Mobile App Backend Tutorial for Hybrid Apps

AWS Tutorial

Sean Senior

Leo Drakopoulos

December 2016

Contents

Overview 3
Architecture 3
Application Flow 4
Prerequisites 5
What We'll Cover 5
Step 1. Create the Amazon Cognito user pool 6
Step 2. Create the Amazon DynamoDB table for task data 8
Step 3. Create the Amazon API Gateway and AWS Lambda function 8
Create the deployment package 9
Create the execution IAM role 14
Create the Lambda function 15
Create the RESTful API in Amazon API Gateway 16
Step 4. Create the Git2It App 19
Configure the sample source code 21
Step 5. Enter Test Data Into Git2It 23
Create a Git2It user account 23
Create a task 24
Update and delete a task 25
Step 6. Test the Git2It app 26
Deploy the app on emulators 26
Deploy the app on real devices 27
Appendix A: Git2It App Code 29
User Management 29
CRUD Operations 33
Document Revisions 38

Overview

Building a mobile app backend to handle user management, push notifications, and integration with social networking services can be a time-consuming process for developers. Each of these services has its own API that must be incorporated into the app, which can add complexity to the development process. Amazon Web Services (AWS) makes it easier for developers to add and configure a cloud-based backend for their mobile apps. With AWS, developers can focus on creating great app experiences instead of building and configuring a backend.

AWS provides a suite of services such as Amazon Cognito, Amazon API Gateway, AWS Lambda and Amazon DynamoDB that enable you to build a flexible, scalable, managed backend for your app. This guide walks you through the process of adding and configuring a backend for a sample Ionic hybrid mobile app called Git2It.

Ionic is a free and open source library of mobile-optimized HTML, CSS, and JS components and tools for building highly interactive native and progressive web apps optimized for AngularJS. You can build apps with the Ionic framework and distribute those apps through native app stores. Leverage Apache Cordova to install your app on devices.

Architecture

Following this tutorial, you will build the following environment in the AWS Cloud.

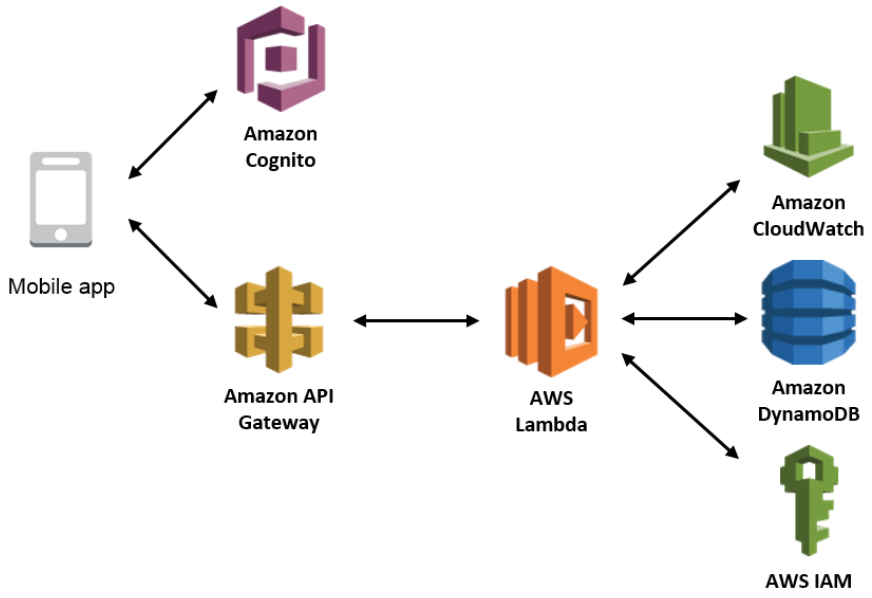


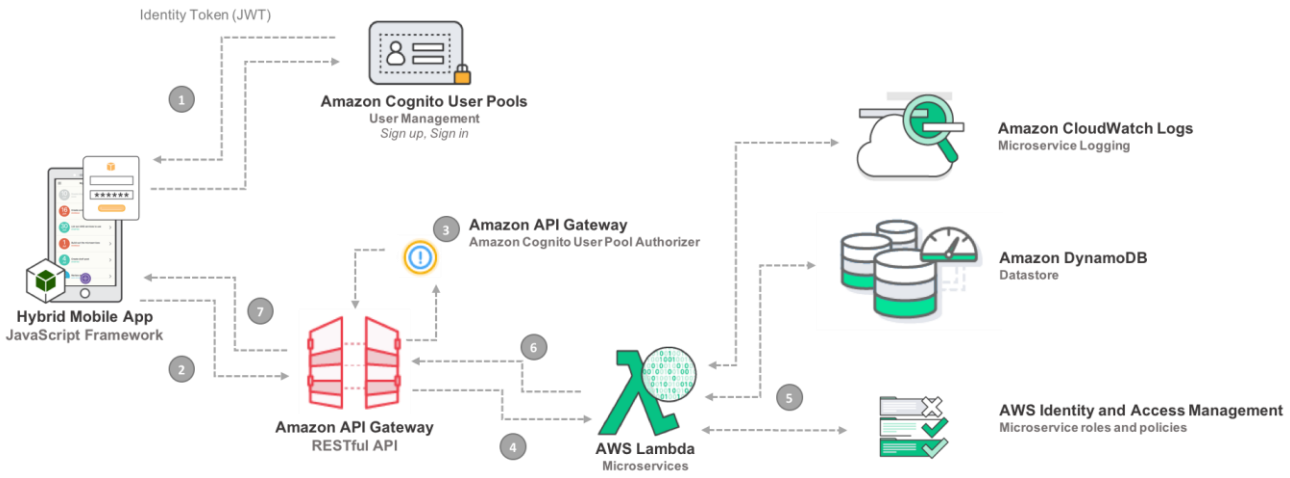
Figure 1: Mobile backend architecture on AWS

The backend leverages Amazon Cognito user pools which can simplify user authentication and authorization, and Amazon API Gateway which can make it easier for mobile developers to create, publish, maintain, monitor, and secure APIs at any scale. For this tutorial, you use Amazon API Gateway to create custom RESTful APIs that trigger the AWS Lambda function which performs CRUD operations for tasks that are stored in the Amazon DynamoDB table.

This backend adds user directory, authorization, traffic management, monitoring, and analytics functionality to your hybrid mobile app. The same general approach can be applied to any framework built around JavaScript such as PhoneGap/Cordova or Framework 7.

Application Flow

The following diagram outlines the architectural flow of a RESTful mobile backend for a mobile app.



1. When a user signs in to the mobile app, the user's credentials are sent to the Amazon Cognito user pool for authentication. After successful authentication, Amazon Cognito returns an ID token to the app.
2. The mobile app sends HTTPS requests to the Amazon API Gateway RESTful interface with the Amazon Cognito user pool ID token in the authorization header.
3. An Amazon Cognito user pool authorizer associated with the API Gateway RESTful API validates that the token in the authorization header is an authenticated user. If it is not a valid user, the authorizer will return a 401 HTTP code. If it is a valid user, an AWS Identity and Access Management (IAM) policy is returned to permit the request to access the appropriate API resource.

4. The API Gateway invokes the AWS Lambda microservice function associated with the requested API resource.
5. AWS Lambda assumes appropriate IAM role to execute a defined task, such as accessing user-specific data in Amazon DynamoDB. All requests that Lambda handles are recorded and stored through Amazon CloudWatch Logs.
6. Lambda returns the results in an HTTP-formatted response to the RESTful API in API Gateway.
7. API Gateway returns the results to the mobile app.

Prerequisites

Before you start, you must install the Ionic framework. You can use the Node Package Manager (npm) to install it. For more information on installing and using Ionic, please see Ionic's documentation on Installing Ionic and its Dependencies.

To build the Ionic mobile app and its backend on AWS, you need an AWS account with an administrator user. You also need to configure the AWS Command Line Interface (AWS CLI). For instructions, see Set Up an AWS Account and the AWS CLI.

You will also need Node.js (v4.3) to create the AWS Lambda function that powers the backend. For instructions on downloading and installing Node.js, see the Node.js website.

To test the app on emulators and real devices, you must install XCode and the Android SDK on your workstation. You can download XCode here. You can download the Android SDK here.

What We'll Cover

The procedure for building this backend on AWS consists of the following steps. For detailed instructions, follow the links for each step.

