Discovery Days Hack – Proctor’s Guide

Deploying and Operating a Microservices Application on Azure

The intent of this Hack is to give attendees the ability to quickly deploy a running Microservices Application in Azure. There are a few caveats that should be emphasized up front:

* Everything being deployed is a container: As a core tenet, microservices are smaller, independent services that focus more on team independence (eg: to allow a polygot approach) that de-emphasize platform. Pick the right platform (.Net/Java/Node JS/etc..) that fits both the problem at hand and skills of the team.
* This hack models a dev/test model, with the pick of technologies and configuration management approach. In a Production mode, a given solution would have a container orchestrator (Kubernetes, Docker Compose, Mesos, etc…) and the management of secrets for the application would be using something like Azure Key Vault.
* Goal of this hack is to not only emphasize the ease of standing up a solution in Azure but also some of the underlying complexities more components in an architecture bring. Remember Microservices isn’t the solution to everything.

Emcee or Proctor Pre-setup

Someone who’s kicking off the hack needs to have access to a running version of the solution to demonstrate the end goal.

1. Clone the https://github.com/Microsoft/WhatTheHack.git repo. In the **009-MicroservicesInAzure\Host** directory are 2 files “deployHack.sh” and “deployHack.ps1” you will need.
2. Login to Azure Portal
3. Start the Cloud Shell
4. Upload the appropriate file through Cloud Shell based on which shell you’re using:
   * 1. BASH: deployHack.sh
        1. **NOTE**: Once uploaded you must make this file executable by running: “chmod 755 ./deployHack.sh”
     2. Powershell: deployHack.ps1
     3. For info on uploading, look at <https://docs.microsoft.com/en-us/azure/cloud-shell/persisting-shell-storage> and look at the section on transferring local files to the Cloud Shell
5. Both scripts take 3 parameters
   * 1. BASH: deployHack.sh <AzureDataCenter> <SubscriptionId> <BaseName>
        1. **NOTE**: BaseName must be globally unique, alpha only and less than 8 characters.
     2. Powershell: deployHack.ps1 -loc <AzureDataCenter> -sub <SubscriptionId> -baseName <BaseName>
6. After successfully deploying it, open up the portal, and go to the resource group you just deployed, and get the URL of the web site.
7. Copy the file Load.WebTest to a temp directory locally, edit it and change the ContextParameter Name="WebServer1" XML Element’s Value attribute to the HTTP (not HTTPS) URL of the web site (look at the bottom of the file).
8. Login to <https://dev.azure.com/> and create a new project.
9. Select Test Plans / Load Tests and add a new Load Test. Choose “Visual Studio test” and select your copy of the Load.WebTest to upload it to this new Load Test.
10. Name the Load Test and click Save.
11. Click the “Settings” tab beside “Web Scenarios” to configure the test.
12. Specify a decent amount of load, eg: 25 max vUsers running for 10 minutes to drive a good amount of traffic.
13. Click Save and then click Run test.

Demonstrating the End State:

After the load test initializes and starts running, you should have data streaming to the App insights. Particular areas of App Insights to highlight:

1. Application Map
2. Live Metrics Stream (while the load test is running)
3. Performance
4. Browsers

# Challenge Set 0: Pre-requisites - Ready, Set, GO!

## Lecture:

* Join the MS Team for this hack (if applicable).
* Brief Intro the Azure Cloud Shell and the Azure CLI

## Challenges:

* Make sure that you have joined the Teams group for this hack. The first person on your team at your table should create a new channel in this team with your team name.
* Login to the Azure Portal
* In a separate window, start the Azure Cloud Shell and list your subscriptions with the Azure CLI
  + Pick either Powershell or Bash based on your preference!
  + We will be running the Azure CLI, if you want to run it on your local machine just make sure you have the latest version.
* We are going to need the Subscription ID (guid format) of the subscription we will be using.
* **Tip**: We will be using the id many times in this hack. You may want to set a variable to hold it and use it in your future shell commands.
* **Tip:** There are 2 different ways to open the Azure Cloud Shell!

## Proctor Notes & Guidelines

* Azure Cloud Shell: <https://docs.microsoft.com/en-us/azure/cloud-shell/overview>
* Install the Azure CLI: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>
* Azure CLI Reference: <https://docs.microsoft.com/en-us/cli/azure/reference-index?view=azure-cli-latest>
* Commands to run (examples need to file in the XXX):
  + az account list
  + For Powershell: $sub=”XXX”
  + For Bash: export sub=”XXXX”
* **REMEMBER**: If the attendee closes the Cloud Shell window, they have to redefine the variables!

