Web Development: Module 2, Lesson 6  
Moving to Azure Table Storage Hands-On Lab

## Overview

Building on the [Module 2 Lesson 5 Lab](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Labs), we will implement a persistent storage solution for our microblog. Without a persistent storage solution, all your data will be lost every time your server crashes or you need to restart it. Obviously, people won't tolerate such loss, so here we will implement persistent storage in the cloud.

## Objectives

In this hands-on lab you will learn how to:

* Implement CRUD using Azure Table Storage NoSQL remote database

## Prerequisites

The following are required to complete this hands-on lab:

* A code editor
* Windows PowerShell, Mac Terminal, or some other shell with node.js and npm installed
* You should have completed [Module 2 Lessons 1, 2, 3, 4, and 5](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) as well as the [corresponding labs](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Labs).

## Exercises

This hands-on lab includes the following exercises:

* Exercise 1: Moving to Azure Table Storage

## Exercise 1: Moving to Azure Table Storage

In this exercise, you will move locally stored data into Azure table storage. Note: Be sure to refer to the [Module 2 Lessons](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) throughout this exercise. See [Module 1 Lesson 1](https://github.com/MSFTImagine/computerscience/blob/master/Complimentary%20Course%20Content/Module1/Labs/) for information on getting an Azure account.

NOTE: The following resource should prove helpful:

* <https://github.com/github/fetch>

The exercise will be completed if you follow these steps:

1. Enter the Azure portal (<https://portal.azure.com>). You should have a subscription from the [Module 1 Lesson 1 Lab](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module1/Labs).
2. Create a new project folder, and copy your [code/lesson5/lab2/app.js](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson4/lab2) code into it
3. Create start.sh, add it to .gitignore and copy your key and storage name into start.sh
   1. start.sh should look like this:

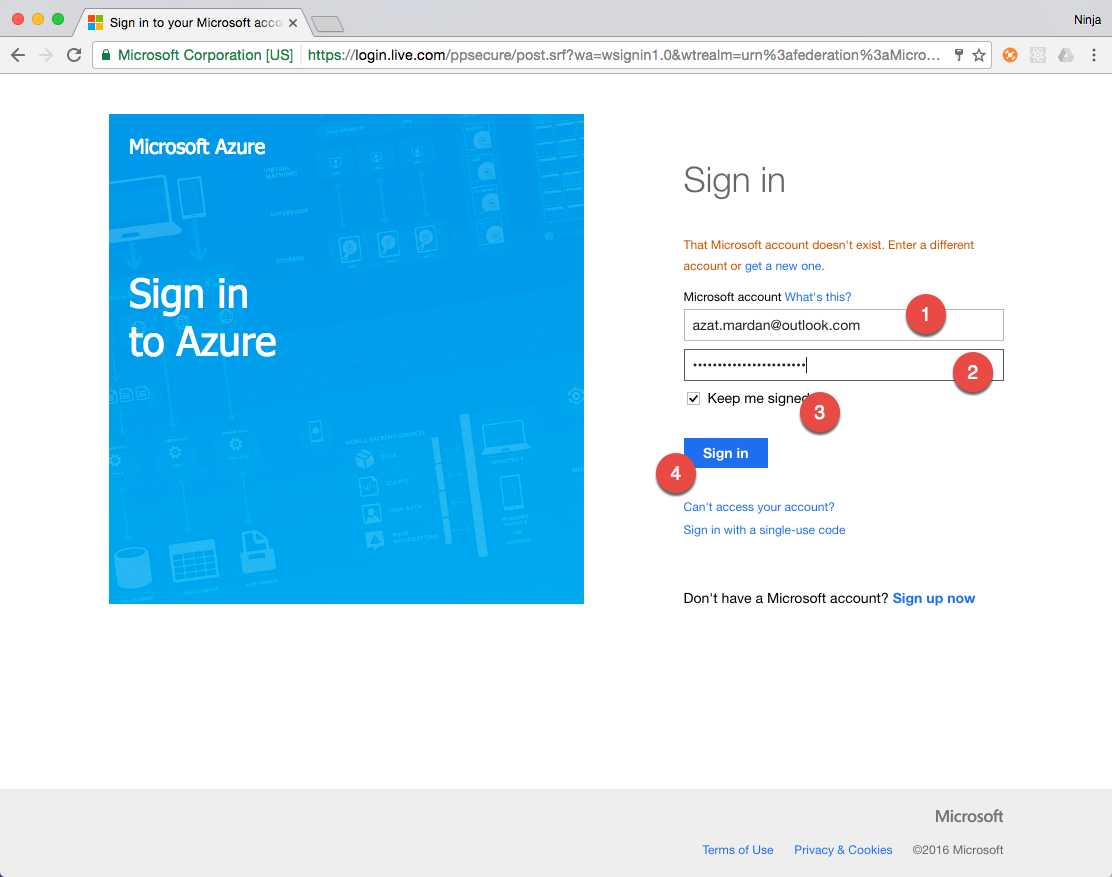
AZURE\_STORAGE\_ACCOUNT=name AZURE\_STORAGE\_ACCESS\_KEY=key node index.js

1. Repeat the same for test.sh: create it, add it to .gitignore and copy your key and storage name into test.sh. test.sh should look like this:

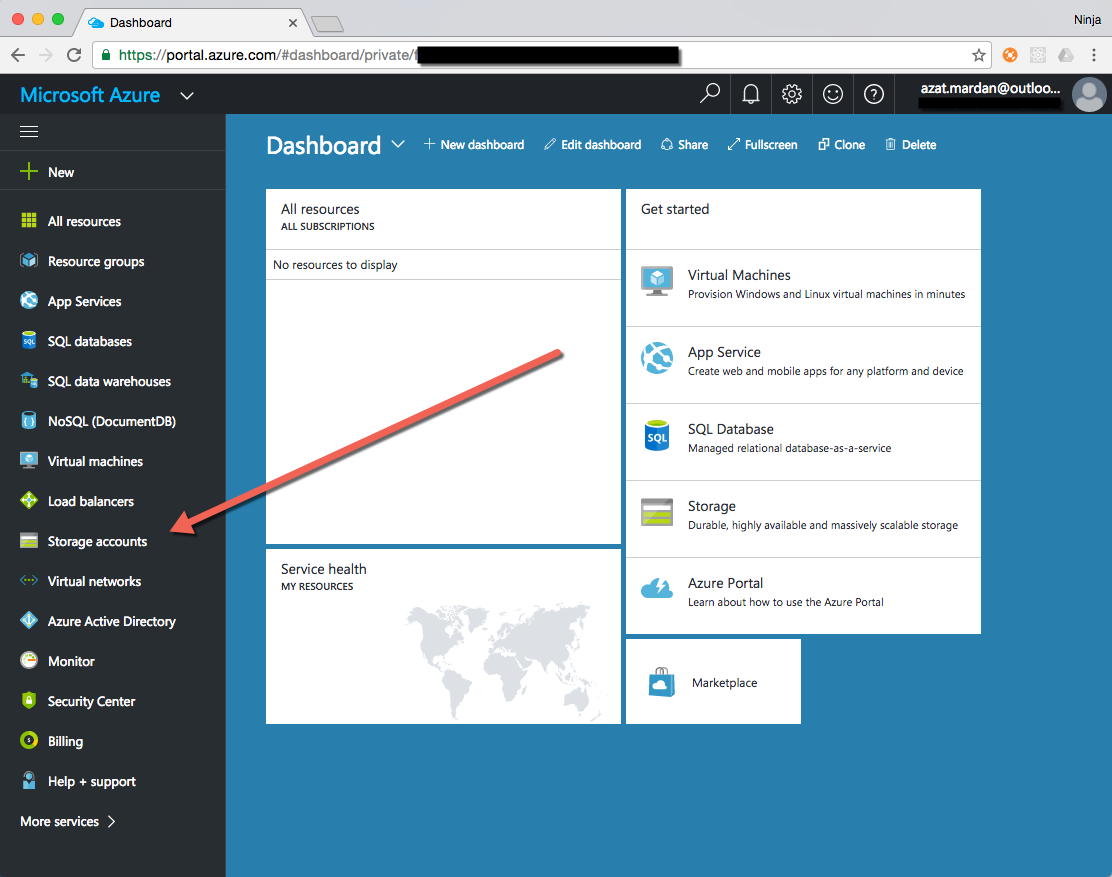
AZURE\_STORAGE\_ACCOUNT=name AZURE\_STORAGE\_ACCESS\_KEY=key ./node\_modules/mocha/bin/mocha app.test.js

