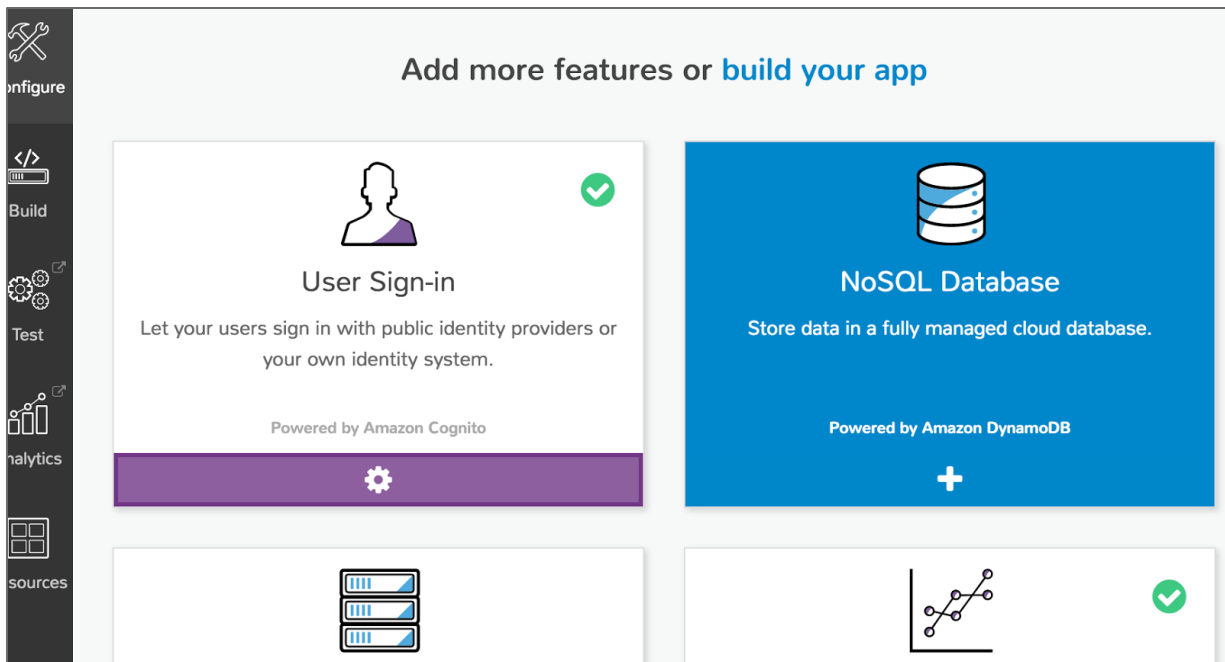
3. Conversation (Protected)

userId(PK) – String	conversationId(SK) - String	createdAt – String	chatRoomId – String	message - String
----------------------------	------------------------------------	---------------------------	----------------------------	-------------------------

Index name: ByCreationDate
Partition key: chatRoom Id – String
Sort key: createdAt – String

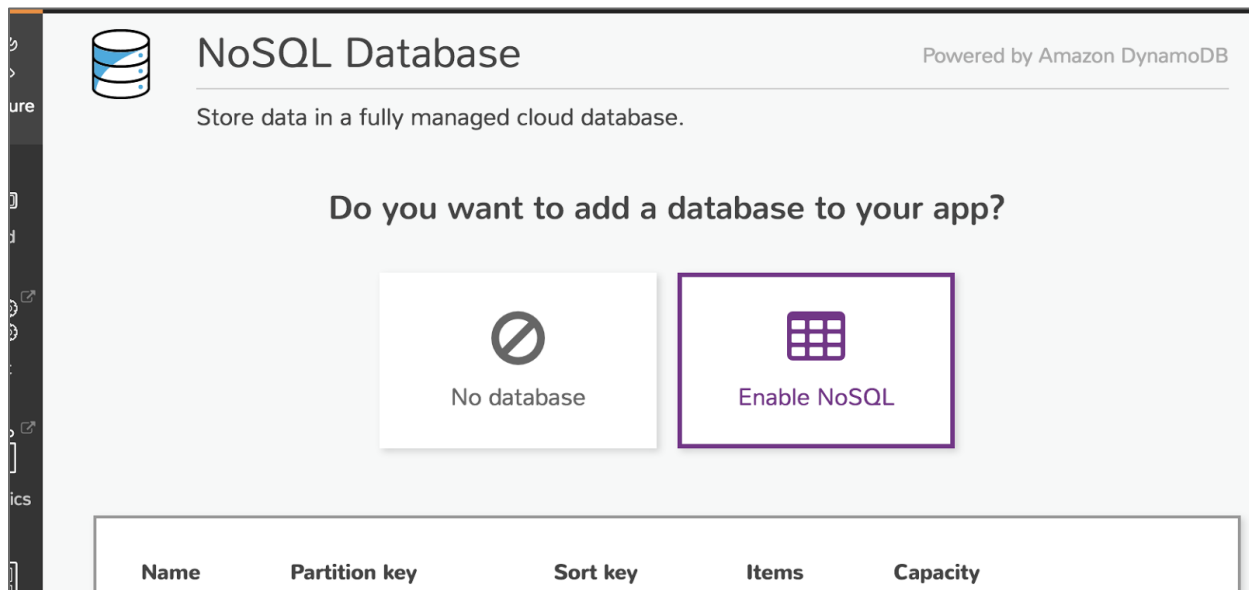
Now that we have prepared a database schema, we can integrate the *Amazon NoSQL Database* feature into our project.

- Open the project dashboard in Mobile Hub and go to its list of customizable features
- Click on the + sign on the “NoSQL Database” box



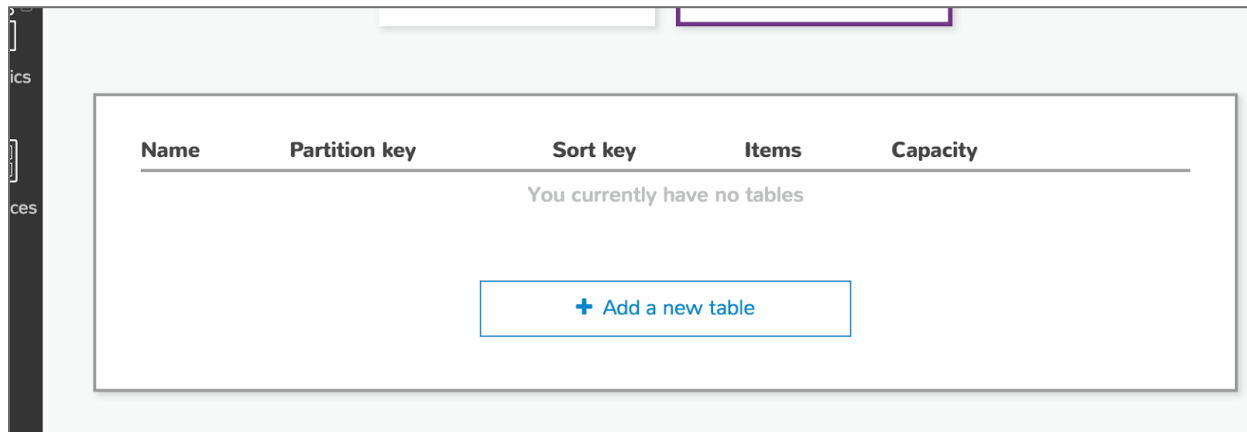
Screen 16: NoSQL Database feature

- Select the “Enable NoSQL” option on the resulting screen.



