# Docker Introduction

* It provides the ability to run one or more application in an isolated environment called a container
* It is a containerization ecosystem

## What is Containerization?

* Involved bundling an application together with all the configuration files, libraries, and dependencies required
* We will be using **Docker** and **Kubernetes**
* You are hosting a virtual OS
* When you create a container, the allocation of resources is dynamic
  + It will only use as many resources as possible the container needs to run

## What is Virtualization?

* It is a creation of a virtual machine that stimulates a real computer with an operating system
* When you create virtual machines, the allocation of resources is static
  + Meaning once you start a virtual machine, you cannot change the resources it currently has
* It is heavy to run

# What is the purpose of Docker?

* It allows developers to work in standardize environments using containers
  + Meaning they can work on any computer with any OS they are comfortable with and still be able to share their application to everyone
* Makes it perfect for CI/CD workflow
* It makes scaling and deployment way easier
  + Docker containers can run on most things (cloud providers, data center, virtual/physical machine)
  + You can scale up or tear down application as business dictates

# Docker artifacts/terminology

## Docker Images

* They are standalone package that includes everything we need to run an application such as code itself, runtime environment, libraries, etc.
* They are immutable file and represents an application and its virtual environment at a specific point in time
  + Great for consistency when sharing your application everyone
  + Immutable meaning once you make it you can’t change it, you’ll have to create a new version of your image if you want to update it
* Essentially, they are a virtual file storage that has a copy of your files and other development platform it needs to run those files

## Docker Container

* **An image cannot run on its own,** it needs a docker container to run the image
* Docker container is the runnable instance of a docker image

## Docker ignore

* It will ignore certain files and not copy them to the image
* Like gitignore

## Docker Registry

* It is a server-side or cloud application where you can store your images and make it easy to distribute to everyone else
* Think of GitHub but just for docker images
* **Docker hub**

## Docker Configuration

* Contains all the information to tell the docker container how to run the image
* We used the docker run command to already configure this (mostly just changing what port the container should run on)

# Multi Stage Builds

* Useful for minimizing the final image size
* It is when at the beginning of the docker file, we used the .NET 5.0 SDK as our base image and then switch to .NET runtime base image after building our application since that’s the image we need to just run the application
* Essentially, we got SDK to build the application then remove the SDK and got the runtime to run the application and, in that way, save our memory
* (We used the from instructions in our docker file twice and in that way did the multi-stage build right)

# Docker Compose

* Something we don’t need right now but nice to learn when we go into microservices
* It is a tool that lets you run multiple containers at the same time
* You configure it via yaml file and with a single command you can start all the different images that is needed to start your entire application
* Essentially it becomes useful once we separate backend and frontend and those two services need to run on different containers at the same time and talk to each other

# Docker instructions

## From

* Initialize our build stage and sets the base image
* Essentially this is where we indicate what we need to be able create/run our application

## Workdir

* Sets the working directory
* Just creates a folder and this is where we will copy and paste our files into that folder

## Copy

* Copy and pastes files into the image’s virtual file system

## Run

* It will run what you put in the terminal

# Docker CLI

* This will build the docker image
  + Docker build -t [username]/[imageName]:[imageVersion] .
* This will run the docker image in a container
  + Docker run -d -p 5001:80 -t [username]/[imageName]:[imageVersion]

# DevOps

* It is a culture that you follow in which you continuously develop your application and continuously deploy it.
* There are three main teams that are part of DevOps
  + Developer team that are responsible for planning and building the application (by adding more features)
  + Operation team that are responsible for deploying and maintain the application (check that the new feature added and can handle real world environment)
    - The application must deal with being used by thousands if not millions of people
  + Business team verifies that the correct product is created and delivered
    - They check if the people even want that new feature

## Cycle time

* You observe what the market needs
* You then decide what options you should pursue to satisfy what the market needs
* Finally, you give the working software that should satisfy what the market needs

## Feedback loop

* Once released, you get feedback if it did satisfy what the market needs

## Validation

* Using the feedback, learn what needs to change and adapt your software to satisfy what the market needs

# Continuous Integration

* The process of automating the building and testing of your code every time someone tries to merge with the main branch of your repository.
* Some automated testing process will occur and if this fails, the system will prevent your code from merging with the main branch
* So, in this way, there is a lot less bugs that can roll out to the real application that is being used by everyone
* Essentially, we take out the human component of someone looking at the merge request and checking everything looks good.

