Serverless Gardens

IoT + Serverless

johncmckim.me

twitter.com/@johncmckim

medium.com/@johncmckim

John McKim

Software Engineer at A Cloud Guru

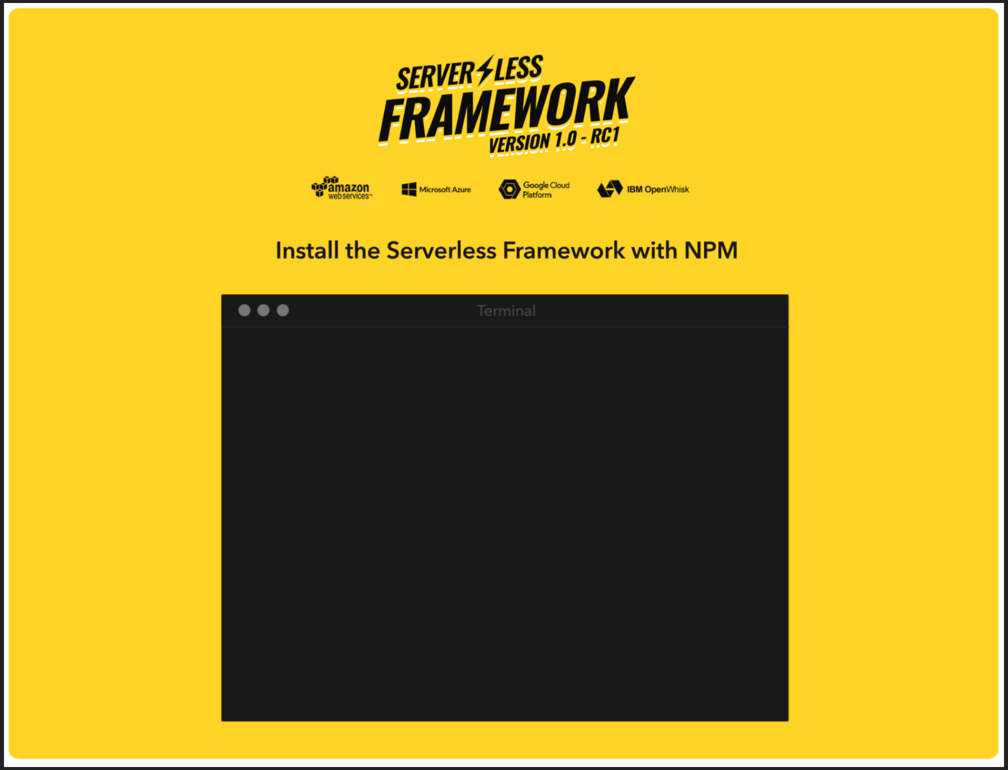
Contribute to Serverless Framework

@johncmckim

A CLOUD GURU

https://acloud.guru

Serverless Framework



https://serverless.com

Agenda

- What is Serverless
- Why I built this project
- Overall Architecture
- Design of each Microservice
- GraphQL + Lambda
- What I learnt
- Questions

What is Serverless?

Serverless

FaaS + The Herd

What is Serverless?

A Serverless Architecture is an event driven system that utilises FaaS *and other fully managed services for logic and persistence.*

Why choose Serverless?

Benefits

- Easier Operations Mangement
- Reduced Operational Cost
- Reduced Development Time / Cost
- Highly Scalable
- Loosely Coupled systems

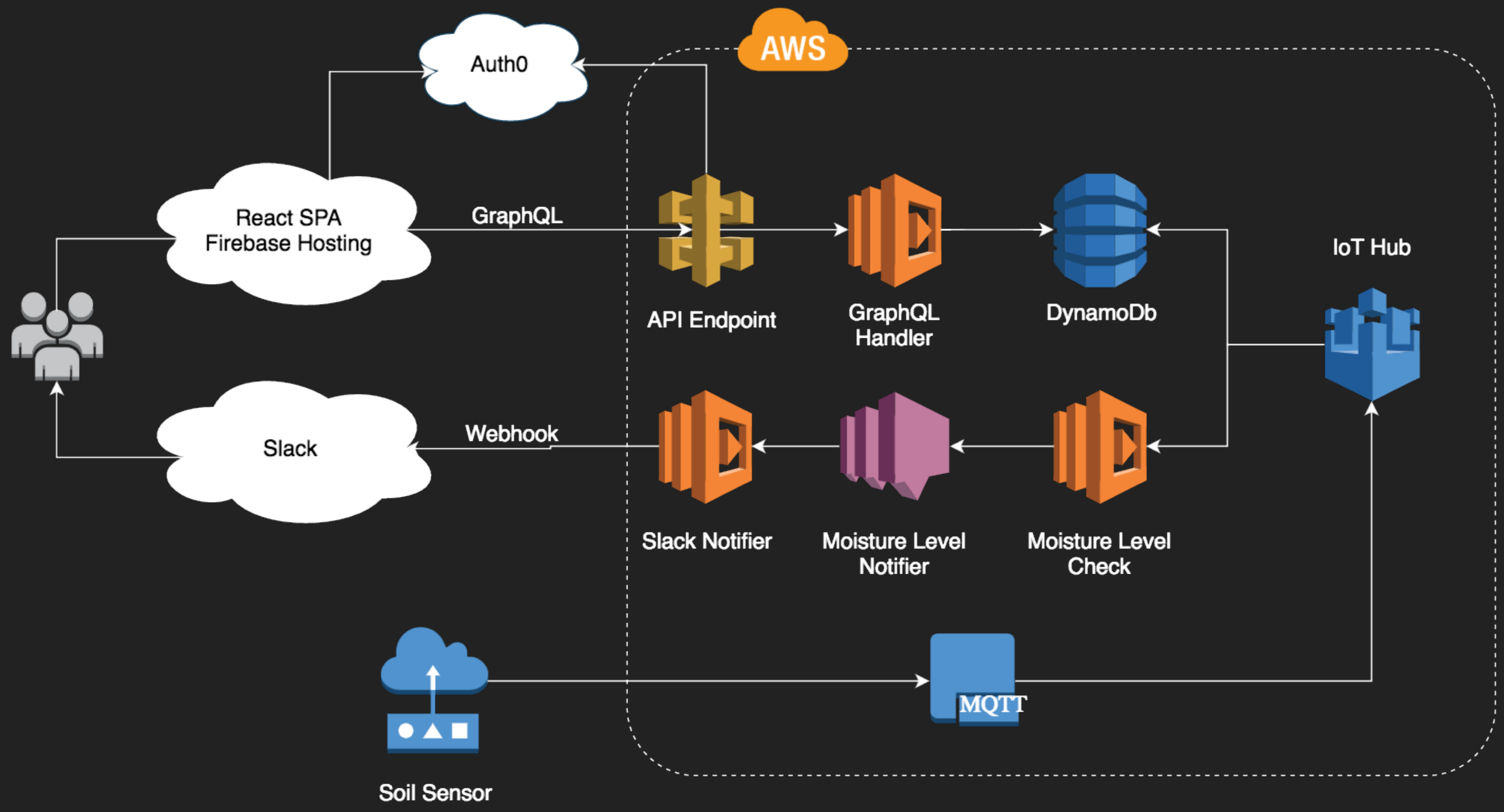
Why build this?

