Revature Notes

**Monday 2/18**

**Week Layout**

1 – C#, NET, OOP, testing

2 – SQL (SQL Server), Entity Framework (ORM)

3 – ASP.NET MVC, HTML, CSS [Project 0 due mid-week]

4 – DevOps, Continuous Integration and Deployment, Azure, Docker [Project 1]

5 – Service Oriented Architecture, REST, JS

6 – TypeScript, Angular (JS)

7…9 – Projects 2 and 3, Panels

10 – ??? Specialized Framework

Monday – Assessments (Quiz, Timed Coding, Mock Interviews, QC Audit)

Tuesday…Thursday – Training

Friday – Training and Project Work Time

12:30 to 1:30 is Lunch.

Sometimes Nick may leave, which is also Project Work time.

SDK: Software Development Kit, contains compiler and basic libraries (and more). Code that allows writing code.

**Commands**

cd *path:* change directory (current location).

mkdir *name*: make directory (folder) with given name.

ls: list files in current folder.

rm: Remove deletes files. -r option recursively deletes inside the folder.

.: Current directory

..: Parent directory

git clone *url*: make local copy of git repository from given url

git status: see status of the git directory you’re in

git pull: update git directory from repo.

git add *file*: to include file in what will be committed (stage changes)

git reset HEAD *file:* to unstage file from what will be committed.

dotnet –version: Check version of .net.

dotnet build: Compiles code

dotnet run: runs compiled code.

dotnet new console: Create new console app.

dotnet sln add *file:* Add file to a dotnet solution.

code *path:* Opens path in VScode.

tab autofills if single option (or as much shared across multiple options with same beginning). Double tab will print out all the remaining possible options.

**VSCode – C#**

alt-shift-F is auto-Format code

compile-time type inference for variables with var but must be initialized!

Copies type of right-hand side.

Can use var when type is obvious from context (or when it is obnoxiously long to type).

var somedata = "String type";

var otherdata; //Left Statement wouldn't work

**Structure**

solution *solName*.sln set of related projects

project/assembly *projName*.csproj Assembly are .exe for apps or dll =dynamic linked library for libraries (What .net created automatically with dotnet new)

file (*filename*.cs)

namespace(*solName*.*name*) -> using *name*  default System

Class (*classname*) want classname to match filename

methods

properties

fields

Logically organized into namespaces, physically organized into projects.

**C# Tour**

Assemblies contain executable code in the form of Intermediate Language (IL) instructions, and symbolic information in the form of metadata. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for #include directives and header files in C#

**Types**

char type represents a UTF-16 code unit.

Nullable value types do not have to be declared before they can be used. For each non-nullable value type T there is a corresponding nullable T?, which can hold an additional value of null.

Single- and multi-dimensional arrays of any type. For example, int[,] is a two-dimensional array of ints, and int[][] is a single-dimensional array of 1d arrays of ints. (Do not have to be declared.)

Class types support single inheritance and polymorphism. (extend and specialize base classes)

Struct types do not support user-specified inheritance; all struct types implicitly inherit from type object. Structs are value types (stores the data of the obj, not a reference to a dynamically allocated object) and do not typically require heap allocation. Structs are particularly useful for small data structures that have value semantics, which can make a large difference in the number of memory allocations an app performs. Exs: key-value pairs, coordinates.

An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces. A class or struct that implements an interface must provide implementations of the interface’s function members.

A delegate type represents references to methods with a parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They are also similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes this in the invocation. Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. An interesting and useful property of a delegate is it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

The class, struct, interface and delegate types all support generics, whereby they can be parameterized with other types.

An enum type is a type with **named constants**. Every enum has an underlying type, which must be **one of the eight integral** types. The set of values of an enum is the same as its underlying.

Values of reference types are treated as objects simply by viewing the values as type object. Values of value types are treated as objects by performing boxing and unboxing operations: object o = i; Boxing

int j = (int) o; Unboxing

**Operators**

The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).

Primary

new T(...){...}: Object creation with initializer

new {...}: Anonymous object initializer

delegate {...}: Anonymous function (anonymous method)

checked(x): Evaluate expression in checked context

unchecked(x): Evaluate expression in unchecked context

typeof(T): Obtain Type object for T

default(T): Obtain default value of type T

Unary

+x: Identity

-x: Negation

~x: Bitwise negation

(T)x: Explicitly convert x to type T

await x: Asynchronously wait for x to complete

Relational and type testing

x is T: Return true if x is a T, false otherwise

x as T: Return x typed as T, or null if x is not a T

Logical XOR

x ^ y: Integer bitwise XOR, Boolean logical XOR

Null coalescing

x ?? y: Evaluates to y if x is null, to x otherwise

Conditional

x ? y : z: Evaluates y if x is true, z if x is false

anonymous function

(T x) => y: Anonymous function (lambda expression)

**Statements**

The checked and unchecked statements are used to control the overflow-checking context for integral-type arithmetic operations and conversions. (Uncheck has overflow, checked exception.)

The lock statement is used to obtain the mutual-exclusion lock for a given object, execute statements in a block, and then release the lock.

The using statement is used to obtain a resource, execute a statement, and then dispose of that resource. Ex: using (TextWriter w = File.CreateText("test.txt") ){ w.WriteLine("Line one"); }

Yield: You use a yield return statement to return each element one at a time, thereby making an Iterator to be used with a foreach statement or LINQ query.

yield return <expression>; returns the expression and keeps track of the location in the code for when execution is restarted from the iterator being called again.

yield break; ends iterating.

The return type of the iterator must be an IEnumerable or IEnumerator, which returns an object, but giving a <T> implicitly converts the expression to the type parameter.

**Accessibility**

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are six possible forms of accessibility.

public Access not limited

protected Access limited to this class or classes derived from this class

internal Access limited to the current assembly (.exe, .dll, etc.)

protected internal Access limited to derived classes OR classes within the same assembly

private Access limited to this class

private protected Access limited to self/derived classes that are also w/i the same assembly

Ranking from most to least public: public, protected internal, protected or internal, private protected, and private.

**Generics**

A generic class type’s definition specifies type parameters after the class name with angle brackets enclosing a list of names. Ex: public class Name<TFirst>{ public TFirst First; }

When the generic class is used, type arguments must be provided. Name<string> variable = new Name<string> { First = “one”;} string s = variable.First;

A generic type with type arguments provided is called a constructed type.

Static field has one storage location shared by all instances; non-statics each have their own.

**CLASSES AND OBJECTS**

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Hidden default is object. An implicit conversion exists from a class type to any of its base class types (upcasting). Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class.

Read-only fields are declared with a readonly modifier. Assignment to a readonly field can only occur as part of the field’s declaration or in a constructor in the same class.

**-Parameters** Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The signature of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters, modifiers, and types of its parameters. The signature of a method does **not** include the return type.

A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed. Can be optional, by specifying a default value when the argument is omitted.

A reference parameter is declared with the ref modifier. The argument passed for a reference parameter must be a variable with a definite value, and during execution of the method, the reference parameter represents the same storage location as the argument variable.

An output parameter is declared with the out modifier. It is used for passing arguments by reference. It's like a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. (In the method, can assign values to args, so that the caller can use their own variables without needing a new object to hold all return values.)

A parameter array permits a variable number of arguments to be passed to a method. A parameter array is declared with the params modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type.

In an invocation of a method with a parameter array, can pass any number of args of the type, and an array instance is automatically created and initialized with the given args. If zero passed, creates an empty array. (Within the method, the parameter array is exactly like a regular array.) If exactly one passed, acts as a value parameter!

C# requires a local variable to be definitely assigned before its value can be obtained.

**Function Members** - Methods A static method (declared with a static modifier) does not operate on a specific instance and can only directly access static members.

A method declared without a static modifier is an instance method. An instance method operates on a specific invoked instance (explicitly accessed as this) and can access both static and instance members. (It is an error to refer to this in a static method.)

When a *virtual* method (declared with virtual modifier) is invoked, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a *nonvirtual* method invocation, the *compile-time type* of the instance is the determining factor.

When an instance method declaration includes an override modifier, the method overrides an inherited virtual method with the same signature; it specializes an existing inherited virtual method by providing a new implementation of that method.

An abstract method is a virtual method with no implementation. An abstract method is declared with the abstract modifier and is permitted only in a class that is also declared abstract. An abstract method must be overridden in every non-abstract derived class.

-Constructors An instance constructor is a member to initialize an instance of a class. A static constructor is a member to initialize a class itself when it is first loaded, declared with static.

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

-Properties Instead of using getters and setters, C# has properties.

private string \_name;

public string Name {

get{ return \_name; }

set {inside "set"

we have implicit argument "value"

could do null/empty-checks, etc.

\_name = value;}

}

access type *name* { get; set; } = default\_value; Read only properties do not have a set.

Static properties are declared with the static modifier, and instance properties are declared without it.

When a property declaration includes a virtual, abstract, or override modifier, it applies to the **accessor(s)** of the property.

-Indexer An indexer is a member that enables objects to be indexed in the same way as an array. (Think List obj accessed thru []). An indexer is declared like a property except that the name of the member is this followed by a parameter list written between the delimiters. The parameters are available in the accessor(s) of the indexer. Indexers can be overloaded, meaning that a class can declare multiple indexers if the number or types of their parameters differ. They can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

-Event An event is a member that enables a class or object to provide notifications. It is declared like a field except it includes an event keyword and the type must be a delegate type.

The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is null.

Clients react to events through event handlers. Event handlers are attached using the += operator and removed using the -= operator. Example:

static void ListChanged(object sender, EventArgs e) { changeCount++; }

public static void Usage() {

List<string> names = new List<string>();

names.Changed += new EventHandler(ListChanged);

names.Add("Liz");

} Add triggers Changed which runs ListChanged, incrementing changeCount.

-Operators All operators must be declared as public and static. An operator is a member that defines the meaning of applying an expression operator to instances of the class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.

-Finalizers The finalizer for an instance is invoked automatically during garbage collection. A finalizer is a member that implements the actions required to finalize an instance of a class. Finalizers cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. Classes should generally not implement finalizers, for many reasons.

**Arrays** Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at runtime using new, which specifies the fixed length of the array instance, and automatically initializes elements of an array to their default, for example, zero for numeric types and null for reference types. Multi-dimensional arrays of size n specified with n-1 commas in the [] delimiters. An array with elements of an array type is also called a jagged array because the lengths of the element arrays do not have to be the same: int[][] a = new int[2][]; a[0] = new int[10]; a[1] = new int[5]; Arrays can be initialized in {}, thereby skipping need to give size, or repeat new and array type.

**Interfaces** An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines. Interfaces may employ multiple inheritance, and classes and structs can implement multiple interfaces. When a class or struct implements an interface, instances of that class or struct can be implicitly converted to that interface type. C# also supports explicit interface member implementations, enabling the class or struct to avoid making the members public, but these can only be accessed when of the interface type.

**Enums** Define enums when you need to define a type that can have a set of discrete values, each a named constant. They use one of the integral value types as their underlying storage. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type. Enum values can be converted to integral values and back using type casts.

**Attributes** C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using attributes. All attribute classes derive from the Attribute base class provided by the standard library. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. The metadata defined by attributes can be read and manipulated at runtime using reflection. When an attribute is requested using this technique, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

**Tuesday 2/19**

**Git** Software for versioning control.

**GitHub** Hosts repositories. Adds some access controls.

Central Version Control System (VCS): One central version (everyone has to merge with)

Distributed - : Everyone has a copy. Can agree on one (GitHub).

Working directory/tree: Has files that everyone (non-git aware programs) can see.

Index/staging area: Temporary place to prepare for commit.

Local Repo: Permanent record of history in the commits.

Remote Repo: Usually GitHub. Default for clone is origin. Someone else’s record of history.

Add: Working tree -> staging area.

Reset: Staging -> tree. (Takes out. The rest all copy over.)

Commit: Staging area -> Local.

Push: Local -> Remote.

Pull: Remote -> Local and leftwards.

Stash: Temporary stack changes. Move stuff out of the way without committing.

Diff: Tree 🡨🡪Staging

Diff --: Staging 🡨🡪 Local

**Modifiers**

**static** classes cannot be instantiated or derived from, they’re just containers for static members.

readonly members are like const but can be set in a constructor.

abstract members must be in an abstract class (other methods can be non-abstract).

**sealed** classes cannot be derived from (mainly to prevent overriding).

**partial** classes are spread across multiple files in the same namespace (helps allow computer generated code).

|  |  |  |
| --- | --- | --- |
|  | Class | Member |
| static | Y | Y |
| virtual | N | Y |
| override | N | Y |
| new | N | Y |
| const | N | Y |
| readonly | N | Y |
| abstract | Y | Y |
| Sealed | Y | N |
| Partial | Y | N |

**Errors**

In VSC, ctrl+click to see metadata of what you clicked.

Can catch all exceptions to log them, but make sure to throw again if you don’t handle it. Can wrap exception in another exception if want both stack traces.

Finally block happens regardless of if an exception happened or not, usually to close resources. Code after finally block won’t happen if exception wasn’t handled.

**Wednesday 2/20**

**GIT**

Origin/master is the remote master branch of what is on GitHub. Your local repo can commit but pushing is what adds your commits to master and updates what your repo considers the master.

Branch is a reference to a commit, which includes the history of its commits, and they can only move forwards.

You can create another branch and merge it with master (git merge *name:* merges named branch with current repo).

Pushing a local branch which has not been merged with master will cause an error, you need to use a different version of push. This will create another remote branch on github, origin/*name,* which now needs a tracker on the local repo.

Pull (fetch and merge – updates remote tracking branches and merges the remote tracking branches into associated current local branches)

2nd person to want to make changes to master must resolve conflicts using merge. New master is child of the master updated from the 1st person and your branch.

HEAD is what commit you are looking at.

git checkout *name*: Switches to named branch. -b creates a new branch from the current commit.

**Collections**

Like arrays, array lists are not really used. Replaced by Generic's List<T>.

Set has Math def of set -> Not ordered, no repeats. Designed to check for membership (ContainsKey or ContainsValue). Fast to Add or Remove.

Dictionary Mapping: Key-value pairs. Both can be any type.

Stack: LIFO. Queue: FIFO. List:

Classes are reference types. Value types like int are not. Structs are value types!

Value types only give a copy of their value(s), so modifying does not change the original.

Reference types give a reference to an object, so modifying them does affect the original.

Value types are deleted from memory once they are no longer in scope.

Reference types need garbage collection since it needs to wait until all variables that reference it pass out of scope.

Managed code has periodic garbage collection to delete unreachable objs.

Unmanaged code needs to manually delete.

Programmer time more valuable than computer time, so use managed code.

Normally == compares value types by value and reference types by reference.

The exception is strings: overloaded to prevent awkward string.Equals()

Should compare object values using .Equals()

Value types are derived from object, so can implicit upcast to an object variable, called boxing.

New object treated as reference type. (Changing original doesn't change the new!)

Can unbox as well by downcasting back to value.