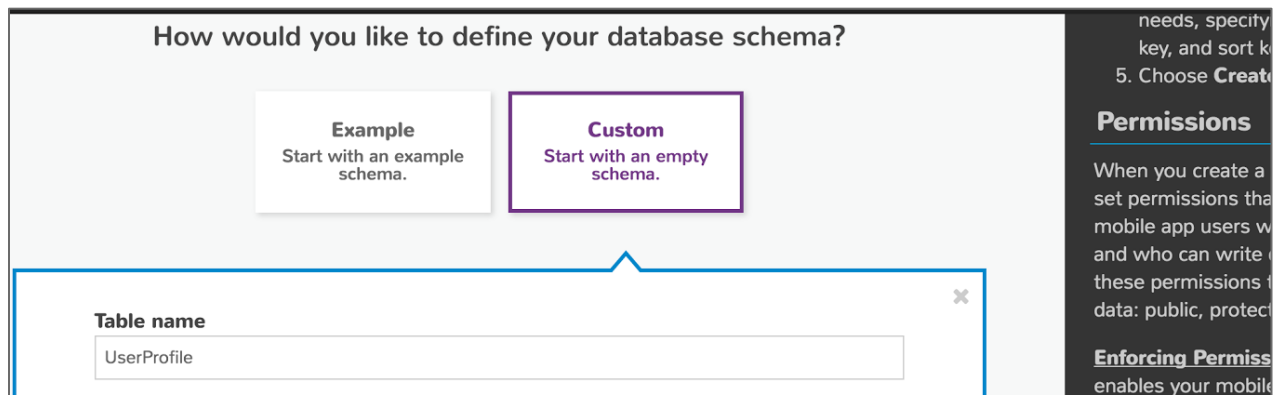
Screen 17: Enabling NoSQL

- Click on the “Add a new table” box in the section below



Screen 18: Adding a table to the database

- Now you must define your database schema. To proceed with a blank, customizable schema, select “Custom”.



Screen 19: Creating a customized schema

How does a database schema help?

The database schema is the design you’ll be using to develop your database in NoSQL. For our messaging application, the database we develop will allow us to store chat room information and messaging history.

- Enter the name of the first table you wish to create. As defined in Section 4, Sample Database Schema sub-section 1, we will write `UserProfile` in the field for Table name
- Now select the permission settings you’d like your database schema to have. Each permission option briefly describes the setting; details for the permission setting appear on the right sidebar. Select “Protected”

How would you like to define your database schema?

Example
Start with an example schema.

Custom
Start with an empty schema.

Table name

UserProfile

Table resource: yourproject-mobilehub-XXXXXXXXXX-UserProfile

What permissions would you like for this table?

Public
Any app user can read and write to any item.

Protected
Any app user can read, only owner can write to item.

Private
Only the owner can read and write the item.

needs, specifying the name, partition key, and sort key for each.

5. Choose **Create table**.

Permissions

When you create a new table, you must set permissions that determine for your mobile app users who can read data from and who can write data. You can set these permissions to control access to data: public, protected, and private.

Enforcing Permissions: This feature enables your mobile app to directly access your NoSQL tables in the Amazon DynamoDB service. Because there is no middle layer between the mobile app and the database service, it is important that an appropriate access policy is used to restrict access to your tables. A fine-grain access control policy is created for your mobile app users when you select permissions to be Public, Protected, or Private and the permissions are enforced by comparing your table's primary index partition key (which must be called 'userId' and be of type 'string') against the mobile app user's Amazon Cognito Identity ID. If you select Protected or

Screen 20: Choosing permission setting for table

- Enter the attributes of the table. As shown in the Sample Database Schema section, add attributes for table name: `UserProfile`

userId(PK) - String	phone(SK) – String	Name - String	pushTargetArn – String set
----------------------------	---------------------------	----------------------	-----------------------------------

First attribute

Enter attribute name: `userId`. Set type as: `string`. Check box for Partition key

Second attribute

Enter attribute name: `phone`. Set type as: `string`. Check box for Sort key

Third attribute

Enter attribute name: `name`. Set type as: `string`

Fourth attribute

Enter attribute name: `pushTargetArn`. Set type as: `string set`

What attributes do you want on this table?

Attribute name	Type	Partition key	Sort key	
<input type="text" value="userId"/>	<input type="text" value="string"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	✕
<input type="text" value="phone"/>	<input type="text" value="string"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	✕
<input style="border: 2px solid blue;" type="text" value="name"/>	<input type="text" value="string"/>	<input type="checkbox"/>	<input type="checkbox"/>	✕

✓ Queries this table can perform.

Primary Index Queries	Examples
-----------------------	----------

Screen 21: Adding table attributes

- After entering all the attributes for the table `UserProfile`, click on the “Create table” button appearing at the bottom of the screen

Primary Index Queries	Examples
Get Item	Find item with userId = ABC and phone = ABC
Query by Partition Key	Find all items with userId = ABC
Query by Partition Key and Sort Condition	Find all items with userId = ABC and phone is < zzz
Query by Partition Key, Sort Condition, and Filter	Find all items with userId = ABC and phone is between aaa and zzz and name begins with XY

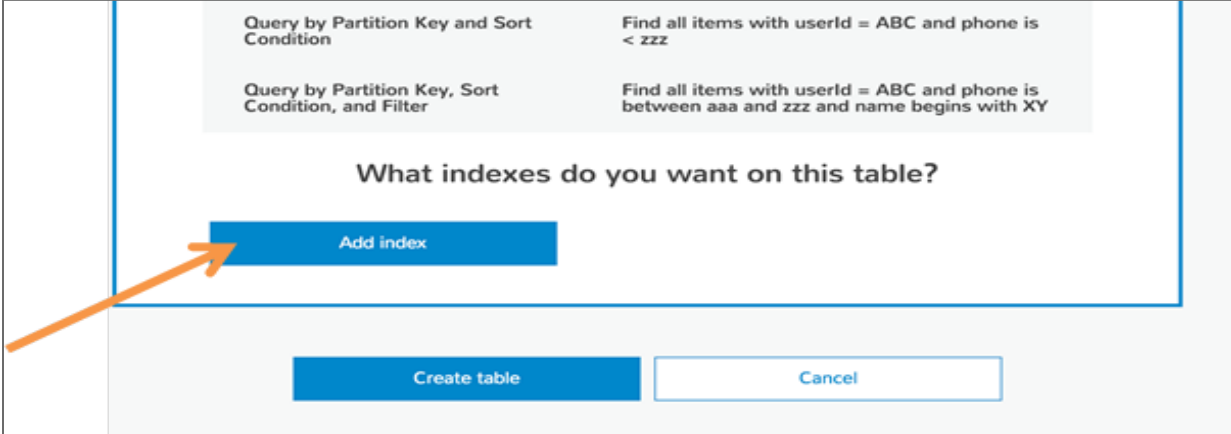
What indexes do you want on this table?