1. Use [code/lesson6/Lab/package.json](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson6/Lab) and npm i to install azure-storage or install it manually
2. Modify app.js (Node/Express server code) to work with Azure storage so that each route such as GET, POST, PUT and POST work with the database and not with the in-memory array
3. Use npm test to verify that your server is working
4. Compare your solution with [code/lesson6/Lab/app.js](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson6/Lab)
5. Add static middleware to serve content from public.

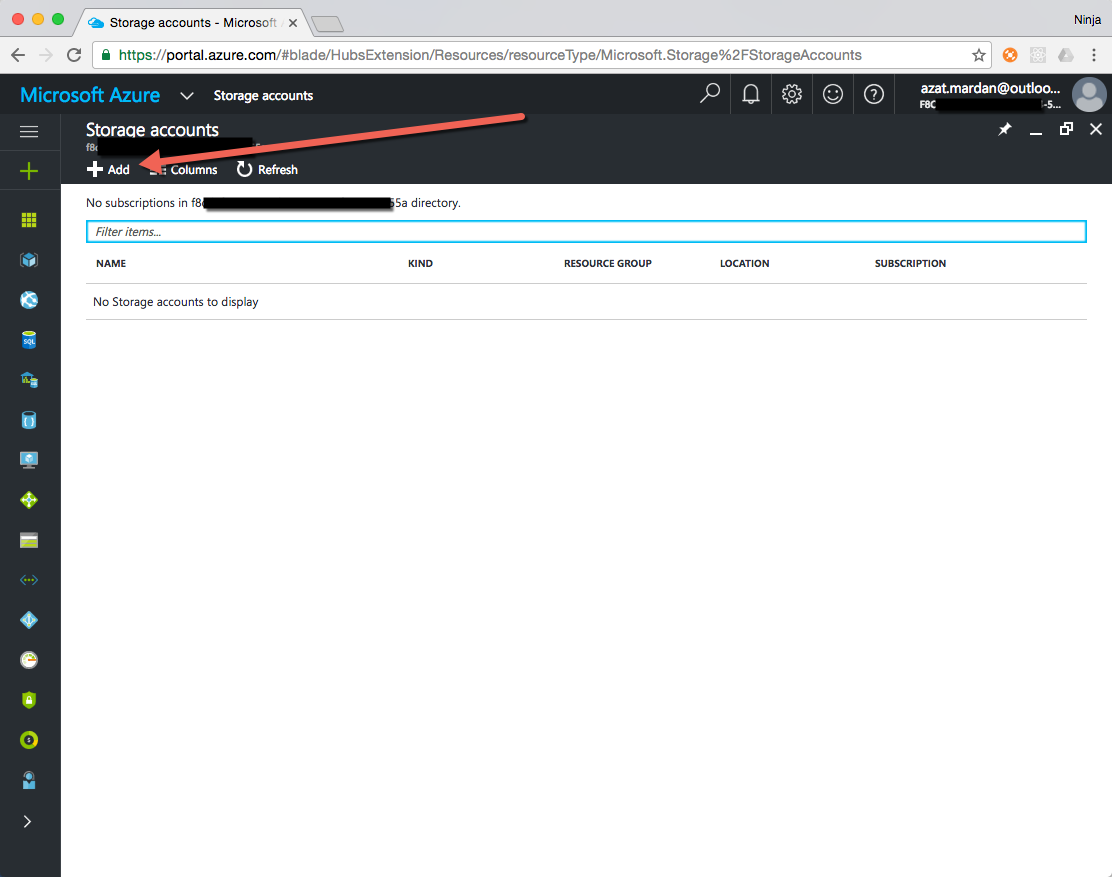
Let’s walk through each of these steps in details. First, you need to have an account with Azure to be able to use their storage service. Go to https://portal.azure.com to sign up for an Microsoft/live.com/outlook.com account if you don't have one. If you have it already, you can use it to sign in. See the screenshot below:



