* Solution Overview
  + .NET Core Backend Stack
    - The **Dependencies** folder is a virtual folder that contains all internal, external, and third-party references
    - The **Controllers** folder contains all the MVC controllers for the project
    - The **Pages** folder than contains Razor Pages
    - **Program.cs**, **Startup.cs**, and **appsettings.json** are root-level files that control the project configuration (modules, middleware, compilation settings, etc.)
  + Angular Frontend Stack
    - The **wwwroot** folder contains the ready-to-publish app contents (CSS, HTML, JS) and fonts, images, and other static resources for our users
    - The **ClientApp** root folder contains the Angular and package config files and has several important sub-folders
    - **ClientApp/src** folder contains the Angular app source code files
    - **ClientApp/e2e** folder contains some sample E2E tests built with Protractor (or is would if it was present
* The .NET Core Backend
  + Unlike typical ASP.NET MVC templates, there is no Views folder; this is typically where our Razor pages would go
  + But, since this is a SPA, there’s only one page!
  + The single page here is actually contained in **ClientApp/src/index.html**; there are static server-side HTML is the Error.cshtml
  + Razor Pages
    - These were introduced in .NET Core 2.0 and are an alternative to ASP.NET Core’s standard MVC pattern (with dedicated views)
    - A Razor Page is similar to a Razor view with similar syntax, but they also contain the controller source code
    - This dependency is shown by the \*.cshtml files having nested \*.cshtml.cs files
  + Controllers
    - We still have a Controllers folder, despite having controller logic contained in the Razor pages
    - This is because not all controllers are meant to serve server-rendered HTML pages; some are meant for JSON, XML, static resources (JPG, JS, CSS, etc.), or simple HTTP responses (e.g., redirects)
    - This decoupling between the standard HTML content and the rest of the HTTP response (i.e. Service APIs)
  + Configuration Files
    - Program.cs
      * This file was added in ASP.NET Core 1.0 and its main purpose is to create a WebHostBuilder
      * This will construct the IWebHost instance that will host our web application
      * Web Host vs Web Server
        + Note that the host here is the execution context of the ASP.NET Core app
        + For ASP.NET Core this must implement the IWebHost interface to expose web features and services
        + The web host references the server that will handle the actual requests
        + The host is responsible for app lifecycle and startup, the server is responsible for serving accepting HTTP requests
        + Note that for ASP.NET 6, it looks like this is all taken care of in the Program.cs file
        + This version configures and runs a WebApplication instance
        + Note that there is a good amount of default config that happens under the hood here
    - Startup.cs
      * This file was first introduced with OWIN-based apps to replace tasks previously handled by e.g. Global.asax
        + Note that the Open Web Interface for .NET (OWIN) was part of project Katana
        + Project Katana was a set of tools that MS came out with in 2013 for building and hosting OWIN-based web apps
      * Note that the .NET Core startup is much leaner than OWIN, and focuses on adding/configuring Services and DI, and configuring the HTTP request pipeline via middleware
      * This middleware includes
        + Adding special exception handling for dev testing
        + Adding HTTPS redirection
        + Setting up using static files (under wwwroot folder) and SPA static files (using ClientApp/src/assets) for dev envs
        + Adding endpoint routing to direct HTTP requests to the correct controller
        + Setting the source app for our Angular application
        + Adding an in-memory Angular CLI server to handle all requests to the Angular app; this makes it easy for us as we don’t need to set this up manually each time
      * Note that the latest template includes all of this in the Program.cs file
      * The general flow goes something like
        + Request will hit the ASP.NET core backend
        + The backend will pass the request through the HTTP middleware you have registered
        + If none of the registrations (e.g. controller mappings) can handle the request, then it will be passed to the Angular app
        + The Angular app will then try to match the request using its client-side routing
    - appsettings.json
      * This replaced the old Web.config file and uses JSON in the place of verbose XML
      * The new model allows for getting settings from a variety of sources including JSON
      * The settings can then be accessed in your controllers via DI
        + E.g.  
          public TestController(IConfiguration config)  
          {  
           var value = config[“Logging:IncludeScopes”];
      * Alternatively, we could populate a strongly-typed POCO object with these settings
      * Keep in mind the environment override files (e.g. appsettings.Development.json)
* The Angular Front-End
  + Workspace
