# DevOps

# Introduction

I’ve put some thoughts down and built some pipelines. It is now time to get feedback and improve on what we want to go forward with. You can find all of the sample microservices and associated DevOps assets here:

<https://dev.azure.com/keith0264/Canterbury>

You can run the WebApp (which calls the two APIs) from the [external IP](https://dev.azure.com/keith0264/Canterbury/_environments/15/providers/kubernetes/10?type=service-item&uid=c9859544-6866-11ea-9674-bab03c0fd8f3) of the WebApp’s load balancer. All of these services were deployed to Azure Kubernetes Service by Azure DevOps.

DevOps is not just about tech. The references below are worth reading.

General references

[DevOps at Microsoft – Driving Culture Change](https://azure.microsoft.com/en-us/resources/devops-at-microsoft-driving-culture-change/)

[DevOps at Microsoft](https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/)

[Configuring your release pipelines for safe deployments](https://devblogs.microsoft.com/devops/configuring-your-release-pipelines-for-safe-deployments/)

# Visual Studio Project Structure

This structure comes from Microsoft Reference Architecture but with updates from Stuart Fraser. The Microsoft structure originally followed ‘clean architecture’ pretty closely but it was a bit cumbersome and they eventually simplified it to what you see here. It is not the ‘normal’ folder structure you’d get with Visual Studio but hey its Microsoft, right?

**Project Root: Canterbury** – **(**I’m following a convention of NZ provincial rugby teams)

* **build**: Scripts for deploying Docker images
  + **azure**-**devops**
* **deploy**: Script for deployment
  + **k8s**
    - **helm** (superseded by manifest in git?)
* **src**: All source projects, including tests
  + **ApiGateways**: Envoy configuration and Aggregators source code
  + **BuildBlocks**: Common components used by several projects.
  + **Mobile**: Mobile apps projects.
  + **Services**: Backend for all services. Including unit and functional tests for some projects.
    - ServiceA
      * Tests: Unit Tests (XUnit…)
      * ServiceA.Api (folders which are appropriate)
        + Controllers
        + Extensions
        + Grpc
        + Infrastructure
        + Integration Events
        + Model
        + Proto
        + ViewModel
    - ServiceB
    - ServiceC
    - ServiceD
    - ServiceE
* **Tests**: General functional application tests.
* **test-results:** Test results
* **Web:** Web applications

# Development Environment

I admit it. I am biased towards developing locally using ‘full fat’ Visual Studio. With the coming of containerised development, there are now more options but only a few of them force us all down any preferred line. Here are the options as I see them:

* Visual Studio 2019 with Docker Desktop
* VS Code with Docker Desktop
* Visual Studio 2019 with Azure Dev Spaces (Docker Desktop not required)
* VSCode with [Kubernetes Tools Extension](https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools) and Azure Dev Spaces (the extension also works with AWS and GCP…) (Docker Desktop not required)
* VS Code Remote Development

I don’t need to talk about Visual Studio or VS Code here as we are all familiar with these tools. However, Dev Spaces and VS Code Remoting is worth mentioning.

Azure Dev Spaces is an extension to AKS that allows you to easily run and debug your microservice code in the context of a larger application. You can test your code end-to-end, hit breakpoints on code running in the cluster, and share a development cluster between team members without interference. You still need Visual Studio or VS Code but not Docker Desktop. It provides a rapid, iterative Kubernetes development experience for teams in Azure Kubernetes Service (AKS) clusters. Azure Dev Spaces also allows you to debug and test all the components of your application in AKS with minimal development machine setup, without replicating or mocking up dependencies. The experience is like working locally – except you’re not. I personally really like it, you’re not really aware you’re running remotely, but since it’s an Azure service it incurs small extra costs and also importantly, Dev Spaces **does net yet** support microservices with gRPC.

VS Code Remoting is really exiting. VS Code Remote Development allows you to use a container, remote machine, or the Windows Subsystem for Linux (WSL) as a fully featured development environment. It effectively splits VS Code in half and runs the client part on your machine and the "VS Code Server" server part basically anywhere else. The Remote Development extension pack includes three extensions:

* Remote - SSH - Connect to any location by opening folders on a remote machine/VM using SSH.
* Remote - Containers - Work with a sandboxed toolchain or container-based application inside (or mounted into) a container.
* Remote - WSL - Get a Linux-powered development experience in the Windows Subsystem for Linux.