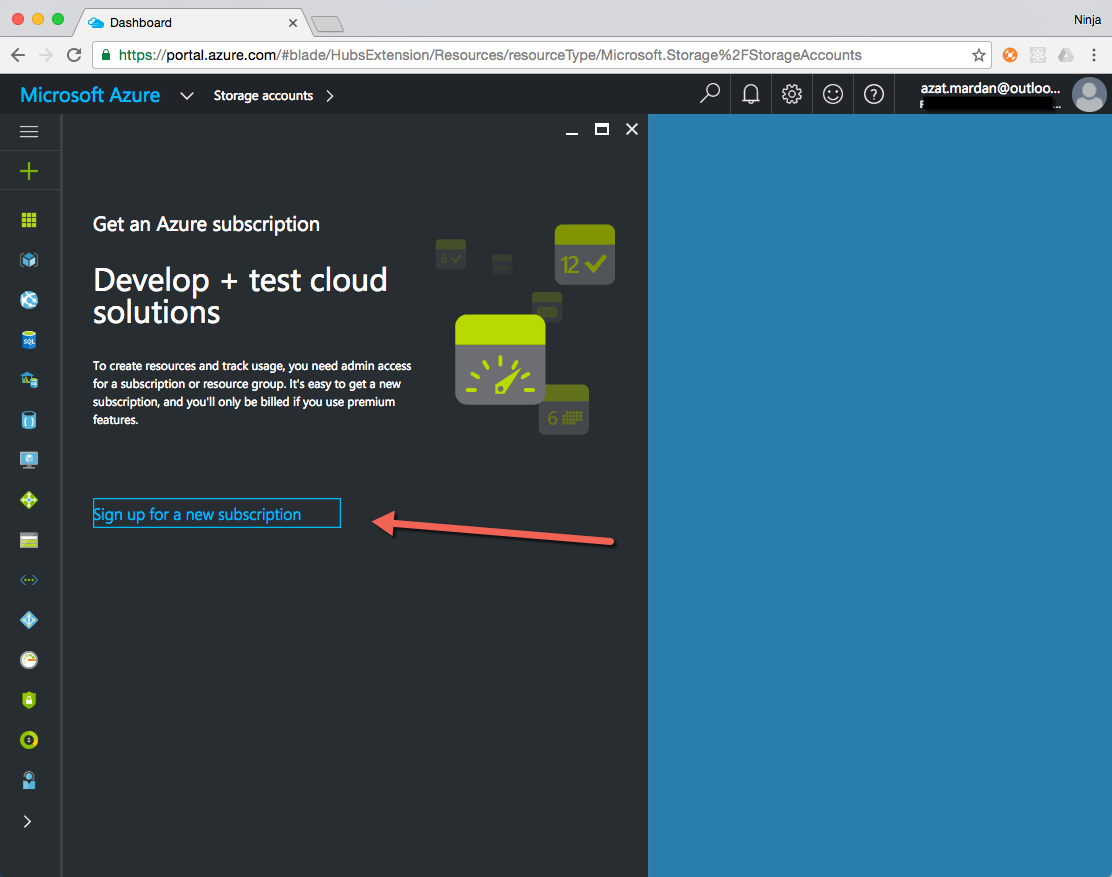
Once you enter the portal, you'll need a subscription to be able to add the storage. You can proceed to storage in the left menu as shown below.



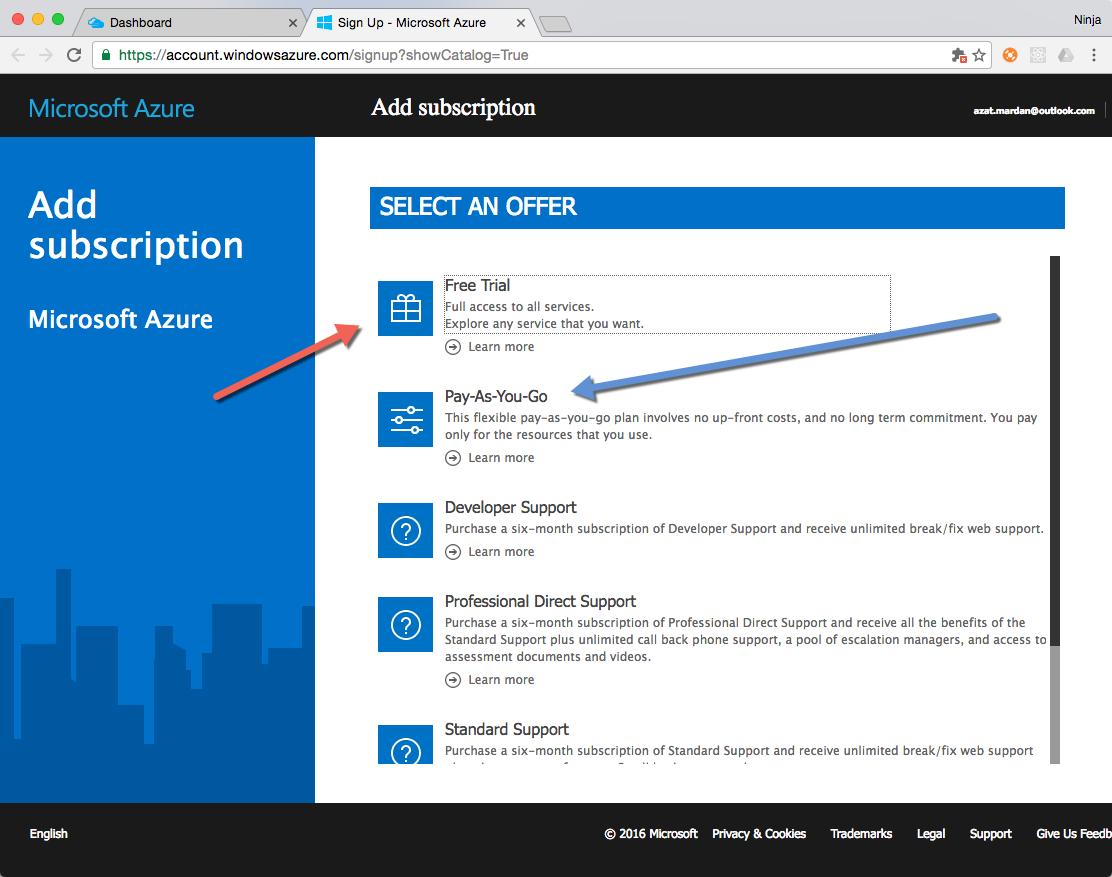
If you have a brand new account with Azure, then you're list of storages and subscription will be empty as mine (shown in the screenshot below). You can click on "Add" to add a new storage.



Most likely you don't have any existing subscription and thus you'll see the page shown below. Click on *Sign up for a new subscription*.