For fun and learning

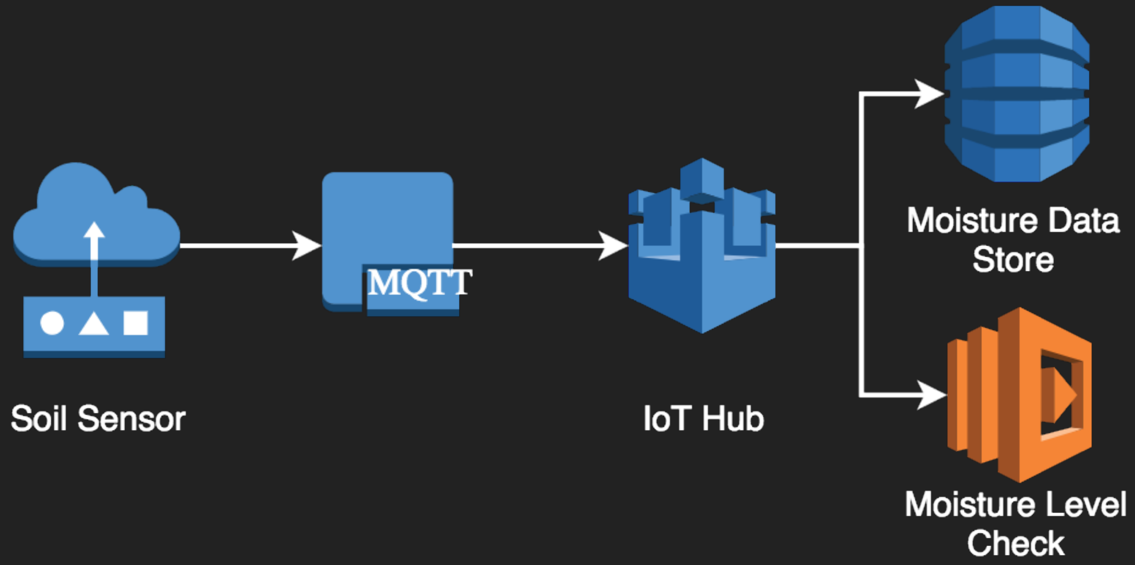
The Problem

Caring for my Garden

Serverless Garden

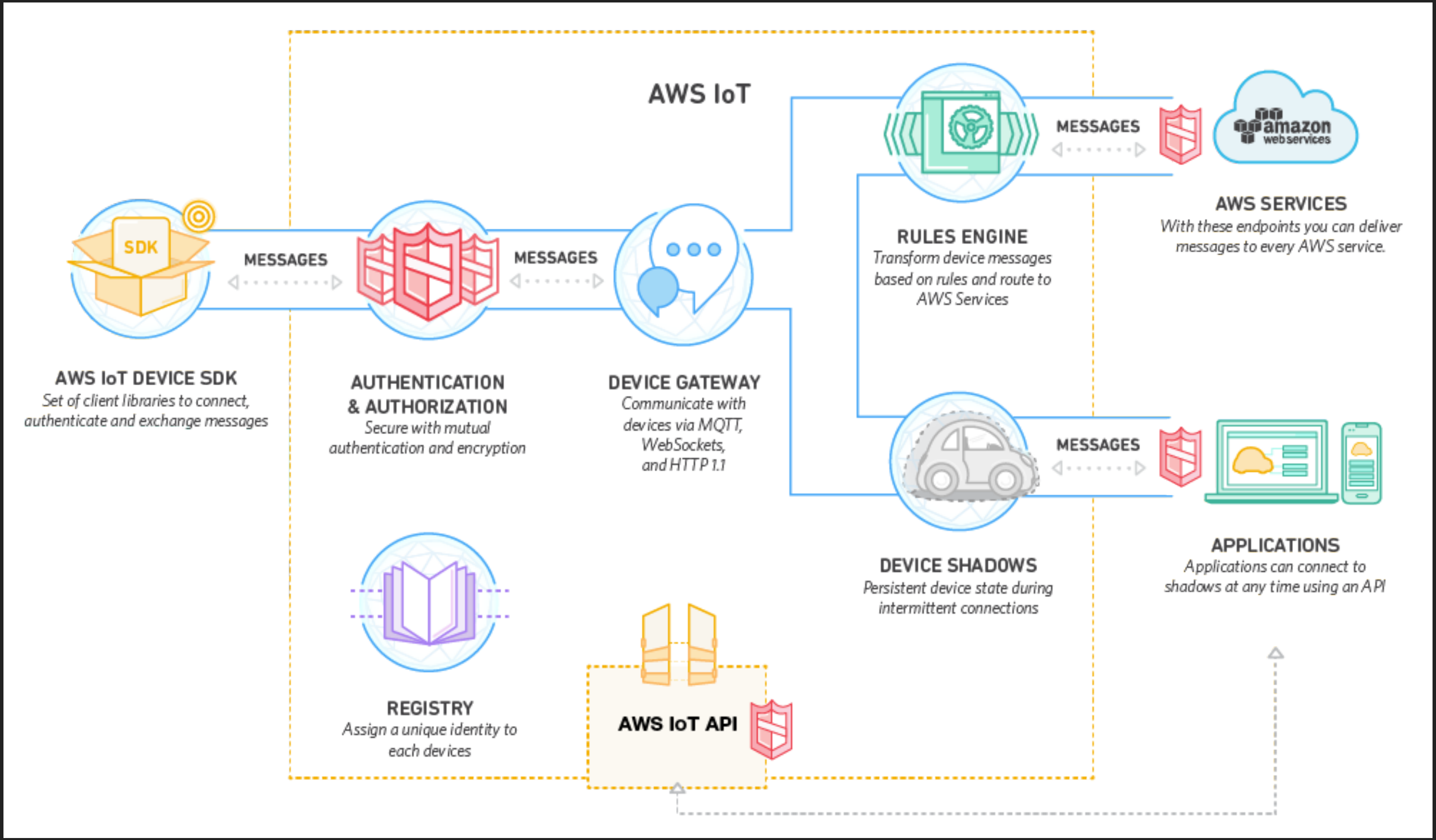


IoT Service


Soil Sensor
MQTT
IoT Hub
Moisture Data Store
Moisture Level Check

AWS IoT Service

How It works


AWS IoT
SDK
MESSAGES
MESSAGES
MESSAGES
MESSAGES
amazon web services
AWS IoT DEVICE SDK
Set of client libraries to connect, authenticate and exchange messages
AUTHENTICATION & AUTHORIZATION
Secure with mutual authentication and encryption
DEVICE GATEWAY
Communicate with devices via MQTT, WebSockets, and HTTP 1.1
RULES ENGINE
Transform device messages based on rules and route to AWS Services
AWS SERVICES
With these endpoints you can deliver messages to every AWS service.
DEVICE SHADOWS
Persistent device state during intermittent connections
APPLICATIONS
Applications can connect to shadows at any time using an API
REGISTRY
Assign a unique identity to each devices
AWS IoT API

Device Gatway

Protocols

- MQTT - devices
- MQTT over Web Sockets - browsers
- HTTP - last resort

Device Gatway

Authentication

- X.509 Certificates - Mutual TLS
- IAM - Signed Requests
- Cognito - tokens

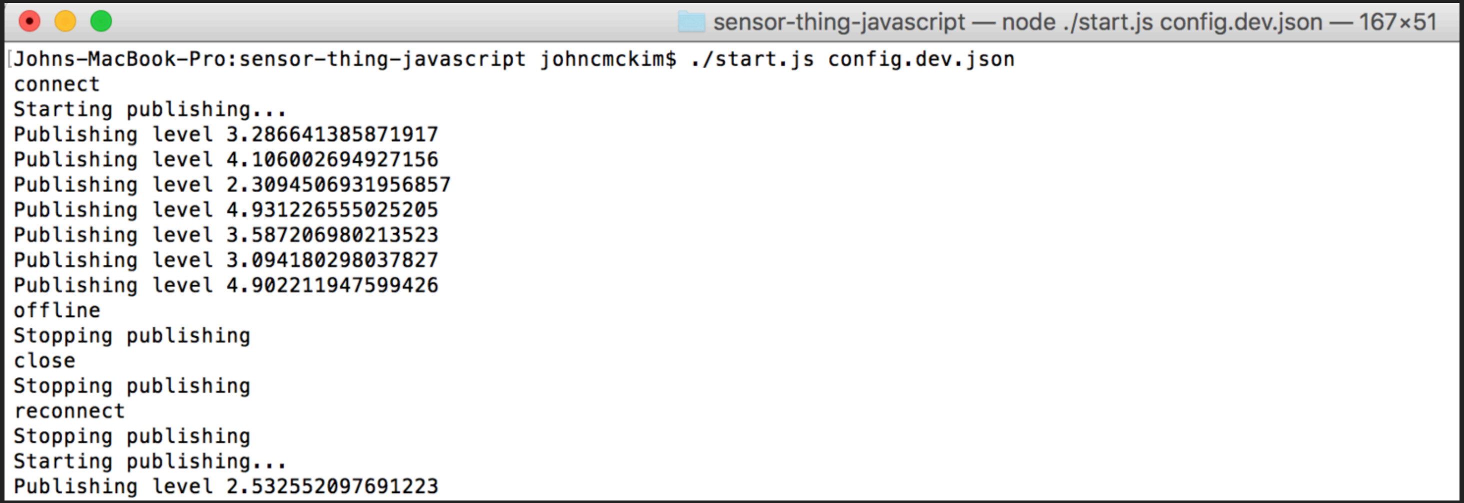
Device

Fake Device

```
const awsIot = require('aws-iot-device-sdk');

const device = awsIot.device({
