Web Development: Lesson 8  
Serverless Twitter-Clone Lab

## Overview

Building on [Module 2 Lesson 8](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons), we will move our microblog app to a serverless solution on Azure Functions. This allows for a quick and easy deployment without having to worry about maintenance (e.g. updating patches to Linux). We will use the UI code of the microblog built in previous labs, and deploy a slightly modified version to Azure Blob storage.

The UI from the Blob storage will make requests to Azure Functions to show and add posts.

## Objectives

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

* Create a new Function app
* Create a new function with POST /messages which saves a message
* Create a new function with GET /messages which shows messages
* Create a function to show an HTML page (micro blog UI)

## Prerequisites

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

* A text editor
* Node version 6+ and NPM version 3+
* Windows PowerShell, Mac Terminal, or some other shell with node.js and npm installed
* Completion of all [Module 2 Lessons](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).
* Azure account. See [Module 1 Lesson 1](https://github.com/MSFTImagine/computerscience/blob/master/Complimentary%20Course%20Content/Module1/Labs/) for information on getting an Azure account.
* Git

## Exercise 1: Creating a serverless microblog app utilizing Azure Functions

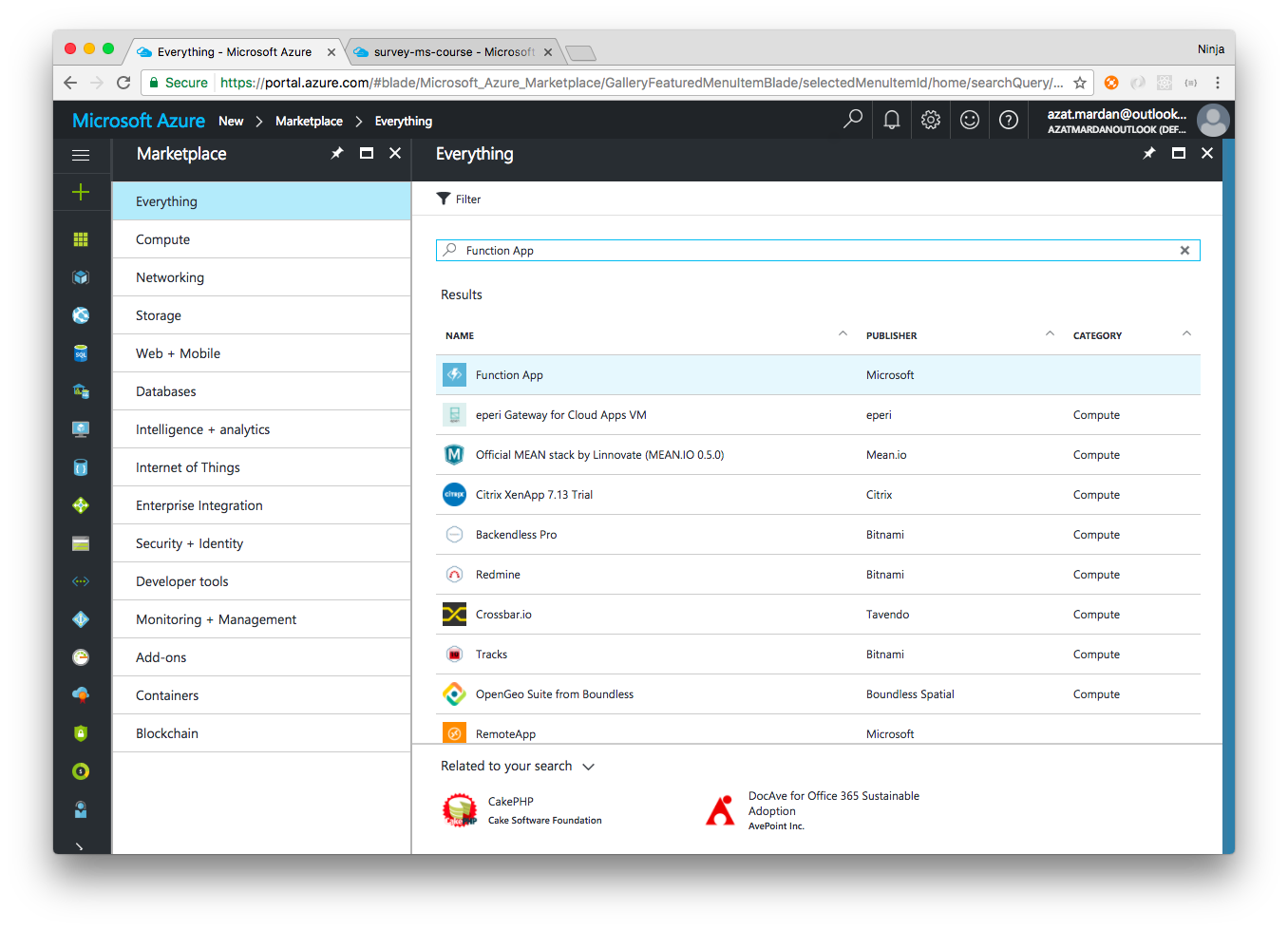
In this exercise, you will use the microblog app developed in the previous lab. Refer to that lesson and lab when needed to complete the following exercise. Here is an overview of the tasks we will walk through:

1. Use an existing Azure Function app or create a new one named microblog. (Function Apps can have multiple functions.)
2. Create a new Azure HTTP Trigger function named posts.
3. Configure trigger inputs and outputs.
4. Implement the API.
5. Set CORS in your Function App settings to allow requests from the blob storage URL.
6. Create an Azure Blob Container.
7. Test the microblog.
8. Optional: Test data in the Storage Explorer app.

To get started, use an existing Azure Function app, or create a new one, e.g., microblog-123. Just to remind you, a Function App is a collection of multiple functions and each function could be triggered by time, HTTP requests or events.