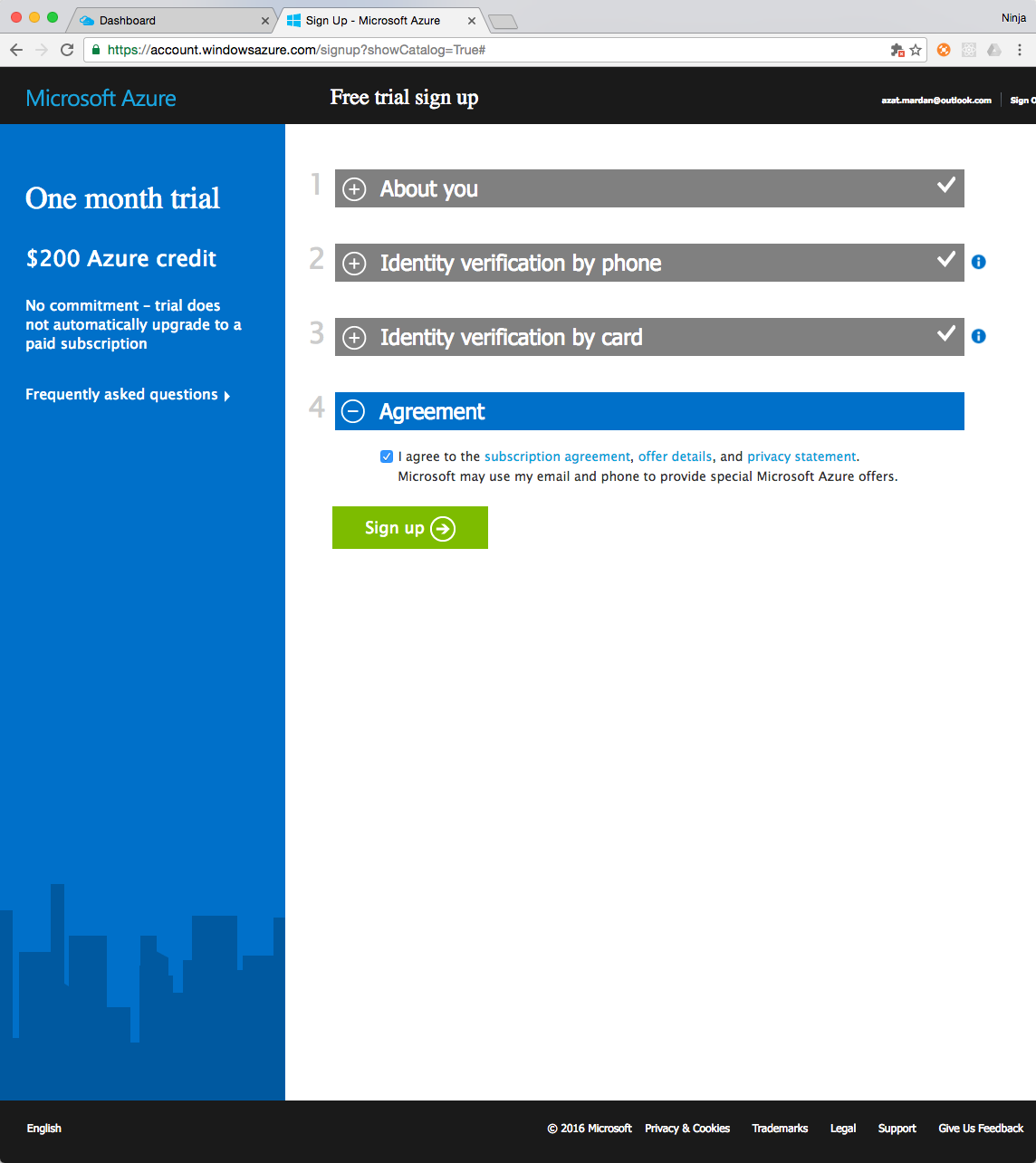
At this point, you'll need a phone number (verification) and a credit card. You might be eligible for a trial account in this case you won't be charged and credit card will be only used for verification and future payments. Otherwise, you can select Pay-As-You-Go. See the screenshot below.



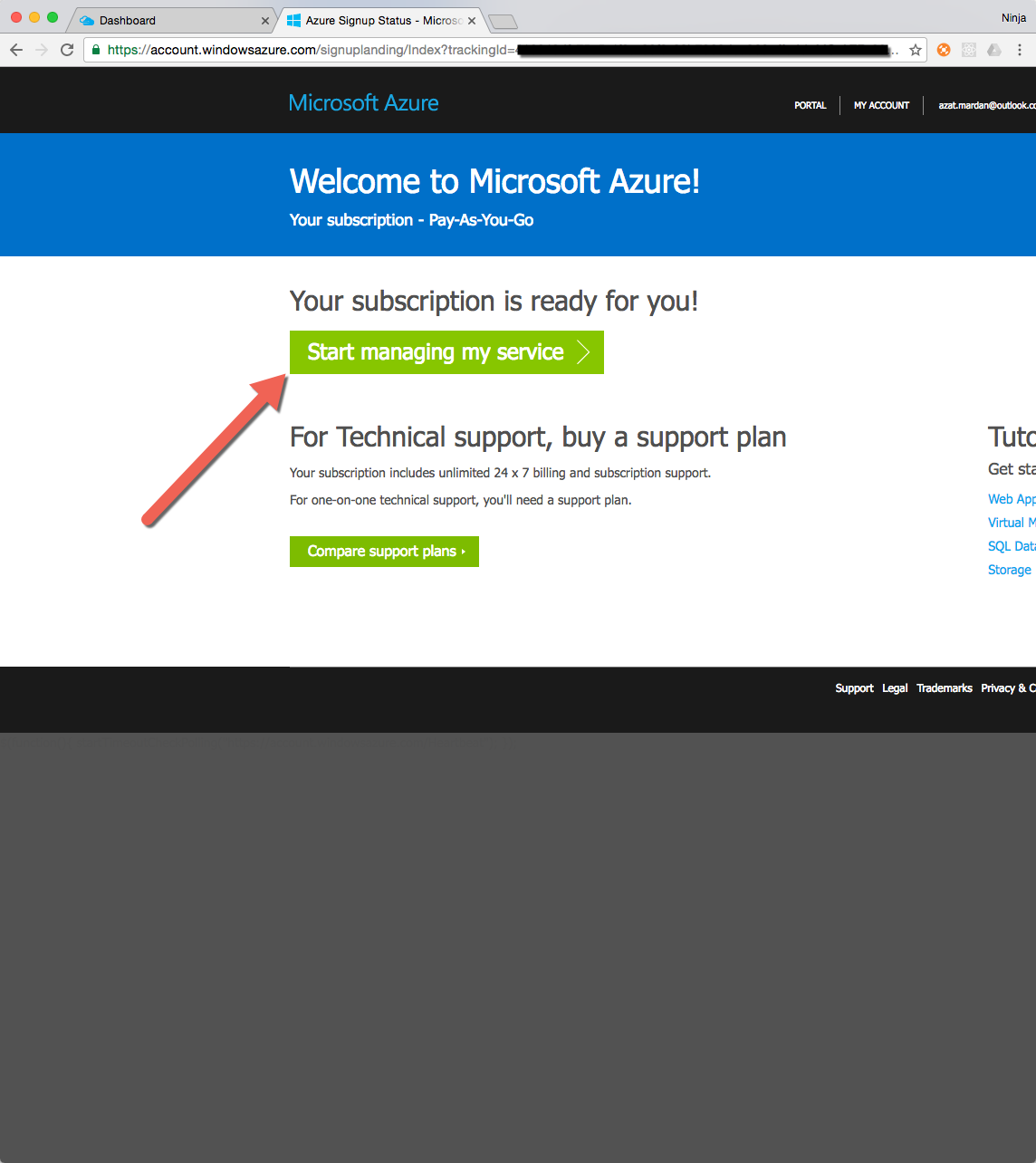
In case you're eligible for a Free trial subscription, you'll need to fill all the information in sections

* About you
* Identify verification by phone (VoIP and some international phone numbers might not work)
* Identity verification by card
* Agreement