  'keyPath': './certificates/private.pem.key',
  'certPath': './certificates/certificate.pem.crt',
  'caPath': './certificates/verisign-ca.pem',
  'clientId': 'garden-aid-client-test-js',
  'region': 'ap-southeast-2'
});

device
  .on('connect', function() {
    const topic = 'garden/soil/moisture';
    const message = JSON.stringify({
      DeviceId: 'test-js-device',
      Recorded: (new Date()).toISOString(),
      Level: level
    });

    device.publish(topic, message, {});
  });
```

Demo

Fake Device


sensor-thing-javascript — node ./start.js config.dev.json — 167×51

```
Johns-MacBook-Pro:sensor-thing-javascript johncmckim$ ./start.js config.dev.json
connect
Starting publishing...
Publishing level 3.286641385871917
Publishing level 4.106002694927156
Publishing level 2.3094506931956857
Publishing level 4.931226555025205
Publishing level 3.587206980213523
Publishing level 3.094180298037827
Publishing level 4.902211947599426
offline
Stopping publishing
close
Stopping publishing
reconnect
Stopping publishing
Starting publishing...
Publishing level 2.532552097691223
```

Rules Engine

Message Selection & Transformation

SQL Statement

- FROM — MQTT topic
- SELECT — transforms the data
- WHERE (optional)

SELECT DeviceId, Recorded, Level FROM 'garden/soil/moisture'