**Testing**

Arrange, Act, Assert

A test method returns void and should be named what it tests.

Should make initial tests that fail and fix them by writing the correct code.

Attribute xunit test with Fact.

**Thursday and Friday 2/21 + 2/22**

Facts don't allow any parameters.

Theories accept sets of parameters, to run the test against all of them.

Flaw in trying to override with "new". you can still run the parent class's implementation using upcasting.

Params type[] name: lets you pass any number of parameters.

Documentation should contain tags such as:

<summary>…</summary>, which gives a brief description of the class/method.

<param name=”*name*”>…</param>, which describes the named parameter.

<exception cref="*name*”>…</exception>, which describes the condition of the exception.

<returns>…</returns>, which describes what the returned value should be.

<remarks>…</remarks>, which gives any further remarks not in summary.

**Constructors** are not inherited, but can manually inherit them by defining child constructor with base:public childClassname() : base() {}, which is done by default already, or

public childClassname(type value) : base(value) {} Thus it uses the parent’s constructor.

C# prefers thin constructors.

**.NET** platform – Many languages. Limited interoperability.

C# most popular. Python, Java, TypeScript/JS, F#, R, VB.NET.

Common Language Infrastructure (CLI, shares acronym with Command Line Interface).

Virtual Execution System (VES) is equivalent to JVM.

.NET code (.cs file) compiled by C# compiler into assembly/Common Intermediate Language (CIL) in a .dll or .exe (bytecode), creating cross-platform instruction set for VES. (CIL is barely below other high-level langs.) Has a Common Type System (CTS) for converting basic types, and a Base Class Library (BCL) for sharing often used classes. Exception-handling, types, managed environ (garbage collection). Happens in build.

JIT (Just In Time) Compiler converts that into native instructions for specific CPU right when it is needed. (In .net, this is no longer considered assembly.) Happens in run.

.NET Implementations:

.NET Framework is a Windows only implementation. 4.7 most recent.

(Mono ported .net framework to Mac and Linux. Exists, but kind of obsolete.)

.NET Core is a truly cross-platform implementation. 2.2 most recent.

Common Language Runtime (CLR) is the VES in both Framework and Core.

.NET Standard is the 100% common subset (intersection) between Framework and Core.

Not an actual implementation. Lots of libraries. Some are still platform-specific but being added to core.

**OOP**

Abstraction: “Separation between needed functionality and implementation details.”

Knowing what something does without needing to know how it does it. e.g. properties, methods

Encapsulation: In the context of C#, “Restricting access to information within objs. Packaging related things together and treating them as one unit.” e.g. access modifiers. Objects containing fields and their methods.

Polymorphism: “Many implementations behind a common contract.” Many names. e.g. Method overriding (class redefines parent method) and overloading (same name but different params).

Inheritance: “Ability for a class to take and extend behavior from another class.” Base, parent fields and methods are present in child, which can add more or specialize them. Is-a relationship.

(Composition: Has-a relationship. e.g. Class has a member of another class, which already defined methods and properties for certain fields. )

**SOLID** principles-

**S**ingle-responsibility principle**:** Class/method should do just one thing; If it does more, divide into smaller pieces.

**O**pen-closed principle: Entities should be open for extension, closed for modification. Should generally add, but not change (affects backwards compatibility and maintenance). e.g. Virtual keyword in parent needed to override.

**L**iskov Substitution principle: Objects should be replaceable by instances of their subtype without affecting correctness of code. e.g. child implicit upcast to parent should run.

**I**nterface Segregations principle:Heavy interface should be split up into multiple light interfaces. e.g. only want some methods, don’t couple with unwanted functionality. Splitting doesn’t really add any cost; Modularity adds benefits.

**D**ependency Inversion Principle**:** Instead of high-lvl code depending on low, or vice versa, both should depend on interfaces. e.g. easier to test and extend using interfaces. (unit testing with dummies vs integration testing) [Abstract classes can’t create new objects, which would create more dependencies, so better than normal classes, but still prefer interfaces.]

**Serializing**

Serialize, meaning to put in a row, will collect data from across memory into another (text or binary) format. Ideally reversible, thereby allowing transmitting across network.

Doesn't transmit type info or methods. Know other side needs class definition to recreate.

**Asynchronous methods** need async modifier and return a **Task<thing>,** or Task for void, and should have async in name (for documentation purposes).

**.Result** makes you wait for code to finish which is bad, cause contrary to async purpose, but we have to because it's Main If Task is returning void, use .Wait() instead, also makes synchronous.

FileMode.Create (over)writes to file with given name. FileMode.Open to read file.

FileStream obj might be null, close with ?.Dispose in finally (stream uses IDisposable).

XmlSerializer needs a stream and does not know about generics, so give typeof obj being serialized: var *name* = new XmlSerializer(typeof(T)); serializer.Serialize(*fileStream*, *obj*);

return (<T>) serializer.Deserialize(*fileStream*); //Deserializer needs to be type casted!

**“using statement”** is different from “using directive” (at top of file). using statement automatically disposes created resources when exiting its block, so we don't need try finally dispose: // should still be try-catching throughout this method

using (var memoryStream = new MemoryStream()) {

using(var fileStream = new FileStream(filename, FileMode.Open)) {

await fileStream.CopyToAsync(memoryStream); //XmlSerializer.DeserializeAsync doesn't exist, so we are using fileStream.CopyToAsync(memoryStream)

//await allows other code to execute while we are running the statement

}

memoryStream.Position = 0; //reset “cursor” to start in order to read contents later

return (T)serializer.Deserialize(memoryStream);

}

**JSON**.NET (aka NewtonSoft JSON) is third-party. Json doesn't care about some xmlAttributes but does care about XmlIgnore.

File.ReadAllTextAsync(“*fileLocation*”); //Asynchronously read data from file

JsonConvert.DeserializeObject<T>(*readData*); //deserializes json formatted data

JsonConvert.SerializeObject(*dataToWrite*); //serialize data to json format

File.WriteAllTextAsync(“*fileLocation*”, *serializedData*); //Asynch write data to file

Normally, backslash is escape character in strings. To treat backslash literally, use **@ string.**

A **"lambda"** is like a method that's anonymous and can be treated like an ordinary value/object.

**LINQ**

Average<TSource>(this IEnumerable<TSource> source, Func<TSource, type> selector);

//Calculate the average of an enumerable obj, based on results from a passed selector function, which takes an input of the Collection’s type and returns type (int, double, etc.) value.

Similar for Sum. Ex: list.Sum(x => x.Count(c => c == 'a') ); //For each string in list, for each character in string, if char is a, add 1 to count for that string, add all returned counts.

Can use Func variable, or method, that matches the typing.

In Visual Studio, can right click and go to properties to get a GUI for modifying the .csproj. (Could also modify directly, but don’t know how to do so properly.)

This enables compiling unsafe code, such as pointers (literal memory location, less abstract than ref variables; shouldn’t be used). In main, place into unsafe block.

We need unsafe code when we are using unmanaged code, like Windows API, that require pointers. Otherwise, avoid it.

**IS null** is a duplicate comparator for == null.

Global assembly cache (C:\Windows\assembly) stores dependencies using strong naming.

Added package manager NuGet for specific projects. dotnet restore fetches dependencies, since integrated with NuGet. (If NuGet is on path, can call nuget restore.)

**NLog** package dependency per project, NLog.config just downloads a .config you can copy over.

<!--

Write events to a file with the date in the filename.

<target xsi:type="File" name="f" fileName="${basedir}/logs/${shortdate}.log"

layout="${longdate} ${uppercase:${level}} ${message}" /> -->

<!--

Write events to the console.

<target xsi:type="Console" name="c" layout="${longdate} ${uppercase:${level}}

-->

Make sure to add target name to logger: <logger … writeTo=”f, c” />

Add “using NLog;” directive to use logger.

Create logger obj: ILogger logger = LogManager.GetCurrentClassLogger();

Multiple event levels, Debug, Info, Warn, Error, and Fatal, which is the worst. Logger logs those of minlevel=”*level*” and above.

Log with: logger.Debug(string); logger.Error(exception); etc.

**Regex** needs using System.Text.RegularExpressions

Regex uses different escape characters, use @strings.

regex syntax - "character classes":

\d for all digits,

\w for all "word characters" (letters, numbers, and underscore)

\s for all whitespace, most of these have an "opposite" version with uppercase

\S for all non-whitespace

[abcd] means, one character, EITHER a, b, c, or d.

a\* means 0 to many a chars. a+ means 1 to many a chars

() are for surrounding groups of characters that you want to extract later.

Group 0 will be whole match and remaining groups are in order of parenthesis.

**Separation of Concerns**:Minimize tight coupling, use loose coupling (easy to switch to another implementation).

**Comment your code**: Use XML comments for the public API (not needed for private), so someone else can auto-generate pages to understand your code.

**DRY**: Don’t Repeat Yourself. Repeated code should be in a separate class/method.

Keep It Simple

Yield return for things with IEnumerable, to return things one at a time. (holds location in method for next call)

--

**Project 0 initial requirements:**

Customers

Stores/locations //location field in store

Order/order history //OrderHist collection of orders

Statistics //methods to run

Inventories (per store. when ordered, decrement)

Something you can buy that’s not 1:1, e.g. pizza is dough and cheese.

Buying in multiplies.

Some constraints, e.g. no more than 10 pizza per order, that user can then correct.

Leverage Menu-based console interface. Simple console: 1+enter = make purchase, 2+enter=review orders, etc. DO LAST!

Make sure to keep console and library parts separate. (Library will be used in next proj.)

Have Testing.

Use Serialization to save and load to/from file(s). //Will be replaced by sql.

**More**

3 projects – UI/ConsoleApp, Library, and Tests.

Suggest repositories.

By Wednesday have library + tests, and then console app.

**Tuesday 2/26**

Private Class: Normally don’t exist but can nest a private class inside another class.

Pluralsight is helpful study website. Microsoft Dev Essentials gives free one month.

Microsoft Virtual Academy as well. Programming in C# jump start recommended for basics.

**Azure** – Create a resource: “Create new” button, name resource group.

Needs a server: “Create new” button, globally unique name. Give admin a name (not admin) and password. Pick server location. (Don’t have to redo all this but need to know how.)

Name database. Unique in server

Firewalls block automatically, need to whitelist. How to change firewall to allow new IP to access it: Go to server. Click firewall settings. Add name, current address (or range) and SAVE.

**In MSSMS**, connect to *servername*.database.windows.net. Make sure correct database is selected in drop down list on toolbar! Often resets.

ctrl+n is new query, click button in toolbar.

Strings are in single quotes ‘ ’, not doubles “ ” .

**Relational Databases**

Divided into Tables, which are made up of Rows and Columns.

Database Management Systems (DBMS)

Relational Database Management Systems (RDBMS)

Structured Query Language (SQL)

SQL Server (AKA T-SQL or Transact SQL). Note: a SQL server is both software language and database, both are named SQL Server in this case. T-SQL is just a name for the language.

Other versions like MySQL, Oracle SQL, PostgreSQL, SQLite.

Functional Dependency: For two sets of columns, X and Y, X🡪Y if for each possible set of X values, there is exactly one possible set of Y values. Given same X, you know same Y. (Y is a fact about X.) Often sets are one column, but not always!

Candidate Key (of a table): Minimal set of columns in a table (as in cannot remove one column and still have the following be true) such that every other column depends on the set.

Can have multiples in a table.

The values of any candidate key can uniquely identify a row.

Composite key: A key of more than one column.

Primary Key (of a table), PK: One of a table’s candidate keys that we choose to actually identify the row. Remember can be multiple columns. (Generally, choose simplest to compare and begin name with \_ . )

Foreign Key: A column that references a primary key of a table to establish a relationship between the rows (can be two separate tables or same table).

Referential integrity: FK should not point at nothing.

Non-atomic value: Column with more than one value (a list). Hard to read values, you have to search through it!

Update anomaly: Problem arising from redundant data, you might not update all copies. (Similar to copy-paste problem; use methods)

Deletion anomaly: If we delete some data (row), we sometimes lose other data.

(Similar to tight-coupling problem.)

Insertion anomaly: Impossible to insert some data without associated data. Related to above.

Normalization:

3rd is considered fully normalized. For simple databases, higher forms can often satisfy the lower forms. Generally, not enough advantage to enforce lower forms (BCNF, 4-6, and lower).

1st: No duplicate rows, enforced with primary key. (Theorists assume this from definition of Relation DB.) Needs atomic values. No “repeating groups of columns” (shouldn’t have color1, color2, color3).

Could copy rows but with different color values, where color column is added to primary key (forcing composite key). Or could add rows (new pk value, individual color values).

Best is to make another table, in 1:N relationship, for the N side of the relationship and use a foreign key in the 1 side to N’s table, or, for M:N relationship, make a new table for both the M and N, with a composite primary key of both, which are FKs back to appropriate tables.

2nd: No non-candidate key column may depend on part of any composite candidate key; No partial dependency, depend on whole key. If no composite keys, then 1st NF implies 2nd NF.

3rd: No non-candidate key column should depend on any other non-candidate key column; No transitive dependencies (everything already dependent on candidate keys).

Summary: (1st) Depend on the keys, (2nd) the whole keys, and (3rd) nothing but the keys.

Normalization – Pros: Less redundancy means less errors. Easier to evolve data model. Easier to read and write consistently.

-Cons: Makes some queries slower (Faster to read denormalized). [Many get faster.]

**SQL**

--Comment with dash dash

Many databases per server. Schema is like namespace. Can have multiple databases with same name in different schema.

Azure SQL DB doesn’t support USE *dbname;* Must use dropdown.

Each database has multiple schemas, which namespace/scope for database objects.

Whitespace doesn’t matter. Semicolon isn’t needed. SQL syntax is case insensitive.

Select (highlight) statement and press F5 to run.

SQL’s default string comparison is case insensitive. It depends on “collation”, which is configurable.

Strings are in single quotes! ‘ ’

SQL doesn’t need to track conceptual candidate keys and dependencies.

SELECT statement: Returns a value. (Others just do something, like void return.)

SELECT \* FROM *tablename* is all columns and all rows in given table.

Replace \* with column names to return those specific named columns.

May need to give schemaName.tablename.

Can rename using AS [*name*]. SQL Server uses [], standard is “”. Can use without either, if name has no spaces in it.

Can compute new values from column values. Ex: Select FirstName + ‘ ’ + LastName AS [Full Name] From SalesLT.Customer; -- Can’t copy this over because ‘’ are wrong characters!

Use WHERE *condition(s)* clauseto return specific rows matching that condition. Allows AND, OR, NOT and not equal (!= or <>).

Use ORDER BY *column(s)* clause to sort results. Orders by first column, and on ties it orders by second. Each individual column comparison is ascending by default, but can be descending with DESC. (can be different for separate columns). Ex: Select \* From SalesLT.Customer Order By FirstName Desc, LastName;

We have ordered comparison of numbers, dates, times and strings. Strings are in dictionary (lexicographic) order, but this is affected by collation. Comparators are <, <=, >, >=.