# Challenge Set 1: First Thing is First: A Resource Group

## Lecture:

* What is a Resource Group

## Challenges:

* Figure out which Azure data center is closest to you. You will be using it for this hack.
* Put the “scripting name” for the Azure Data Center that the Azure CLI uses in a variable called **loc**
  + **Hint**: South Central US scripting name is ***southcentralus***
* In your shell, create a Resource Group in that data center
* **Tip:** Create a 6 to 8 letter label that is unique to you that can be used to compose names for Azure resources, since several items we are creating today will have public DNS names that must be globally unique.
* **Tip**: Again, you might want to put the resource group name in a shell variable.

## Proctor Notes & Guidelines

* Azure Naming Conventions: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/naming-conventions>
* Commands to run (substitute XXX for the unique prefix):
  + For Powershell:
    - $loc = "centralus"
    - $rg = "rg-XXX"
  + For Bash:
    - export loc="centralus"
    - export rg="rg-XXX"
  + az group create --name $rg --location $loc --subscription $sub

# Challenge Set 2: Always need App Insights

## Lecture:

* What is App Insights
* What is an ARM template

## Challenges:

* In your shell, deploy a public ARM Template to deploy an App Insights resource to have the various applications log to
  + **ARM Template URL:** [**aka.ms/wth-microservices-template**](https://aka.ms/wth-microservices-template)
  + **Two Parameters:**
    - **name:** Name of the App Insights Resource to Create
    - **regionId:** Scripting Name of the region to Provision in
* Put the InstrumentationKey of the App Insights resource you just created in a variable called ***appInsightsKey***

## Proctor Notes & Guidelines

* App Insights: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview>
* ARM Templates: <https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates>
* Commands to run (fill in XXX and YYY with actual names)
  + az group deployment create --name XXX --template-uri https://aka.ms/wth-microservices-template --parameters name=YYYY regionId=southcentralus --resource-group $rg --subscription $sub
  + **For Powershell:** 
    - $appInsightsKey=az resource show --resource-group $rg --subscription $sub --resource-type Microsoft.Insights/components --name YYYY --query "properties.InstrumentationKey"
  + **For Bash:**
    - export appInsightsKey=$(az resource show --resource-group $rg --subscription $sub --resource-type Microsoft.Insights/components --name YYYY --query "properties.InstrumentationKey" | tr -d '"')

# Challenge Set 3: Get some Data

## Lecture:

* What is a CosmosDB

## Challenges:

* In your shell, create a CosmosDB Account
  + **NOTE**: When creating large resources, be patient for it to end.
* Copy the PrimaryMasterKey of the CosmosDB Account you just created as we will be needing it later
  + **Tip**: Once again, think about using a shell variable for this.

## Proctor Notes & Guidelines

* **NOTE**: You will not see a progress status when you are deploying via PowerShell and will take several minutes for the CosmosDB Account to be created.
* Cosmos DB: <https://docs.microsoft.com/en-us/azure/cosmos-db/>
* Commands to run (fill in XXX and YYY with actual names):
  + az cosmosdb create --subscription $sub --resource-group $rg --name XXX
  + **For Powershell:** 
    - $cosmosDbAccountName = "XXX"
    - $cosmosPrimaryKey=az cosmosdb list-keys --resource-group $rg --subscription $sub --name $cosmosDbAccountName --query primaryMasterKey
  + **For Bash:**
    - export cosmosDbAccountName="XXX"
    - export cosmosPrimaryKey=$(az cosmosdb list-keys --resource-group $rg --subscription $sub --name $cosmosDbAccountName --query primaryMasterKey | tr -d '"')

# Challenge Set 4: Deploy some containers to ACI

## Lecture:

* What is a Container
* What is ACI

## Challenges:

* There are three different containers you are going to deploy as Azure Container Instances (ie: az container)
  + Data API**: microservicesdiscovery/travel-data-service**
  + Itinerary API**: microservicesdiscovery/travel-itinerary-service**
  + DataLoader utility (to setup and seed Cosmos DB)**: microservicesdiscovery/travel-dataloader**
* All of these containers will need 3 environment variables defined:
  + **DataAccountName**: Name of the Cosmos DB Account
  + **DataAccountPassword**: Primary Key of the Cosmos DB Account
  + **ApplicationInsights\_\_InstrumentationKey**: Instrumentation Key of the App Insights Resource
* The Data API and Itinerary API containers, both need to specify a DNS Name, so they are easily addressable by the Web Site.
* After you get the Data API deployed, use the Azure CLI to query the deployed container for its full qualified domain name (FQDN) and store it in a variable called: **dataServiceUri**
* After you get the Itinerary API deployed, use the Azure CLI to query the deployed container for the FQDN and store it in a variable called: **itineraryServiceUri**
* Verify the Data Service by browsing the URL: **http://$dataServiceUri/api/airport** it will return a list of Airports
* Verify the Itinerary Service by browsing the URL: **http://$itineraryServiceUri/api/itinerary/AAA** it will return a 204.
  + **NOTE:** It should return a 404, but to get a positive test response, it returns a 204 (which represents “no content”) to verify the service is up and running.
* After you get the DataLoader deployed, you can see what it actually did by viewing its logs.
  + Use the Azure CLI to print out the logs for the Data Loader container.
  + Try this with the other containers as well just for fun.