Step 1. Create the Amazon Cognito user pool

- Create and configure an Amazon Cognito user pool.

Step 2. Create the Amazon DynamoDB table for task data

- Create an Amazon DynamoDB table to store a user's task data.

Step 3. Create the Amazon API Gateway and AWS Lambda function

- Create a deployment package.
- Create the AWS Identity and Access Management (IAM) roles.

- Create the AWS Lambda function and test it.
- Create the RESTful API in API Gateway.

Step 4. Create the Git2It app

- Create the Git2IT app on the Ionic framework.

Step 5. Enter test data into the Git2It app

- Sign in to the Git2It app and create a task.

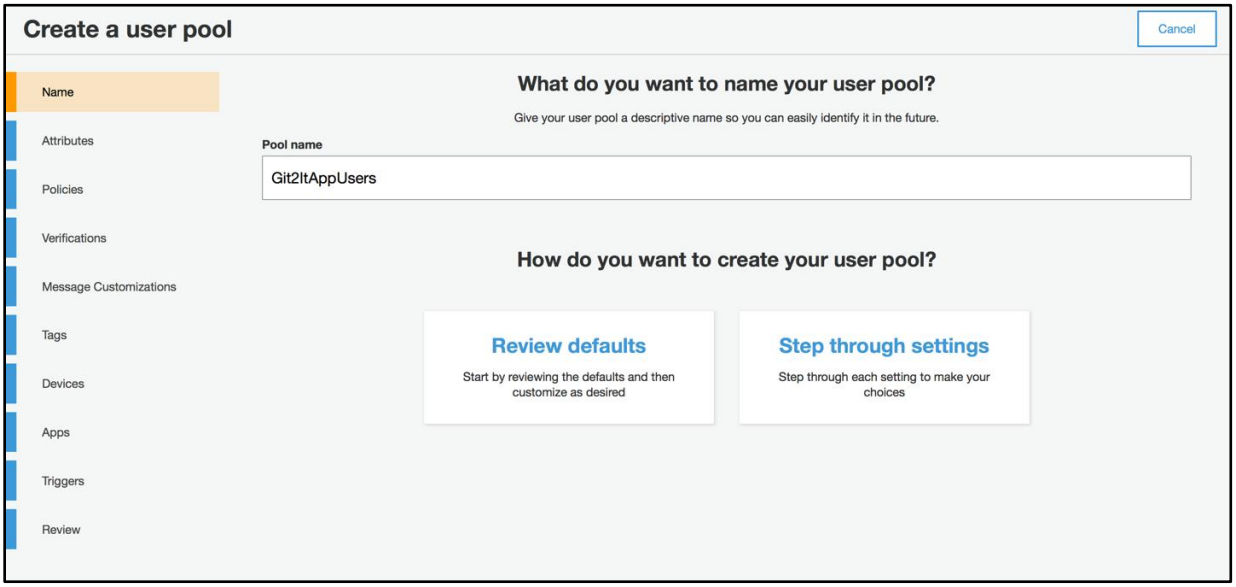
Step 6. Test the Git2It app

- Verify that the backend works correctly.

Step 1. Create the Amazon Cognito user pool

Once your environment is configured, create an Amazon Cognito user pool for your user directory.

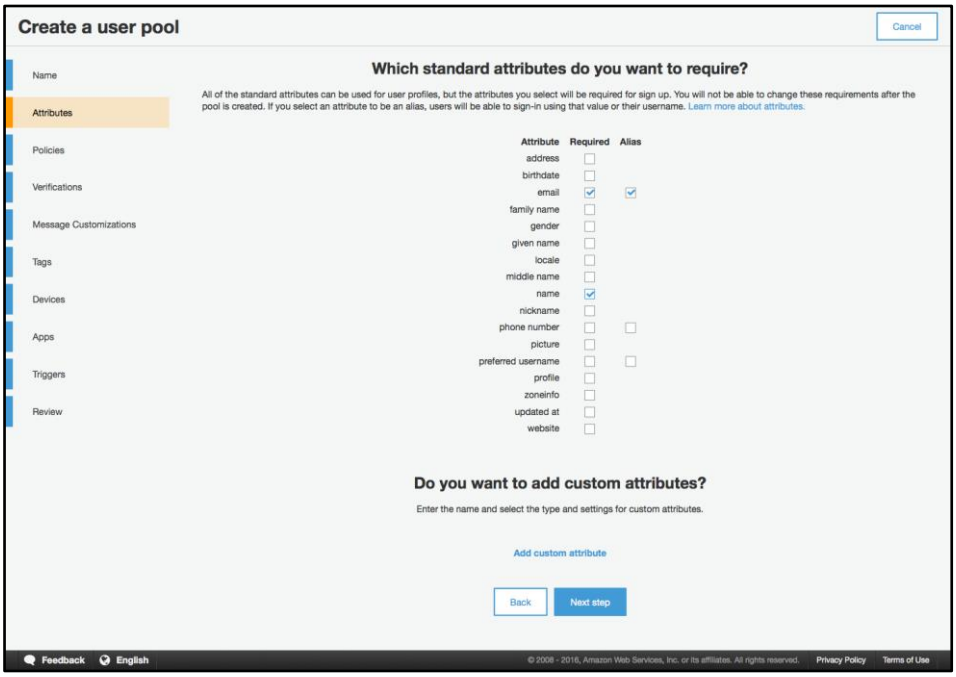
1. Sign in to the Amazon Cognito console.
2. On the **Your User Pools** page, choose **Create a User Pool**.
3. In the **Pool name** field, type `Git2ItAppUsers`.



4. Choose **Step through settings**.
5. For the **email** attribute, select **Required** and **Alias**.

Selecting **Alias** for the **email** attribute allows users to sign up and sign in with an email address instead of a username.

6. For the **name** attribute, select **Required**.



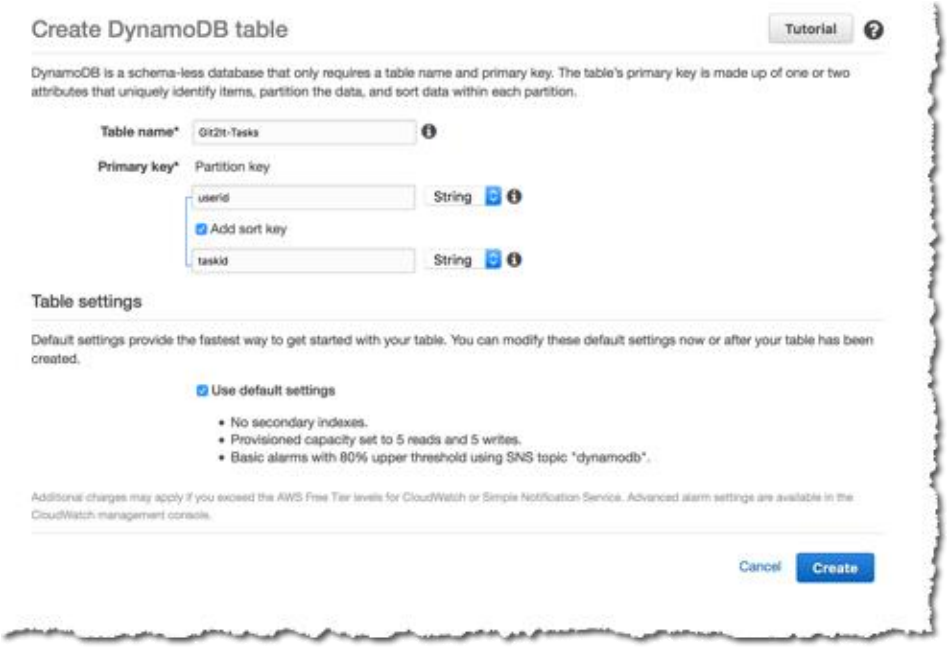
7. Choose **Next step**.
8. Under **What password strength do you want to require?**, clear **Require special character**.
9. Select **Next step** until you get to the **Message Customizations** tab.
10. Under **Do you want to customize your email verification messages?**, type the following values:
    - **Email subject:** Your Git2It verification code
    - **Email message:** Your Git2It verification code is {`####`}.
11. Select **Next step** until you get to the **Devices** tab.
12. Under **Do you want to remember your user's devices?**, verify that **No** is selected.
13. Select **Next step**.
14. Choose **Add an app**.
    Adding an app gives the Git2ItApp permission to call APIs that do not have an authenticate user.
15. Under **App name**, type `Git2ItApp`.