Ex: Select \* From SalesLT.Customer Where FirstName >= 'c' AND FirstName < 'd';

-- Can’t use > b because would include names that start with b. Everything with c and more letters is sorted after c, so use >=c.

LIKE comparator: In SQL Server, [abc] is a or b or c. % is any number of any character (ultimate wildcard). \_ for one of any character.

In SQL Server, comment and uncomment shortcuts are the same (ctrl+k,c and ctrl-k,u).

Data types –

Numeric:

TINYINT (1byte, equivalent to C# byte),

SMALLINT (2bytes, equivalent to C# short),

INT 4 bytes, like C#, this is what we generally use (when there are no special requirements)

BIGINT (8bytes, equivalent to C# long)

Floating Point:

FLOAT, REAL, and DECIMAL/NUMERIC (highest precision and custom precision).

Decimal(4,3) means 4 digits with 3 after the decimal point.

Currency: MONEY

String: Use 1byte-per-char encoding specified in collation.

CHAR/CHARACTER(n), fixed-length character array (string) of size n, not really used.

VARCHAR/CHARACTER VARYING(n), variable length char array (string) up to size n.

NCHAR(n), Unicode char.

NVARCHAR(n), Unicode varchar. This is what we use all the time.

BIT: A literal bit (0 or 1).

Can use (max) to set to maximum size for a varchar.

‘abc’ is a varchar, implicitly converted to nvarchar when comparing.

N’abc’ is a nvarchar (or nchar).

Date:

DATE for dates.

TIME for times.

DATETIME for time and day, but low-precision and limited range (default c# datetime value is out of range of this) so use ->

DATETIME2(n) for high-precision, wide-range timestamps. N is for the precision.

DATETIMEOFFSET for intervals of time.

We can EXTRACT to get parts of these, like YEAR from a DATETIME2.

Implicit conversions from strings, useful for comparisons.

Advanced SELECT -

GROUP BY clause:

Useful with aggregate functions, which take in many values and return one. Built-in are COUNT, SUM, AVG, MAX, and MIN.

Group By accepts a list of columns, and then all rows which share the same value of all those columns are combined into one row in the result set. Columns which are not part of group by cannot be used in select because it can be different in the rows that were originally combined), so either you make it part of the group by (make it a basis for combining the rows), or you run some aggregate function which says how to turn those many values into one value.

Can’t use an aggregate in the Where clause!

First rows from the tables are filtered with Where, then we run any aggregations with Group By. If we want to run with conditions on the aggregate rows, then we need the HAVING clause.

Ex: Select FirstName From SalesLT.Customer Where LastName <= ‘n’ Group By FirstName Having Count(Firstname) =1;

Note we can use columns in condition that are not in are Select because…

Runs in order we write them except Select clause is after Having and Before Order By:

5. Select

1. From

2. Where

3. Group By

4. Having

6. Order By

**Wednesday 2/27**

SOLID Principles -

**S**ingle Responsibility: If multiple responsibilities, should be able to break it down into smaller components.

**O**pen/Closed: Children classes should generally add functionality, but not change the original.

**L**iskov Substitution: Child class should work in any case as its base class. (Implicit upcasting)

**I**nterface Segregation: Similar to single responsibility, but for interfaces. Child class shouldn’t have to implement methods it won’t need. Want modularity.

**D**ependency Inversion: Decrease coupling. High Level code should not depend on low level and vice-versa, both should depend upon abstractions (interfaces).

VES is Virtual Execution System, the runtime system of the Common Language Infrastructure.

vs

CLR is Common Language Runtime, .NET Framework's implementation of VES.

Func vs lambda.

Both are anonymous. Func<> takes some number of input and returns an output. If a lambda returns a value, it is a Func delegate. If it doesn't, it is an Action delegate.

--

Can use an alias (AS *name*) before it is defined.

Ex: Select e1.\*, e2.\* From Employee AS e1 CROSS JOIN Employee AS e2;

CROSS JOIN: Same as cartesian product, multiply. Every row combination, with result having the columns from both tables.

INNER JOIN: Most common type of join, takes row pairs which match given condition. Most common type of condition is ON t1.ForeignKey = t2.PrimaryKKey (that FK references).

OUTER JOINS -

FULL JOIN: Keep all rows from both tables, with null where there is no matching row from the other table.

LEFT JOIN: Keep all rows from left, with nulls on right.

RIGHT JOIN: Keep all rows from right, with nulls on left.

JOINS are done in FROM clause.

SUBQUERIES

--every track that has never been purchased

Select \*

From Track

Where TrackId NOT IN (Select TrackId From InvoiceLine);

--Most popular track

Select \*

From Track

Where TrackId = (Select TOP(1) TrackId --TOP(n) takes just first N results from query

From InvoiceLine

Group By TrackId

Order By Count(\*) Desc); --If you use order by in a subquery, need top (or some other things) to work

Subqueries return a table, so can join on them.

For readability, can define subquery above: WITH *name* AS (subquery) query;

EXISTS and NOT EXISTS: If any rows returned, true. If none, false. And the opposite.

Can use comparators (>=, <=, etc.) as well as ANY and ALL for or-ing or and-ing the condition check on all the rows.

Subquery Operators: Exists, Not Exists, Some/Any and All.

LEN() gives length of string.

Union, Intersect and Except (Set Difference) from Math are used.

The numbers and types of the two queries' result columns must be compatible.

By default, gives distinct rows. Use query *setOperation* ALL query, to allow duplicates.

Union gives row values in either result set (OR), Intersect gives values in both result sets (AND), and except is values in one result set but not the other result set (A-B).

Data Manipulation Language (DML) operates on rows.

5 operations: Select, Insert, Update, Delete, Truncate.

Insert Into *tablename* (*columnNames*,…) Values (*values*,…), *moreRows*, …;

Can leave off column name section, but this is more readable/less error prone, and allows skipping columns with acceptable default values.

Can be more complex, like inserting the result of a query. Doesn't need "Values".

Update *tablename* Set columnName = newValue, column2 =new2,… Where condition; --leaving out where condition would affect every row in the table.

Substring(*stringName*, *startingPosition*, *length*)

Delete From *tablename* Where condition; --w/o where, would delete every row, one at a time

--Truncate Table *tablename*; --deletes every row, all at once. Faster, but less info in logs.

Deleting a row which has a foreign key pointing at it would violate referential integrity, so DB Raise Error: Throws an error, which is the default.

Set Null: 1 solution is to set those keys to null. (has to be a nullable column)

Cascade: Other is to delete all dependent rows.

Data Definition Language (DDL) operates on tables at a time. Can't see individual rows. Works with views, function, procedure, triggers, constraints, etc.

Create, Alter, and Drop, at multiple levels.

Create Schema *name*;

--GO is keyword to separate Batches of commands. Some commands need to be in their own batch.

Create Table *schemaName.tablename* (*column1Name c1type, column2Name c2type*,…);

ALTER allows adding or deleting columns (and more).

Alter Table *schemaName.tablename* Add *columnName columnType*;

Drop allows deleting table's rows and its definition. Ex: Drop Table *schemaName.tablename;*

Constraints – Can set constraints after column's type one after another (no commas).

Not Null: Value must be given. (Default allows null.)

Primary Key: Makes PK. Enforces uniqueness, implies Not Null (but being explicit is helpful), sets clustered index.

Unique: Self-explanatory.

Can set constraints after defining all columns, which allows specifying groups of columns. These should be named. Ex: Constraint *constraintname constraintType* (*columns*)

Check(*x*): Arbitrary condition x that must be true for any row.

Default(*x*): If no value given, sets value to this given default, x. (If constraint not given, is null).

Foreign Key: Makes FK. Constraint *constraintName* Foreign Key (*yourColumnNames*) References *TableName* (*otherColumnName*);

Identity: Cannot provide a value. Sets value based off the value in the previously created row. Can set the initial value and the amount it increments with Identity(*x, y*). Default is 1,1 (auto-incrementing). Does not allow repeating IDs of deleted rows.

--Adding column without a default is not allowed. One fix is allowing null, fixing existing rows, and then adding Not Null constraint

[Constraint *columnName\_NotNull* Check(c*olumnName* IS Not Null); ]

Alter Column *columnName type constraints(s)*; is to change the definition of a column.

Convert(*type, argument*) is for explicit type conversion.

**Thursday 2/28**

Can have computed columns, can cache the value on the row (persistent) or be recomputed whenever you need it (at run time). Ex: Alter Table *name* ADD *columnName* As (*computation*);

Create View *schemaName.viewName* AS *selectQuery*;

Views can be selected from, like a table. Ex: Select \* From *schemaName.viewName*;

Can add rows to views, which also adds to underlying table.

Ex: Select Into *schemaName.viewName* (*columnName,…*) Values (*values,…*);

Can run Insert, Update, and Delete on views, but only on columns that map to real columns in the base tables (not computed columns).

Variables

Declare with: Declare @*varName* AS *type*;

Set with: Set @*varName* = *value*;

Table Variable: Declare @*tableVarName* AS Table *col1Name type, col2Name type*,…)

Can insert into @*tableVar*, etc.

//No underlying table like with a view. Does not update automatically. Can change this table without affecting the base.

Functions

CREATE FUNCTION *schema*.*funcName*(@*param pType*)

Returns *rType*

AS

BEGIN

--example function

Declare @result int;

Select @result = Count(\*)

From Movie.Movie

Where YEAR(ReleaseDate) = @year;

Return @result;

END

--Functions can only read. not write!

--In SQL, string indexing starts at 1!

Ex: Substring(columnName, 1, 1) extracts one char starting at first character.

Multiplicity

0/1 to 1: 0/1's table has a Unique, Not Null FK to the 1 side.

1 to N: N's table has a Not Null FK to 1 side.

M to N: New table, with PK consisting of two FKs to the M and N tables.

Called junction table, or joint table.

Triggers

Create Trigger *schema.triggerName* On *schema.tableName*

*triggerType* *operationType*

AS

BEGIN

--Inside, two special variables, inserted and deleted (what is added, not necessarily inserted, and removed)

UPDATE

Set

//

END

Operation type: Can put triggers on insert, update or delete.

Trigger types: Can be Before, After, or Instead Of the operation. //For is before

Statement is once, Row is for each row inserted/affected.

Procedures are like functions, but they allow any sql command, they don't have to return anything, and must be called with Execute. Can contain while loops, if-else, and try catch.

Can only run one at a time.

-- Raiserror is correct spelling (there is no 2nd e, for some reason).

If Executing, make sure Declare is in same batch.

**ACID** –

**A**tomic transactions: perform all operations successfully, or none of them.

**C**onsistent: operations should not be allowed to violate constraints. (go from one valid state to another)

**I**solation: Even if multiple transactions run concurrently, end result should be as if each operated on their own, sequentially

**D**urable: Not complete until results are written to persistent storage.

**Isolation Levels –**

Read\_uncommitted: Allows reading data that has not been committed, called dirty read.

Read\_committed: Default. Allows other finished transactions to modify a row between one transaction's selects, called nonrepeatable read.

Repeatable\_read: Allows other finished transactions to insert rows that meet conditions one transaction is filtering on, called phantom read.

Serializable: Uses locks. Most isolation, lowest performance.

Begin Transaction //starts a transaction, a group of statements. Default Rollbacks to here

Statement(s)

Save Transaction *name* //Creates a named savepoint that you can rollback to

Statement(s)

Commit Transaction

End Transaction

Using Rollback transaction returns the state to before beginning the transaction (latest commit).

Using Rollback *name* rolls back to the named save.

A single statement is a transaction.

Indexes

Clustered index: Directly related to the physical layout of the table. By default, PK adds one.

Non-clustered index – Can have many. By default, Unique columns add these. Usually beneficial to create non-clustered index on each FK.

More indexes slows down write performance, because you have to update them all when you modify rows.

**ADO.NET**

Originally meant active data object, just ADO.

Now it is a namespace/brand name for all .NET data access stuff.

System.Data.SqlClient package dependency allows setting up SqlConnection objs, for interacting with an sql db.

Connection strings: List of key value pairs, used for giving server, database, login information (username, password), and other options. Don't include this in plaintext in the code.

Connected: Gets some data and works on it, keeping open connection. Faster for you.

Create SqlConnection(connectionString)

1. Open connection: connection.Open()

Create SqlCommand(commString, connection)

2. execute query: Create SqlDataReader reader = command.ExecuteReader()

**3. Process results (one at a time)**

**4. Close connection: connection.Close()**

vs Disconnected: Gets all data and closes connection. Slower for you.

1. Same

Create SqlDataAdapter //which internally uses reader

2. Execute query, save to DbSet

**3. Close connection**

**4. Process result set.**

Object-relational impedance mismatch: Difficulties in using relational data in DB for object-oriented data in application(s), especially in mapping.

Object-Relational Mapping (ORM): Converting database to objects and vice-versa.

.NET has **Entity Framework**

Java has Hibernate.

NHibernate Dapper

in .NET Framework: EF 6

**in .NET Core: EF Core 2.2** (missing lazy loading, many to many without X, class for junction table).

Co-generation and scaffolding: Computer generated obj for us.

Dataset is for ADO.NET.

DbSet is for EF Core.

database-first: Create database, turn it into code.

EF database-first approach steps:

1. have startup project, and data access library project.

2. reference data access from startup project.

3. add NuGet packages to the startup project:

- Microsoft.EntityFrameworkCore.Tools

- Microsoft.EntityFrameworkCore.SqlServer

and to the data access project:

- Microsoft.EntityFrameworkCore.SqlServer

4. open Package Manager Console in VS

( View -> Other Windows -> Package Manager Console

5. run command (your solution needs to be able to compile) (one-line):

Scaffold-DbContext "<your-connection-string>"

Microsoft.EntityFrameworkCore.SqlServer

-Project <name-of-data-project> -Force

(alternate 4/5. run in git bash/terminal:

dotnet ef dbcontext scaffold "<your-connection-string>"

Microsoft.EntityFrameworkCore.SqlServer

--project <name-of-data-project> -Force

6. delete the OnConfiguring override in the DbContext, to prevent

committing your connection string to git.

(7. any time we change the database (add a new column, etc.), go to step 4.)

code-first: Write objects and then migrate to DB. Steps covered later.

"fluent" Automated Procedural Instructions (API) because method returns an obj which allows more methods to be run. (Flowing method calls.)

EF changes default solution of referential integrity violation to Set Null.

By default, EF scaffolding will use fluent API. Best way to do it, since it has strongest separation of concern and more flexibility.

Could use -DataAnnotation option to put the configuration on the classes themselves with attributes. Not as flexible. Adds EF specific stuff to the classes (tighter coupling).

Third way is "convention-based".

Making contexts is cheap, so okay to make many and throw them away.

Need to explicitly call Add and Remove, but not update (tracks changes) for local context.

Need to run context.SaveChanges(); to push local changes to db.

read -> async Linq methods (for those that actually do things)

write -> SaveChanges async

**Friday 3/1**

Eager loading means that the related data is loaded from the database as part of the initial query.