# Continuous Delivery

* The extension of CI since all CI does is checks if you can merge with the main branch and now, we must deal with taking that main branch and releasing to everyone else
  + Anything from CI has been tested correctly
* Essentially, it is the automated process of delivering the new changes in the main branch to the people
* However, there is a release manager that will check on everything and make sure it looks good before allowing the entire process to continue
  + They might test it in a more real environment than the testing environment
    - Things like stress testing the application by emulating thousands or millions of people using it
  + They might release the application to a smaller set of people and see how it goes from there (beta testing for games)
  + They essentially have the ultimate power to push your code to production (real world)

# Continuous Deployment

* Everything is automated from building and testing your code (what CI does) to deploying your new version to the people (what CD does).
* Doesn’t have any human intervention so as soon as your unit testing passes, it will be deployed to the people
* Not really common in the industry since simple changes can cost millions of monies.
  + [Knightmare: A DevOps Cautionary Tale – Doug Seven](https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/)

## Creating a DevOp pipeline

1. Get publish profile in your web app
2. Go to github actions of your repository
3. Create .Net workflow as your default
4. Edit the yml to have what I have in my training repo
   1. Make sure that it is referencing your actual application (the file path might differ for yours)
5. Commit the changes
6. Go to github action and find the build agent running
7. Check that everything step went through just fine
8. Add connection strings in the configuration of your web app
9. Finally open the url to your webapp and check everything is working as intended

# SDLC

* Stands for Software Development Lifecycle
* It is the process of dividing the software development work into distinct phases
  + Let’s us manage our projects and improve the design of our projects

## 6 stages of SDLC

1. Planning/Requirement Analysis
   1. Where you plan your project
   2. You check the feasibility (if it is even possible to do) of the project
   3. All potential options you can pursue when solving the problem
      1. You find the best options pursue
2. Define the project requirements
   1. You define the document the product requirements of the project and get them approved by the customer/client
3. Design Product Architecture
   1. Based on the requirements, you design more than one design approach for the product architecture
   2. Product architecture is just all about the tech stack you will be using to deliver the product
4. Building/Developing the product
   1. Where you start coding for the application
   2. The development environment must be first established for every developer to follow
5. Testing the Product
   1. As the name suggest, this is where you will test your application and find bugs to fix
   2. The application must be in quality standard you promised with your customer/client
6. Deployment and Maintenance
   1. This is when you make your application available to the client or the client’s audience
   2. After deployment, you maintain the application to make sure there are no interruptions from the daily use of your end-users

# Different types of SDLC

* Waterfall model
  + Each phase depends on the previous phase as being completely done.
  + So, once you finish a phase you cannot go back and can only go forward.
  + It is less iterative and less flexible
  + Highly documented
  + Very rigid for most situation
* Iterative model
  + Development cycle in smaller portions at a time
  + It can be as simple as going through the entire SDLC just to implement one feature and keep doing it until a full system has been implement
  + Backtracking is possible
* V-Model
  + Means verification and validation model
  + Extension of the waterfall model exception the testing phase is done for every corresponding development
  + The testing phase of the development phase is planned in parallel
* Big Bang model
  + Does not follow any specific process
  + Requirements are implemented without much analysis
  + Usually for small project with small teams
    - Those projects are usually prototypes
    - Any project that is due tomorrow
  + Also good for just learning new language or implementing new features you never tried

# Agile

* It is more of a concept than an actual methodology unlike the previous models
* It is all about creating a working software than spending your time creating extensive documents
* You **consistently communicate** with the client about what they want in the project

## Priorities

* An early CI/CD pipeline is made
* Quick acceptance to any change in the requirements
* Quick deliver of a working project
* Regular sessions of meetings (stand ups)

## Scrum

* Implementation of the Agile concept
* It is a framework that helps team work together
* It encourages collaboration and helping each other solve problems that might arise during a sprint
* It is consisting of a series of sprints where each team has specific user stories to tackle
  + A user story is simple description of a feature in laymen’s term
  + Ex: “I want my website to have a to do list”

### What are sprints?

* It is short period that completes a set amount of work
* Usually lasts around 2-4 weeks
* After a sprint, teams will demonstrate what they’ve accomplished to each other and to the client
* During a sprint:
  + Daily stand ups
    - What you did yesterday
    - What you’ll do today
    - What blockers (problems) you have