Screen 22: Creating a table

- Now repeat the same steps for tables `ChatRoom` and `Conversation`.

Adding an index

- To add index for a particular table, click on “Add index” after entering the table’s attributes but before clicking on “Create table”



Query by Partition Key and Sort Condition Find all items with userId = ABC and phone is < zzz

Query by Partition Key, Sort Condition, and Filter Find all items with userId = ABC and phone is between aaa and zzz and name begins with XY

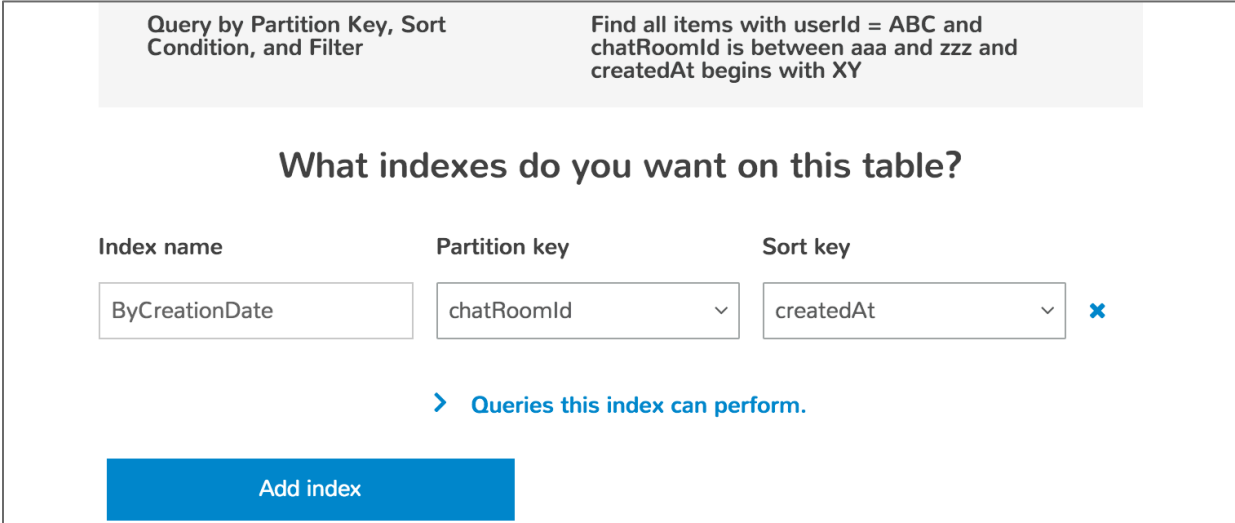
What indexes do you want on this table?

[Add index](#)

[Create table](#) [Cancel](#)

Screen 23: Adding an index to a table

- Select index attributes and click on “Add index” once again



Query by Partition Key, Sort Condition, and Filter Find all items with userId = ABC and chatRoomId is between aaa and zzz and createdAt begins with XY

What indexes do you want on this table?

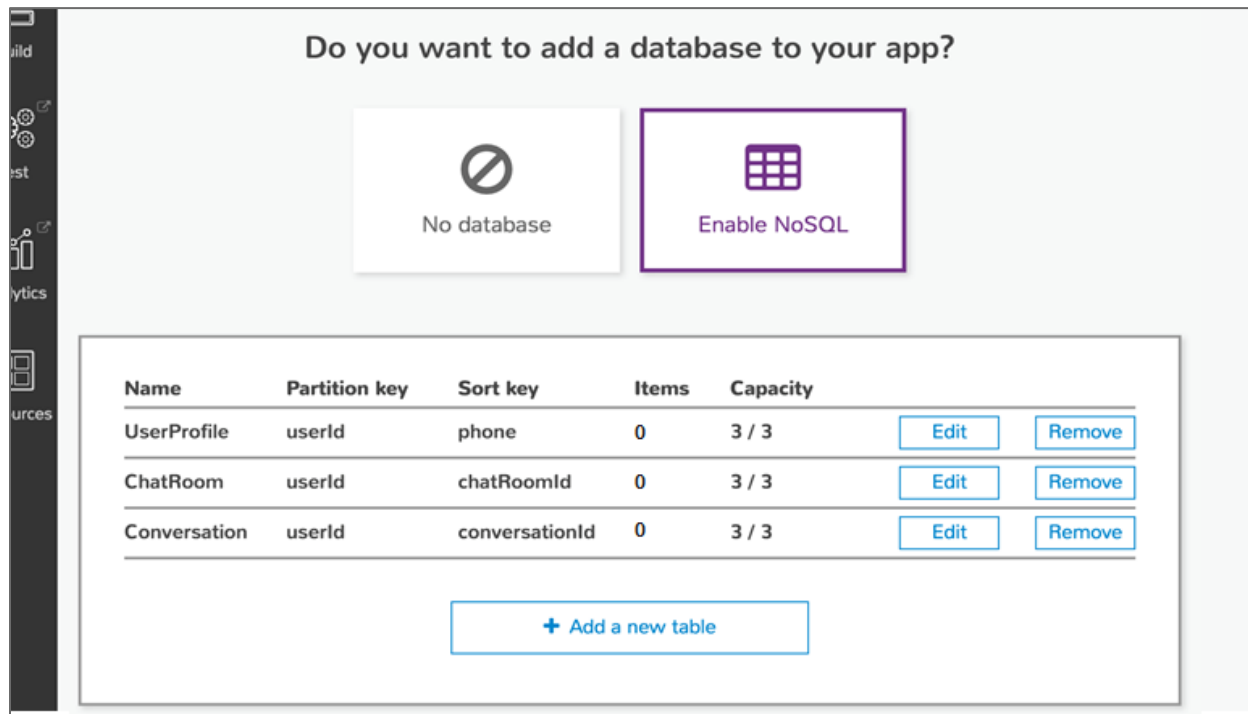
Index name Partition key Sort key

[> Queries this index can perform.](#)

[Add index](#)

Screen 24: Selecting index attributes

- Once all the steps are completed, this is how the tables should appear.



Screen 25: Final look of the tables

- Finally, build your app and download the Swift package

Creating Chat Rooms with Multiple Recipients

Once we have defined the database schema by creating the tables with their corresponding attributes, and have enabled NoSQL Database, we must enable our application to use that database to create chat rooms. Our messaging app needs to be able to allow users to easily create chat rooms and initiate conversations with one or multiple contacts from the device's address book. In order to successfully enable this, we must perform the following tasks:

- Create the function `loadUsersWithPhoneList` to get all selected recipients from the `UserProfile` table. Executing this code will allow us to load all the contacts from the logged-in user's database.