16. Clear **Generate client secret**.
17. Choose **Create app**.
18. Select **Next step** until you get to the **Review** tab.
19. Review your settings and select **Create pool**.

Step 2. Create the Amazon DynamoDB table for task data

Create an Amazon DynamoDB table to store the data from your user's tasks.

1. In the navigation pane of the Amazon DynamoDB console, select **Tables**.
2. Select **Create table**.
3. For **Table name**, type `Git2It-Tasks`.
4. For **Partition key**, type `userid`.
5. Select **Add sort key** and type `taskid`.



The default table setting has a provisioned capacity of five reads and five writes.

Step 3. Create the Amazon API Gateway and AWS Lambda function

This backend uses AWS Lambda to host the CRUD logic that Amazon API Gateway invokes in response to HTTPS requests from our hybrid mobile app. This allows us to put minimal logic in the app, making it easier to scale and update.

Create the deployment package

Create a working directory to develop your deployment package. This backend uses three additional libraries (moment.js, node-uuid and underscore.js) outside of the AWS SDK for JavaScript.

1. Navigate to your working directory, open a text editor, and copy the following code.

```
{
    "name": "TaskMicroservice",
    "description": "A Lambda function for CRUD operations for tasks
in the Git2It Ionic mobile app",
    "main": "handler.js",
    "author": {
        "name": "AWS-Sample"
    },
    "version": "0.0.1",
    "private": "true",
    "dependencies": {
        "node-uuid": "*",
        "moment": "*",
        "underscore": "*"
    }
}
```

2. In your working directory, install the additional Node.js libraries for our CRUD microservices function:

```
npm install
```

3. In your working directory, open a text editor and copy the following code.

```
'use strict';
console.log('Loading function');

let moment = require('moment');
let uuid = require('node-uuid');
let _ = require('underscore');
let AWS = require('aws-sdk');

let creds = new AWS.EnvironmentCredentials('AWS'); // Lambda provided
credentials
const dynamoConfig = {
    credentials: creds,
    region: process.env.AWS_REGION
};
const docClient = new AWS.DynamoDB.DocumentClient(dynamoConfig);
const ddbTable = 'Git2It-Tasks';
```

```
/**
 * Provide an event that contains the following keys:
 *
 *   - resource: API Gateway resource for event
 *   - path: path of the HTTPS request to the microservices API call
 *   - httpMethod: HTTP method of the HTTPS request from microservices API
call
 *   - headers: HTTP headers for the HTTPS request from microservices API
call
 *   - queryStringParameters: query parameters of the HTTPS request from
microservices API call
 *   - pathParameters: path parameters of the HTTPS request from
microservices API call
 *   - stageVariables: API Gateway stage variables, if applicable
 *   - body: body of the HTTPS request from the microservices API call
 */

module.exports.handler = function(event, context, callback) {

    console.log(event);
    let _response = "";

    let invalid_path_err = {
        "Error": "Invalid path request " + event.resource + ', ' +
event.httpMethod
    };

    if (event.resource === '/tasks' && event.httpMethod === "GET") {
        console.log("listing the tasks for a user");
        var params = {
            TableName: ddbTable,
            KeyConditionExpression: 'userid = :uid',
            ExpressionAttributeValues: {
                ':uid': event.queryStringParameters.userid
            }
        };

        docClient.query(params, function(err, resp) {
            if (err) {
                console.log(err);
                _response = buildOutput(500, err);
                return callback(_response, null);
            }

            if (resp.Items) {
                switch (event.queryStringParameters.filter) {
                    case "week":
                        resp.Items = _.filter(resp.Items, function(task) {
                            return moment(task.datedue).utc() >=
moment().utc().startOf('week') &&
                                moment(task
```

```
                                    .datedue).utc() <=
                                moment().utc().endOf('week');
                        });
                        break;
                    case "today":
                        resp.Items = _.filter(resp.Items, function(task) {
                            return moment(task.datedue).utc() >=
moment().utc().set('hour', '00').set(
                                    'minute',
                                    '00')
                                .set('second', '00').set('millisecond',
'00') && moment(task.datedue).utc() <=
                                moment().utc().set('hour',
'23').set('minute', '59');
                        });
                        break;
                    case "doingnow":
                        resp.Items = _.filter(resp.Items, function(task) {
                            return task.stage == "Started";
                        });
                        break;
                    case "done":
                        resp.Items = _.filter(resp.Items, function(task) {
                            return task.stage == "Done";
                        });
                        break;
                    default:
                        resp.Items = resp.Items;
                        break;
                }

                if (resp.Items) {
                    resp.Count = resp.Items.length;
                } else {
                    resp.Count = 0;
                }
            }

            _response = buildOutput(200, resp);
            return callback(null, _response);
        });

    } else if (event.resource === '/tasks/{taskid}' && event.httpMethod ===
"POST") {
        console.log("creating a new task for a user");

        let task = JSON.parse(event.body);

        //create unique taskid for the new task
        task.taskid = uuid.v4();
        //set the created datetime stamp for the new task
        task.createdAt = moment().utc().format();
```

```
        let params = {
            TableName: ddbTable,
            Item: task
        };

        docClient.put(params, function(err, data) {
            if (err) {
                console.log(err);
                _response = buildOutput(500, err);
                return callback(_response, null);
            }
            _response = buildOutput(200, task);
            return callback(null, _response);
        });

    } else if (event.resource === '/tasks/{taskid}' && event.httpMethod ===
"PUT") {
        console.log("updating a task for a user");

        let task = JSON.parse(event.body);

        let params = {
            TableName: ddbTable,
            Item: task
        };

        docClient.put(params, function(err, data) {
            if (err) {
                console.log(err);
                _response = buildOutput(500, err);
                return callback(_response, null);
            }
            _response = buildOutput(200, task);
            return callback(null, _response);
        });

    } else if (event.resource === '/tasks/{taskid}' && event.httpMethod ===
"DELETE") {
        console.log("delete a user's task");

        let params = {
            TableName: ddbTable,
            Key: {
                userid: event.queryStringParameters.userid,
                taskid: event.pathParameters.taskid
            }
        };
        docClient.delete(params, function(err, data) {
            if (err) {
                console.log(err);
                _response = buildOutput(500, err);
```

```
                return callback(_response, null);
            }
            _response = buildOutput(200, data);
            return callback(null, _response);
        });

    } else if (event.resource === '/tasks/{taskid}' && event.httpMethod ===
"GET") {
        console.log("get a user's task");

        let params = {
            TableName: ddbTable,
            Key: {
                userid: event.queryStringParameters.userid,
                taskid: event.pathParameters.taskid
            }        };

        docClient.get(params, function(err, data) {
            if (err) {
                console.log(err);
                _response = buildOutput(500, err);
                return callback(_response, null);
            }
            _response = buildOutput(200, data);
            return callback(null, _response);
        });

    } else {
        _response = buildOutput(500, invalid_path_err);
        return callback(_response, null);
    }

};

/* Utility function to build HTTP response for the microservices output */
function buildOutput(statusCode, data) {

    let _response = {
        statusCode: statusCode,
        headers: {
            "Access-Control-Allow-Origin": "*"
        },
        body: JSON.stringify(data)
    };

    return _response;
};
```

4. Save the file as `handler.js`. Then, zip your entire working directory and save it as `TaskMicroservices.zip`.

Create the execution IAM role

Create an AWS Identity and Access Management (IAM) role that allows our AWS Lambda function to perform CRUD operations on our Amazon DynamoDB table and create logs in Amazon CloudWatch.