## Step 1: Create an Azure Function App

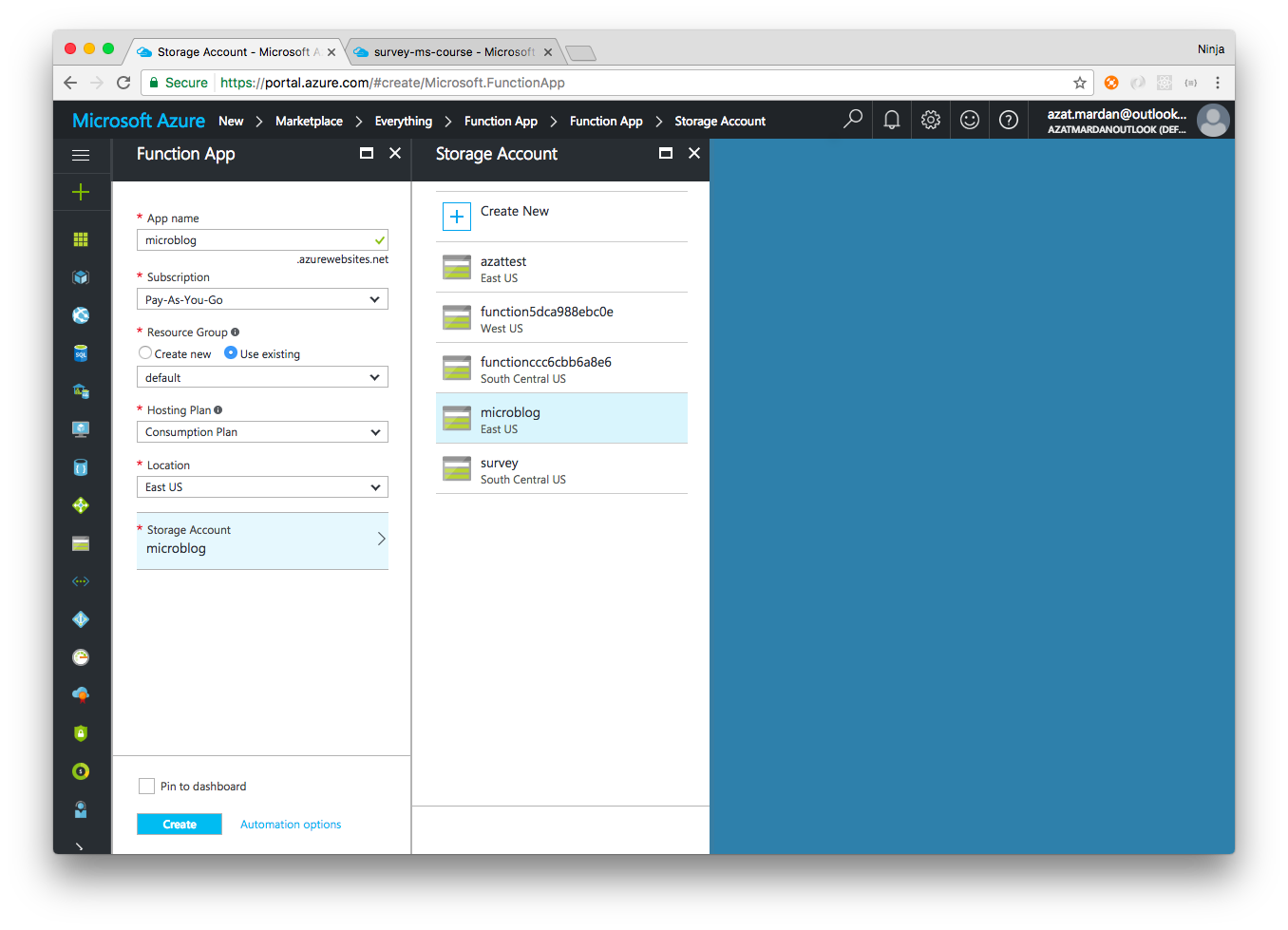
To create a function app, select the “+” button in the left sidebar, type "function app" and select **Function App** in the Results:



Then, select the **Create** button to create a new Azure Function App:



When creating the Function App, specify name and other parameters such as Storage Account. The name must be unique so you won't be able to use "microblog" since we used it already. Also, the app will need to use a Storage Account.

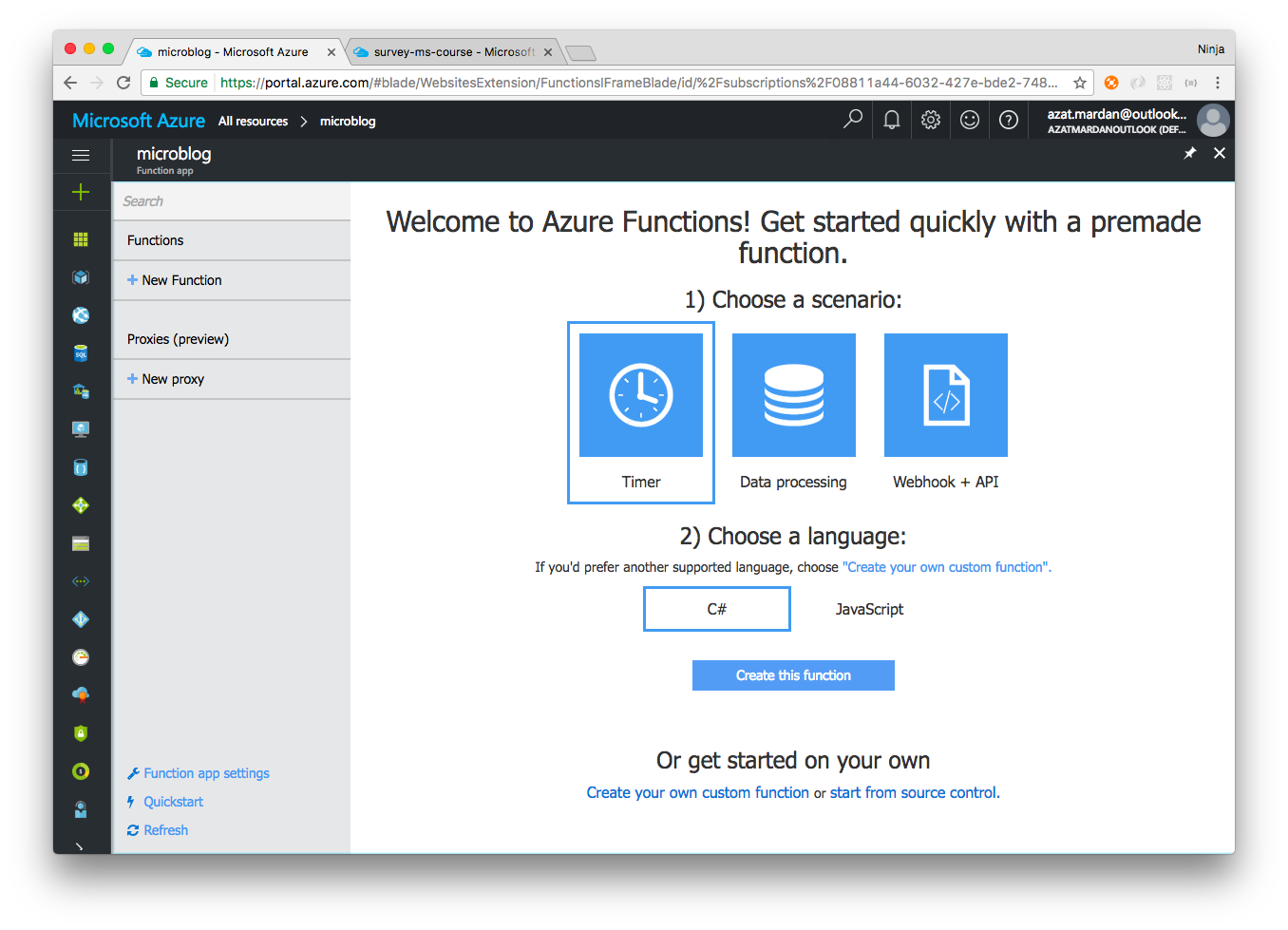


You can re-use an existing storage account (from previous labs) or create a new one.