Note: In the accompanying sample application, while creating a new chat room, a user can only add recipients from his/her address book, who are already registered with the application, and who provided the same telephone number, during their first sign-in, that exists in the user's address book.

```

func loadUsersWithPhoneList(phoneList:Array<String>)->AWSTask {
    let scanExpression = AWS DynamoDB ScanExpression()
    var filters = Dictionary<String,String>()

    for index in 0...phoneList.count-1 {

        filters[":val\\(index)"] =
(phoneList[index].componentsSeparatedByCharactersInSet(NSCharacterSet.decimalDigitCharacterSet().invertedSet)).joinWithSeparator("")
    }

    let allKeys = Array(filters.keys)
    let keysExpression = allKeys.joinWithSeparator(",")
    scanExpression.filterExpression = "phone in \\(keysExpression)"

    scanExpression.expressionAttributeValues = filters

    return dynamoDBObjectMapper!.scan(UserProfile.self,
expression:scanExpression).continueWithBlock { (task) -> AnyObject? in

        if task.error != nil || task.exception != nil {
            print(task.exception)
            return AWSTask(error: NSError(domain: "", code: -11, userInfo: [
                NSLocalizedDescriptionKey: "Users are not found!"
            ]))
        }

        if task.result != nil {
            print(task.result)

            let paginatedOutput:AWS DynamoDBPaginatedOutput = task.result as!
AWS DynamoDBPaginatedOutput;

            for rect in paginatedOutput.items {
                print(rect)
            }

            return AWSTask(result: paginatedOutput.items)
        }
        return nil
    }
}

```

- Create the function `saveNewChatRoom` to insert new chat room in `ChatRoom` table. This code would enable the app to allow the user to create a new chat room with one or more recipients from the previously loaded data.

```

func saveNewChatRoom(chatRoomName:String?,userProfiles:[UserProfile])->AWSTask {
    let chatRoom = ChatRoom();

    chatRoom._chatRoomId = NSUUID().UUIDString
    chatRoom._createdAt = NSDate().formattedISO8601
    chatRoom._userId = AWSIdentityManager.defaultIdentityManager().identityId!
}

```

```

chatRoom._name = chatRoomName
chatRoom._recipients = Set<String>()

for userProfile in userProfiles {
    chatRoom._recipients?.insert(userProfile._userId!)
}

//Save
return dynamoDBObjectMapper!.save(chatRoom).continewithBlock { (task) -> AnyObject? in
    if task.error != nil || task.exception != nil {
        print(task.exception)
        return AWSTask(error: NSError(domain: "", code: -11, userInfo: [
            NSLocalizedDescriptionKey: "Chat room is not created"
        ]))
    }

    if task.result != nil {
        return AWSTask(result: "Chat Room Created")
    }
    return nil
}
}

```

- Create function in `CreateChatRoomVC` to call the two functions above in sequence.

```

@IBAction func createChatRoom(sender: UIButton) {

    if recipeintsDataSource.count > 0 {
        showBusyIndicator(true)
        userServices.loadUsersWithPhoneList(getAllSelectedPhone()).continewithBlock({ (task) ->
AnyObject? in
            if let _userProfiles = task.result as? [UserProfile] where _userProfiles.count > 0 {
                let name = self.chatRoomNameTextField.text?.isEmpty == true ? nil :
self.chatRoomNameTextField.text
                return self.chatServices.saveNewChatRoom(name,userProfiles: _userProfiles)
            }
            return nil
        }).continewithBlock({ (task) -> AnyObject? in

            if let result = task.result as? String {
                print(result)

                dispatch_async(dispatch_get_main_queue(), {
                    self.navigationController?.popViewControllerAnimated(true)
                    NotificationCenter.defaultCenter().postNotificationName("ReLoadChatRooms",
object: nil)
                })
            }
            self.showBusyIndicator(false)
            return nil
        })
    }
    else {
        let alertController = UIAlertController(title: "Error", message: "Please Add at least
one recipient user", preferredStyle: .Alert)

```

```

        let doneAction = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
        alertController.addAction(doneAction)
        presentViewController(alertController, animated: true, completion: nil)
    }
}

```

Load Chat Rooms on Screen

In the previous step, we allowed our app to create chat rooms, using information from the user's database. Now we must enable it to load all created chat rooms on one screen [as shown in Screen 15]. To allow our app to load a chat room onto the screen, we must create two functions

- We must first implement function `loadUserChatRooms` to load all associated chat rooms for a logged-in user

```

func loadUserChatRooms()->AWSTask {
    let loggedInUserId = AWSIdentityManager.defaultIdentityManager().identityId!
    let scanExpression = AWS DynamoDBScanExpression()

    scanExpression.filterExpression = "userId = :userId or contains(recipients, :userId)"
    scanExpression.expressionAttributeValues = [":userId":loggedInUserId]
    return dynamoDBObjectMapper!.scan(ChatRoom.self,
    expression:scanExpression).continueWithBlock { (task) -> AnyObject? in

        if let _result = task.result {
            print(_result)
            let paginatedOutput:AWS DynamoDBPaginatedOutput = _result as!
AWS DynamoDBPaginatedOutput;
            return AWSTask(result: paginatedOutput.items)
        }

        if let _error = task.error {
            print(_error)
        }

        if let _exception = task.exception {
            print(_exception)
        }
        return AWSTask(error: NSError(domain: "", code: -11, userInfo: [
            NSLocalizedDescriptionKey: "Recipient is not found"
        ]))
    }
}

```

- Then, we implement function `showChatRooms` in ChatTableViewCell

```

@IBAction func showChatRooms(sender: UIRefreshControl) {
    if AWSIdentityManager.defaultIdentityManager().loggedIn == false {
        return
    }
    self.refreshControl?.beginRefreshing()
    chatServices.loadUserChatRooms().continueWithBlock { (task) -> AnyObject? in
        sender.endRefreshing()
    }
}

```



```

    if let _chatRooms = task.result as? Array<ChatRoom> {
        self.chatRoomDataSource?.removeAll()
        self.chatRoomDataSource = _chatRooms
        //Sort loaded chat rooms by creation date
        self.chatRoomDataSource?.sortInPlace({ (item1, item2) -> Bool in
            let date1 = NSDate().formattedISO8601Date(item1._createdAt!)
            let date2 = NSDate().formattedISO8601Date(item2._createdAt!)
            return date1.compare(date2) == .OrderedDescending
        })
    }
    self.tableView.reloadData()
    return nil
}
}

```

[Download source code](#)

5. Enabling Amazon S3 feature from Mobile hub dashboard

In addition to creating a new chat room with one or more recipients and viewing all active chat rooms on a single screen, we want our application to allow participants to send and receive images to and from each other. To be able to send and receive files such as images, our app needs to have a defined storage space. This is where Amazon's Simple Storage Service, or **Amazon S3**, comes in. For Amazon S3 to be successfully incorporated into our application, we must first enable our application's User Data Storage feature from Mobile Hub.