1. Sign in to the AWS IAM console.
2. Complete the steps in Creating a Role to Delegate Permissions to an AWS Service using the following information:
   - For **Role name**, use a name that is unique within your AWS account. For example, `git2it-lambda-execution-role`.
   - For **Select Role Type**, choose **AWS Service Roles**, **AWS Lambda**.
   - After you create the role, update the role and attach the following inline permissions policy to the role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:[your-region]:*:table/Git2It-Tasks"
      ]
    },
    {
      "Sid": "",
      "Resource": [
        "arn:aws:logs:*:[account-id]:log-
group:/aws/lambda/TaskMicroservice:*"
      ],
      "Action": [
        "logs:CreateLogGroup",        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
```

```
}
```

Note: Replace `[your-region]` and `[account-id]`, with the AWS Region you are working with for this project and your account ID respectively.

Note the role Amazon Resource Name because you need it to create the Lambda function.

Create the Lambda function

1. In the AWS CLI, create the function:

```
$ aws lambda create-function \
--region aws-region \
--function-name TaskMicroservice  \
--zip-file fileb://file-path/TaskMicroservice.zip \
--role execution-role-arn \
--description "Microservice logic for Git2It task app" \
--handler handler.handler \
--runtime nodejs4.3 \
--memory-size 512 \
--timeout 10
```

2. Complete the steps in [Invoke the Lambda function Manually and Verify the Results, Logs, and Metrics].
   In **Sample event template**, choose **Hello World** and then replace the data using the following code.

```
{
    "resource": "/tasks",
    "httpMethod": "GET",
    "pathParams": "{}",
    "queryStringParameters": {"userid":"testuser_test_com"},
    "payload": {}
}
```

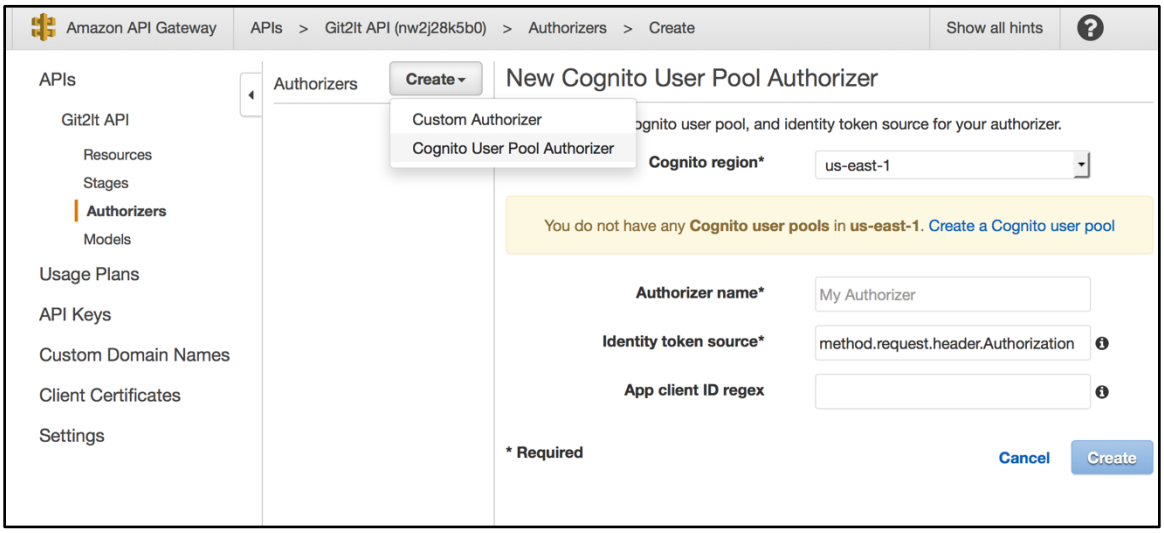
   Expect the following response.

```
{
  "statusCode": 200,
  "headers": {
    "Access-Control-Allow-Origin": "*"  },
  "body": "{\"Items\":[],\"Count\":0,\"ScannedCount\":0}"
}
```

Create the RESTful API in Amazon API Gateway

Create a RESTful API in Amazon API Gateway and associate the AWS Lambda function with each API operation. When an HTTPS request is sent to an API operation, Amazon API Gateway will invoke the `TaskMicroservice` Lambda function.

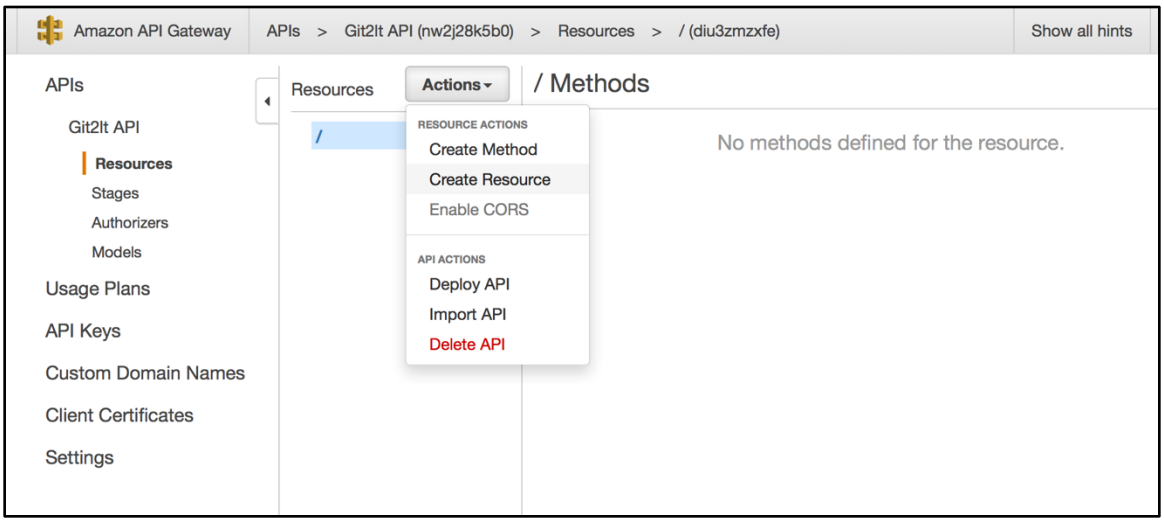
1. Create the Git2App API:
   a. In the Amazon API Gateway console, select **Create API.**
   b. Select **New API**.
2. Create an Amazon Cognito user pool authorizer in the Git2It API.
   The Amazon Cognito user pool is integrated with a RESTful API in Amazon API Gateway as an operation authorizer. When calling operations with an authorizer enabled, the API client includes the user's identity token provisioned from the Amazon Cognito user pool in the request headers.
   a. In the navigation pane under **API**, **Git2ItAPI**, select **Authorizers**.
   b. Select **Create, Cognito User Pool Authorizer**.



   c. Complete the required fields.
      - **Cognito region:** `us-east-1`
      - **Cognito user pool:** `Git2ItAppUsers`
      - **Authorizer name:** `Git2ItAppUsers`
      - **Identity token source:** `method.request.header.Authorization`

Identity token source is automatically set to `method.request.header.Authorization`. By using the default, Authorization will be the name of the incoming request header to contain an API caller's identity token.

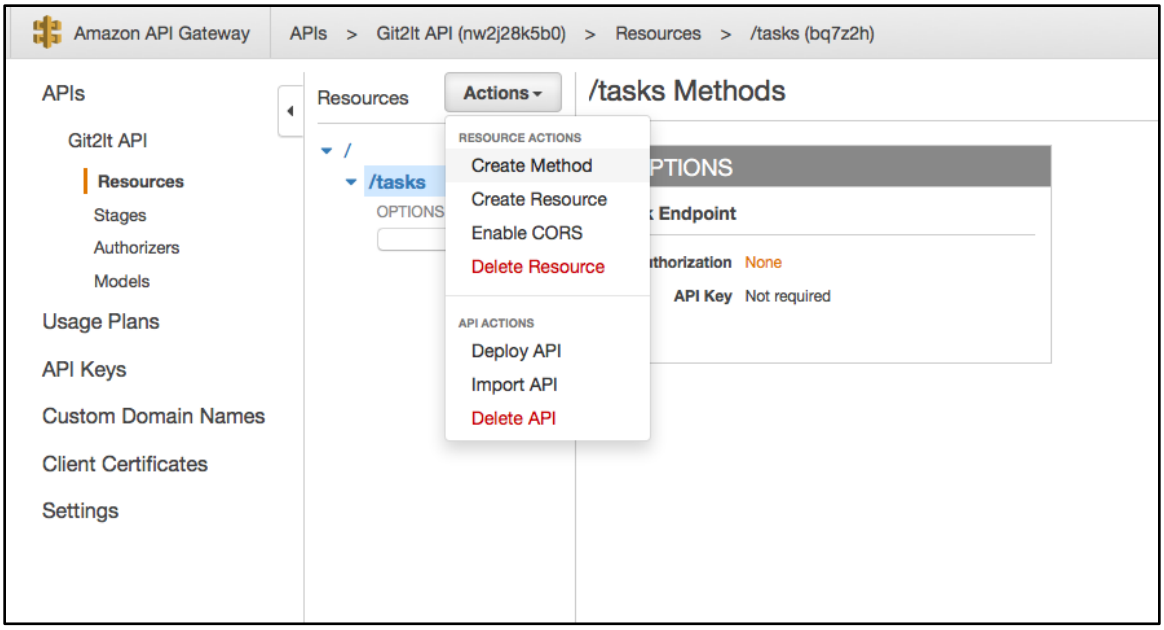
3. Select **Create**.
4. In the navigation pane, under **API**, **Git2ItAPI**, select **Resources**.
5. Select **Actions, Create Resource**.



6. Complete the fields:
   a. **Resource name:** `tasks`
   b. **Resource path:** `tasks`
7. Select **Enable API Gateway CORS**.

> **Important:** API Gateway cross-origin resource sharing (CORS) is required for Git2It API's resources to receive requests from external domains. Enabling CORS allows the API to respond to the OPTIONS preflight request with the proper CORS-required response headers (`Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`, `Access-Control-Allow-Origin`). When CORS is enabled, the API Gateway will provision the OPTIONS method for your resource.

8. Select **Create resource**.
9. Select **Actions, Create Method**.



10. Select **ANY**.
11. Choose the integration point for the **ANY** method.
    a. For **Integration type**, select **Lambda function**.
    b. Select **Use Lambda Proxy integration**.
       Lambda Proxy integration applies a default mapping template to send the entire request to your functions, and it automatically maps Lambda output to HTTP responses.
    c. For **Lambda Region**, select the same region where you deployed the Lambda function.
    d. For **Lambda Function**, type `TaskMicroservice`.
12. Select **Save**.
13. When prompted to **Add Permission to Lambda Function**, choose **OK**.
14. Choose **Method Request**.
15. Choose the pencil icon and select **Cognito user pool authorizers, Git2ItAppUsers**.
16. To save your selection, choose the check mark icon.
17. Select **Actions, Create Resource**.
18. Complete the fields:
    a. **Resource name:** `taskid`

b. **Resource path:** `{taskid}`

> **Note:** The curly brackets around `{taskid}` in the URI path definition indicate that `taskid` will be treated as a path variable within the URI.

19. Select **Enable API Gateway CORS**.
20. Select **Create Resource**.
21. To create an **ANY** method for the `tasks/{taskid}` resource, repeat steps 9 – 16.
22. Select **Actions, Deploy API**.
23. For **Stage name**, type `Prod`.
24. Select **Deploy**.
25. In the API Gateway console, verify that you see the newly created Git2ItApp resource.

Step 4. Create the Git2It App

1. In the AWS CLI, generate an Ionic app skeleton:

```
ionic start git2it blank
cd git2it
```

Once you run the command, you will see the following directory structure.

For more information about the structure of Ionic projects, please visit [Ionic Concepts: Structure].

2. Install the sample source code.

   a. In the `git2it` working directory, delete the `www` directory.

b. Download the source code for the sample Git2It mobile app to the `git2it` working directory.

c. Unzip the `git2it_sample_app.zip` file.

d. In the `git2it` working directory, verify that you have a `www` directory that contains the UI and logic for the sample mobile app.

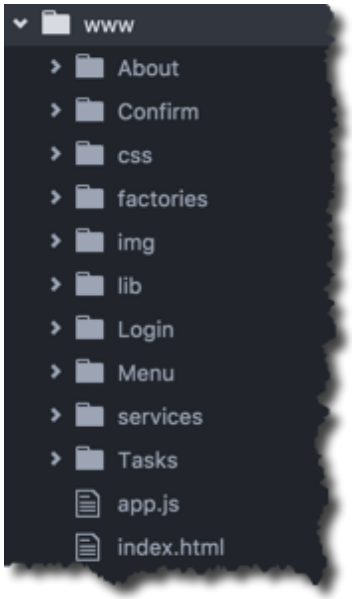


Figure 2. Sample source code structure

Directory/File	Description
./About/*	About view and controller for the mobile app
./Confirm/*	Sign-up email confirmation view and controller for the mobile app. This view provides the integration point for the email confirmation required for the Amazon Cognito user pool we configured.
./css/*	Custom styles for the mobile app
./factories/utils.js	A helper factory to access local storage of the webview where the mobile app UI runs. The local storage is used to store the temporary AWS Security Token Services tokens.
./factories/TaskFactory.js	A factory for managing CRUD operation interfaces for the task data in the mobile app
./img/*	Image files used by the mobile app
./lib/angular-messages/*	An open source, AngularJS module that provides enhanced support for displaying messages within templates
./lib/angular-resource/*	An open source, AngularJS module that interacts with RESTful server-side data sources

./lib/aws-sdk/*	AWS SDK for JavaScript library
./lib/congnito/*	The supporting JavaScript libraries required to setup the Amazon Cognito Identity SDK for JavaScript as outlined in Setting up the AWS SDK for JavaScript in the Browser
./lib/ionic/*	The Ionic framework libraries
./lib/jsbn/*	An open source, library for BigInteger computations
./lib/moment/*	An open source, full featured date library for parsing, validating, manipulating, and formatting dates
./lib/sjcl/*	Stanford JavaScript Crypto Library, an open source JavaScript library that is used to build a secure, powerful, fast, small, easy-to-use, cross-browser library for cryptography in JavaScript
./lib/underscore/*	An open source JavaScript library that provides useful functional programming helpers without extending any built-in objects
./lib/aws-variables.js	Configuration file for the Git2It Ionic mobile app to store common settings used across the app
./Login/*	Sign-up view and controller for the mobile app. This view provides the integration point for the sign-up of users in the Amazon Cognito user pool we configured.
./Menu/*	Side menu view for the mobile app
./services/AuthService.js	A service for managing the mobile app interfaces with the Amazon Cognito user pool
./Tasks/*	Task list view, individual task view and controllers for the mobile app
./app.js	Initialization and configuration of the AngularJS/Ionic application for the mobile app
./index.html	Root view container for the mobile app

Configure the sample source code

1. In the `lib` directory, open `aws-variables.js`.

```
var APIG_ENDPOINT = 'API-GATEWAY-ENDPOINT-FOR-GIT2IT-API';
var YOUR_USER_POOL_ID = "AMAZON-COGNITO-USER-POOL-ID";
var YOUR_USER_POOL_CLIENT_ID = "AMAZON-COGNTIO-USER-POOL-APP-CLIENT-
ID";
var APP_VERSION = '0.4.0';
```

2. To find the Amazon Cognito user pool ID, copy the following command, paste it into the AWS Command Line Interface (AWS CLI), and press **Enter**.

```
$ aws cognito-idp list-user-pools \
--max-results 60
```

3. In the response, under `User Pools`, note the `ID` value.

```
{
    "UserPools": [
        {
            "CreationDate": 1467998892.357,
            "LastModifiedDate": 1467998892.357,
            "LambdaConfig": {},
            "Id": "AMAZON-COGNITO-USER-POOL-ID",
            "Name": "Git2ItAppUsers"        }
    ]
}
```

4. In `aws-variables.js`, replace the `AMAZON-COGNITO-USER-POOL-ID` with the `Id` value.
5. To find the Amazon Cognito user pool app client ID, copy the following command, paste it into the AWS CLI, and press **Enter**.

```
$ aws cognito-idp list-user-pool-clients \
--user-pool-id AMAZON-COGNITO-USER-POOL-ID \
--max-results 60
```

6. In the response, under UserPoolClients, note the `ClientId` value.