# Introduction to Code Analysis

* Another way to further your code
* However, it doesn’t test the functionality of your code, but it tests how well written your code is
* It will perform a **static code analysis**
  + It scans your source code line by line and finds out if there is any patterns of commonly made bad coding practice
  + This is performed by an automated tool, if you do it manually it’ll be called a code review

## Terminology

* Coverage Review
  + Determines how much of your code has been thoroughly tested with unit tests
  + Also known **code coverage** that gives a percentage of how much of your code has been tested
* Technical Debt
  + Estimated time it would take to fix all maintainability issues/code smells
* Code Smells
  + Some issue in your code that might cause maintainability in the future
  + Basically, it tells you how “easy” it’ll be to make some changes in your code without breaking whole thing
  + Removing your code smells mean it’ll be easy to maintain your code and also expand it
* Duplication
  + When it found identical lines of code that can, in theory, be separated into a method to just be called or by using SOLID principles
* Quality Gate
  + The is essentially the standard you want every project to meet in terms of quality
  + Ex: Every project must be above 80% code coverage and <10 code smells then it passes the quality gate

# Introduction to Cloud

* Allows to deliver computing services over the internet (“the cloud”) to offer some sort of resource, faster application, and scalability.
* End-users don’t have to worry about maintenance or upkeep of the servers.

## Pros

* Costs
  + Don’t have worry about creating an entire infrastructure of servers.
  + You don’t have to worry about the utility’s costs.
  + Don’t have to hire IT people to establish this infrastructure.
* Scaling
  + Very simple to upscale your server as your application grows/audience who uses your application grows
  + Can be done automatically by the hosting service (Kubernetes)
* Security
  + Provided by the cloud provider.
  + Most of them are high level companies that know how to deal with cyber attacks

# Different Scaling

## Horizontal Scaling

* Adding more instances of service, resource, or application
* You must worry about data inconsistency
* Usually cheaper

## Vertical Scaling

* Adding more hardware specifications such as adding more ram, adding more storage, or replacing the CPU to have higher speeds.
* More expensive since hardware is not cheap and in terms of increase speeds cause of the hardware, it is very tiny.

# Cloud setup

## Region

* US-east or US-west

## Zones

* Actual infrastructure that has servers

## Service Level Agreements (SLA)

* A contract that promises that your server is incredibly reliability
* If you lose any data loss because your server goes down, then you can blame the cloud provider
  + Compensation will be given

# Cloud Deployment Model

## Private

* Only accessible by a certain organization or just your organization

## Public

* Open for use by the general public

## Hybrid

* Some resources are private and some resource public

## Community

* Sharing is caring
* Sharing resources when you are part of a community
* Ex: University

# Cloud Services Types

* They dictate how much control you have over the service you are using

## IaaS

* Infrastructure as a Service
* You get to design what type of infrastructure you want for your application
* How many computers/servers
* What specs for those computers?
* What type of OS these servers should have?
* Essentially like you have the infrastructure(building) itself and you manage it
* All of it managed over the internet so you don’t have to come to the physical address of the infrastructure

## PaaS

* Platform as a Service
* Windows Azure services is a PaaS
* You are more focused on the actual computer itself
* You can select how much specs we want to add (as you saw during our demo)
* You can select where to put this service
* We have no control of the actual infrastructure itself we just dictate what it needs
* Dictating how much network, storage, OS, data, applications it has

## SaaS

* Software as a Service
* You provide software and the clout provider will run and manage it with their service
* The least amount of control you have
* You just give the application they’ll handle the rest
* You can update the application at least

# Git Branching

* Incredible feature when it comes to working with multiple people
* Right now, we only have the main branch but in git you can create multiple branches
* Why do you want to have branches?
  + It lets developers work at the same time on the same application with their own computer and after they are finish can just merge their own work to the original branch
* Essentially, it lets you work on multiple versions of the project at the same time

# CLI

* Git checkout -b [nameOfBranch]
  + Creates a new branch and copies everything from whatever branch you current are in
* Git checkout [nameOfBranch]
  + Lets you switch branches
* Git branch
  + It lets you view all the branches in your local repo
* Git branch -d [nameOfBranch]
  + Deletes that branch

## Pull Request

* Allows you to see all the changes that was made from a feature branch and you decide if you want to merge to the main branch

## Merge conflict

* When two different versions of the same file are being merge together and GitHub doesn’t know which version to keep
* You as a developer must decide which lines of code needs to be kept/change/manipulated to make it work in the main branch
* At least your CI pipeline can fail if you didn’t resolve the conflict right