This last option, WSL2, could be a really attractive option with the release of WSL2 as part of the Windows 10 Spring 2020 Update.

Recommendation: Visual Studio / VS Code with Docker Desktop

References

[Working with Kubernetes in Visual Studio Code](https://code.visualstudio.com/docs/azure/kubernetes)

[Azure Dev Spaces Documentation](https://docs.microsoft.com/en-gb/azure/dev-spaces/)

[VS Code Remote Development](https://code.visualstudio.com/docs/remote/remote-overview)

[Visual Studio Code Remote Development May Change Everything](https://www.hanselman.com/blog/VisualStudioCodeRemoteDevelopmentMayChangeEverything.aspx)  (Scott Hanselman)

# Environments

I suggest the following environments:

## Development

This is a local environment on a developer’s machine

## Test (Integration Test)

This is the first environment when components come together and testing beyond unit test can take place – integration testing

## QA

The quality assurance environment is intended for us to use to complete quality assurance testing, such as functional or performance tests, and to test integrations. This QA environment will more closely resemble Production perhaps not the scale of production but big enough to do load testing on and get meaningful results. We should not have any sensitive data (e.g. PII) in the QA environment. I am assuming that we will have to generate sensible test data for use in this environment.

This is the supported, access-controlled production environment with live users and user data. If under our control it should be in its own Azure subscription as this acts as a security boundary and developers should not have day to day or uncontrolled access to this environment. Also, Azure Privileged Identity Management (PIM) should be used to grant time limited access. PIM also has an audit log where we can prove who did or did not have access. There should be a manual approval step when upgrading microservices in production. This will be automatically logged and archived by Azure DevOps.

Recommendation

Environments as above

What about Pilot environments? – these are time-limited, numbers restricted exercises but use real customers and ‘*real’* data.

There is a best practice to use a dedicated Azure Container Registry for Production and another one for Dev, Test etc.

As for k8s, we should go forward with a dedicated cluster for Production and another, separate cluster for Test, QA etc, logically separated by namespace and policy.

# Git Strategy

Essentially there are two approaches:

* Git flow
* Github flow (aka trunk-based development)

The general idea behind git-flow is to have several separate branches that always exist, each for a different purpose: e.g. master, develop, feature, release, and hotfix. The process of feature or bug development flows from one branch into another before it’s finally released.

Git Flow may be worth it when:

* When you have discrete named or numbered releases
* When you need to freeze development on a release candidate while still continuing to develop and integrate features for a subsequent release
* When multiple versions of the software need to be supported and maintained independently

With Git flow it is definitely harder to deal with in a continuous deployment model.

GitHub Flow is a simplified alternative to Git flow. GitHub has some of the same elements as Git Flow, such as feature branches. But unlike Git Flow, GitHub Flow combines the mainline and release branches into a “master” and treats hotfixes just like feature branches. This simplified model is better suited to continuous delivery models where changes can be quickly made and easily deployed, sometimes multiple times a day. Microsoft use a trunk-based development flow called Release Flow.

Recommendation: Github Flow (as shown in the DevOps demo)

References

[Trunk-Based Development](https://trunkbaseddevelopment.com/)

[Microsoft Engineering and Release Flow](https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/release-flow)

# Git Repo Approach

When you start moving to microservices, the first question you ask is: How do I organise the codebase? Do you create a repository for each service, or do you create a single ‘mono repo’ for all services? The two approaches are:

* “Monorepo” – each microservice has its own folder in a single repository
* “Multirepo” - each microservice has its own dedicated git repository

Obviously, there are pros and cons to each approach. If you need to restrict access to only certain repos then multirepo is the way to go - but we don't have this security requirement. Setting up DevOps CI and CD is with multirepo is much easier was you let each repository have its own process for being deployed. When using a monorepo, there needs to be additional logic for sorting through the folders that make up the different projects within the monorepo. This is really setting up a trigger on a folder rather than the repo file set as a whole. For multirepos: There is clear ownership and smaller codebases are easier to manage. If a codebase gets too big then git clones, pulls, and pushes can take too much time. The downside is that the code base does not match the architecture and you want every engineer to understand the bigger picture. Additionally, it could be more challenging for us to enforce standardisation of code across repositories. We really want one team with knowledge across all of the services. For me, the decision comes down to not wanting to git clone a whole bunch of different repositories at one time (20?). Apparently, Google uses a monorepo approach… A monorepo approach feels like the right solution for us.