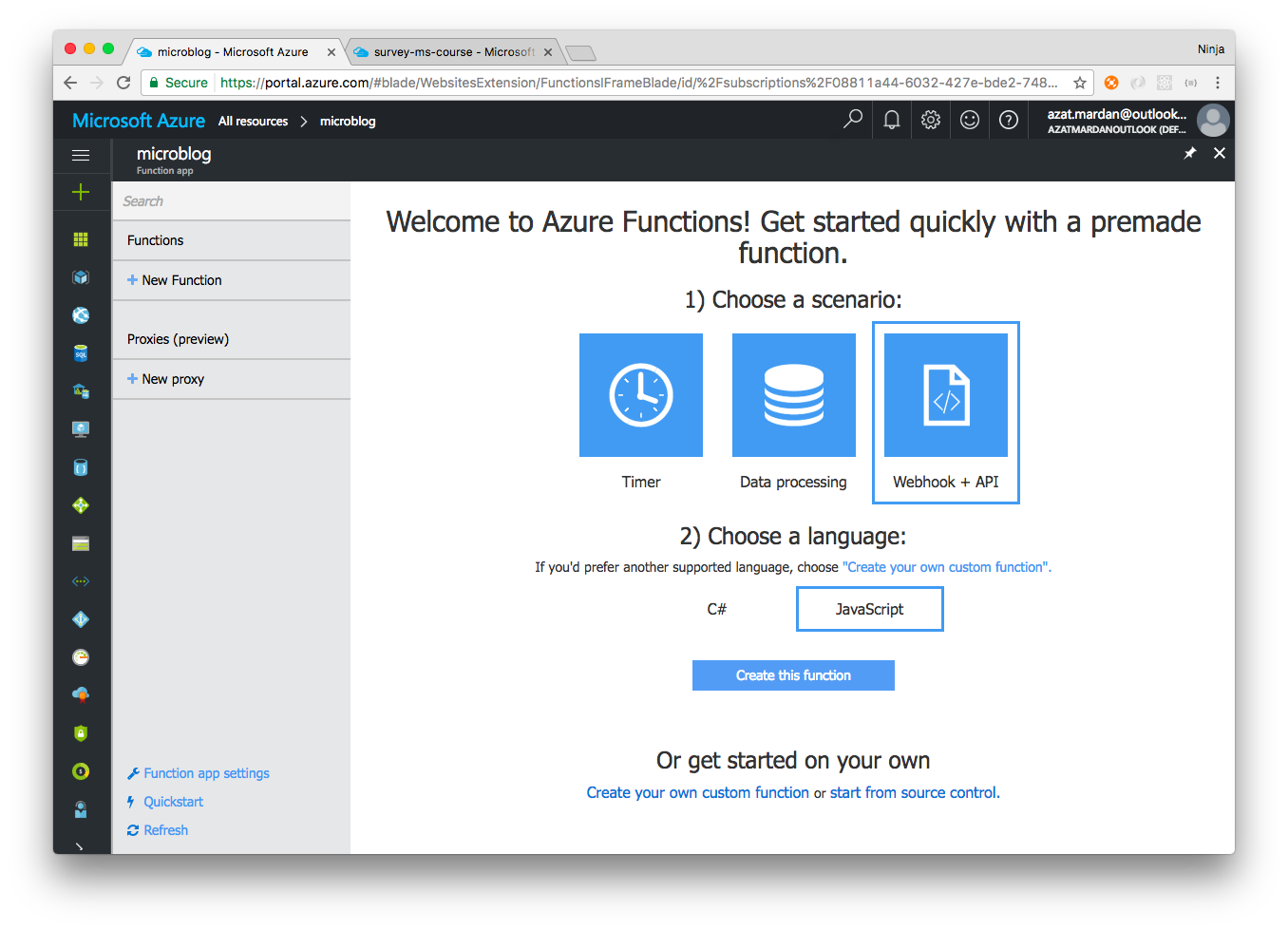
You can use a single storage account to store three things required for this lab: function app (and functions), posts data in Table, and static HTML, JS, CSS files in Blob.

## Step 2: Create an Azure HTTP Trigger Function

Next, create a new Azure HTTP Trigger function named posts.



Select **Webhook + API** (that's an HTTP triggered function) and **JavaScript** (which is actually Node.js):



Enter the function name. This name will appear in the URL. You can change the name later, but currently (Mar, 2017) it is a multi-step and somewhat clunky process (follow these [instructions](https://github.com/Azure/azure-webjobs-sdk-script/wiki/Renaming-a-Function), if you need change the name later). Therefore, choose the function name wisely.

We recommend using one function for a resource. This way you can easily follow the REST methodology. For example, api/accounts function will handle create, read, update, delete (CRUD) for the accounts resource. In our case, we need to handle posts so name the function posts.

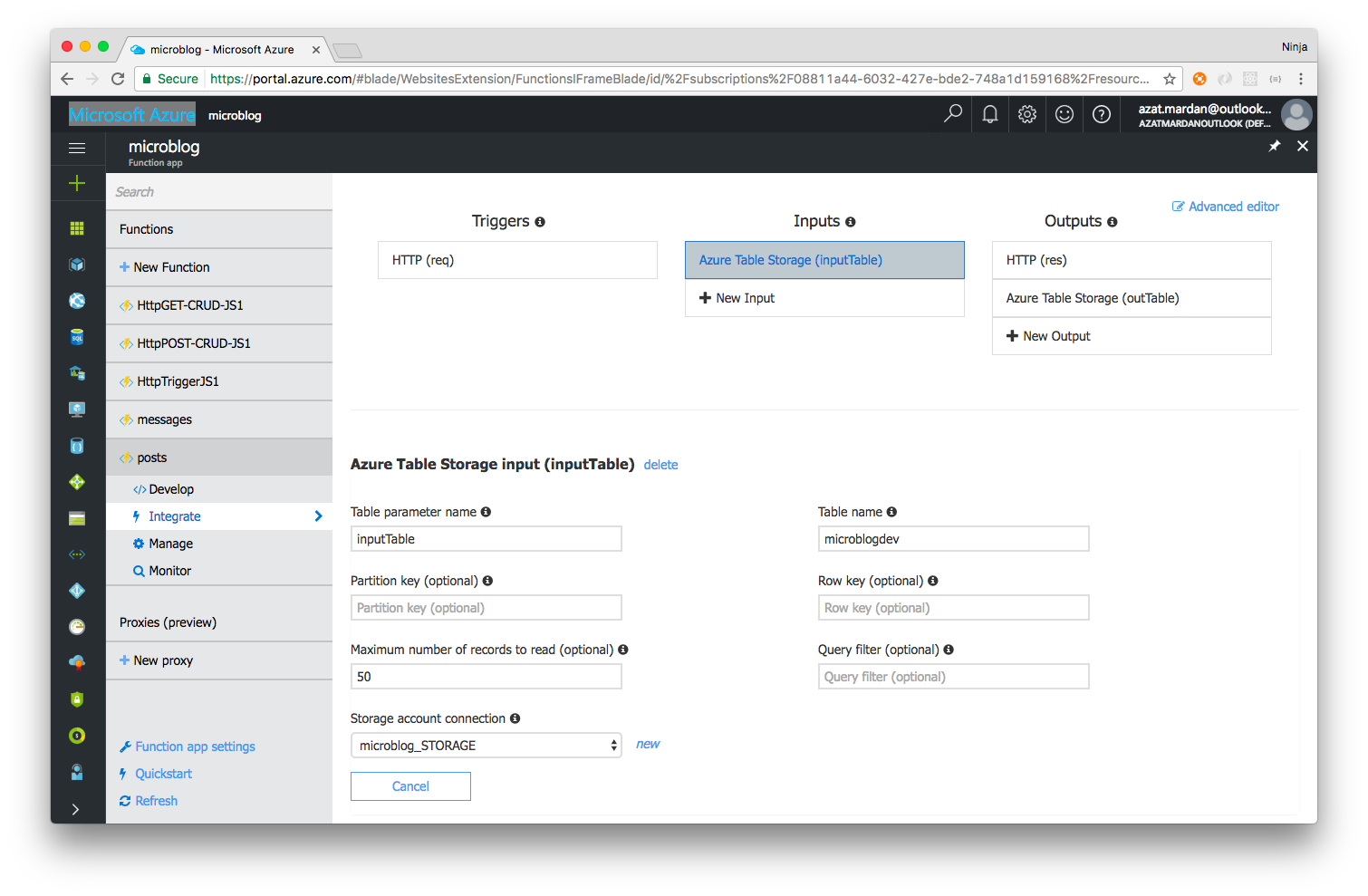
Then, configure the HTTP input trigger by selecting **Integrate** | **Input**. Set the input Authorization level to **Anonymous** in the Integrate pane. This allows you to make requests without using a key (query string parameter ?code=....  in the URL). Of course, if or when you need to have the proper authorization, make sure to change the input trigger to "Function".