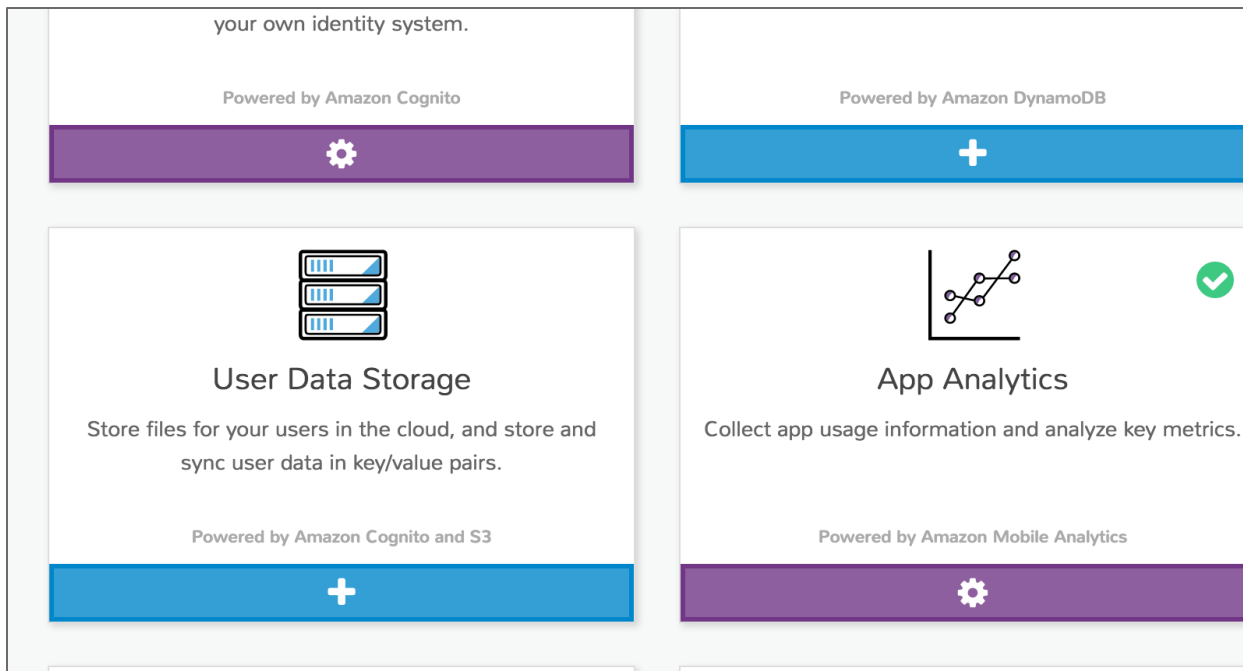
For more information about Amazon's Storage Service, visit [Amazon S3](#).

What does it cost?

Amazon S3 customers can get started with a **free tier**.

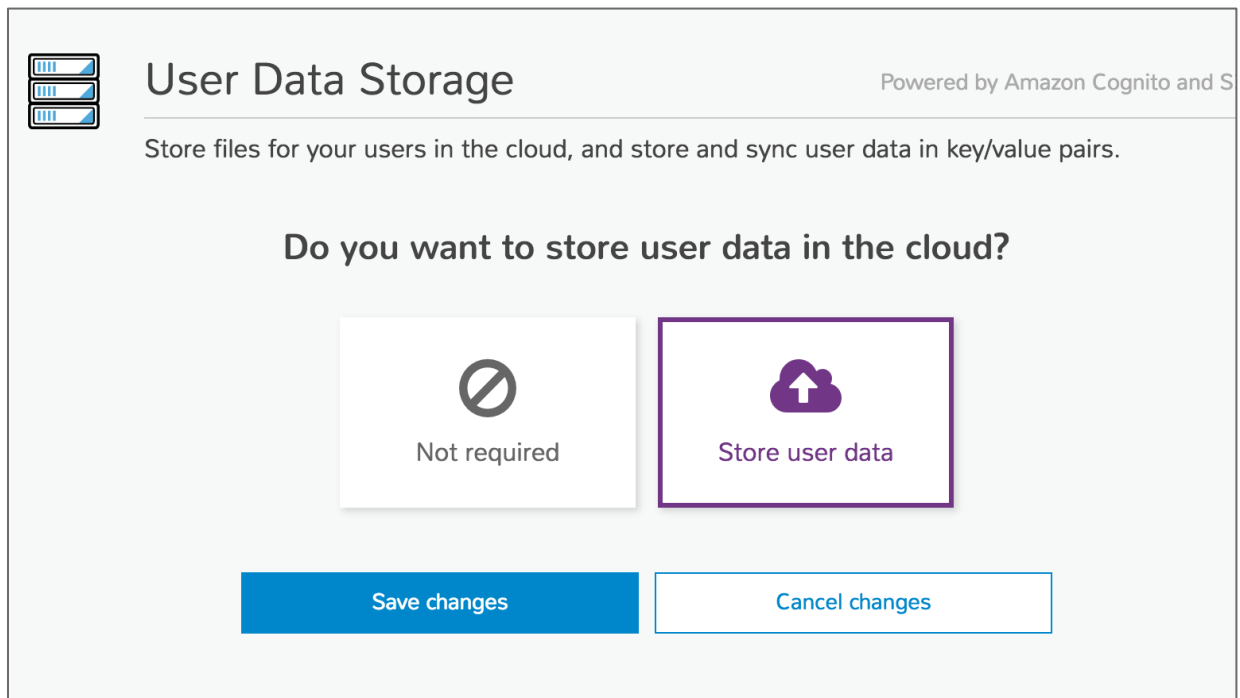
Visit [S3 Pricing](#) for more information about all the pricing tiers available.

- Go to the project's features from Mobile Hub and click on the + sign on the "User Data Storage" box from feature list



Screen 26: Selecting 'User Data Storage' feature

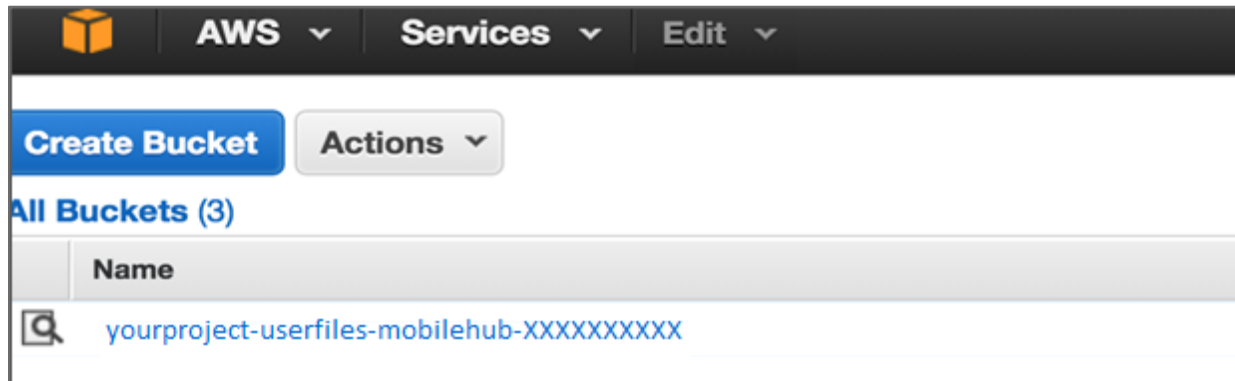
- Select "Store user data" from the two options available and "Save changes".