Explicit loading means that the related data is explicitly loaded from the database later. Uses .Reference(*o =>o.property)* or .Collection(*o =>o.property*), depending on type of navigation property, with .Load() to actually load the data.

Lazy loading means that the related data is transparently loaded from the database when the navigation property is accessed.

Lazy loading: (is off by default). Will load related objects when needed. In practice, is too slow, due to running many small queries, as opposed to one big query with joins.

*context.table*.Include() lets you pass a lambda which says what fields you need from other tables, through foreign keys (will add an appropriate join to the query it runs).

Ex: context.a.Include(a => a.b).Include(a => a.c);

**/ b**

a

**\ c**

.ThenInclude() similar to include, but is for fields in the property you fetch in an include.(nested)

/ b **– d**

a

Would be context.a.Include(a => a.b).ThenInclude(b => b.d); //should show depth through tabs

Navigation properties tied to foreign key. Redundant data. Changing one won't affect other until running context.SaveChanges() (change one and then run SaveChanges)

Help with optimizing –

Azure and AWS have some data about queries being run.

MSSMS has Execution Plan (and live query statistics), which can give recommendations.

Non-clustered Indexes can help. (seek vs scan. Seek better search when possible.)

Restructuring queries.

UI Layer

Business Logic Layer

Data Access Layer

Lower layers should not depend on higher. (Kind of breaks dependency inversion principle.)

(UI still needs to create DbContext, or Irepo r = new Repo(); )

Domain Driven design is focusing on the business logic and everything else revolving around it.

If you can, test data access, but not as important.

EF uses unit testing in memory (vs sqlite).

**Tuesday 3/5**

old HTML -> HTML4 (exceptions to syntax) -> xHTML (strict subset of, ie. valid as, XML; error handling). Web devs/browsers Skipped past to HTML5!

WorldWideWeb Consortium (W3C). Strict standards.

WHATWG (Web Hypertext Application Technology Working Group): Less strict/freeform improvements.

Separation of concerns between HTML, CSS, and JavaScript:

HTML is text content, structure and semantics of page (meaning).

CSS: Appearance and layout (visuals).

JavaScript: Dynamic Client-side behavior.

Mozilla Developer Network better than W3Schools.

list of values in attr: space separated. eg. data = "1 2 3"

id: identifier for element, unique on page.

class: grouping of elements for css/js to target.

title: hover tooltip text.

Cascading Style Sheets (CSS)

Selectors-

\* \* all elements

tag div all with tag (all divs)

.class .login-form all with class (all class="login-form")

#id #logo-img the one with id (id="logo-img")

[attr=val] [src=logo.png] all with attribute set to value

[attr~=val] [class~=asdf] all with attribute containing val

Selector Combinators

, p, div All matching either selector (OR).

(concatenate) p.login All matching both selectors (AND) (all p elements w/class[.] login)

> p>div All matching 2nd selector that are children of some matching 1st Parent-child. (some div directly inside a paragraph)

(space) p div All matching 2nd selector that are descendant of some matching 1st

Ancestor-descendant. (some div somewhere inside a p)

+ p+p All matching 2nd selector that are siblings directly after some matching 1st. Next-sibling.

~ p~p All matching 2nd selector that are siblings with some matching 1st. Any sibling.

Pseudo-class – Select element in some pseudo-class (matching some state). Go at end of selector and start with colon.

:valid if correctly matches

:invalid if incorrectly matches

:link An element that has not yet been visited. It matches every unvisited<a>, <area>, or <link> element that has an href attribute.

:visited Links that the user has already visited.

:active An element (such as a button) that is "being activated" by the user. When using a mouse, "activation" typically starts when the user presses down the primary mouse button.

:focus An element (such as a form input) that has "received focus". It is generally triggered when the user clicks or taps on an element or selects it with the keyboard's "tab" key.

:hover What the mouse is hovering over.

:nth-child(aN+b) Ex: nth-child(2N) and nth-child(2N+1) every even, and odd child.

:first-child

:last-child

Pseudo-element - Use two colons, used to select parts of an element.

<selector> :: before

:: after Space after the element matching selector

:: selection What the user is highlighting

user-agent styles: Browser defaults.

Author styles: CSS included in documents.

-1: External. In head of html, <head> <link rel="stylesheet" href="main.css"> </head>

Best practice. Separation of concerns, reusability.

-2: Internal. In style of html, <style> p {font-weight: bold} </style>

-3: In-line style. In body element tags, <body> <p style="border: 1px solid black"> </p></body>

Worse practice.

The Cascade: When multiple rules set the same property of an element, which one wins.

1. Importance. After rule, put !important. Overrides others. Bad Practice.

2. Specificity. Inline wins (vs internal or external) More ids, if tie, more classes, if tie, more tags.

3. If still a tie, check source order. (later rule wins)

Box Model: Margin, Border, Padding, Content. Have property for top, right, bottom, and left of each, including shorthand. Can set

All are considered content for the element containing this element.

---

Start with introducing yourself, what the project is.

Mention Techs: C#, SQL Server hosted on Azure, connected with Entity Framework.

Then demo:

Display order hist of location and customer.

Show one sort.

One example of input validation.

Place order.

Show error.

Logging.

Preload database with data suitable for presentation.

Finish strong.

---

Display property – Determines what is inline or block using the following values:

inline

block

Inline-block (In between. Makes inner elements stay together.)

If it is seen or not in document with the following:

None: As if it didn't exist at all.

Visible property -

Visible (Can be seen.)

Hidden (Takes up space, but not visible.)

Position property -

Static (default): Elements render in order, as they appear in the document's flow.

Relative: Top, right, bottom, left. Offset to its static, normal position.

Absolute: Similar, but instead offset to its positioned (not static) ancestor.

Fixed: Similar, but instead offset to your view/screen. Doesn't change position as you scroll.

CSS Layout Tools

Flexbox: The amount of space something takes up, in terms of parts for each thing.

Grid Layout: Divide into grid.

Column Layout: Divide into columns. What bootstrap uses.

color: Foreground color.

background-color: Background color.

font-family: Can accept list of fonts, where it checks availability of fonts, in order, on user system. Can give entire family as last resort.

font-weight: boldface (how bold)

font-style: italic or not

font-size: Set font size as pixel length or percentage of parent's font size.

font-stretch: Some fonts can be narrow/condensed or wider/expanded.

Inheritance: Children can inherit some properties from parent element. (Which are which generally make sense.) It is customizable!

background-color, font-family, are inherited. All the box model is not.

CSS resets: Files that undo user-agent styles (browser default). Sets to 0. For consistency.

Normalize.css: Removes defaults but sets to another standard, "reasonable start point" (not zero).

Media query: if browser has some default (screen aspect ratio, height + width, if touchscreen or mouse, "print preview"), then apply some property.

Adaptive design. Write many css files, 1 file per resolution.

Responsive design: 1 css file, page auto-formats without refresh (as you change browser size). Css libraries help achieve this. (Bootstrap, Zurb foundation).

Bootstrap has 12-column grid. Generally, don't need more granularity than that. Give a bootstrap class to the element for where you want it located.

Many companies use some Bootstrap, but not all.

IIS Express (IISE): Internet Information Service (lite)

Can give link to server hosting bootstrap.css, instead of path to local directory.

Browser requests from server (w/IIS Express), which has static files (css, js) on disk that it sends back.

Html dynamically generated (from template) based on conditions and given to IISE. If request comes, server asks for dynamic file and sends it.

ASP = Active Server Pages. "web forms" = .aspx. Server side used to generate html response.

Special templating syntax inside html, replaced with actual html by aspx. Poor separation of concerns.

ASP.NET MVC: Framework for building web applications in .NET with strong separation of concerns, using the MVC design pattern.

Model-View-Controller -

Model: Object-oriented data sent to/received from the client.

View: Templates for html (layout, appearance). Display logic.

Controller: Part that receives the request from the client, marshals the models and view together and constructs the response. Gives view its model(s) and renders them into html.

Thus, View is dependent on Model, and Controller is dependent on both View and Model.

In startup, = is assigning default. (So default is Home controller and Index action.)

@RenderBody(): Views are embedded into layout here.

Some colored tags are razor syntax as well. Seems like html.

Razor expression introduced with @, and we type a C# expression. Compiler will try to figure out what is the code, but you can explicitly wrap it in parenthesis.

In <a > </a> tags,

asp-route="*MapRouteName*"

asp-route-*id*=""

In form, can give data to controller

<div>

<form asp-action="IndexWithUser"> (Needs to exist on HomeController)

<input >

</form>

</div>

[HttpPost] attribute shows that it receives POST not GET.

public IactionResult IndexWithUser(User user){return View("Index", user);}

or

public IactionResult IndexWithUser(IformCollection collection){return View("Index", new User{Username = collection["Username"]);}

Add reference to BusinessLogic.

Right click controllers -> Add -> Controllers

select with "read/write"

right click index

add view

HTML helpers look like razor C#.

tag helpers look like html.

More-in depth explanation later.

Project 3 –

Automated standardization

Infrastructure

DevOps

Policies on branches. Feature branches, team lead reviews, project lead reviews .

Code should match what has been agreed upon. Is it clean?

People certified about standard need to be involved in code review.

Sonar static analysis. Look at tests, security flaws, some practices.

In .cshtml, Razor comments disappear, html comments are just html, so stay in generated html.

Best practice to use ViewModels, so business logic doesn't care about I/O considerations in it, and so we can use attributes to rename the display of variables with proper names.

Silent error on name mismatch not correctly binding, just uses default value!

Giving [Required] tag (sometimes) prevents default from being used.

Don't rely on client-side validation; Need server-side validation!

Don't make repo in controller. New controllers are made all the time by the framework. Use dep injection.

ASP.NET allows dependency to be injected. Steps –

1. Register the service in Startup.ConfigureServices

2. Request the service (typically as ctor parameter)

Also need to add for dependencies in the service we're adding. (Recursively check and add.)

Add DbContext and your classes.

services.AddScoped<MovieRepository>(); //adds service for movie repository type, with scoped lifetime. Also, singleton lifetime: services.AddSingleton<Ilist<Movie>>(\_moviesDb);

Singleton lets us make the object ourselves, and all requests get the same object.

Scoped means one concrete object per request.

services.AddDbContext<MovieDbContext>(builder => builder.UseSqlServer(Configuration.GetConnectionString("MovieDB"))); //inside Configuration, inside Connection String, look for MovieDB value. Add it to user secrets (right click project). Spelling needs to be exact!

"ConnectionStrings" :

{

"MovieDB" : "<your-connection-string>"

}

(Could put it in app settings, but that's tracked by Git.)

Code First EF: Define C# classes and use tool (migrations) to generate SQL database.

1. Conventions: property named Id or *Class*Id will automatically be chosen for primary key.

2. We have fluent API – code in DBContext.OnModelCreating. Most flexible way. Allows using them as entities directly. Keep entity classes clean (of annotations).

3. DataAnnotations on entity class properties.

Steps for code first EF:

1. Have separate data access and class library project

2. Add NuGet package Microsoft.EntityFrameworkCore.SqlServer to both

and Microsoft.EntityFrameworkCore.Tools to Data Access

3. Implement your Context, inheriting from DbContext

-Constructor receiving DbContextOptions

-DbSets

-OnModelCreating

4. Register DbContext in Startup.ConfigServices using connection string from user secrets

5. Add initial migrations from Data Access Package Manager Console

6. Update Database from Package Manager Console

Can scaffold a DB, switch to a new DB, and Migrate the generated code.

Migration: A precisely defined commit of a whole DB schema. Allows rolling back or forward to different versions. (Has both Up and Down.)

In OnModelCreating:

//define what a Movie entity is:

modelBuilder.Entity<Movie>(builder => {

builder.HasKey(m => m.Id);

builder.Property(m => m.Id).UseSqlServerIdentityColumn();

builder.Property(m => m.Title)

.IsRequired() //column will be not null

.HasMaxLength(128); //column will be nvarcar(128)

builder.Property(m => m.DateReleased);

builder.HasOne(m => m.Genre) //under the hood, a "shadow property" will be made for foreign key

.WithMany(g => g.Movies); //Configure both directions of navigation prop

});

modelBuilder.Entity<Genre>(builder => {

builder.HasMany(g => g.Movies);

});

Register an interface: use two parameters, one for interface and one for implementation.

Because DbContext is scoped, and the repo uses it, it has to be scoped or lesser.

Review of Razor Markup –

Razor Expressions: Exs: @ModelName, @(new Data {Name = "A"}.Name

The result of the C# expression is converted to string, html-encoded, and inserted onto the page.

(Html encoding is to prevent inserting of actual code and improper characters.)  
Razor blocks: Ex: @{var data = 5;}

Runs code during view render. Doesn't necessarily insert anything into html.

Control statements: @if @while @foreach @switch

HTML inside these will be conditionally/repetitively inserted.

Directives: @model <*type-name*>

Makes view strongly-typed.  
Comments; @\* \*@ Text inside is removed at render time when converted to html.

HTML Helpers to know – Look like razor code, renders into html. Purpose is to binds model object to html.

DisplayFor

DisplayNameFor

EditorFor

TextBoxFor

ActionLink

DropDownListFor

Tag Helpers – Look like html, but isn't, renders into actual html. Has server side code help render html by targeting html tags.

a (anchor)

asp-action

asp-controller

asp-route

asp-route-*paramname*: Gives value for route parameter

input asp-for

select asp-for asp-items

label asp-for

(span) asp-validation-for

form asp-action

CSRF: Cross-Site Request-Forgery

Normally, client makes request, server responds with form, and client posts back.

A malicious party (cross-site) can create a counterfeit form which they get a client to fill out with some information and then send to server.

A solution: Server can create a unique token inside forms and reject response without them.

In ASP.NET, use [ValidateAntiForgeryToken]!

(begin-form doesn't auto-insert anti-forgery token, use AntiForgeryToken.)

DataAnnotations Attributes used by EF and ASP.NET for slightly different purposes. Validation checks and metadata.

Required, MaxLength, Range, RegularExpression, Display(Name),

DataType: Zipcode, EmailAddress, Phone, Password

Dependency Injection (DI)

in Startup.ConfigureServices

services.AddSingleton: One instance for entire app lifetime.

services.AddScoped & services.AddDbContext: One instance for each http request.

services.AddTransient: One per use (can get multiple inside one request.)

Can use interface and specific implementation when 2nd value references an implementation.

Jquery looks at attributes and input helpers to add client-side validation. (Model state still needs to be checked whenever non-server info comes in. Validate!)

Could make a class which inherits from ValidationAttribute to make your own constraints.

ModelState.AddModelError(property with problem, message) will send back the message to the property so user knows what the problem is and how to fix it!

Things disappear after requests (want server as stateless as possible, for performance).

Strongly-typed and weakly-typed views –

JS doesn't check types of variables at all. Model can make strongly typed view.