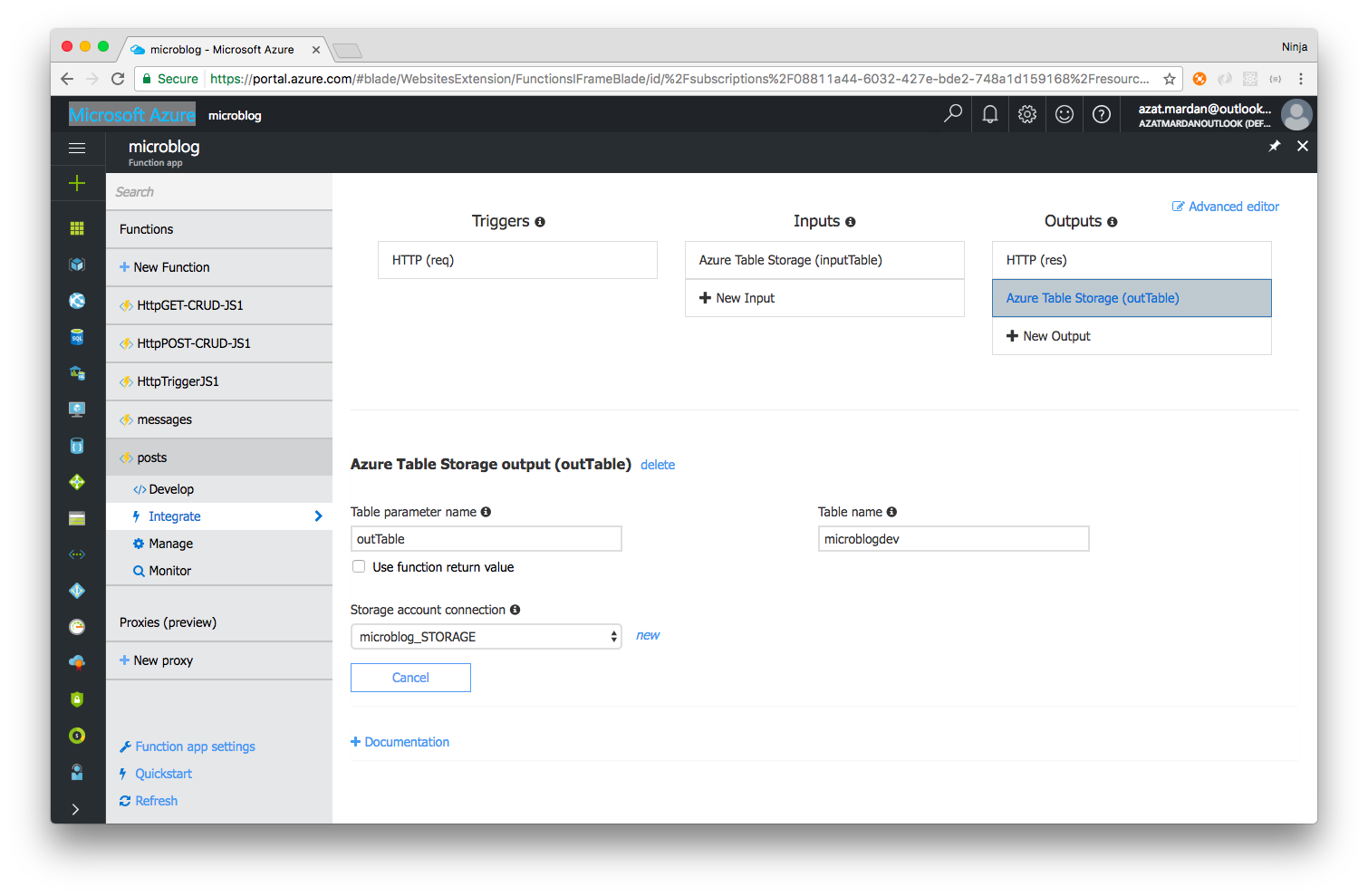
## Step 3: Configure Azure Trigger Inputs and Outputs

While in the Integrate Pane, add an Input and an Output as **Azure Table Storage** with the connection to your Storage account as shown in the next two captures. Create one if you don't have it already (in the Storage account connection pull-down menu).

Make sure you use the right table name. Use the same name you used in the previous labs, e.g., microblogdev.



In the Outputs, connect to the storage account and set the table name.



The HTTP configuration must have res as the name as shown in the screenshot below:



This could be setup another way. Navigate to **Develop** | function.json (in View Files), and set auto row key increment and partition key: {rand-guid}

and postsPartitionA.

Also, make sure ALL OTHER settings are exactly the same as the configuration below with your own associated table names and storage account connections. Avoid typos, by utilizing the following JSON.

{

"bindings": [

{

"authLevel": "anonymous",

"type": "httpTrigger",

"direction": "in",

"name": "req"

},

{

"type": "http",

"direction": "out",

"name": "res"

},

{

"type": "table",

"name": "outTable",

"tableName": "microblogdev",

"partitionKey": "postsPartitionA",

"rowKey": "{rand-guid}",

"connection": "microblog\_STORAGE",

"direction": "out"

},

{

"type": "table",

"name": "inputTable",

"tableName": "microblogdev",

"take": 50,

"connection": "microblog\_STORAGE",

"direction": "in"

}

],

"disabled": false

}

## Step 4: Implement the API

Implement GET /messages and POST /messages code which will read and write from/to Azure Storage Table.

Select **Develop** | index.js. Use req.method to differentiate between HTTP method types. Implement GET /messages code which reads from Azure Storage Table by using the intable argument.

module.exports = function (context, req, intable) {

if (req.method == 'GET') {

context.log("Retrieved records:", intable)

context.res = {

status: 200,

body: intable

}

context.done()

// ...

Next, implement POST /messages code which saves to Azure Storage Table by using context.binding.outTableparameter.

} else if (req.method == 'POST') {

if (!req.body

|| !req.body.author

|| !req.body.text) {

context.res = {

status: 400,

body: 'Invalid request object'

}

return context.done()

}

let {author, text} = req.body

context.bindings.outTable = {

author,

text

}

context.res = {

status: 201,

body: 'Table Storage Created'

}

context.done()