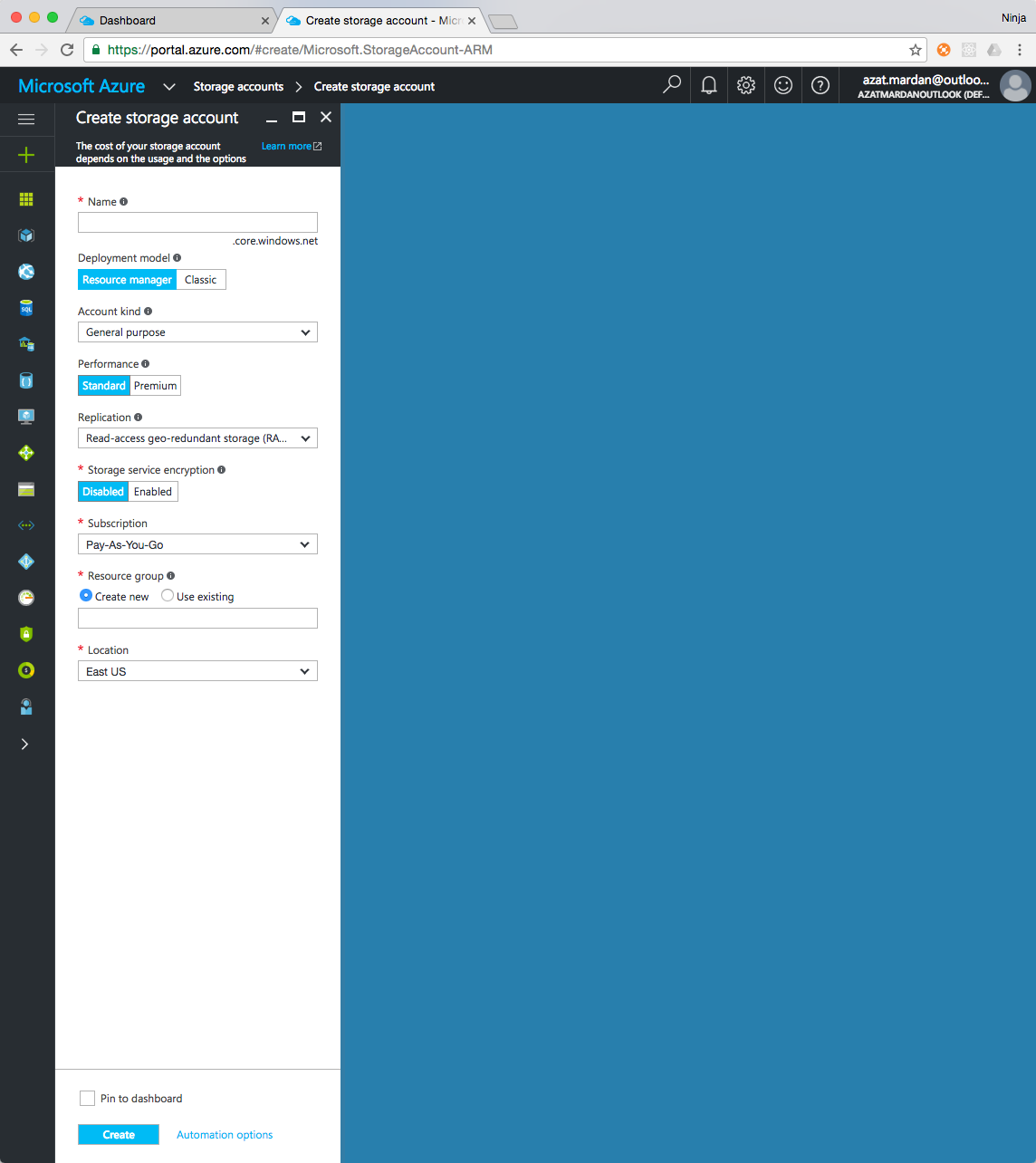
See the screenshot below



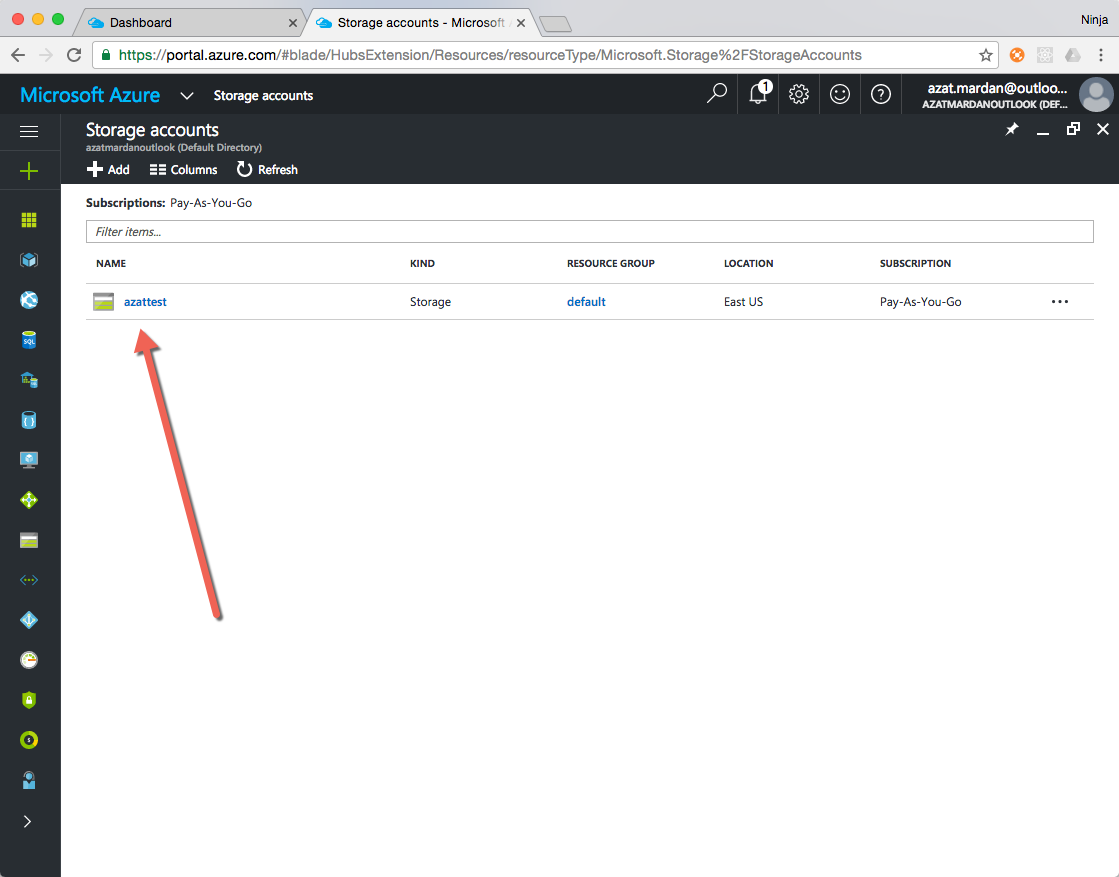
In case you cannot get a trial subscription or ready for a paid plan, then you'll see this message once you provide the payment and the information. Go ahead and click on the green button *saying Start managing my service*:



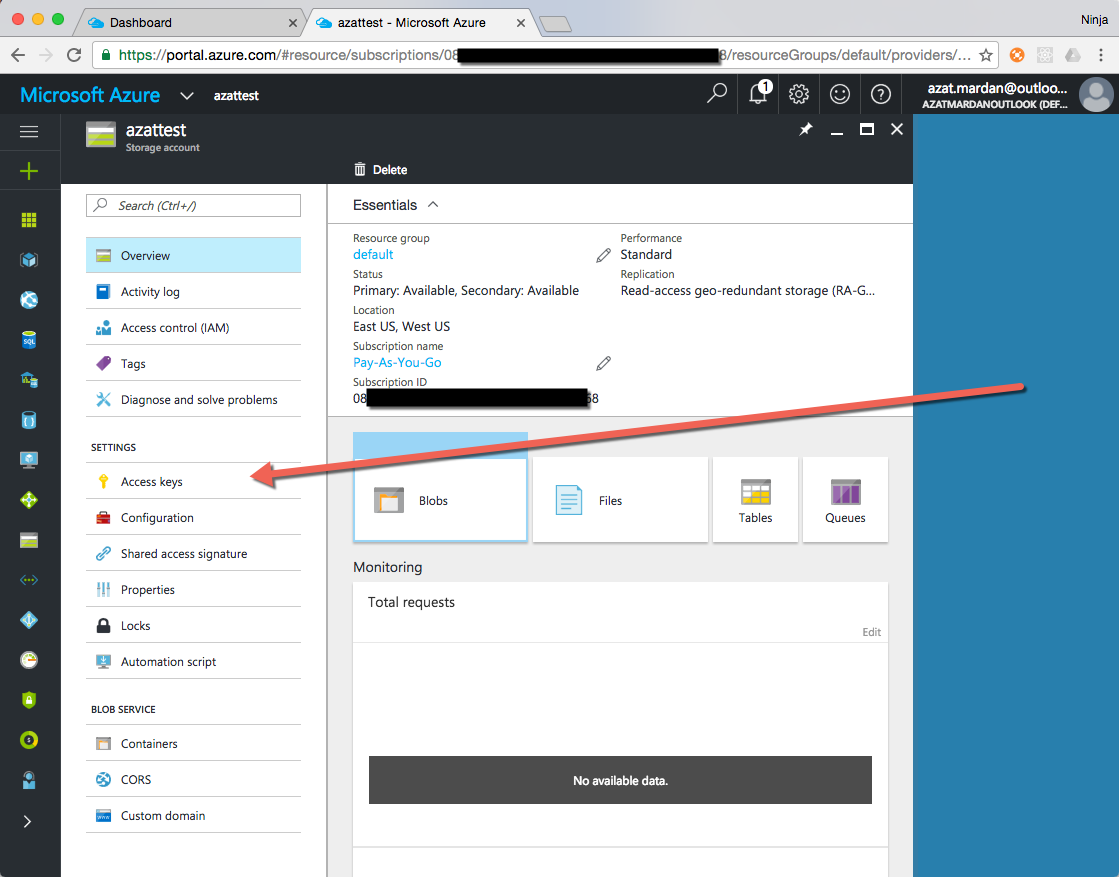
You'll be taken to the storage creation page where you can specify the name and other options. At the bare minimum set the name and the group. You can pick arbitrary names.



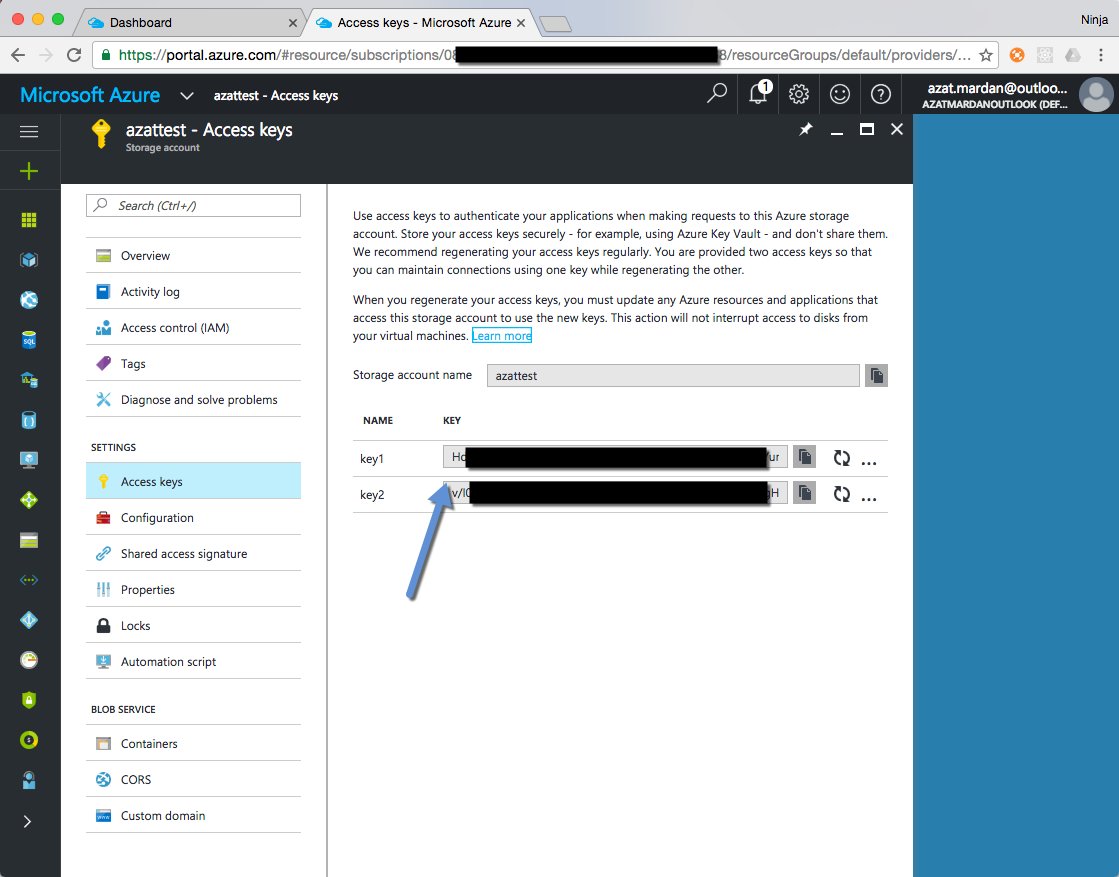
Once the storage is created, you'll see it in the list of storage accounts.



Now click on the name which will open the account in a detailed Overview view as shown below. Finally, we can get to the keys by clicking on *Access keys* in the left sidebar under *Settings*.



Copy the name and the key. We will need them so our app can access the storage.