## Proctor Notes & Guidelines

* Azure Container Instances: <https://docs.microsoft.com/en-us/azure/container-instances/>
* **NOTE**: These containers are stored in a Docker Hub repository named: **microservicesdiscovery** that holds container images specific to this hack.
* Commands to run (fill in XXX1, XXX2 and XX3 with the actual names):  
    
  **Data API:**  
  + az container create --subscription $sub --resource-group $rg --name XXX1 --image microservicesdiscovery/travel-data-service --dns-name-label XXX1 --environment-variables DataAccountName=$cosmosDbAccountName DataAccountPassword=$cosmosPrimaryKey ApplicationInsights\_\_InstrumentationKey=$appInsightsKey
  + **For Powershell:** 
    - $dataServiceUri=az container show -g $rg -n XXX1 --query "ipAddress.fqdn"
  + **For Bash:**
    - export dataServiceUri=$(az container show -g $rg -n XXX1 --query "ipAddress.fqdn" | tr -d '"')

**Itinerary API:**

* + az container create --subscription $sub --resource-group $rg --name XXX2 --image microservicesdiscovery/travel-itinerary-service --dns-name-label XXX2 --environment-variables DataAccountName=$cosmosDbAccountName DataAccountPassword=$cosmosPrimaryKey ApplicationInsights\_\_InstrumentationKey=$appInsightsKey
  + **For Powershell:** 
    - $itineraryServiceUri=az container show -g $rg -n XXX2 --query "ipAddress.fqdn"
  + **For Bash:**
    - export itineraryServiceUri=$(az container show -g $rg -n XXX2 --query "ipAddress.fqdn" | tr -d '"')

**DataLoader**:

* + az container create --subscription $sub --resource-group $rg --name XXX3 --image microservicesdiscovery/travel-dataloader --environment-variables DataAccountName=$cosmosDbAccountName DataAccountPassword=$cosmosPrimaryKey ApplicationInsights\_\_InstrumentationKey=$appInsightsKey --restart-policy OnFailure
  + az container logs --resource-group $rg --name XXX3

# Challenge Set 5: Deploy the Web Site

## Lecture:

* What is an App Service

## Challenges:

* In your shell, create a Standard Linux App Service Plan
* In your shell, create a Web App and set the **microservicesdiscovery/travel-web** as the container image for the Web App
* The following Application Settings need to be added:
  + **DataAccountName**: Name of the Cosmos DB Account
  + **DataAccountPassword**: Primary Key of the Cosmos DB Account
  + **ApplicationInsights\_\_InstrumentationKey**: Instrumentation Key of the App Insights Resource
  + **DataServiceUrl:** The URL to the Data Service, only over HTTP
  + **ItineraryServiceUrl:** The URL to the Itinerary Service, only over HTTP
* Verify that you can browse to the URL of the App Service

## Proctor Notes & Guidelines

* **NOTE**: You will not see a progress status when you are deploying via PowerShell or CLI.
* App Service Plans: <https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans>
* App Services: <https://docs.microsoft.com/en-us/azure/app-service/>
* Commands to run (fill in XXX and YYY with the actual names):
  + az appservice plan create --name XXX --resource-group $rg --is-linux --location $loc --sku S1 --number-of-workers 1 --subscription $sub
  + az webapp create --subscription $sub --resource-group $rg --name YYY --plan XXX -i microservicesdiscovery/travel-web
  + az webapp config appsettings set --resource-group $rg --subscription $sub --name YYY --settings DataAccountName=$cosmosDbAccountName DataAccountPassword=$cosmosPrimaryKey ApplicationInsights\_\_InstrumentationKey=$appInsightsKey DataServiceUrl="http://$dataServiceUri/" ItineraryServiceUrl=http://$itineraryServiceUri/
* Students might try to use raw FQDNs to pass in for the “\*ServiceUrl” application settings. These instead need to be fully addressable URLs with the http schema. Let them struggle through figuring that out.