```
{
    "UserPoolClients": [
        {
            "ClientName": "Git2ItApp",
            "UserPoolId": "AMAZON-COGNITO-USER-POOL-ID",
            "ClientId": "AMAZON-COGNTIO-USER-POOL-APP-CLIENT-ID"
        }
    ]
}
```

7. In `aws-variables.js`, replace the `AMAZON-COGNITO-USER-POOL-APP-CLIENT-ID` with the `ClientId` value.
8. To find the Amazon API Gateway endpoint for Git2It, navigate to the Amazon API Gateway console.
9. Select **Git2It API**.
10. In the navigation pane under **Git2It API**, select **Stages**.
11. Select **Prod**.
12. Note the **Invoke URL** value.
13. In `aws-variables.js`, replace the `API-GATEWAY-ENDPOINT-FOR-GIT2IT-API` value with the **Invoke URL** value.

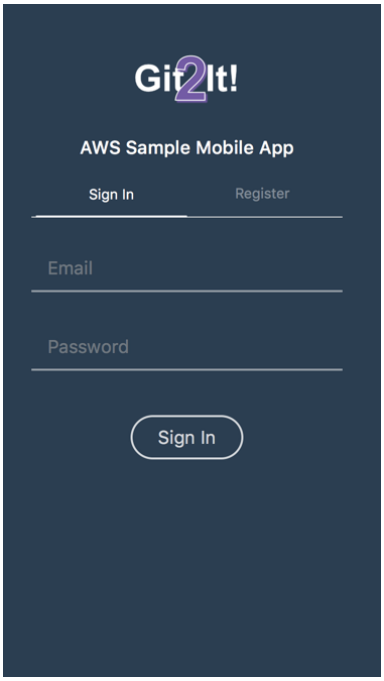
Step 5. Enter Test Data Into Git2It

Start a local development server for app development and testing.

Create a Git2It user account

1. In the AWS CLI, from the root of your Git2It Ionic App project directory, run `Ionic serve`.
   The `Ionic serve` command starts the local development server. It also starts `LiveReload` which is used to monitor changes to the file system. When you save a file, the browser refreshes automatically. To learn more about Ionic serve, please see the Ionic's Testing and Live Development webpage.

2. Verify that the Git2It app is running in a web browser.
   If the Git2It app is not running in a web browser, check to make sure the development server started and open http://localhost:8100.



3. Choose **Register**, type the required information, and choose **Register**.

4. Check the email address you provided when you registered for an email that contains your registration code.

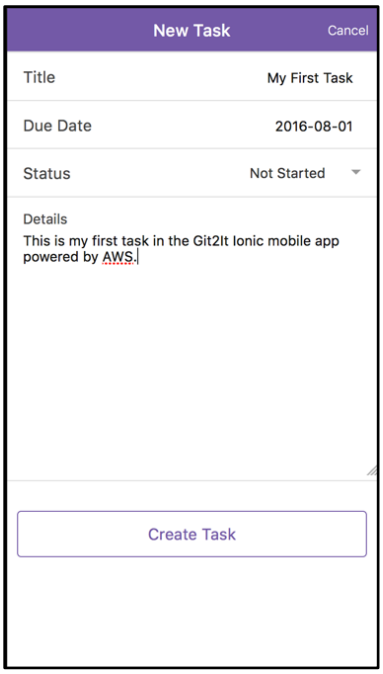
5. On the **Confirm Your Account** page, type the email you used to register and the verification code.

6. Choose **Confirm Account**.
   After your account is confirmed, the sign-in page displays.

7. Enter your credentials and choose **Sign In**.

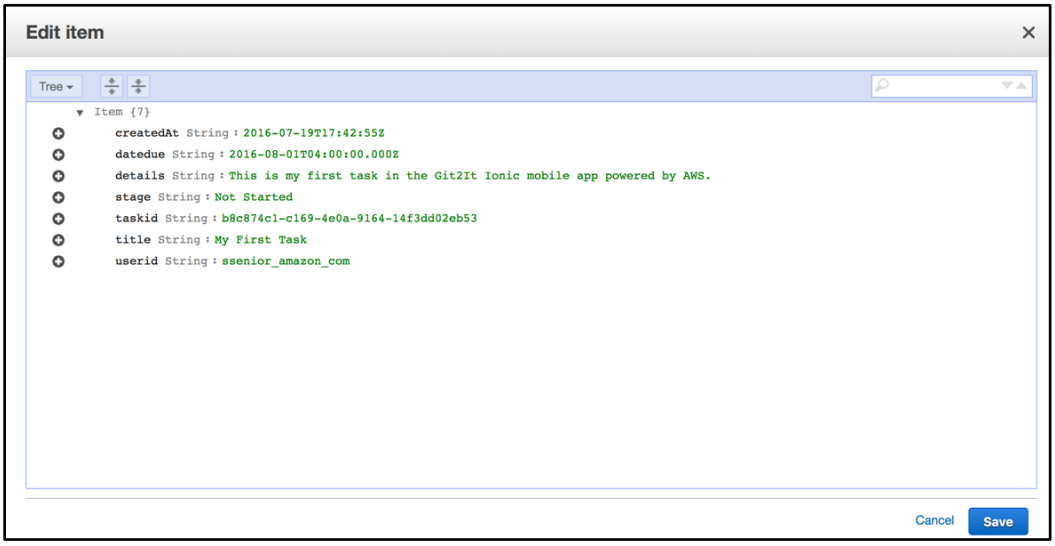
Create a task

1. At the bottom of the browser window, choose +.
2. Complete the fields and choose **Create Task**.



After you create the task, it will display on your **My Tasks** page.

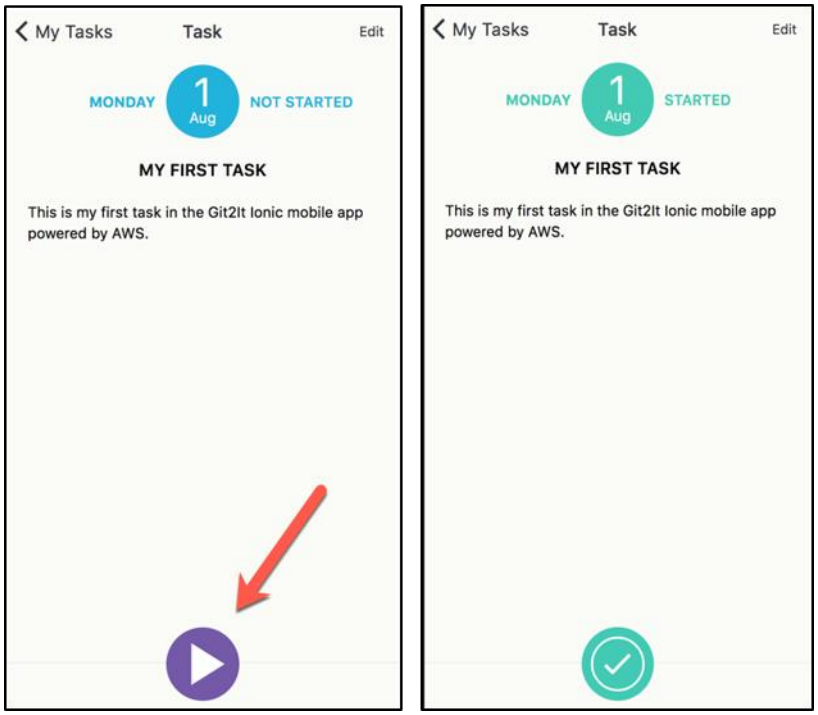
3. Navigate to the Amazon DynamoDB console.
4. In the **Git2It-Tasks** table, verify that a record shows for the task you created.



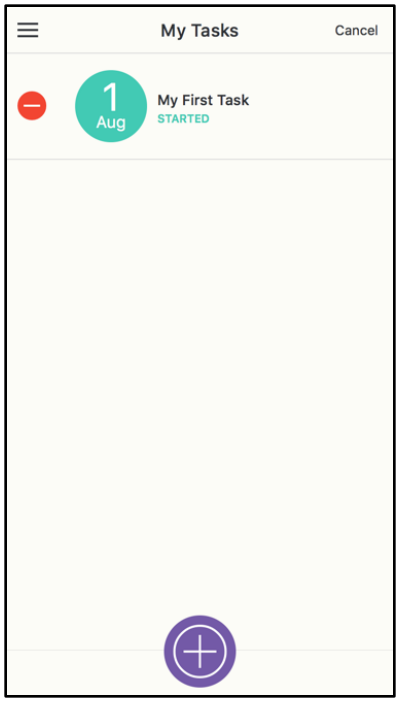
5. To view the task you created, choose it.

Update and delete a task

1. To test updating a task, choose the **play** icon.
   The task status will change from **Not Started** to **Started**.



2. In the app header, choose < **My Tasks**.
3. To test deleting a task, in the header of the app, choose **Edit.** Then, choose the – icon**.**



4. In the top left corner of the **My Tasks** screen, choose the **sidebar menu** icon.
5. Choose **About**.

   The About page shows attributes of the user from the Amazon Cognito user pool.
6. Choose **Sign Out**.

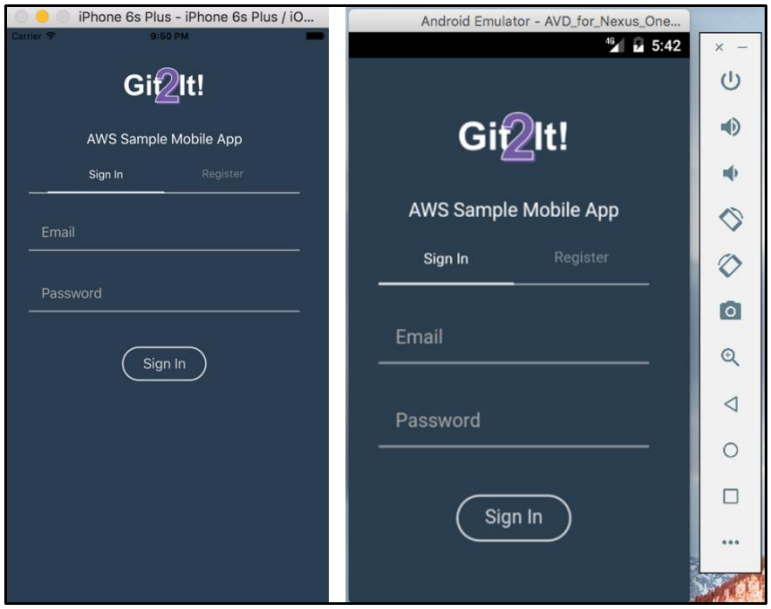
Step 6. Test the Git2It app

Deploy the app on emulators

1. In the AWS CLI, from the root of your Git2It Ionic App project directory, run `Ionic build ios android`.
   This will add the appropriate binaries required to deploy your code on the Android and iOS emulators.
2. Verify that you have XCode and the Android SDK installed on your workstation.