\*not quite true. You can have sub-repos but this sounds truly awful.

Recommendation: Git monorepo (as shown in the DevOps demo)

References

[Journey to Microservices](http://blog.shippable.com/our-journey-to-microservices-and-a-mono-repository)

[Repo Style Wars – Mono v Multi](http://www.gigamonkeys.com/mono-vs-multi/)

[Monorepos in Git](https://www.atlassian.com/git/tutorials/monorepos) (Atlassian)

# Hosting on Kubernetes

We seemed to have decided almost by default to more to Kubernetes from Service fabric. This is not a good way to make an important decision. There is undoubtedly an enormous take up of Kubernetes in the tech industry. But is it right for us? We are aware that the service Fabric is no longer being ‘pushed’ by Microsoft, but what other drivers for migration do we have? If we have cross-cloud as a dominant requirement then we could easily push .Net code from Azure DevOps to either the Azure App Service or to the Elastic Beanstalk Service in AWS. This is very easy to do and both hosting platforms are mature, fully managed PaaS services that can scale from ~£50 per month to whatever scale is required. The decision is clouded by yet more options. The Azure App Service can also host containers (indeed multiple Linux containers). Deploying what we have as a monolith in a container is a viable option. You can also host API endpoints as Azure Function proxies either on a consumption plan or in an App Service Plan. All of these options give you scale up and down as reasonable prices and cross-cloud.

However, Kubernetes is our way forward then let’s get stuck into it. Kubernetes is an open-source orchestration software for deploying, managing and scaling containers. Other tools you may have used or heard of include Azure Service Fabric, Docker Swarm, Mesos and others. Applications are increasingly built using containers whether or not they follow Domain Driven Design (DDD), which are microservices packaged with their dependencies and configurations. Kubernetes is the Greek word for the helmsmen of a ship or a pilot. It is usually abbreviated as “*k8s*”.

While you can install and run ‘raw’ Kubernetes on Azure as (IaaS), the Azure Kubernetes Service (AKS) offers a much easier way to handle the complexity and operational overhead of Kubernetes by offloading much (but not all!) of the responsibility to Azure. Please note that imho, AKS is still not a fully managed PaaS service like the Azure App Service, you’re still conscious that you’re dealing with a cluster of VMs and associated load balancer and VNET. AKS handles critical tasks like monitoring and maintenance for you. You only have to manage and maintain the agent nodes. AKS orchestrates clusters of virtual machines and schedules containers to run on those virtual machines based on their available compute resources and the resource requirements of each container. Containers are grouped into pods, the basic operational unit for Kubernetes, and those pods scale to your desired state. Kubernetes also automatically manages service discovery, incorporates load balancing, tracks resource allocation and scales based on compute utilisation. And, it also checks the health of individual resources and enables apps to self-heal by automatically restarting or replicating containers.

Until very recently (Q4 2019) the most common way to deploy containers to AKS via Azure DevOps was through Helm Charts (remember Kubernetes was a helmsman…). Helm v1 and v2 and also had a required server component called Tiller. With k8s now supporting rbac, Tiller is no longer required and is not shipped with Helm 3.0. Still, Helm charts were a pain to work with as you had to understand Helm, author the Charts, version control them, install Helm on your DevOps server and issue Helm commands. In Q4 2019, Microsoft shipped a new task in Azure DevOps that greatly simplifies the deployment of containers to Kubernetes, the Kubernetes Manifest Task. This is what I have used in this demo as I think it saves a lot of time and seems to be a good bet for the future. It means we can get away with knowing less about k8s infrastructure – *but not zero*.