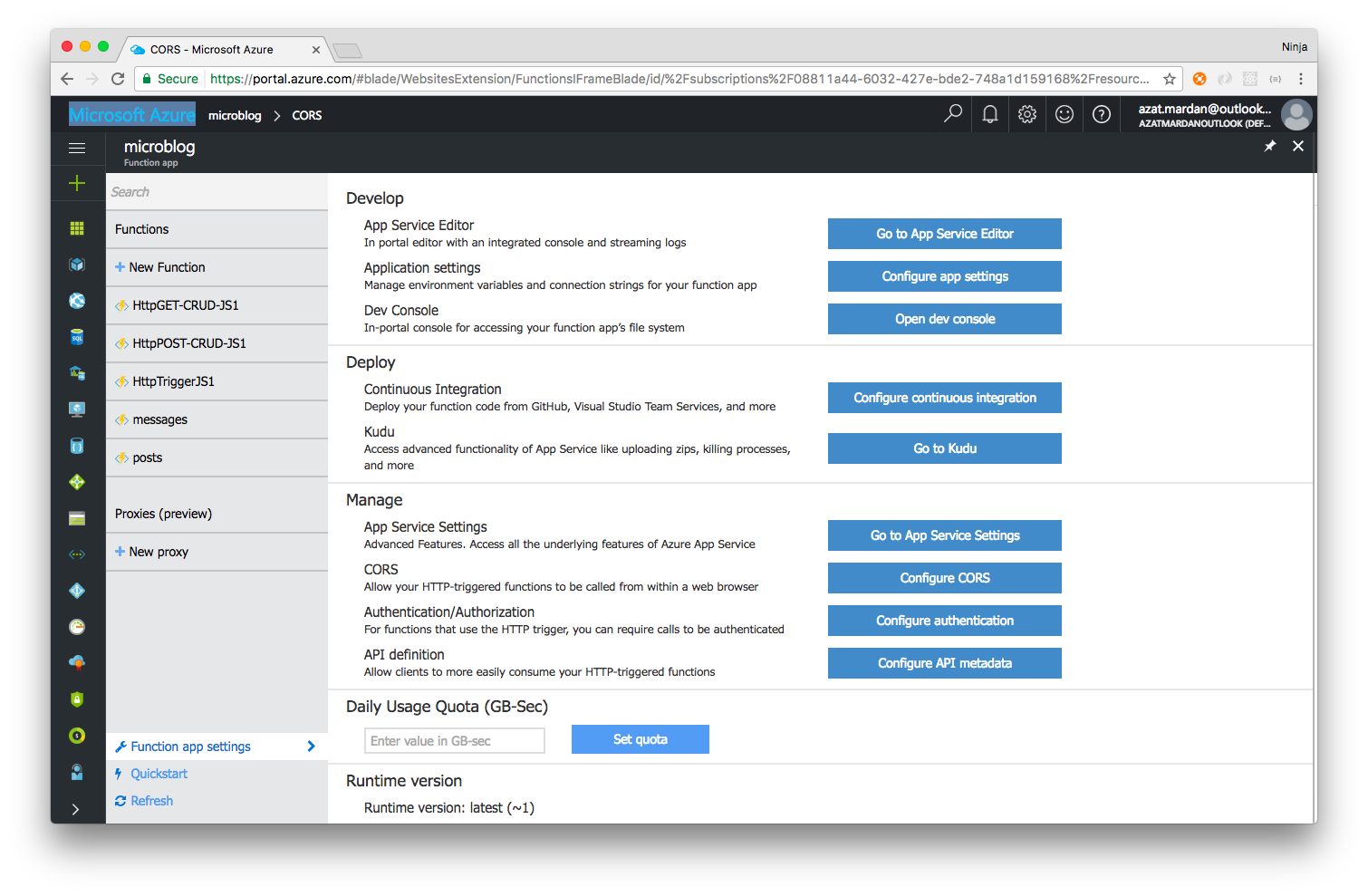
}

}

We are done with the index.js, but we need to enable cross-origin resource sharing (CORS) so our static HTML page hosted on a different domain can make AJAX/XHR calls to the API.

## Step 5: Enable Cross-Origin Resource Sharing (CORS)

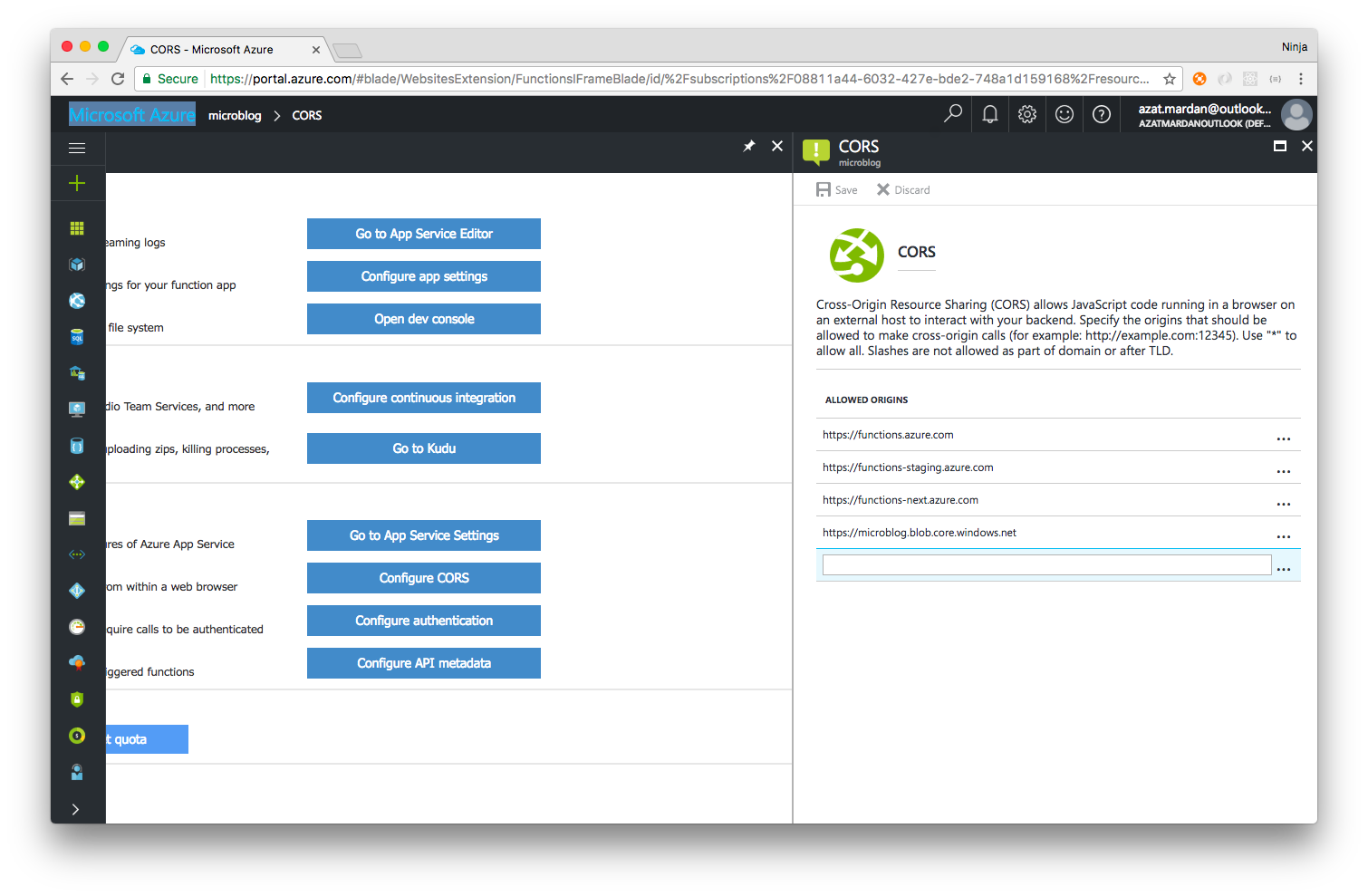
Set CORS in your Function App settings to allow requests from the blob storage URL. Navigate to the **Function app settings** as shown here:



## Step 6: Create and configure an Azure Blob Container

Create and Azure Blob Container and upload the static HTML and JS files for the UI.