   You can install XCode here. You can install the Android SDK here.
3. For Android, you must install packages through the Android SDK manager and create an Android Virtual Device.
4. In the AWS CLI, from the root of your Git2It Ionic App project directory, run `Ionic emulate android`.
5. In the AWS CLI, from the root of your Git2It Ionic App project directory, run `Ionic emulate ios`.

After you execute those commands, the Android and iOS emulators will launch with the app.



Deploy the app on real devices

To test the Git2It mobile application on real devices, leverage the AWS Device Farm. AWS Device Farm is an app testing service that lets you test and interact with your Android, iOS, and web apps on many devices at once, or reproduce issues on a device in real time. AWS Device Farm allows you to view video, screenshots, logs, and performance data to pinpoint and fix issues before shipping your app.

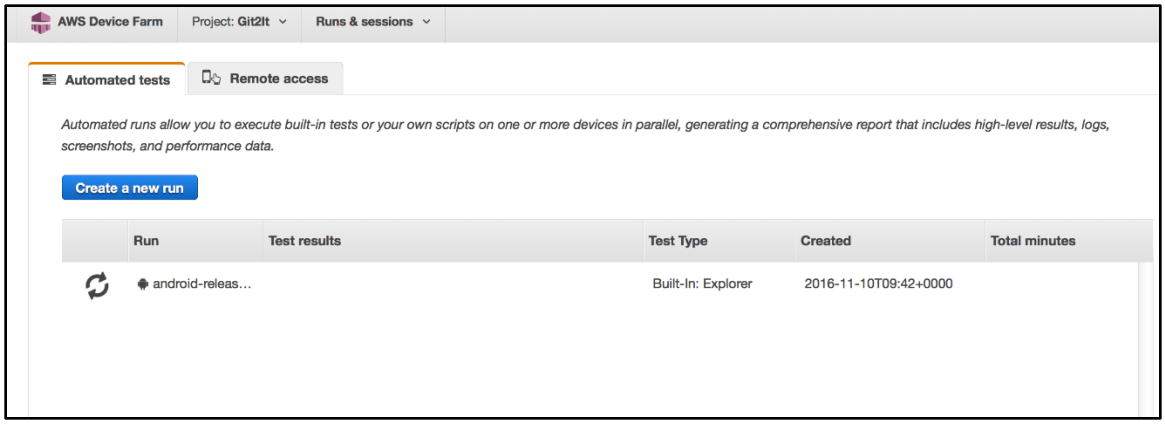
This test will generate an .apk file to install the app on an Android device.

> **Note:** Apple requires all iOS apps to have an .ipa file signed with a certificate from Apple. Since you will launch your packaged Ionic app on real iOS devices in AWS Device Farm, your generated .ipa file must be signed with a certificate from Apple. Please review the Cordova documentation to learn how to get your .ipa file signed.

1. In the AWS CLI, from the root of your Git2It Ionic App project directory, run `cordova build --release Android`.
   The `android-release-unsigned.apk` file will be created in the `./platforms/android/build/outputs/apk` directory.
2. In the AWS Device Farm console, choose **Create a new project**.
3. For **Project name**, type `Git2It`.
4. Choose **Create project**.

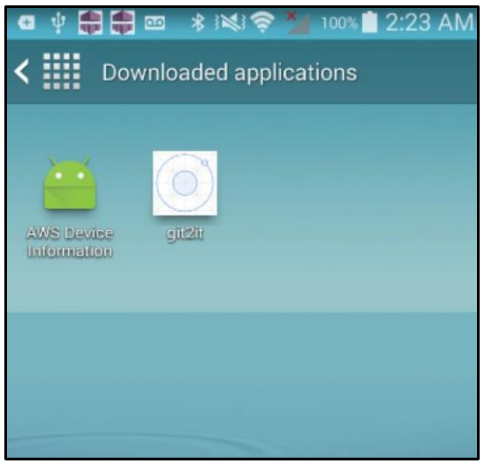
5. On the **Automated tests** tab, select **Create a new run**.
6. Choose the Android and Apple logos.

7. Choose **Upload**, navigate to the `android-release-unsigned.apk` file, and choose **Open**.
8. Choose **Next step**.
9. Choose **Built in: Explorer**.
   Built-in: Explorer allows you to test basic user flows without writing scripts. AWS Device Farm will captures screenshots, logs, and performance data. All results and artifacts are collated into a Device Farm report and also made available through the [Device Farm API](#).
10. Choose **Next step** until the test starts.



After the test completes, a performance report is generated to show how the Git2It app performed on different devices.

11. On the **Remote Access** tab, choose **Start a new session**.
12. Choose the desired Android Device and select **Confirm and start session**.
13. Choose **Upload**, navigate to the `android-release-unsigned.apk` file, and choose **Open**.
14. Navigate to **Downloaded apps** and verify that the Git2It shows.



15. Select the Git2It icon and test the app.

Appendix A: Git2It App Code

The Git2It app integrates with Amazon Cognito user pools for user management, and Amazon API Gateway for access to CRUD operations for the task data.

User Management

The `AuthServices.js` file (`./www/services/AuthServices.js`) contains the code that is responsible for user management.

The following code snippet from the `AuthService.js` allows users to sign-up in the Amazon Cognito user pool.

