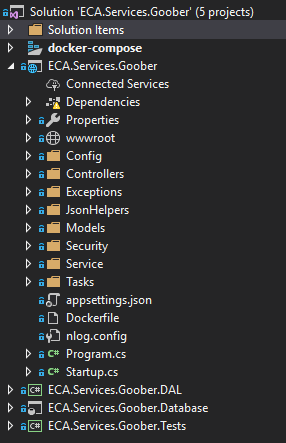
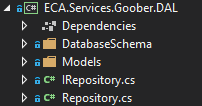
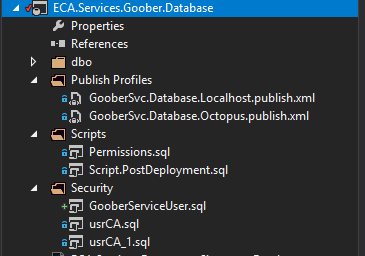
## Namespaces & Directory structure

1. Namespace – ECA.Services.{service-name} … service should be succinct. 2nd-tier services live here. These are not platform specific, more generialized, highest potential for re-use.
2. Namespace – ECA.Facilitator.Service.{service-name} – 1st tier services are platform specific



1. Directories
   1. Config – nlog.config contains service configurable items, the Config directory is where classes live that deal with configuration storage (read/write)
   2. Controllers – RESTApi controllers for routing of commands. No business logic here.
   3. Exceptions – all classes exceptions for the service live here. Use throw(), not status in design.
   4. JsonHelpers (optional) – we will likely have better classes in here for JSON schema
   5. Models – if we hand sew, POCO or we use tools, we create POCO classes for RESTApi Request/Response
   6. Security – classes for handling Authentication token; other classes/wrappers to API calls for permissions (later)
   7. Service – the “meat” of the service business logic; if we have unit tests, it would be around these classes. Feel free to use GOF (Gang of Four patterns), this is inner Microservice logic domain, where we do the work of the service.
   8. Tasks – (optional) thread management subsystem
2. Files
   1. Appsettings.json – config settings; database connection, constants, etc.
   2. Dockerfile – used to create the image
   3. Nlog.config – used (currently) for local Nlog table ([Log]) if setting up local DB; ToDo: see about a centralized Logger service, leverage Nlog and create provider that routes log info
   4. Program.cs - entry point to program psv Main()
   5. Startup.cs – class is instantiated; overloaded methods called inside to
      1. Configure Cors options
      2. Configure serialization for MVC
      3. Add a service - TaskManager ( can be commented out if not using threads)
      4. Configure auto documentation of API with Swagger
      5. Set up Core2 DI (Dependency Injection) for classes. Use DI, not new for classes wherever possible. Makes code easier to maintain, and refactor (move) around.
      6. Set up DB Context (optional) if using local db
3. Other Projects  
   Data Access Layer (DAL) and DB project can be omitted if service is not storing locally.  
   Services should first be written to “store no data”. After consideration, persistence should be looked at. Services should be reentrant, thread safe, and save no state. This allows them to “run down” (be turned off) and on without concerns of data (flushing caches, etc.).
   1. DAL – Data Access Layer (Simple, EF Core2 based)  
      
      1. IRepository – we will use this locally to define interactions with db
      2. Repository – concrete implementation of DB interaction (this uses EF for Core2)
      3. Models – DB models; may be necessarily different from REST models
   2. Database (SQL Server)  
      
      1. Database Project file (uses Visual Studio Database Project, sqlproj)
      2. Publish Profiles – allow for both release (thru Octopus) and Localhost (local db) creation
      3. Script.PostDeployment.sql – use for pick lists, and seed data and other post table creation sql commands
      4. Security – for user / logon information necessary for the service access to db via connection

## What is new and leveraged from ASP NET Core 2

1. Dependency injection (see startup.cs file)
2. Kestrel – is the server ( a very lightweight non-IIS server), read more here: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-2.1&tabs=aspnetcore2x>
3. IActionResult is a new return from API calls

## Swagger

1. We want to use swagger in all cases to “Document” the API, it is done with an attribute in front of the controller method:

[HttpGet("ping")] // ping

[SwaggerResponse((int)HttpStatusCode.OK, typeof(Response))]

public IActionResult GetPing()

{

## Exception handling

Code that is written to throw exceptions, is cleaner, more lightweight, easier to maintain, and exceptions can bubble (and stack from inner calls) all the way back to the client, depending on the service (if it wants to trap, or pass along the failure). Exception classes aren’t hard to write, and a developer testing cycle will determine where prudent and pragmatic use of an exception may be warranted in the code being written. Here, if we can’t read the config file, or a setting within the config file … sounds like a good idea to throw an exception:

namespace ECA.Services.Goober.Exceptions

{

public class ConfigFileReadError : Exception

{

public ConfigFileReadError()

{

}

public ConfigFileReadError(string message)

: base(message)

{

}

}

}

## Dependency Injection

Is build into ASP NET Core 2.x so we leverage that dependency injection. If you find yourself using the new operator, to instantiate a class, you should think “Can this be injected?”

Example:   
In startup file:

services.AddTransient<IJsonConfiguration, JsonConfiguration>();

In class file:

private IJsonConfiguration \_config;

public Recipe( IJsonConfiguration config) // ctor

{

\_config = config;

}

Injection happens during construction, when the constructor references the Interface , which has previously been registered with the concrete class in the .AddTransient() in the startup.cs

## Interface files

Interface files (because of dependency injection), possible FAKEs and Unit Tests, too.  
It is a good idea. Interface files are same as class name, start with “I” of course, and are in their own separate source file.

## Other minimul style guidelines

1. For class module global variables, use the underscore prefix, as in \_config above
2. For any static text or values, use ReadOnly Const global vars, without the underscore and in all caps, example private const int START\_LEVEL = 1;  
   Better: if a hard coded value is in code ~ should it be in the appsettings.json config file?
3. Normal C# casing applies
   1. Variables start lower case, exhibit camel-case
   2. Class names start with Upper case char