Corndel DevOps Engineering Programme

in association with Softwire

Module 11: Project Exercise

Corndel Digital.

DB3M

Module 11

Project Exercise Brief

In this exercise we are going to migrate our application into a single cloud platform, Microsoft Azure. This will set us up for the final modules of the course and be more representative of 'real world' applications.

Azure has an offering for almost everything - hosting, Databases, DevOps pipelines with CI/CD, and even private docker-container registries that can be securely linked into specific projects.

However, we are going to focus on moving our Flask App and database to Azure. We'll be hosting our production application, in a docker container, as an Azure App Service, and we'll be using Azure's CosmosDB, which has an API compatible with MongoDB.

We're going to keep Travis for CI/CD, and Dockerhub as our container registry. While we could move these to Azure it is simpler and cheaper not to do so.

Using Azure means we have to do some additional work to set-up the infrastructure we need. There are multiple ways of doing this:

- through the Azure Portal
- through the Azure CLI
- by using Visual Studio's GUI.

We're going to be using a mix of the first two, which is the most common method for setting things up. The Azure Portal gives a nice overview of all the available options, however it is a very manual process. While the CLI is easier to automate, it can be significantly more complicated than the Portal. Using both should help give you the best of both worlds. In this exercise we give instructions for both Portal and CLI for each step, you should vary which you use to gain familiarity with both approaches and their drawbacks.

Part 1: Getting Ready

Step 1: Set up Azure Account

You should already have an Azure Account set up for you for this course. You should have received an email for this, and it will have been the account you used in the workshop for this module.

If you do not have such an account, please contact a trainer.

Alternatively, and especially if you want to continue working on your project after the course, you can follow the instructions below to set up your own account. You may already have a Microsoft Account created for you by your work, but we suggest not using such an account - using your own account will ensure you a) have all the permissions you need by default; and b) don't inadvertently set-up resources in a place they're not supposed to be.

- Go to portal.azure.com and create an account.
- When prompted selected the "Start with an Azure free trial"
    - This gives 12 months of access to some resources, and $150 credit for 30 days, along with "always free" tiers for most common resources.
    - We'll be sticking with the Free Tier for these exercises, ensure you select the "FREE Tier" or "SKU F1" when creating resources, as the default is usually the cheapest paid option.
    - The Free Tier and Free Subscription have some downsides: the size, scalability and some functionality of resources is limited, and you can only have one or two of each resource type taking advantage of the Free Subscription.
    - If you see an error like "`The subscription you have selected already has an app with free tier enabled`" then you should either delete the existing resource in the Portal, or opt for the cheap-but-paid Basic Tier for your new resource.

Step 2: Install the cli

- Follow the instructions at: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli to install the CLI on your machine if you haven't already.
- Open up new terminal window and enter `az login`, which will launch a browser window to allow you to log in.

Part 2: Setting Up Your Cloud

Step 1: Add Resource Group

> *A resource group is a logical container into which Azure resources, such as web apps, databases, and storage accounts, are deployed and managed. More Azure Terminology here.*

1. If using the Azure Portal, you can navigate to "Resource Groups" and click "Add".
   - Alternatively, just skip to Step 2 below and select "Create new" in the Resource Group option when creating your database.
2. Or via the `az` CLI, you can enter:

```
az group create -l uksouth -n <resource_group_name>
```

Step 2: Set up a Cosmos database with Mongo API

- Azure offers a database resource called Cosmos DB which can be interacted with via a MongoDB API. This allows us to just provide an alternate `MONGODB_CONNECTION_STRING` environment variable to our app, which `pymongo` will be able to use with no python code changes required.
- With the Portal:
  - New -> CosmosDb Database
  - Select "Azure Cosmos DB for MongoDB API" in the API field

    - You can also configure secure firewall connections here, but for now you should permit access from "All Networks" to enable easier testing of the integration with the app.
- With the CLI:
  - Create new CosmosDB Account

```
az cosmosdb create --name <cosmos_account_name> --resource-
group <resource_group_name> --kind MongoDB
```

  - Create new MongoDB database under that account

```
az cosmosdb mongodb database create --account-name
<cosmos_account_name> --name <database_name> --resource-
group <resource_group_name>
```

Step 3: Connect your app to the CosmosDB

- Get a connection string to pass in as an environment variable
- via CLI:

```
az cosmosdb keys list -n <cosmos_account_name> -g
<resource_group_name> --type connection-strings
```

- via Portal:
  - Open the resource, then go to the "Connection String" page in the sidebar.
- Alter the given connection string to tell `pymongo` to use the default database:
  - Add `/DefaultDatabase` after the port number in the given connection string, eg: `mongodb://<database_name>:<primary-master-key>@<database_name>.mongo.cosmos.azure.com:10255/`DefaultDatabase`?ssl=true&<config>`
- Now build & run the production app locally using this connection string, and try using it to check all functionality works.