    - This is a place in your filesystem containing the Angular files (i.e., the app files, libraries, assets, etc.)
    - In most templates this is the **ClientApp** folder
    - The workspace here was created as part of the **dotnet new** command, it could also have been done via the Angular CLI
    - Note that all Angular CLI commands will be executed from the workspace folder
    - angular.json
      * This file is created by the CLI and is the workspace configuration file
      * It contains workspace-wide and project-specific configuration details for build and dev tools in the Angular CLI
      * **version**: the config file version
      * **newProjectRoot**: where new projects are created relative to the workspace root; note that our app only has one project which is fine for our case
      * **projects**: container that has sub-sections for each project in the workspace
      * **defaultProject**: the default project; any CLI command that doesn’t specify a project will use this one
      * This file follows a standard generic-to-specific cascading rule for configurations (workspace levels configs can be overridden by project configs can be overridden by CLI args)
    - package.json
      * This is the npm config file that contains the npm packages that will be restored prior to the project starting
      * npm is basically Nuget for JavaScript
      * It started as the default package manager for the JavaScript Node.js runtime environment
      * Now npm is the de facto package manager for pretty much all JS projects, libraries, and frameworks, including Angular
      * For VS you basically configure the package.json file and the packages will be downloaded and placed in the **node\_modules** folder
      * This file consists of a JSON object made up of key-value pairs where the keys are the packages and the values are the versions
      * Note you can use standard npmJS sytax to specify auto-update rules (e.g. ~ for latest patch, ^ for latest minor release+patch)
      * Upgrading/Downgrading Angular
        + Note that the Angular packages here are all locked to the major build number
        + This must mean reliability has increased, as previously these numbers were locked down to the patch
    - tsconfig.json
      * This is the TypeScript config file
      * We will use TS for this project because it includes features like static typing, interfaces, and classes
      * It is an MS product and so also has pretty good support in VS and allows us to build more robust code
      * This file configures VS and the TypeScript Compiler (TSC) to properly transpile the TS code in the ClientApp folder
      * Here we add a configuration for the Angular Ahead-of-Time (AoT) compiler to report syntax errors immediately
    - Other Workspace Level Files
      * **.editorconfig**: workspace-specific config for code editors
      * **.packagelock-json**: provides version info for all packages installed currently in the node\_modules folder by npm client
      * **.tslint.json**: default TSLint options for all projects in the workspace; these can be overwritten and/or integrated with project-specific tslint.json files in the project roots
  + The /ClientApp/src/ Folder
    - There are three folders in the src folder here
      * **app**: this is where our TS source code for our client-side Angular app goes
      * **assets**: this contains all the app’s images and other asset files that will be copied to the wwwroot folder as-is when the app is built
      * **environment**: contains build config options for specific environments
    - Other root-level files
      * **index.html**: the main HTML page that is served for your site; the CLI automatically adds all the JS and CSS files when building yoru app
      * **karma.conf.js**: app-specific config for Karma, a tool for running Jasmin-based tests
      * **main.ts**: the main entry point for your app; compiles the app via the JIT compiler and bootstraps the app’s root module (AppModule here) to run in the browser; the AOT compiler can be used here by adding --aot to the CLI build/serve commands
      * **polyfills.ts**: provides polyfill scripts for browser support
      * **styles.css**: a list of CSS files that supply styles for the project
      * **test.ts**: the main entry point for the project’s tests
      * **tsconfig.\*.json**: these are project specific configs for aspects of the app (app-level, server-level, test-level) that will override the options in the root-level tsconfig.json
    - The /app/ Folder
      * This folder contains all the Modules, Services, and Components for our Angular app; also includes templates and styles
      * AppModule
        + NgModules are the basic building blocks of Angular applications
        + These collect related code into functional sets
        + Angular apps require a root module (typically called the AppModule) that tells Angular how to assemble the app
        + This enables bootstrapping and application lifecycle management
        + The root module also contains a list of all available Components
        + The remaining modules are feature modules
        + The main.ts file bootstraps the AppModule, which loads the AppComponent, which loads all other components

