Web Development: Module 2, Lesson 8  
Deploying to Azure Hands-On Lab

## Overview

Building on [Module 2 Lesson 7](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons), it's time to make the application public. This application will be a huge success so we need a scalable solution which starts small and cheap but can rev up as the traffic increases. This is a perfect time to leverage the cloud.

## Objectives

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

* Learn how to use Azure CLI tool
* Prepare your project for deployment
* Deploy your project to cloud

## Prerequisites

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

* A text editor
* 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

## Exercises

This hands-on lab includes the following exercise:

* Exercise 1: Installing Azure CLI and Deploying a web app to Microsoft Azure via Git and CLI

## Exercise 1: Installing Azure CLI and Deploying a Web app (Recommended)

In this exercise, you will use npm to install the Azure CLI. You will use the Azure CLI to deploy the microblog app developed in [Module 2 Lesson 8](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) (be sure to refer to lesson 8 throughout this lab). Here is an overview of the steps we will walk through:

1. Open a shell and install Azure CLI
2. Log in to Azure CLI
3. Prepare your project by initializing package.json
4. Create a web.config file.
5. Create Azure Site (app) with git
6. Put Azure Storage environment variables into this app's cloud settings (AZURE\_STORAGE\_ACCOUNT and AZURE\_STORAGE\_ACCESS\_KEY)
7. Add code to the local repository and deploy by pushing code to Azure

Let’s dive deeper into using CLI to install apps. If you are familiar with Git, this approach will be the easiest for you since it won't require clicking around and going to Azure Portal.

Firstly, open a shell (a.k.a. terminal or command prompt) and enter the command to install Azure CLI version 0.10, if you don't have it already. We are using npm for it so you need to have npm (version 3 or higher):

$ npm i -g azure-cli@0.10.1

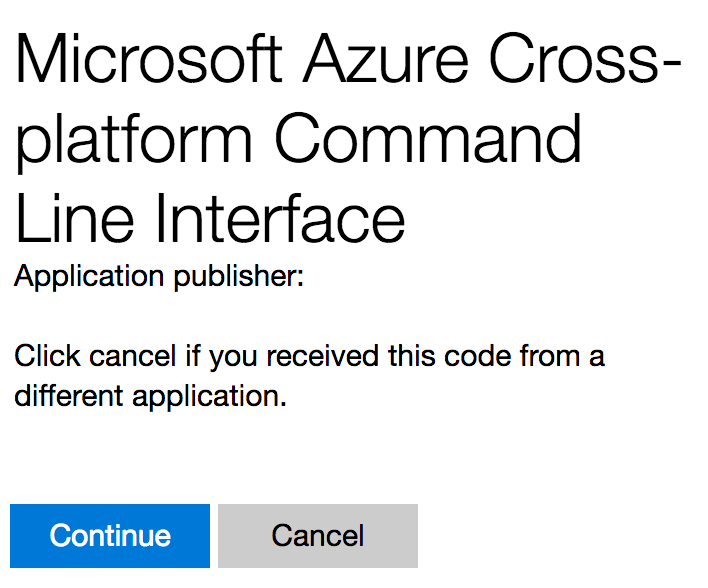
After you install Azure CLI, log in to Azure CLI using the command below. Keep your credentials nearby. You'll need to do it just once.

$ azure login

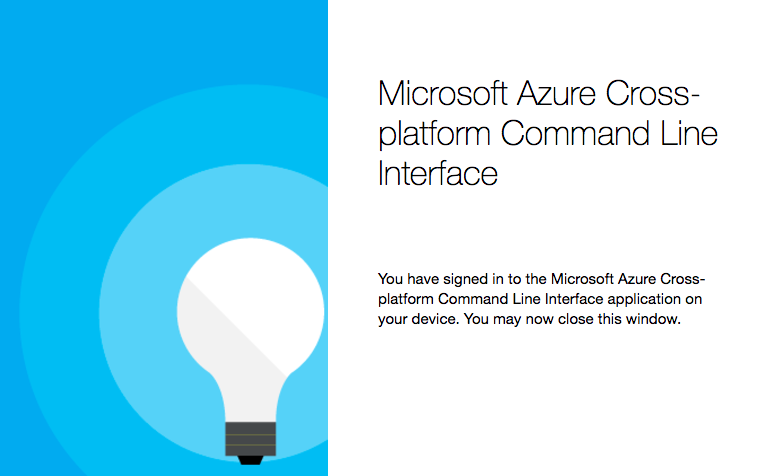
You should see this message:

info: Executing command login

- Authenticating...info: To sign in, use a web browser to open the page https://aka.ms/devicelogin. Enter the code ABC to authenticate.

Open https://aka.ms/devicelogin in a browser and enter the code (yours will be different from ABC). After entering your credentials you will be redirected to the following page

Select continue and return to your terminal. You will see this screen on your browser.



You should see a message on the terminal similar to the message below once you are connected.

\info: Added subscription Pay-As-You-Go

info: Setting subscription "Pay-As-You-Go" as default

+

info: login command OK

Now that you are logged in, we can prepare your project. Navigate to the folder with the source code you want to deploy. Create Azure Site (app) with the --git option (or add Git remote manually) where {appname} name of your app, e.g., microblog-prod-3:

$ azure site create {appname} --git

You should see the below information asking you to pick a server location. Pick a locale that is close to where you are located.

info: Executing command **site create**

+ Getting sites

+ Getting locations

help: Choose a location

1) South Central US

2) North Europe

3) West Europe

4) Southeast Asia

5) East Asia

6) West US

7) East US

8) Japan West

9) Japan East

10) East US 2

11) North Central US

12) Central US

13) Brazil South

14) Australia East

15) Australia Southeast

16) Canada Central

17) Canada East

18) West Central US

19) West US 2

20) UK West

21) UK South

:

Then pick an Azure Git and FTP deploy password (if you don't have it already). Azure wants you to use different name for Git/FTP than your main login. Pick password and then confirm it. These credentials are only for pushing the code (deploying), not for logging into the Azure portal. At this point the app is created in the cloud and configured for deployment via Git and FTP... but it does NOT have the code yet.

Run git remote -v to ensure the Azure CLI added the remote repository which is these values in my case:

azure https://microblog-git@microblog-prod-3.scm.azurewebsites.net/microblog-prod- 3.git (fetch)

azure https://microblog-git@microblog-prod-3.scm.azurewebsites.net/microblog-prod-3.git (push)

Update package.json to include the node and npm versions

"engines": {

"node": "6.9.0",

"npm": "3.10.8"

},

Add a web.config file to the project containing the following code. This web.config file tells Azure how to route traffic and what Node file to launch (app.js). The configurations in the comments are just for your reference and are NOT needed for this exercise.

<!--

This configuration file is required if iisnode is used to run node processes behind

IIS or IIS Express. For more information, visit:

https://github.com/tjanczuk/iisnode/blob/master/src/samples/configuration/web.config

-->

<configuration>

<system.webServer>

<handlers>

<!-- indicates that the app.js file is a node.js application to be handled by the iisnode module -->

<add name="iisnode" path="app.js" verb="\*" modules="iisnode" />

</handlers>

<!-- Make sure error responses are left untouched -->

<httpErrors existingResponse="PassThrough" />

<rewrite>

<rules>

<!-- Don't interfere with requests for logs -->

<rule name="LogFile" patternSyntax="ECMAScript" stopProcessing="true">

<match url="^[a-zA-Z0-9\_\-]+\.js\.logs\/\d+\.txt$" />

</rule>

<!-- Don't interfere with requests for node-inspector debugging -->

<rule name="NodeInspector" patternSyntax="ECMAScript" stopProcessing="true">

<match url="^server.js\/debug[\/]?" />

</rule>

<!-- First we consider whether the incoming URL matches a physical file in the /public folder -->