Ways to get data from controller to view –

1. Strongly typed view, the model. (Often view model.) View can only take one model! (So if you need several, you need a collection, or one view model that contains the several you need.)

2. ViewData: Key-value pair dictionary. reachable via a property on constructor. We can assign value in controller and access them in view. Accessed by ViewData["*name*"]. Weakly typed because compiler can't tell if you spelled key's name correctly, in order to actually get the correct value. (One of the things that goes away on redirects.)

3. ViewBag: Dynamic type (objs whose operations will be resolved at runtime). Allows adding new properties after an object has been created but turns off compile-time type checking!

4. TempData: A key-value pair dictionary. Values inside survive across requests! By default, stored using cookies sent to client, which are then sent back to server on following requests. If user declines using cookies (or they are configured incorrectly), TempData can break!

Can configure other providers for TempData in startup. (We can use this to build a model that needs many forms to be submitted.)

-In TempData, Regular key access deletes the values after the request. Can circumvent with Peek("key") which accesses value w/o marking it for deletion; and Keep("key") which unmarks for deletion (stays for next request). Need to typecast objects from TempData

The other way to build something (like an order) is with hidden form fields corresponding to view model properties.

1. Convention-based routing: Is a convention and a route; description of possible URLs

(Ex: … template: "controller=Home;action=Index" …) . Global routing.

2. Can use Attribute-based routing: Prevents using convention-based routing.

Controller gets route attribute(s) and action gets route attribute(s). Each can have multiple.

Can have route parameters in the route attribute, which you can then use inside the action.

Mostly used for identification, not options (that's more from query string).

In action's parameters, with attributes, can specify if FromQuery (query string in url, like ?key=value) or

FromForm/FromBody to get it from a form submission,

FromRoute to get from route parameter (defined in attribute on global route),

FromServices to ask for service like constructor parameters do.

Convert from view model to BLL when needed (view talks in view model, repo in BL)!

If need class from NuGet package, don't use VS recommendation which hardcodes filepath in csproj. Add the nuget package!

Testing: Have folder for matching each project (controller) and inside one for each large class (in it, a controller, moviecontroller).

Can right click on test, debug test!

IsAssignable tests null?

Can compare collections have equal elements with Assert.Equal and SequenceEqual (uses = = when we often want value equality for reference types).

Seems like a problem to need a repo for creating a movie controller, in order to test it.

Automated test should not alter actual app's database!

Could use InMemory or SQLite for a DbContext to provide a MovieDbRepo, but now far from "unit test," actually an "integration test."

1. Using interface, implement irepo with a fakerepo which implements the needed methods for the test. We assume fake works (simple logic).

Ex: EmptyViewForEmptyGenres()

-1. Check what method actually returns (ActionResult type that returns a ViewResult).

Ex: ViewResult viewResult = Assert.IsAssignableFrom<ActionResult>(resultVar);

-2. Check the ViewModel.

MovieViewModel viewModel = Assert.IsAssignableFrom<MovieViewModel>(viewResult.MovieViewModel?)

-3. Assert genres is empty

Assert.Empty(viewModel.Genres);

2. Mocking – Moq NuGet package. Better than fakes, but more learning curve.

Ex:

var genres = new List<Genre> { new Genre{Id = 1, Name ="Action"} };

var movieRepo = new Mock<ImovieRepository>();

Set up mock object: movieRepo.SetUp(r => r.AllGenres()).Returns(genres);

var sut = new MovieControllerRepo(mockRepo.Object);

var result = sut.Create();

var viewResult = Assert. IsAssignableFrom<ViewResult>(null);

var viewModel = Assert. IsAssignableFrom<ViewResult>(viewResult.Model);

Assert.Equal(genres.Count, viewModel.genres.Count); //check sizes are equal

for (var i = 0; i<genres.Count;i++){ //check each objects id and name are equal

Assert.Equal(genres[i].Id, viewModel.genres[i].Id);

Assert.Equal(genres[i].Name, viewModel.genres[i].Name); }

Can give parameters to the lambda we setup with It.IsAny and It.IsNotNull

Ex: *(obj.method*(It.IsNotNull<*Type*>() ) );

Can check lambda method throws Exceptions with (*obj.method*).Throws<*ExceptionType*>();

ASP.NET Core: Sequence of middleware plugged in order (using .Use*Name* in Startup.Config).

Request -> Other middleware (exception handling, static-files, etc.)

-> Routing middleware

-> MVC ->

(->And response is sent in reverse order.)

Controller/action selection (from routing)

Filters are applied to controllers/actions using attributes or global using startup!

-> Authorization filters -> (cached)

-> Resource filters

-> Exception filters

-> Model binding

-> Action filters (wrap action method execution: run right before action and after.)

Before is good place for model valid check on all actions, so put in startup.

-> Result filters (wrap result execution [view rendering]: run right before and after

Razor markup is executed/rendered into HTML.

Difficult to unit test action? completely. Can unit test ??

Can factor out common logic into Filters.

Know the kinds of filters, what they are for, and the general order!

Can name an IndexAsync method and it will know it is still just Index in routes and stuff.

IEnumerables over the network need to be async and called ToList() to make into list.

void methods that are asynch need to instead return a Task with no actual parameters.

Any method call that is async needs to have await before it.

MVC knows how to handle async action methods, so bubbling up can stop there. (Tests also need to be async.)

Remember to use GET, not POST, for URL query string.

asp-all-route-data Can use dictionary of string, string

Can use hidden data (Input) on form to keep a value (give and get back). Make sure to still validate. (Even if not visible, is editable client-side).

If .Any() on Enumerable (true if there is anything in table) then show table. Otherwise doesn't.

If copy template to another view, point it back to correct controller.

On an interface, can right-click and go to implementation.

Partial View:

A partial view is a Razor markup file (.cshtml) that renders HTML output within another markup file's rendered output. Let's us define something to be reused often (like the table of current objects in both edit and delete views) [Reduces duplication] and break larger files into smaller components (single file for page structure and partial view for each logical piece).

Can be a ViewResult. Doesn't run \_ViewStart.cshtml.

Reference a partial view with ether a partial tag helper (<partial name="\_PartialName" />) or a async HTML helper (@await Html.PartialAsync("\_PartialName")). Giving .cshtml makes it a path, defaults to same folder as markup file calling the view, can specify from app root with ~/*path* or /*path*, can do relative path with ../*path*.

Can be contained in any markup file (even other partial views, so long as there is not a circular reference).

Partial view has copy of parent's ViewData. Changes are not persisted to parent's (they are lost when the partial returns). Can also be passed a model.

At runtime, partials are rendered into parent's markup's output which is rendered into \_Layout.cshtml.

A partial cannot have an @Layout page reference.

not like a view which is only in layout. (Can only have one view at a time.) ?

GITHUB

Fetch other branch, then resolve merge conflicts. (Responsibility on person doing merge.)

Fork is new repo whose initial state is a copy of another repo's commits. Monitors changes to original, but your changes need a pull request.

Clone: Copy repo.

Pull Request (PR): From some branch on my github to another branch in another github (doesn't need to be the same branch). (On gitlab, called merge request?)

Pull request is a request to a repo to merge your branch into their target branch. Stays open and tracks (summarizes diffs) until the user with authority over the target branch merges them.

Allows tracking development of a feature with comments.

**Monday**?

Software Development Lifecycle:

Waterfall - Gather initial requirements, Design architecture, Build application, Test it, Deploy.

Cons: Takes time. Not adaptable.

Pros: Can have more security (separation). Accountability (set procedure shows who messed up).

Problem: Requirements change. Can't be 100% sure of design.

Big Bang Model – Do a little of everything at once. Suitable for prototypes (explore tech to see if it meets your needs).

Iterative Models –

Spiral: Mini-waterfalls. More responsive to client feedback.

Agile – More responsiveness.

Scrum Model: Break project up into features/small pieces, called user stories.

Run individual user stories through development.

User story: Some desired functionality (not specific to implementation). Bugfix. Often from top down (user point of view). All layers, but for one specific task.

Scrum board: Organize all user stories in the project and their stages of development.

Notes about implementation, design, and dependencies, as well as assigned workers and time.

Scrum master: Leads daily standup

Sprint: 2-3 weeks. Chunk of work. Starts with sprint planning. Formalize the user stories for the sprint.

Stakeholder: Anyone with a stake in the project. Talk to them continuously; show them as features are implemented to get feedback and change if necessary.

Daily Standup: Everyone present. Short talk per person = "What did you do yesterday/will you do today?" Identify blockers preventing you from doing your job.

Retrospective/Review: "How many stories did we get done? Did we over/underestimate work?"

Capacity: Productive work hours/points available (don't count lunch/time you are not working).

Story has estimated effort in points. (Round to Fibonacci number.) Decided during sprint.

Velocity: Sum of effort points completed during sprint. Gives feedback for next sprint's estimations to be more accurate.

Backlog: All stories uncompleted during sprint.

- Kanban: Continuous planning and reflection/reevaluation. Each column in board has a maximum number of points it can contain. Helps see bottlenecks coming before they happen. Attempts to limit max work in progress.

- CCMI: "Process-heavy" version of agile. Has levels (More security and responsibility).

For Agile, Client should be continuously providing feedback. Needs way for user to provide feedback and plug that into project board!

Framework-Dependent Deployment Mode: Requires the running environment to already have dotnet core.

Self-contained Deployment: Doesn't, but size is larger, in order to contain all that's needed.

Generally, checkmark the remove additional files option.

ConnectionStrings\_\_*Name* (Double underscore) is to match json to go one level deeper.

IIS always running but designed to use minimal resources when no requests. (Should be okay.)

When publishing to Azure, setup your connection string in App settings -> config -> Connection strings.

Human and computer readable ?

YAML pipeline vs Designer. Different ways of creating pipeline.

Fetch code

build code

run tests

deploy to azure app service

Azure DevOps contains a handful of services which contain a handful of ?

In pipeline's yml, the trigger is what branch to automatically run when it is updated.

pool is what virtual machines may run it.

Lists are denoted by dashing one line followed by a new line with a dash.

Project settings, pipelines -> service connections, type connection name.

1st, Copy into yml file (from <https://docs.microsoft.com/en-us/azure/devops/pipelines/targets/webapp?view=azure-devops&tabs=yaml#azure-app-service-deploy-task>) -> Deploy a Web Deploy package yaml:

- task: AzureRmWebAppDeployment@3

inputs:

azureSubscription: '<Azure service connection>'

WebAppName: '<Name of web app>'

Package: $(System.ArtifactsDirectory)/\*\*/\*.zip

where azure service connection is name of connection you entered above

and name of web app is name of app service hosted on azure (subdomain of website).

Add second task to publish: Task dotnet core cli ??

Publish web projects should be false.

removeAdditionalFilesFlag should be added and set to true.

Development (DEV): Developers work on source code. (This is where the (SDLC) Software Development LifeCycle takes place.) Send compiled code to IT.

Package Repo is for dlls, to keep compiled versions of working source code.

IT runs the working version on Dev Server and pushes fixes.

Dev Server is for testing/working-on data (not necessarily the real DB).

"Source of truth" is the agreed upon version of the code that the developers have to pull from, on the Dev Serv.

Test Server: IT pushes source code to Test Server so QA can test correct functionality on there.

Moves to Staging Server (prepare to deploy).

Moves to Production Server (real world deployment).

Operations (OPS): Want the servers to work, so resistant to change ("what if it breaks?")

DevOps: An extension of the Agile values to deployment, maintenance and production.

Using automation with parameters that are agreed upon by DevOps stakeholders, in order to deploy frequently and with quality, via automated checks.

"People Process Products"

Continuous Integration (CI):

Very frequently, each dev's code is integrated with the others, with automated checks (minimum: Code should build/compile. Unit tests should pass, sometimes more tests, like functionality. Static analysis). Version Control System (VCS)/Source Control Management (SCM), like GIT, is important here (commit often). Build pipeline can perform checks when we commit.

Deploy on every change.

Automated pipeline:

Fetch latest code from Git. Build dependencies (dotnet restore, if no changes should use cached). Build code (dotnet build). Run tests (dotnet test). Static analysis (sonar cloud). Package (dotnet publish). Deploy to Azure app services. Should notify of errors when something is not able to pass to next stage, so some developers can work on it.

Continuous Delivery:

Automated deployment all the way to production server with **some manual approvals** before production.

Continuous Deployment:

**100% automated** all the way to production.

Azure DevOps

"Path to solution" parameter default (\*\*\\*.sln) is fine for only one solution in your repo hierarchy, but otherwise you need to set it!

Can link parameters (easier to maintain. don't repeat yourself).

**Make sure to set up Azure connection parameters!**

Tasks -

Default path for test searches for files with test in the name.

Enable code coverage for VsTest!

Build outputs can be published as an Artifact, so it can get used in release.

(Nothing else can get out of build.) (Artifact examples: Published packaged dll, Diagrams ?)

"Batch changes" queues one build for all changes that have not yet been built in a timeframe.

If you commit multiple small changes quickly, wastes more time building repeatedly; If you want more detail later (on what specific change failed the build), turn on.

Triggers -

Can enable pull request validation. Allows checking before agreeing to merge it in.

Can schedule builds for specific times, as opposed to whenever change commit. (Still specifies which commits passed/failed.)

Retention - How long to keep logs and artifacts. Can specify time to keep everything, and how many successful builds to keep (regardless of time).

History – Who changed what and when.

Add extra level of wildcards (\*\*\) to test path (between buildconfig and test).

Change test to capital Test in testAssemblies.

Change test to your test name.

Build configuration variable, change release to capital Release.

SonarCloud – (Cloud hosted SonarQube)

Checks code to give helpful information through static analysis. (Method is too long, so separate it into smaller sections, etc.)

SonarQube

Scanner program scans code (of some given specificity) and sends it to a server.

Server applies rules and builds its analysis.

When running dotnet sonarscanner, make sure you're in folder of project.

Azure Boards is Agile board plus infrastructure.

Epics

Features

PBI Project Backlog Item Main one?

Task

Effort points for each.

Design plan and thoughts should be in Description.

Acceptance Criteria should state what exactly defines work being done. (Someone else can verify it is completed using this description).

Give priority

Assign effort points (Fibonacci)

Value area - architectural vs design (for business end).

Related work (link to dependent tasks …

After making, can comment on it.

Project manager moves new items into approved when happy with its information (good item).

In capacity, can determine lunch hours, etc. and assign to groups of activities.

Burndown: Shows work to do over time. Ideal is almost linear line.

Queries: Have to build a query in advance.

In tasks, Prepare Code Analysis before build. Run and Publish are after build and test.

Test coverage is Lines, blocks, methods, etc. of code which are touched by tests.

Should exclude tests themselves from the coverage!

Technical debt: Concept that sloppy, quick-code gets the job done, but needs to be paid for later.

Can ignore sonars default/create your own rules by creating a quality profile (for organization, not per project). Then set project to use this quality profile.

Different repo per part of the project (solution). One person can have control of overall project, with three inners having their own pipeline.