Rules Engine

Actions

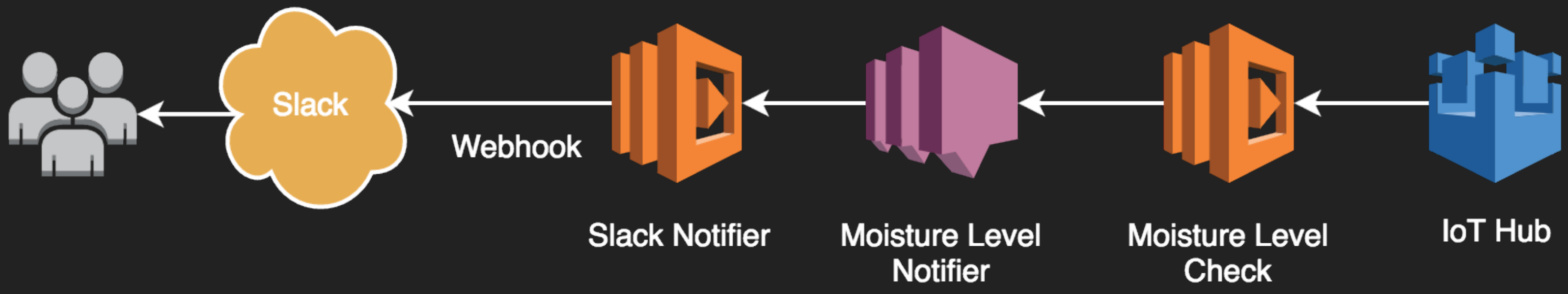
- Lambda
- DynamoDB
- ElasticSearch
- SNS
- SQS
- Kinesis
- CloudWatch
- Republish to another MQTT topic.

Rules Engine

IoT Rule in serverless.yml

```
SensorThingRule:
  Type: AWS::IoT::TopicRule
  Properties:
    TopicRulePayload:
      RuleDisabled: false
      Sql: "SELECT DeviceId, Recorded, Level FROM '${{opt:stage}}/garden/soil/moisture
      Actions:
        -
          DynamoDB:
            TableName: { Ref: MoistureData }
            HashKeyField: "ClientId"
            HashKeyValue: "${clientId()}"
            RangeKeyField: "Timestamp"
            RangeKeyValue: "${timestamp()}"
            PayloadField: "Data"
            RoleArn: { Fn::GetAtt: [ IotThingRole, Arn ] }
        -
          Lambda:
            FunctionArn: { Fn::GetAtt: [ checkMoistureLevel, Arn ] }
```

Notifications Service



- Single purpose functions
- High cohesion
- Loose coupling

Messaging Options

Amazon Simple Queue Service (SQS)

Fully Managed message queuing service.

Benefits

- Dead letter queues
- Reliable

Drawbacks

- No integration with Lambda
- Difficult to build scalable processor
- Single processor / queue

Messaging Options

Amazon Kinesis Streams

Capture and store streaming data.

Benefits

- Integrates with Lambda
- Batched messages
- Ordered messages

Drawbacks

- Single lambda / shard
- Scale per shard
- Log jams
- Messages expire

Messaging Options

Amazon Simple Notification Service (SNS)

Full managed messaging and Pub/Sub service

Benefits

- Integrates with Lambda
- Fan out multiple Lambdas

Drawbacks

- Small message size
- 3-5 retry's then drop message

Notification Service

Check Level

```
const AWS = require('aws-sdk');
const sns = new AWS.SNS();

const publish = (msg, topicArn, cb) => {
  sns.publish({
    Message: JSON.stringify({
      message: msg
    }),
    TopicArn: topicArn
  }, cb);
};

module.exports.checkLevel = (event, context, cb) => {
  if(event.Level < 2.5) {
    const msg = 'Moisture level has dropped to ' + event.Level;

    const topicArn = process.env.mositureNotifyTopic;

    publish(msg, topicArn, cb);
    cb(null, { message: msg, event: event });
    return;
  }

  cb(null, { message: 'No message to publish', event: event });
}
```