Screen 27: Configuring the data storage feature

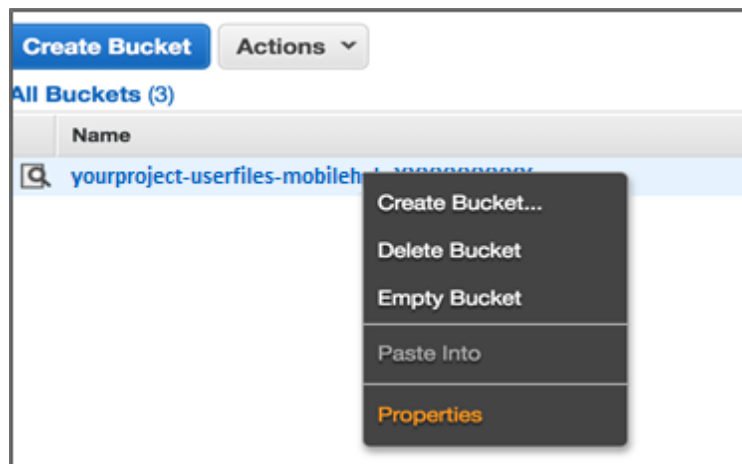
Adding policy to bucket public folder

- Open the Amazon S3 URL: <https://console.aws.amazon.com/s3>
- You will see a bucket name similar to `yourproject-userfiles-mobilehub-XXXXXXXXXX`. This bucket was created when you performed the previous step.



Screen 28: Project bucket on Amazon S3 console

- Right-click on the bucket name and go to “Properties”



- The Properties will now open in a panel on the right side. Select “Permissions” and select “Add bucket policy”.

Bucket: yourproject-userfiles-mobilehub-XXXXX... x

Bucket: yourproject-userfiles-mobilehub-XXXXXXXXXX

Region: US Standard

Creation Date: Mon Mar 28 19:46:34 GMT+500 2016

Owner: dev

Permissions

You can control access to the bucket and its contents using access policies. [Learn more.](#)

Grantee: Everyone

☒ List

☒ Upload/Delete

☒ View Permissions

☒

x

Edit Permissions

+ Add more permissions

Add bucket policy

Add CORS Configuration

Save

Cancel

Static Website Hosting

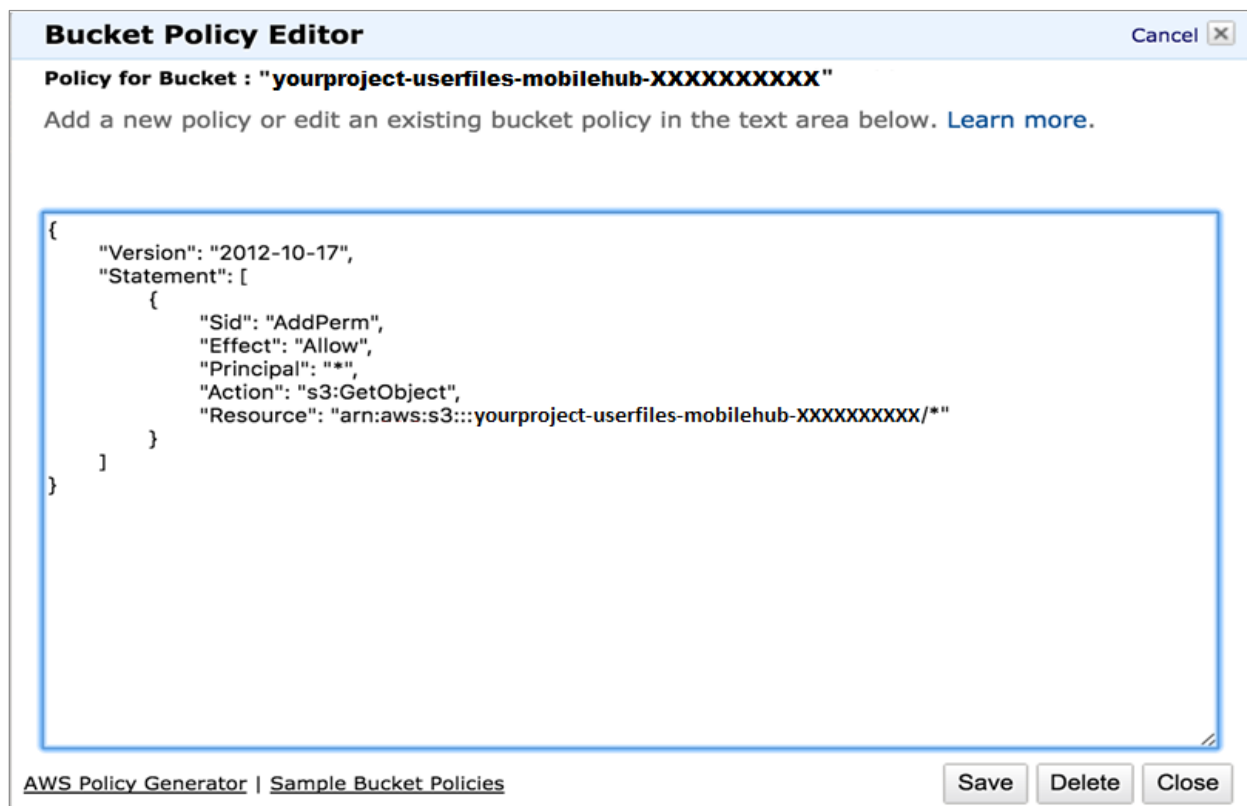
Logging

Events

Screen 29: Bucket "Properties" panel

28

- Enter the following permissions in the Policy Editor and click “Save”.



Screen 30: Bucket Policy Editor

Uploading an image from code

If you want the user to be able to send a picture to another user in the chat room, pick an image from the photo gallery using `UIImagePickerController` and pass the selected image as an `NSData` to this function. We will create and initialize conversation object and then call the uploading method

Following is the code illustrating it:

```
private func uploadWithData(data: NSData) {
    // convert current date into string
    let createdAt = NSString(format:@"%@",NSDate()) as String
    //set image name with current date
    let imageName = "\(createdAt).png"
    //set upload destination folder in key
    let key = "public/\(imageName)"
    // create & initialize Conversation object
    let conversation = createConversation()
    conversation._imageUrlPath = imageName
    conversation._message = "IMAGE"
    //get An instance of `AWSLocalContent` that represents data to be uploaded.
    let localContent = userFileManager.localContentWithData(data, key: key)
```

```

        //start uploading from this function
        uploadLocalContent(localContent , conversation: conversation)
    }
    private func uploadLocalContent(localContent: AWSLocalContent , conversation:Conversation) {
        showUploadingStatusView(true)

        localContent.uploadWithPinOnCompletion(false, progressBlock: {[weak self](content:
AWSLocalContent?, progress: NSProgress?) -> Void in
            // You can get uploading progress here ..
        }, completionHandler: {[weak self](content: AWSContent?, error: NSError?) -> Void in
            guard let strongSelf = self else { return }
            strongSelf.showUploadingStatusView(false)
            if let error = error {
                print("Failed to upload an object. \(error)")
            } else {
                content?.getRemoteFileURLWithCompletionHandler({ (url, error) in
                    // get full path of uploaded image
                    let imagePath = url?.absoluteString.componentsSeparatedByString("?").first
                    // store into conversation object
                    conversation._imageUrlPath = imagePath
                    print(url?.absoluteString)
                    // send conversaiton object to Conversation Table
                    strongSelf.sendMessageToServer(conversation)
                })
            }
        })
    }
}

```

[Download source code](#)

6. Sending Push Notifications to Chat Room Participants

Our application now allows users to create chat rooms, view all chat rooms, and send/receive images. Now we must enable it to send/receive messages. This is where **Amazon SNS** comes in. Amazon's Simple Notification Service, or SNS, is the technology that allows you to enable Push Notifications for your application. In the context of a messaging app like ours, Push Notifications would allow our user to send a message to an identified recipient, and notify said recipient that they have received a new message. To implement Push Notifications for our application, we must first enable it on our application's project on Mobile Hub.