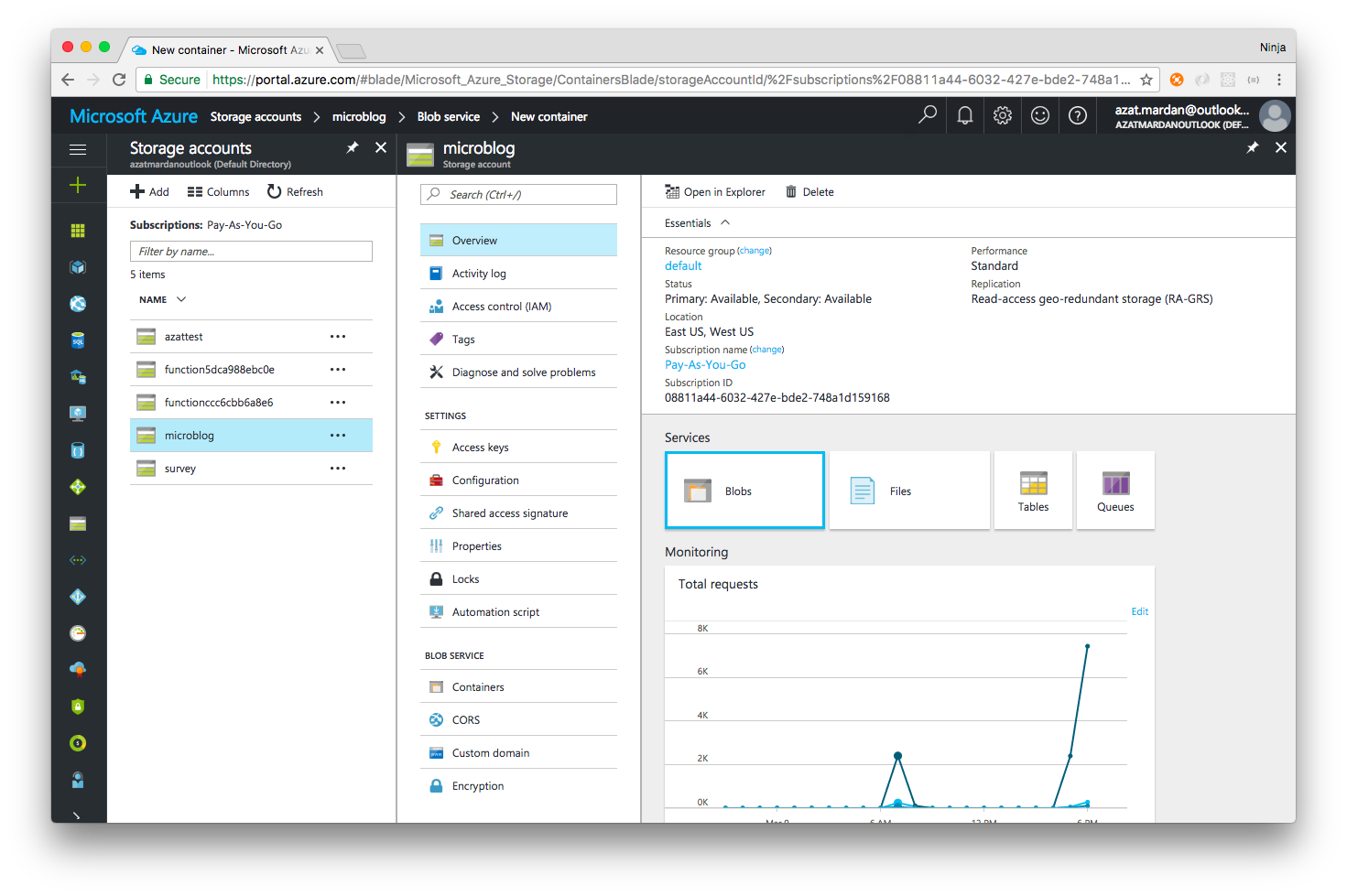
Then, find **Configure CORS** and select it. In a new pane, enter a new origin https://microblog.blob.core.windows.net where static files will be hosted via Azure Blob.



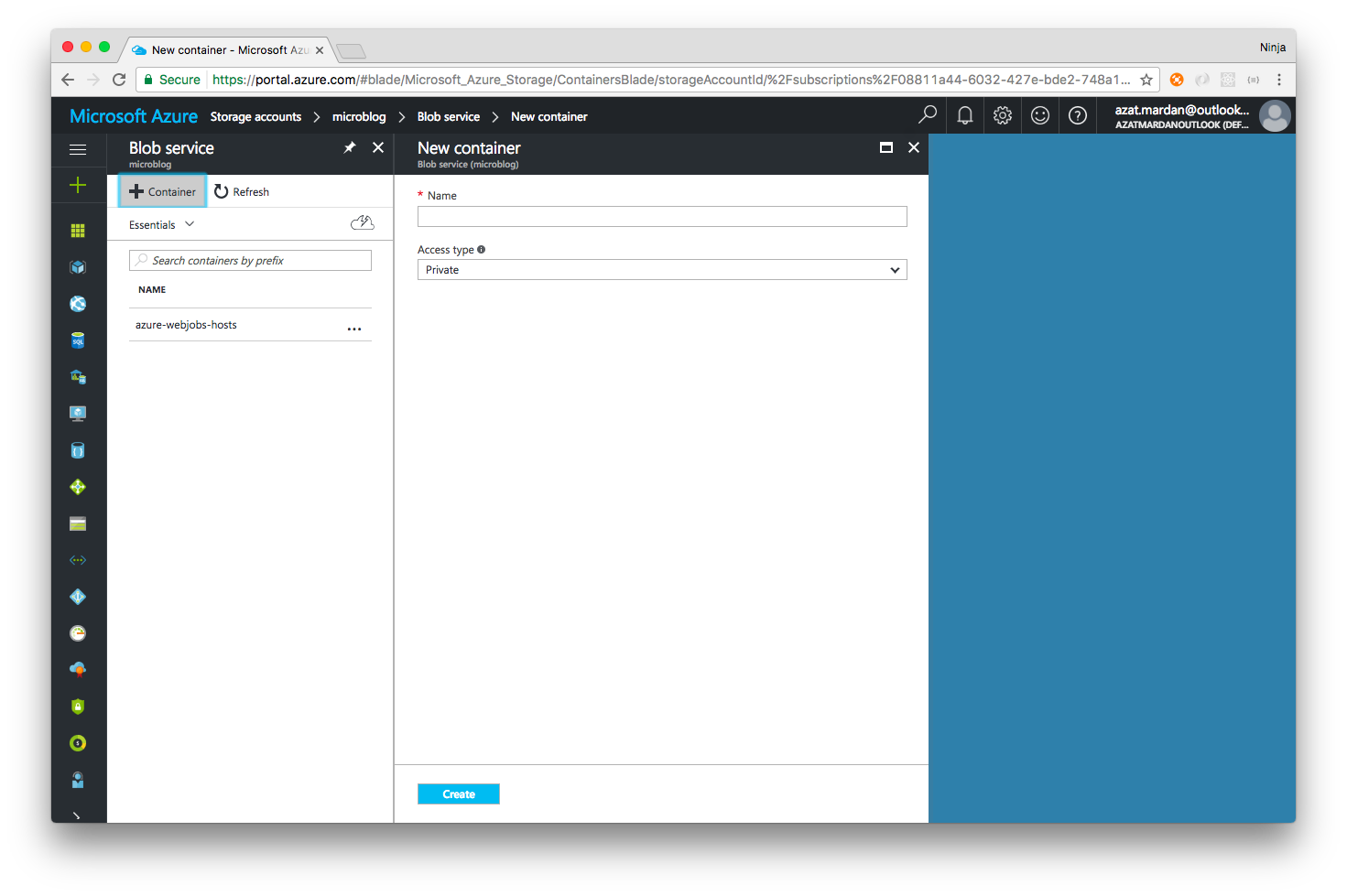
We are done with the function. You can test it with **Test** window. Make sure to provide proper JSON payload for POST.

Next, you will host the UI. To do so, create an Azure Blob container and upload the static HTML and JS files for the UI. You can re-use the same storage account. Select