At last we can start coding. Create a new folder for this project:

$ mkdir azure-table

$ cd azure-table

Right after, copy files from the lab on REST services (module-2-lesson-5-lab) so we can build on the previously implemented project. The project structure will look like this when we done:

/module-2-lesson-6-lab

/node\_modules

.gitignore

app.js

app.test.js

package.json

README.md

start.sh

test.sh

Let's copy the key and name of the storage account you created in Azure to the shell files so our Node scripts can access those values from environment variables (env vars are the best practice because they'll allow you to NOT store sensitive information in the source code).

1. Create start.sh
2. Add Azure key and storage name to start.sh in the following format: AZURE\_STORAGE\_ACCOUNT=name AZURE\_STORAGE\_ACCESS\_KEY=key node app.js
3. Repeat step 2 for test.sh: AZURE\_STORAGE\_ACCOUNT=name AZURE\_STORAGE\_ACCESS\_KEY=key ./node\_modules/mocha/bin/mocha app.test.js
4. Add start script to package.json: "start": "sh start.sh" for convenience
5. Add test script to package.json: "test": "sh test.sh" for convenience

Install azure-storage library which will help us connect to Azure storage from Node:

npm i azure-storage@1.0.1 -S

We cannot run test or the server yet because we didn't change anything but when we are ready, we can use npm script commands which will run shell scripts with env vars:

1. Run npm test to test with Azure (first time you run it, it will create table so run it again)
2. Run npm start to start the server on 3000 with connection to Azure

Following the setup, we move on to implementing of the tests

**Implementing Azure Table REST API Tests**

The The test file is very similar to the one in which we used the inmemory REST API except that the Azure Table data will have a slightly different format. It will contain RowKey.\_ field. Therefore, in responses, we need to check for that field (Of course, if you prefer, you can eliminate this field from your responses in the API server, then you won't need to test for it or use it on the client side. In the end, it comes down to the API design.)

At a higher level there are these parts in app.test.js:

var superagent = require('superagent')

var expect = require('expect.js')

var app = require('./app.js')

let baseUrl = 'http://localhost:3007/api'

before(function(){

// Prepare for the tests (optional)

})

describe('express rest api server', function(){

var id // ID shared between multiple it statements

it('posts an object', function(done){

// Create a new post

})

it('retrieves an object', function(done){

// Fetch the post

})

it('retrieves a collection', function(done){

// Fetch the list of posts

})

it('updates an object', function(done){

// Update the post

})

it('checks an updated object', function(done){

// Check that the updated post has new values

})

it('removes an object', function(done){

// Remove the post

})

it('checks an removed object', function(done){

// Check that the post is no longer present

})

})

after(function(){

// Clean up (Optional)

})

This is the final version of the test for the Azure Table REST API which tests CRUD for /posts:

var superagent = require('superagent')

var expect = require('expect.js')

const app = require('./app.js')

let baseUrl = 'http://localhost:3007/api'

before((done)=>{

app.listen(3007, done)

})

describe('express rest api server', function(){

var id

it('posts an object', function(done){

superagent.post(`${baseUrl}/posts`)

.send({ author: 'John',

text: `There's a better alternative to the ubiquitous JSON as the communication protocol of the web. It's Protocol Buffers (protobuf). In a nutshell, protobuf offers a more dense format (faster processing) and provides data schemas (enforcement of structure and better compatibility with old code). `

})

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body['RowKey'].\_).to.be.ok

id = res.body['RowKey'].\_

done()

})

})

it('retrieves an object', function(done){

superagent.get(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.RowKey.\_).to.be.ok

expect(res.body.RowKey.\_).to.eql(id)

expect(res.body.author.\_).to.eql('John')

done()

})

})

it('retrieves a collection', function(done){

superagent.get(`${baseUrl}/posts`)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body.length).to.be.above(0)

expect(res.body.map(function (item){return item.RowKey.\_})).to.contain(id)

done()

})

})

it('updates an object', function(done){

superagent.put(`${baseUrl}/posts/`+id)

.send({author: 'Peter', id: id})

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.msg).to.eql('success')

done()

})

})

it('checks an updated object', function(done){

superagent.get(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.RowKey.\_).to.eql(id)

expect(res.body.author.\_).to.eql('Peter')

done()

})

})

it('removes an object', function(done){

superagent.del(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.msg).to.eql('success')

done()

})

})

it('checks an removed object', function(done){

superagent.get(`${baseUrl}/posts/`)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body.map(function (item){return item.RowKey.\_})).to.not.be(id)

done()

})

})

})

after(()=>{

process.exit()

})

Obviously, running the tests now will most likely yield an error. We should add Azure to our server next.

**Implementing REST API with Azure Table**

Building upon the lab with in-memory storage REST API, we are going to have these parts in app.js:

var express = require('express'),

bodyParser = require('body-parser'),

logger = require('morgan'),

crypto = require('crypto')

//let posts = require('./posts.json')

let tableName = 'microblogdev'

let partitionKey = 'postsPartitionA'

var azure = require('azure-storage')

var tableSvc = azure.createTableService()

tableSvc.createTableIfNotExists(tableName, function(error, result, response){

// Error handling

})

var entGen = azure.TableUtilities.entityGenerator

var app = express()

app.use(bodyParser.json())

app.use(bodyParser.urlencoded({extended: true}))

app.use(logger('dev'))

app.get('/', function(req, res, next) {

res.send('please select a collection, e.g., /posts')

})

app.get('/api/posts', function(req, res, next) {

// Query Azure and send back the list of posts

})

app.post('/api/posts', function(req, res, next) {

// Create an object from JSON payload, insert into Azure Table and send back the results

})

app.get('/api/posts/:id', function(req, res, next) {

tableSvc.retrieveEntity(tableName,

partitionKey,

req.params.id,

function(error, result, response){

if (error) return next(error)

// result contains the entity

res.send(result)

}

)

})

app.put('/api/posts/:id', function(req, res, next) {

// Massage the incoming data and update the record in Azure Table, send back the results

})

app.delete('/api/posts/:id', function(req, res, next) {

// Prepare the data and remove the object/entity from the Azure Table storage

})

if (require.main === module) {

app.listen(3000, function(){

console.log('Express server listening on port 3000')

})

} else {

module.exports = app

}

Phew. Are you ready for the full source code implementation? Let's do it!

Firstly, we import what npm modules we need. crypto is not an npm module. It's a core module meaning we don't have to install it. It's part of Node platform and it'll be available to us for free. We need crypto to create unique IDs for the entities in the Azure Table.

var express = require('express'),

bodyParser = require('body-parser'),

logger = require('morgan'),

crypto = require('crypto')

Speaking of Azure Table, let's save name of the table and partition key since we'll be using it multiple times in various queries.

let tableName = 'microblogdev'

let partitionKey = 'postsPartitionA'

Next, we import the azure-storage library which will allow us to connect to the remote Azure storage, make queries to fetch and update data.

var azure = require('azure-storage')

Keep in mind that before we can work with the data we should create the table. We can use createTableService() for to get the table service object and then createTableIfNotExists which you might guess will create a new table if it does NOT exist already. Hence, every time we start a server, it will make sure we have a table to work with. The best practice is to handle errors in each callback which we can do by checking for !error, i.e., error is not null.