```
    this.signup = function(newuser) {
        var deferred = $q.defer();

        newuser.username = newuser.email.replace("@", "_").replace(".",
"_");
        var poolData = {            UserPoolId: YOUR_USER_POOL_ID,
            ClientId: YOUR_USER_POOL_CLIENT_ID,
            Paranoia: 8
        };
```

```
        var userPool = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserPool(poolData);

        var attributeList = [];

        var dataEmail = {
            Name: 'email',
            Value: newuser.email
        };

        var dataName = {
            Name: 'name',
            Value: newuser.name
        };

        var attributeEmail = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserAttribute(dataEmail);
        var attributeName = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserAttribute(dataName);

        attributeList.push(attributeEmail);
        attributeList.push(attributeName);

        userPool.signUp(newuser.username, newuser.password, attributeList,
null, function(err, result) {
            if (err) {
                console.log(err);
                deferred.reject(err.message);
            } else {
                deferred.resolve(result.user);            }        });

        return deferred.promise;

    };
```

The following code snippet from the `AuthService.js` authenticates users and establishes a user session with the Amazon Cognito Identity service.

```
    this.signin = function(user) {
        var deferred = $q.defer();

        var authenticationData = {
            Username: user.email,
            Password: user.password,
        };

        var authenticationDetails = new
AWSCognito.CognitoIdentityServiceProvider.AuthenticationDetails(
            authenticationData);
```

```
        var poolData = {
            UserPoolId: YOUR_USER_POOL_ID,
            ClientId: YOUR_USER_POOL_CLIENT_ID,
            Paranoia: 8
        };

        var userPool = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserPool(poolData);
        var userData = {
            Username: user.email,
            Pool: userPool
        };

        var cognitoUser = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUser(userData);

        try {
            cognitoUser.authenticateUser(authenticationDetails, {
                onSuccess: function(result) {
                    console.log(cognitoUser)
                    console.log('access token + ' +
result.getIdToken().getJwtToken());
                    $localstorage.set('username',
cognitoUser.getUsername());
                    deferred.resolve(result);
                },
                onFailure: function(err) {
deferred.reject(err);
                },

            });
        } catch (e) {
            console.log(e);
            deferred.reject(e);
        }

        return deferred.promise;

    };
```

The following code snippet from the `AuthService.js` retrieves the user from local storage and determines if the current session is still valid.

```
    this.isAuthenticated = function() {
        var deferred = $q.defer();
        var data = {
            UserPoolId: YOUR_USER_POOL_ID,
            ClientId: YOUR_USER_POOL_CLIENT_ID,
            Paranoia: 8
        };
```

```
        var userPool = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserPool(data);
        var cognitoUser = userPool.getCurrentUser();

        try {
            if (cognitoUser != null) {
                cognitoUser.getSession(function(err, session) {
                    if (err) {
                        deferred.resolve(false);
                    }
                    deferred.resolve(true);
                });
            } else {
                deferred.resolve(false);
            }
        } catch (e) {
            console.log(e);
            deferred.resolve(false);
        }

        return deferred.promise;

    };
```

The following code snippet from the `AuthService.js` logs a user out of the identity service.

```
    this.logOut = function() {

        var data = {
            UserPoolId : YOUR_USER_POOL_ID,
            ClientId : YOUR_USER_POOL_CLIENT_ID ,
            Paranoia: 8
        };
        var userPool = new
AWSCognito.CognitoIdentityServiceProvider.CognitoUserPool(data);
        var cognitoUser = userPool.getCurrentUser();

        console.log(cognitoUser)
        if (cognitoUser != null) {
            console.log("signing out");
            cognitoUser.signOut();
            return;
        } else {
            return;
        }

    };
```

CRUD Operations

The `TaskFactory.js` file (`./www/factories/TaskFactory.js`) provides an interface between the UI and the Amazon API Gateway to invoke the Git2It RESTful API.

The following code snippet from the `TaskFactory.js` lists a user's tasks:

```
    var tasks_resource = function(token) {
        var _url = [APIG_ENDPOINT, 'tasks'].join('/');
        return $resource(_url, {}, {
            query: {
                method: 'GET',
                headers: {
                    'Authorization': token
                }
            }
        });
    };

    ...


    factory.listTasks = function(filter, cb) {
authService.getUserAccessTokenWithUsername().then(function(data) {
            tasks_resource(data.token.jwtToken).query({
                userid: data.username,
                filter: filter
            }, function(data) {
                return cb(null, data.Items);
            }, function(err) {
                return cb(err, null);
            });
        }, function(msg) {
            console.log("Unable to retrieve the user session.");
            $state.go('login', {});
        });

    };
```

The following code snippet from the `TaskFactory.js` creates a new task for a user:

```
    var tasks_task_resource = function(token) {
        var _url = [APIG_ENDPOINT, 'tasks/:taskId'].join('/');        return
$resource(_url, {
            taskId: '@taskId'
        }, {
            get: {
                method: 'GET',
                headers: {
```

```
                    'Authorization': token
                }
            },
            create: {
                method: 'POST',
                headers: {
                    'Authorization': token
                }
            },
            update: {
                method: 'PUT',
                headers: {
                    'Authorization': token
                }
            },
            remove: {
                method: 'DELETE',
                headers: {
                    'Authorization': token
                }
            }
        });
    };
    ...
    factory.createTask = function(taskTitle, taskDateDue, taskDetails,
taskStage, cb) {

        authService.getUserAccessTokenWithUsername().then(function(data) {
            var _task = {
                userid: data.username,
                title: taskTitle,
                datedue: taskDateDue,
                details: taskDetails,
                stage: taskStage
            };

            tasks_task_resource(data.token.jwtToken).create({
                taskId: "new"
            }, _task, function(data) {
                if ($_.isEmpty(data)) {
                    return cb(null, data);
                }
                return cb(null, data);
            }, function(err) {
                return cb(err, null);
            });
        }, function(msg) {
            console.log("Unable to retrieve the user session.");
            $state.go('login', {});
        });
    };
```

The following code snippet from the `TaskFactory.js` updates an existing task:

```
var tasks_task_resource = function(token) {
    var _url = [APIG_ENDPOINT, 'tasks/:taskId'].join('/');
    return $resource(_url, {
        taskId: '@taskId'
    }, {
        get: {
            method: 'GET',
            headers: {
                'Authorization': token
            }
        },
        create: {
            method: 'POST',
            headers: {
                'Authorization': token
            }
        },            update: {                method: 'PUT',
            headers: {
                'Authorization': token
            }
        },
        remove: {
            method: 'DELETE',
            headers: {
                'Authorization': token
            }
        }
    });
};

...

factory.updateTask = function(task, cb) {

    authService.getUserAccessTokenWithUsername().then(function(data) {
        tasks_task_resource(data.token.jwtToken).update({
            taskId: task.taskid,
            userid: data.username
        }, task, function(data) {
            if ($_.isEmpty(data)) {
                return cb(null, data);
            }
            return cb(null, data);
        }, function(err) {
            return cb(err, null);
        });
    }, function(msg) {
        console.log("Unable to retrieve the user session.");
        $state.go('login', {});
```

```
        });
    };
```

The following code snippet from the `TaskFactory.js` deletes a task:

```
    var tasks_task_resource = function(token) {
        var _url = [APIG_ENDPOINT, 'tasks/:taskId'].join('/');
        return $resource(_url, {
            taskId: '@taskId'
        }, {
            get: {
                method: 'GET',
                headers: {
                    'Authorization': token
                }            },            create: {
                method: 'POST',
                headers: {
                    'Authorization': token
                }
            },
            update: {
                method: 'PUT',
                headers: {
                    'Authorization': token
                }
            },
            remove: {
                method: 'DELETE',
                headers: {
                    'Authorization': token
                }
            }
        });
    };

    ...

    factory.deleteTask = function(taskid, cb) {

        authService.getUserAccessTokenWithUsername().then(function(data) {
            tasks_task_resource(data.token.jwtToken).remove({
                taskId: taskid,
                userid: data.username
            }, function(data) {
                return cb(null, data);
            }, function(err) {
                return cb(err, null);
            });
        }, function(msg) {
            console.log("Unable to retrieve the user session.");
            $state.go('login', {});
```

```
        });
    };
```

The following code snippet from the `TaskFactory.js` retrieves a user's task:

```
    var tasks_task_resource = function(token) {
        var _url = [APIG_ENDPOINT, 'tasks/:taskId'].join('/');
        return $resource(_url, {
            taskId: '@taskId'
        }, {
            get: {                method: 'GET',                headers: {
                    'Authorization': token
                }
            },
            create: {
                method: 'POST',
                headers: {
                    'Authorization': token
                }
            },
            update: {
                method: 'PUT',
                headers: {
                    'Authorization': token
                }
            },
            remove: {
                method: 'DELETE',
                headers: {
                    'Authorization': token
                }
            }
        });
    };

    ...

    factory.getTask = function(taskid, cb) {

        authService.getUserAccessTokenWithUsername().then(function(data) {
            tasks_task_resource(data.token.jwtToken).get({
                taskId: taskid,
                userid: data.username
            }, function(data) {
                if ($_.isEmpty(data)) {
                    return cb(null, data);
                }
                return cb(null, data.Item);
            }, function(err) {
                return cb(err, null);
            });
```

```
        }, function(msg) {
            console.log("Unable to retrieve the user session.");
            $state.go('login', {});
        });
    };
```

Document Revisions

Date	Change	In sections
December 2016	Initial Release	--

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.