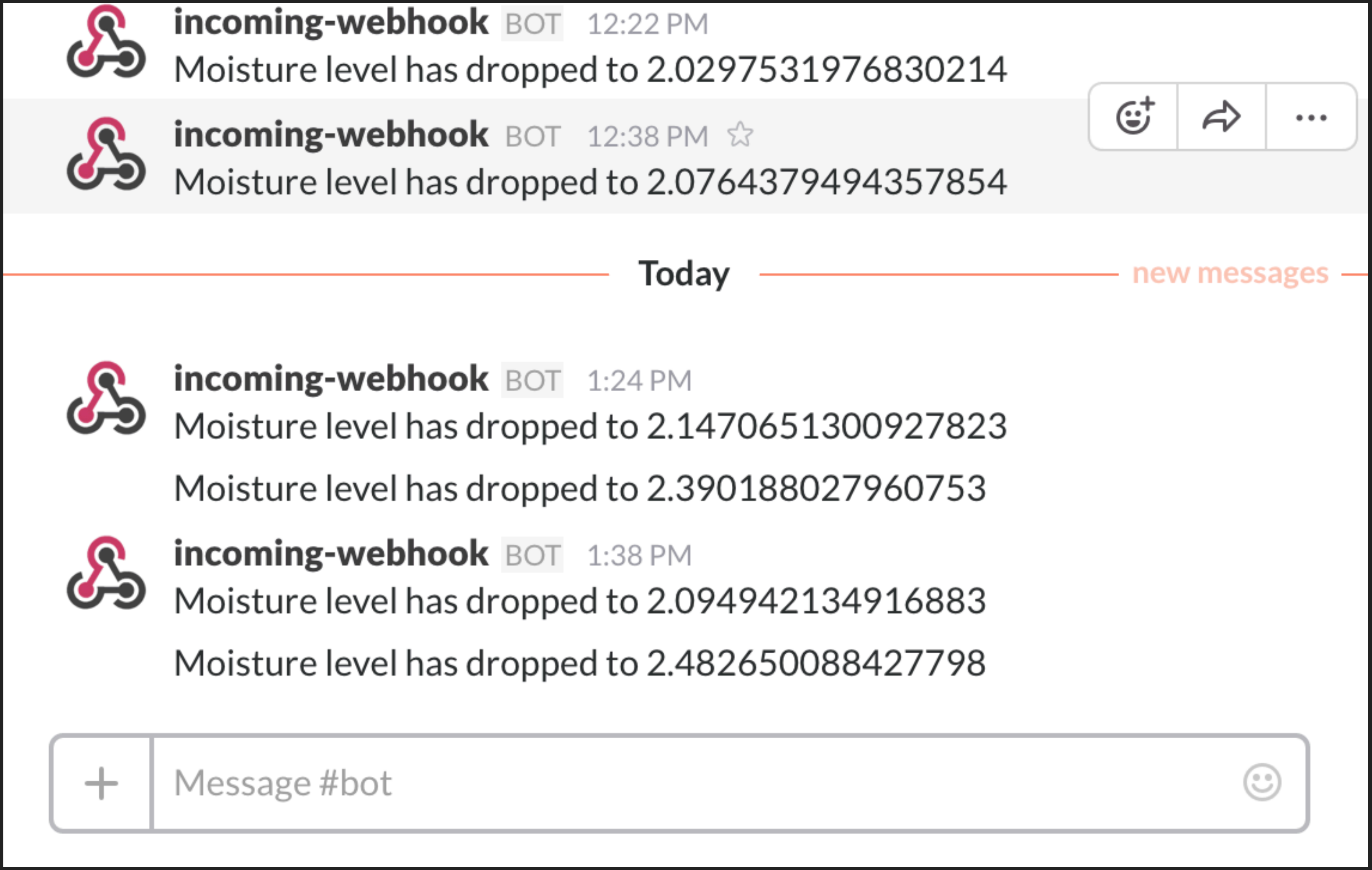
Notifications Service

Slack Notifier

```javascript
const BbPromise = require('bluebird');
const rp = require('request-promise');
const util = require('util');

const notify = (msg) => {
  return rp({
    method: 'POST',
    uri: process.env.slackWebHookUrl,
    json: true,
    body: {
      text: msg,
    },
  });
}

module.exports.notify = (event, context, cb) =>  {
  console.log(util.inspect(event, false, 5));

  const promises = [];

  event.Records.forEach(function(record) {
    if(record.EventSource !== 'aws:sns') {
      console.warn('Recieved non sns event: ', record);
      return;
    }
```

Demo

Slack Notifications


incoming-webhook BOT 12:22 PM
Moisture level has dropped to 2.0297531976830214

incoming-webhook BOT 12:38 PM
Moisture level has dropped to 2.0764379494357854

Today — new messages

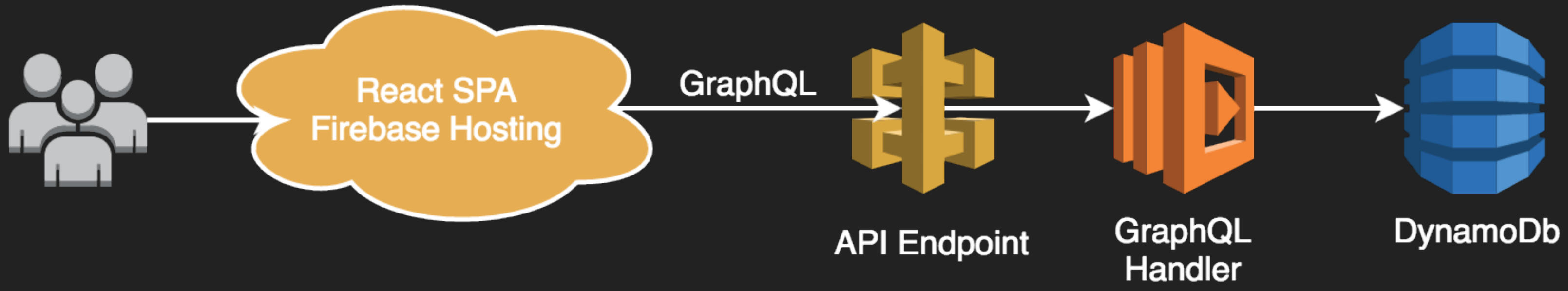
incoming-webhook BOT 1:24 PM
Moisture level has dropped to 2.1470651300927823

Moisture level has dropped to 2.390188027960753

incoming-webhook BOT 1:38 PM
Moisture level has dropped to 2.094942134916883

Moisture level has dropped to 2.482650088427798

Web Services



Web Client

- React SPA
- Firebase Hosting
- Auth0 for authentication

Web Backend

- GraphQL API
- API Gateway + Lambda
- Data in DynamoDB
- Custom authoriser

Web Services

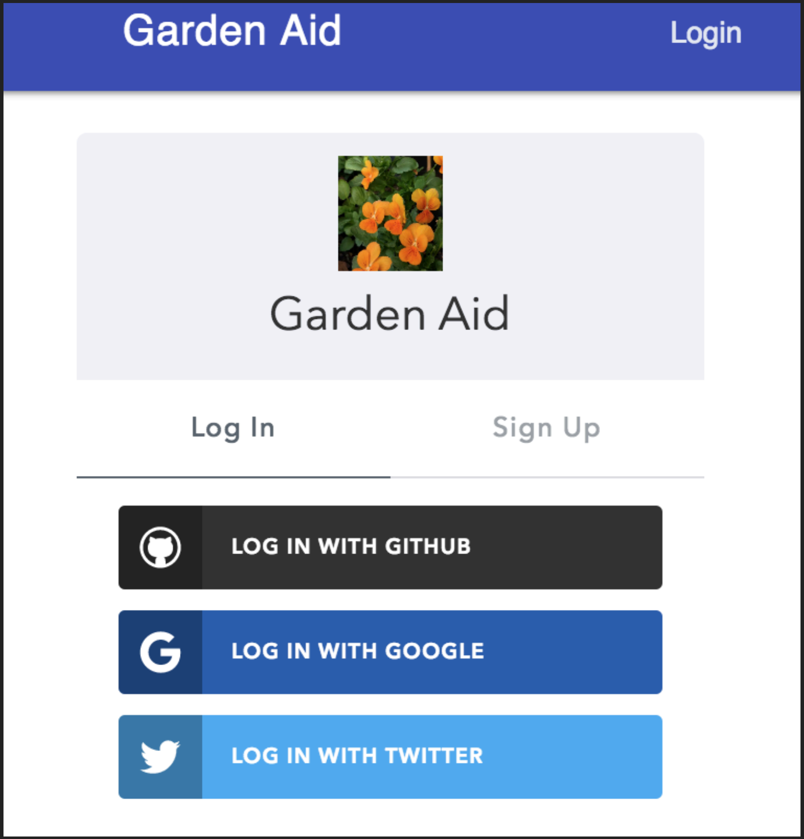
API Gateway

What is it?