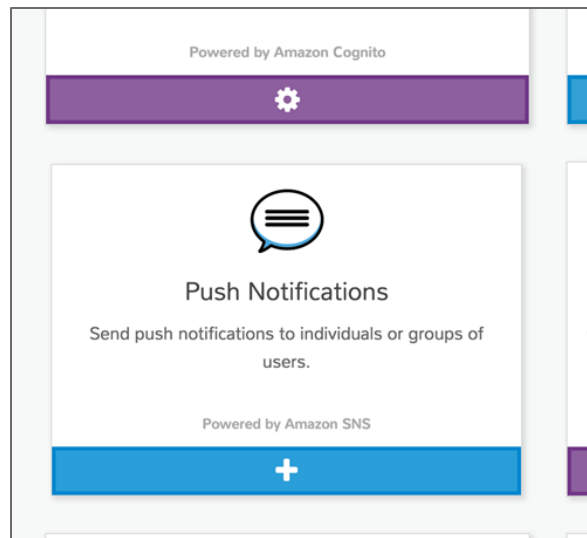
For more information about Amazon's notification service, visit [Amazon SNS](#)

What does it cost?

Amazon SNS customers can get started with a **free tier**.

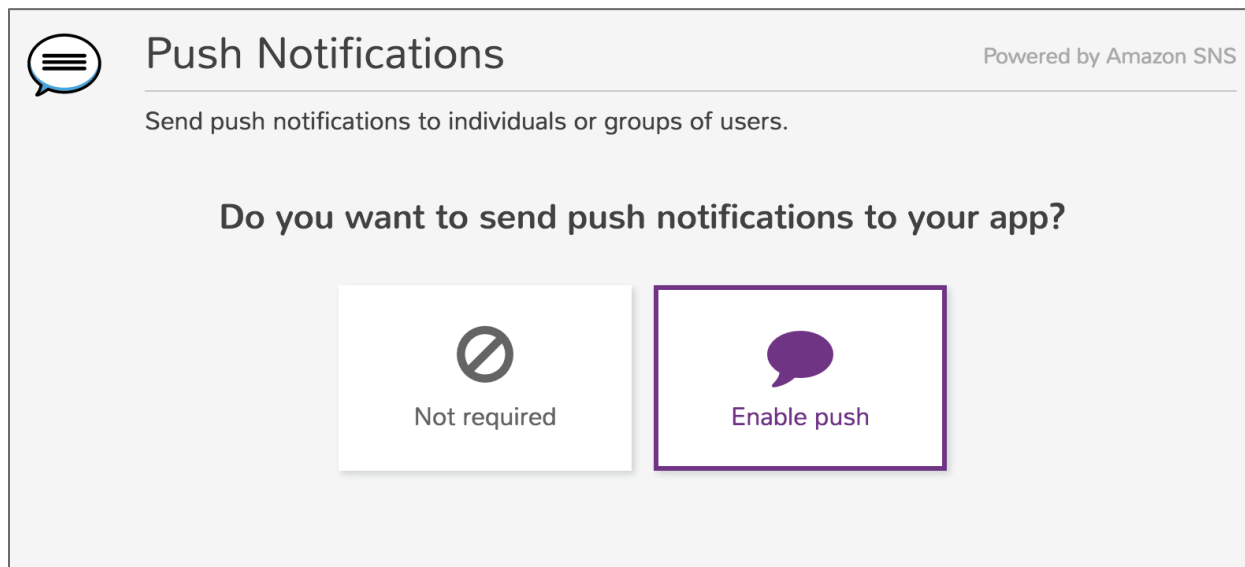
Visit [SNS Pricing](#) for more information about all the pricing tiers available

- Go to the project dashboard on Mobile Hub and click on the + button on the "Push Notifications" box.



Screen 31: Push notification feature

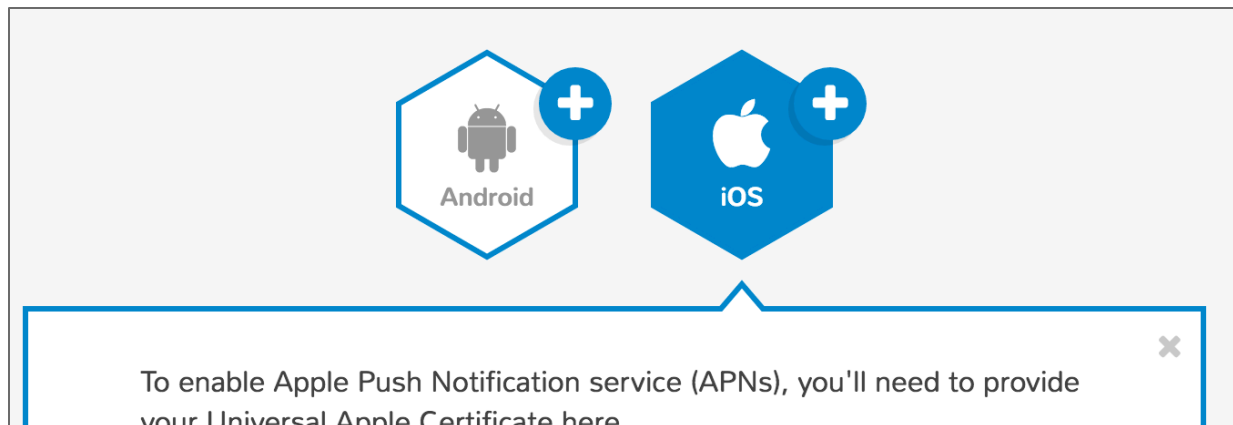
- The next screen would ask you whether you wish to enable push notifications for your app or not. Select “Enable Push”