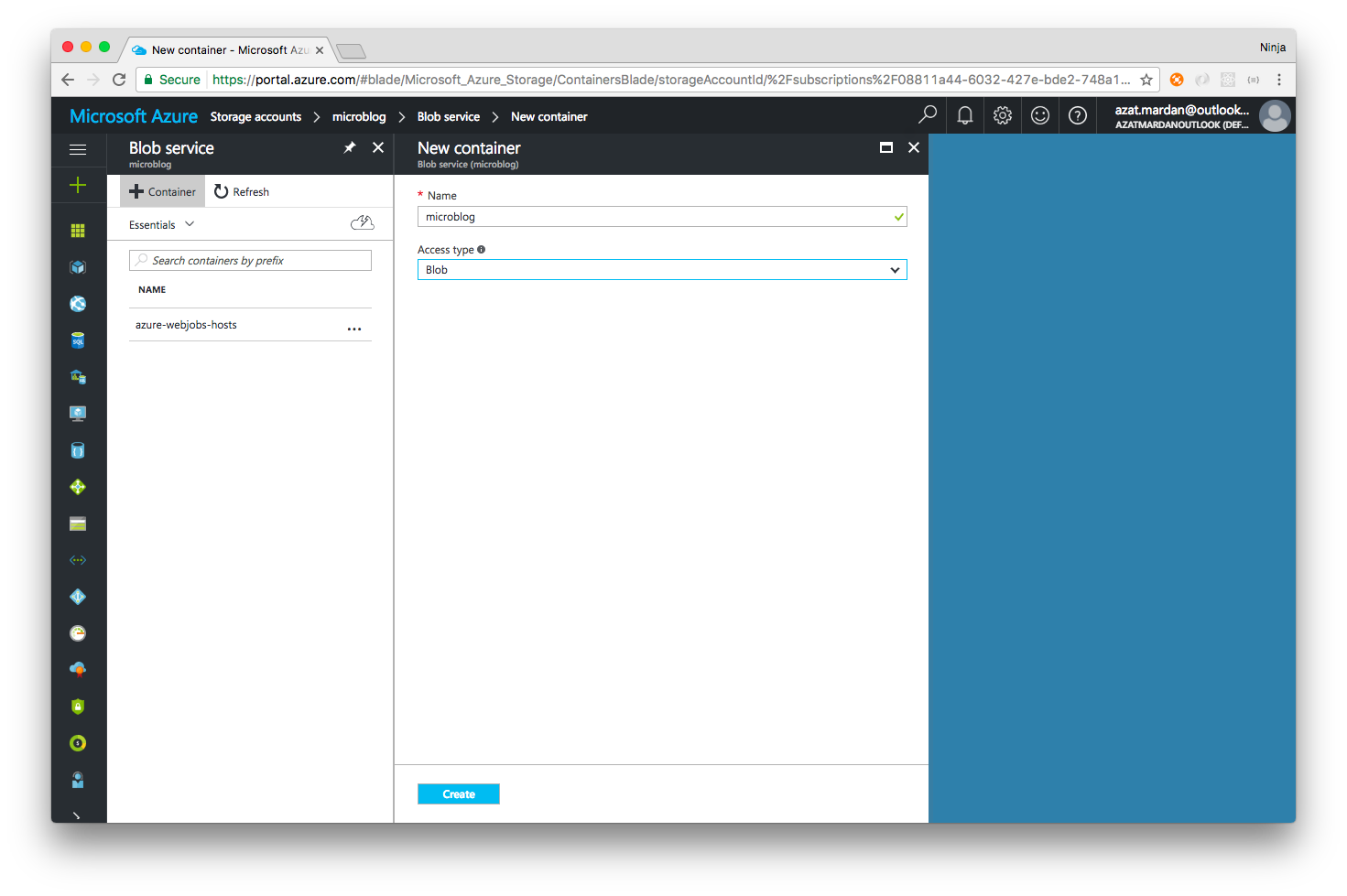
**Storage accounts** | **{storage account by name}** | **Overview** | **Blobs**:



The blob has containers which in turn have files. We need to create a container before we can upload and host files. Once inside of the blob, select **+ Container** to create a container:

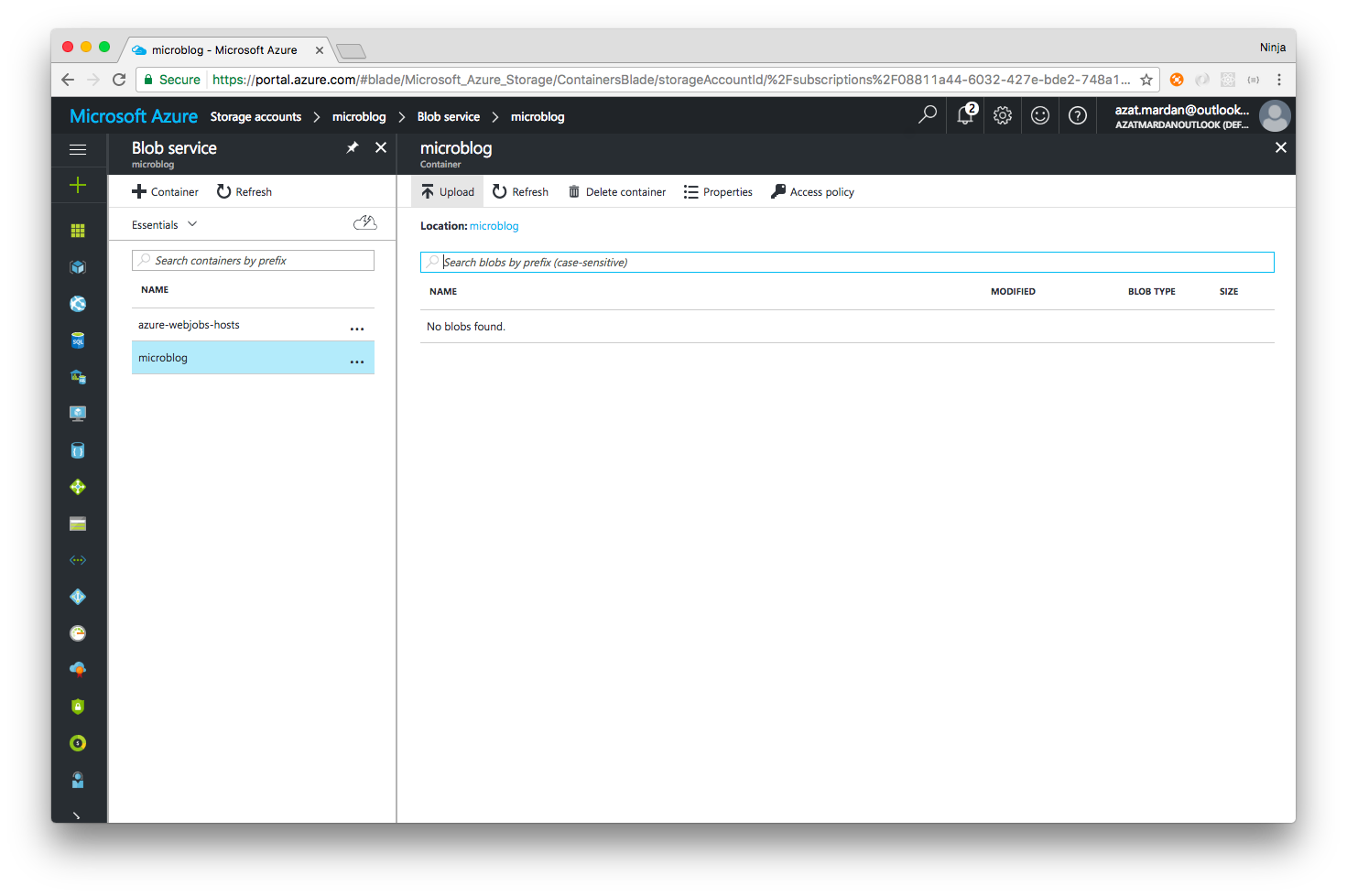


Then, enter a name of a new container (can have / to simulate subfolders, otherwise containers are flat and don't support folders):



Select **Create** to create your new container.

Select the newly created container name and then select **Upload** as shown below:



The UI files are modified from the previous lab to make the structure flat (containers don't support folders).

Open and modify the bundle.js file to point to your new API hosted by Azure Functions. Edit the variable baseUrl in bundle.js.