- HTTP Endpoint as a Service
- Integrates with Lambda
- Convert HTTP Request to Event
- Can delegate Authorization

Web Services

Auth0 Authentication



Web Services

Authentication with GraphQL

```
const networkInterface = createNetworkInterface(GRAPHQL_URL);

networkInterface.use([{

  applyMiddleware(req, next) {
    if (!req.options.headers) {
      req.options.headers = {}; // Create the header object if needed.
    }

    // get the authentication token from local storage if it exists
    const idToken = localStorage.getItem('idToken') || null;
    if (idToken) {
      req.options.headers.Authorization = `Bearer ${idToken}`;
    }
    next();
  },
}]);
```

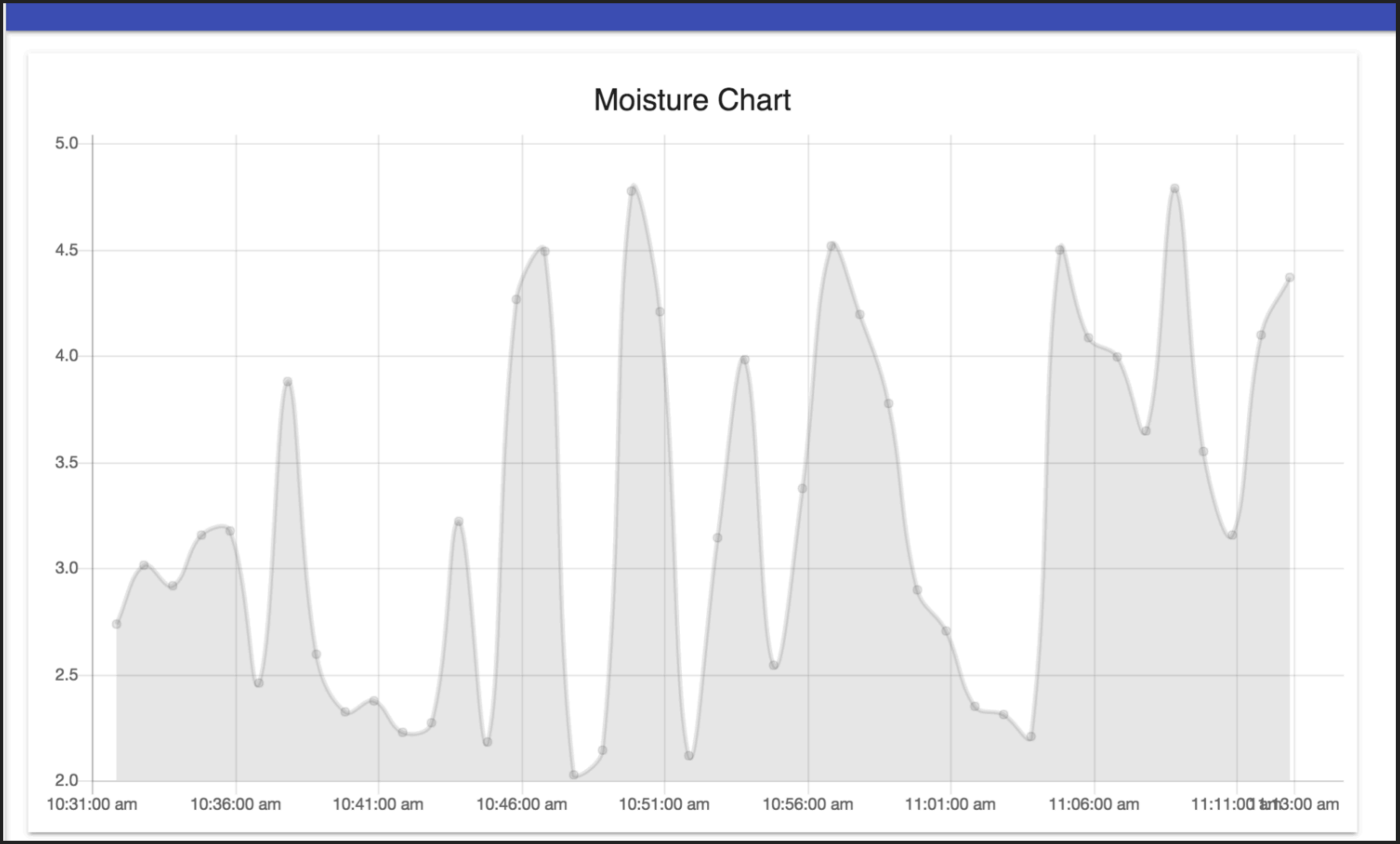
Web Services

Custom Authorizer

```
const utils = require('./auth/utils');
const auth0 = require('./auth/auth0');
const AuthenticationClient = require('auth0').AuthenticationClient;

const authClient = new AuthenticationClient({
  domain: process.env.AUTH0_DOMAIN,
  clientId: process.env.AUTH0_CLIENT_ID,
});

module.exports.handler = (event, context, cb) => {
  console.log('Received event', event);

  const token = utils.getToken(event.authorizationToken);

  if (!token) {
    return cb('Missing token from event');
  }

  const authInfo = utils.getAuthInfo(event.methodArn);

  return authClient.tokens.getInfo(token)
```

Demo

Dashboard



What is GraphQL?

Schema

```
type Project {
  name: String
  stars: Int
  contributors: [User]
}
```

Query

```
{
  project(name: "GraphQL") {
    stars
  }
}
```

Results

```
{
  "project": {
    "stars": 4462
  }
}
```

Why GraphQL?