<rule name="StaticContent">

<action type="Rewrite" url="public{REQUEST\_URI}" />

</rule>

<!-- All other URLs are mapped to the Node.js application entry point -->

<rule name="DynamicContent">

<conditions>

<add input="{REQUEST\_FILENAME}" matchType="IsFile" negate="True" />

</conditions>

<action type="Rewrite" url="app.js" />

</rule>

</rules>

</rewrite>

<!-- You can control how Node is hosted within IIS using the following options -->

<!--<iisnode

node\_env="%node\_env%"

nodeProcessCommandLine="&quot;%programfiles%\nodejs\node.exe&quot;"

nodeProcessCountPerApplication="1"

maxConcurrentRequestsPerProcess="1024"

maxNamedPipeConnectionRetry="3"

namedPipeConnectionRetryDelay="2000"

maxNamedPipeConnectionPoolSize="512"

maxNamedPipePooledConnectionAge="30000"

asyncCompletionThreadCount="0"

initialRequestBufferSize="4096"

maxRequestBufferSize="65536"

watchedFiles="\*.js"

uncFileChangesPollingInterval="5000"

gracefulShutdownTimeout="60000"

loggingEnabled="true"

logDirectoryNameSuffix="logs"

debuggingEnabled="true"

debuggerPortRange="5058-6058"

debuggerPathSegment="debug"

maxLogFileSizeInKB="128"

appendToExistingLog="false"

logFileFlushInterval="5000"

devErrorsEnabled="true"

flushResponse="false"

enableXFF="false"

promoteServerVars=""

/>-->

</system.webServer>

</configuration>

In the app.js file, change the way we bootup the server to accommodate Azure's iisnode which imports app.js:

if (process.env.NODE\_ENV == 'production' || require.main === module) {

app.listen(3000, function(){

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

})

} else {

module.exports = app

}

Add Azure Storage and some other environmental variables into this app's cloud settings. We don't want to deploy secrets in start.sh file.

AZURE\_STORAGE\_ACCOUNT, AZURE\_STORAGE\_ACCESS\_KEY as well as NODE\_ENV are set in the cloud from CLI similarly. To start, get a list of current variables by running the following command:

$ azure site appsetting list

Then, set the variables one by one using this command (change the key and value):

$ azure site appsetting add NODE\_ENV=production

$ azure site appsetting add AZURE\_STORAGE\_ACCOUNT=*yourstorageaccount*

$ azure site appsetting add AZURE\_STORAGE\_ACCESS\_KEY=*yourstoragekey*

Now we are ready to deploy. Add code to the local repository with git add . and git commit -m 'initial commit'... and deploy by pushing code to Azure:

$ git add .

$ git commit –m ‘initial commit’

$ git push azure master

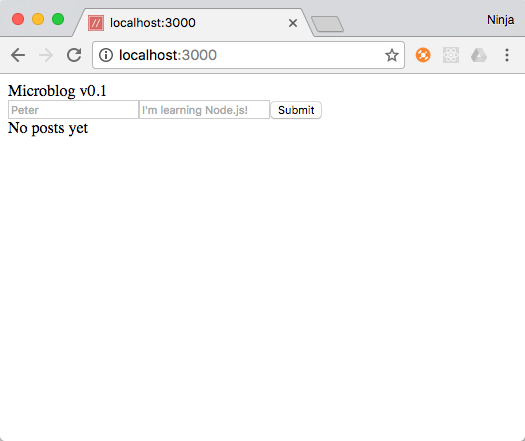
When you make a change to the code and want to re-deploy, you only need to execute the following commands:

git add .

git commit -m 'changes'

git push azure master

Open your app at http://APP\_NAME.azurewebsites.net where APP\_NAME is the name of your app. You should see the microblog home page:



## Summary

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

* Learn how to use Azure CLI tool
* Prepare your project for deployment
* Deploy your project to Azure