Release pipeline:

Can set pre- and post-conditions and gates (needed approvals, time limits, stage flows) starting from build's artifacts' post-conditions to actual deployment.

Setup Slack Notification for your pipeline

Diagrams -

Normal: One hardware, one OS on top of it, with apps on top of that.

Dual boot: One hardware, two OSs, with their own apps. (No virtualization.)

OS (Operating System) Level Virtualization: One hardware has one host OS, with apps and a hypervisor on top. The hypervisor can have guest OSs on top with their own apps on top.

Each guest OS + apps is a virtual machine. Can store its state as a disk image in memory (which stops using its active resources).

Hypervisor: Exposes virtual hardware interface (acts like HW for the OSs on top, so they can act with their default behavior, preventing need to change them at all). Hypervisor connects them to real hardware on host OS.

It is quick to copy VM's disk image and quick to get rid of them.

Container:

On top of the OS is the container engine. On top of that are containers (grouping of app and its dependencies underneath, like nuget packages and .net core runtime). Containers are isolated.

No change to the app. Container engine connects the app to the OS (cannot change the OS).

(Can combine virtualization and containers for cross platform work.

Both VMs and containers can be saved as images, files in the actual hw, when shut down. Containers are not generally saved when shut down. )

When starting virtualization, you must allocate fixed block of RAM. (Restart to increase.)

Containers flexibly use resources like memory, in the same way as regular processes.

Faster to change, which enables DevOps.

Docker -

Since we don't all have windows 10 professional, or Linux, we don't have support for real docker. Instead we use Docker Toolbox, which uses a small Linux VM to run Linux containers on Windows.

Docker technically uses client-server architecture. The client is Docker CLI, which uses REST API to connect to Docker Toolbox, which has the Docker Daemon as the server.

Run Docker QuickStart to perform one-time setup of the VM.

When you need to use docker, make sure it is running by the command docker-machine start

Docker needs admin privileges to build. (Can turn it off with docker-machine stop.)

docker run *name:* starts a container with the given name, downloading that image if not present.

docker pull *name*: just downloads/updates the named image

Docker image: Is the template for container to start. Has the whole filesystem and a pointer to some program to run when we start the container.

Every image is layered, built upon some other image.

Dockerfile in VS code. Install extension. Has instructions for creating a new image.

FROM *image*: Specifies what the base image of the Dockerfile is.

Specify registry, name/repository and tag. For example,

v v v

mcr.microsoft.com/dotnet/core/runtime:2.2

COPY folder1/ /folder2: Copies from folder in path relative to the Dockerfile (next to) to folder inside Docker image.

CMD ["command"]: Tells image the command to run on starting container.

docker run -it: it tells to attach running image to the current console window.

docker build . : Uses the Dockerfile in current directory to build an image. Gives hashcode to pass argument for run.

docker build -t *tagname* . : Gives built container the given tagname, so you can pass the tag to run command. (docker run *tagname*)

alpine is minimum Linux (doesn't include bash).

Can do building (compiling) inside Dockerfile.

Don't base image of runtime, instead mcr.microsoft.com/dotnet/core/sdk:2.2.

WORKDIR *filepath*: Sets current working directory, creating missing directories.

Use this instead of copy pasting your path everywhere.

RUN *command*: Have building the image run the command.

Can include dotnet publish in Dockerfile to get repeatable build environment.

CMD is an argument passed to the ENTRYPOINT.

Docker allows using some layers temporarily to copy some files out of them, so the final version is smaller.

Can name stages AS *stagename* to make multistage build.

Example: COPY --from=*stagename stagepath currentpath* to copy from previous stage to current

Can make a repeatable build environment and have smaller file size.

Running docker can have localhost ports inside docker that you can't access. Could map to port outside the container, but we also have to deal with virtual machine; access with docker machine's IP, not localhost.

docker container ls lets you see active docker containers. Inactive are not listed.

Docker containers you create remain on system until you remove them.

See nonrunning with: docker container ls -a

Remove with: docker rm *container-id*

Can also check images: docker image ls And with -a as well.

docker image prune: Will get rid of dangling images.

Running container takes control of current console.

Use -d in command to detach the container and run in background.

Also run --rm to automatically remove the container after it is finished running.

Can't really run debugger through container.

Can inject program to poke around while the container is running, printing some data, such as directory contents.

.dockerignore: List files to ignore (cannot be seen, so ignore changes on) when doing a docker build. Otherwise, any change requires a new build.

There is a tool to add all of a gitignore to a dockerignore (since if you're ignoring one you probably want to ignore the other).

EXPOSE *port*: tells whoever runs the container of this image to bind he port to something.

Doesn't actually do anything, basically documentation.

#If .csproj doesn't change (no new dependencies)

#then we can use the cache as below

COPY ./\*.csproj ./

RUN dotnet restore

#changing a .cs file will cause the following to run

COPY . ./

Common themes btwn containers and cloud: Separation from hardware. Fast setup and teardown.

The Cloud:

On-premises service/resource: Physically located at the location.

Cloud: Hosted "on the cloud," some server accessed through a nw, not your own server.

Hybrid Cloud: Some on-premises, some cloud.

Multi-Cloud: Using more than one cloud provider.

Cloud provider/platforms: Amazon Web Services (AWS), Microsoft Azure, Google Cloud, etc.

SLA – Service Level Agreement. "How often will it work?" Expected keeps getting higher.

Uptime vs downtime (not working).

Region: General physical area that the server is. Allows faster connection time if closer.

Availability Zone: Stuff that should be separated such that failure in one (earthquake, power outage, etc.) shouldn't affect the other zones.

Software as a Service SaaS: Abstract away everything (hw, nw, language, etc) but the app itself.

Ex: Webapps, gmail, outlook, github.

Infrastructure as a Service IaaS: Abstract away many physical concerns (location/power, etc).

Ex: VMs, Storage (disk, file, Azure blob/ Amazon S3).

Disk storage: Connect to one vm at a time.

File storage: Connect to many at once, like a fileshare.

Blob storage: Unstructured (no directory/filesystem). Better for large files and streamed + static.

Platform as a Service PaaS: Abstract away operating system, disk, network, hardware. Provider makes sure everything is updated. Just plug in your code!

Ex: Azure Pipelines, Amazon CodeBuild, Amazon CodePipeline.

Container as a Service CaaS: AKS, Azure Kubernetes Service.

DBaaS: Azure SQL.

Public vs Private: Server is not dedicated to just you, or it is.

Government Cloud: Government security contracts. NW never leaves government's border.

CI tools: AzurePipelines, Jenkins, CircleCI, TravisCI, Appveyor, AWS CodePipeline/CodeBuild, GoCD (first to promote continuous deployment)

--

Project 2 general requirements:

Web app with DB for persistent data.

Data Model: At least two many-to-many relationships

User interactions: More than just do a couple buttons. (View certain things, search, place order with multiple things.)

Authorization: At minimum, at least one with different privileges (like an admin that can do everything).

[CI-pipeline deployed to app service.]

Got to be MVC, EF, some SQL, plus anything you want to learn yourself.

Need team, in one paragraph what the project is and what the user can do. User stories. Multiplicities in the data model.

Minimum Viable Product (MVP): If you have an overall ambitious project, have a part to focus on to have an acceptable project.

**Tuesday 3/19**

Service-Oriented Architecture (SOA): Next level for separation of concerns / loose coupling.

Split application into web services. Instead of linking our layers at build time as dlls, they will communicate at runtime over the web.

Advantages: More granularity also allows independent replication. (Ex: Don't need whole copy of backend in order to access db. Can have 100 API and 5 MVC apps.)

Client lacks information on your implementation (security).

Client is no longer just browser. Could be MVC or anything else that accesses the web service.

API and Web API are often synonymously used with web service. (API generally means communication standard)

Expose: (Service exposes itself.) Many ways. We will cover SOAP and REST (for project).

Consume: Read/write/access the service.

(WCF) Windows Communication Foundation: A framework for building service-oriented applications. Some stuff ported to .net core but not most of it. No hosting but consuming (reading).

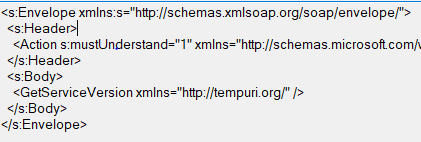
---

WCF has test client. Running Service starts this instead of what it claims (IIS on Google Chrome). Has format tab for giving input + seeing output visually, and xml tab for seeing actual xml request and response.

---

SOAP: SOAP is a protocol. Communicates using XML over a variety of transport protocols (not just http). Rigorously define all needed information about service in a Web Services Description/Definition Language (WSDL) document.

Every soap message is in XML. Has a required envelope. Has a header (optional). Has a required body. Then the closes for all of those.



XML supports XML Schema which adds structure to xml, including data types (can create your own). Also Supports namespaces. (xmlns: *name*)

WSDL –

Definitions: contain the definition of service(s). Optional name, targetNamespace for logical namespace of the document, xmlns:soap specifies soap specific info.

types: Information about complex (not simple) data types used in the doc.

message: Abstract definition of data being communicated.

operation: An abstract description of the action supported by the service.

\*portType: Define abstract set operations supported at endpoint(s). Each element is an operation (Default is an input and an output.) Like an Interface.

binding: Concrete protocol and data formats of the operations and messages.

port: Single endpoint address of the binding.

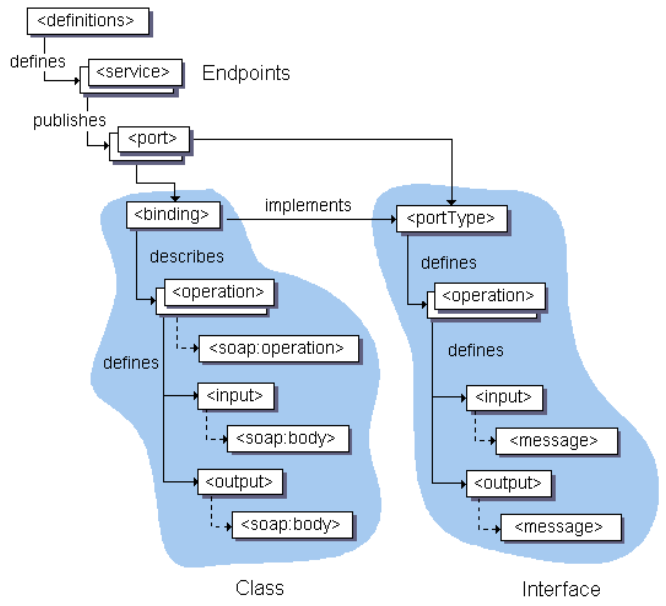
\*service: Port address(es) of the binding. (Collection of network endpoints or ports).

WCF – The 'abc's:

address: URL (where is the service).

binding: Underlying protocol. (Language being used).

contract: what are all operations/messages you support.



Make sure to update client through connected service when you update server code. (Knows through wsdl.)

Don't confuse SOAP with wcf.

Transaction scope used so that wcf rollbacks according to ACID.

SOAP is based on xml. Can waste time on structure that is not necessary! Json can take up less space for same information (faster over network!, plus faster to parse).

REST often used for client, but server to server often still uses SOAP.

HTTP Request has Method and other headers. Has Body for some methods (like POST).

200 OK means the request was processed without throwing an error.

Can change POST to return an IActionResult, specifically 201 Created with information.

---

API Controller template includes needed routing by default.

Don't use tightly coupled entity framework template!

Use API controllers, not MVC controllers.

From Visual Studio, can make a consumer for API, instead of using Postman.

**Wednesday 3/20**

SOAP message goes in POST request in the body.

RPC "webservice.com/methodname?param1=23".

Both kind of ignores what http and the web want us to do.

HTTP has semantics (GET for read, POST for write) and traditional web services like SOAP ignore them.

Richardson Maturity Model (RMM) – Idea of steps towards REST.

Lvl 0: Pure HTTP to one service endpoint.

Lvl 1 - Resources: Split further actions into accessing separate resources.

Lvl 2 – HTTP Verbs: Split GET (safe operation that can take advantage of caching) from other non-safe verbs, like POST and PUT. (Handle similar situations the same way.)

Lvl 3 – Hypermedia Controls: Server responds with Hypermedia controls for further actions. Self-documenting. Allows client to just use response, so server can make changes without breaking clients (who normally work on assumptions).

REpresentational State Transfer (REST): Is an architecture. Communicates using many representations (often JSON) over HTTP only. Info of service either 1. not rigorously defined (not really REST). 2. hypermedia / HATEOAS (110% REST, no one really does it). 3. one of many API description language (like WSDL for SOAP) e.g. OpenAPI/Swagger, API Blueprint.

Guiding Principles of REST –

1. Uniform Interface: Use http semantically in headers, methods, status codes. Identify resources and have 1-1 relationship between URLs and resources. Distinguish resource from its representations.

2. Client-Server Architecture: Use client-server to separate user interface from data storage, which improves UI portability and server scalability. Client only needs to know about server's exposed functionality.

3. Statelessness: Server doesn't need to have state info for client requests. (Can save to db, but server itself shouldn't save info. Causes scaling problems.) Client needs to send all data needed.

Ex: Partial orders. Server sends data to client and gives it responsibility to send the data back.

4. Cache-ability: Server (and client, depends) can cache some (labeled) responses. Decreases processing time.

5. Layered System: Architecture composed of hierarchical layers constraining component behavior such that each cannot see beyond the layer they are interacting with. (Stuff changes in backend and client doesn't know the difference)

6. Code-On-Demand: Theoretical/Optional (not generally done in practice). Client asks server for code to read/understand a response that it got.

Layered Architecture: Payoff in a while, when requirements change

Service Oriented Architecture: Payoff takes even longer, like when a framework no longer used.

Hypermedia: payoff so long that business doesn't even want to do it.

HTTP Verbs/Methods –

GET: is for retrieval of resource.

POST: is for creation of resource.

PUT: is to replace resource.

DELETE: is to delete resource.

PATCH: is for partial update of resource.

----

HEAD: is like GET but just for headers, not body.

OPTIONS: gets Allow header, which tells what methods you can use on this resource.

TRACE: is to get exact response you sent back, basically for debugging.

Constraints for HTTP –

Safe: A safe method should not change the system's state. Enables cache-ability. Includes GET and HEAD and OPTIONS.

Idempotent: Multiple copies of the same (identical) request should have the same effect on the system state as only one copy of the request. (Response can be different.) Automatically includes Safe methods. Includes PUT and DELETE. Ex: Set-on and Set-off methods, as opposed to a Toggle. (Forms use POST which isn't, so why browser asks for confirmation to resend.)

HTTP Status Codes -

100 to 199: Informational Category.

200 to 299: Success.

300 to 399: Redirection.

400 to 499: Client Error (ex: badly formatted request).

500 to 599: Server Error.

100: Continue. If request is being sent in multiple pieces and the server is ready for the next part.

\*200: OK. Generic success msg. Particularly for GET response.

\*201: Created. Success for POST response. Should return representation of the created resource (in the body) and the resource's location (in the header).