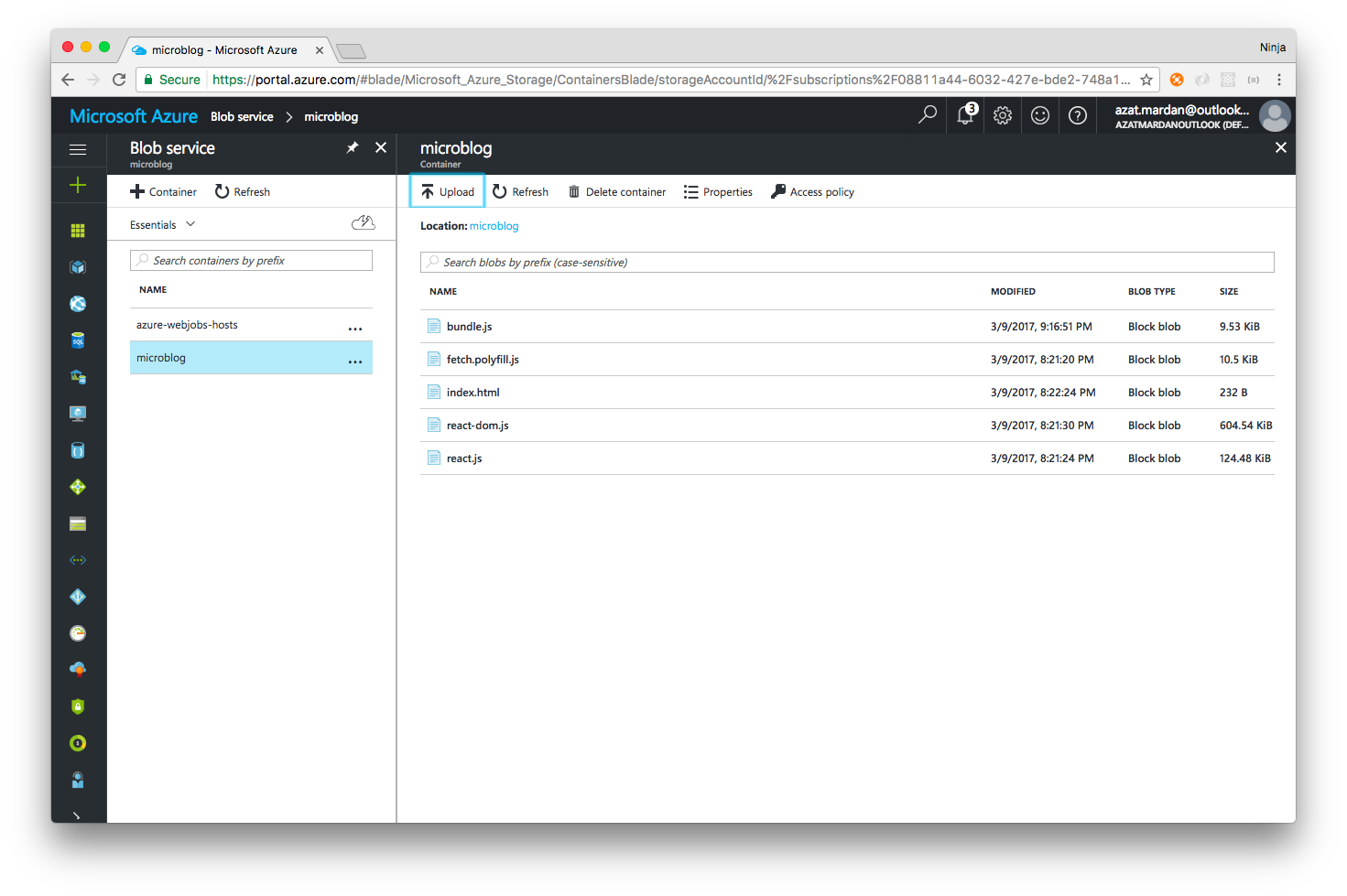
Instead of the line below, have your API base URL (change the name of the app from microblog).

...

var baseUrl = 'https://microblog.azurewebsites.net/api';

...

Next, save bundle.js and upload all five UI static files from the code folder of the lab. You will be able to observe them in the container once the upload is done:



You are now finished.

## Step 7: Test the microblog

Test Twitter-Clone (microblog) live in your browser by going to the URL of the function with HTML.

Test Twitter-Clone (microblog) live in your browser by visiting the URL of the blob index.html file. In this example:

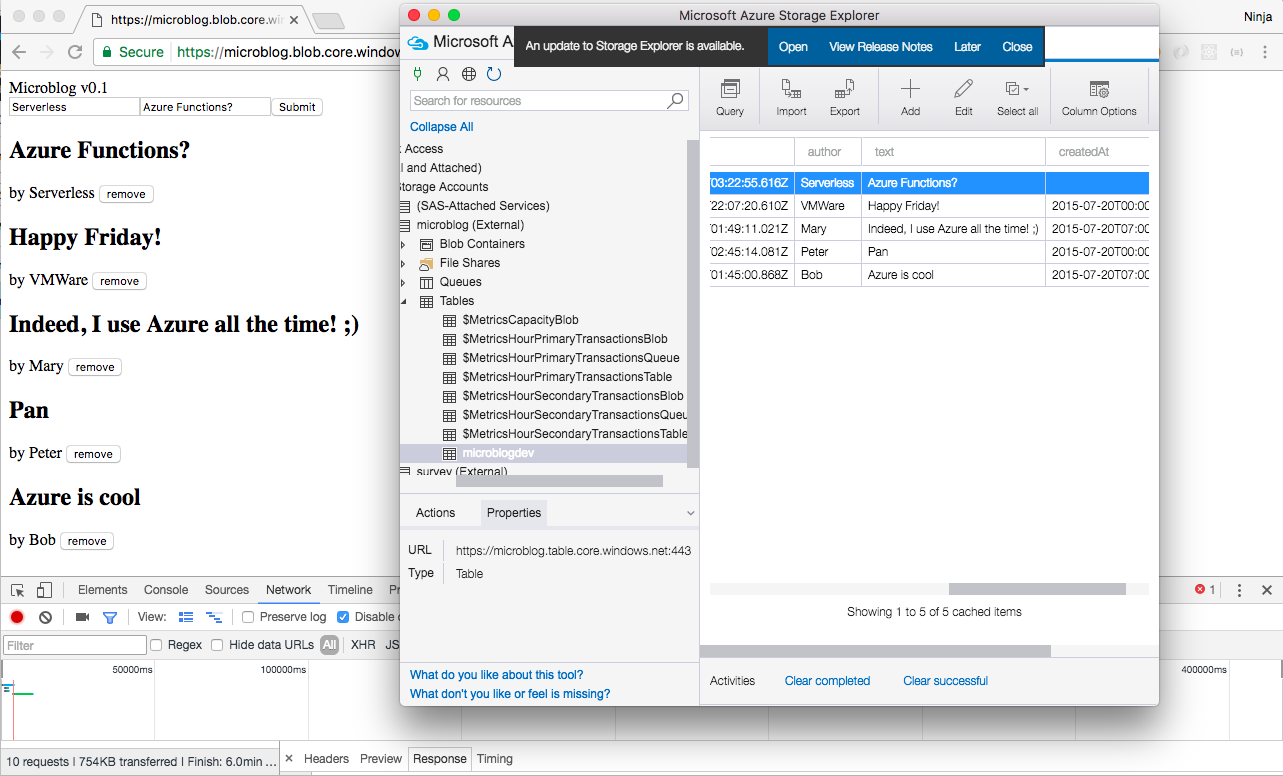
https://microblog.blob.core.windows.net/microblog/index.html

Based on structure, visit:

https://{storage}.blob.core.windows.net/{container}/{filename}

## Step 8: Optional: Test data in the Storage Explorer app.

Optional: Test data in [the Storage Explorer app](http://storageexplorer.com/). The results will be similar to this:



Troubleshooting:

* If you get 401 for the GET or POST, then make sure your HTTP trigger's authorization is set to **Anonymous**.
* If you get Access Allow Origin error, then make sure your Function App CORS is enabled for the blob URL.
* If you don't see response "Table Storage Created", then make sure your HTTP response is set to res and "Use function return value" is unchecked.