Recommendation: Formalise the Kubernetes Choice

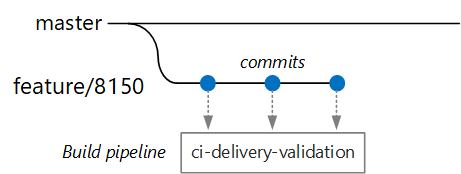
# DevOps with Microservices

For the demo I have followed the first reference below with a couple of differences. I have not implemented a further build on a merge to master although that’s easy to do. I have also used the new Kubernetes Manifest Tasks to deploy to k8s rather than Helm Tasks. The approval gate is in Next Steps. I have used our decisions above:

* The code repository is a monorepo, with folders organised by microservice (Service A, Service B and a WebApp)
* The branching strategy is based on trunk-based development.
* The release strategy uses release branches to manage releases. Separate releases are created for each microservice e.g. Service A, Service B etc.
* The CI/CD process uses Azure Pipelines to build, test (coming!), and deploy the microservices to AKS.
* The container images for each microservice are stored in an Azure Container Registry.
* The new Kubernetes Manifest Task is used in the pipeline to deploy containers from the ACR to AKS for the sample Service A microservice.

## Build Pipeline

A developer is working on a microservice called the Service A. While developing a new feature, the developer checks code into a feature branch. This can be created either in Azure DevOps Boards directly on a work item, or by the Git command. By convention and by necessity for the demo to work, feature branches are named ***feature/servicea\**** e.g. feature/servicea/8150



The build definition file includes a trigger that filters by the branch name and the source path. Using this approach, each microservice can have its own build pipeline. Only code that is checked into the **/Services/ServiceA/** folder triggers a build of Service A. Pushing commits to a branch that matches the filter triggers a CI build. At this point in the workflow, the CI build runs some minimal code verification:

1. Build the code.
2. Run unit tests. (<todo>)

The goal is to keep build times short, so the developer can get quick feedback. Once the feature is ready to merge into master, the developer opens a PR from this feature branch into master.

At this point we could trigger another CI build that performs some additional checks:

1. Build the code.
2. Run unit tests.
3. Build the runtime container image.

I have not put these in but since they are a superset of what has been done it would be easy to incorporate them.

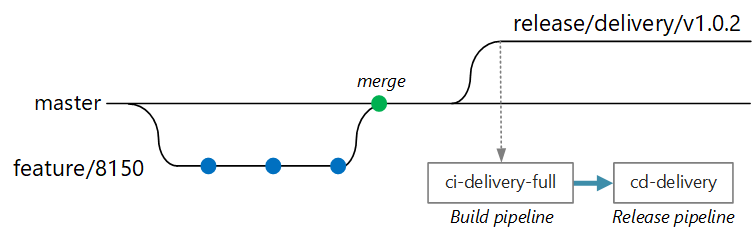
Required Assets:

A single simple build pipeline with one step:

1. Docker step ‘command build’ (see azure-pipelines.yml)

## Release Pipeline

At some point, the team is ready to deploy a new version of Service A. The release manager creates a branch from master with this naming pattern: release/<microservice name>/<semver>. For example, release/servicea/v1.14.0.



Creation of this branch triggers a full CI build that runs all of the previous steps plus:

1. Push the container image to Azure Container Registry. The image is tagged with the version number – right now it’s the build number but it should come from the branch name. <todo>
2. Approval stage (<todo>)

Assuming this build succeeds, it triggers a deployment (CD) process using an Azure Pipelines release pipeline. This pipeline has the following steps:

1. Run the k8s manifest tasks to deploy the service.

Once we get our environments sorted out, we can introduce approver steps and the retagging of the Docker image for the production namespace in Azure Container Registry. For example, if the current tag is canterbury.azurecr.io/servicea:v1.14.1, the production tag is canterbury.azurecr.io/prod/servicea:v1.14.1.

Even in a monorepo, these tasks can be scoped to individual microservices, so that we can deploy microservices with high velocity. The process has some manual steps: Approving PRs, creating release branches, and approving deployments into the production but I think these are good things and that you probably want all of them.