\*204: No Content. Success, with no response body. For PUT (and PATCH)

301: Moved Permanently. Includes where the new location is.

\*302: Found. Moved, but not necessarily permanently. Commonly, browser redirects w/get to new location automatically.

304: Not Modified. Response to If-Modified-Since request header (for client-side caching). Resource has not changed (since the given time).

\*400: Bad Request. Generic client error.

\*401: Unauthorized. Really for unauthenticated.

\*403: Forbidden. May be authenticated, but not authorized to access.

\*404: Not Found. Resource does not exist.

405: Method Not Allowed. Server does not implement method on this resource.

\*406: Not Acceptable. (Negotiation thru headers. Accept, on Requests, can give options for format.) Server cannot provide any of the requested formats.

\*500: Internal Server Error. Generic server error.

501: Not Implemented. Yes, that's a method, but the server hasn't implemented it yet.

\*502: Bad Gateway. This server didn't have problem, but some other server it was talking to did.

Joke codes - 402: Payment Required. 418: I am a Teapot.

Authentication: Prove who you are (identity). Authorization: Prove what you can do. (privileges)

Theoretically could use just 200 OK, but proper REST should take advantage of correct HTTP success codes. (Similarly, for other code types as well. )

Resource: Any info that can be named (doc, image, temporal service, collection of other resources, non-virtual/real world obj, etc.). Has an identifier.

Resource Representation: The state of a resource at any time. Includes the data itself, its metadata, and hypermedia links for client to transition the resource to other states.

Media Type: The data format of a representation, which shows how the representation should be processed. (Truly RESTful API looks like hypertext; every unit of info has an address.)

Uniform Resource Identifier (URI)

Headers –

Content-Type: Is on both requests and responses. Tells what type of data is in the body, so the recipient knows how to interpret the binary data.

Exs: application/json , application/xml , text/plain , image/png , etc.

Don't use separate urls for different versions. Instead use accept header and content-type header.

(Accept header is what type of formats are valid in response).

---

Whenever an action method returns anything that is not an IActionResult, it is automatically wrapped in a 200 OK response. Not good in real life. Should be IActionResult or subtype, or ActionResult<*bodyDataType*>

Swashbuckle package: swashbuckle.AspNetCore

In try block, access the database, in catch, return some server error.

---

Cookies are key-value pairs.

Cookie Header in request is for the client to send all non-expired cookies it has for the domain.

Response has Set Cookie header which sends a cookie to a client that lasts until the given expiration.

--

[Authorize] data annotation (w/o any more qualifications) means any logged in user (authenticated) can use it.

More specific (action) overrides less specific (controller).

---

Add controller just for authentication (empty, named account controller). Create constructor.

Delete cookie by setting it to empty string.

Account controller needs Login page, Logout (HTTP post … or get), and Register.

Can use Models folder for viewModels and make an APImodels folder. Separate viewmodel for annotations is good idea.

Generate View Create template for login model. Change "Create" to "Login".

For login, check model state (return info). Else [Add cookie to Response to browser.

If browser sends cookie, needs to copy them onto http client to send to api. ] **Nick said to copy his code!**

AServiceController should inherit from Controller. Receive an http client from dependency injection, and URL and cookiename.

Add Json for convert.

public HttpRequestMessage CreateRequestToService (HttpMethod method, string relativeUrl, object body = null)

Will do all the work for the controller of creating the request to the API, but not sending it.

AccountController needs to send HttpClient and IConfiguration to parent AServiceController, do so using constructor. Calls the above request method and sends to httpclient. Add any errors to model state and return. Otherwise, try to pass cookies to client (by using another of nick's methods), if fail, respond with error. Then, you've succeeded, so redirect appropriately.

Similar for other controllers, with appropriate httpmethods.

--

Add HttpClient to Startup as singleton (otherwise memory leak), for needed dependency injection.

Appsetting should have non-secret information, such as correct serviceurl. Overriding for development environment or not (production).

---

Claims: General purpose auth. Key-value pair. (has username which is this, etc.) Authorization evaluates claims against something.

Roles: What we're using. Roles has a many-to-many relationship with users. Can check user's role for authorization.

---

Add RoleManager<IdentityRole> to controller.

Make sure role exists with

if(!await RoleManager.RoleExistsAsync("*name*")) {var role = new IdentityRole("admin"); var result2 = await RoleManager.CreateAsync(role); if(!result2){return StatusCode(StatusCodes.Status500InternalServerError, "failed to create role"); }}

in the register method (partial fail. user registered, but desired role not created)

On API, can authorize with specific roles. [Authorize(Roles = "name")].

Need to change cookie configuration to change OnRedirectToAccessDenied to return Status403Forbidden.

If we don't want to use Authorized data annotation, we have User.Identity.IsAuthenticated; ,

User.IsInRole("name"); , User.Identity.Name;

In MVC, class for accountdetails: bool IsAuthenticated, string username, IEnumerable<string> roles. and endpoint. Can apply global filter to every action to get details.

IsAuthenticated = User.Identity.IsAuthenticated, username = User.Identity.Name, roles = Roles.GetRolesForUser();

Add class GetAccountDetailsFilter : IAsyncActionFilter.

Inside can have async method (pre-action filter) with the action context and delegate:

public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next){}

Add the filter to Startup:

---

View can hide options client-side based on role.

Server still needs to verify though (hidden fields are still accessible by client).

--

JavaScript

ECMA, with cooperation from all major vendors, has standardized JavaScript, called ECMAScript. Version 5 (ES5) is the modern baseline, what all decent browsers support.

ES6 (aka ES2015), then ES2016 (ES7). The Standard is ahead of browsers!

Web developers use the subset of ES6 that all browsers support.

JavaScript is dynamically typed. Variables are not locked to a given type. JS has types (Actual objects have types).

JS can use browser console with console.log();

Declare a variable with let. Ex: let x;

Can assign a variable as undefined. Has "undefined" type and value. Ex: x = undefined;

Number: A 64-bit IEEE floating-point number. Has whole numbers, decimals, negatives, and Infinity. and -Infinity. (Dividing by zero gives infinity.) NaN, which is Not a Number. (Instead of giving error, will often get value NaN.) Is infectious (further operations will get NaN)!

String: In either double or single quotes. (Can use the other version of quote as a literal inside of it.) Has backslash as escape character.

Boolean: Has Booleans. x=true; x=false; x = (3==3); etc. All comparison operators the same.

Object: Declare object that is not based on class using braces. Ex: x = {}; //object literal

Can assign an object. Ex: x = console;

Can give properties in declaration. Ex: x = {"name": "Nick", age: 26}; //note that it doesn't need the quotes, used to read exactly, and thus add whitespace.

Can add properties to object after its declaration. Ex: x.newproperty = 123;

Accessing property that doesn't exist does not throw error, just adds it!

Arrays are just objects.

Null: Is a type. Has value null. typeof null is object! Null used to indicate absence of a value.

Function: Similar to C# delegate/lambda. Are basically type object, but typeof says differently.

Ex: x = function(x) {return x+2;}

ES6 added a new type called symbol.

7 types: String, number, boolean, null, undefined, object, and symbol.

Can declare function, but don't declare parameter or return type. No return type = undefined.

Ex: function printName(name) { console.log(name); }

Function Expression to do the same thing.

Ex: let printName2 = function (name) {console.log(name); };

ES6 has lambda functions, called Arrow Functions. Can do same thing.

Ex: let printName3 = name => {console.log(name); };

Control statements:

Has if, else if, else. Has for loops. Has while and do-while loops. Has switch statements.

Has operators (==, !=, <=, &&, || ).

Has debugger breakpoint statement. (Opens up debugger in sources tab in browser.)

1. Extra arguments are silently discarded!

2. Not enough arguments => Unprovided arguments become undefined!

**Friday 22**

'use strict'; Is opt-in to strict mode, which fixes some bad historical behavior. (didn't want to break backwards compatibility, so made opt-in)

Undeclared variables became ReferenceError. Some old silent errors became thrown errors.

Before ES6, we had two scopes, global and function scope. (Doesn't have block scope, so can use variables outside their declared block, {not just inside these brackets}. )

In ES6, adds let. Let has block scope!

(var has "hoisting" behavior: declarations are effectively moved to top, so it isn't a reference error to use the variable before it has been declared, just has undefined from not being assigned)

Also added const. Similar to let but cannot change value after assigning, Throws TypeError!

Has try and catch and throwing errors.

Has for of (instead of foreach loop).

JavaScript is interpreted. Runs the source code line by line directly. (Able to get new lines of code while running and still handle it. ) …Can do just in time compilation of some kind ( for optimization). Can write ES6, ES2016, ES2018, etc. and compile down to ES5; Babel does this.

String Interpolation (Template literals): ` around the whole thing and ${*variable*}.

` allows multiple lines as the same string.

Loose Equality vs Strict Equality, == vs === (2 vs 3 equal signs) -

Number 1 compared to string '1' is true. True compared to '1' is true. [1] == 1 is also true.

This is due to == using Type Coercion (comparing value without type).

To do both, use ===. Values of different types will never be equal with triple equals.

Note 0 === -0 is true.

(Always use 3 equals!)

Must compare against NaN using function isNaN(x).

Reduce Function: Can use as functional way to sum array.

Type coercion to Boolean happens in if condition. Truthy evaluates to true, falsy to false.

Should know these!

Falsy: undefined, null, 0 and -0, NaN, empty string, false.

Truthy: Everything else.

JS has prototypal inheritance, not class-based inheritance. Object inherits from other concrete objects, called its prototype.

In JS, "this" is special. When you call a function, this is set to the object the function was called on (left of the .*funcName*). In strict, it will throw an error if there is no obj.

Ex: Assuming fname uses this. var func = obj.fName; func();

Exception to that is ES6 arrow functions. "this" is set when the function is written and it does not change. Works like "this" in c#. (Thus, don't use arrow functions as methods.)

We can make new objects from a template constructor function.

Ex: function Student(name) {this.name = name; this.sysName = function: () {console.log(name);} } var student = new Student('Nick');

Inheritance: function StudentWithBirthday(name, birthday) { this.\_\_proto\_\_ = new Student(age);

this.birthday = birthday; this.checkBirthday = function (date) {if(date===birthday) { console.log(true); } } }

Every object has a base prototype object with some methods, like toString(), etc.

In JS, property access will look at the prototype object if it is not found on the current object.

In ES6, classes were added as "syntactic sugar" around prototypal inheritance. (Just a rewording. Interpreter still treats it the same as the old.)

class Student { constructor (name) { this.name = name;} sayName() {console.log(name);} }

class StudentBirthday extends Student{ constructor(name, birthday) {super(name); this.birthday = birthday;} }

Callback Functions: Similar to delegate in C#. We can write code that accepts other code as parameter to 1. provide extensibility to code (polymorphism). 2. For asynchronous stuff.

Nested callbacks are a common occurrence in JS.

Common to register an event handler: give func to run when event happens.

JS has closure. JS functions "close over" their environment. Nested functions keep variables around them (outside function, but in its block), that they reference, alive for as long as the function itself is alive.

(In JS, script contents are concatenated, so we try to avoid putting things in global scope.)

Immediately Invoked Function Expression (IIFE): This will hide our variables from global scope and still run immediately. Ex: (function () {} )()

Can have functions inside, but only return one function (almost like private and public methods).

ES6 has modules. A file can be a module, which has its own global scope, and only what is explicitly exported, and then explicitly imported by other files, can be seen in those files.

ES6 Features (don't need to memorize?)

Block scope, with let and const.

arrow functions

method syntax

default parameters

string interpolation

classes and extends (inheritance)

many built-in functions (like searching a string)

ES6 Modules:

Set and Map objects, and methods

for of loop (like foreach loop)

get/set properties

internationalization ?

spread: Split iterable thing into its components (elements of an array, characters of a string, etc.)

destructuring: split object into its individual variables. Ex: var {ops, left, right} = Node();

Promises (for async stuff)

**Monday 3/25**

Document Object Model (DOM): HTML is the serialized format of the DOM.

(Elements tab reserializes the current state of the DOM.)

Want to register event handler for right when the page starts.

OnEvent properties run when the event happens.

Load is when object is completely finished loading.

Thus, normal log in html will happen first, then when done loading, the onload func will be called, logging that second (even though it was higher in html itself).

use window.addEventListener('load', () => *func* ) to register multiple at once.

But load is also after all images and scripts, etc. are downloaded. Usually, you can use document.addEventListener('DOMContentLoaded', () => *func* ) to run earlier instead.

Could add to other things as well. (Example, Event handler inside event handler => callbacks).

document.addEventListener('DOMContentLoaded', () => button.addEventListener('click', () => {count++; textPara.innerHtml = $"{count}");}

After loading, add a button that increments whenever it is clicked.

Instead of adding event handler to a button click, could use the submit of a form.

location.href ="*url*" to navigate to new webpage.

document.querySelector('name') will get the first of the given element in the document, and .querySelectorAll('name') will get all.

type is the name of the event type (click, mouseover, etc.)

target is the element the event was fired on.

currentTarget the element whose handler is running right now.

"this" is set to currentTarget for an event handler.

.stopPropagation(); will stop the event from propagating out any further, which is bad practice!

Browsers support both default "bubbling mode", which is innermost nested element to outer, and "capturing mode", which is outermost to inner (old opposite order), used by setting 3rd parameter of .addEventListener equal to true.

Order of event handlers: First, we go down in capturing phase, second, we run everything on target in target phase, and then, we go back up in bubbling phase.

Common browser behavior is just default event handlers. Can use .preventDefault()

to prevent redirecting when clicking on an <a> (link) element

"Walking the DOM" = traversing the DOM.

Root node is the document

document.documentElement is its child, the <html> tag.

document.head is its child, the <head> tag.

document.body is also html's child/head's sibling, the <body> tag. Can be null.

Nodes have parentNode, previous- and nextSibling, as well as first- and lastChild. All children are in the .childNodes collection.

Text element of a tag is the tag's child.

Descendants are nested children of the node's (direct) children.

JSON gives two functions: JSON.parse(*obj*), to deserialize, and JSON.stringify(*obj*), to serialize.

AJAX (**A**synchronous **J**S **A**nd **X**ML):

A set of tools/techniques to send HTTP requests from JS and process the results w/o browser reloading the page.

Traditional tool is XMLHttpRequest (xhr) - (Kind of cumbersome; lots of copy-paste code)

readyState: The state the XMLHttpRequest client is in.

Value State Description

0 UNSENT Client has been created, but open() not called yet.

1 OPENED open() has been called.

2 HEADERS\_RECEIVED send() has been called, and headers and status are available.

3 LOADING Downloading; responseText holds partial data.

4 DONE The operation is complete.

status: The HTTP status code.

responseText: The server's response data, as a JS string.

Fetch API - Instead, could use the Fetch API. Doesn't add anything new but uses promises.

Promise: An object which represents some value which we will eventually get or fail to get. Can resolve to success or failure (w/error for why).