- One endpoint (per service) to access your data
- The client chooses the response format
- No versioning *

GraphQL Query

```
import gql from 'graphql-tag';
import { connect } from 'react-apollo';
import MoistureChart from '../../pres/Moisture/Chart';

export default connect({
  mapQueriesToProps({ ownProps, state }) {
    return {
      moisture: {
        query: gql`{
          moisture(hours: ${ownProps.hours}, clientId: "${ownProps.clientId}") {
            date, moisture
          }
        }`,
        variables: {},
        pollInterval: 1000 * 30, // 30 seconds
      },
    };
  },
})(MoistureChart);
```

GraphQL Schema

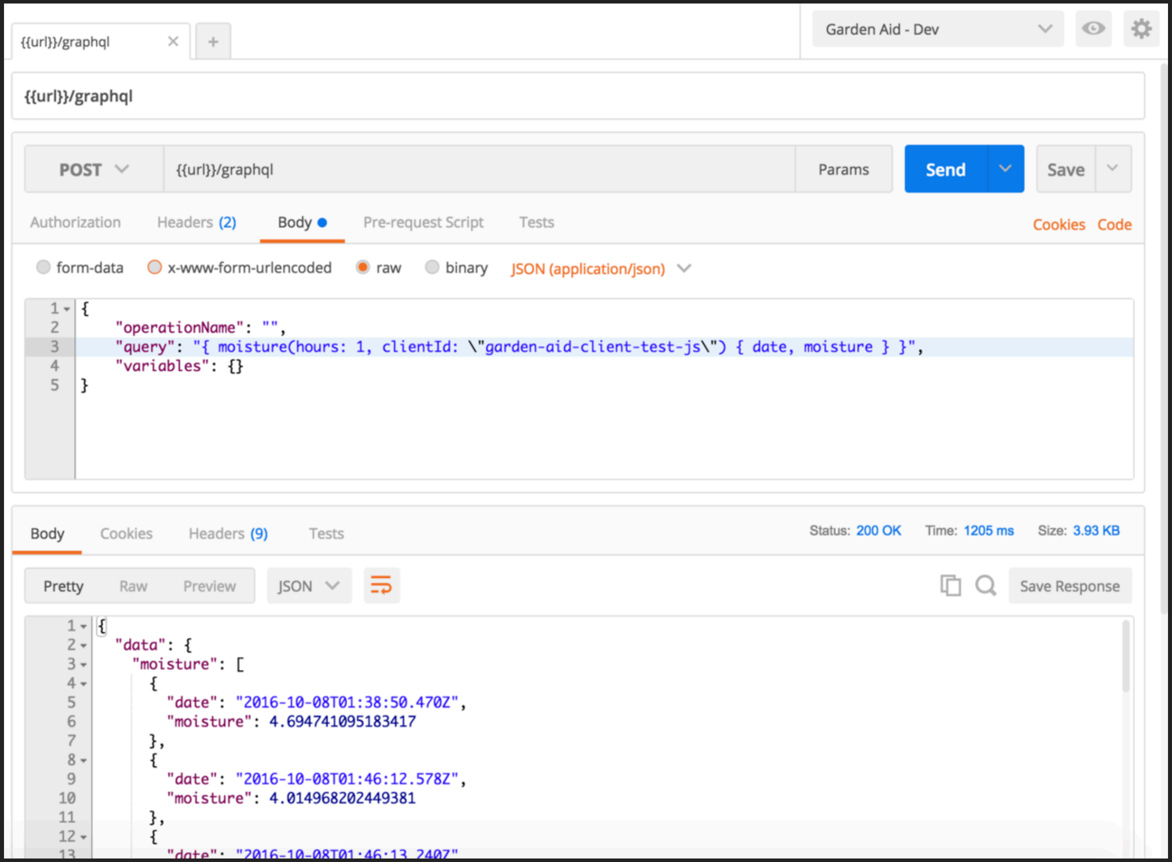
```
const graphql = require('graphql');
const tablesFactory = require('./dynamodb/tables');
const MoistureService    = require('./services/moisture');

const tables = tablesFactory();
const moistureService = MoistureService({ moistureTable: tables.Moisture });

const MoistureType = new graphql.GraphQLObjectType({
  name: 'MoistureType',
  fields: {
    date: { type: graphql.GraphQLString },
    moisture: { type: graphql.GraphQLFloat },
  }
});

const schema = new graphql.GraphQLSchema({
  query: new graphql.GraphQLObjectType({
    name: 'Root',
    description: 'Root of the Schema',
    fields: {
      moisture:
        name: 'MoistureQuery',
        description: 'Retrieve moisture levels',
        type: new graphql.GraphQLList(MoistureType),
```

AWS Lambda

```
const graphql = require('graphql');

const schema = require('./schema');

module.exports.handler = function(event, context, cb) {
  console.log('Received event', event);

  const query = event.body.query;

  return graphql.query(schema, event.body.query)
    .then((response) => {
      cb(null, response)
    })
    .catch((error) => {
      cb(error)
    });
}
```