Required Assets:

A single build and deploy pipeline:

1. Docker step ‘command buildandpush’
2. Kubernetes manifests (createsecret and deploy)

(see azure-pipelines-deploy.yml)

Plus, a manifests directory containing service.yml and deployment.yml

Each microservice requires these four files. In the sample project structure, I have put them in a directory called DevOps. It is now easy to add DevOps for new microservices.

References

[Building a CI/CD pipeline for microservices on Kubernetes](https://docs.microsoft.com/en-gb/azure/architecture/microservices/ci-cd-kubernetes)

# Walkthrough

## Setup:

* Visual Studio 2019 and Docker Desktop
* Project solution structure as above
* Asp.Net Core 3.1 project with 3 microservices (1 web app and 2 Apis)
* Code in an Azure Git repo
* Azure DevOps Boards (with dummy work items)
* Azure DevOps Pipelines (effectively a Build and a Release pipeline)

This is all in an Azure DevOps project [here](https://dev.azure.com/keith0264/Canterbury). If you don’t have access, please ping me.

## Step-By-Step

1. In Azure Devops Boards, add a dummy Issue and a dummy tasks. From the Issue, create a feature branch for this work following this convention feature/featurenumber e.g. feature/123
2. In VS, pull the latest branch in via a Fetch from the Team Explorer. You should now be able to Checkout the new feature branch and work on the changes
3. You can make a dummy change to the code – say in what one of the APSs return. Commit your change(s)
4. When you do a Push back to Azure DevOps, a trigger will be fired on an update to this branch (feature/\*) and this path (/Services/ServiceA/). This triggers a build and the <todo> running of unit tests either from AzureDevOps or from inside the docker container dotnet test…
5. When ready, create a pull request in VS <your branch> into master
6. In Azure DevOps go to Pull Requests and approve your own request (normally there should be a policy to stop you approving your own request). The new microservice code is now in master.
7. When the Release manager is ready to make a release, he creates a new release branch from master using the following convention release/servicea/v1.14.0
8. This triggers a new build, test and new image creation and push to an Azure Container Repository (ACR). The trigger is on a yaml override on a branch filter release/servicea/\*
9. The pipeline continues with a deployment to a Kubernetes cluster. For PROD, we should have a manual approval step.

# Gotchas Along the Way

* Yaml – this has taken over from point and click Azure DevOps tasks (‘classic’!) so get used to it.
* Formatting
* Dockerfile paths
* Helm – ditched it for k8s manifests
* Pace of change of DevOps
* Being aware of costs, e.g. working in your VS Ent subscription with the free £115 per month – otherwise Dev Spaces would have been a strong contender
* VS solution structure and VS code generation

# Next Steps

(Note: For visibility, I will put these in Azure DevOps as work items)

* Complete the three microservice trial in DevOps (100%)
* Complete the method for splitting out the DevOps for each service’s yaml in Azure DevOps (100%)
* Include a simple .Net Test – we can actually do this in the Dockerfile (this is quite new) but still export the test results thru’ the normal Azure DevOps Task (0%)
* Sort out the need Environments (no one seems to have firm views) (50%)
* Include the ARM templates in the DevOps flow (50%)
* Introduce a manual approval step for release to Production etc (Product Owner, etc) – of course there is a decision and agreement to be made here – <todo> needs agreement
* Finish the tagging of production versions (e.g.v1.14.0) <todo> agree format. Also, best practice is to use a dedicated ACR for Prod
* Look at including a service mesh (for resiliency) like LinkerD or Istio. This could give us big benefits, and this is the time to do it – <todo> strongly recommend that we look at this now
* Design the Config in the solution for running locally (for Devs) and in a production environment – likewise this is the time to do it and get it right – <todo> see where SF is in new codebase
* General DevOps questions – do we deploy CosmosDB in a Task or in code? Etc for the other persistent data resources <todo>

# Solution Configuration <todo>

The intention of this section is to describe a way of configuring the solution so that we can handle not only local deployment and testing but remote as well via environment variable substitution in Azure DevOps.

### Dev (local)

* Local mongo in container

### Remote

* Cosmos DB (mongo protocol) for HA

## Notes

* Ocelot does not support gRPC