Uses the two methods .then (event when promise is resolved) and .catch (handle any error along the way). Can give parameters to handle specific errors.

Can chain .then directly w/o needing to indent (like with event handlers /). Adds readability.

--

Node.js – A JS runtime environment.

Node package manager, NPM: Kind of like nuget for node packages, but more sophisticated.

NPM is the command that we run to install dependencies.

NPM init will create a new project, a package.json file. The name is if your gonna put your code out on npm.

--

Compile / transpile one version of ES to another (ES2017 -> ES5), is done by transpiler, often Babel. (Not really a change in abstraction like normal compile, more like moving sideways.)

---

Typescript (TS) –

JS plus (opt-in) strict compile time type-checking.

Can be transpiled (tsc) into ES5.

Everything in JS is in typescript (it is a superset of JS).

By default, everything is in global scope. However, can export and import to turn into module similar to ES modules.

Uses *name* : *type*

Combine js files using webpack (bundles/ module loader) into less (not necessarily one) bundle js file(s). Minimizes file size (for better nw performance) by removing whitespace and shortening names.

package-lock file is to not use the latest versions, but the deterministic ones we know about.

npm install is equivalent to dotnet restore.

Generated code for Modules is happening.

Type checking is only compile-time.

---

Angular -

AngularJS is a library for JS.

Angular is a Framework for typescript.

ng new *name*

**Tuesday 3/26**

A (ng) module is a chunk of program that can be downloaded.

An ng module is published in a package. Ng module knows about node stuff.

Import class (component, service, etc.) from module, can export its own classes for others to see.

App has a root module, which is bootstrapped, with a root component.

A ng module is a (TS) class with an (Angular, not TS!) decorator, @NgModule.

Decorators provide metadata that Angular needs to define the class as an Angular module. Every component needs to be declared in a module.

Its imports array lists the other modules we want components/directives from.

To access something from 3rd party npm package:

1. npm install *package*

2. (TS) Import into module file that needs it

3. Put into (Ng) imports array.

Similarly, a Component is a class with a decorator that turns it into a component, @Component.

Selector kind of defines the tag you should use in html, for elements that will be replaced by this component.

templateUrl references an html template file to define a view, a physical area on the page with some logic and its own lifetime. (Often just call the view a component.) Can define the template inline, but generally don't (bloats file), unless it is very small.

styleUrls references CSS style file(s). Could also have css inline, but don't.

Angular default uses Jasmine framework instead of xUnit for testing.

All of a component's imports are referenceable in the component.html template.

Can use "interpolation" in it. The value of a JS expression in double brackets, {{ }}, will be converted to string and HTML-encoded. Is dynamic, since any change to the value will cause the view to be updated.

ng generate component *name*

Command adds component to module's declaration array, and it generates some default files: *name*.component.ts / .html / .css / .spec.ts

app-routing.module.ts' routes array should contain the routes (you want to be known). Route is path and component.

In component.html, routerLink comes from Router module.

Two-Way Binding:

ngModule comes from FormsModule, so import from @angular/forms and add to app.module.ts' imports array. Enables two-way binding, which is communication (auto update) between DOM and Component and vice-versa.

With Angular, put validation attributes in the html code.

Event Binding: (*event*)="*handler*" uses DOM events to call a handler in the component. Could be an arbitrary JS expression, called template statement.

Any logic not tied to this particular view (could be used by other component), should be wrapped into service and injected.

@Injectable decorator defines scope the service is reachable. Default providedIn: 'root' allows anything in app.

HttpClient is an object in Angular. Make sure to import from @angular/common/http (not anywhere else!), in both service.ts and app.module.ts and add to its imports array.

Default response type for angular HttpClient.get(*url*); is json.

Observable is similar to promise, but instead of just one subscriber with one .then, allows any number of subscribers. Is asynchronous. Comes from rxJS.

In TS, can put access modifier on constructor parameter to automatically make it a property of the class.

Typescript has generics. (with angular brackets, <>)

Can give a type parameter to deserialize into specific type of observable.

Structural Directives:

\*ngIf and \*ngFor are equivalent to Razor's @if and @foreach.

If condition is true, then the element will appear (immediately) in the DOM, if false, it won't.

Ex: <ul \*ngIf="pokemon && pokemon.length > 0" >

For will repeat element according to how many things are in the thing after the of.

Ex: <li \*ngFor="let p of pokemon" > {{p.Name} </li>

---

In subscribe, can have error ??

API can send back validation error msg. Angular could display that information directly.

Angular should have RESTful handling (assumptions of errors API could give, so display appropriate error).

---

Angular tutorial:

You develop apps in the context of an Angular workspace.

A workspace contains the files for one or more projects.

A project is the set of files that comprise an app, a library, or end-to-end (e2e) tests.

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

You'll find the implementation of the shell AppComponent distributed over three files:

app.component.ts— the component class code, written in TypeScript.

app.component.html— the component template, written in HTML.

app.component.css— the component's private CSS styles.

The double curly braces are Angular's interpolation binding syntax.

You always import the Component symbol from the Angular core library ('@angular/core') and annotate the component class with @Component.

@Component is a decorator function that specifies the Angular metadata for the component.

The CLI generated three metadata properties:

selector— the component's CSS element selector

templateUrl— the location of the component's template file.

styleUrls— the location of the component's private CSS styles.

The ngOnInit is a lifecycle hook. Angular calls ngOnInit shortly after creating a component. It's a good place to put initialization logic.

[You should avoid complex constructor logic. Don't fetch data in a component constructor (You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it.) Constructors should do no more than set initial local variables to simple values. Instead, ngOnInit() is a good place for a component to fetch its initial data. ]

Always export the component class so you can import it elsewhere ... like in the AppModule.

Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several built-in pipes and you can create your own.

Ex: The word uppercase in the interpolation binding, right after the pipe operator ( | ), activates the built-in UppercasePipe. Thus, the data will be displayed in uppercase.

[(ngModel)] is Angular's two-way data binding syntax.

(Can bind from the property to the HTML textbox and the textbox back to the property.)

Although ngModel is a valid Angular directive, it isn't available by default.

It belongs to the optional FormsModule and you must opt-in to using it!

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires, aka metadata.

Some of the metadata is in the @Component decorators that you added to your component classes. Other critical metadata is in @NgModule decorators.

The most important @NgModule decorator annotates the top-level AppModule class.

The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project. This is where you opt-in to the FormsModule, with: import { FormsModule } from '@angular/forms'; and adding FormsModule to the imports array, like so: [imports](https://angular.io/api/core/NgModule#imports): [ [BrowserModule](https://angular.io/api/platform-browser/BrowserModule), [FormsModule](https://angular.io/api/forms/FormsModule) ],

(This actually requires refresh of browser to update.)

Every component must be declared in exactly one NgModule.

Note the Angular CLI declared HeroesComponent in the AppModule (src/app/app.module.ts) when it generated that component. (It imports it and adds to the @NgModule declarations array, just like with AppComponent.)

The \*ngFor is Angular's repeater directive. It repeats the host element for each element in a list.

(In other words, a for each loop.)

For example, <ul class="heroes"> <li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"> <span class= "badge"> {{hero.id}} </span> {{hero.name}} </li> </ul>

<li> is the host element, heroes is a list from the HeroesComponent class, and hero holds the current hero object for each iteration through the list.

You set the basic styles for the entire application in styles.css. You could add more styles to styles.css and keep growing that stylesheet as you add components.

You may prefer instead to define private styles for a specific component and keep everything a component needs— the code, the HTML, and the CSS —together in one place. This approach makes it easier to re-use the component somewhere else and deliver the component's intended appearance even if the global styles are different.

You define private styles either inline in the @Component.styles array or as stylesheet file(s) identified in the @Component.styleUrls array.

<li \*ngFor="let hero of heroes" (click)="onSelect(hero)"> This is an example of Angular's event binding syntax. The parentheses around click tell Angular to listen for the <li> element's click event. When the user clicks in the <li>, Angular executes the onSelect(hero) expression that you define.

When the app starts, the selectedHero is undefined by design. Binding expressions in the template that refer to properties of selectedHero — expressions like {{selectedHero.name}} — must fail because there is no selected hero. The component should only display the selected hero details if the selectedHero exists; Add Angular's \*ngIf directive to the <div> and set it to selectedHero.

The Angular class binding makes it easy to add and remove a CSS class conditionally. Just add [class.some-css-class]="some-condition" to the element you want to style.

The HeroDetailComponent template binds to the herodetail's component's hero property which is of type Hero, so we need to import it.

The hero property must be an Input property, annotated with the @Input() decorator,

so the external HeroesComponent will bind to it: <app-hero-detail [hero]= "selectedHero"> </app-hero-detail> (add this to heroes.component.html)

^ This is property binding, a one-way binding from the slectedHero property of the Heroes component to the hero property of the target element which maps to the hero property of the HeroDetail component.

Amend the @angular/core 's import statement to include Input: import { Component, OnInit, Input } from '@angular/core';

Then add the property to the class with the input decorator: @Input() hero: Hero;

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service. Removing data access from components means you can change your mind about the implementation anytime, without touching any components. They don't know how the service works.

Services are a great way to share information among classes that don't know each other.

Notice that the new service imports the Angular Injectable symbol and annotates the class with the @Injectable() decorator. This marks the class as one that participates in the dependency injection system. It can provide an injectable service and it can have its own dependencies.

You must make the HeroService available to the dependency injection system before Angular can inject it. You do this by registering a provider. A provider is something that can create or deliver a service.

Then you need to make sure that the HeroService is registered as the provider of this service. You are registering it with an injector, which is the object that is responsible for choosing and injecting the provider where it is required.

(By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @Injectable decorator; If you look at the @Injectable() statement right before the class definition, you can see that the providedIn metadata value is 'root'.)

When you provide the service at the root level, Angular creates a single, shared instance of the service and injects it into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

---

(In template) (In component)

Interpolation: {{name}} <- this.name

Property binding: < … [disabled="disabled"] > -> this.disabled

Event binding: <…(click)="handClick(event)" -> handleClick() {}

Two-way binding: <…[(ngModel)]="data"> <- -> this.Data

Requires FormsModule

Patch version release is for bug fix.

Semantic Versioning -

Uses given name (on left of assignment operator, :, in devDependencies array ) with version number at end.

"^*version*": Updates whenever a new major version is released.

"~*version*": Updates, but locks to minor versions. Want to manually edit when an accepted major version is out.

"*version*": Locks to the exact version given.

--

environment.ts file lets you set the development/production variables!

Make sure to import from correct environment. (Remember httpClient default import is wrong.)

--

Same origin policy: (For requests made specifically from browser.) By default, accepts same origin requests (same domain, subdomain, scheme and sometimes port).

A server (Angular) cannot send request through browser to another server (API).

You must enable Cross Origin RequestS (CORS) on the other server, for the first server:

In Startup add: readonly string *OriginPolicyName* = "\_myAllowSpecificOrigins";

In Startup's ConfigureServices, add: services.AddCors(options => {

options.AddPolicy(*OriginPolicyName*,

builder => {

builder.WithOrigins("*allowedUrl*",);

});

});

In Startup's Configure, add: app.UseCors(*OriginPolicyName*);

Create login component for authorization.

Give it a route. Have fields needed to login.

Login template has form for ngSubmit, with inputs for all the fields and a submit button. Make sure form has client-side validation (and server side, I would assume).

--

With canactivate, we will be prevented, according to auth guards.

--

For project Angular

More than one component. with more than just get. Use routing. Do some styling.

(Login is good.)

Need generated tests to pass. Don't need any extra tests.

--

Jasmine testing (has a sort of fluent API) is done with many javascript frameworks and even w/o.

Unit test methods, specs, are about one object, so we use describe (global Jasmine function with string name and function test composed of its) and it (global Jasmine function with string name and function code which contains expectations) and expect (a true or false assertion, given the actual value and a Matcher function with the expected value). One false expectation fails a spec.

spy is another name for a mock object.

default for watch is true.

--

npx can run command line stuff inside your modules.

Can grab and install stuff to temp directory

When we do build that is the Angular path…

Angular and our code is inside the files.

--

In TS/JS

Can always turn Observables into Promises using .toPromise();

Better when you only expect one response down the pipe.

If you have await, then you need to add async to function declaration. Outside function, don't need to worry about anything, just call it.

---

**Monday 4/1**

Angular is a framework for building single page applications using typescript.

Angular changes DOM, not changing to different webpages.

Document Object Model

Binding links the DOM with properties and methods.

Typescript classes are special due to decorators which make them Angular.

Angular Decorators –

@Component: Turn TS class into an Angular component.

@NgModule: Make TS class into an Angular module. Provide namespace and compilation syntax for components. (Knows about node)

@Injectable: Make TS class into services (view independent logic that supports dependency injection. Needs to be added to providers array).

@Directive: Custom directives possible.

Attach extra meaning to class or property. (Actual meaning is determined by framework.)

Component: Manages view (area on screen). Can be nested.

References a template (html + [binding + directives] <- angular stuff )

Angular Module: A class with a @NgModule decorator. Knows about ng stuff. Has imports, exports declaration.

Root module has default name appmodule but that can be changed.

TS Module:

A .ts file which has imports at the top and/or exports. Pulls file content out of JS global scope.

Import is similar to "using" statement for namespaces.

A directive is anything that can go into template that is not valid html.

NgStyle: ngClass:

Add or remove new elements from dom.

Structural: \*NgFor (for loop), \*NgIf

app-root (selector for root component.) app-*componentname*

Angular tutorial gives implementation and conceptual.

Fundamentals of angular guide …is conceptual?

**Tuesday 4/2**

Project 3 -

Microservices: (Further SOA.) More separation of concerns. Single responsibility principle applied to everything. Each has its own DB and API.

Container Orchestration:

Docker Swarm: Kubernetes was first and good, but docker wanted to compete for service of their containers. (Azure is becoming only Kubernetes and deprecating others.)

ngUI has selector module and forecast module, each with their own service.

2 published npms (used by one deployment app) and 2 service deployments which both connect to 3 user, room, batch, for 8 deployments.

Angular library (not an app). Modules published to npm so that others can include them, like our app. Uses np-packagr format.

Auth team: Conrad (Team Lead), Lee, Oswaldo, Andy, and Daniel.

Auth with Azure Ad, UI improvements/refactoring. Forecast doesn't actually connect to lower levels.

I'm on Queues team with: Storm (Team Lead), Matt K, Ben and Will.

Implemented Message queues with Azure Service Bus. Selection service (something is broken).

When new data gets published, add to queue that service can check later.

Last batch not plugged into ASP.NET lifecycle correctly. Improved logging (basic already exists). Want Serillog unless Nlog is better for some reason (good service bus already exists or something).

Upper services are PostgreSQL w/EF.

Back end is nonrelational DB, MongoDB, which is "document based".

Seed data is in every DB (not using persistent data across all deployments).

Use pivotal tracker.

Users shouldn't really be changing from above, but room will have changes pushed down.

Anything to make more RESTful.