Screen 32: Enabling Push Notifications for project

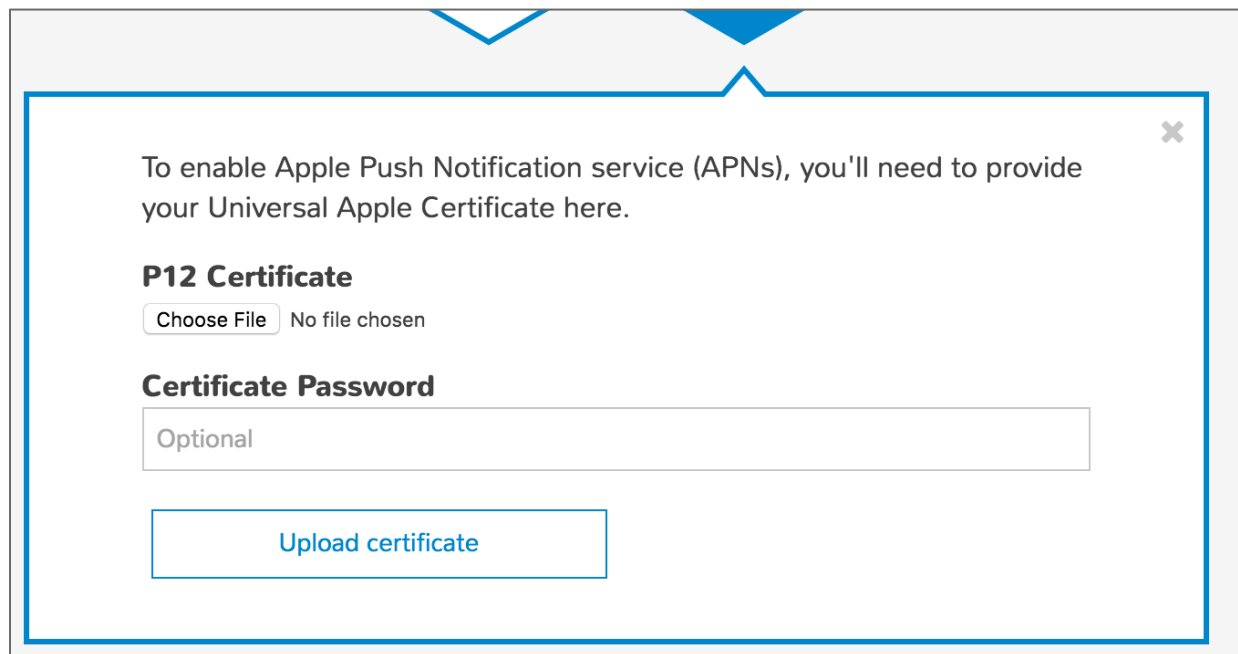
Once you enable the functionality, you will be required to select one of the two platforms to enable notifications for – Android or iOS.

- Since this app is iOS-based, select the iOS platform logo



Screen 33: Platform selection for SNS

- Browse your system for the P12 Certificate. Select the file and click on “Upload certificate”



Screen 34: Uploading certificate for APNs

- Finally, build the app and run the code.

Before we can send out push notifications to a particular recipient (and, as a consequence, the recipient can acknowledge receiving the push notification), we need to identify the endpoint that we wish to send the notifications to. A single user can use multiple devices, so our app needs to be able to distinguish each device as a different recipient; hence, we need to store each device’s identity, separately, in the database. Each device has a unique identity, referred to as targetArn. To be able to send out push notifications to a particular device, we need to get its user ID or targetArn.

How to save user device ID (targetArn)

- Get device `targetArn` using the following code

```
class func getDeviceArn() -> String? {
    let pushManager: AWSPushManager = AWSPushManager.defaultPushManager()
    if let _endpointARN = pushManager.endpointARN {
        return _endpointARN
    }else{
        pushManager.registerForPushNotifications()
    }
    return nil
}
```

- Right after the user is successfully logged in, update the current user's `pushTargetArn` field in the `UserProfile` table.

Now that we have the `targetArn`, we can send push notifications from the code.

Sending Push Notifications from Code

- Get `targetArn` of the recipients
- Set attributes for class `AWSSNSPublishInput.swift`. Attributes include `targetArn` (recipient device ID), `messageStructure` and `Message`.
- Pass the `AWSSNSPublishInput.swift` object to the `AWSSNS` method `publish`

Following is the code illustrating this:

```
func sendPush(conversation:Conversation) {
    let credentialsProvider = AWSCognitoCredentialsProvider(regionType:.USEast1,
                                                            identityPoolId:"us-east-1:bdad4021-
75b8-44c1-a079-3b6e9e565b47")
    let configuration = AWSServiceConfiguration(region:.USEast1,
credentialsProvider:credentialsProvider)
    AWSServiceManager.defaultServiceManager().defaultServiceConfiguration = configuration

    for userProfile in recipientUsers {
        //skip to current user
        if userProfile._userId == AWSIdentityManager.defaultIdentityManager().identityId {
            continue
        }

        if let targetArns = userProfile._pushTargetArn {
            for deviceTargetArn in targetArns {
                do {
                    let sns = AWSSNS.defaultSNS()
                    let request = AWSSNSPublishInput()
                    request.messageStructure = "json"
                    let senderName = AWSIdentityManager.defaultIdentityManager().userName

                    let devicePayload = ["default": "Message sent by \(senderName!)",
"APNS_SANDBOX": "{\"aps\":{\"alert\": \"Message sent by \(senderName!)\",\"sound\":\"default\"",
```

```
        let jsonData = try NSJSONSerialization.dataWithJSONObject(devicePayload,
options: NSJSONWritingOptions.init(rawValue: 0))

        request.subject = "Message Sent By \(senderName)"
        request.message = NSString(data: jsonData, encoding::NSUTF8StringEncoding)

as? String

        request.targetArn = deviceTargetArn

        sns.publish(request).continewithBlock { (task) -> AnyObject! in
            print("error \(task.error), result: \(task.result)")
            return nil
        }
    } catch let parseError {
        print(parseError) // Log the error thrown by `JSONObjectWithData`
    }
}
}
```

- Get `chatRoomId` from the data sent to receiving device
- Load conversation

```
func application(application: UIApplication, didReceiveRemoteNotification userInfo: [NSObject :
AnyObject]) {
    AWSMobileClient.sharedInstance.application(application, didReceiveRemoteNotification:
userInfo)

    if let chatRoomId = userInfo["chatRoomId"] as? String {
        print(chatRoomId)
        ChatDynamoDBServices().getChatRoomWithChatRoomId(chatRoomId).continueWithBlock({ (task) ->
AnyObject? in
            if let chatRoom = task.result as? ChatRoom {
                print(chatRoom)
                var defaultMessage = ""

                if let _defaultMessage = userInfo["aps"]!["alert"]! as String! {
                    defaultMessage = _defaultMessage
                }
                self.showMessageInConversation(chatRoom, defaultMessage: defaultMessage)
            }
            return nil
        })
    }
}
```

```

    }

    func showMessageInConversation(chatRoom:ChatRoom , defaultMessage:String) {
        guard let _navigationController = self.window?.rootViewController as? UINavigationController
    else{
        showPushAlert(defaultMessage)
        return
    }

        guard let conversationVC = _navigationController.topViewController as?
ConversationViewController where conversationVC.selectedChatRoom!._chatRoomId ==
chatRoom._chatRoomId else{

            showPushAlert(defaultMessage)
            return
        }

        conversationVC.selectedChatRoom = chatRoom
        conversationVC.loadRecipientsAndConversations(false)
        print(conversationVC)
    }

    func showPushAlert(defaultMessage:String) {

        dispatch_async(dispatch_get_main_queue(),{

            let alertController = UIAlertController(title: "Message", message: defaultMessage,
preferredStyle: .Alert)
            let doneAction = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
            alertController.addAction(doneAction)
            self.window?.rootViewController?.presentViewController(alertController, animated: true,
completion: nil)

        })
    }

```

Conclusion

In this tutorial we have shown you how you can use multiple AWS services to quickly and easily build an iOS-based messaging application. AWS technologies and services used in this tutorial included:

- AWS Mobile Hub
- Amazon Cognito Identity
- Amazon Mobile Analytics
- Amazon DynamoDB
- Amazon S3
- Amazon SNS

With the information of this tutorial in hand, you can begin to envision much richer and advance mobile experiences based on AWS services.