main.ts > app.module.ts > app.component.ts

* + - * + The root module for this sample App is in /app/app.module.ts
        + This file contains a bunch of import statements and arrays referencing Components (basically just a reference file)
      * Server-Side AppModule for SSR
        + The additional app.server.module.ts file enables the use of Angular Universal Server-Side Rendering (SSR)
        + SSR renders Angular apps on the server (if the backend supports it)
        + Note that .NET Core does natively support SSR
        + With SSR, the client side initializes via main.ts > app.module.ts > app.component.ts, and server side via main.server.ts > app.server.module.ts > app.component.ts
      * AppComponent
        + Components are the pieces that put the Angular app together
        + An Angular app is basically a tree of components
        + Components define Views, sets of screen elements Angular chooses from and modifies based on app logic
        + Components use Services, which provide specific functionality not related to a certain view
        + Service Providers can be injected into Components to make code modular, reusable, and more efficient
        + The AppComponent is the cornerstone of all components
        + Per Angular convention, only AppComponents should be at the root level; all other components should be in subfolders of the root
        + These subfolders act as namespaces for those additional components
        + There are two files in the AppComponents by default

**app.component.ts**: defines Component logic; the source code for the Component class

**app.component.html**: defines the Component HTML template; this HTML can exist in the source code, but it is good practice to break this out

* + - * + There are other files that components usually have that AppComponent does not since it is lightweight

**app.component.css**: the base CSS stylesheet for a component; can also be in source code but is better separated

**app.component.spec.ts**: defines unit tests for the component

* + - * + Other Components

There are four other components in this basic template

CounterComponent

FetchDataComponent

HomeComponent

NavMenuComponent

* + Testing the App
    - HomeComponent
      * The HomeComponent is the home view for our application
      * Note that a View is basically the combined HTML template generated by the Component (including Sub-Components)
      * A View corresponds to a given navigation route
    - NavMenuComponent
      * This is a sub-component in all other components in our app
      * This component doesn’t have a route for itself
      * In this app, this is where we implement all first-level navigation routes
        + These are routes that we want users to be able to reach in a single click (w/o navigating through other components)
        + This includes home, counter, and fetch-data for this sample app
    - First App Test
      * We do this by running **npm test** in our ClientApp (this runs the test script in the package.json)
* Getting to Work
  + Static File Caching
    - Caching static files can obviously be helpful for production environments, but can be annoying in production
    - The AngularCliServer in the SPA middleware will fix this for TS files and static assets provided by Angular
    - We want this same behavior on the back-end server
    - To do this we are going to set options in the UseStaticFiles() middleware addition in the pipeline
    - To do this we set various headers in the response for the static file caching (Cache-Control, Pragma, Expires)
    - Note that we added these setting to the appsettings so that we have no cache for dev/debug but otherwise have a cache
  + Client App Cleanup
    - So we delete the counter component and the fetch-data component since we will not be using them going forward
    - The AppModule Source Code
      * Angular modules (known as NgModules) were introduced in Angular 2 RC5
      * They help devs consolidate sets of Components, directives, and pipes into reusable blocks
      * Every Angular app since this release must have at least one module (the root module)
      * By convention, the root module is typically named AppModule and is split into two code blocks
        + A list of import statements for all references required by the application
        + The root NgModule block; this is basically an array of named arrays, with each containing Angular objects serving a common purpose (e.g., directives, components, pipes, modules, providers)
        + The last one contains the list of component that we want to bootstrap (in most cases this is AppComponent)
    - Updating the NavMenu