The reason why azure will be able to locate exactly *your account* is because you put the storage account name and the access key into the shell script. The shell script will populate environment variables which will be used by the azure-storage library to make the proper connection. Magic!

var tableSvc = azure.createTableService()

tableSvc.createTableIfNotExists(tableName, function(error, result, response){

if(!error){

console.log('Table exists or created', result)

} else {

console.log('Error creating table', error)

}

})

Using azure reference to the azure-storage library, we create an alias for entityGenerator which will help us create Azure Table entities.

var entGen = azure.TableUtilities.entityGenerator

The next step is to instantiate the Express app and apply the middleware:

var app = express()

app.use(bodyParser.json())

app.use(bodyParser.urlencoded({extended: true}))

app.use(logger('dev'))

Everything is ready to start implementing routes. The first will be GET to / and /posts. The former is straightforward while the latter will need a query object which we can create with azure.TableQuery(). The query is limited by top and where clauses. In where, we fetch the data using the table name and query. In the callback, we check for error to be null and send results to the client using res.send().

app.get('/', function(req, res, next) {

res.send('please select a collection, e.g., /posts')

})

app.get('/api/posts', function(req, res, next) {

var query = new azure.TableQuery()

.top(100)

.where('PartitionKey eq ?', partitionKey) // Use partition key

tableSvc.queryEntities(tableName, query, null, function(error, result, response) {

if (error) return next(error) // Ooops, error. Exit.

// Query was successful

res.send(result.entries)

})

})

Okay. The next step is to implement the insertion of new posts which is POST to /api/posts. This route will need the help of crypto to generate a unique ID based on the author name (req.body.author) and the current timestamp (Date.now()).

app.post('/api/posts', function(req, res, next) {

let id = `${req.body.author}${Date.now()}`

// console.log(id)

id = crypto.createHash('sha256').update(id).digest('hex')

Promptly after generation the ID we can form the entity to store in the table. The structure of Azure Table entities is not the same as JSON. It uses \_. You can use entGen or do it manually as shown below:

var task = {

PartitionKey: {'\_':partitionKey},

RowKey: {'\_': id},

author: {'\_': req.body.author},

text: {'\_': req.body.text},

createdAt: {'\_': new Date(2015, 6, 20), '$':'Edm.DateTime'}

}

Thereafter, you can utilize insertEntity to inject the new entity into your table. The echoContent will show you the resulting entity in the response argument of the callback. We need it to send the newly created data back to the client.

tableSvc.insertEntity(tableName,

task,

{echoContent: true},

function (error, result, response) {

if (error) return next(error) // Oops, something went wrong

res.send(result) // Entity inserted, send it to the client

})

})

So far we haven't implemented a fetching of an individual post. We can do it easily with retrieveEntity. All we need is to provide the table name, partition key, and the entity ID. Boom! You got the data in the result so we can send it back to the client.

app.get('/api/posts/:id', function(req, res, next) {

tableSvc.retrieveEntity(tableName,

partitionKey,

req.params.id,

function(error, result, response){

if (error) return next(error)

res.send(result) // result contains the entity

}

)

})

For the next route (update post), we check for the presence of author or text fields and form the object updatedPost accordingly. We use updatedPost in replaceEntity which takes table name and the data as arguments. The partition key must be part of the updatedPost so tableSvc could find the entity.

app.put('/api/posts/:id', function(req, res, next) {

let updatedPost = {}

if (req.body.author)

updatedPost.author = {'\_': req.body.author}

if (req.body.text)

updatedPost.text = {'\_': req.body.text}

updatedPost.PartitionKey = {'\_': partitionKey}

updatedPost.RowKey = {'\_': req.params.id}

tableSvc.replaceEntity(tableName, updatedPost, function(error, result, response){

if (error) return next(error)

// Entity updated

res.send({msg:'success'})

})

})

Ultimately, we need to be able to remove the posts in our microblog. Hence, the DELETE /api/posts route which uses partition key and row key (entity key) to remove the entity in a table. deleteEntity uses table name along with the post object which has partition and entity keys. We get the URL parameter :id from req.params.id. As previously, we send a success message if there were no errors.

app.delete('/api/posts/:id', function(req, res, next) {

let post = {}

post.PartitionKey = {'\_': partitionKey}

post.RowKey = {'\_': req.params.id}

tableSvc.deleteEntity(tableName, post, function(error, response){

if (error) return next(error)

// Entity deleted

res.send({msg: 'success'})

})

})

The rest of the code is untack from the previous lab. All it does is starting up a server if this script was launched as a standalone Node program (node app.js), or exports the code if this script was imported (require('./app.js')) from another module such as test app.test.js.

if (require.main === module) {

app.listen(3000, function(){

console.log('Express server listening on port 3000')

})

} else {

module.exports = app

}

At this point, if you implemented everything correctly, running npm test should yield success. This is what I have when I run tests and the table is already created. You should get a similar result except for the etag and ID values.

express rest api server

Table exists or created { isSuccessful: true,

statusCode: 200,

TableName: 'microblogdev',

created: false }

POST /api/posts 200 834.654 ms - 772

✓ posts an object (878ms)

GET /api/posts/5aa67eb9b1e28a18d35788d25cac9f259bf04d8e197549c66570ff09c7281bdc 200 549.715 ms - 772

✓ retrieves an object (560ms)

GET /api/posts 200 505.700 ms - 688

✓ retrieves a collection (512ms)

{ '.metadata': { etag: 'W/"datetime\'2016-11-16T22%3A40%3A26.964605Z\'"' } }

PUT /api/posts/5aa67eb9b1e28a18d35788d25cac9f259bf04d8e197549c66570ff09c7281bdc 200 510.261 ms - 17

✓ updates an object (515ms)

GET /api/posts/5aa67eb9b1e28a18d35788d25cac9f259bf04d8e197549c66570ff09c7281bdc 200 514.358 ms - 400

✓ checks an updated object (518ms)

DELETE /api/posts/5aa67eb9b1e28a18d35788d25cac9f259bf04d8e197549c66570ff09c7281bdc 200 517.220 ms - 17

✓ removes an object (521ms)

GET /api/posts/ 200 530.480 ms - 2

✓ checks an removed object (534ms)

If you have any errors, check your code against the official solution provided in the repository. Note: if you are getting *StorageError: Server failed to authenticate the request. Make sure the value of Authorization header is formed correctly including the signature*, then make sure you have the proper storage account name and the access key populated in the environment variables. Check your start.sh and test.sh.

Feel free to start the server in a standalone mode with npm start and try to insert and fetch the data with Postman or CURL. It's fun to watch that your data is persistent in a cloud meaning even if you kill the server (close the terminal or kill the process), the data is safely stored in the cloud and can be accede later by a new instance of the server or from other clients.

## Summary

In this hands-on lab, you learned how to:

* Transfer data into Azure Table Storage
* Use azure-storage Node module
* Form Azure Table entities manually