Demo

GraphQL Query

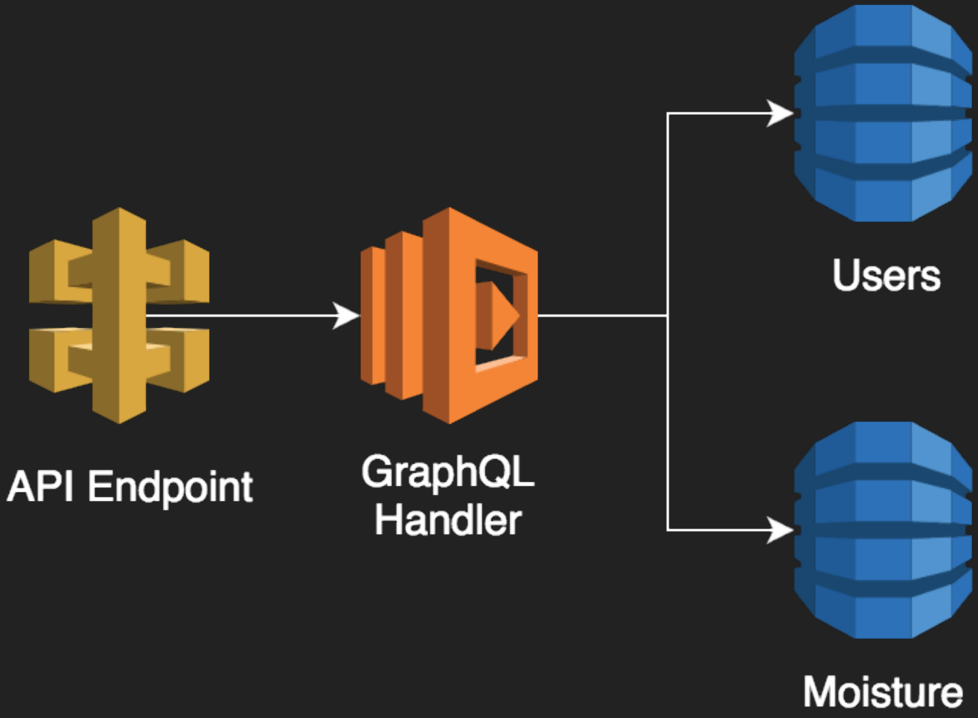


```
{
    "operationName": "",
    "query": "{ moisture(hours: 1, clientId: \"garden-aid-client-test-js\") { date, moisture } }",
    "variables": {}
}
```

```
{
  "data": {
    "moisture": [
      {
        "date": "2016-10-08T01:38:50.470Z",
        "moisture": 4.694741095183417
      },
      {
        "date": "2016-10-08T01:46:12.578Z",
        "moisture": 4.014968202449381
      },
      {
        "date": "2016-10-08T01:46:13.240Z"
```

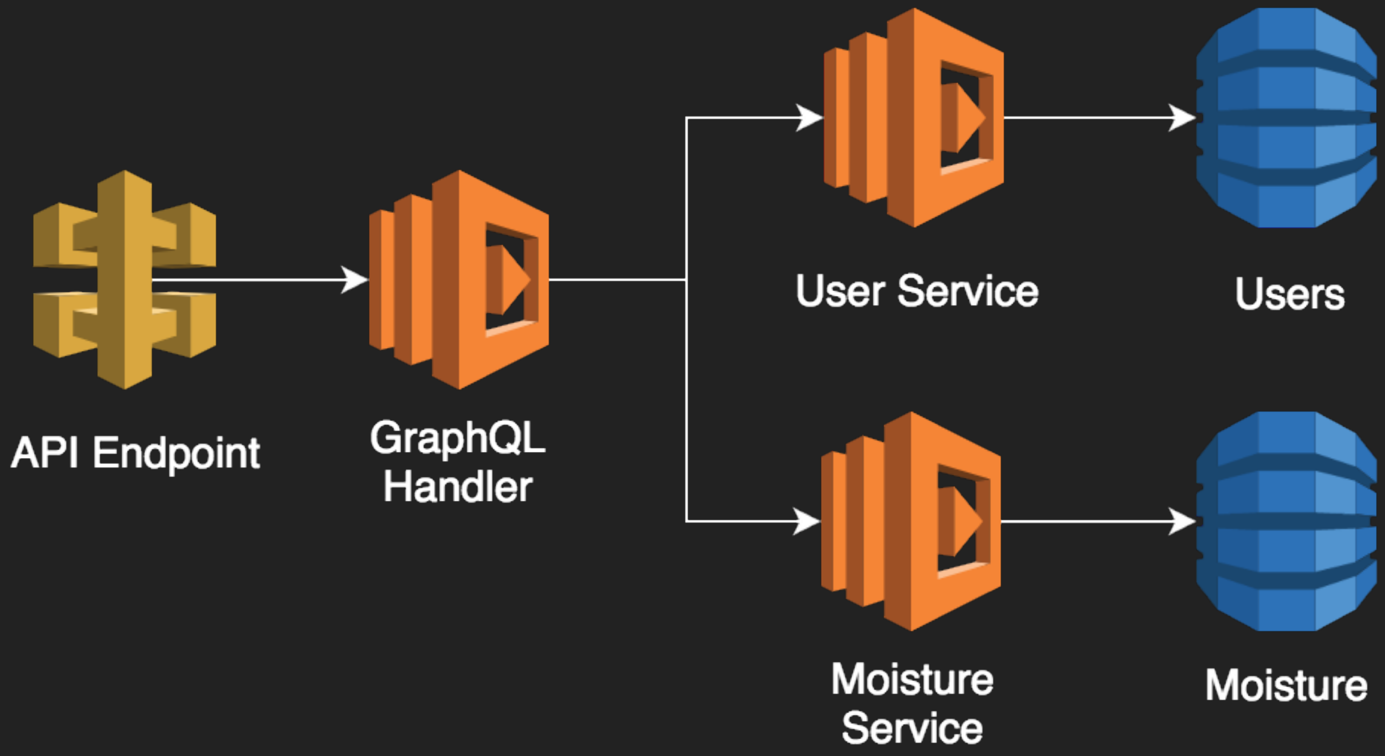
GraphQL on AWS Lambda

Single Lambda Design



GraphQL on AWS Lambda

Lambda Tree Design



Summary

Serverless + IoT

My Experiences

- No server operations
- Cost - $0
- Use *aaS services
- Focus on developing functionality
- Iterate quickly & scale

Alternative Options

IoT Service

Device Shadows

- Stores Device State
- Get current state
- Track state

Alternative Options

Notifications Service

- Monolithic Notification Lambda
- Other notification services
    - Facebook Messenger
    - Sms - Twillio, Nexmo

Alternative Options

Web Services

- Front-end Framework
  - Angular
  - Vue
- Elastic Search instead of DynamoDB
- Web Service own Data Store

What did I learn?

Many things

- Know your services well
- Know what services exist
- Selecting Boundaries is hard
- Automation is always worth it
- GraphQL is awesome

Resources

Code + Reading

- github.com/garden-aid
- serverless.zone

Frameworks & Tools

- serverless.com
- AWS
- Firebase
- Auth0

Thanks for Listening!

Questions?

johncmckim.me

twitter.com/@johncmckim

medium.com/@johncmckim