Part 3: Host the frontend on Azure Web App

Side note: "Azure App Service with Containers" vs "Azure Container Instances"

- Azure App Services are set up for hosting long-running web applications, with nice setup for front-facing web applications, with SSL and domain names built in, and allows either raw-code or a Container Image to be uploaded.
- Azure Container Instances are for lightweight short-run "compute" containers, faster to deploy and billed per-second, with better orchestration for multiple containers, but without the nice domain and SSL addons for web.
- Also App Services are part of the "Always Free" offering from Azure, another reason for us to use them here.

Step 1: Put Container Image on DockerHub registry

- Azure Container Registry costs money, so we will be using DockerHub as the registry to store our production container images, if you weren't already.
- Update your Travis `before_deploy` script, ensuring that the production build image is pushed to DockerHub.

Step 2: Create a Web App

Note: your `<webapp_name>` needs to be *globally* unique, and will form part of the URL of your deployed app: `https://<webapp_name>.azurewebsites.net`.

- Portal:
  - Create a Resource -> Web App
  - Fill in the fields, be sure to select "Docker Container" in the "Publish" field, and choose your newly-created Resource Group.

    - On the next screen, select DockerHub in the "Image Source" field, and enter the details of your image.
- CLI:
    - First create an App Service Plan:

```
az appservice plan create --resource-group
<resource_group_name> -n <appservice_plan_name> --sku Free
--is-linux
```

    - Then create the Web App:

```
az webapp create --resource-group <resource_group_name> --
plan <appservice_plan_name> --name <webapp_name> --
deployment-container-image-name <dockerhub_username>/todo-
app:latest
```

Step 3: Set up environment variables

- Portal:
    - Settings -> Configuration in the Portal
    - Add all the environment variables as "New application setting"
- CLI:
    - Enter them individually via

```
az webapp config appsettings set -g <resource_group_name> -n
<webapp_name> --settings FLASK_APP=todo_app/app.
```

    - Or you can pass in a JSON file containing all variables by using `--settings @foo.json`, see here.

Step 4: Confirm the live site works

- Browse to `http://<webapp_name>.azurewebsites.net/` and confirm no functionality has been lost.

Part 4: Set up Continuous Deployment

If you turn on Continuous Deployment for your Web App, Azure App Service sets up a webhook url. Post requests to this endpoint cause your app to restart and pull the latest version of the container image from the configured registry.

- Enable CD
  - Portal: Deployment Centre -> Settings tab
  - CLI: `az webapp deployment container config --enable-cd true --resource-group <resource_group_name> --name <webapp_name>`
- Test webhook
  - Take the webhook provided by the previous step, add in a backslash to escape the `$`, and run: `curl -dH -X POST "<webhook>"`
  - eg: `curl -dH -X POST "https://\$<deployment_username>:<deployment_password>@<webapp_name>.scm.azurewebsites.net/docker/hook"`
  - This should return a link to a log-stream relating to the re-pulling of the image and restarting the app.
- Add this `curl` command to your Travis `deploy` script, passing in the url as an environment variable, and removing any remaining references to Heroku.

Stretch Goals

Part 5: Add restrictions and security

Secure your database

- We don't want to allow any connections to the database other than via the web app.
- As mentioned in the Security module, you can either:
  1. Find the IP address for your web app, and restrict the Cosmos DB to only allowing access from that IP via by using a firewall.
  2. Set up an Azure Virtual Network to connect your webapp and only allow access to your CosmosDB through that VNet.

Add access restrictions to the web app

- IP-based restrictions are a common feature to restrict access to a bespoke app for only employees accessing it via a company's VPN.
- It is also possible to restrict usage from a particular location or service for legal or other reasons.
- See App Service access restrictions docs for more information.